

OpenTelemetry in Production

DESIGNING, DEPLOYING, AND OPERATING OBSERVABILITY AT SCALE

A practical, end-to-end guide to building and running
production-grade observability with OpenTelemetry



METRICS

Understand
system health
and performance



LOGS

Troubleshoot
issues and
find answers



TRACES

Follow requests
and optimize
experiences



COLLECT

Reliable data
collection at scale



PROCESS

Transform, correlate,
and enrich telemetry



EXPORT

Deliver to the backends
you trust



OPERATE

Secure, reliable,
and cost-effective
observability



END-TO-END COVERAGE

From fundamentals
to advanced
production patterns



PRACTICAL GUIDANCE

Real-world examples,
configurations,
and code samples



PRODUCTION FOCUS

Performance tuning,
security hardening,
and reliability



COST AWARE

Control telemetry
costs without
sacrificing value



TROUBLESHOOT WITH CONFIDENCE

Proven techniques
to detect, diagnose,
and resolve issues

KEVIN GRANT

OpenTelemetry in Production

Designing, Deploying, and Operating Observability at Scale

Steve Publications

This book is available at <https://leanpub.com/opentelemetryinproduction>

This version was published on 2026-08-22



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Steve Publications

Contents

Designing, Deploying, and Operating Observability at Scale	1
Introduction: The Observability Crisis and OpenTelemetry’s Answer . .	2
The Cost of Fragmented Observability	2
What OpenTelemetry Is (and Is Not)	3
How This Book Is Organized	4
Chapter 1: Foundations: Signals, Standards, and Architecture	7
The Three Pillars of Telemetry	7
OpenTelemetry Project Structure and Governance	8
APIs, SDKs, and Specifications	9
Semantic Conventions and Their Importance	11
The End-to-End Architecture	12
Chapter 2: Distributed Tracing in Depth	15
How Distributed Tracing Works Under the Hood	15
Context Propagation Across Service Boundaries	16
Span Design and Semantic Conventions	18
Sampling Strategies for Production Scale	20
Baggage and Trace Attributes	21
Chapter 3: Metrics That Matter	24
Metric Types and Aggregation Semantics	24
Cardinality: The Silent Killer	24
Histograms, Summaries, and Latency Measurement	24
Designing SLOs and SLIs with OpenTelemetry Metrics	24
Exporting Metrics to Backends	24
Chapter 4: Logs as First-Class Telemetry	25
Structured Logging and OpenTelemetry	25
Correlating Logs with Traces and Metrics	25

CONTENTS

Log Levels and Severity Mapping	25
High-Volume Log Collection Strategies	25
PII and Sensitive Data in Logs	25
Chapter 5: Instrumentation Strategies	26
Auto-Instrumentation: How It Works and When to Use It	26
Manual Instrumentation Patterns and Best Practices	26
Hybrid Instrumentation for Complex Systems	26
Language and Framework Considerations	26
Migrating from Legacy Instrumentation	26
Chapter 6: The OpenTelemetry Collector: Architecture Deep Dive	27
Collector Architecture and Component Model	27
Receivers and Protocols	27
Processors: Transformation and Enrichment	27
Exporters and Backend Integration	27
Connectors and Cross-Signal Pipelines	27
Configuration Patterns and Validation	27
Chapter 7: Deployment Patterns for Production	29
Agent vs Gateway vs Hybrid Architectures	29
Kubernetes Deployment with DaemonSets and Deployments	29
Sidecar Patterns in Service Mesh Environments	29
Multi-Cluster and Multi-Region Topologies	29
High Availability and Disaster Recovery	29
Chapter 8: Performance, Scaling, and Capacity Planning	30
Understanding Telemetry Volume at Scale	30
Batching and Queue Configuration	30
Backpressure Handling and Flow Control	30
Load Balancing Collector Deployments	30
Capacity Planning and Cost Management	30
Diagnosing Performance Bottlenecks	30
Chapter 9: Security, Privacy, and Governance	32
Transport Security with TLS and mTLS	32
Authentication and Authorization Models	32
Handling PII and Sensitive Data	32
Multi-Tenancy and Isolation	32
Governance and Policy Enforcement	32

Chapter 10: Integration with Observability Backends	33
Vendor-Neutral Backend Architecture	33
Integrating with Open Source Backends	33
Commercial Backend Integration Patterns	33
Multi-Backend Export Strategies	33
Designing Your Observability Stack	33
Chapter 11: Operations, Monitoring, and Troubleshooting	34
Monitoring the OpenTelemetry Infrastructure	34
Debugging Missing or Malformed Telemetry	34
Collector Health Checks and Diagnostics	34
Upgrade Strategies and Version Management	34
Operational Runbooks for Common Issues	34
Chapter 12: Advanced Patterns and Future Directions	35
Custom Instrumentation and Extensions	35
Dynamic Configuration and Service Discovery	35
Observability Pipelines and Data Quality	35
Advanced APM Patterns with OpenTelemetry	35
The Future of OpenTelemetry and Observability	35
Conclusion: Building an Observability Culture	36
From Tooling to Capability	36
Measuring the Value of OpenTelemetry	36
Continuous Improvement in Observability	36
References and Further Reading	37
Official OpenTelemetry Specifications and Documentation	37
Project Governance and Community	37
Collector Components and Implementation	37
Language SDKs and Instrumentation	37
Deployment and Kubernetes	37
Backend Integration	37
Security and Privacy	38
Community Resources and Articles	38

Designing, Deploying, and Operating Observability at Scale

This book is for software engineers, SREs, DevOps engineers, platform engineers, architects, and technical leaders who need to build, deploy, operate, and scale production-grade observability using OpenTelemetry. It covers the full lifecycle from foundational concepts through advanced deployment patterns, performance tuning, security hardening, troubleshooting, and cost management. Every chapter provides conceptual depth followed by concrete implementation guidance with configuration examples, code samples, architecture descriptions, and operational workflows you can apply directly in real environments. The content is grounded in official OpenTelemetry specifications, current best practices, and production-tested patterns from organizations running telemetry at massive scale.

