Sponsored by chronosphere
A PALO ALTO NETWORKS COMPANY

Google Cloud

Cloud Observability IN ACTION

Selected Chapters

Michael Hausenblas

MANNING

chronosphere
A PALO ALTO NETWORKS COMPANY

Google Cloud

Align Observability Cost to Business Value and Guide Resolution at Scale

Chronosphere, a Palo Alto Networks company, is the observability platform built for control in the modern, containerized world. Chronosphere empowers customers to focus on the data and insights that matter by reducing data complexity, optimizing costs, and remediating issues faster. Its Observability Platform reduces data volumes and associated costs by 84% on average while saving developers thousands of hours. Recognized as a leader by major analyst firms, Chronosphere is trusted by the world's most innovative brands.

DOORDASH

Reduced spend by $48.7M over three years

affirm

Saves 14,000 developer hours annually

Zillow

Achieved 85% data volume reduction

Want to learn more?

Cloud Observability in Action

SELECTED CHAPTERS

MICHAEL HAUSENBLAS

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com


Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor:	Ian Hough
Technical editor:	Jamie Riedesel
Review editor:	Aleks Dragosavljević
Production editor:	Deirdre Blanchfield-Hiam
Copy editor:	Christian Berk
Proofreader:	Katie Tennant
Technical proofreader:	Ernest Gabriel Bossi Carranza
Typesetter:	Dennis Dalinnik
Cover designer:	Marija Tudor

ISBN: 9781633433472
Printed in the United States of America

contents

about the author vii

1 Agents and instrumentation 1

1.1 Log routers 2

Fluentd and Fluent Bit 3 ▪ Other log routers 8

1.2 Metrics collection 9

Prometheus 9 ▪ Other metrics agents 11

1.3 OpenTelemetry 12

Instrumentation 14 ▪ Collector 16

1.4 Other agents 25

1.5 Selecting an agent 26

Security for and of the agent 26 ▪ Agent performance and resource usage 27 ▪ Agent nonfunctional requirements 28

2 Backend destinations 30

2.1 Backend destination terminology 31

2.2 Backend destinations for logs 32

Cloud providers 32 ▪ Open source log backends 34
Commercial offerings for log backends 35

2.3 Backend destinations for metrics 36

Cloud providers 38 ▪ Open source metrics backends 39
Commercial offerings for metrics backends 41

2.4 Backend destinations for traces 41

Cloud providers 41 ▪ Open source traces backends 42
Commercial offerings for trace backends 43

2.5 Columnar data stores 43

2.6 Selecting backend destinations 49

Costs 49 ▪ Open standards 51 ▪ Back pressure 51
Cardinality and queries 51

3 Cloud operations 55

3.1 Incident management 56

Health and performance monitoring 56 ▪ Handling the incident 58 ▪ Learning from the incident after the fact 59

3.2 Alerting 59

Prometheus alerting 59 ▪ Using Grafana for alerting 67
Cloud providers 67

3.3 Usage tracking 68

Users 68 ▪ Costs 69

about the author

Michael Hausenblas works in the Amazon Web Services (AWS) open source observability service team, where he leads the OpenTelemetry activities. He has more than 20 years of experience in data engineering and cloud-native systems. Before AWS, Michael worked at Red Hat on Kubernetes, Mesosphere (now D2iQ) on Mesos and Kubernetes, MapR (now part of HPE) as chief data engineer, and spent more than a decade in applied research in the symbolic AI space.

1 Agents and instrumentation

This chapter covers

- What we mean by agents and instrumentation
- What types of agents exist, for what signal types
- What OpenTelemetry is and how you can benefit from it
- Criteria for selecting an agent

Observability is all about getting actionable insights from the signals the systems under observation expose: without the agents and instrumentation, there would be little to observe. In this chapter, we will learn how to instrument code (and automate that task) as well as select and deploy agents that collect, aggregate, filter, downsample, redact, and route logs, metrics, and traces.

The telemetry industry is, at the time of writing, in the midst of a tectonic shift. This transformation from vendor-specific or signal-specific instrumentation and agents to an industry standard called OpenTelemetry started in 2019. In the context of this book, we consider vendor-specific as well as signal-specific agents as traditional agents, in contrast to OpenTelemetry.

Cloud providers, from AWS to Azure to Google Cloud, as well as observability vendors, from Datadog to Splunk to New Relic to Lightstep to Dynatrace to

Honeycomb to Grafana Labs, have decided to assemble behind OpenTelemetry. This effectively means the telemetry industry has decided to make telemetry table stakes, to commoditize it, and, instead, compete on the destinations (storage and query as well as frontends).

OpenTelemetry covers, in order of maturity and at the time of writing, traces, metrics, and logs. Going forward, the community has decided to also cover profiles (resource usage and timing within a process) in OpenTelemetry.

Since logs are still a work in progress in 2022, I will use vendor-specific or signal-specific setups to demonstrate logs in this chapter. For metrics and traces, on the other hand, I will show you how to benefit from the open and vendor-agnostic standard that OpenTelemetry and its agent, the collector, is.

1.1 Log routers

First, we will have a look at log routers. These agents allow you to collect logs from various sources, be that on a system level or your applications, and send them to a central place for lookup. In this context, you often hear also about log forwarding, log collection, and log aggregation. However, to keep things simple, we will gloss over the details and lump all these together as *log routers*.

At time of writing (2023), vendor-specific agents, such as Splunk's Universal Forwarder or the Datadog agent, as well as signal-specific agents, including Fluentd/Fluent Bit or the widely used Logstash. Expect that to change and move to OpenTelemetry, going forward.

We will differentiate between system logs and application logs. System logs come from, but are not limited to, the following sources:

- The operating system (e.g., in Linux, from `/var/log` and `journalctl`)
- The Kubernetes control plane (http://mng.bz/BmWv) logs (including from the API server, etcd, and controller manager) and the worker node logs (http://mng.bz/d10D), such as from the kubelet or the container runtime
- User activity and API usage logs, like the ones provided by AWS CloudTrail (https://aws.amazon.com/cloudtrail/)
- Infrastructure automation logs (e.g., from HashiCorp Terraform or AWS CloudFormation)

Application logs, on the other hand, are what you emit in your code. You may end up using different agents for system logs and application logs; however, if possible, you should try to standardize by choosing one, for example, from the ones discussed in the following sections. This is mostly for operational reasons, as it means there will be less to patch (think of the Log4j challenges, for example) and upgrade.

NOTE In case you're not familiar with the Log4j challenges I mention here (or maybe you've forgotten about them), CVE-2021-44228 (https://nvd.nist.gov/vuln/detail/CVE-2021-44228), informally known as Log4Shell, was a

zero-day vulnerability in Log4j, discovered in November 2021. It caused a lot of work (to apply the security patches) in a lot of places, from internal IT to cloud providers, and serves as an example of how much operational impact such a thing can have.

While logs are stable from an API and model perspective, in OpenTelemetry it may still take until mid-2024 until they are supported throughout in the SDKs and distributions. This means you will need a temporary strategy until you can, eventually, migrate to OpenTelemetry.

OK, that's enough theory. Let's see log routing in action. First up is a popular choice in the context of containers: Fluentd and Fluent Bit.

1.1.1 Fluentd and Fluent Bit

The CNCF projects Fluentd (https://www.fluentd.org/) and Fluent Bit (https://fluentbit.io/) are popular cloud-native agents that started out in the logging domain. Initially, there was only Fluentd (written in Ruby and C), offering a rich ecosystem (with over 1,000 plug-ins) but coming with a relatively big memory footprint (dozens of MB) for the agent.

In 2014, the folks behind Fluentd, Treasure Data, acknowledged the need for a lightweight log router specifically for constraint environments (containers, IoT). This insight led to the development of Fluent Bit (written in C) with a memory footprint of a couple of hundreds of KB and many fewer plug-ins than Fluentd (less than 100).

At time of writing, Fluent Bit is expanding its scope from logs-only to other signal types, such as metrics and traces. For details, have a look at the GitHub issue "Fluent Bit v2.0 Is Coming" (https://github.com/fluent/fluent-bit/discussions/5993). We will focus on the logging aspects, in this section, however, since there are still many WIP items in flight. Let's have a look at a basic usage example in a containerized setup and discuss some properties along the way.

To demonstrate log routing, we're using an OpenSearch with Docker Compose (https://opensearch.org/docs/latest/opensearch/install/docker/) setup. Don't worry about OpenSearch for now; we will cover it in detail in chapter 2. Just think of it as the place logs can be sent to and which you can use to look up, search for, and query log lines. For example, you may be interested in all `ERROR`-level logs lines from a specific microservice, within a certain time range (say, the last 24 hours).

In figure 1.1, you can see our example logging pipeline—we're using `hoily` (http://mng.bz/rWxg) as the log signal source running in a Docker container. The container is exposing port 8765 for its HTTP API and uses the Fluentd logging driver to send log lines to Fluent Bit (the main log router), ingesting log lines into OpenSearch as the log backend destination, which you then can query and visualize in OpenSearch Dashboards.

Moving on to the implementation, we're once again using Docker Compose to stand up the pipeline, which is defined in the ch04/fluent-bit/docker-compose.yaml file. Let us review the pipeline step by step. I've configured the `hoily` container in the

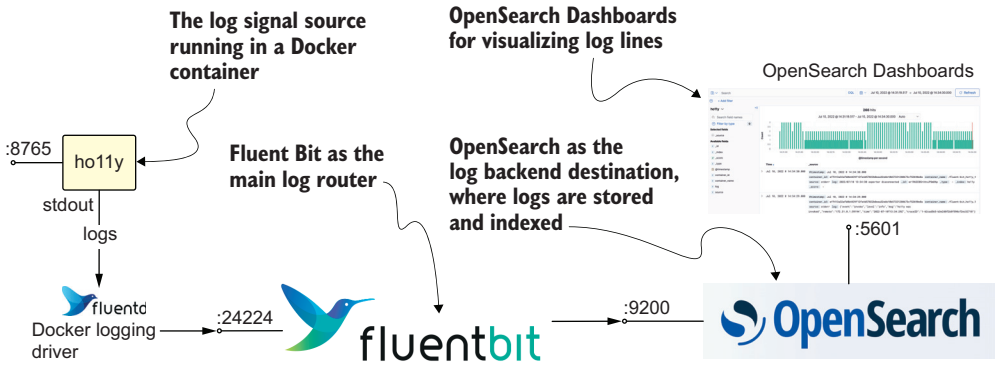


Figure 1.1 Example logging pipeline using Fluentd and Fluent Bit

Docker Compose file in a way that it is using the `fluentd` logging driver (since using Fluent Bit as the driver is not directly support there, yet).

Listing 1.1 Docker Compose configuration snippet for the signal source `ho11y`

```
ho11y:
    image: public.ecr.aws/mhausenblas/ho11y:stable
    ports:
      - 8765:8765
    logging:
      driver: fluentd
      options:
        tag: ho11y
        fluentd-address: localhost:8765
    depends_on:
      - fluent-bit
```

Exposes the ho11y HTTP API service port on port 8765 to make it accessible from the host machine (so we can invoke it with curl)

Configures the container to use the Fluentd logging driver

Defines that the ho11y container depends on the Fluent Bit log router, forcing the start-up sequence order

Next up is the Fluent Bit configuration, as seen in the following listing. As you can see, this takes, as one parameter, a file called fluent-bit.conf, which we will discuss later in this chapter.

Listing 1.2 Docker Compose configuration snippet for Fluent Bit

```
fluent-bit:
    image: fluent/fluent-bit:1.9.8
    volumes:
      - ./fluent-bit.conf:/fluent-bit/etc/fluent-bit.conf
    ports:
      - "24224:24224"
      - "24224:24224/udp"
```

Defines where the Fluent Bit configuration (routing info) can be found

Makes sure we open port 24224 so that we can listen on incoming log lines from the Fluentd logging driver in Docker

Last but not least, let's see how the logs' backend OpenSearch is configured (in the Docker Compose file in the following listing) in a single-node setup along with the frontend/UI, OpenSearch Dashboards.

Listing 1.3 Docker Compose configuration snippet for OpenSearch (and Dashboards)

```
opensearch-node1:
    image: opensearchproject/opensearch:2.1.0
    container_name: opensearch-node1
    environment:
      - cluster.name=opensearch-cluster
      - node.name=opensearch-node1
      - bootstrap.memory_lock=true
      - "OPENSEARCH_JAVA_OPTS=-Xms512m -Xmx512m"
      - "DISABLE_INSTALL_DEMO_CONFIG=true"
      - "DISABLE_SECURITY_PLUGIN=true"
      - "discovery.type=single-node"
    volumes:
      - opensearch-data1:/usr/share/opensearch/data
    ports:
      - 9200:9200
  opensearch-dashboards:
    image: opensearchproject/opensearch-dashboards:2.1.0
    container_name: opensearch-dashboards
    ports:
      - 5601:5601
    environment:
      - 'OPENSEARCH_HOSTS=["http://opensearch-node1:9200"]'
      - "DISABLE_SECURITY_DASHBOARDS_PLUGIN=true"
```

Exposes port 9200 so that we can ingest log lines (from Fluent Bit)

Exposes port 5601 to make OpenSearch Dashboards (frontend/UI) accessible from the host machine

Configures OpenSearch Dashboards (frontend) to use OpenSearch (backend for logs)

At this point, we've effectively implemented the logging pipeline shown in figure 1.1. Now, let's have a closer look at the Fluent Bit configuration itself. As depicted in listing 1.4, I'm using the Forward (https://docs.fluentbit.io/manual/pipeline/inputs/forward) input plug-in in Fluent Bit, taking in the log stream from Fluentd (from the Docker runtime) and, via the OpenSearch output section, configuring Fluent Bit to route the log lines to OpenSearch.

Listing 1.4 Fluent Bit configuration

Here, we're configuring the Forward plug-in capable of receiving log items from Fluentd (from the Docker log driver).

The Forward plug-in will listen on all network interfaces (that is, 0.0.0.0).

The default port for the Forward plug-in (that Fluentd connects to)

```
[INPUT]
    Name forward
    Listen 0.0.0.0
    Port 24224
```

```
[OUTPUT]
    Name opensearch
    Match **
    Host opensearch-node1
    Port 9200
    Index ho11y
    Suppress_Type_Name on
```

The destination configuration—in our case, the OpenSearch plug-in as the backend (downstream) for the log items

Here, we define that we, indeed, take any log item (we could be more selective here regarding which ones we're routing to OpenSearch, but for now, we use all of them).

The name of the index we want to create in OpenSearch

Tells the OpenSearch node that we don't have to set a type (since we don't care, in the context of this setup)

The hostname (which could also be a raw IP address) of the OpenSearch node

Phew, that was a lot of configuration! Now, it's time to launch the pipeline and generate some log lines. First off, we need to launch the Fluent Bit–OpenSearch combo:

```
docker-compose -f docker-compose.yaml up
```

In a different terminal session, launch a simple load generator (in Bash shells) like so:

```
while true ; do \
  curl -s -o /dev/null -w "%{http_code}"
  ➥ localhost:8765 ; \
  sleep 1 ; \
done
```

Uses curl to invoke ho11y and generate log lines, discarding the output and only showing the HTTP status code returned

Pauses for 1 second and then repeats from the top

Now, you can navigate to http://localhost:5601/. You should see the Welcome page of OpenSearch Dashboards (in the ELK stack—Elasticsearch, Logstash, and Kibana—the equivalent component would be Kibana). Click on the burger icon in the upper-left corner (the three lines), and select Stack Management followed by Index Patterns; you should see something like what is shown in figure 1.2.

Next, click on Create Index Pattern, and in the input box that's labeled Index Pattern Name, enter `ho11y` and then click Next Step. Then, in the Time Field dropdown list, select `@timestamp` followed by clicking the Create Index Pattern button. The result you should see is something like what is depicted in figure 1.3.

Now, head back to the OpenSearch Dashboards home page, click on the burger icon in the upper-left corner (the three lines) and then click Discover. You should then see something akin to what is shown in figure 1.4.

You now have a functioning logging pipeline based on Fluent Bit as a log router at your disposal. From this point on, feel free to drill into log lines or use the Dashboards Query Language (DQL; https://opensearch.org/docs/latest/dashboards/dql), a domain-specific query language, to find data in OpenSearch indexes, to query for specific terms in the `ho11y` logs.

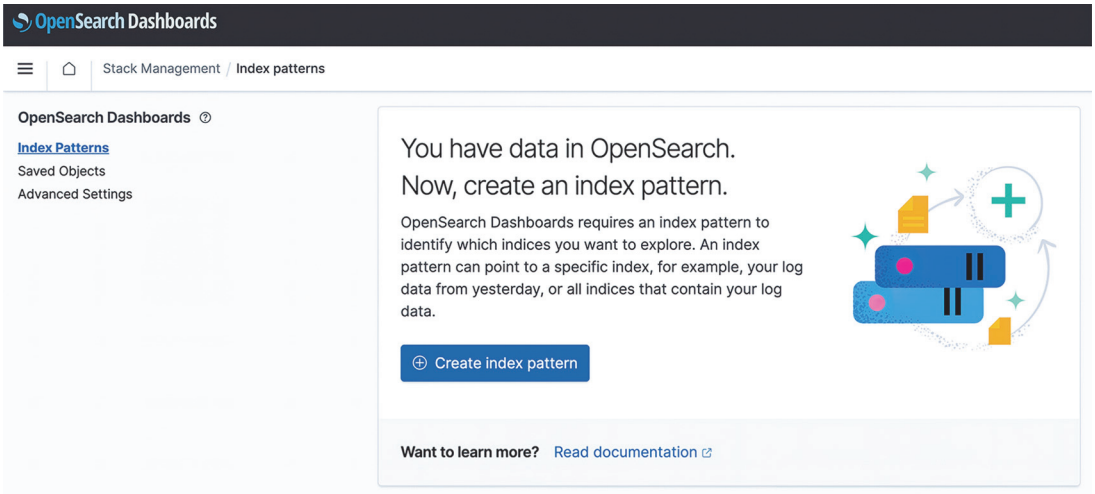


Figure 1.2 Creating an index pattern in OpenSearch Dashboards

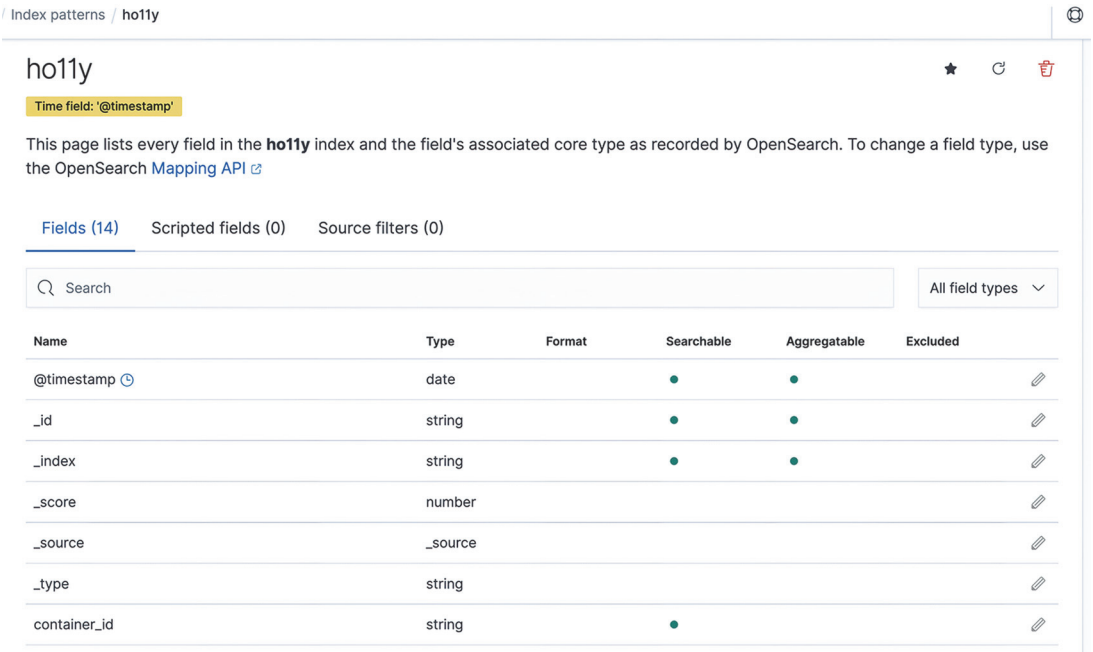


Figure 1.3 The resulting index pattern in OpenSearch Dashboards

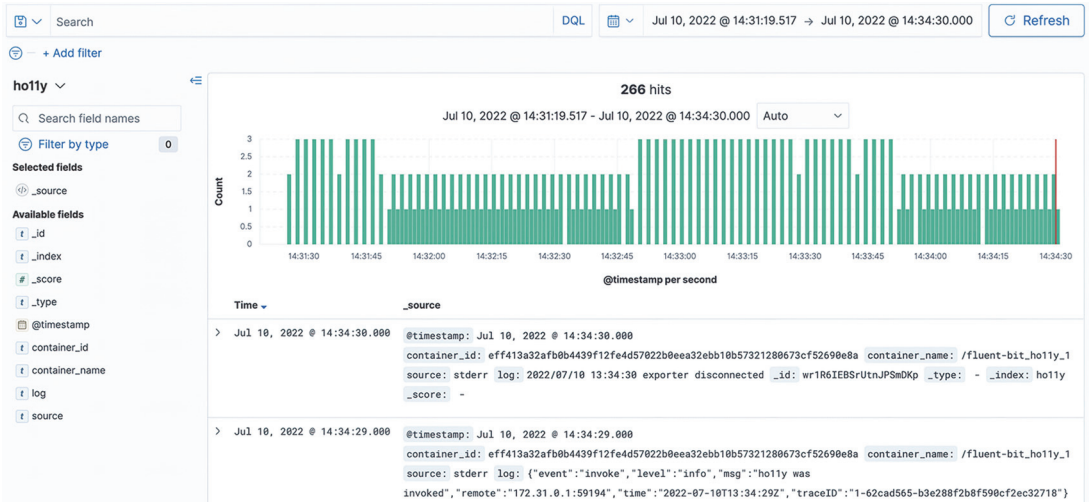


Figure 1.4 OpenSearch Dashboard's Discover view, with our `ho11y` index selected

If you're interested in understanding how exactly Fluentd and Fluent Bit work in greater detail and best practices for building log routing pipelines in the context of containers (especially Kubernetes), I highly recommend studying the wonderful Manning book *Logging in Action* by Phil Wilkins (2022; https://www.manning.com/books/logging-in-action).

