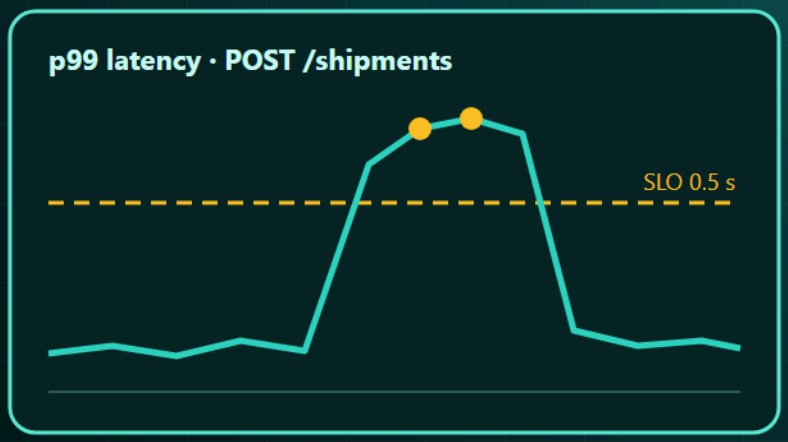





```
histogram_quantile(
  0.99,
  sum by (le) (
    rate(http_server_
    request_duration_
    seconds_bucket[5m])
  ))
```

```
{"level":"info","event":"shipment.created","trace_id":"7d3e9a1c…"}
```

A COMPLETE, PRACTICAL GUIDE

Observability in Practice

Logs, metrics, and traces with OpenTelemetry, Prometheus, and Grafana

Structured logging · OpenTelemetry · Collector · PromQL
Grafana · Loki · Tempo · Jaeger · SLOs · Burn-rate alerts

INSTRUMENT · QUERY · ALERT · RESPOND

THENEXBOOKS

Observability in Practice

Logs, Metrics, and Traces with OpenTelemetry, Prometheus, and Grafana
From your first structured log line to SLOs, burn-rate alerts, and calm incident response

First Edition

THENEXBOOKS

Observability in Practice
Logs, Metrics, and Traces with OpenTelemetry, Prometheus, and Grafana

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples, configuration files, and queries may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. OpenTelemetry, Prometheus, Grafana, Loki, Tempo, Jaeger, Kubernetes, and the other tools discussed evolve quickly; component stability, configuration keys, defaults, and command-line flags can change between releases. Always consult the official documentation for the versions you run, and test configuration changes in a non-production environment first.

OpenTelemetry, Prometheus, Jaeger, and Kubernetes are trademarks or registered trademarks of The Linux Foundation. Grafana, Loki, Tempo, Mimir, Pyroscope, and Alloy are trademarks of Grafana Labs. Docker is a trademark of Docker, Inc. Python is a trademark of the Python Software Foundation. Node.js is a trademark of the OpenJS Foundation. Java is a trademark of Oracle and/or its affiliates. Linux is a trademark of Linus Torvalds. PostgreSQL, Redis, Apache Kafka, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by any of these organizations.

Ternway, the parcel-delivery platform used as the running example in this book, is fictional, as are its services, merchants, couriers, staff, traffic figures, incidents, and all sample data. Domain names use the reserved `.example` top-level domain.

Preface

Every on-call engineer knows the feeling. A message arrives: "Shipments are slow." The dashboards are green. CPU is fine, memory is fine, every health check passes. Somewhere between five services, a database, a cache, and a message broker, something is wrong — and the tools you have can only answer the questions someone thought to ask months ago.

Observability is the ability to answer new questions about a running system from the telemetry it already emits: which requests are slow, for whom, since when, and why. It is not a product you buy or a single dashboard you build. It is a practice — structured logs that carry context, metrics that are cheap to aggregate, traces that follow a request across every hop, and the discipline to connect them and turn them into decisions about reliability.

This book teaches that practice with the open, vendor-neutral tools that most teams now reach for: **OpenTelemetry** to produce telemetry, **Prometheus** to store and query metrics and drive alerts, **Grafana** to visualize, **Loki** for logs, and **Tempo** or **Jaeger** for traces. We do not stop at installing tools. We instrument a realistic system end to end, define service level objectives with exact numbers, build burn-rate alerts that page only when users are hurting, and walk through real-feeling incidents minute by minute.