Introduction: The Observability Crisis and OpenTelemetry's Answer

The Cost of Fragmented Observability

In 2019, a major fintech company ran into a problem that has become familiar to engineering organizations worldwide. Their payments platform spanned twelve microservices across three cloud providers. When latency spiked during peak trading hours, the on-call engineer needed data from four different dashboards: Datadog for application metrics, New Relic for JVM profiling, Splunk for logs, and Jaeger for traces. Each tool had its own agent installed on every host. Each consumed CPU and memory independently. None of them shared context with the others, so correlating a slow database query in the traces with an error burst in the logs required manual timestamp matching and service name guessing. The incident lasted four hours instead of forty minutes because the engineer could not see the full picture quickly enough.

This scenario is not exceptional. It is the default state for organizations that build distributed systems without a unified observability strategy. As services multiply, so do instrumentation concerns: which agent to install on each language runtime, how to propagate trace context across service boundaries when different teams use different tracing libraries, how to ensure log records include the same identifiers as traces, how to avoid vendor lock-in when the current provider raises prices or deprecates features, and how to manage rising costs as telemetry volume grows faster than revenue.

The fragmentation problem has three dimensions that compound each other:

First, tool sprawl creates operational overhead. Every observability agent is another process to configure, monitor, upgrade, and troubleshoot. Agents compete for system resources, occasionally conflict with each other, and introduce their own failure modes into the infrastructure they are supposed to help you understand.

Second, data silos undermine the value of observability itself. When traces live in one system, metrics in another, and logs in a third, you cannot answer questions that span multiple signals without leaving your primary interface. The entire point of observability is to enable investigation through correlation. Correlation requires data to be connected, not scattered.

Third, vendor lock-in constrains both technical and business flexibility. Proprietary instrumentation APIs tie your application code to a specific provider's ecosystem. Migrating away from that provider becomes an engineering project rather than a configuration change. When contracts come up for renewal, you are negotiating from weakness because the cost of leaving is measured in developer months and risk.

What OpenTelemetry Is (and Is Not)

OpenTelemetry emerged from the merger of two competing CNCF projects: OpenTracing, which defined APIs for distributed tracing, and OpenCensus, which provided libraries for metrics and stats collection. The merger happened in 2019, just as the industry was realizing that treating traces, metrics, and logs as separate concerns was part of the problem rather than the solution.

OpenTelemetry is a specification, not a product. It defines how telemetry data should be generated, represented, transmitted, and consumed. The project produces language-specific APIs and SDKs that application developers use to instrument their code. It produces the OpenTelemetry Collector, a standalone process that receives, processes, and exports telemetry data. It produces semantic conventions, which are standardized attribute names and values that ensure consistency across different services and teams. And it produces OTLP, the OpenTelemetry Protocol, which is the native wire format for transporting telemetry between components.

What OpenTelemetry is not: a storage backend, a visualization layer, an alerting system, or a complete observability platform. OpenTelemetry does not replace Prometheus, Grafana, Datadog, New Relic, or any other observability tool. It replaces the proprietary instrumentation and collection layers that tie you to those tools. You still need somewhere to store your telemetry data and interfaces to query it. What you gain is the ability to choose those backends independently of how you instrument your applications, and to change them without rewriting instrumentation code.

This distinction matters because OpenTelemetry shifts where complexity lives. Before OpenTelemetry, complexity was distributed across every vendor SDK you used, each with its own configuration model, performance characteristics, and upgrade cycle. With OpenTelemetry, complexity concentrates in two places: the instrumentation layer (which becomes standardized and portable) and the collection pipeline (which becomes a first-class piece of infrastructure that requires deliberate design). The trade-off is worth it because you now control the architecture instead of inheriting whatever each vendor decided works best for their ecosystem.

OpenTelemetry's maturity has accelerated dramatically in recent years. As of 2026, it is a CNCF Graduated project with over 12,000 contributors from more than 2,800 organizations. Every major observability vendor accepts OpenTelemetry data as a primary ingestion format. The core signals (tracing, metrics, and logs) are all stable. Semantic conventions for HTTP, database, RPC, and many other domains have graduated to stable status. Auto-instrumentation is production-ready for Java, Python, Node.js, .NET, Go, and several other languages. The Collector ecosystem includes hundreds of receivers, processors, exporters, connectors, and extensions that cover virtually every integration scenario you can imagine.

This book assumes OpenTelemetry is no longer a question of whether to adopt it but how to do so correctly. Getting it wrong is easy: misconfigured sampling that drops critical traces, uncontrolled metric cardinality that crashes your backend, Collectors that OOM under load, telemetry pipelines that leak PII into storage, or architectures that create new single points of failure. Getting it right requires understanding not just the components but how they interact at scale, what trade-offs each design choice entails, and how to operate the resulting infrastructure reliably over time.

How This Book Is Organized

The chapters are structured as a progression from foundations through production architecture to advanced operations. Each chapter provides conceptual explanations followed by implementation guidance, configuration examples, code samples, and operational considerations. The goal is that you can read the book cover to cover for a complete understanding or jump to specific chapters when facing concrete problems.