1.1.2 Other log routers

We have seen Fluent Bit in action, but this is not our only option. There are, indeed, a number of both commercial and open source log routers available, such as these:

- The Beats (https://github.com/elastic/beats/tree/master) family as well as Logstash (https://github.com/elastic/logstash), part of the Elastic Stack.
- Data Prepper (https://github.com/opensearch-project/data-prepper), an AWS open source project that allows for stateful processing.
- syslog-ng (https://www.syslog-ng.com/products/open-source-log-management/) is a syslog implementation that ingests messages from systemd journal by default, applies filters, and sends them to files.
- rsyslog (https://www.rsyslog.com/) is an open source tool for routing and filtering log messages over an TCP/IP network, introduced as a replacement for syslogd.
- Graylog (https://www.graylog.org/) is a log management software company that provides logging solutions, with a focus on security use cases.
- Cribl (https://cribl.io/) is a cloud offering with powerful ingest and process capabilities.

Remember, as OpenTelemetry matures and logs continue stabilizing, make sure you have a migration strategy in place to move off of whatever log routing agent you are currently using with the OpenTelemetry collector. Let's now move on to metrics.

1.2 Metrics collection

In contrast to logs, the landscape is less fragmented concerning metrics. Over the past years (effectively, with the rise of Kubernetes, since 2016), the dominant player is the Prometheus project, which put forward both the way metrics are collected as well as the way metrics are represented on the wire—the *exposition format*. Let's first have a look at Prometheus and then take a quick peek at other established agents in the metrics space.

> **NOTE** While Prometheus established pull-based collection—that is, the agent queries the metrics source via an HTTP endpoint defaulting to `/metrics`—the situation is more nuanced in the context of OpenTelemetry. There, the ingress part is still pull based; however, the egress part (toward the backend destination), is typically push based. We will get into the details in the next section on OpenTelemetry.

1.2.1 Prometheus

Prometheus (https://prometheus.io/) is a graduated CNCF project that grew up in tandem with Kubernetes, where it also is predominantly used. It covers the ingestion, storing, and query and federation interfaces for metrics, including a simple UI. Further, part of the Prometheus project is alerting capabilities (a separate binary but tightly integrated). Let's have a closer look at how Prometheus does things, in greater detail.

A time series in Prometheus is uniquely identified by its metric name and a number of (optional) key–value pairs referred to as *labels*. A metric name captures the general feature you want to measure. For example, `api_requests_total` could be used for the total number of API invocations. Labels, on the other hand, represent Prometheus's dimensional data model. Let's say you want to capture two dimensions with the API invocation metric: the HTTP method used and what the URL path was. You could use the two labels `method` and `path` for this, resulting in the following, for example:

```
api_requests_total{method="GET", path="/some/interesting/thing"}
```

There are different types of metrics Prometheus supports, including counters, gauges, and histograms, and while Prometheus has naming conventions for metrics and labels (https://prometheus.io/docs/practices/naming/), it doesn't define or enforce semantics for them. The lack of defined semantics in Prometheus is, in practice, not a huge problem, since Prometheus offers a feature called *relabeling* (https://prometheus.io/docs/prometheus/latest/configuration/configuration/#relabel_config), which allows you to standardize on naming.

With relabeling, you can add, remove, or modify labels on a per-target basis. A concrete example would be to rename a label to standardize on a label name. Let's assume our source would expose the URL path under the label `handler`, but we want to use `path` in our setup. In this case, we want to do the following relabeling:

```
api_requests_total{method="GET", handler="/some/interesting/thing"}

=>

api_requests_total{method="GET", path="/some/interesting/thing"}
```

You can do this with a relabeling rule that you would define in your Prometheus scrape config (https://prometheus.io/blog/2015/06/01/advanced-service-discovery/#scrape-configurations-and-relabeling), like the following:

```
- source_labels: [handler]
  action: replace
  target_label: path
- regex: "handler"
  action: labeldrop
```

Matches the handler label name and adds the path label with the same label value

Once done, it drops the handler label name, leaving path as the one label capturing the URL path.

Tip

If you want to try out relabeling interactively, you can visit https://relabeler.promlabs.com/ and use the relabel rule (in the left-hand Relabeling Rules input box) and the following as an input for the right-hand side (Object Labels):

```
method: "GET"
handler: "/some/interesting/thing"
```

When you now click the Analyze Rules button, you should see the rule engine in action, yielding the expected result.

Further, to improve query performance, you can define recording rules (http://mng.bz/V1vX) in your Prometheus scrape config. This feature allows us to precompute frequently used or expensive expressions, effectively trading space for time, saving the result as a new time series. Let's assume we want to use a recording rule for capturing the server-side error rate of our API invocation example (with `api_requests_total` being a counter metrics type):

The recording rule name

```
- name: api_invocation_server_error
  interval: 5m
  rules:
  - record: http_status_5xx:api_requests_total:ratio_rate_5m
    expr:
      sum(rate(api_requests_total{code=~"5.."}[5m]))
        /
      sum(rate(api_requests_total[5m]))
```

How often to evaluate the rule (here, every 5 minutes)

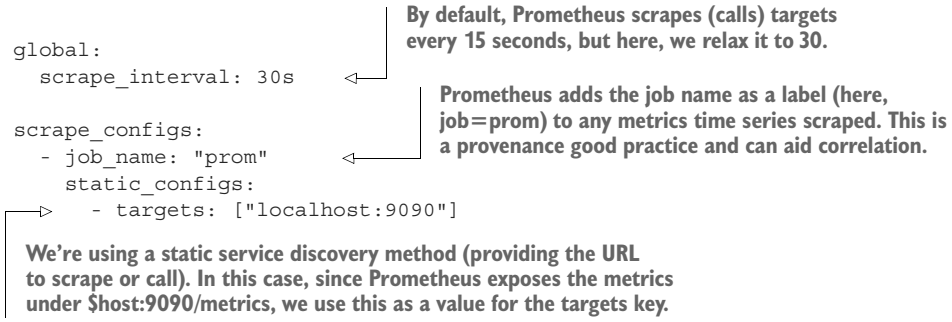
The name of the resulting metric (note the usage of :, which is, by convention, reserved for recording rule metrics)

The PromQL expression to evaluate at the current time, resulting in a new time series with the metric name defined by the preceding record value—http_status_5xx:api_requests_total:ratio_rate_5m, in our case

You can use Prometheus in two ways (that is, you're using the same binary, with different configurations):

- Prometheus server (default mode)
- Prometheus agent mode (https://prometheus.io/blog/2021/11/16/agent/), optimized for scraping and remote read–write for federation

The core of Prometheus is its scrape config. This is where you define what Prometheus should scrape (its targets) and how it is supposed to find the targets (service discovery) as well as, optionally, recording and alerting rules. For example, to scrape the metrics that Prometheus itself exposes (i.e., self-scraping), you could use something like the following in your scrape config:




```
global:
  scrape_interval: 30s

scrape_configs:
  - job_name: "prom"
    static_configs:
      - targets: ["localhost:9090"]
```

Note that scrape configs continue to be important, going forward, since OpenTelemetry aims to be compatible with Prometheus and allows you to reuse your scrape configs (with minor adaptations). You will see all of this in action in the end-to-end example in this chapter.

If you want to learn more about how to use Prometheus as well as how it works internally, I strongly recommend the ultimate reference: *Prometheus: Up & Running*, by Julien Pivotto and Brian Brazil, *Second Edition* (O'Reilly; http://mng.bz/x458).

1.2.2 Other metrics agents

While Prometheus is the established metrics platform in the cloud-native ecosystem, especially in the Kubernetes context, there are other agents you may come across, for various reasons, including legacy usage or because your organization decided to go all in with a certain vendor or project. This includes, but is not limited to, the following agents (in alphabetical order):

- *Collectd (https://collectd.org/)*—Written in C, with this agent, you can collect (operating) system and application performance metrics efficiently. Several cloud providers and ISVs provide integrations with collectd, such as Amazon CloudWatch, Splunk, and Sumo Logic.
- *Grafana Agent (https://grafana.com/docs/agent/latest/)*—Created by Grafana Labs, this agent is optimized for Prometheus metrics. The code for this agent was donated to the Prometheus project and forms the basis of the agent mode in

Prometheus, there. In agent mode, Prometheus deactivates several functions, including the UI, alerting, and local storage.

- *Graphite (https://grafana.com/oss/graphite/)*—Now also under the Grafana Labs umbrella, this agent represents an example of the telemetry of the previous generation of machine-centric monitoring.
- *Nagios (https://www.nagios.org/)*—This is a venerable open source system (including agent) for monitoring machines and networks.
- *StatsD (https://github.com/statsd/statsd)*—Developed by Etsy, this agent is a Node.js tool that has a client–server model, allowing you to collect and aggregate application metrics and send the metrics to backends, such as Graphite.
- *Telegraf (https://github.com/influxdata/telegraf)*—From InfluxData, this is a rich and powerful agent with many plug-ins, originally written for use with their InfluxDB time series open source datastore and now also supporting OpenTelemetry.

This section is intentionally brief, since I think it's important to be aware of other agents in this space; however, in practice, you can expect that in the majority of cases, you will be dealing with Prometheus, in one form or another. If you come across a legacy environment where one of the preceding agents is used, you want to make sure that there is a clear path to migrate to Prometheus or OpenTelemetry in its place.

1.3 OpenTelemetry

OpenTelemetry is an incubating cloud-native computing foundation project that started out as a merger of two projects, OpenCensus and OpenTracing, that were focused on distributed traces. OpenTelemetry expanded its scope, starting in 2019, to metrics and logs. The project is the second-largest CNCF project after Kubernetes, and in its core, it is a collection of APIs, SDKs, and tools you can use to instrument, collect, process, and ingest telemetry signals.

It provides a vendor-agnostic model for emitting traces, metrics, logs (and, going forward, also profiles [https://github.com/open-telemetry/oteps/issues/139]) with both backward compatibility (via receivers and exporters) and native functions via a dedicated protocol. OpenTelemetry includes auto-instrumentation support for some popular program languages and runtimes, such as Java/JVM, .NET, JavaScript/Node.js, and Python. If you're interested in the current status of the many components OpenTelemetry has, you can

- Check out the high-level project status (https://opentelemetry.io/status/) page.
- View fine-grained information, since the project keeps track of the detailed compliance of the implementations with the specification (http://mng.bz/Aojp) as well.

Note that in addition to the specifications (APIs and data model), OpenTelemetry comes with a set of semantics, such as the standard names for system/runtime metric instruments (http://mng.bz/Zqmm).

In figure 1.5, you can see the OpenTelemetry concept of signal collection, with a strong focus on correlation. Starting from the bottom and moving to the top, the figure shows the flow of the signals from applications and infrastructure, through the OpenTelemetry collector, eventually ingested into backend destinations.

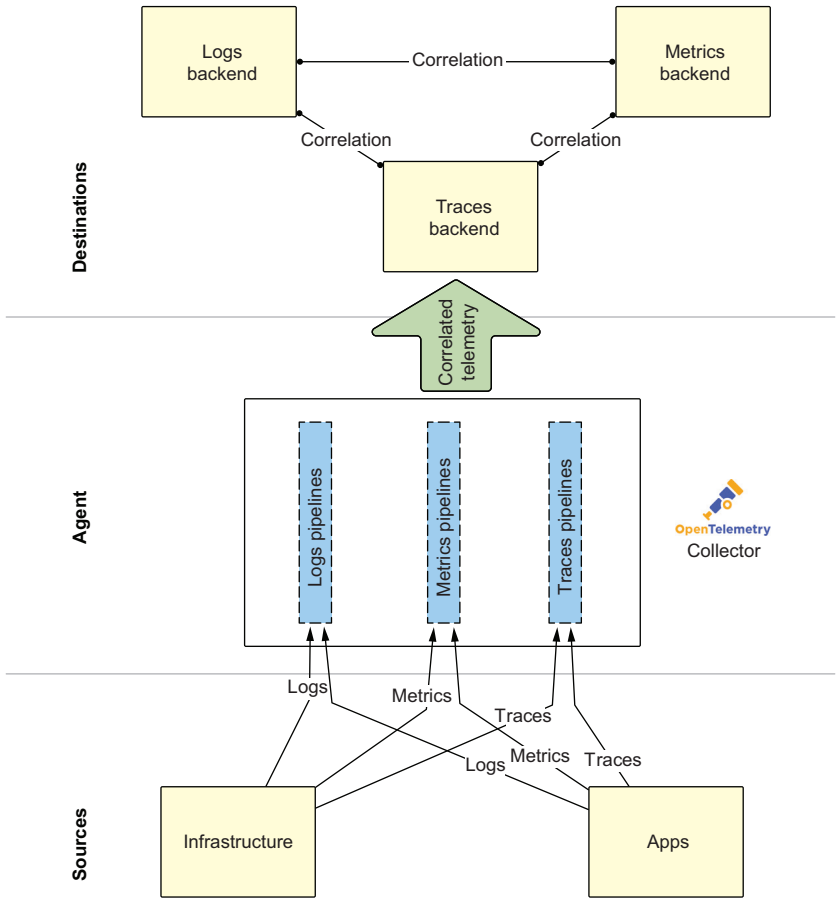


Figure 1.5 Concept of OpenTelemetry collection (from upstream docs; https://opentelemetry.io/docs/reference/specification/logs/overview/)

In the collector, the various signal types (logs, metrics, and traces, currently) are processed in so-called pipelines, dedicated to a signal type. We will discuss this agent's inner workings in greater detail a little later this chapter, but let's first dive into the application code side of the OpenTelemetry house: the instrumentation.

1.3.1 Instrumentation

In the context of the code you own, the first thing you want to tackle is emitting signals, using OpenTelemetry. OpenTelemetry provides instrumentation support for 11 programming languages at the time of writing. More specifically, OpenTelemetry supports C++, .NET, Erlang/Elixir, Go, Java, JavaScript, PHP, Python, Ruby, Rust, and Swift.

In OpenTelemetry, instrumentation is organized such that users get a library software development kit for each programming language and, potentially, support for automatic instrumentation. Some languages offer this via separate repos (e.g., Java), while others, like Ruby, maintain a single repo that supports both manual and auto-instrumentation. Remember that instrumentation doesn't come for free. Even in the best case of auto-instrumentation, where you don't have to change your code, there will be a small impact on your app's performance (http://mng.bz/Rx5j).

Let's have a closer look at the different instrumentation methods in OpenTelemetry. We will start with automatic instrumentation, since it's a great starting point and requires less effort.

AUTOMATIC INSTRUMENTATION

With automatic instrumentation, or simply auto-instrumentation, in OpenTelemetry, the steps required are as follows:

1. First, enable auto-instrumentation. You typically need to add one or more dependencies. At a minimum, these dependencies will add OpenTelemetry API and SDK capabilities.
2. Then, configure the auto-instrumentation, either via environment variables (and/or language-specific mechanisms, such as system properties in Java). You must, at the very least, provide a service name to identify the service you're instrumenting and, optionally, other configuration options, including things such as source configuration, exporter configuration, and resource configuration.

OpenTelemetry (at the time of writing) provides auto-instrumentation for generating traces for certain programming languages, including

- Java auto-instrumentation (https://github.com/open-telemetry/opentelemetry-java-instrumentation), which is part of OpenTelemetry, in the form of a JAR file you include (using `-javaagent`). You can configure this agent in a fine-grained manner (http://mng.bz/2DQ8), from sanitizing database queries to capturing predefined HTTP headers, which will dynamically inject JVM bytecode to capture signals from popular Java frameworks.
- JavaScript (https://github.com/open-telemetry/opentelemetry-js), also officially supported via the OpenTelemetry core, also allows yout to use all of your JavaScript apps. Note there are additional, framework-specific SDKs, such as the OpenTelemetry Node SDK (http://mng.bz/1q41), that enable auto-instrumentation.

- Python (http://mng.bz/Pz2Y) is also in the officially supported languages for auto-instrumentation.
- .NET (https://opentelemetry.io/docs/instrumentation/net/automatic/) supports auto-instrumentation as a .NET profiler to inject additional instrumentations at runtime, using monkey patching.
- Go (https://github.com/keyval-dev/opentelemetry-go-instrumentation) is a user contribution, based on eBPF. Strictly speaking, eBPF is a Linux syscall and Virtual Machine (VM), with certain requirements regarding the kernel version and symbol support, allowing you to push down user space functionality into the Linux kernel, triggering it based on events. In this context, it means to automatically capture telemetry signals for you.

Going forward, you can expect auto-instrumentation support for more programming languages as well as supporting metrics. Auto-instrumentation is a good baseline to always consider, since you get some insights essentially for free. However, it has its limitations: auto-instrumentation treats the service as a black box, meaning you won't get business-logic-related signals out of it. For this, you'd need to use manual instrumentation, so let's have look at it.

Manual instrumentation

Manual instrumentation means that you modify your own code to manually emit signals. The steps required to do so, language agnostically, are

1. Import OpenTelemetry in your code—that is, take on a dependency on the language-specific API and the SDK.
2. For traces and metrics, first, create a provider (usually, there are defaults implemented), which, in turn, allows you to create an instance (along with a name and version).
3. For services (in contrast to a library you may instrument), in addition, configure the SDK, with appropriate options for exporting your signals.
4. After all the setup is complete, focus on the main job: create events (both traces and metric).

> **NOTE** Logs (http://mng.bz/JgrK) don't follow the previously shown pattern. Instead, they leverage existing setups; in other words, while OpenTelemetry takes a clean-slate approach for metrics and traces (via specifying new APIs and providing full implementations of said APIs across the programming languages), the approach with logs is different, in that the focus is on supporting existing logs and logging libraries.

There are many possible use cases and combinations of manual and auto-instrumentation, and if you're, indeed, looking for hands-on guidance on how to use this in the context of your setup, you will find a wide array of excellent articles and tutorials on the web. For example, "OpenTelemetry and Python: A Complete Instrumentation Guide"

(http://mng.bz/wvK2), "Auto-Instrumenting Node.js Apps With OpenTelemetry" (http://mng.bz/qrpJ), "Trace Context Propagation With OpenTelemetry" (http://mng.bz/7DAe), and "OpenTelemetry in Java: Tutorial & Agent Example" (https://www.containiq.com/post/opentelemetry-in-java) are all excellent resources. Next up, we have a look at the OpenTelemetry collector—the "universal signal type" agent or "universal telemetry" agent.

1.3.2 Collector

OpenTelemetry's agent is called the *collector*. It comprises pipelines that assemble collector components, such as receivers, processors, and exporters, as depicted in figure 1.6. Think of a pipeline as a combination of components that define how and where you get the signals from (receivers), process them at the edge (transform, batch, etc., via processors and connectors), and into which backend to ingest the signals (exporters).

You can define one or more pipelines in a collector. Each pipeline is dedicated to a signal type (e.g., metrics). You can also have multiple pipelines for the same signal type, such as different environments or to enforce different policies. Each of the pipelines acts independently, collecting signals from the sources and ingesting them into backends.

The native way to talk to the collector is via the OpenTelemetry Protocol (OTLP; https://tinyurl.com/mngbz), a general-purpose telemetry data-delivery protocol. The OTLP specification describes the encoding, transport, and delivery mechanisms of signals between the downstream sources, intermediate nodes (like collectors), and backend destinations.

OTLP itself was designed by the OpenTelemetry project and is an incremental improvement over the OpenCensus protocol, focusing on high reliability of delivery and clear visibility when the data cannot be delivered. In addition, OTLP is L7-load-balancer friendly, supporting the correct mapping of imbalanced incoming traffic to balanced outgoing traffic. Further, OTLP allows backpressure signaling from the destinations all the way back to sources, enabling you to build out reliable multihop delivery via intermediate nodes (e.g., OpenTelemetry collectors), each with potentially different processing capacities.

From a protocol perspective, OTLP uses a request–response style: the downstream clients send requests, and the upstream server replies with corresponding responses. OTLP is rather flexible—you can run it over gRPC, HTTP, and (going forward) WebSockets.

