Chapter 1

What Is Observability?

In modern software development and operations, observability has emerged as a fundamental concept essential for maintaining and improving the performance, reliability, and scalability of complex systems. But what exactly is observability? At its core, observability is the practice of gaining insights into the internal states and behaviors of systems through the collection, analysis, and visualization of telemetry data. Unlike traditional monitoring, which primarily focuses on predefined metrics and thresholds, observability offers a more comprehensive and dynamic approach, enabling teams to proactively detect, diagnose, and resolve issues.

This chapter will explore the principles and components of observability, highlighting its significance in today's distributed and microservices-based architectures. Through a deep dive into the three pillars of observability—metrics, logs, and traces—you will understand the groundwork for how observability can transform the way resilient systems are built and managed.

IN THIS CHAPTER, YOU WILL LEARN TO:

- Differentiate between monitoring and observability
- Explain the importance of metadata
- Identify the differences between telemetry signals
- Distinguish between instrumentation and data collection
- Analyze the requirements for choosing an observability platform

Definition

So, what is observability in the realm of modern software development and operations? While many definitions exist, they all generally refer to observability providing the ability to quickly identify availability and performance problems, regardless of whether they have been experienced before, and help perform problem isolation, root cause analysis, and remediation. Because observability is about making it easier to understand complex systems and address unperceived issues, often referred to in the software industry as *unknown unknowns*,[1] the data collected must be correlated across different telemetry types and be rich enough and immediately accessible to answer questions during a live incident.

The Cloud Native Computing Foundation (CNCF), described more fully later in this chapter, provides a definition for the term *observability*:[2]

> *Observability is a system property that defines the degree to which the system can generate actionable insights. It allows users to understand a system's state from these external outputs and take (corrective) action.*
>
> *Computer systems are measured by observing low-level signals such as CPU time, memory, disk space, and higher-level and business signals, including API response times, errors, transactions per second, etc. These observable systems are observed (or monitored) through specialized tools, so-called observability tools. A list of these tools can be viewed in the Cloud Native Landscape's observability section.*[3]
>
> *Observable systems yield meaningful, actionable data to their operators, allowing them to achieve favorable outcomes (faster incident response, increased developer productivity) and less toil and downtime.*
>
> *Consequently, the observability of a system will significantly impact its operating and development costs.*

While the CNCF's definition is good, it is missing a few critical aspects:

- The goal of observability should be where a system's state can be *fully understood* from its external output *without the need to ship code*. This means you should be able to ask *novel questions* about your observability data, especially questions you had *not* thought of beforehand.
- Observability is not just about collecting data but about collecting *meaningful data*, such as data with context and correlated across different sources, and storing it on a platform that offers rich analytics and query capabilities *across signals*.
- A system is truly observable when you can troubleshoot *without prior knowledge of the system*.

The OpenTelemetry project, which will be introduced in Chapter 2, "Introducing OpenTelemetry!," provides a definition of observability that is worth highlighting:

> *Observability lets you understand a system from the outside, by letting us ask questions about that system without knowing its inner workings. Furthermore, it allows you to easily troubleshoot and handle novel problems—that is, "unknown unknowns." It also helps you answer the question, "Why is this happening?"*
>
> *To ask those questions about your system, your application must be properly instrumented. That is, the application code must emit signals such as traces, metrics, and logs. An application is properly instrumented when developers don't need to add more instrumentation to troubleshoot an issue, because they have all of the information they need.*[4]

In short, observability is about collecting critical telemetry data with relevant context and using that data to quickly determine your systems' behavior and health. Observability goes beyond mere monitoring by enabling a proactive and comprehensive understanding of system behavior, facilitating quicker detection, diagnosis, and resolution of issues. This capability is crucial in today's fast-paced, microservices-driven, distributed environments, where the

complexity and dynamic nature of systems demand robust and flexible observability solutions. Through the lens of the CNCF and OpenTelemetry, you can see observability is not just defined as a set of tools and practices but as a fundamental shift toward more resilient, reliable, and efficient system management.