Who This Book Is For

- **Backend developers** who want their services to explain themselves — and who are tired of adding a log line, redeploying, and waiting for the problem to happen again.
- **DevOps and platform engineers** who run the telemetry pipeline, the Collector, Prometheus, and Grafana, and must keep them reliable and affordable.
- **SREs and on-call engineers** who need SLOs, error budgets, and alerts that are urgent, actionable, and rare.
- **Team leads and architects** who must choose conventions and tools that a whole organization can live with.

You should be comfortable reading Python and JavaScript, know HTTP and the basics of Docker and Linux, and have some idea of how services are deployed. No prior experience with OpenTelemetry, Prometheus, or Grafana is assumed.

What You Will Learn

- What each signal — logs, metrics, traces, and profiles — is good at, what it costs, and how the signals link together.
- How to write structured logs with consistent fields, correlation IDs, and no secrets.
- How distributed tracing works: spans, context propagation with W3C Trace Context, baggage, and sampling.

- How to use OpenTelemetry: the API and SDK, OTLP, semantic conventions, resources, and the stability status of each signal per language.
- How to instrument Python and Node.js services, with zero-code and manual instrumentation, plus how Java and Go fit in.
- How to run the OpenTelemetry Collector: pipelines, processors, tail sampling, redaction, and connectors.
- How Prometheus stores metrics, how to write PromQL from simple rates to histogram percentiles and joins, and how to use exporters, recording rules, alerting rules, and Alertmanager.
- How to build Grafana dashboards that answer questions, query logs with LogQL, search traces with Tempo and Jaeger, and jump between signals with exemplars and trace-to-logs links.
- How to define SLIs and SLOs, compute error budgets, and alert on burn rates with multiwindow rules.
- How to run on-call and incident response with evidence, and how to write blameless postmortems.
- How to observe Kubernetes, control cardinality and cost, and operate the observability stack itself.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains why observability matters, the signals, structured logging, and tracing concepts, and introduces the Ternway lab. **Part II — OpenTelemetry** (Chapters 6–10) covers the OpenTelemetry architecture and semantic conventions, instruments the Python and Node.js services (with Java and Go briefly), and builds the Collector pipeline. **Part III — Metrics and Alerting with Prometheus** (Chapters 11–15) covers the Prometheus data model, PromQL, exporters, rules, and Alertmanager. **Part IV — Dashboards, Logs, and Traces** (Chapters 16–20) covers Grafana, Loki, Tempo and Jaeger, cross-signal correlation, and an overview of profiling and eBPF. **Part V — Reliability** (Chapters 21–24) covers SLIs, SLOs, error budgets, burn-rate alerting, symptom-based paging, on-call, and incident response. **Part VI — Production at Scale** (Chapters 25–28) covers Kubernetes, cardinality and cost, operating the stack, and a complete case study. The appendices provide a cheat sheet, templates and reference configurations, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Ternway

Throughout the book we instrument **Ternway**, a fictional parcel-delivery platform that serves online merchants from three depots with same-day, next-day, and standard delivery. Ternway runs five microservices: a Node.js **gateway** at the API edge, a Python **orders** service that quotes and creates shipments, a Python **dispatch** service that assigns couriers, a Node.js **tracking** service backed by Redis and PostgreSQL, and a Java **notifications** service that consumes events from Kafka and sends text messages and emails. We follow one golden shipment — `shp_7Q2K9M`, tracking number

`TW4821000173` — through every service, define five SLOs with exact targets, and investigate a series of incidents, from a slow database query to a cardinality explosion. Every number in the examples is consistent across chapters, so you can check the arithmetic yourself.

Versions and Conventions

Examples target recent versions of the OpenTelemetry SDKs and Collector, **Prometheus 3.x**, recent Grafana releases, **Loki 3.x**, **Tempo 2.x**, and **Jaeger v2**, with Python 3.12 and a current Node.js LTS release. OpenTelemetry moves quickly and its components mature at different speeds: some signals are stable in one language and still experimental in another. Where a feature, convention, or component is not yet stable, a Note says so — and you should check the project's current status pages before relying on it. In the lab's compose file, image versions come from an `.env` file; pin them to the current releases when you build your own.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version and stability differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause outages, data loss, runaway costs, or security problems.