While in the future, we may live in a world where everyone and everything natively supports OpenTelemetry—that is, speaks OTLP—this is not yet the case at the time of writing, in 2023. The OpenTelemetry community found a clever way to help jump-start things: by introducing a vast array of ingress and egress components (receivers and exporters) that can read existing formats and protocols from sources as well as write existing formats and protocols to the destinations.

How does this work, in detail? Let's have a closer look at figure 1.6 now to see how you'd use the OpenTelemetry collector to define signal-type-specific pipelines to collect signals from downstream and ingest them upstream.

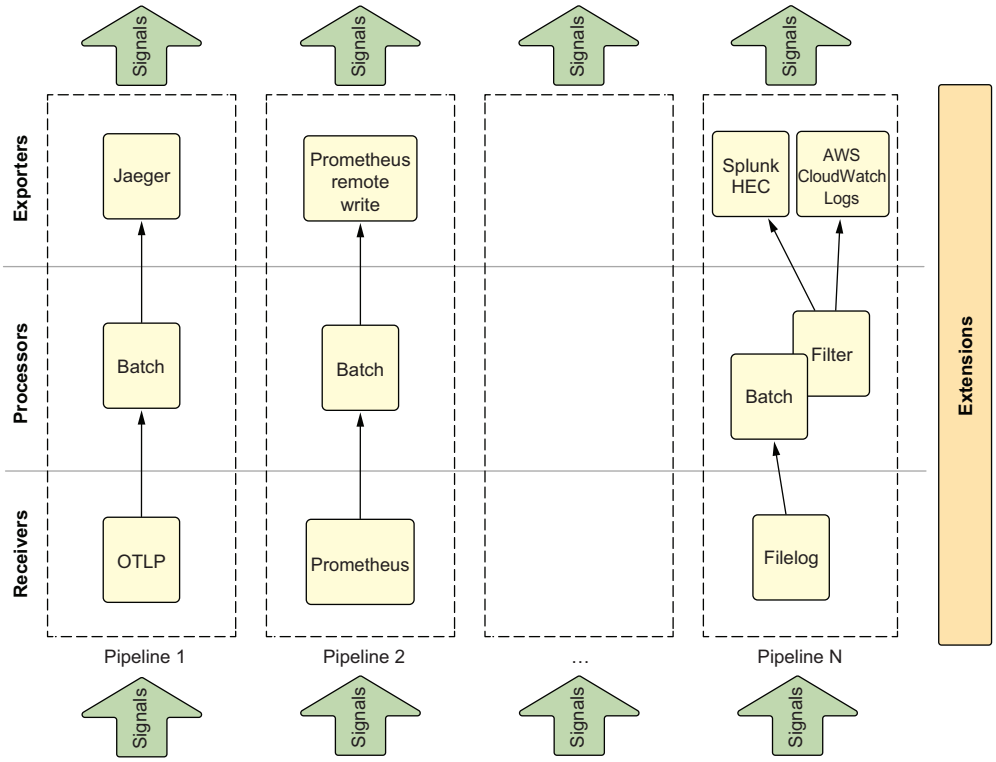


Figure 1.6 The OpenTelemetry collector and its components

Now, we will focus on the pipelines in the collector. Each pipeline uses one or more components:

- In the ingress path, collecting signals from the downstream sources, one or more receivers that can come from two places.
  - For the native OTLP, from the opentelemetry-collector (http://mng.bz/mVM4) repo
  - For all other receivers (Prometheus, Jaeger, Kafka, etc.), from the opentelemetry-collector-contrib (http://mng.bz/5wVa) repo
- The core manipulation component—one or more processors that can come from two places.
  - For a small number of generic processors, such as Batch, from the opentelemetry-collector (http://mng.bz/6DxA) repo

  - For all other processors (Filter, Transform, Tail Sampling, etc.), from the opentelemetry-collector-contrib (http://mng.bz/o1gp) repo
- In the egress path, ingesting signals to the upstream destinations, one or more exporters that can come from two places.
  - For the native OTLP (and the logging exporter), from the opentelemetry-collector (http://mng.bz/nWa2) repo
  - For all other exporters (Prometheus Remote Write, Elasticsearch, AWS X-Ray, etc.), from the opentelemetry-collector-contrib (http://mng.bz/vn04) repo

Let's have a high-level look at some example pipelines from figure 1.6:

- The `pipeline 1` is a traces pipeline that can collect traces via OTLP inbound, batch them up, and ingest them into a Jaeger instance.
- The `pipeline 2` is a metrics pipeline that scrapes metrics via the Prometheus receiver inbound, batches them up again, and uses the Prometheus remote write API to ingest them into, say, Thanos or Cortex.
- The `pipeline N` is a logs pipeline that collects log lines via the Filelog (http://mng.bz/rjJE) receiver inbound, then batches them again and filters them, and can then ingest the log lines into two different backend destinations, Splunk and CloudWatch, via the respective exporters.

In addition, the collector also offers several extensions (https://opentelemetry.io/docs/collector/configuration/#extensions) to operate the agent, including health checks, service discovery, and rich telemetry (e.g., logs, metrics in Prometheus format, traces in OTLP, and profiles in `pprof` format).

> **TIP** There are different ways to deploy (https://opentelemetry.io/docs/collector/deployment/) the collector, usually differentiated between close to the workload (agent mode) and in a central place (gateway mode). Running the collector in gateway mode (effectively, a number of collectors behind a load balancer) can help to enforce policies centrally and support separation of concerns around access control and authentication, including API keys and IAM roles.

Let's have a look at a concrete example that shows the OpenTelemetry collector in action, taking care of both metrics and traces collection and routing; our example setup looks like what is depicted in figure 1.7. We're again using `holly` as the signal generator, exposing metrics in Prometheus exposition format and emitting traces in OTLP format, with the OpenTelemetry collector configured to scrape the metrics and collect the traces and ingest them to respective backend destinations—Prometheus for metrics and Jaeger for traces.

We will now build the overall setup shown in figure 1.7, again using Docker Compose, so let's have a closer look at each of the components in turn. Listing 1.5 shows the sample app `holly`, generating both metrics and traces. Pay special attention to the

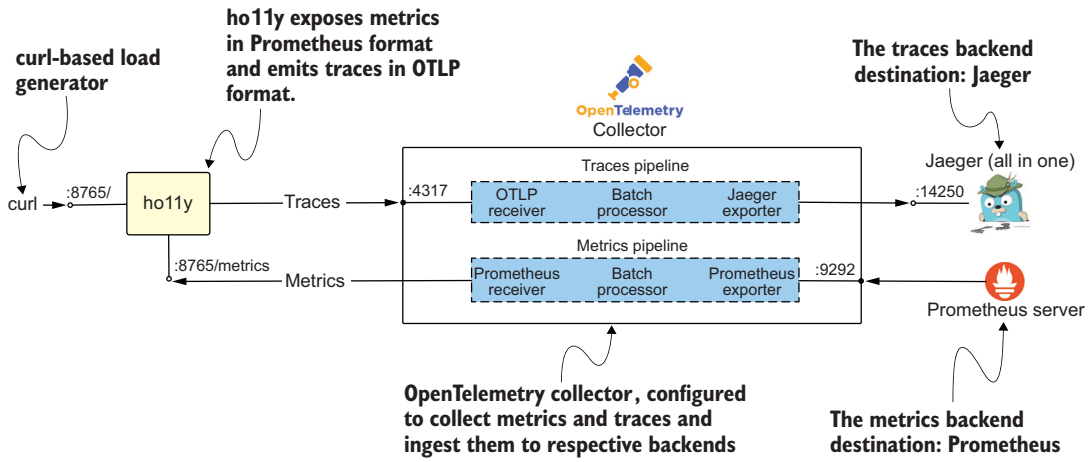


Figure 1.7 Example OpenTelemetry pipeline for traces and metrics

configuration for connecting to the OpenTelemetry collector via the `OTEL_EXPORTER_OTLP_ENDPOINT` environment variable.

Listing 1.5 Docker Compose configuration snippet for `ho11y`

```
ho11y:
    image: public.ecr.aws/mhausenblas/ho11y:stable
    ports:
    - "8765:8765"
    environment:
    - "OTEL_EXPORTER_OTLP_ENDPOINT=otel-collector:4317"
    - "OTEL_RESOURCE_ATTRIB=ho11y"
    - "HO11Y_INJECT_FAILURE=enabled"
    depends_on:
    - otel-collector
```

The port for the HTTP API

Configures the downstream OpenTelemetry collector (for OTLP over gRPC)

Listing 1.6 shows the OpenTelemetry collector using a custom configuration (`otel-collector-config.yaml`) we discuss a little later. We expose all these ports there to make them accessible from outside of Docker— that is, you can troubleshoot from the host machine more easily with this setup.

Listing 1.6 Docker Compose configuration snippet for the OpenTelemetry collector

```
otel-collector:
    image: otel/opentelemetry-collector-contrib:0.55.0
    command: ["--config=/conf/otel-collector-config.yaml"]
    volumes:
    - ./otel-config.yaml:/conf/otel-collector-config.yaml
    ports:
    - "4317:4317"
    - "8888:8888"
```

The collector config to use

The config file (mounted into the container from the host file system)

4317 is the OTLP receiver port.

8888 is the port the collector uses for metrics self-telemetry.

```
- "9292:9292"
- "13133"
depends_on:
- prometheus
- jaeger
```

9292 is where we expose the Prometheus exporter.

13133 is the port used for collector health checks.

Listing 1.7 shows Prometheus configured as a metrics backend destination to land metrics that we then can query via CLI or using the built-in UI. Note that the Prometheus scrape configuration (prom-config.yaml) referenced here is something we will be discussing a bit later in the chapter (see listing 1.12).

Listing 1.7 Docker Compose configuration snippet for backend Prometheus

```
prometheus:
    image: prom/prometheus:latest
    ports:
    - "9090:9090"
    volumes:
    - ./prom-config.yaml:/etc/prometheus/prometheus.yml
```

The Prometheus config to scrape the OpenTelemetry collector

Last but not least, listing 1.8 shows how we configure Jaeger, the trace's backend destination. Here the two things to note are port 14250, which we will use in the OpenTelemetry collector as the target to send traces to, as well as the environment variable `COLLECTOR_OTLP_ENABLED`, which we (at the time of writing, for the version of Jaeger we're using here) still need to set so that the OTLP over gRPC connection between the collector and Jaeger works.

Listing 1.8 Docker Compose configuration snippet for backend Jaeger

```
jaeger:
    image: jaegertracing/all-in-one:1.36
    ports:
    - "5778:5778"
    - "14250:14250"
    - "16686:16686"
    environment:
    - "COLLECTOR_OTLP_ENABLED=true"
```

5778 is Jaeger's config interface port.

16686 is the port where Jaeger exposes its frontend UI.

14250 is the port where Jaeger receives model.proto, which is OTLP over gRPC.

Now that we know all the components of our setup, let's move on to the configuration of the OpenTelemetry collector and Prometheus. To collect metrics and traces from the sample app, we will configure the OpenTelemetry collector with two pipelines, one for metrics and one for traces, as show in listings 1.9 through 1.11 (splitting the config into receivers, exporters, and the actual services configuration), which you can find in the ch04/otel/otel-config.yaml file.

Let's begin with the ingress path, the receivers.

Listing 1.9 OpenTelemetry collector configuration, receivers

```
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: "0.0.0.0:4317"
  prometheus:
    config:
      scrape_configs:
       - job_name: "holly"
         static_configs:
         - targets: ["holly:8765"]
```

Enables and configures the OTLP over gRPC receiver (http://mng.bz/4Drj) we're using for ingesting traces. We're declaring here that it should listen on port 4317, the official, IANA-registered port for this scheme and protocol.

Here, we enable the Prometheus receiver (http://mng.bz/QP9v) and configure it so that it scrapes our sample app.

Note that Docker Compose makes it so that the `holly` app will be reachable under said name. Let's move on to the egress path, the exporters.

Listing 1.10 OpenTelemetry collector configuration, exporters

```
exporters:
  logging:
    loglevel: debug
  prometheus:
    endpoint: "0.0.0.0:9292"
  jaeger:
    endpoint: "jaeger:14250"
    tls:
      insecure: true
```

We enable and configure the Prometheus exporter (http://mng.bz/XNla), turning our collector, effectively, in a pass-through.

Next up is the Jaeger gRPC Exporter (http://mng.bz/yQjd), which, as the name suggests, allows us to export (trace) data via gRPC to a Jaeger backend destination.

One thing before we move on: remember that listing 1.8, which showed the Jaeger configuration in Docker Compose, pointed out port 14250? That's exactly what we plug in here, in listing 1.10, so that the OpenTelemetry collector knows where to send the traces.

For the sake of keeping things simple in this example, we're using the Prometheus exporter to allow scraping by a Prometheus server. In a production setup, where federation and long-term storage are relevant, you likely will want to instead use the Prometheus Remote Write Exporter (http://mng.bz/MBGB) here. This enables you to ingest metrics into backends, such as Cortex, Thanos, or managed-OSS services, like Amazon Managed Service for Prometheus (https://aws.amazon.com/prometheus/).

> **TIP** In the collector config, components (receivers, processors, exporters, connectors, and pipelines) are defined by a component identifier in the `type[/name]` format. You can define components of a given type more than once as long as the identifiers are unique. For example, you could have two metrics pipelines, one called `metrics/dev` and one `metrics/prod`, with different backends.

Finally, we get to use all these receivers and exporters to define our pipelines, as shown in the following listing.

Listing 1.11 OpenTelemetry collector configuration, services (pipeline definitions)

```
service:
  telemetry:
    logs:
      level: debug
    metrics:
      level: detailed
  pipelines:
   metrics:
      receivers: [prometheus]
      processors: [batch]
      exporters: [logging, prometheus]
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [logging, jaeger]
```

A good practice in dev/test is to configure the collector telemetry to emit fine-grained agent logs and metrics (available via port 8888 on the collector).

This is where the pipeline definitions starts; you can have as many as you like.

Here, we define the metrics pipeline using the Prometheus receiver in the ingress path and the Prometheus exporter for egress.

We configure the traces pipeline that uses OTLP to ingest traces from ho11y and routes them to our downstream configured Jaeger instance.

WARNING If you only enable and configure a component such as a receiver or exporter, like we did in the previous steps, but don't use them in a pipeline definition, then it is like the component doesn't exist. The collector simply ignores it, and nothing works as you would expect. This is a very common pitfall I've seen in the wild and, admittedly, run into myself a couple of times.

Also, you might have noticed that in the `exporters` part, there are two entries. The reason is simple: by including the `logging` exporter, you get to see what's going on (e.g., metrics scraped) directly on the screen, since said exporter sends signals to `stdout`.

To wrap up the configuration, let's have a quick look at how downstream Prometheus (our metrics backend destination) is configured (listing 1.8): it scrapes itself, which is a good practice, and most importantly, for our overall setup to work, it scrapes the OpenTelemetry collector via `otel-collector:9292` (this is where we told the Prometheus exporter to expose its endpoint).

Listing 1.12 Prometheus server config, scraping the OpenTelemetry collector

```
global:
  scrape_interval: 5s
  evaluation_interval: 5s
scrape_configs:
  - job_name: "prometheus"
    static_configs:
    - targets: ["localhost:9090"]
  - job_name: "otel-collector"
    static_configs:
    - targets: ["otel-collector:9292"]
```

Self-scraping the Prometheus server

Scraping the Prometheus exporter in the OpenTelemetry collector

Alright, that was a lot of YAML code to look at. Now, let's run things. Fire up one terminal session and `cd` into `ch04/otel/`, and then do

```
docker-compose -f docker-compose.yaml up
```

NOTE This time, I've included the poor person's load generator, based on `curl`, in the Docker Compose file, so no further steps are required at this point.

Now, it's finally time to reap the fruits of our hard YAML wrangling, so open up your browser of choice and enter the two URLs for Prometheus (localhost:9090) and Jaeger (localhost:), and you should see something happening.

Switch to Prometheus and enter the PromQL query

```
sum by(http_status_code) (ho11y_total)
```

You should see (on the Graph tab) something like what is depicted in figure 1.8—that is, the sum of the overall sample app invocations (HTTP API calls), grouped by the HTTP status code returned (it's configured in a way to, on average, cause a 10% error rate).

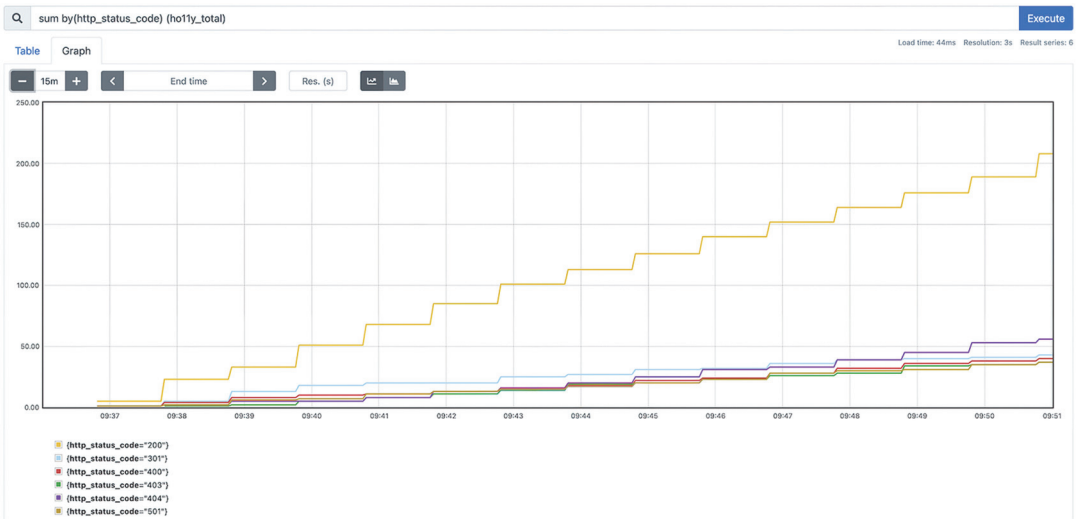


Figure 1.8 The Prometheus UI in action, showing a simple `ho11y` metric plotted over time

Moving on to traces: in Jaeger, you first have to select the service. You do this by choosing it from the respective dropdown box. Further, in the Tags field, you will enter

```
http.status_code=403
```

This means you're only interested in traces that carry this specific tag—in other words, traces that capture events where an HTTP status code 403 was returned by the `ho11y` sample app. Now, press the Find Traces button in the left-hand navigation, and the result should be akin to what is depicted in figure 1.9.

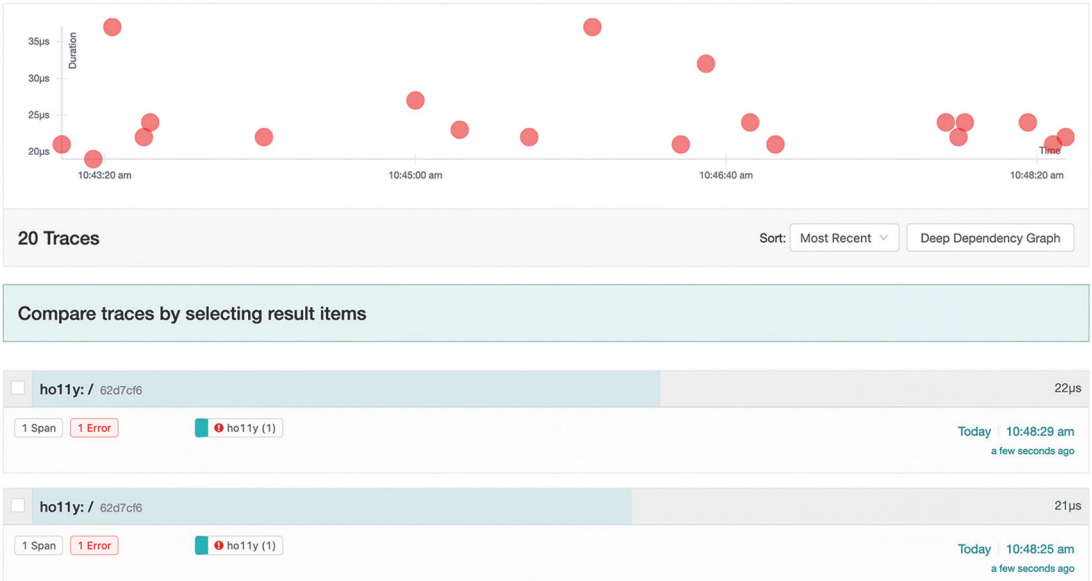


Figure 1.9 The Jaeger UI in action, showing `ho11y` traces filtered by HTTP 403 status code

Keep in mind that looking at an individual trace in Jaeger is as simple as clicking it in the right-hand list of traces, so go ahead and pick a random one. This should lead you to a view that resembles what figure 1.10 shows.