Chapter 1 establishes the conceptual foundation: what telemetry signals are, how OpenTelemetry's architecture works, the relationship between APIs, SDKs, and specifications, why semantic conventions matter, and how all the pieces fit together in an end-to-end system.

Chapters 2 through 4 dive deep into each telemetry signal. Chapter 2 covers distributed tracing: context propagation, span semantics, sampling strategies, baggage, and practical implementation across service boundaries. Chapter 3 focuses on metrics: cardinality management, aggregation models, histograms versus summaries, SLO design, and production-grade metric collection. Chapter 4 treats logs as first-class OpenTelemetry telemetry: structured logging, correlation with traces and metrics, severity mapping, and high-volume log collection strategies.

Chapter 5 addresses instrumentation strategies: auto-instrumentation mechanics and trade-offs, manual instrumentation patterns, hybrid approaches for complex systems, language-specific considerations, and migration from legacy or vendor-specific agents.

Chapters 6 through 8 cover the OpenTelemetry Collector in depth. Chapter 6 explains its architecture, component model, pipeline configuration, and how receivers, processors, exporters, and connectors work internally. Chapter 7 covers deployment patterns: agent versus gateway architectures, Kubernetes deployments, sidecar patterns, multi-cluster topologies, and high availability design. Chapter 8 focuses on performance and scaling: batching, queues, backpressure handling, load balancing, resource sizing, capacity planning, and bottleneck diagnosis.

Chapter 9 addresses security, privacy, and governance: TLS and mTLS configuration, authentication models, PII handling, multi-tenancy, RBAC for telemetry data, and compliance considerations.

Chapter 10 covers integration with observability backends: patterns for connecting OpenTelemetry to Prometheus, Grafana stack components, Jaeger, Tempo, Datadog, New Relic, Splunk, Elastic, cloud provider services, and multi-backend export strategies.

Chapters 11 and 12 focus on operations. Chapter 11 provides guidance for monitoring the telemetry infrastructure itself, debugging missing or malformed data, operational runbooks, upgrade procedures, and incident response. Chapter 12 covers advanced patterns: custom instrumentation, dynamic configuration, service discovery, observability pipelines, APM design

with OpenTelemetry, and where the project is heading.

The Conclusion synthesizes the book's lessons into a framework for building observability as an organizational capability, emphasizing that technology alone does not solve observability challenges; culture, processes, and continuous improvement matter equally.

Throughout the book, I distinguish between what is defined by the OpenTelemetry specification, what is production-proven practice, what remains experimental or evolving, and where vendor-specific functionality diverges from the core standard. The goal is that you finish with not just knowledge of how OpenTelemetry works but judgment for making sound decisions about how to deploy it in your own environment.

Chapter 1: Foundations: Signals, Standards, and Architecture

The Three Pillars of Telemetry

Observability rests on three primary signals: traces, metrics, and logs. Each serves a distinct purpose, each has its own strengths and limitations, and effective observability requires using all three in concert rather than relying on any single one. OpenTelemetry treats these as first-class citizens with unified APIs, shared context propagation, and consistent data models.

Traces represent the flow of requests through a distributed system. A trace is a directed acyclic graph of spans, where each span represents a named, timed operation. The root span corresponds to an incoming request or initiating event. Child spans represent downstream calls, database queries, cache lookups, message processing, or any other discrete unit of work. Spans are linked by parent-child relationships that reconstruct the full path a request took through your system. Traces answer questions about latency, dependencies, and failure propagation: which service introduced the delay, which call failed first, how many hops did this request make before timing out.

Metrics are numeric measurements recorded over time. OpenTelemetry defines four primary instrument types: counters for monotonically increasing values like request counts or bytes transferred, gauges for point-in-time measurements like current memory usage or queue depth, histograms for distributions of observed values like response latencies or payload sizes, and up-down counters for values that can increase or decrease like active connections or available capacity. Metrics answer questions about system behavior at aggregate level: how many requests are failing, what is the p99 latency, is memory trending upward, did throughput drop after the deployment.

Logs are timestamped records of discrete events. Each log record has a severity level, a message body, and optional structured attributes that provide context. Logs answer questions about specific occurrences: why did this particular request fail, what input caused the error, what state was the system

in when the crash happened. Modern logging with OpenTelemetry emphasizes structured formats where each attribute is machine-readable, enabling filtering and aggregation rather than regex parsing of free-form text.

The power of combining these signals lies in correlation. When a metric shows elevated error rates, you want to see the traces that represent those failed requests. When a trace shows an unexpected span, you want to read the logs emitted during that operation. OpenTelemetry enables this correlation through shared identifiers and context propagation: every log record can carry a trace ID and span ID, every metric data point can reference exemplar traces, and every signal shares resource attributes that identify which service produced it.

A fourth concept underlies all three signals: resources. A resource represents the entity producing telemetry: a process running in a container on a Kubernetes pod in an AWS region. Resources are described by key-value attributes like `service.name`, `deployment.environment`, `cloud.provider`, and `k8s.pod.name`. Resource attributes are attached to every span, metric, and log record produced by that entity, enabling you to filter, group, and aggregate telemetry across dimensions like environment, cluster, team, or infrastructure provider. Resources make it possible to answer questions like “show me all errors in production” or “compare latency between our US and EU clusters” without embedding those dimensions into every individual data point.

OpenTelemetry Project Structure and Governance

OpenTelemetry is a CNCF Graduated project, which is the highest maturity level in the foundation’s lifecycle model. Graduation requires demonstrated technical stability, broad community adoption, transparent governance, and sustainable maintenance practices. For OpenTelemetry, graduation in May 2026 reflects years of work consolidating competing observability standards into a single vendor-neutral specification.