Code, configuration, and queries appear in `monospace`; long lines are wrapped to fit the page. Shell commands start with `$`. File captions such as `otel/collector.yaml` show where each snippet lives in the Ternway repository. Metric names are shown as they appear in Prometheus, for example `http_server_request_duration_seconds`, and all times are UTC.

Contents

Preface

PART I — FOUNDATIONS

1 Why Observability: Beyond Dashboards and Health Checks

2 The Signals: Logs, Metrics, Traces, and Profiles

3 Structured Logging Done Right

4 Distributed Tracing Concepts: Spans, Context, and Propagation

5 Meet Ternway: Architecture, the Lab Stack, and Conventions

PART II — OPENTELEMETRY

6 OpenTelemetry Architecture: API, SDK, OTLP, and Stability

7 Semantic Conventions and Resources: Naming Telemetry Consistently

8 Instrumenting Python Services: Orders and Dispatch

9 Instrumenting Node.js Services, with Go and Java Briefly

10 The OpenTelemetry Collector: Pipelines, Processors, and Sampling

PART III — METRICS AND ALERTING WITH PROMETHEUS

11 Prometheus Fundamentals: Data Model, Scraping, and OTLP Ingestion

12 PromQL from First Principles: Selectors, Rates, and Aggregation

13 PromQL in Practice: Histograms, Percentiles, and Joins

14 Exporters and Infrastructure Metrics

15 Recording Rules, Alerting Rules, and Alertmanager

PART IV — DASHBOARDS, LOGS, AND TRACES

16 Grafana Dashboards That Answer Questions

17 Logs with Loki and LogQL

18 Trace Backends: Tempo and Jaeger

19 Connecting the Signals: Exemplars, Trace-to-Logs, and Correlation

20 Continuous Profiling and eBPF: An Overview

PART V — RELIABILITY

21 SLIs and SLOs: Measuring What Users Feel

22 Error Budgets and Burn-Rate Alerting

23 Alerting on Symptoms: Pages That Matter

24 On-Call and Incident Response with Observability

PART VI — PRODUCTION AT SCALE

25 Kubernetes Observability

26 Cardinality, Cost, and Retention Control

27 Operating the Observability Stack: Scaling, Reliability, and Security

28 Case Study: Ternway Peak Week, End to End

APPENDICES

A Observability Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

Why observability matters for distributed systems, what each signal is good at, how to write logs that machines and people can both use, how distributed tracing ties requests together across services, and a tour of Ternway, the parcel-delivery platform we instrument throughout the book.

1 · Why Observability

2 · The Signals

3 · Structured Logging Done Right

4 · Distributed Tracing Concepts

5 · Meet Ternway

CHAPTER 1 · PART I — FOUNDATIONS

Why Observability: Beyond Dashboards and Health Checks

Green health checks and calm CPU graphs can sit right next to angry customers. In this chapter we look at why that happens, what observability adds on top of classic monitoring, and how the rest of this book turns a fictional parcel-delivery platform into a system you can question freely while it is running.

A Merchant Says Shipments Are Slow

Let us start with a story rather than a definition. Ternway is a fictional parcel-delivery platform. Online merchants call its public API to get a price quote, create a shipment, and track parcels as couriers carry them from one of three depots (`east`, `west`, and `central`) to the recipient. Merchants choose between three service levels: `same_day`, `next_day`, and `standard`. Behind the API sit five small services, a PostgreSQL database, a Redis cache, and a Kafka topic. It is a perfectly ordinary modern backend, which is exactly why it makes a good teaching example.

Imagine a Tuesday afternoon. Sam Okoro, Ternway's support lead, receives a message from Basalt Outfitters, an outdoor-gear merchant (merchant ID `m_1002`). Their warehouse team says that creating shipments "feels slow today", and their packing station is backing up because each label takes several seconds to appear. Sam forwards the message to the engineering channel and asks a simple question: *is something wrong?*

Now imagine that Ternway's engineers have only two tools to answer that question. The first is a set of host dashboards: CPU, memory, disk, and network for every machine. The second is a set of health checks. Every Ternway service exposes `GET /healthz` (liveness) and `GET /readyz` (readiness), and an uptime checker calls them every thirty seconds. That is a common starting point, and it is far better than nothing. Let us see how far it gets us.