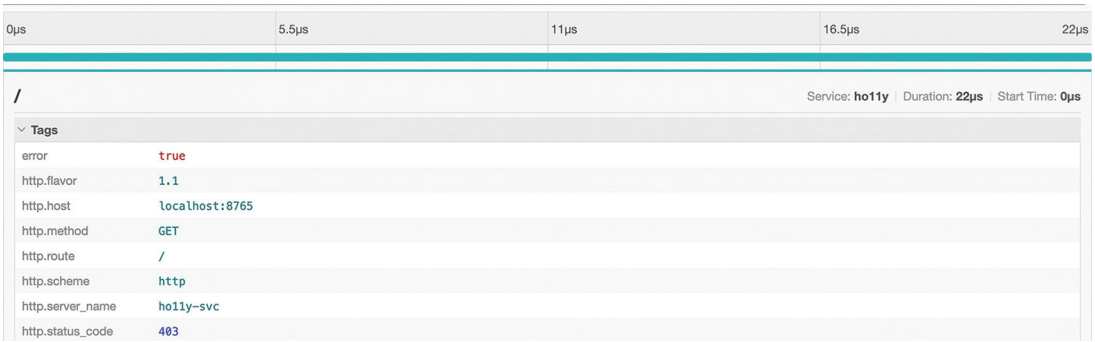


Figure 1.10 Drilling into a specific trace in Jaeger, showing all the available details

To sum it all up, in this OpenTelemetry in action walk-through, we have

1. Generated traces and metrics via manual instrumentation.
2. Configured the OpenTelemetry collector to collect these signals and ingest them into two different backend destinations (Jaeger, for traces, and Prometheus, for metrics).

3. Had a look at the signals in the backends. In Prometheus, we plotted the overall invocations by HTTP response code, and in Jaeger, we looked at client-side error spans (containing the 403 HTTP response code).

OK, that was a lot—congratulations on sticking with it! I hope this little OpenTelemetry walk-through gave you a first taste and an idea of the power that's available to you via this universal telemetry agent.

> **TIP** When it comes to OpenTelemetry distributions (http://mng.bz/a12m), there are several ways you can go about it: you can use the upstream, prebuilt distribution (http://mng.bz/gBnv) readily available, you can choose the distribution of a specific cloud provider or o11y vendor (e.g., the AWS distro [http://mng.bz/5wBB]), or you can decide to roll your own distribution using the OpenTelemetry collector Builder (http://mng.bz/5wVO), a tool provided by the project.

If you want to dive deep into OpenTelemetry, learn where it came from, what the overall vision is, and how to apply it in your setting, I recommend perusing the following books:

- Alex Boten's excellent book, *Cloud Native Observability with OpenTelemetry* (https://www.cloudnativeobservability.com/), providing you more context on OpenTelemetry and many hands-on examples.
- *Practical OpenTelemetry: Adopting Open Observability Standards Across Your Organization* by Daniel Gomez Blanco (Apess, 2023; http://mng.bz/6Dxe), which is an awesome book for learning but also for help introducing OpenTelemetry to an organization (based on his own experiences).
- *Learning OpenTelemetry* by Austin Parker and Ted Young (O'Reilly; http://mng.bz/o1gZ), the most recent book in this space, promises to be a reference book, with the two authors being driving forces behind OpenTelemetry itself.

To round off our agent review, let's have a quick look at established agents that are currently in use and may be replaced by OpenTelemetry in the future.

1.4 Other agents

In addition to the Fluent family for logs, Prometheus for metrics, and the OpenTelemetry collector as the universal telemetry agent, there are several established and widely used agents. Most of them have been written by and are distributed by cloud providers, but additionally, some observability vendors have introduced agents, so let's have a quick look at them (the same idea applies: in the fullness of time, all of them likely will be replaced by OpenTelemetry or its collector):

- Amazon CloudWatch Agent (https://github.com/aws/amazon-cloudwatch-agent/) can be used for logs and metrics, supporting AWS's proprietary embedded metric format (EMF; http://mng.bz/nWa5). The CloudWatch agent already supports OpenTelemetry via an embedded collector and has been replatformed on OpenTelemetry (http://mng.bz/G91q). In addition to the Cloud-

Watch agent, AWS already offers an OpenTelemetry-compliant collector via its AWS Distro for OpenTelemetry (https://aws.amazon.com/otel/).

- Azure is in the process of deprecating its Log Analytics agent and Telegraf agents in favor of the new Azure Monitor agent (AMA; http://mng.bz/2DNd). And just like the other cloud providers, Azure is (in the long term) standardizing on OpenTelemetry (http://mng.bz/vn0p).
- Similar to Azure, Google Cloud is moving away from its legacy, collectd-based monitoring agent, in favor of the new Ops Agent (https://cloud.google.com/monitoring/agent/ops-agent), which uses Fluent Bit for log collection and routing, as well as the OpenTelemetry collector, for metrics.
- Vector (https://vector.dev/docs/about/what-is-vector/) by Datadog is an interesting open source project that, to some extent, competes with OpenTelemetry, from the universal signal type support angle; however, it also supports OpenTelemetry. It seems like a good choice of agent if you're all in with Datadog.

Now that we have a good understanding of which agents exist, their coverage, and use cases, let us move on to the topic of how you should go about selecting an agent.

1.5 Selecting an agent

As is often the case, there are several choices available to you. It's important to make sure you pick an agent that is a good fit for your workloads, environment, and organizational capabilities.

It doesn't hurt to consume general material on the topic, such as blog posts and articles—for example, see "Logstash, Fluentd, Fluent Bit, or Vector? How to Choose the Right Open-Source Log Collector" (http://mng.bz/4DrR) or "Who Is the Winner: Comparing Vector, Fluent Bit, Fluentd Performance (https://medium.com/ibm-cloud/log-collectors-performance-benchmarking-8c5218a08fea); however, you should use them as an inspiration or starting point rather than a "buyer's guide." In other words, it is very likely you will have to do your own evaluation, based on your needs and preferences, to properly select an agent. In no particular order, I present you with criteria you can use as, at the very least, a starting point to evaluate agents and pick what's right for you.

1.5.1 Security for and of the agent

Where I work, at AWS, security is the highest priority, so I start with this. It is important to consider both the security *of* the agent and the security *for* the agent, by which I mean the following:

- With security *for* the agent, ask yourself and/or the vendor the following:
  - Who handles CVEs and security patches?
  - Is the agent pen tested and, if so, when (e.g., one-off, regularly, or when major components change)?

  - Can you get reproducible builds for artifacts, such as container images of the agent?
  - Is the agent's supply chain known and/or documented?
- With security *of* the agent, try to determine the following:
  - Does the agent support TLS/HTTPS and equivalent cryptographically secured traffic?
  - Can you apply policies across a fleet of agents (e.g., credentials, access, and filters)?
  - Are mechanisms in place to deal with sensitive data, such as personally identifiable information (PII), as required by certain legislations, such as in Europe, through the General Data Protection Regulation (GDPR)?

Ideally, you will get this information up front, via a checklist from the open source project, ISV, or cloud provider vending the agent.

1.5.2 Agent performance and resource usage

Agent performance—that is, how fast signals can be processed, typically measured via the throughput along with the resources used—is a key evaluation criteria. The following are some examples of such performance-related reports:

- The upstream "OpenTelemetry Collector Performance" docs (http://mng.bz/QP96).
- The AWS Distro for OpenTelemetry Collector (ADOT) collector performance report (http://mng.bz/N2J1).

Zooming in on performance, one important criterion is, in a nutshell, related to the resource footprint of an agent. For example, a criterion might be how much RAM and/or CPU seconds the agent consumes in various phases (e.g., idle, on average, or peak) to collect, mutate, and ingest signals.

Along with this, you want to be able to have breakdowns across signal types in case of a multisignal agent, like the OpenTelemetry collector. Figure 1.11 shows an example screen capture of a Grafana dashboard (https://ref.otel.help/otel-collector-ops/otel-collector-dashboard.json) capturing OpenTelemetry collector resource usage over time.

In the context of resource usage, you want to make sure the resources the agent uses are few compared to your footprint. Don't forget the cumulative effect—that is, while a single agent might only use up a few MB across an entire fleet with several thousands of agents, that might sum up to the high TB range. Most agents nowadays have mechanisms that allow you to control resource usage (e.g., in the case of the OpenTelemetry collector, you can use the memory limiter processor to prevent out-of-memory (OOM) scenarios.

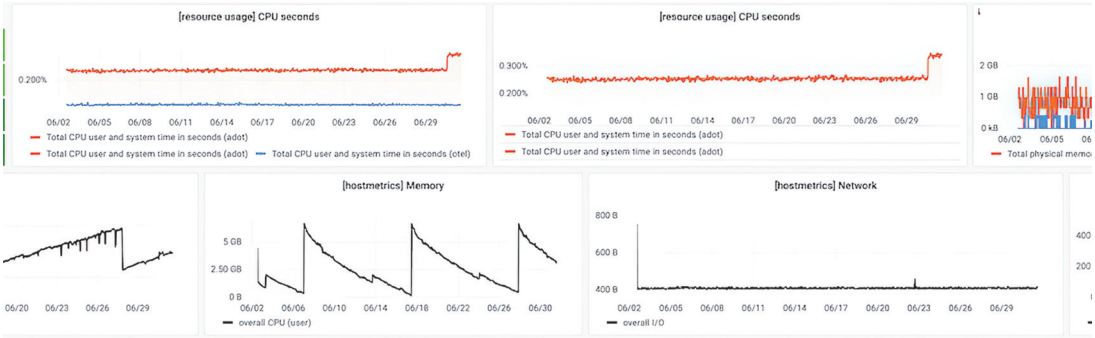


Figure 1.11 Grafana dashboard of OpenTelemetry collector resource usage

1.5.3 Agent nonfunctional requirements

In addition to the functional aspects, it's important to ask yourself the following:

- Is the agent open source, extensible, or backed by a single vendor or community?
- Who watches the watcher, or, put differently, what about the agent telemetry? In 2023 (and beyond), you should expect not only to have access to the agent logs but also get metrics (ideally, in Prometheus format) as well as traces and profiles from your agent.
- Is the agent capable of dealing with one or more signal types? A signal-specific agent means you need multiple agents for 360 coverage.

There is much more to nonfunctional requirements than we've discussed, but again, these questions form the basics, and you may choose to extend them with organizational requirements or preferences when selecting an agent.

Summary

- There are two types of agents: vendor-specific ones, like the Datadog agent, and signal-specific ones (usually open source), like Fluent Bit for logs.
- In this book, we consider vendor-specific and signal-specific agents as traditional agents, in contrast to the open source, all-encompassing OpenTelemetry collector.
- OpenTelemetry is a set of specifications, SDKs, a protocol (OTLP), and agents.
- OpenTelemetry allows you to collect, process, and ingest logs, metrics, traces, and (in the future) profiles in a vendor-neutral way.
- With OpenTelemetry, you get rich metadata both on where the signals came from (resource attributes) and about the telemetry signals themselves.
- Based on the semantic conventions, OpenTelemetry enables you to implement effective correlation in the backends.

- Auto-instrumentation is an important baseline method to automatically generate telemetry without changing the application code.
- If a vendor or programming language (in OpenTelemetry) supports auto-instrumentation, use it.
- Where auto-instrumentation doesn't deliver what you need (business logic), you need to manually instrument your code with an OpenTelemetry SDK.
- Use the OpenTelemetry collector to collect and route traces and metrics, and eventually logs, from any source to any destination, forming a universal telemetry agent.
- Agents need to be secure, performant, and lean (minimal resource usage).
- Selecting an agent can take time and may require some effort but is worth the investment in the long run.

2 Backend destinations

This chapter covers

- What we mean by backend destinations and what types of destinations exist
- Backend destination options for handling logs, metrics, and traces
- What time series data stores are and what you can use them for
- Introducing columnar data stores and formats along with their use cases
- Considerations for selecting backend destinations

In this chapter, we will dive deep into backend destinations, the targets or sinks for agents. Backend destinations, or backends for short, are the source of truth for your observability questions. They allow you to store and query signals and make them available to frontend destinations.

Why do I make the distinction between backend and frontend? Well, there are a few examples that fall exactly in one or the other category. For example, Grafana is a pure frontend destination (sometimes also called the *presentation stage*) that queries backend destinations to visualize or alert on signals. The Cloud Native Computing Foundation (CNCF) project Cortex, on the other hand, is a pure backend destination (besides a rudimentary admin Web UI, you only have the API to ingest and query metrics).

Then, there are offerings from dedicated observability vendors, such as Splunk, Datadog, Dynatrace, Honeycomb, New Relic, Lighstep by ServiceNow, AppDynamics by Cisco, and Sumo Logic, which typically combine frontends and backends, providing an integrated experience. In those cases, you can't use the frontend (UI) separately from the backend. There are also mixed cases in the open source space, like CNCF Jaeger (which sports an ingestion pipeline but requires an additional storage system).

Another special case is cloud-provider-native offerings, especially Amazon CloudWatch, Azure Monitor (including Application Insights), and the Google Cloud operations suite, which cover a wide range of features and provide integrations with different backends and frontends.

> **NOTE** Regarding why I am using the logs, metrics, and traces boundaries to present topics: I am making the frontend–backend distinction mostly on a conceptual level. This should aid in your understanding; however, by now, you know that the reality is much messier and more complicated.

Now, let's get into it. In this chapter we will

- Have a look at some commonalities and general backend terminology
- Review options for and usage of logs backends
- Review options for and usage of metrics backends
- Review options for and usage of traces backends
- Discuss columnar data stores
- Discuss how to go about selecting a backend

Ready? Let's dive in!

2.1 Backend destination terminology

Backends are the source of truth for all your observability questions, be that a human asking them (ad hoc, based on a hunch) or a piece of code (think an alert that queries a backend to evaluate if a certain condition is met). Before we get into the different types of backends and their pros and cons, we will first step back a bit and look at their common characteristics along with some basic terminology we need for the rest of the chapter.

In general, there are three main parts you want to consider with backends, whether that is concerning performance or costs (see section 2.6 for more):

- *Ingestion*—How the signals arrive at the backend. Various protocols and formats exist, such as OpenTelemetry's OTLP or the Prometheus formats.
- *Storage*—Usually, there's the expectation that the signals ingested are stored for a longer time period (weeks, months, or even years), allowing for historical query and data analytics. But this doesn't always have to be the case, as pure in-memory solutions exist (e.g., CNCF Pixie) and have their uses, such as live cluster debugging, keeping only hours of (volatile) data around.
- *Query*—The more exact and common term we use for *asking a backend a question* is to *query the backend*. What exactly *query* means can differ from backend to backend and can range from a declarative style, such as SQL, to an API call, such as is the case with many tracing backends. You will also often come across domain-specific languages (DSLs), such as PromQL, and you'll see several examples of these DSLs in use throughout this chapter. With this in mind, let's move on to the first, and probably most widely adopted, kind of backend, for all sorts of logs.

2.2 Backend destinations for logs

In this section, we will take a cursory look at backend destinations that you can use to ingest logs and query them. Logs are timestamped, typically capturing events; have a (hopefully structured) payload; and are primarily meant for human consumption. The sheer number of different log types and formats makes it sometimes difficult to figure out what the optimal backend is. Usually, it depends heavily on the use case and your preferences, such as open source and open standards first, or you might prefer a managed offering over a DIY approach (or the other way round). Having said that, and of course this also applies to the rest of the backends for metrics and distributed traces, let's see what we have here.

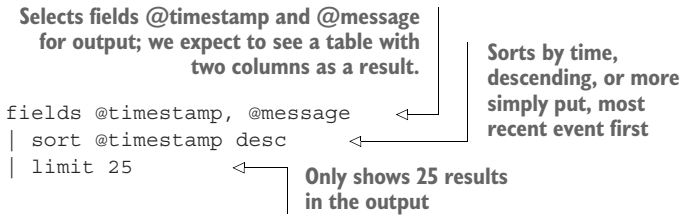
2.2.1 Cloud providers

Every cloud provider has, at the very least, a service in their offerings that allows you to land logs—more specially, that is (focusing on the big three hyperscalers):

- Amazon CloudWatch Logs (http://mng.bz/yQjp) organizes different kinds of logs in log groups that, in turn, comprise one or more log streams (think of a never-ending series of events), which are made up of log events. It uses a proprietary way to ingest the logs and offers the embedded metric format (EMF; http://mng.bz/nWa5), a format that allows you to embed custom metrics alongside logs. In other words, EMF is more event-centric (supporting high cardinality) than a pure log format. You can use CloudWatch Logs Insights, which provides a purpose-built query language to query your logs.

- Azure Monitor Logs (http://mng.bz/MBGQ) allows you to collect, organize, and analyze log and performance data from monitored resources in Azure (from resource groups to the app level). It organizes logs in workspaces, with queries written in Kusto Query Language (KQL), which is somewhat similar to what you find in CloudWatch Logs Insights. It sports an interesting feature called *functions*, enabling you to reuse query logic as well as rich export functionality (rule-based with various destinations, from storage to event hub).
- Google Cloud Logging https://cloud.google.com/logging/) is part of the Google Cloud's operations suite, and it includes storage, query, and log routing. As the top-level unit, Google Cloud Logging uses the log to organize entries. The log comprises log entries (events) that can stem from Google Cloud services or third parties (platform-level) as well as applications. What stands out in this context is how flexible the destinations for log routing are: from JSON files stored in Cloud Storage buckets (think the equivalent of S3 buckets) to BigQuery (a large-scale, ad hoc SQL distributed query engine for data warehousing) to Pub/Sub (Google's event stream) to things that are more comparable with what other clouds provide—that is, log "buckets" that support customizable retention periods.

Let's have a look at a concrete logs example using CloudWatch. Examine the following CloudWatch Logs Insights query (where CloudWatch reports it scanned 3.7 GB of data, covering 3.1 million records):




```
fields @timestamp, @message
| sort @timestamp desc
| limit 25
```

WARNING With the `limit 25` in the preceding CloudWatch Logs Insights query example, we instructed CloudWatch to limit the output displayed to 25, but CloudWatch still needs to scan a large number of logs, depending on the time range selected. The time range, and with it the volume of the data scanned, is something you should always consider. On the one hand, it can take a long time, up to several minutes, for the query to return, which is far from interactive usage; and on the other hand, you're oftentimes charged for the amount of data that has been scanned.

Figure 2.1 shows the result of the preceding CloudWatch Logs Insights query. Typically, cloud providers land all the system-level logs (sometimes called *platform logs*, such as syslog and journald but also cloud-platform/API logs) in their own offerings, which makes it a good default starting point, especially for all nonapplication-level logs. If you're curious how to route your application-level logs to these destinations, refer to

chapter 1, where we discussed options and usage for agents. Next up, let's have a look at open source offerings in this space.

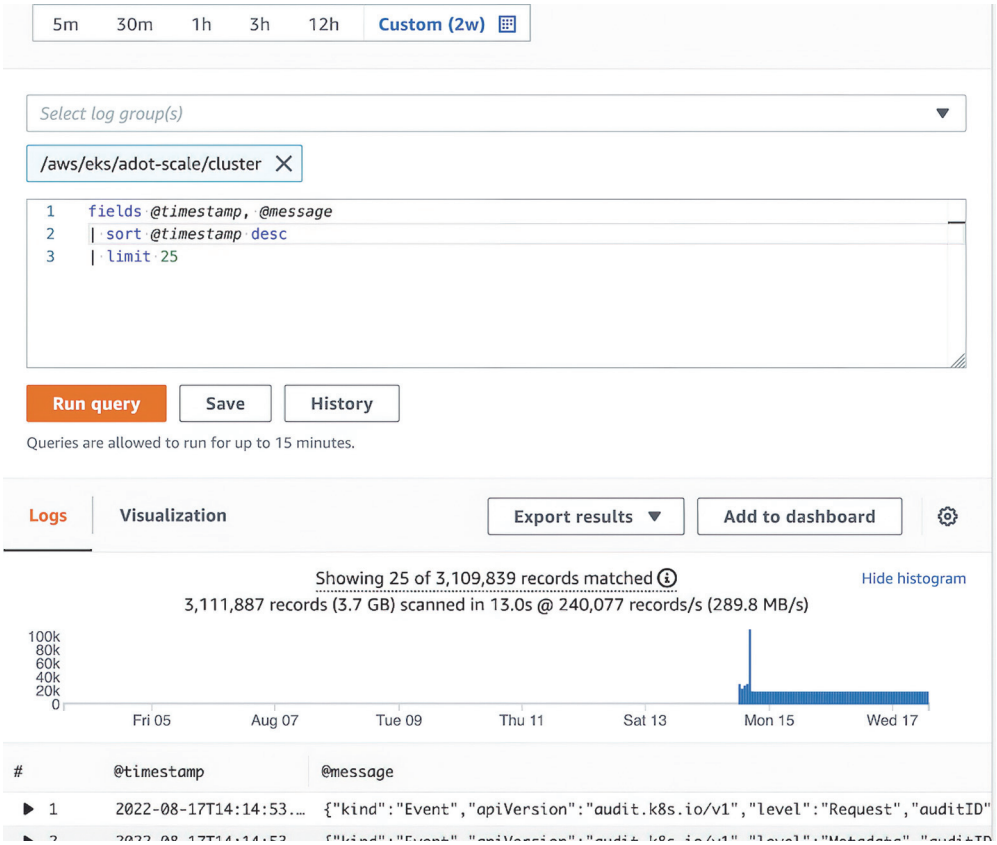


Figure 2.1 CloudWatch Logs Insights query result showing the 25 most recently added log events

2.2.2 Open source log backends

There are not that many widely used, open source–based log backends available (compared to metrics and traces, although as you will see later, log and trace backends share a lot of characteristics, and backends often store them similarly):

- *Elasticsearch and OpenSearch*—I'm lumping Elasticsearch (https://www.elastic.co/elasticsearch/) and OpenSearch (https://opensearch.org/) together—ES/OS for short, going forward—because it really is a family of two projects and products (managed services) with a common root. They are both extending Apache Lucene (https://lucene.apache.org/), a veteran, comprehensive text search engine written in Java to make it easy to use for signal ingestion and

query, even in a distributed setup. OpenSearch started out as a fork of Elasticsearch, sponsored by Amazon, and reached general availability (http://mng.bz/a12j) with its 1.0 version in July 2021. As time goes by, the ES/OS code bases are expected to diverge, but from the get-go, the major difference was the license under which each project was distributed. However, ES and OS have the fact that they are logs-first backends in common (optimized for storage and retrieval of logs). As a reminder, in chapter 1, we used OpenSearch as a destination for the Fluent Bit agent.