The project operates under a Governance Committee elected by the community, supported by a Technical Committee that handles architectural decisions. Day-to-day development happens in Special Interest Groups (SIGs) focused on specific areas: each language has its own SIG (Java, Go, Python, JavaScript, .NET, Ruby, PHP, C++, etc.), plus SIGs for Collector, Specifications, Semantic Conventions, Configuration, and others. This structure ensures that

decisions are made by people with deep expertise in the relevant domain rather than a centralized team.

Vendor neutrality is not just a principle but an operational requirement. No single company can dominate decision-making. Major observability vendors including Datadog, New Relic, Splunk, Dynatrace, Grafana Labs, Honeycomb, and others are all contributors and maintainers. Cloud providers like AWS, Google Cloud, and Microsoft contribute through their own distributions and integrations. This broad participation is why OpenTelemetry has become the de facto standard: competitors who would normally guard proprietary formats have aligned behind a common specification because it serves their mutual interest in reducing fragmentation.

The project's scale is notable. As of 2026, there are over 12,000 contributors from more than 2,800 organizations. The core repositories on GitHub include the specification itself, language SDKs for over a dozen programming languages, the Collector and its contrib components, semantic conventions, protocol definitions, documentation sites, demo applications, and tooling like the Operator for Kubernetes and the Collector Builder for creating custom distributions. This is not a niche framework maintained by a handful of enthusiasts; it is infrastructure that underpins observability across a significant portion of the industry's distributed systems.

APIs, SDKs, and Specifications

OpenTelemetry's architecture separates concerns into distinct layers: the API, the SDK, semantic conventions, and contrib packages. Understanding these layers and their relationships is essential for using OpenTelemetry correctly and avoiding common pitfalls.

The API layer defines interfaces for instrumentation without prescribing implementation details. When you write `tracer.startSpan("process-order")`, you are calling the tracing API. The API provides abstractions for creating spans, recording metrics, writing logs, managing context propagation, and handling baggage. Critically, instrumentation code depends only on the API, never directly on the SDK. This separation enables multiple implementations to coexist in the same application environment and allows the SDK to evolve independently without breaking instrumented code.

The SDK layer provides a reference implementation of the API. It manages span creation and export, metric aggregation, log record handling, resource

detection, sampling decisions, and batching behavior. The SDK is configurable: you choose exporters, set sampling rates, define resource attributes, configure processors, and tune performance parameters. Application owners control the SDK configuration because they understand their observability requirements and constraints. Instrumentation authors (people who write libraries that emit telemetry on behalf of application developers) must never reference the SDK directly. If a library hardcodes an exporter or sampling strategy, it steals configuration authority from the application owner and creates conflicts when multiple libraries try to configure the same provider.

This distinction between instrumentation authors and application owners is one of OpenTelemetry’s most important design principles. Consider a web framework that includes tracing for HTTP requests. The framework author writes instrumentation using the API: “when an HTTP request arrives, start a span; when it completes, end the span with appropriate status.” The application owner configures the SDK: “export spans to our Collector at localhost:4317, sample 10 percent of traces, and label everything as belonging to the payments service.” Neither party needs to know how the other operates.

The specification defines requirements that all compliant implementations must satisfy. It covers signal semantics (what a span is, what attributes it carries), API contracts (method signatures, behavior guarantees), SDK responsibilities (aggregation algorithms, export mechanics), wire protocols (OTLP format and transport), and context propagation rules. The specification uses precise language: **MUST** for mandatory requirements, **SHOULD** for recommended practices, **MAY** for optional features. This rigor ensures interoperability across implementations from different vendors and languages.

Semantic conventions are a critical but sometimes overlooked piece of the puzzle. They define standardized attribute names and values for common operations. When you instrument an HTTP server, semantic conventions tell you to use `http.request.method` for the request method, `url.path` for the path template, and `http.response.status_code` for the response code: not `method`, `path`, and `status` or any other variation. Without semantic conventions, every team would invent its own attribute names, and correlating data across services would require manual mapping. With them, a query like “show all HTTP 500 errors” works uniformly across every service that follows the conventions.

Semantic conventions have stability levels: **Development** for early proposals, **Release Candidate** for nearly finalized definitions, and **Stable** for

production-ready standards that will not change except through deprecation. As of 2026, conventions for HTTP, database operations, RPC frameworks, messaging systems, cloud providers, Kubernetes, and many other domains are stable. New domains like generative AI observability are still maturing. The key practice is to use stable conventions wherever available and be prepared to adapt when new conventions graduate from development to stability.

Contrib packages provide optional integrations that extend the core SDK with functionality for specific frameworks, libraries, or platforms. Examples include auto-instrumentation agents for Spring Boot or Express.js, resource detectors for AWS ECS or Google Cloud Run, exporters for Datadog or Prometheus, and processors for enriching telemetry with Kubernetes metadata. Contrib packages are maintained by the community with less stringent stability guarantees than core components. They are valuable for accelerating adoption but should be evaluated carefully in production environments because they may change more frequently or have varying quality across different integrations.

Semantic Conventions and Their Importance

Semantic conventions deserve their own discussion because getting them wrong is one of the most common causes of observability pain. The problem is subtle: when you first instrument a service, any attribute name works fine. You call it `endpoint` instead of `url.path`, or `error` instead of `exception.type`, and your traces look great in the dashboard. Six months later, you have twenty services with twenty different naming schemes, and building cross-service queries becomes a nightmare of inconsistent field names.