What the classic tools can tell us

Arjun Mehta, the backend engineer who owns the gateway, opens the host dashboards. The gateway machines look normal. The orders machines look normal. The database host shows CPU a little higher than usual, but it has been higher on other afternoons too, and nothing is near a limit. Memory is flat everywhere. The uptime checker reports that all five services have answered their health checks successfully for the whole day.

So Arjun can say, with confidence, a few things that are true and not very useful:

- No service is down. Every process answers its health endpoint.

- No machine is out of CPU, memory, or disk.
- The database host is a bit busier than usual, which may or may not mean anything.

What the classic tools cannot tell us

The questions Sam and the merchant actually care about are different, and Arjun has no way to answer them:

- Is shipment creation really slower right now, or does it just feel slow to one warehouse?
- If it is slower, is it slower for every merchant, or only for Basalt Outfitters?
- Is it slower for every depot, or only for `west`, where Basalt Outfitters ships from?
- Which of the five services is adding the time? Is it the gateway, orders, dispatch, or the database?
- When did it start, and did anything change at that moment, such as a deployment?
- How many requests have been affected, and is this an emergency or a minor annoyance?

None of these answers lives in a CPU graph, and none of them lives in a health check. A health check asks a process "are you alive?" and the process says "yes". It does not ask "are your users getting what they came for?" The CPU graph shows how hard a machine is working, not whether the work produces the right result fast enough.

This gap is not hypothetical. Later in the book we walk through incident INC-2041 in detail. On 2026-03-10, a new version of the dispatch service (2.4.0) shipped a courier query that could not use its database index. The database began scanning the whole `couriers` table for every assignment. About 62% of shipment-creation requests started taking longer than half a second, yet not a single request failed and not a single health check turned red. Everything was "up". Everything was slow.

> **NOTE**
> Health checks are still valuable. Orchestrators such as Kubernetes need liveness and readiness signals to restart stuck processes and to stop sending traffic to instances that are not ready. The problem is not that health checks exist; it is treating them as the answer to "are users happy?" They answer a much narrower question.

Here is a Ternway health endpoint, so you can see exactly how little it checks. This is the liveness handler in the orders service, written with FastAPI:

services/orders/app/main.py (excerpt)

```
from fastapi import FastAPI

app = FastAPI(title="orders")


@app.get("/healthz")
```

```
async def healthz() -> dict[str, str]:
    # Liveness: the process is running and the event loop answers.
    return {"status": "ok"}
```

During INC-2041, this handler kept returning `{"status": "ok"}` in a millisecond or two. It was telling the truth: the orders process was alive. A readiness check that also pings the database would have stayed green too, because the database was answering, just slowly for one particular query in a different service. A health check can only detect failures that its author predicted, and nobody predicted "a new version of dispatch will stop using an index".

Monitoring and Observability

The words "monitoring" and "observability" are often used as if they meant the same thing, and vendors have stretched both. It is worth taking a minute to separate them, because the distinction shapes everything we build later.

Monitoring answers questions you asked in advance

Monitoring is the practice of collecting predefined measurements and checking them against predefined expectations. You decide ahead of time that disk usage above 90% is bad, that a service returning errors is bad, that a health check timing out is bad. You build a graph or an alert for each of those conditions. When one of them trips, you get notified.

Monitoring is excellent at **known unknowns**: failure modes you can name in advance but cannot predict the timing of. You know a disk can fill up; you just do not know when. You know a certificate expires; you can even compute the date. Monitoring handles these well and cheaply, and every system should have it.

Observability lets you ask questions you did not plan for

The term **observability** comes from control theory, where a system is called observable if you can infer its internal state from its external outputs. In software, the practical meaning is close to that: a system is observable when you can understand what is happening inside it, including situations you have never seen before, by examining the telemetry it already emits, *without shipping new code to ask a new question*.

That last part is the test we use throughout this book. If, during an incident, the only way to answer "is this slow only for the `west` depot?" is to add a new log line, open a pull request, wait for review, deploy, and wait for fresh data, then the system is not very observable for that question. If you can type a query and get the answer in a few seconds, it is.