I won't cover ES/OS in detail in this book, as there are plenty of resources available on the topic, so if you want to dive deep, I'd recommend reading the excellent second edition of the Manning book *Elasticsearch in Action* by Madhusudhan Konda (2023; https://www.manning.com/books/elasticsearch-in-action-second-edition).

We will get back to the ES/OS topic in the context of traces later in this chapter, since this is a signal type ES/OS have decided to cover nowadays as well (support for metrics is also available; however, it is not as full blown as for the other two types).

- *Grafana Loki (https://grafana.com/oss/loki/)*—This is a horizontally scalable, multitenant log aggregation system, released under the AGPLv3 license, and its design was inspired by Prometheus. It does not index the contents of the logs themselves (as ES/OS do) but rather a set of labels for the ingested log streams. It's certainly something to take a close look at; currently, few managed offerings based on it exist.
- *ZincObserve (https://github.com/zinclabs/zincobserve)*—This is a search engine you can use for indexing and looking up logs, with a focus on reducing log storage costs.

Now, let's move on to commercial offerings in the space.

2.2.3 Commercial offerings for log backends

The commercial offering space for log backends is well set up, and you will find some common features across of all of these, such as paid ingestion, strong multitenancy support, and a wide range of built-in integrations with other signal sources, from apps to cloud provider offerings. The offerings include but are not limited to the following:

- *Splunk (https://www.splunk.com/)*—The established logs incumbent, with a range of enterprise features and strong OpenTelemetry commitment
- *Instana (https://www.instana.com/)*—Acquired by IBM and provides many interesting features and integrations
- *SolarWinds (https://www.solarwinds.com/)*—Acquired formerly independent SaaS offerings Loggly and Papertrail as well as the monitoring offering Pingdom
- *Logz.io (https://logz.io/)*—A start-up with an amazing range of offerings in the cloud-native domain (using OpenSearch and Jaeger)

No matter where you look, logs are always in use, so the question is not *if* but *which* backend you should use. We will get back to the selection criteria later in the chapter, but for now, you can rest assured that you will need a log backend. With this, you have a rough idea what logs backends can do and what options exist so let's move on to metrics.

2.3 Backend destinations for metrics

Time series databases (TSDB) are the bread and butter of storing and querying signals of the metrics type. You can use TSDBs to track a range of different metrics, from low-level systems metrics, such as the memory usage of a container, to application-level metrics, such as the number of invocations of your service.

What is a TSDB, and how does it work? As shown in figure 2.2, the main index is the time. The unit of bookkeeping is the metric that comprises one or more time series. Each time series, in turn, keeps track of a value and, typically, you can have many of these time series active at the same time (in the sense that you can use them

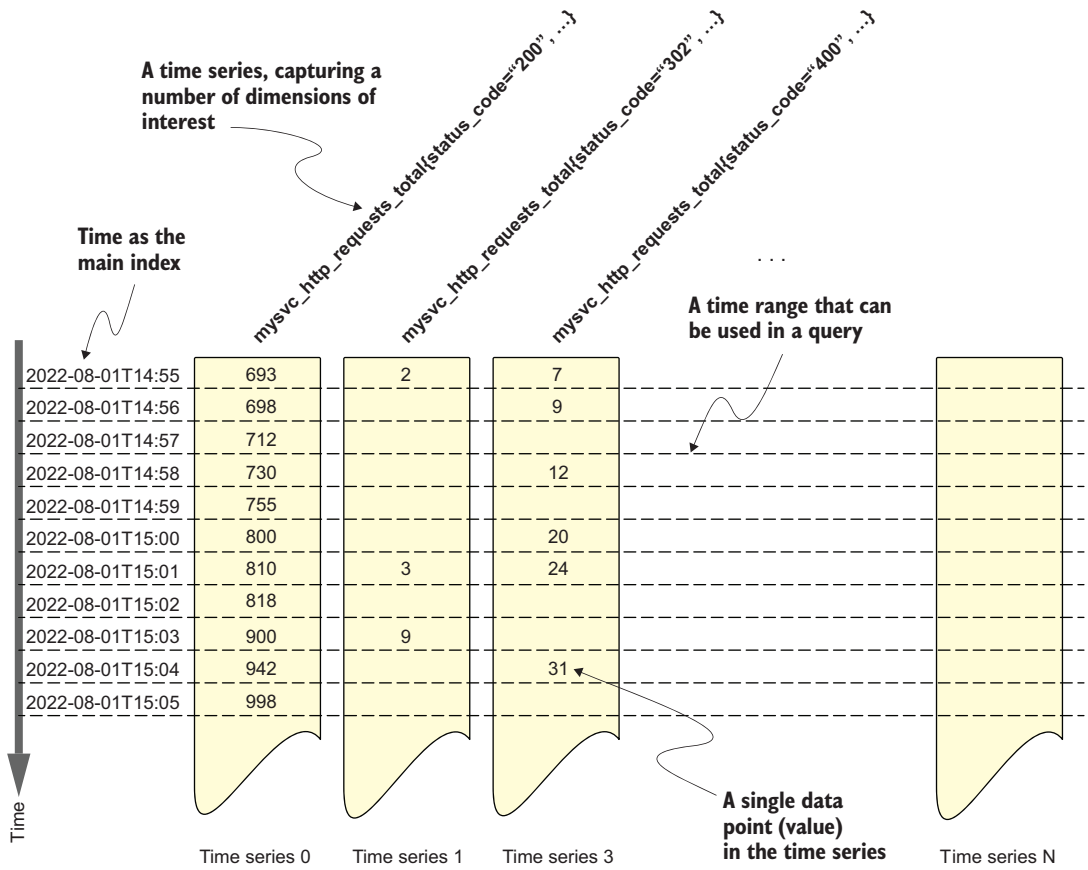


Figure 2.2 Time series database concept, showing N time series of the `mysvc_http_request_total` metric

in a query, with their values readily being available). The typical way to interact with the TSDB is to ask, "In the time frame from *x* to *y*, what are the values or value aggregates matching these conditions?"

The main index used in a TSDB is the time, usually divided into equal buckets. To keep things simple, figure 2.2 uses minute intervals, but usually, the intervals can be as small as seconds or milliseconds in granularity.

Each time series has a unique name (e.g., `mysvc_http_request_total{status_code="400"}`), capturing several dimensions of interest. The dimension of interest here is the HTTP status code returned (`status_code`), and many more, like the HTTP URL path, HTTP method, and so on might also be something you want to track. The metric this time series belongs to is of the type `counter`. That means its values increase strictly monotonically, and it captures the overall number of invocations of the `mysvc` service's HTTP API. In this context, the net change between time periods will give you the number of invocations in a period. A single data point, addressed by time and unique time series, represents the value. In our case, as shown in figure 2.2, it means that at time `2022-08-01T15:04`, the TSDB has seen a total of 31 invocations that resulted in a `400` HTTP status code.

Oftentimes, you want to query a time range, rather than a particular instant in time. You can do this by specifying the start (`2022-08-01T14:58`) and end (`2022-08-01T15:01`). You can, for example, ask the TSDB how many invocations of a certain type occurred; in our case, if you're interested in client errors, you'd ask how many `400` HTTP status codes were witnessed in the time frame, and the answer would be 12 (at the end of range, we have 24, and at the start, we had 12).

Cardinality explosion

You might wonder what happens if you start adding dimensions to a metric you want to track, with a metric being the umbrella term for all related time series. In this example, let's say a dimension you're interested in is the user ID.

In contrast to, say, the HTTP status code, which has a small, bounded number of values (a couple of dozens, from 1xx to 5xx), the user ID could have many values. If your app is successful, potentially serving many hundreds of thousands of users, this means the TSDB has to ingest and store hundreds of thousands of time series (for that single metric only, and you may have thousands of metrics you want to track).

This idea of many different values a dimension can take on being multiplied by the number of metrics is what we call *cardinality explosion* and can result in slow query times, since the TSDB needs to inspect each time series in a given time range, for each dimension. That systems property makes TSDBs for certain types of observability use cases not the best fit: at the very least, you should be aware of this and avoid adding dimensions that you know or suspect of "exploding," in terms of their cardinality.

If you would like to dive deeper into the topic of TSDBs and their use cases in general, consider reading the article "What Is Time-Series Data? Definitions & Examples" by Ajay

Kulkarni, Ryan Booz, and Attila Toth (https://www.timescale.com/blog/time-series-data/). Now, let's switch gears and get practical by looking at several open source and commercial offerings you can use as TSDB backends, especially in the context of Prometheus-compatible cases. We covered the Prometheus basics in chapter 1 and will not repeat them here. We will focus on mid- to large-scale production use cases in this chapter—in other words, when a single Prometheus instance typically is not sufficient anymore. Let's again look at the options for cloud providers, open source offerings, and commercial offerings.

> **NOTE** Prometheus is, by design, not horizontally scalable. That is, you can add more RAM and locally attached persistent storage to scale Prometheus vertically to handle more metrics or deal with a longer time period. What you can't expect is that running multiple copies of Prometheus automatically means the telemetry data is managed for you in such a distributed setup.
>
> If you do have use cases such as a fleet of Kubernetes clusters, where you want to land metrics from many machines in a single place (federation), or you want to be able to query long time periods, such as the past three years, and a single Prometheus instance can't handle the task, you are looking for long-term storage (LTS) solutions. The following options can handle these use cases.

2.3.1 Cloud providers

Every cloud provider has, at the very least, a service in their offerings that allows you to ingest metrics as well as some open source or Prometheus-based ones, in addition to their native offerings. The following are some of the most prominent hyperscaler options, offering both platform and application-metrics ingestions:

- *Amazon CloudWatch Metrics (http://mng.bz/gBne)*—Covers metrics from over 100 AWS services (vended metrics) and keeps them around for 15 months, allowing you to query up-to-the-minute data and historical data equally. You can use the property query language as well as SQL (via CloudWatch Metrics Insights) to query metrics as well as alert (called alarms) on certain conditions. It also comes with dashboards that provide insights out of the box.
- *Amazon Managed Service for Prometheus (http://mng.bz/e169)*—A Prometheus-compatible monitoring service that uses CNCF Cortex in its data plane. You can ingest metrics via the Prometheus remote write interface (http://mng.bz/pPe5) and use PromQL to query it.
- *Amazon Timestream (http://mng.bz/Ox0K)*—A serverless time series database service for IoT and operational applications that offers an SQL-based query language and enables you to use it in a more traditional big data use case.
- *Azure Monitor Metrics (http://mng.bz/Y1Bo)*—Allows you to collect, organize, and analyze metrics from monitored resources in Azure. It supports namespaces to logically group time series as well as rich exploration and fine-grained storage and retention options.

- *Google Cloud Monitoring (https://cloud.google.com/monitoring)*—Part of the Google Cloud's operations suite; it enables you to collect metrics, events, and metadata from Google Cloud, AWS, hosted uptime probes, and application instrumentation.
- *Google Cloud Managed Service for Prometheus (http://mng.bz/GyaM)*—Runs Prometheus-based collectors as DaemonSets in Kubernetes and ensures scalability by only scraping targets on colocated nodes and pushing the scraped metrics to the central data store (Monarch). You can use PromQL to query the metrics, for alerts or, for example, as a data source in Grafana.

Figure 2.3 shows an example metrics explorer output—here, the Metrics Explorer being part of the Azure Monitor Metrics service. Next, let's look at open source offerings for metrics LTS and federation.

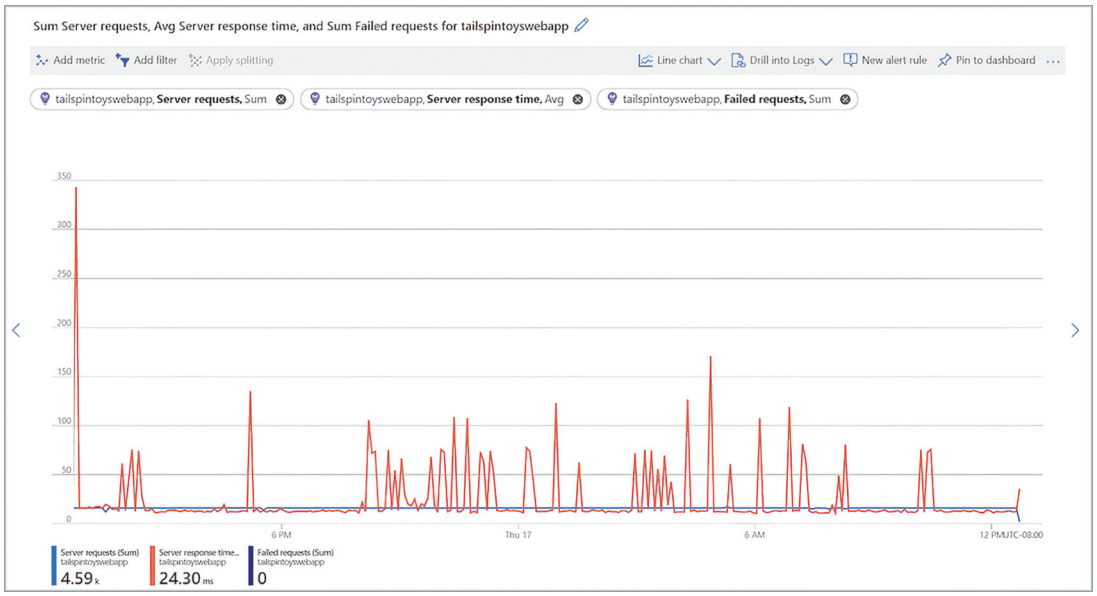


Figure 2.3 Azure Metrics Explorer example output (Azure docs, http://mng.bz/Y1Bo)

2.3.2 Open source metrics backends

The metrics backend space is rather crowded, which is good for you, in terms of options available, but it can also be challenging to keep track of developments and make the best decision for workload:

- *CNCF Cortex and Thanos*—Two CNCF projects with different philosophies about how to ingest and query Prometheus timeseries. Cortex (https://cortexmetrics.io/) is used in Amazon Managed Service for Prometheus; and Thanos (https://thanos.io/), founded by Improbable, is used in a variety of cases by many large

companies, from Amadeus to eBay to SoundCloud to Tencent. Cortex has a push approach (using the `remote_write` Prometheus interface) to allow a Prometheus fleet to ingest metrics, whereas Thanos initially had a pull approach (using the `remote_read` API), allowing one to federate metrics from many Prometheus setups (but now, it also supports push).

- *Clymene (https://clymene-project.github.io/)*—A time series data and logs collection platform for distributed systems that allows the storage of signals in different types of databases, such as for logs or metrics. It is a relatively new system, but in its design, it is cloud native.
- *Grafana Mimir (https://grafana.com/oss/mimir/)*—A horizontally scalable, highly available, multitenant, long-term storage option for Prometheus, available under the AGPLv3 license. It started in 2022 as a fork of the CNCF Cortex project, and Grafana Labs has added many improvements, making it part of their SaaS offering as well.
- *Icinga (https://github.com/icinga/icinga2)*—An open source monitoring system that checks the availability of your network resources, notifies users of outages, and generates performance data for reporting. You don't find Icinga widely used in cloud-native setups, but you might still need to be able to interop with it.
- *InfluxDB (https://github.com/influxdata/influxdb)*—A popular open source time series platform spanning a wide range of functionality, including offline process of the data, alerting, and exploration, with good Prometheus compatibility.
- *LinDB (https://github.com/lindb/lindb)*—A scalable, high-performance, and highly available distributed time series database that features its own SQL-like query language, called LinQL.
- *M3DB (https://m3db.io/)*—Originally from Uber, this time series data store is used by Walmart, LinkedIn, and many others. It's capable of handling billions of metrics (see this FOSDEM 2020 talk on its scaling: https://archive.fosdem.org/2020/schedule/event/m3db/) is now also available via Chronosphere as a managed service.
- *Nagios Core (https://github.com/NagiosEnterprises/nagioscore)*—An open source app that enables you to monitors systems, networks, and infrastructure.
- *Netdata* (https://github.com/netdata/netdata)—An open source, high-fidelity infrastructure monitoring and troubleshooting tool (agent and UI).
- *Nightingale (https://github.com/ccfos/nightingale/blob/main/README_en.md)*—A cloud-native monitoring system you can use to wrap and enhance the Prometheus, Alertmanager, and Grafana combo.
- *OpenTSDB (https://github.com/OpenTSDB/opentsdb)*—A distributed, scalable TSDB written on top of HBase. Its heyday was in the mid-2010s, when HBase was widely used.
- *Promscale (https://www.timescale.com/promscale)*—A unified observability backend for Prometheus metrics and OpenTelemetry traces, built on PostgreSQL and a TSDB extension called TimescaleDB (https://www.timescale.com/).

- *QuestDB (https://github.com/bluestreak01/questdb)*—A high-performance, open source SQL database for applications from financial services to observability. It includes endpoints for PostgreSQL wire protocol; high-throughput, schema-agnostic ingestion, using InfluxDB Line Protocol; and a REST API for queries, bulk imports, and exports.
- *Zabbix (https://github.com/zabbix/zabbix)*—An open source distributed monitoring solution. Like others in this list (Nagios, Icinga, and OpenTSDB come to mind), it is not widely used today in cloud-native setups, but you should be aware of it.

Some of the open source projects are packaged and offered as services. Let us move on to this topic now.

2.3.3 Commercial offerings for metrics backends

There are a few offerings dedicated to metrics, including the following:

- *Chronosphere (https://chronosphere.io/)*—Essentially "M3DB as a service." They make good progress in terms of features and OpenTelemetry compatibility.
- *VictoriaMetrics (https://github.com/VictoriaMetrics/VictoriaMetrics)*—A monitoring solution with integrated TSDB that you can use standalone or as a service. While Prometheus compliance (https://github.com/prometheus/compliance) is, at time of writing, a growth area for VictoriaMetrics, the overall offering is cloud native and powerful.

In most, if not all, cases (Kubernetes, ECS), you want and need metrics and, with them, a backend. Choose your backend based on Prometheus compatibility and your specific usage pattern (e.g., ingest, storage, or query distribution).

2.4 Backend destinations for traces

Now, we have a look at various options to ingest and query distributed traces. In this space, you would typically find wide support for OpenTelemetry, although Zipkin support is still needed in some places, such as PHP.

2.4.1 Cloud providers

Just as with logs and metrics, your cloud provider of choice has something to offer if you want to ingest and use tracing:

- *AWS X-Ray (https://aws.amazon.com/xray/)*—Amazon's managed distributed-tracing offering, which allows you to analyze and debug applications. It used to be a standalone service and is now being integrated in CloudWatch. X-Ray has standardized on OpenTelemetry and captures traces from a number of compute and database AWS services as well as apps written in Java, Node.js, and .NET.
- *Distributed tracing in Azure (http://mng.bz/zXWw)*—Part of Application Insights with SDKs for .NET, Java, and Node.js/JavaScript. Application Insights now also supports distributed tracing via OpenTelemetry.

- *Cloud Trace (https://cloud.google.com/trace)*—Google's offering for distributed tracing, which supports Go, Java, Node.js, and Python via OpenTelemetry.

Figure 2.4 provides an example output of Google Cloud Trace, a sophisticated distributed tracing backend. It shows the analytics of traces, specifically the latency distribution. Next, we will cover open source tracing solutions.