OpenTelemetry semantic conventions solve this by providing a shared vocabulary. The conventions are organized into domains that correspond to real-world concepts: HTTP for web requests, DB for database operations, RPC for remote procedure calls, messaging for queue systems, FAAS for serverless functions, and so on. Each domain defines which attributes should be set, what values they can take, and when they apply.

For example, the HTTP semantic conventions define that a server span representing an incoming HTTP request should include:

- `http.request.method`: The HTTP method (GET, POST, etc.)

- `url.path`: The path component of the URL with dynamic segments replaced by placeholders (e.g., `/users/{id}` instead of `/users/12345`)
- `http.response.status_code`: The HTTP response status code as an integer
- `network.protocol.version`: The HTTP version (1.1, 2, etc.)

A client span making an outgoing HTTP request includes a similar but distinct set of attributes reflecting the caller's perspective. These conventions ensure that when you query “show all requests to `/users/{id}` with status 500,” the results include data from every service instrumented correctly, regardless of which team wrote it or which language it uses.

The most important convention is arguably for span names themselves. OpenTelemetry recommends that span names be low-cardinality templates describing the type of operation, not high-cardinality values that vary per request. A span name like `GET /users/{id}` is correct because it describes a category of operations. A span name like `GET /users/12345` is wrong because it creates a unique span name for every user ID, which destroys aggregation and explodes storage costs. This pattern applies across all domains: use templates with placeholders, never embed identifiers directly in names.

Semantic conventions are not optional decoration. They are the foundation that makes telemetry data queryable, aggregatable, and comparable at scale. Teams that ignore them pay the price later when they need to build dashboards or alerts that span multiple services and discover that nobody agreed on what “success” looks like. The investment in learning and following conventions pays dividends immediately through consistent data quality and compounds over time as your system grows.

The End-to-End Architecture

OpenTelemetry's end-to-end architecture spans from instrumentation code inside application processes to storage and visualization backends, with the Collector serving as the central processing hub. Understanding this flow is essential for designing reliable observability infrastructure.

At the source, instrumented applications generate telemetry using OpenTelemetry SDKs. The SDK collects spans, metric data points, and log records in memory, applies sampling decisions where configured, batches them for

efficiency, and exports them via an OTLP exporter. The export destination is typically a local or nearby Collector instance rather than directly to a backend, because the Collector provides critical processing capabilities that individual applications should not be responsible for.

The Collector receives telemetry through receivers, which support multiple protocols including OTLP (the native format), Jaeger, Zipkin, Prometheus, Fluentd, and many others. Receivers normalize incoming data into OpenTelemetry's internal representation regardless of the source format. This normalization is important because it allows you to ingest telemetry from legacy systems using older protocols alongside new systems using OTLP, all flowing through the same processing pipeline.

Processors transform, enrich, filter, or reduce telemetry as it passes through the Collector. Common processors include batching (grouping data for efficient export), memory limiting (preventing out-of-memory crashes under load), sampling (selecting representative traces for storage), attribute manipulation (adding, removing, or modifying metadata), resource detection (discovering environment information and attaching it to telemetry), filtering (dropping unwanted data), and redaction (removing sensitive information). Processors are arranged in a pipeline order that matters: memory limiting should always come first to protect the Collector, sampling decisions should happen before expensive enrichment operations, and redaction should occur before any export.

Exporters send processed telemetry to backends. Exporters support OTLP for sending to other Collectors or OTLP-compatible backends, as well as native protocols for specific systems like Prometheus remote write, Jaeger Thrift, Kafka, file output, and many vendor-specific formats. A single Collector can export to multiple backends simultaneously, enabling multi-tenant architectures, data replication across regions, or gradual migration from legacy tools.

Connectors are a specialized component type that bridges pipelines by acting as both an exporter in one pipeline and a receiver in another. The most common connector is spanmetrics, which generates aggregate metrics (request rates, error counts, latency histograms) directly from trace data, eliminating the need for separate instrumentation. Other connectors enable routing based on attributes, forwarding data between signal types, or creating service dependency graphs from traces.

Extensions provide infrastructure support that does not process telemetry

directly but enables Collector functionality: health check endpoints for Kubernetes probes, pprof interfaces for performance profiling, zpages for diagnostic inspection, authentication mechanisms for securing receivers and exporters, and configuration providers for dynamic updates.

This architecture is designed for composition and flexibility. A simple deployment might have a single Collector instance receiving OTLP from local applications and exporting to one backend. A production-grade deployment at scale typically uses multiple tiers: lightweight agent Collectors running on each host collecting telemetry locally, gateway Collectors aggregating data from agents and applying centralized processing like sampling and enrichment, and possibly additional tiered gateways for multi-cluster or multi-region topologies before final export to backends. Each tier can have different configurations optimized for its role.

The key architectural insight is that OpenTelemetry separates concerns deliberately: applications focus on generating accurate telemetry with minimal overhead, Collectors handle transformation and routing logic centrally, and backends specialize in storage and analysis. This separation enables each layer to evolve independently and scale according to its own requirements rather than being constrained by the needs of the others.

Chapter 2: Distributed Tracing in Depth

How Distributed Tracing Works Under the Hood

Distributed tracing answers a fundamental question about complex systems: when a user makes a request, what actually happened behind the scenes? In a monolithic application, you can answer this with a single log file or debugger. In a distributed system where one request triggers calls across dozens of services, databases, caches, and message queues, you need a way to reconstruct the full journey.

A trace is a complete record of a request's path through your system. It consists of spans, each representing a discrete unit of work: an HTTP handler processing a request, a database query executing, a cache lookup completing, a downstream service call finishing. Spans are organized hierarchically with parent-child relationships that reflect the causal structure of operations. The root span has no parent and represents the entry point into the system. Every other span has exactly one parent, which is the operation that triggered it.