Observability is aimed at **unknown unknowns**: failure modes nobody anticipated, emerging from combinations of conditions. A new query plan in one service, combined with the busy hour of one depot, combined with a retry policy in another service, produces a slowdown that no dashboard was

designed to show. You cannot write an alert for every such combination. What you can do is make sure the system emits rich enough, well-connected data that you can slice it along whatever dimension turns out to matter.

A comparison you can keep in mind

Aspect	Classic monitoring	Observability
Main question	Is a known thing broken?	What is happening, and why?
Failure modes	Known unknowns	Unknown unknowns as well
Typical data	Host metrics, health checks, thresholds	Metrics, structured logs, traces, profiles, linked by shared IDs
New question needs	Often new code or a new check	Usually just a new query
Point of view	Machines and processes	Requests, users, and business outcomes

> **TIP**
>
> A quick way to judge how observable a system is: pick a recent support complaint and try to answer "how many requests were affected, for whom, since when, and in which component?" using only existing telemetry. Time yourself. If the honest answer is "we would need to deploy something first", you have found your starting point.

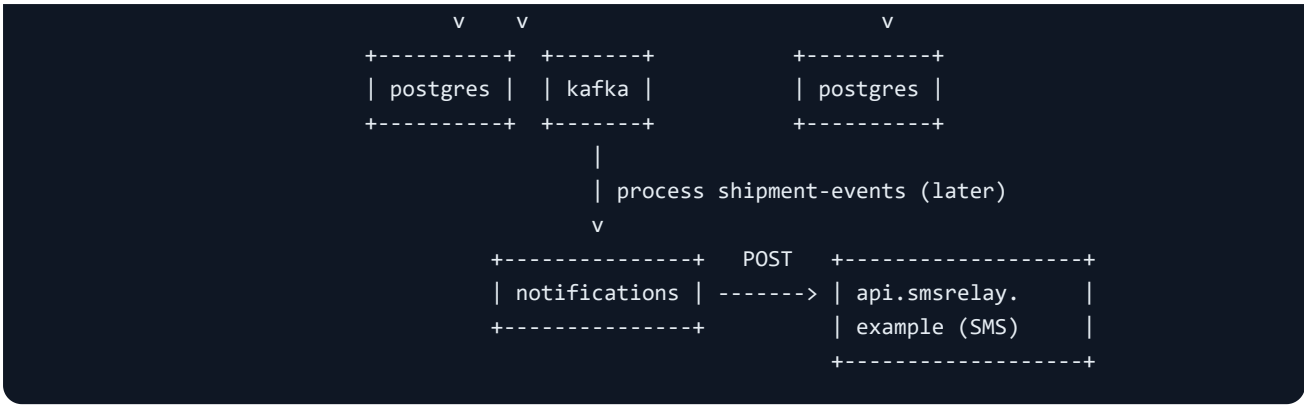
Why Microservices Make Debugging Harder

In a single-process application, a slow request has one obvious place to look. You can attach a profiler or a debugger, read one log file, and follow one stack trace from the top of the request to the bottom. Many teams built their debugging habits in that world, and those habits stop working once a request crosses process boundaries.

Ternway's shipment creation is a good illustration. When Basalt Outfitters creates a shipment, the request travels through several services and stores, some of them synchronously and some of them later:

Shipment creation path (simplified)





Several things make this harder to debug than a monolith:

- **Time is spread across processes.** The merchant experiences one number: how long the gateway took to answer. That number is the sum of work in the gateway, the network hop to orders, the work in orders, the database insert, the call to dispatch, dispatch's own database query, and the return trip. A slow request could be slow in any of those places, and the gateway's own CPU graph says nothing about which.
- **Logs are scattered.** Each service writes its own logs, on its own machines or containers, with its own timestamps. Without a shared identifier, matching the gateway's log line for a request to the dispatch log line for the same request means guessing by time and hoping clocks agree.
- **Some work is asynchronous.** Orders publishes a message to the `shipment-events` Kafka topic, and notifications picks it up later to send an SMS or email. If notifications falls behind, nothing fails at the API. The merchant's shipment was created successfully; the recipient just hears about it late. Later in the book, incident INC-2063 is exactly this: notifications delayed for 45 minutes with no HTTP errors anywhere.
- **Failures are partial.** In a monolith, the application is usually either working or not. In a distributed system, one depot, one service level, one database query, or one replica can misbehave while everything else is fine. Averages hide partial failures very effectively.
- **Many versions run at once.** Five services deploy independently. At any moment, the "system" is a particular combination of versions, and a rollout of one service changes behavior for requests that pass through four others. Knowing which version handled a given request becomes essential.