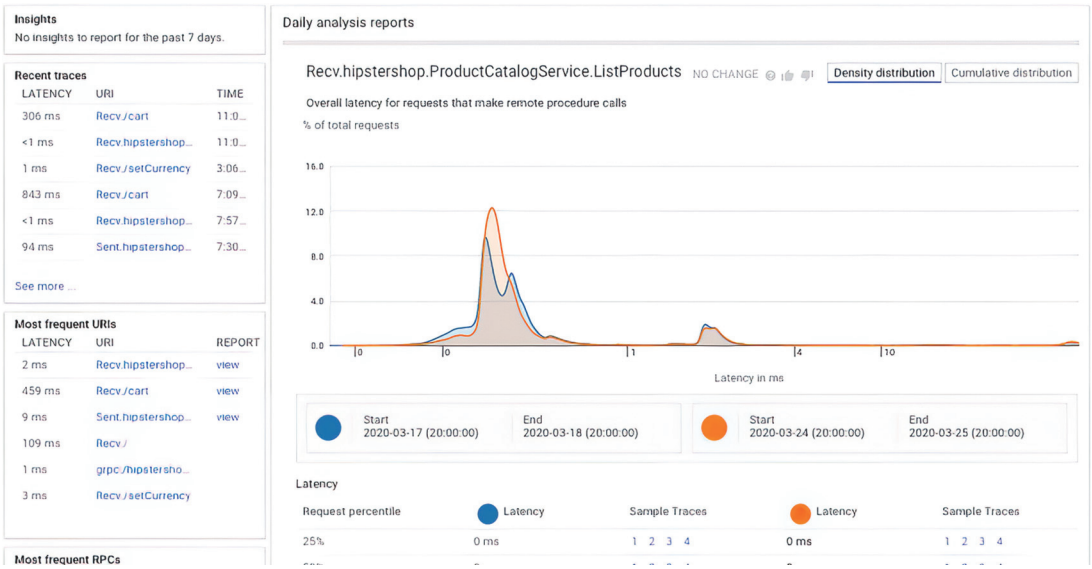


Figure 2.4 Viewing traces in Google Cloud Trace (Google Cloud docs; https://cloud.google.com/trace/docs/trace-overview)

2.4.2 Open source traces backends

Open source tracing solutions do not offer as many options as with metrics backends, but they still offer some choice:

- *CNCF Jaeger and Zipkin*—Both Jaeger (https://github.com/jaegertracing/jaeger) and Zipkin (https://github.com/openzipkin/zipkin) are distributed tracing systems hosted by CNCF. These days, Jaeger seems to be the more actively developed project, but Zipkin is still widely used and supported. I recommend you have a look at the respective communities (Slack and GitHub issues) and make up your own mind.
- *Elasticsearch and OpenSearch*—These options have expanded from logs to support handling traces. Both have excellent frontends (Kibana and Dashboards).
- *Grafana Tempo (https://grafana.com/oss/tempo/)*—A distributed tracing backend available under the AGPLv3 license. It uses object storage (e.g., S3) in the

persistence layer and is integrated with Grafana, Prometheus, and Loki. It supports Jaeger, Zipkin, and OpenTelemetry ingestion.

Now, let's move on to commercial offerings, which are, in contrast to open source offerings, very richly equipped. This starts at correlation features and ranges all the way to sampling and integrations.

2.4.3 Commercial offerings for trace backends

Originally called *application performance monitoring* (APM) vendors, nowadays, the tracing aspect (or *observability solution*) is often highlighted.

While distributed tracing hasn't yet reached the same level of adoption as metrics and logs, it's increasingly important for cloud-native setups. Even if you don't have a short-term strategy around it, consider testing it in smaller-scoped projects or products. Now, we'll shift gears to discussing a class of backends that helps address the aforementioned challenge of cardinality explosion: columnar data stores and formats.

2.5 Columnar data stores

There are many so-called columnar storage–based engines or simply data stores that you can use as backends for telemetry data. *Columnar* refers to the way the data is laid out in memory or on disk, boosting query performance.

Row-oriented databases, such as MySQL and PostgreSQL, keep all of the data associated with a record in close proximity in memory (or on disk). This means, in general, row-oriented DBs are a good fit for online transaction processing (OLTP) use cases, such as e-commerce, in which you have in-place updates of records. They are optimized for reading and writing rows, which is a great fit when you have many individual transactions with a potential for low data volume, such as is the case when inserting or updating single records in an online web application (e.g., putting a product into a shopping basket, following someone on a social media site, and so on).

Column-oriented or columnar data stores (see Mike Stonebraker et. al., "C-Store: A Column-oriented DBMS." Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005; see also https://people.brandeis.edu/~nga/papers/VLDB05.pdf) store the data of columns next to each other and are typically a good choice for so-called online analytical processing (OLAP) use cases. These are cases where you have few transactions but are dealing with a high volume of queries. The write path is the ingest of, say, time series data, and the read path is, for example, an ad hoc query over a time range. These columnar data stores are, hence, well suited for historical (time series) data where we want to extract actionable insights, typically in an interactive manner, from large datasets (many billions of rows and, potentially, petabytes of data) or, in other words, observability use cases. Let's have a closer look at this: figure 2.5 shows row-oriented versus column-oriented storage strategies.

Starting with the logical layout, a tabular view on columns (vertical) and rows (horizontal), a particular value has the coordinates `CiRj` (e.g., the value of the third column, second row would be referenced by `C2R1`). Now, when you are storing the

values row-oriented (the left-hand side of figure 2.5), you start in the first row and then move on to the second row, and so on, resulting in a storage layout (on disk or in memory) that stores the first row, the second row, and so on, until the last row. In contrast, when storing the values column-oriented (the right-hand side of figure 2.5), you start in the first column and then move on to the second column, and so on, resulting in a storage layout (on disk or in memory) that stores the first column, the second column, and so on, until the last column.

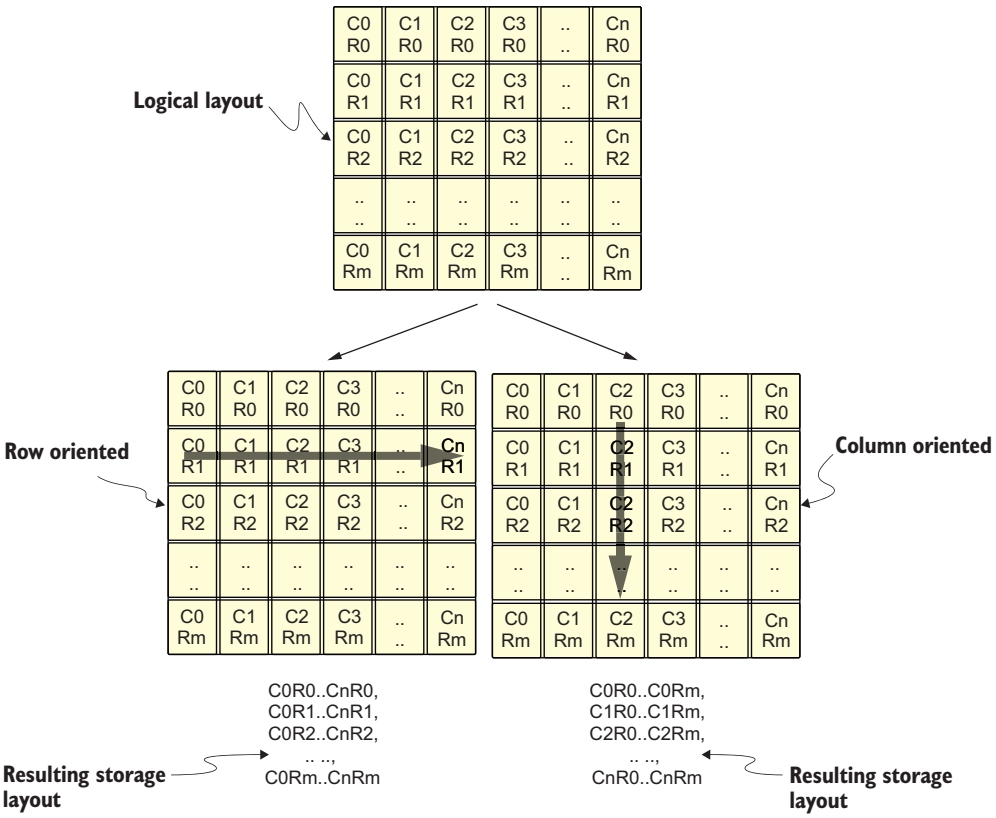


Figure 2.5 Row-oriented vs. column-oriented storage

You may be wondering, why bother with column-oriented storage? It turns out, columnar storage has a number of advantages for OLAP use cases, which includes our observability use cases.

In observability use cases, you can have many dimensions of interest, resulting in high cardinality and many columns. For any given query, there may only be a few columns relevant for a query. Columnar data stores allow for faster and fewer scans, since

fewer unnecessary values need to be read (that then are immediately discarded, as they don't contribute to an answer).

Further, column-oriented design allows you to apply storage optimization techniques, such as differential storage, like the XOR compression Facebook developed in the context of the Gorilla TSDB (see Tuomas Pelkonen et. al., "Gorilla: A Fast, Scalable, In-Memory Time Series Database." Proceedings of the VLDB Endowment, Vol. 8, No. 12, 2015; see also http://www.vldb.org/pvldb/vol8/p1816-teller.pdf) and run-length encoding (RLE; see D. Abadi, S. Madden, M. Ferreira, "Integrating Compression and Execution in Column-Oriented Database Systems." Proceedings of the 33th International Conference on Management of Data [SIGMOD]; https://doi.org/10.1145/1142473.1142548), yielding a smaller footprint and less resource usage.

In addition to compression, columnar design also enables vectorized execution (multiple data items per CPU cycle) and effective partitioning in the context of a single machine, and sharding a replication for scaling out, allowing you to answer queries over hundreds of millions of rows in subsecond time frames.

Prominent examples of columnar data stores, columnar formats, or services that use this strategy include the following:

- *Apache Cassandra (https://cassandra.apache.org/)*—Written in Java—a C++ variant, ScyllaDB, is also available (https://www.scylladb.com/).
- *Apache Druid (https://druid.apache.org/)*—An open source real-time analytics database with observability use cases by Airbnb, Alibaba, eBay, Lyft, and many more. You can run Druid in Kubernetes (http://mng.bz/0KDp).
- *Apache Pinot (https://pinot.apache.org/)*—An open source real-time distributed OLAP data store used by LinkedIn (http://mng.bz/KeOP), Uber, Slack, Stripe, and many more companies.
- *ClickHouse (https://clickhouse.com/)*—An open source columnar database, originally developed in 2008 by Yandex. You can use ClickHouse for all signal types, both for a single machine and in various distributed settings (sharded and replicated data with Apache Zookeeper as coordinator). When running it on Kubernetes, you should consider using the Altinity Operator for ClickHouse (https://github.com/Altinity/clickhouse-operator). ClickHouse has more-than-decent adoption, including Uber (logs), Cloudflare, and SigNoz, where it is being used as a building block in o11y solutions.
- *FrostDB (https://github.com/polarsignals/frostdb)*—A columnar database allowing for semistructured schemas, using Apache Parquet for storage as well as Apache Arrow for queries, developed as open source by Polar Signals.
- *Snowflake (https://www.snowflake.com/)*—An integrated platform delivered as a service. It features storage, compute, and global services layers that are physically separated but logically integrated.

If you're interested in learning more about the implementation details of some of the aforementioned data stores, I recommend reading the excellent article, "Comparison

of the Open Source OLAP Systems for Big Data: ClickHouse, Druid, and Pinot," by Roman Leventov (http://mng.bz/9DRx).

In this context, it's interesting to note that columnar storage is not limited to backends. For example, in OpenTelemetry there is ongoing work to introduce columnar encoding (https://github.com/open-telemetry/oteps/pull/171) in the collector and protocol, to speed up processing and improve performance and scalability.

Tip

The space of columnar data stores and databases, formats, and open source projects has exploded in the past decade. In fact, my first industry job (after some 12 years in research) was in the context of a Hadoop start-up on the now-retired Apache Drill project (which was similar to the open source version of Google's Dremel, which forms the basis of BigQuery).

There are many interesting data stores that you may be interested in, including MonetDB, Hypertable, Apache Kudu, and Greenplum. Covering them in any detail goes beyond the scope of this book. If you want to learn more, I recommend reading Martin Kleppmann's excellent book *Designing Data-Intensive Applications* (O'Reilly, 2017; https://dataintensive.net/) and *Fundamentals of Data Observability* (O'Reilly, 2023; http://mng.bz/jPQz) by the one and only Andy Petrella.

After that much theory, we deserve some action! So let's dive deep into an example with an open source columnar data store that gained a lot of traction in the past few years: ClickHouse.

In this walkthrough, you will (again) use the `hol1y` generator, configured to send its logs via the Fluentd logs driver to the OpenTelemetry collector with a pipeline using the Fluent Forward receiver (http://mng.bz/WzD4), and the ClickHouse exporter (http://mng.bz/8rZZ) to ingest the logs into ClickHouse, where you then will be able to query the logs using SQL.

Excited? Let's go!

We start off with the Docker Compose configuration shown in the next listing. You know most of the setup already from previous examples, so we will focus on the new parts of this setup.

Listing 2.1 Snippets of the Docker Compose configuration for the ClickHouse setup

```
hol1y:
    image: public.ecr.aws/mhausenblas/hol1y:stable
    ports:
      - 8765:8765
    logging:
      driver: fluentd
      options:
        fluentd-address: "localhost:8006"

  otel-collector:
    image: otel/opentelemetry-collector-contrib:0.55.0
```

Here, we provide the Docker Fluentd driver with the address of the fluentforward receiver (part of the OpenTelemetry collector pipeline).

```
    command: ["--config=/conf/otel-collector-config.yaml"]
    volumes:
    - ./otel-config.yaml:/conf/otel-collector-config.yaml
    ports:
    - "8006:8006"

  clickhouse-server:
    image: clickhouse/clickhouse-server
    ports:
    - "8123:8123"
    - "9000:9000"
    ulimits:
      nproc: 65535
      nofile:
```

We need to expose the Fluent Forward receiver port here so that we can receive log lines from Docker.

This is where we set up ClickHouse: the port 8123 for the Web UI and port 9000, the default native TCP endpoint for clients (allowing the OpenTelemetry exporter to talk to it).

There are several low-level Linux settings, such as the number of processes and files allowed for ClickHouse that we're setting here.

For more details on the ClickHouse configuration and good practices, see the ClickHouse operations docs (https://clickhouse.com/docs/en/operations/settings), specifically, the excellent article on ClickHouse Networking (http://mng.bz/EQyo). Now, we will move on to the telemetry agent setup. The following listing shows the OpenTelemetry collector configuration I put together for our ClickHouse example.

Listing 2.2 OpenTelemetry collector configuration for the ClickHouse setup

```
receivers:
  fluentforward:
    endpoint: 0.0.0.0:8006

processors:
  batch:
    timeout: 5s
    send_batch_size: 100

exporters:
  logging:
    loglevel: debug
  clickhouse:
    dsn: tcp://clickhouse-server:9000/default
    ttl_days: 3

service:
  telemetry:
    logs:
      level: debug
  pipelines:
    logs:
      receivers: [ fluentforward ]
      processors: [ batch ]
      exporters: [ logging, clickhouse ]
```

We configure the Fluent Forward receiver here to listen on all addresses on port 8006 for TCP connections. If all works, the Docker Fluentd driver will connect here.

Here, we set the batch size to 100 log lines. In practice, this would be much higher, likely in the thousands (depends on how long you want to wait until the logs are accessible in ClickHouse). I chose this low number to speed up things for you to see results.

This is the place where we set up the ClickHouse exporter, configuring it with the address of where the ClickHouse server is running. From the server perspective, our exporter is just another client talking TCP with it. The ttl_days setting limits the time data is kept around (not really important for our demo).

Finally, here, we assemble the pipeline using the components we configured previously.

Phew! That was some YAML wrangling again, so let's get that applied. In a new terminal session, go to `ch05/clickhouse/`, and then execute (self-contained, no need for an external load generator) the following:

```
docker-compose -f docker-compose.yaml up
```

Now you can go to http://localhost:8123/play, which is ClickHouse's playground, and enter the following SQL query (for reference, on my machine, it takes some milliseconds to read around 1,000 rows):

```
SELECT
  Timestamp as ts,
  Body AS payload,
  JSONExtractString(Body, 'traceID') AS traceID
FROM otel_logs
WHERE
  match(payload, 'invoked') AND
  ts >= NOW() - INTERVAL 30 SECOND
```

We pull out certain columns of interest, especially the associated trace ID.

We do a regex match against the body of the log message, requiring it to contain the string invoked.

We limit the output to the last 30 seconds.

The table the OpenTelemetry ClickHouse exporter set up for us to query.

The result of the SQL query execution is what you see in figure 2.6. Note the query stats in the right area above the output table.

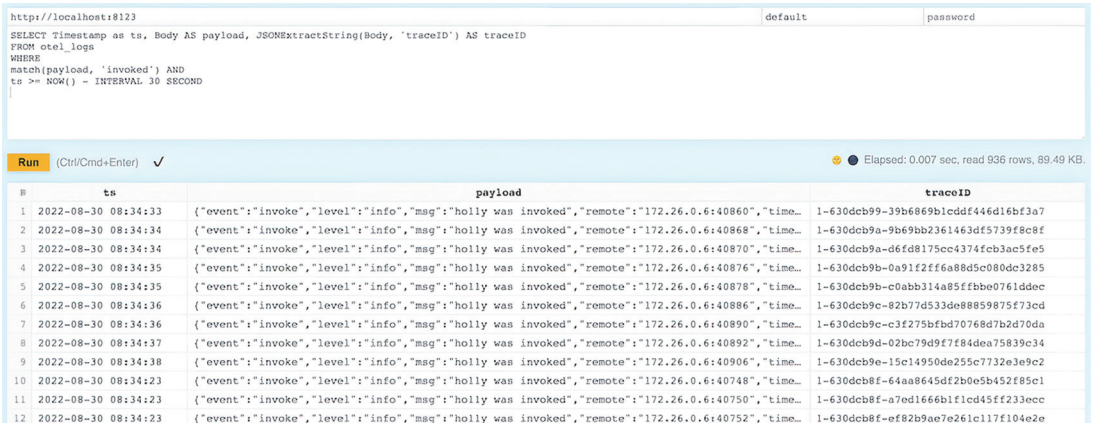


Figure 2.6 The ClickHouse Web UI playground, executing our SQL query, and showing the results

TIP If you find yourself working more with ClickHouse, then I'd recommend you check out ClickCat (https://github.com/clickcat-project/ClickCat), a nice and simple user interface that lets you search, explore, and visualize data in ClickHouse.

Another (related) category is distributed query engines, which can, among other things, use columnar data formats. These include

- PrestoDB (https://prestodb.io/), open sourced by Facebook (Meta)
- Cloud provider offerings, such as Amazon Redshift (https://aws.amazon.com/redshift/) and Google BigQuery (https://cloud.google.com/bigquery)

In the last part of this chapter, we shift gears and move to a hot topic: out of those many choices, how to best go about selecting one (or more) backends.

2.6 Selecting backend destinations

Selecting backends can be tough. To give you a feeling for the sheer number of options, have a look at the CNCF landscape in the Observability and Analysis (https://landscape.cncf.io/card-mode?category=observability-and-analysis) space. It sports many different options, from open source to SaaS.

Now that you have an idea about the spectrum of offerings in the backends domain, let's turn our attention to picking a backend that is a good fit for your needs. Working backward from your workload and your requirements is something I recommend. Also, be aware that it is not uncommon to end up using multiple types of backends in concert. For example, you might use CNCF Cortex for metrics LTS and federation along with OpenSearch for log and trace storage and query. Finally, remember that certain frontends, such as Grafana, as the canonical example, allow you to bring together signals from various backends, effectively building out a single pane of glass for your observability needs.

With backends, return on investment (ROI) is of central importance. Let's dive into the costs aspect first.

2.6.1 Costs

In the ROI context, backends require investment, typically in the mid- to long-term range—that is, years. This is due to the fact that data has gravity, meaning wherever the majority of your data resides defines your primary environment (think on-prem versus cloud or between different cloud providers). It costs time and money to transfer data between environments, and you want to avoid unnecessary egress and ingress data transfer costs.

More specifically, the monetary investments for backends can break down into

- *Ingestion costs*—An agent, like the OpenTelemetry collector, ingests (or writes) signals into the backend. For example, for a Prometheus-compatible backend, this may be measured in USD per million samples ingested.
- *Storage costs*—After ingestion, signals need to be stored for long-term retrieval. These costs are usually not the biggest contributor to your bill (most often ingestion or query is).

  Concerning storage, one often useful property is the "retention period." With this, your provider expresses how long your signal will be stored and is available

for answering queries. This can be a static or fixed property (e.g., one year) or something that you can configure (potentially, along with downsampling strategies in metrics backends, like CNCF Thanos already offers).

- *Query costs*—Using the signals to answer your o11y questions means you need to be able to look them up, scan them, and query them; these costs can be expressed in samples processed or can be more volume oriented (e.g., GB scanned). Understanding where and how query occurs (besides the maybe obvious ad hoc query you, as a human, perform interactively) is crucial.

  For example, think of the case in which you define an alert or alarm (compare chapter 3), and then a condition is evaluated in regular time periods, causing one or more queries to be executed. Again, at scale, this can sum up, and I have seen surprises (not the good kind) from customers not aware of this.

In addition to these variable costs, there can be flat costs, such as per-seat license costs; however, those are fairly predictable (in contrast to the variable costs, such as ingestions or query) and more commonly found with frontends or all-in-one solutions. One more consideration in the context of the costs of a backend—beyond the actual task at hand, like troubleshooting or performance optimization—is that there may be regulatory requirements to keep certain types of signals (usually logs) around for several years.

> **NOTE** Regulatory requirements around the storage and protection of telemetry signals is especially pertinent in regulated verticals, such as financial industries or health care. You may need to prove this capability in the context of achieving a certification, such as the case of an app that handles credit cards usually requiring Payment Card Industry Data Security Standard (PCI DSS) compliance or privacy requirements in the context of System and Organization Controls (SOC).

Here is one quick word on buy versus build in the context of backends: unless you are an observability vendor (or cloud provider), please consider not building your own backend from scratch. It can take many years to build something reliable and scalable, and if you have *N* innovation tokens or engineering cycles to spend, they are usually better invested in your core business and what sets you apart from the competition, rather than building the next time series data store.