Each span carries several pieces of essential information:

- A unique span ID (8 bytes) identifying this specific operation
- A trace ID (16 bytes) shared by all spans in the same trace
- A parent span ID linking back to the operation that created it
- A name describing what kind of operation this is
- Start and end timestamps defining its duration
- Status indicating success, error, or unknown outcome
- Attributes providing context like HTTP method, database statement, or queue name
- Events recording notable occurrences within the span's lifetime
- Links connecting to other spans that are related but not causally parent-child

The trace ID is the glue that holds everything together. Every span belonging to the same logical request shares the same trace ID. When you

query for a trace in your observability backend, it collects all spans with that trace ID and reconstructs the full picture. Without the shared identifier, you would have isolated fragments of information with no way to connect them.

The parent-child relationship establishes causality. If span B is a child of span A, we know that operation A triggered operation B. This relationship enables latency analysis: the root span's duration represents total request latency, while child spans show where time was spent within that request. When something goes wrong, the hierarchy reveals which service failed first and how the failure propagated downstream.

Links are a more flexible relationship type for cases where strict parent-child causality does not apply. Links are useful for batch operations (one span triggers many others simultaneously), retries (a retried operation links back to the original attempt), or cross-trace references (an async callback triggered by an earlier trace). Links enable representing complex workflows that do not fit a simple tree structure.

Under the hood, spans exist only in memory until they are exported. The SDK collects completed spans, batches them for efficiency, and sends them to an exporter configured by the application owner. The exporter typically uses OTLP over gRPC or HTTP to deliver spans to a Collector, which then processes and forwards them to storage backends. This pipeline introduces latency between when a span ends and when it appears in your dashboards (usually seconds to minutes depending on batching configuration), which is an important operational consideration when debugging real-time issues.

Context Propagation Across Service Boundaries

The hardest part of distributed tracing is not creating spans inside a single process but connecting them across service boundaries. When service A calls service B, service B must know that its work is part of the same trace as service A's request. This requires propagating trace context (the trace ID, current span ID, and sampling decision) from caller to callee through whatever transport mechanism they use.

OpenTelemetry implements context propagation based on the W3C Trace Context specification, which defines standard HTTP headers for carrying trace information:

- **traceparent:** Contains the trace ID, span ID, trace flags (including sampling decision), and version number in a compact format like `00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01`
- **tracestate:** An optional vendor-specific header for carrying additional state that different tracing systems use for their own purposes

When service A makes an outgoing HTTP request, its OpenTelemetry instrumentation automatically injects the current trace context into the request headers before sending. When service B receives the request, its instrumentation extracts the context from incoming headers and uses it to create a child span of the calling span. This happens transparently through auto-instrumentation for most popular HTTP frameworks and libraries.

The mechanism works identically for other transports: gRPC metadata instead of HTTP headers, message attributes for queue systems like Kafka or RabbitMQ, custom headers for internal protocols. The OpenTelemetry specification defines propagation rules for each transport type, and instrumentation libraries implement them so developers do not need to handle context injection manually in most cases.

Context propagation is fragile in practice because it depends on every hop in the chain participating correctly. If any service fails to extract incoming context or inject outgoing context, the trace breaks at that point. Common failure modes include:

- Uninstrumented services that do not understand OpenTelemetry headers and drop them
- Custom HTTP clients that bypass instrumented libraries and send raw requests without context
- Message brokers that strip headers during forwarding unless explicitly configured to preserve them
- Load balancers or proxies that remove unknown headers as a security precaution
- Browser-based frontends where JavaScript SDKs must manually propagate context in fetch calls

Debugging broken context propagation requires systematic verification at each boundary. The most reliable approach is to temporarily enable debug-level logging on the OpenTelemetry SDK and inspect whether context is being

injected into outgoing requests and extracted from incoming ones. Many teams also add a temporary span or log statement that prints the current trace ID at key points in the request flow to visually confirm continuity.

Context propagation also applies beyond traces. The same mechanism carries baggage (custom key-value pairs discussed later) and enables correlation of logs with their originating trace. When an application writes a log record, it can include the active trace ID and span ID so that the log appears as part of the trace in your observability backend. This correlation is one of the most powerful debugging features available: you can see not just the sequence of operations but also the detailed logs emitted during each operation.

Span Design and Semantic Conventions

Good spans make traces useful. Bad spans make them noisy, expensive, or misleading. The difference lies in how carefully you design span names, attributes, events, and status codes to convey meaningful information without overwhelming your backend with unnecessary data.

Span names should describe the type of operation, not the specific instance. A span named `GET /users/{id}` tells you that this is a user lookup request. A span named `GET /users/12345` creates a unique span name for every user ID, which destroys aggregation and makes it impossible to answer questions like “what is the average latency for user lookups?” OpenTelemetry semantic conventions provide recommended span naming patterns for common operations: HTTP handlers use `{method} {route_template}`, database queries use `DB_SYSTEM operation TABLE_NAME`, message consumers use `SYSTEM consume TOPIC`.

Attributes provide context that makes spans queryable and analyzable. Semantic conventions define which attributes to include for each type of operation. For an HTTP server span, you should include the request method, URL path template, response status code, protocol version, and user agent. For a database span, include the database system type, connection string (without credentials), statement or query template, and row count if available. The key principle is to use standardized attribute names so that queries work uniformly across all services.