None of these problems is solved by adding more host graphs. They are solved by telemetry that follows the *request*: data that records what happened to each unit of work as it crossed each boundary, tagged with enough context to slice it later, and linked together by shared identifiers. That is the core of what we build in this book.

Meet Ternway, Briefly

Chapter 5 introduces Ternway's architecture, repository, and lab environment in full detail. For now, a short tour is enough to follow the examples in the next few chapters.

Service	Language	What it does
`gateway`	Node.js, Express	Public API edge: authentication and routing to the backends
`orders`	Python, FastAPI	Calculates quotes, creates shipments, publishes events
`dispatch`	Python, FastAPI	Assigns an available courier to a new shipment
`tracking`	Node.js, Express	Tracking lookups (Redis cache plus PostgreSQL) and courier location pings
`notifications`	Java, Spring Boot	Consumes shipment events and sends SMS or email

The golden shipment

Throughout the book we follow one particular request, which we call the **golden shipment**. On 2026-03-10 at 09:14:05 UTC, Basalt Outfitters created shipment `shp_7Q2K9M`: a 2,400-gram parcel going 18.5 km from the `west` depot, with `next_day` delivery. Orders priced it at 952 cents ($9.52), gave it tracking number `TW4821000173`, and dispatch assigned courier `cr_0418`. The whole request took about 72 milliseconds at the gateway.

That single request touched the gateway, orders, the database, dispatch, the database again, and Kafka, and it later triggered work in notifications. By the end of Part II, you will be able to find this request by its shipment ID in the logs, open its trace, see each of those steps as a span with its own duration, and jump from the trace to the log lines every service wrote while handling it. The golden shipment is our reference for what "normal" looks like, which is exactly what you need when something stops being normal.

The Same Afternoon, With Observability

Let us replay INC-2041 in the version of Ternway we build in this book. We will not explain every tool here; that is what the rest of the book is for. The goal is to see the shape of the difference.

1. **14:02 UTC.** Dispatch 2.4.0 finishes rolling out. Within a minute, latency for shipment creation begins to rise.
2. **14:19.** Nobody has to wait for a merchant to complain. An alert called `ShipmentsCreateLatencyBurnFast` pages Arjun, who is on call. It is based on a *service level objective* (SLO): Ternway promises that 99% of shipment-creation requests complete within 0.5 seconds, measured over a rolling 28 days. The alert fires because the rate of slow requests would exhaust that promise's budget far too quickly.
3. **14:21.** Arjun acknowledges the page. Ines Duarte, the SRE lead, takes the role of Incident Commander.
4. **14:27.** On the latency dashboard, the p99 line is far above normal. Next to it are small dots called *exemplars*, each pointing to a specific trace that contributed to the measurement. Arjun clicks one. The trace shows the request's journey; almost all of its 2.4-plus seconds sit in one span: dispatch's

`SELECT couriers` query, at 2,410 ms.

5. **14:33.** With the culprit narrowed to one query in one service, Kofi Mensah, who owns dispatch, reads the diff between dispatch 2.3.2 and 2.4.0 and spots the changed `WHERE` clause.
6. **14:36 to 14:41.** The team decides to roll back and completes the rollback to 2.3.2.
7. **14:44.** Latency is back to normal. The incident is resolved at 14:49.

Compare the questions from the beginning of the chapter with what the team could now answer. *Is shipment creation slower?* Yes, and the SLO alert said so before any customer did. *Which service?* Dispatch, visible directly in the trace. *Which step?* One database query. *When did it start, and what changed?* Right after the 2.4.0 rollout. *How bad is it?* That one we can compute exactly.