Real World Scenario

Riley Joins Jupiterian

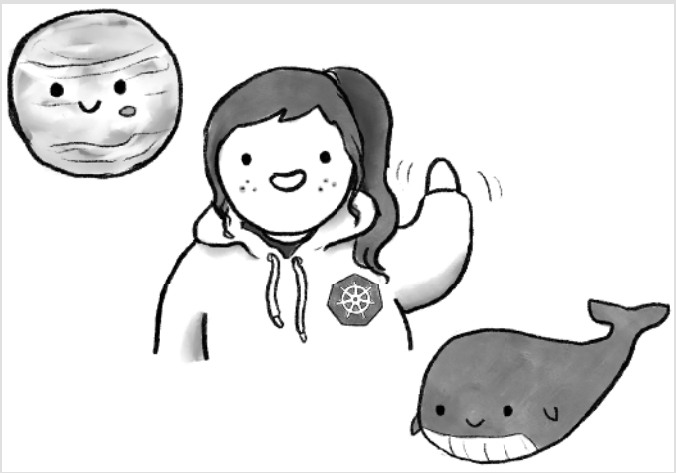
Riley (she/her) is an experienced site reliability engineer (SRE) with deep observability and operations experience. She recently joined Jupiterian to address their observability problems and work with a new vendor. Riley joined Jupiterian from a large private equity (PE) advertising company, where she was the technical lead of the SRE team and was responsible for a large-scale, globally distributed, cloud native architecture. Before that, she was the founding member of a growth startup where she developed observability practices and culture while helping scale the business to over three million dollars in annual recurring revenue (ARR). Riley was excited about the challenge and opportunity of building observability practices from the ground up at a public enterprise company transitioning to the cloud.

Jupiterian is an e-commerce company that has been around for more than two decades. Over the last five years, the company has seen a massive influx of customers and has been on a journey to modernize its tech stack to keep up with demand and the competition. As part of these changes, it has been migrating from its on-premises monolithic application to a microservices-based architecture running on Kubernetes (K8s) and deployed in the cloud. Recently, outages have been plaguing the new architecture—a problem threatening the company and one that needed to be resolved before the annual peak traffic expected during the upcoming holiday season.

For the original architecture, the company had been using Zabbix, an open source monitoring solution to monitor the environment. The IT team was beginning to learn about DevOps practices and had set up Prometheus for the new architecture. Given organizational constraints and priorities, they did not have the time to develop the skill set to manage it and the ever-increasing number of collected metrics. In short, a critical piece of the new architecture was without ownership. On top of this, engineering teams continued to add data, dashboards, and alerts without defined standards or processes. Not surprisingly, this resulted in the company having difficulty proactively identifying availability and performance issues. It also resulted in various observability issues, including Prometheus availability, blind spots, and alert storms. In terms of observability, the company frequently experienced infrastructure issues and could not tell if it was because of an architecture limitation or an improper use of the new infrastructure. As a result, engineers feared going on-call, and innovation velocity was significantly below average.

The Jupiterian engineering team had been pushing management to invest more in observability and SRE. Instead, head count remained flat, and the product roadmaps, driven primarily by the sales team, continued to take priority. With the service missing its service-level agreement (SLA) target for the last three months, leadership demanded a focus on resiliency. To address the problem, the Chief Technology Officer (CTO) signed a three-year deal with Watchwhale, an observability vendor, so the company could focus on its core intellectual property (IP) instead of managing third-party software. An architect in the office of the CTO vetted the vendor and its technology. Given other organizational priorities, the engineering team was largely uninvolved in the proof of

concept (PoC). The Vice President (VP) of Engineering was tasked with ensuring the service's SLA was consistently hit ahead of the holiday period as well as the adoption and success of the Watchwhale product. He allocated one of his budget IDs (BIDs) for a senior SRE position, which led to Riley being hired.