I do not mean to discourage you from building an internal observability platform based on, for example, integrating various open source backends, with an open source frontend as a single pane of glass. However, you should be aware of the total cost of ownership (TCO), so be prepared to take the costs of your engineering team, support, training, and security patching (to name a few common ones) into account when you make a decision. Let's move on to the topic of open source and open standards next.

2.6.2 Open standards

Building on open standards is a good choice: try to avoid backends that are exclusively accessible via proprietary APIs, or at the very least, have a strategy in mind regarding how you can make these proprietary APIs comply with (industry) standards.

For example, in the context of our observability considerations, support for OpenTelemetry (at least on the road map) is something you want to include in your checklist. Every major cloud provider and ISV already now supports OpenTelemetry—see the respective vendor listing (https://opentelemetry.io/vendors/) on the project home page that details OpenTelemetry support (distributions and OTLP endpoints support).

Not all signal types are equal in this respect:

- Logs tend to have many open standards and formats in use. Standardization is not practical, and hence, standards like OpenTelemetry typically limit themselves to declaring normative mappings from and to established standards, like syslog.
- Metrics are increasingly represented in Prometheus exposition format or its successor, OpenMetrics (which sports exemplars support). Note, however, that there is a lot of old-school stuff, especially from or in non-cloud-native setups, that is still using things like Graphite format, the InfluxDB line protocol, or StatsD.
- Nowadays, it is important to ensure traces are OpenTelemetry compliant.

Next, we will look at a common challenge in the ingestion path.

2.6.3 Back pressure

When looking at ingesting signals at scale, you may come across the challenge of dealing with sources that generate, say, log lines or spans at a high rate. To deal with the back pressure—that is, not overloading the backends—you can use a queue between the source, agent, and backend. This allows you to decouple the producer side (signal source or agent) and consumer side (backend).

For example, you could use Apache Kafka (https://kafka.apache.org/) in the egress path of an OpenTelemetry collector configured with the Kafka Exporter (http://mng.bz/N2zN) component in a pipeline. In this configuration, the collector would serve as a short-term, in-memory queue publishing into a Kafka broker, and the backend could, for example, be an S3 bucket that gets populated on a schedule, via a lambda function. Next, we will cover how to manage too many values.

2.6.4 Cardinality and queries

In this section, we discuss two related problems when selecting a backend: cardinality and queries. We discussed cardinality explosion in the metrics backend section already. What we mean is that a metric's dimension (e.g., session ID or user ID)

could potentially take on highly varying values, causing issues when ingesting and storing the metrics (costs) as well as when querying them. In the context of queries, high cardinality might cause slow or hanging queries—that is, a query might time out before returning a result.

The best way to address this problem is to avoid high cardinality when using metrics backends. By avoiding it, I mean carefully controlling which metrics you scrape (or collect) and working backward from what you actually use. For example, your guidance could be that only metrics that are actively used in a dashboard or an alert are "allowed in." You can defer the decision by implementing this logic in the agent—for example, via an OpenTelemetry processor in one of your pipelines.

Alternatively, if you can't or don't want to limit yourself, you can use dedicated data stores that can handle a high volume of metrics, such as ClickHouse or commercial offerings. For example, Honeycomb initially spent a considerable time to build a proprietary data store for high-cardinality signals, from scratch. In their case, it made sense, since that's literally their business model. Again, this is a quick reminder of buy versus build in this context.

In general, query performance (how fast a backend can return results as a reaction to a query) is an important topic; however, details are beyond the scope of this chapter. It does have a time–speed tradeoff in some systems; for example, Prometheus and InfluxDB are fast performers if you're within their cardinality limits and are doing "alerting"- style queries. Distributed systems, on the other hand, tend to be slower in the near term but handle long-range queries (i.e., "tell me how we performed for the last quarter") far better.

If you're interested in setting up your own query performance TSDB benchmark, have a look, for example, at what Aliaksandr Valialkin shared in "High-Cardinality TSDB Benchmarks: VictoriaMetrics vs. TimescaleDB vs. InfluxDB" (http://mng.bz/D4mn).

Concerning the way queries are expressed as well as wire formats, some commonly used query languages, schemas, and formats are as follows (note that I intentionally mix up transport schema and query languages here to keep things simple):

- For the logs signal type:
  - Elastic Common Schema (ECS; http://mng.bz/lWy6) with the query syntax of Elastic-flavored Lucene
  - Grafana Labs' LogQL (https://grafana.com/docs/loki/latest/logql/)
  - CloudWatch Logs Insights query syntax (http://mng.bz/BmJ0)
  - The syslog protocol (https://datatracker.ietf.org/doc/rfc5424/), as per RFC 5424
  - Apache's Common Log Format (https://en.wikipedia.org/wiki/Common_Log_Format)
  - NGINX access and error log (https://docs.nginx.com/nginx/admin-guide/monitoring/logging)

  - Graylog Extended Log Format (GELF; https://go2docs.graylog.org/5-0/getting_in_log_data/gelf.html)
  - The Windows Event Log schema (http://mng.bz/d1AN), as a query format if you're using Windows Event Explorer
  - Common Event Format (CEF; https://docs.nxlog.co/userguide/integrate/cef-logging.html)
- In the time series databases realm or for the metrics signal type:
  - PromQL (http://mng.bz/qrGw), the Prometheus query language
  - The Prometheus exposition formats (http://mng.bz/rW7B) and OpenMetrics (https://openmetrics.io/)
  - InfluxData's Flux (https://www.influxdata.com/products/flux/), a standalone data scripting and query language
- For the traces signal type, there are no widely established query standards for traces yet, but Grafana Labs is working on this topic. Also, OpenTelemetry OTLP, as the default wire format, is widely adopted in this context.
- For signals of type profiles, `pprof` (https://github.com/google/pprof) is widely used.

Note that for most of the aforementioned signal types (profiles excluded, expected to be available in 2023), the OpenTelemetry OTLP (http://mng.bz/V1KP) is already, or will soon be, the standard way to perform data exchange. In most cases, you want to choose a backend that supports an (open) industry standard, such as OpenTelemetry or OpenMetrics, over proprietary (only) ones, but see also the previous section on open standards, regarding the expected caveats in this context.

With this selection guide, we've reached the end of the chapter on backends. Next, we will discuss frontends (e.g., Grafana) and all-in-one solutions, such as Apache SkyWalking.

Summary

- Backends are the source of truth for your observability questions.
- The backends store signals in a durable and long-term manner, supporting various query languages, typically declarative.
- There are various offerings for logs, metrics, and traces backends. You can choose between what cloud providers offer, open source (and run-yourself) options, and managed offerings from a wide range of providers.
- Some backends (e.g., OpenSearch in the OSS space and most commercial offerings from observability vendors) can handle more than one signal type; however, in practice, one can end up with different backends per signal type.
- Being able to move from one signal type in a backend to another signal type in another—potentially different—backend is crucial to answer arbitrary questions.
- Cardinality explosion is the challenge that a metric dimension can take on many different values (think user ID).

- TSDBs, such as Prometheus, are a great choice for the metrics signal type (numerical aggregates), until you run into cardinality explosion.
- Use columnar data stores and formats for high-cardinality signals and use cases.
- Make sure you have a good idea of what exactly you want to store (e.g., logs versus metrics).
- Due to data gravity, careful selection of your backends is crucial (to avoid large export–import costs when transferring data).
- Be aware of the direct and indirect costs in the ingestion, storage, and query phases when selecting a backend.
- Choose support for open standards concerning query languages and wire formats in backends over proprietary ones to minimize lock-in and maximize interoperability.

3 Cloud operations

This chapter covers

- How to manage incidents
- Health monitoring and alerts
- Governance and usage tracking

In this chapter, we will discuss an aspect of cloud-native solutions I call *cloud operations*, which spans several topics you will likely come across, especially in an operations role.

We start off with incidents: how to detect when something is not working the way that it should, react to abnormal behavior, and learn from previous mistakes. Then, we focus on alerts, or alarms (I'm using these terms interchangeably here, though an alert can be a triggered condition, and alarms can potentially cover multiple alerts)—that is, the automated process to check for a condition and inform someone responsible for a service or a piece of infrastructure about it. In the final part of this chapter, we talk about usage tracking, be that what your internal or external users access or the costs of the resources you're using to provide a cloud-native app.

Why can cloud operations be tricky? Well, in the context of cloud-native solutions, be that a workload running on Kubernetes or a bunch of lambda functions, what all of them have in common is many moving parts with potentially many different owners. So when end users of your cloud-native app are impacted—for example, by slower-than-usual response times or (partial) failures—the murder mystery starts: Which of the services along the request path contributed to the problem? Who owns it and, hence, should be notified to fix it?

Remember that for the end user, the fact that you're running, say, hundreds of microservices is unknown and, frankly, shouldn't matter. What they perceive is that something is not working, and it's your organization's responsibility to make sure you detect such issues—ideally, before you learn of them from your users—and have strategies in place not only to fix the issues as fast as you can but also learn from an issue to make sure it doesn't happen again. In this context, we can consider tracking the customer experience (not always clear who owns this) versus infrastructure tracking (usually clearly defined ownership). Let's start with incidents and how to detect and respond to them in a structured manner.

3.1 Incident management

In the context of this book, we'll consider two incidents: unexpected behavior of services or the platform that negatively affects its function or performance, and a repeat of unexpected behavior that is yet to have the learnings applied. For the latter case, the repeated unexpected behavior, it can be helpful to have a runbook of how previous incidents were resolved for handling the repeat situation.

Generally, it is best to have an incident response process in place. This incident response process is typically a set of procedures that define how you go about resolving and learning from an incident. You can think of it in terms of three phases:

1. First, you need to be able to detect that something is going wrong (health and performance monitoring).
2. Then, you need to be able to react to and handle an incident in a structured manner.
3. Last but not least, you need to learn from an incident. You want to avoid repeating a similar occurrence in the future (what action items should be completed to not have repeat occurrence of the same problem). We'll start by covering detecting an issue.

3.1.1 Health and performance monitoring

We previously discussed that the first step in resolving an incident is to be able to understand that something is broken in the first place. This may sound simple, but in a cloud-native system with many components, it is a nontrivial issue. In other words, manual approaches, such as visiting your website in a browser and checking if everything looks good, are not something you want to consider. Instead, aim to automate as many of the checks against your services and the "outside view"—that is, customer

experience (CX) as possible. In this context, it's crucial to emphasize this outside view; imagine a situation in which all your services report they are healthy and working as expected, but your end users can't access the app due to, say, a DNS issue.

The actual mechanisms you need to use to perform health and performance monitoring are really a spectrum, including the following:

- *Reachability monitoring*—Perform a simple call (ICMP, TCP, HTTP) to check for reachability or to ensure an endpoint returns a predefined status code.
- *Synthetic monitoring*—Execute scripts to simulate typical end user actions. These simulated end users are sometimes called *canaries* (a reference to the original use case, in which those little birds were used to detect gas leaks in mines). Several open source and commercial offerings allow you to perform synthetic monitoring—for example, Datadog (https://docs.datadoghq.com/synthetics/, Elastic (http://mng.bz/5wDO), New Relic (http://mng.bz/6Dle), and Amazon CloudWatch (http://mng.bz/o17Z). See figure 3.1 for an example.

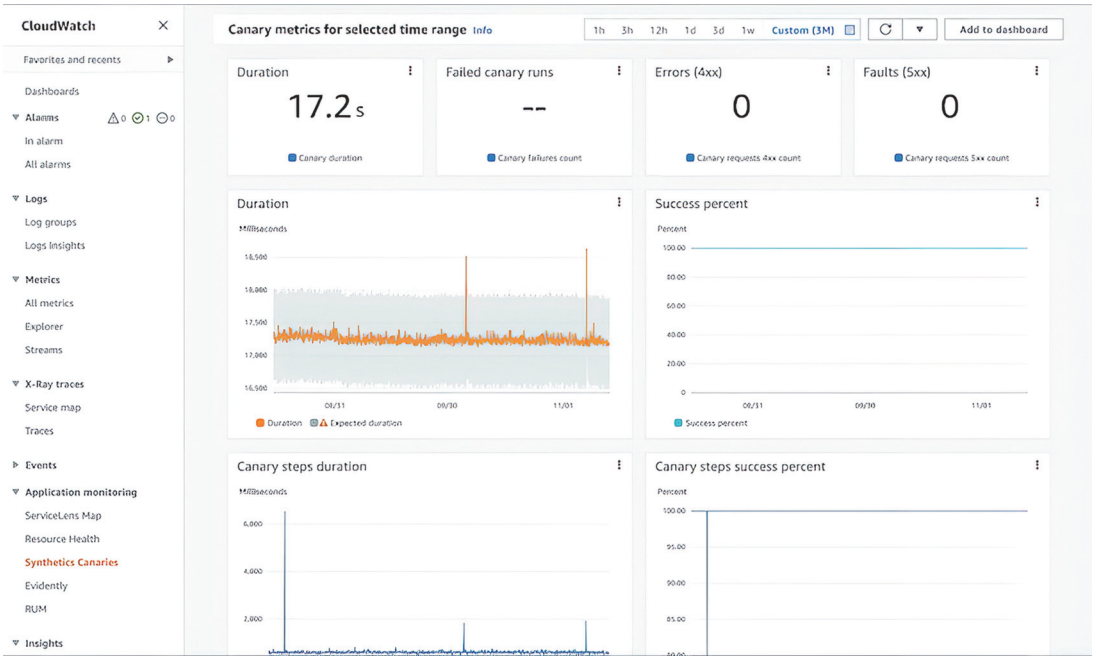


Figure 3.1 CloudWatch, an example of synthetic monitoring, tracking the availability of a website

You typically want to automate the process of checking for a condition and inform someone responsible about it, which is also known as *alerting* or *alarming*. There is a nice high-level article, "When to Alert on What?" (https://ali.sattari.me/posts/2023/

when-to-alert-on-what/), which is a useful article for getting started on the topic, ranging from manual to working with service-level objectives.

On a related note, how do you know if something is out of the normal? You need a baseline. You can determine the baseline (i.e., "How does the service or system look when everything is healthy?") manually by observing and noting things or by employing more sophisticated techniques, usually referred to as *anomaly detection* (https://www.reddit.com/r/sre/comments/zxdd2y/anomaly_detection/). With the detection basics out of the way, we now focus on how to deal with the incident after learning about it.

3.1.2 Handling the incident

Now that we're in a position to detect when something goes sideways in our cloud-native app by having health and performance monitoring in place, we move on to the question of how to handle an incident. Let's first discuss the "who" and then move on to the "what" and "how." In general, a human needs to be in the loop, assigned to handle the incident. This is oftentimes done via an on-call mechanism. For a certain time period, ranging from hours to a week or maybe even more, the person on call is notified by the monitoring system and needs to get on top of things as quickly as possible.

> **TIP** There are some offerings, sometimes called *AIOps*, that attempt to optimize the human-in-the-loop experience (e.g., Shoreline, https://www.shoreline.io/).

Nowadays, this notification is usually delivered via an app (and in parallel via email, Slack, MS Teams ping, phone call, etc.) and can come at any time of the day or night. You can probably imagine that if such a notification (or page) reaches you at, say, 2 a.m. and you have, say, 15 minutes to check in, you want to make sure you have a structured approach.

There are several things to consider when talking about on-call roles and models:

- *Traditional vs. on call*—In a traditional setting, where development and operations are strictly separated, the operations (ops) team is on call (technical responder). This can cause issues, since not only is the on-call setting demanding, but also, the person most familiar with a certain service is certainly not the ops person on call. Alternatively, one can establish a model where development and on-call are combined. For example, this is how AWS service teams work. This provides incentives for developers to properly instrument their code, enabling on-call to resolve issues as fast as possible.
- *Engineering and communication*—It goes without saying that to fix a technical issue, you need engineers on call. There is, however, another role that may be separated out: communication (incident manager or commander). On one hand, that can be internal communication (keeping stakeholders up to date), and on the other hand, it can refer to communication of the issue to customers. Someone needs to determine if customers are, indeed, impacted (versus

canaries); how many customers are impacted; and where they are impacted (if we're talking about a global system). Depending on the outcome, you might see an update on a more personalized level, such as the AWS Personal Health Dashboard (http://mng.bz/nW75), or a global view, relevant for all users, as with the GitHub Status (https://www.githubstatus.com/) page.

Regarding the *what* and *how*, the most important thing to remember is to stop the bleeding— that is, try to fix the problem and, ideally, gather data points along the way. The analysis (why it went wrong) comes later; don't let analysis slow down your efforts to stop the bleeding.

To support the person on call, the organization usually develops playbooks and runbooks (http://mng.bz/vn7p) that describe what to check during an incident and documents dependencies to understand downstream effects or possible sources of upstream trouble. Also, some teams practice the processes or material available (including coverage of runbooks) in something usually called a *game day* (http://mng.bz/4DNR).

3.1.3 Learning from the incident after the fact

Let's assume the on-call team did a good job and fixed the problem. That isn't the end of the story. You want to have a process in place that looks at the incident and allows you to achieve two things: understand what the underlying issue was and ensure you don't repeat the same mistake.

I recommend following the path of a blameless postmortem (http://mng.bz/QPO6); in AWS, we call these *corrections of error* (COE; https://wa.aws.amazon.com/wat.concept.coe.en.html). These sessions help you to determine what caused an issue and how to avoid it in the future, typically employing the five whys technique (https://www.lucidchart.com/blog/5-whys-analysis). You may also come across the term *root cause analysis* (RCA; https://www.techtarget.com/searchitoperations/definition/root-cause-analysis), which refers to testing various theories about what originally caused the problem with empirical data.

After this theory-heavy section, let's move on to how to handle notifications for on-call roles—in other words, *alerting*.

3.2 Alerting

In this section, we start by looking at alerting in the Prometheus ecosystem. We then move on to alerting with Grafana, and finally, we discuss alerting offerings of the cloud providers and beyond.

3.2.1 Prometheus alerting

We talked about Prometheus and saw it in action in chapter 2. Now that we have a more complete picture in terms of instrumentation and backends, let's see how Prometheus handles alerting.

In the context of Prometheus, alerting comprises two distinct parts:

- *Prometheus itself*—You define what an alert is. This effectively means you define conditions using PromQL, and Prometheus evaluates them periodically. Once a condition is met, Prometheus considers this to be the alert "firing" and sends it to the Alertmanager API.
- *Alertmanager API*—This provides an API that enables clients (typically Prometheus) to send in alerts and takes care of the notifications. For example, you could define a policy that sends a notification for certain types of alerts via email or implement a policy such as *during work hours, send the notification to Slack; after work, send a text message*. But that's not the only thing the Alertmanager does: it deduplicates alerts and allows you to group and suppress notifications to reduce alert fatigue and lessen alert storms.

Figure 3.2 shows how Prometheus and Alertmanager work together to handle alerts. The scrape targets (e.g., your instrumented applications or Prometheus exporters, such as the Node exporter) expose metrics in the Prometheus exposition format, which Prometheus scrapes (in other words, it calls the endpoints, which are by default expected to be at /metrics). If you have alerting rules defined in Prometheus, Prometheus continuously evaluates them and calls the Alertmanager in case the defined threshold is reached. The Alertmanager, in turn, processes alerts with the goal of sending out as few and concise notifications as possible. Via the configured receivers in the Alertmanager, it delivers a notification to one or more destinations, where the operator (a human) finally consumes the notification and acts upon it accordingly.

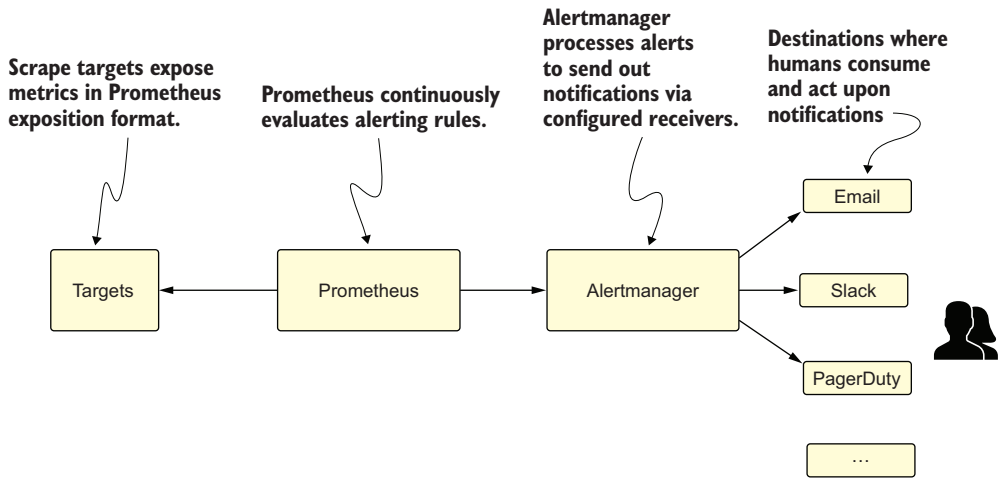


Figure 3.2 Prometheus and the Alertmanager

The separation of Prometheus and the Alertmanager, as shown in figure 3.2, allows for flexibility and scalability. For example, you can scale or restart either of the tools without interrupting the other.