Putting a number on "how bad"

The impact window ran from 14:03 to 14:44, which is 41 minutes. Shipment creation averages 4 requests per second, and about 62% of requests during the window were slower than 0.5 seconds. Over 28 days, Ternway receives 9,676,800 shipment-creation requests, and a 99% target allows 1% of them, 96,768 requests, to be slow. A few lines of Python turn those facts into an impact statement:

impact.py (scratch file)

```
REQS_PER_SECOND = 4.0
impact_minutes = 41
slow_fraction = 0.62
slo_target = 0.99
window_requests = 9_676_800

requests = REQS_PER_SECOND * 60 * impact_minutes
slow = round(requests * slow_fraction)
budget = window_requests * (1 - slo_target)

print(f"requests in impact window: {requests:,.0f}")
print(f"slow requests:             {slow:,}")
print(f"28-day error budget:       {budget:,.0f}")
print(f"budget consumed:           {slow / budget:.1%}")
print(f"burn rate:                 {slow_fraction / (1 - slo_target):.0f}x")
```

Terminal

```
$ python3 impact.py
requests in impact window: 9,840
slow requests:             6,101
28-day error budget:       96,768
budget consumed:           6.3%
burn rate:                 62x
```

So the incident consumed about 6.3% of the latency SLO's 28-day error budget. That is a useful sentence for a status update, a postmortem, and a conversation with Basalt Outfitters. It replaces "it was slow for a while" with a measured statement. The "burn rate" of 62 says the team was spending its

error budget 62 times faster than the sustainable pace. Chapters 21 and 22 explain where these concepts come from and how to turn them into alerts that page for real problems and stay quiet otherwise.

> **WARNING**
>
> None of the fast diagnosis above happens by accident. The exemplar existed because the services recorded trace IDs alongside latency measurements. The trace existed because every service propagated trace context to the next. The SLO alert existed because someone defined the SLO and wrote the rule. If you install a tracing backend but skip context propagation, or collect metrics without consistent service names, the dots will not connect when you need them most. Observability is a property you design in, not a product you switch on.

The Observability Loop: Telemetry, Question, Action

It helps to think of observability as a loop rather than a collection of tools. Every investigation, from a quick "is the deploy healthy?" check to a two-hour incident, runs through the same cycle:

The observability loop



Each step has its own failure mode, and recognizing which step is broken tells you what to improve:

- **Telemetry is missing.** The data you need was never emitted. For example, latency is recorded, but without the depot attribute, so you cannot tell whether `west` is special. The fix is better instrumentation, covered in Part II.
- **The question is hard to ask.** The data exists, but asking the question takes an expert twenty minutes of query writing. The fix is good dashboards, recording rules, and shared queries (Parts III and IV).
- **The answer is ambiguous.** Two signals disagree, or a metric says "slow" but you cannot find which requests were slow. The fix is correlation: shared IDs and links between signals (Chapter 19).
- **The action is unclear.** You know what is wrong but not what to do. The fix is runbooks, SLOs that tell you how urgent it is, and practiced incident roles (Part V).

What Good Telemetry Looks Like

Chapter 2 examines each signal in depth. Before that, here are the properties that make telemetry useful for asking new questions. Use them as a checklist when you review any instrumentation, including your own.

It carries context

A log line that says "assignment failed" is a start. A log line that also says which shipment, which depot, which service level, which version of dispatch, and which trace it belongs to lets you ask "do failures cluster in one depot?" or "did this start with the new version?" without any code change. Here is what Ternway's structured log for the golden shipment looks like, pretty-printed for the page:

orders log line (pretty-printed)

```
{
  "timestamp": "2026-03-10T09:14:05.266Z",
  "level": "info",
  "service": "orders",
  "version": "3.2.1",
  "env": "production",
  "event": "shipment.created",
  "message": "shipment created",
  "trace_id": "7d3e9a1c4b2f48e6a0c5d81f2b6e9c47",
  "span_id": "c4e1f09a7b3d2e58",
  "shipment_id": "shp_7Q2K9M",
  "tracking_number": "TW4821000173",
  "merchant_id": "m_1002",
  "depot": "west",
  "service_level": "next_day",
  "price_cents": 952,
  "duration_ms": 61
}
```

Every field earns its place. `service` and `version` say who wrote the line. `event` is a stable, searchable name. The business fields say what the line is about. And `trace_id` and `span_id` link the line to the exact trace and span that produced it. Chapter 3 covers this format field by field.