Not every piece of information belongs in an attribute. Attributes have costs: they increase span size, which increases storage consumption and

export latency. They also increase metric cardinality when spans are converted to metrics via the spanmetrics connector. Be selective about what you include. Ask whether each attribute will be used for filtering, grouping, or analysis. If the answer is no, it probably does not belong in a span attribute.

High-cardinality attributes are particularly dangerous. An attribute like `user.id` with millions of unique values creates a different combination for every user, which can crash metrics backends that index all attribute values. The OpenTelemetry specification explicitly warns against using high-cardinality identifiers as metric dimensions. If you need to track per-user behavior, use traces and logs (which handle high cardinality better) rather than metrics, or aggregate users into buckets like “authenticated” versus “anonymous.”

Span status codes should reflect the outcome from the operation’s perspective, not just whether an exception was thrown. OpenTelemetry defines three statuses: UNSET (the default, meaning the operation completed without explicitly indicating success or failure), OK (explicitly successful), and ERROR (something went wrong). Setting ERROR status with a descriptive description message enables filtering traces by failures across your entire system. Many teams make the mistake of leaving spans as UNSET even when they know the operation failed, which makes it harder to identify problematic requests.

Events record discrete occurrences within a span’s lifetime that are notable but do not change the span’s overall status. Examples include logging an exception that was caught and handled gracefully, recording a retry attempt, or noting a configuration change detected during processing. Events are useful for debugging without cluttering the primary span attributes with transient details.

Links connect spans across trace boundaries when strict parent-child relationships do not apply. A batch job that processes multiple items might create one root span with links to individual item spans in separate traces. A retry mechanism might link a retried span back to the original attempt’s trace. Links enable representing complex workflows without forcing everything into a single tree structure.

The most important rule for span design is consistency. Follow semantic conventions religiously so that your traces are queryable and comparable across services. Use low-cardinality names with placeholders instead of embedded identifiers. Include attributes that will actually be used for analysis. Set meaningful status codes. These practices make the difference between

traces that help you debug production incidents in minutes and traces that become an expensive liability nobody can interpret.

Sampling Strategies for Production Scale

At scale, collecting every trace is economically impossible. A moderately sized microservices system generating 10,000 requests per second with each request creating 50 spans produces 500,000 spans per second. At roughly 1 KB per span, that is approximately 43 TB of trace data per month before compression. Even with aggressive sampling and retention policies, storing all traces from a large system costs more than most organizations can justify.

Sampling selects a representative subset of traces for export while dropping the rest. The challenge is to reduce volume without losing visibility into what matters: errors, slow requests, unusual patterns, and production incidents. OpenTelemetry supports two fundamental sampling strategies with very different characteristics: head-based sampling and tail-based sampling.

Head-based sampling makes the decision at the beginning of a trace, typically when the root span is created. The most common approach is probabilistic sampling: each trace has a fixed probability (say 10 percent) of being sampled, determined by hashing the trace ID against a threshold. This approach is simple and efficient because the decision happens immediately with no state to maintain. Downstream services propagate the sampling decision via context headers, so they know whether to create full spans or minimal ones.

The problem with head-based sampling is that it cannot see what will happen in the trace before deciding whether to keep it. A 10 percent sample rate means you capture roughly 10 percent of errors and slow requests, but also 90 percent of perfectly normal traces. More critically, rare events become invisible: if something fails once every million requests, a 10 percent sample rate gives you only a one-in-ten-thousand chance of capturing it. For debugging intermittent issues or monitoring tail latency, head-based sampling alone is insufficient.

Tail-based sampling makes the decision after seeing most or all spans in the trace. It collects traces temporarily, evaluates them against configurable policies, and then decides whether to export or drop them. Policies can be based on:

- Status code: always keep traces with ERROR spans
- Latency: always keep traces slower than a threshold like 2 seconds
- Attribute matching: keep traces from specific services or with specific attributes
- Probability: sample a percentage of remaining traces after applying other policies

Tail-based sampling solves the visibility problem: you can guarantee capturing all errors and slow requests while still dropping normal traffic. The trade-off is operational complexity. Tail samplers must maintain state for in-flight traces, which consumes memory proportional to trace volume and decision latency. They introduce their own point of failure: if the tail sampler crashes, all buffered traces are lost. And they require careful tuning of parameters like `decision_wait` (how long to wait before deciding) and `num_traces` (how many traces to buffer).

In production, most organizations use a hybrid approach. Head-based sampling at the application level reduces volume entering the pipeline to a manageable percentage (say 20 percent). Tail-based sampling at the Collector level then applies intelligent policies to that subset: keep all errors, keep all slow traces, sample 10 percent of the rest. This combination achieves both efficiency and visibility while limiting the memory requirements of tail sampling.

The OpenTelemetry Collector provides processors for both strategies. The `probabilistic_sampler` processor implements head-based sampling within the Collector pipeline. The `tail_sampling` processor implements tail-based sampling with configurable policies including status code, latency, string attribute matching, numeric attribute thresholds, and composite AND/OR logic. Both can be combined in a single pipeline to create sophisticated sampling architectures.

Sampling decisions have downstream implications that teams often overlook. Sampled traces are not statistically representative if you use biased strategies like always-keep-errors, so you cannot accurately calculate overall error rates from trace data alone. For metrics-driven monitoring, rely on counters and histograms that record every event, not sampled traces. Trace-based metrics derived via the `spanmetrics` connector should only be computed before sampling occurs to avoid distortion. And always maintain a small probabilistic sample of normal traffic to ensure your dashboards show realistic behavior patterns rather than just failures.