> **TIP** From an operations perspective, at the end of the day, you want to avoid alert fatigue (https://en.wikipedia.org/wiki/Alarm_fatigue). That is, you want to minimize the notifications to act on while making sure you don't miss something substantial. Imagine the case in which you're responsible for a Kubernetes cluster with multiple namespaces. If there are data plane problems with the nodes not being reachable or rebooting, you only want to get one notification, rather than, say, one notification for every single pod in the cluster.

Now, let's dive deeper into how alerting rules and the Alertmanager work.

PROMETHEUS ALERTING RULES

Starting with Prometheus, we now work our way through defining and handling alerts. In the Prometheus configuration, you define the alerts using alerting rules (http://mng.bz/XNj9). An alerting rule defines the condition you want Prometheus to evaluate. A concrete example looks as follows (adapted from the excellent reference site Awesome Prometheus Alerts (https://awesome-prometheus-alerts.grep.to/), which provides a large collection of curated alert examples you can use):

The name of the alert. The convention is CamelCase.

The condition to evaluate—a PromQL expression Prometheus executes

Optionally, you can specify for how long the condition needs to be met before sending the alert to the Alertmanager.

Key–value metadata for filtering and routing in the Alertmanager

Key–value context for direct human consumption (content of the notification) using templating

```
- alert: PrometheusAllTargetsMissing
  expr: sum by (job) (up) == 0
  for: 0m
  labels:
    severity: critical
  annotations:
    summary: Prometheus all targets missing {{ $labels.instance }}
    description: "A Prometheus job has no alive targets.\n
                  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
```

While the `for` field is optional, you should consider setting it. The reason is that, in cloud-native systems, you're dealing with a lot of ephemeral components, network connectivity issues, and timeouts or retries. You don't want a trigger-happy alert spamming you while, really, it was just a temporary issue with the telemetry, for example. Let's say we would have chosen `for: 3m` in the preceding example. In this case, Prometheus initially considers the alert in the pending state when the condition is met for the first time. The alert is considered to be in the firing state and sent to the Alertmanager if the condition persists for 3 minutes. In the preceding snippet, we used `for: 0m`, and because of this, the alert went directly to firing.

Concerning the `annotations` field, you want to provide rich context for the human operator receiving a notification based on the alert. Consider that the notification may reach the person on call at an early hour, and the last thing you want in this situation is to start digging for references, wasting valuable time to stop the bleeding, while still trying to wake up. For example, you could, in addition to the `summary` and `description` fields we defined in the preceding example, add the following:

- A link to a dashboard that shows the time series for the relevant time frame
- An assessment of who or what is impacted (whether these are real customers or canaries)
- A link to a runbook, for known issues and contributing factors. If you can follow these links from your phone without having to log in to, say, eight systems to get them to render, you've done a good job.

Now that we know what alerting rules are and how to define them, let's see how to set up Prometheus to include the alerting rules and establish the connection to an Alertmanager. This is done as follows in the Prometheus configuration file prometheus.yml:

```
global:
  scrape_interval: 30s
  external_labels:
    environment: dev
    region: us-west-1
rule_files:
  - alert.rules
alerting:
  alertmanagers:
  - static_configs:
    - targets: ["alertmanager.local:9093"]
scrape_configs:
  ...
```

Adds external labels to provide context for the Prometheus instance

Points to the file where the alerting rules are defined

Configures the Alertmanager Prometheus should use

ALERTMANAGER

Let's now move on to the Alertmanager (http://mng.bz/yQ7p). First, we have a look at how the Alertmanager processes alerts. This is done in the so-called notification pipeline, consisting of the following phases:

1. *Inhibition*—Suppress notifications for alerts if other (related) alerts are already firing. For example, think of the preceding case, where you don't want pod-level notifications when you're already notified of a cluster data plane issue.
2. *Silencing*—Suppress notifications ad hoc, in a proactive manner. For example, if you're upgrading nodes in an Kubernetes cluster, you already know in advance that nodes will go down temporarily, so you don't want to get a notification for this.
3. *Routing*—Decide who gets a notification.
4. *Grouping and throttling*—Ensure you get one notification per event.

5 *Notification*—Using so-called receivers, define how the notification should be delivered, such as via email, Slack, PagerDuty, or Amazon Simple Notification Service (SNS; https://aws.amazon.com/sns/).

An example config (http://mng.bz/MBaQ) for the Alertmanager should give you some idea of how you can configure the notification pipeline, but let's have a look at a concrete end-to-end example to fully appreciate how things work.

In this example, we implement a simple Prometheus self-monitoring case. The goal is to receive a notification once a certain number of Prometheus API calls is exceeded.

As usual, we need to stand up a Prometheus and an Alertmanager instance using the Docker Compose file in listing 3.1, which you launch (from `ch07/prometheus/`) with the following command:

```
docker-compose -f docker-compose.yaml up
```

In the Docker Compose file, we pass on the environment variables and the configuration files.

Listing 3.1 The Docker Compose configuration for the Prometheus alerting setup

```
version: "3.5"
services:

  prometheus:                          # Configures Prometheus with alerts and Alertmanager
    image: prom/prometheus:latest
    ports:
    - "9090:9090"
    volumes:
    - ./prom-config.yaml:/etc/prometheus/prometheus.yml
    - ./alerting.yaml:/etc/prometheus/alert.rules

  alertmanager:                        # Configures Alertmanager with grouping and notification
    image: prom/alertmanager:latest
    volumes:
    - ./am-config.yaml:/etc/alertmanager/alertmanager.yml
    ports:
    - "9093:9093"

  webhook:                             # Sets up a generic webhook the Alertmanager can call
    image: kennethreitz/httpbin
    ports:
    - "9999:80"
```

The next listing shows the complete Prometheus configuration in prometheus.yml, defining where to find alerts and connecting to the Alertmanager.

Listing 3.2 The Prometheus config

```
global:
  scrape_interval: 5s
```

```
  external_labels:
    env: "docker"
rule_files:
- alert.rules
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      - "alertmanager:9093"
scrape_configs:
  - job_name: "prometheus"
    static_configs:
    - targets: ["localhost:9090"]
```

References where the alerting rule is defined

Configures the Alertmanager

Defines what to scrape (in this case, Prometheus self-scrape)

Our alert is defined in alert.rules, shown in the following listing. We want to fire the alert once the total number of HTTP requests Prometheus sees exceeds a certain number (40, in our example).

Listing 3.3 Alerting rule

```
groups:
- name: alert.rules
  rules:
  - alert: TooManyAPICalls
    expr: sum(prometheus_http_requests_total) > 40
    labels:
      severity: "critical"
    annotations:
      summary: "Too many API calls"
      description: "The Prometheus instance {{ $labels.instance }} is
     experiencing too
      many API calls"
```

The name of our alerting rule

The PromQL expression Prometheus evaluates

Using a Prometheus label to communicate how important the alert is

The Alertmanager config alertmanager.yml is depicted next.

Listing 3.4 Alertmanager configuration

```
route:
  group_by: ["env"]
  receiver: "mywebhook"
receivers:
- name: "mywebhook"
  webhook_configs:
  - url: "http://webhook:80/anything"
```

The grouping definition. We want all alerts with the env label grouped together.

Defining the receiver (how to send the notification) for this route

Configures the receiver. In this case, we want the notification via a webhook.

If you go to http://localhost:9090, you should see the Prometheus UI. You can find the alerting rules and their status, as shown in figure 3.3, via the top-level Alerts menu item. If you want to speed up things, you need to perform a few queries (e.g., using the PromQL query `sum(prometheus_http_requests_total)`) or, otherwise, wait a few minutes until you see the alert firing.

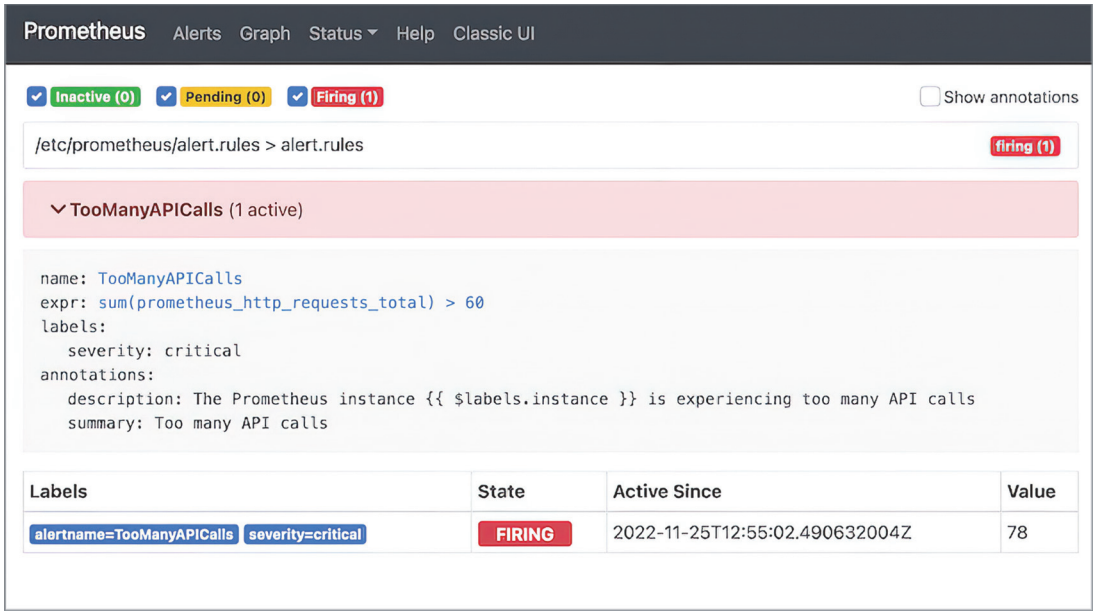


Figure 3.3 Screenshot of firing alert in Prometheus

Now, you should see in Prometheus that our alert is firing; hence, the Alertmanager is invoked. So once you see the alert `TooManyAPICalls` firing in Prometheus, head over to http://localhost:9093, where you should see the alert (figure 3.4) once you filter by `env=docker`.

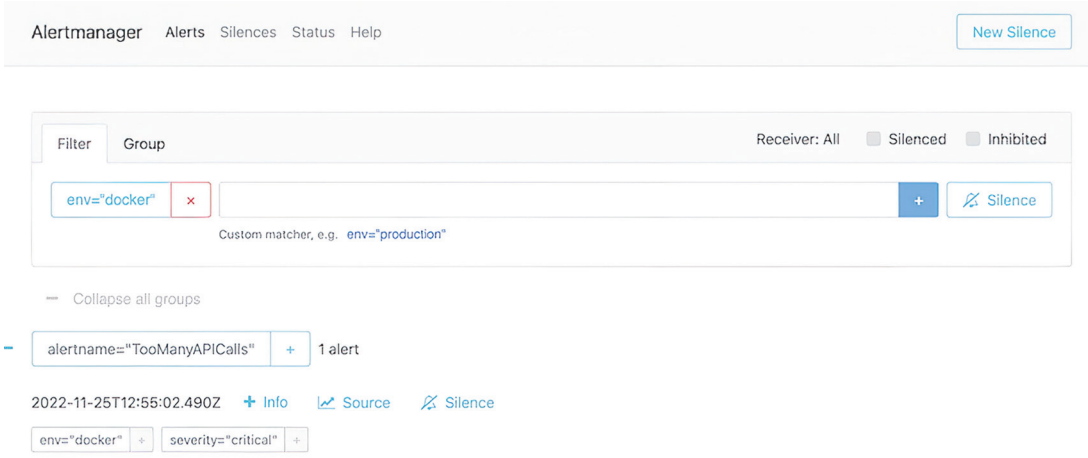


Figure 3.4 Screenshot of an alert being processed in Alertmanager

Figure 3.5 shows the notification you would have received if you had configured an email receiver rather than the generic webhook. It contains the labels as defined in the alerting rule and the external label `env`.

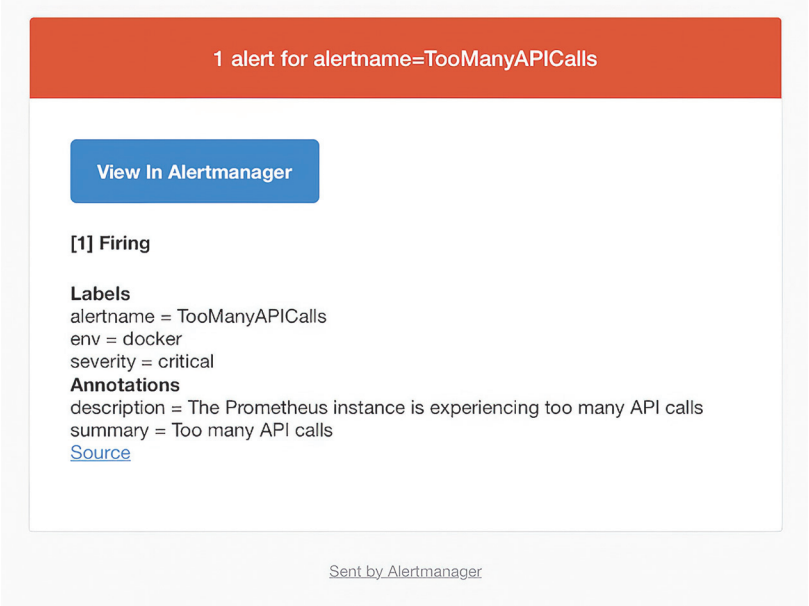


Figure 3.5 Screenshot of the notification email

> **TIP** The Alertmanager offers a number of excellent customizations that you can use to make your notifications more actionable and helpful (e.g., notification templates, http://mng.bz/a1rj).

With this, we've completed the Prometheus alerting end-to-end example. The concepts discussed here also apply to other tooling in the Prometheus ecosystem, such as the CNCF projects Cortex (http://mng.bz/gB2e) and Thanos (http://mng.bz/e1Z9); both support Alertmanager-compatible alerting rules. This makes it easy to migrate to a federated setup, effectively allowing you to reuse all the configuration.

If you want to dive deeper into the topic of alerting with Prometheus, here are a few resources I can recommend:

- *Prometheus Up and Running* by Brian Brazil (2018, O'Reilly, http://mng.bz/OxeK)
- "Improved Alerting With Prometheus and Alertmanager" by Julien Pivotto (2019, PromCon talk, http://mng.bz/Y1Wo)
- "Life of an Alert" by Stuart Nelson (2018, PromCon talk, https://www.youtube.com/watch?v=PUdjca23Qa4)

Next, we will have a quick look at alerting with Grafana, enabling you to manage alerts and notifications beyond Prometheus.

3.2.2 Using Grafana for alerting

Let's have a brief look at Grafana's alerting capabilities. Up until version 8, Grafana supported a dashboard-centric alerting model. This means an alert was confined to a dashboard, making it hard to manage notification policies. For example, answering the question, "What sends alarms via PagerDuty to a certain destination?" would require manually checking all dashboards. We will focus on unified alerting (http://mng.bz/GyvM), available from Grafana version 9 and beyond.

First off, why would you want to use Grafana for alerting? In general, your use case covers multiple backends (chapter 2). That is, imagine you are using Prometheus to store metrics and OpenSearch to store logs and traces and want to define alerts across these backends in a central place.

With unified alerting, Grafana now supports these use cases with the following:

- There is support for a range of data sources, including Amazon CloudWatch, Azure Monitor, Google Cloud Monitoring, InfluxDB, Jaeger, and Prometheus.
- You can organize alerts by group (similar to what we've seen in Prometheus).
- There is support for different Alertmanagers, including an integration with the Prometheus one.
- For notifications (http://mng.bz/zX7w), you use so-called contact points, supporting a wide array of types (https://grafana.com/docs/grafana/v9.5/alerting/fundamentals/contact-points/), including Discord, Email, Kafka, PagerDuty, Slack, Telegram, and WebHook.
- You can suppress notifications using silences. Note that the alert rule still gets evaluated by Grafana, causing potential costs for calling the backend, but the notification does not get created.

Now, let's move on to alerting solutions from cloud providers.

3.2.3 Cloud providers

Every major cloud provider offers an alerting solution, typically deeply integrated with its offerings. These offerings are a great baseline you should always consider, since they cover the cloud provider's APIs. These include

- Amazon CloudWatch alarms (http://mng.bz/0Kjp), which uses Amazon SNS to send email or SMS notifications
- Azure Monitor Alerts (http://mng.bz/KeyP), which supports email, SMS, and push notifications
- Google cloud alerting (https://cloud.google.com/monitoring/alerts/), supporting a range of notification types, from email to Pub/Sub

As previously mentioned, you could use these alerting solutions also from Grafana (which has a CloudWatch data source available if you want to control alerts from that pane), providing a single pane of glass. We're switching gears now for a quick look at end user tracking as well as resource usage tracking.

3.3 Usage tracking

Alerting is by and large a reactive process. This means something is happening, and based on this, someone gets notified, reacting to the event. In this section, we cover a different cloud operations technique that you can use for a variety of cases, from troubleshooting to security to planning. I'm talking about tracking users and what they access. Let's start with end users of your cloud-native app.

3.3.1 Users

To understand how your end users use your app, you need to have a way to track their activity within your app. This allows you to identify popular features and potential UX issues. Over the years, the term *real user monitoring* (RUM; https://github.com/open-telemetry/oteps/issues/169) has been popularized for this. For example, there is, for web applications, the widely used Google Analytics (https://analytics.google.com/analytics/web/), and you can use CloudWatch RUM (http://mng.bz/9Drx) in AWS. All these offerings require the up-front embedding of some small JavaScript snippet in your web pages to see the usage information (demographics, region, device types, usage paths, etc.) in a dedicated place. If you'd like to dive deeper into the topic, I recommend perusing the excellent article, "An Introduction to Real User Monitoring (RUM): Monitoring and Observability," by Chinmay Gaikwad (http://mng.bz/jPJz).

> **NOTE** One thing to be aware of with RUM is the rise of single-page application (SPA; https://en.wikipedia.org/wiki/Single-page_application) websites. In the context of SPAs, the traditional metrics, such as page load time, no longer matter (or make sense), since the page only loads once, at the beginning of a session, and the UI updates are effectively local renderings. For more on this topic, check out episode 60 of the O11ycast, "Customer-Centric Observability With Todd Gardner and Winston Hearn" (http://mng.bz/Wz94), as well as the article, "Observable Frontends: the State of OpenTelemetry in the Browser," by Jessica Kerr (http://mng.bz/8rdZ).

RUM is also something the CNCF OpenTelemetry project considers; the OpenTelemetry Enhancement Proposal OTEP 169 (http://mng.bz/EQoo) covers this by proposing a new data model and semantics, supporting the collection and export of RUM telemetry, a new signal type, alongside traces, metrics, logs, and profiles. While still in its early days, you can expect more development in this space, especially since the logs signal type stabilized in June 2023 (meaning you can use OpenTelemetry to handle logs in prod).

While RUM focuses on end users, there is another user-tracking category: internal users, including developers and operators. For example, you might want to have a record of the actions taken by a user or a service account for compliance or risk auditing reasons. An example of this kind of service is AWS CloudTrail (http://mng.bz/N2gN), supporting a long list of services, including Amazon API Gateway, AWS CloudFormation, Amazon CloudWatch, Amazon EKS, AWS Lambda, and AWS Systems Manager.

You can also set up alarms from the aforementioned internal-user audit logs. With that said, let's now move on to costs.

3.3.2 Costs

A totally different category of tracking access to resources is cost tracking. No matter if you want to educate internal users about costs and usage (showback) or offer services that require charging certain tenets (chargeback), you want to have an idea of what the actual costs are on as fine-grained a level as possible. We can distinguish between metering, which is counting the access (e.g., API endpoint XYZ was called 3,478 times in the last 15 minutes), and the actual charges that can depend on many things, from upfront commitments to special deals you may be able to negotiate based on the volume you are consuming.

Cloud providers typically offer services that allow you to (interactively) explore your usage (e.g., AWS Cost and Usage Reports [CUR], http://mng.bz/D46n). In addition to these native offerings, you may wish to opt into using solutions that provide for cost allocation and resource usage monitoring down to single units of execution, like OpenCost (https://www.opencost.io/) does for Kubernetes environments.

While most of the topics we've discussed are primarily important to understand and owned by folks in operator roles, with cloud-native systems, it's crucial that the people in every role involved, from developers to testers to release managers, have a solid grasp of the concepts. This can, in the case of a fully serverless environment, mean that, indeed, developers themselves are on call (http://mng.bz/lW76).

Summary

- To successfully operate cloud-native applications, you need to have a structured approach to detecting, handling, and learning from incidents.
- Health and performance monitoring allows you to have a system-external view and learn that something is not working as expected.
- If you're on call for a service and an incident happens, focus on stopping the bleeding; the analysis of what went wrong comes later, ideally in a blameless postmortem.
- The goal of alerting is for a human to automatically be notified of an event impacting the system.
- You want to avoid alert fatigue and, hence, need policies in place that allow you to receive a meaningful number of notifications (but not too many).

- Prometheus allows you to define alerting rules, and the grouping, filtering, suppression, and notification is handled by a separate component: the Alertmanager.
- If you want to centrally manage alerts across various backends, you can use Grafana.
- Usage tracking means capturing end user interactions in the app or capturing access to system resources as well as costs.