It uses consistent names

If the gateway calls the service "orders", the dashboard calls it "order-service", and the logs call it "orders-api", every cross-signal query becomes a translation exercise. Ternway uses one name per service everywhere: in traces, metrics, logs, dashboards, and alerts. The OpenTelemetry semantic conventions extend the same idea to attribute names such as the HTTP method, route, and status code. Chapter 7 is entirely about naming.

It is affordable

Telemetry has a cost: CPU and memory in the application, network bandwidth, storage, and query time in the backend. Good telemetry is designed with that in mind. For example, Ternway never puts a shipment ID or a tracking number on a metric label, because each unique value would create a new time series. In incident INC-2088 a release of the tracking service did exactly that and the number of series in Prometheus rose from about 182,000 to about 1,140,000 in six hours. Chapter 26 turns that story into guardrails.

What This Book Builds

The book is organized as a single, cumulative project. We start with an uninstrumented Ternway and finish with a system you could put in front of a demanding on-call rotation. Here is the route.

Part	Chapters	What you get
I — Foundations	1–5	Concepts, structured logs, tracing fundamentals, and the Ternway lab stack
II — OpenTelemetry	6–10	All five services instrumented; the Collector routing telemetry to backends
III — Metrics and Alerting	11–15	Prometheus, PromQL, exporters, recording rules, and Alertmanager
IV — Dashboards, Logs, Traces	16–20	Grafana dashboards, Loki, Tempo or Jaeger, correlation, profiling
V — Reliability	21–24	SLOs, error budgets, burn-rate alerts, runbooks, incident response
VI — Production at Scale	25–28	Kubernetes, cost control, operating the stack, and a peak-week case study

The toolset is open source and vendor-neutral. Applications emit telemetry through **OpenTelemetry**, the Cloud Native Computing Foundation (CNCF) project that standardizes how software produces traces, metrics, and logs. The **OpenTelemetry Collector** receives that telemetry

and routes it. **Prometheus** stores metrics and evaluates alert rules, **Alertmanager** routes alerts to people, **Loki** stores logs, **Tempo** or **Jaeger** stores traces, and **Grafana** brings everything together in dashboards and an exploration interface.

We picked these because they are widely used, they interoperate through open protocols, and you can run all of them on a laptop. We do not claim they are the best choice for every organization. Because applications talk to the backends only through OpenTelemetry and the Collector, you can swap any backend for an alternative, such as Mimir, Thanos, or VictoriaMetrics for long-term metrics, Elasticsearch or OpenSearch for logs, a ClickHouse-based store, or a commercial platform, without touching application code. That portability is one of the main reasons to start with OpenTelemetry.

> **NOTE**
>
> Software in this space moves quickly. The book targets recent versions: Prometheus 3.x, Loki 3.x, Tempo 2.x, Jaeger v2, recent Grafana releases, and recent OpenTelemetry SDKs and Collector releases. Wherever a feature is new, experimental, or differs between versions, we say so. When you build the lab, pin every image to the current release in an `.env` file, and check each project's documentation for anything that has changed since this was written.

How each chapter works

Every chapter is hands-on. We show the configuration, the code, the query, and realistic output from the Ternway lab, and then explain what to look for. Numbers in sample outputs are consistent across the book: the golden shipment always costs 952 cents, shipment creation always averages 4 requests per second, and the incidents always unfold at the same minutes. That consistency lets you check your understanding. If a query result in Chapter 13 disagrees with an SLO calculation in Chapter 22, one of them is wrong, and you should be able to tell which.

Honest Limits

It would be easy to oversell observability. Here is what it does not do, so you can set fair expectations with your team and your managers.

It costs money and effort

Instrumentation takes engineering time. Telemetry pipelines and storage take infrastructure. Dashboards, alerts, and runbooks need maintenance, or they rot. The cost is real and ongoing. The return is faster diagnosis, fewer long incidents, safer deployments, and better decisions about reliability work, but you should treat observability as an investment to manage, not a free upgrade. Part VI is devoted to keeping the cost under control.