Baggage and Trace Attributes

Baggage is OpenTelemetry's mechanism for propagating application-defined key-value pairs across service boundaries alongside trace context. Unlike span attributes, which exist only within a single span, baggage entries travel with the trace through every hop that respects context propagation. This makes baggage useful for carrying contextual information that downstream services need to include in their own telemetry: user identifiers, tenant IDs, feature flags, request priorities, or any other metadata that should be available throughout the request lifecycle.

The mechanism is straightforward. When service A sets a baggage entry like `tenant-id=acme-corp`, that entry is included in the W3C Baggage header (`baggage: tenant-id=acme-corp`) when making outgoing requests. Service B extracts the baggage from incoming headers and can read it, add more entries, or use entries to populate attributes on its own spans and logs. The baggage persists through the entire trace unless explicitly modified or cleared.

Baggage is powerful but dangerous. Every baggage entry increases header size for every request in the trace. In a system with many services making multiple calls per request, even small baggage values multiply across the wire. A 1 KB baggage payload on a request that touches fifty services generates 50 KB of extra header traffic per request. At high volume, this overhead becomes significant.

More critically, baggage travels in HTTP headers where it is visible to anyone who can inspect network traffic: load balancers, proxies, CDNs, third-party APIs, and potentially attackers. The W3C Baggage specification explicitly warns against putting sensitive data in baggage because there are no integrity checks or encryption guarantees. User emails, authentication tokens, session identifiers, and other PII should never be propagated via baggage unless the entire communication path is secured end-to-end with TLS and you have verified that intermediate systems do not log headers.

Best practices for using baggage:

- Keep it minimal: only propagate what downstream services actually need
- Use short keys and values: every byte compounds across the trace
- Never include PII or secrets: treat headers as potentially exposed data
- Set it at the edge: establish baggage when the request enters your system, do not add incrementally at each hop

- Clear it when done: if baggage is only needed for a portion of the trace, remove it explicitly

Baggage entries are not automatically included as span attributes. Applications must explicitly read baggage and set corresponding attributes on spans or logs where they want the data recorded. This separation is deliberate: it prevents accidental leakage of propagated data into storage while giving applications control over what gets persisted. A common pattern is to set tenant-id in baggage at the API gateway, then have each service read that baggage entry and add it as an attribute to its spans for filtering and grouping in the backend.

The distinction between baggage and attributes is important:

- Baggage propagates across services via headers but must be explicitly recorded into telemetry
- Attributes are attached directly to a span or log record and do not propagate
- Baggage has network overhead; attributes have storage overhead
- Baggage is visible in transit; attributes are only visible in storage

Use baggage for data that needs to travel with the request. Use attributes for data that belongs only to a specific operation. Understanding this distinction prevents both missing context where it is needed and leaking sensitive data where it should not go.

Chapter 3: Metrics That Matter

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Metric Types and Aggregation Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Cardinality: The Silent Killer

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Histograms, Summaries, and Latency Measurement

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Designing SLOs and SLIs with OpenTelemetry Metrics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Exporting Metrics to Backends

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Chapter 4: Logs as First-Class Telemetry

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Structured Logging and OpenTelemetry

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Correlating Logs with Traces and Metrics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Log Levels and Severity Mapping

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

High-Volume Log Collection Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

PII and Sensitive Data in Logs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Chapter 5: Instrumentation Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Auto-Instrumentation: How It Works and When to Use It

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Manual Instrumentation Patterns and Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Hybrid Instrumentation for Complex Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Language and Framework Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Migrating from Legacy Instrumentation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Chapter 6: The OpenTelemetry Collector: Architecture Deep Dive

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Collector Architecture and Component Model

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Receivers and Protocols

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Processors: Transformation and Enrichment

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Exporters and Backend Integration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Connectors and Cross-Signal Pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Configuration Patterns and Validation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Chapter 7: Deployment Patterns for Production

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Agent vs Gateway vs Hybrid Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Kubernetes Deployment with DaemonSets and Deployments

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Sidecar Patterns in Service Mesh Environments

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Multi-Cluster and Multi-Region Topologies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

High Availability and Disaster Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Chapter 8: Performance, Scaling, and Capacity Planning

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Understanding Telemetry Volume at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Batching and Queue Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Backpressure Handling and Flow Control

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Load Balancing Collector Deployments

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Capacity Planning and Cost Management

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Diagnosing Performance Bottlenecks

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Chapter 9: Security, Privacy, and Governance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Transport Security with TLS and mTLS

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Authentication and Authorization Models

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Handling PII and Sensitive Data

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Multi-Tenancy and Isolation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Governance and Policy Enforcement

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Chapter 10: Integration with Observability Backends

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Vendor-Neutral Backend Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Integrating with Open Source Backends

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Commercial Backend Integration Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Multi-Backend Export Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Designing Your Observability Stack

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Chapter 11: Operations, Monitoring, and Troubleshooting

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Monitoring the OpenTelemetry Infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Debugging Missing or Malformed Telemetry

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Collector Health Checks and Diagnostics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Upgrade Strategies and Version Management

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Operational Runbooks for Common Issues

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Chapter 12: Advanced Patterns and Future Directions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Custom Instrumentation and Extensions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Dynamic Configuration and Service Discovery

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Observability Pipelines and Data Quality

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Advanced APM Patterns with OpenTelemetry

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

The Future of OpenTelemetry and Observability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Conclusion: Building an Observability Culture

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

From Tooling to Capability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Measuring the Value of OpenTelemetry

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Continuous Improvement in Observability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

References and Further Reading

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Official OpenTelemetry Specifications and Documentation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Project Governance and Community

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Collector Components and Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Language SDKs and Instrumentation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Deployment and Kubernetes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Backend Integration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Security and Privacy

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.

Community Resources and Articles

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/opentelemetryinproduction>.