Background

The term *observability* has been around since at least the mid-20th century and is mainly credited to Rudolf E. Kálmán, a Hungarian American engineer who used it in a paper about control theory.[5] Since then, the term has been used in various fields, including quantum mechanics, physics, statistics, and perhaps most recently, software development. Kálmán's definition of observability can be summarized as a measure of how well the internal states of a system can be inferred from knowledge of its external outputs.[6]

Observability Abbreviation

Observability is often abbreviated as O11y (the letter *O*, the number *11*, and the letter *y*), as there are 11 characters between the letter *O* and the letter *y*. While it is the number 11, the ones are pronounced as the letter *l*—thus, the abbreviation is pronounced *Ollie*. This abbreviation standard is common for longer words in software. For example, Kubernetes, a popular cloud native open source project, is often referred to as K8s and pronounced *kay-ates* for the same reason.

Cloud Native Era

In software, the term *observability* has become popular due to the rise of cloud native workloads. Since the turn of the century, the software industry has seen a progression that has included

moving from bare metal machines to virtual machines (VMs) to containers. In addition, there has been a shift from owning, deploying, and managing hardware to leasing data center equipment to deploying in the cloud. But what does *cloud native* mean? One way to answer this question is to look to the CNCF. The foundation is part of the Linux Foundation and defines itself as:

> *The open source, vendor-neutral hub of cloud native computing, hosting projects like Kubernetes and Prometheus to make cloud native universal and sustainable.*[7]

Perhaps not surprisingly, the CNCF has created a definition for the term *cloud native*:

> *Cloud native practices empower organizations to develop, build, and deploy workloads in computing environments (public, private, hybrid cloud) to meet their organizational needs at scale in a programmatic and repeatable manner*. They are characterized by loosely coupled systems that interoperate in a manner that is secure, resilient, manageable, sustainable, and observable.
>
> *Cloud native technologies and architectures typically consist of some combination of containers, service meshes, multi-tenancy, microservices, immutable infrastructure, serverless, and declarative APIs—this list is non-exhaustive.*

Monitoring Compared to Observability

Before the cloud native era, it was common to see patterns including on-premises software, monoliths, separate development and operations teams, and waterfall software development with long release cycles. In this prior generation, the term *observability* had not been adopted yet, and instead, the term *monitoring* was used. Sometimes, these terms are used interchangeably, but their meanings are not identical. The Merriam-Webster dictionary defines monitoring as the ability "to watch, keep track of, or check usually for a special purpose."[8] It defines observability as the ability "to come to realize or know especially through consideration of noted facts."[9] The distinction between monitoring and observability is important. With monitoring, you track items but must infer why something occurred or how it is related to another event. With observability, you use information to prove facts and use that knowledge to determine how or why something behaves the way it does. Observability allows for first principle thinking, or the ability to validate assumptions not deduced from another assumption.[10]

In software, both observability and monitoring rely on specific data types—primarily metrics and logs with some tracing—but the usage of the data differs. Before the cloud native era, most software ran on-premises and was often developed and deployed as a monolith or single code base or application. As a result, problem isolation, or where the problem originated, was easy to identify when issues occurred, and scaling typically consisted of adding more resources to the monolith, known as *scaling up* or *scaling vertically*. When issues arose, the problem was either the monolith, the infrastructure the monolith was running on top of, or whatever application was calling into or called by the monolith (see Figure 1.1). To monitor the monolith, operational teams needed the ability to be alerted about specific, known symptoms, sometimes referred to as *known knowns*. Monitoring systems did exactly that.

To provide monitoring, either your application needs to be instrumented to emit health data or you are required to infer the health of the application by watching its external behavior. In either case, the data collected needs to be able to track and answer questions about availability, performance, and security. This data collection needs to be added before issues happen; otherwise, you cannot proactively determine nor quickly resolve the problems as they arise.

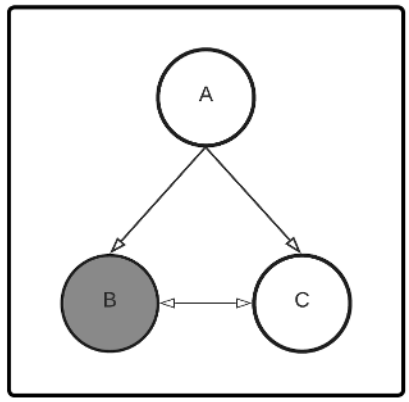


Figure 1.1
An example of a monolithic application experiencing an issue. The square represents the monolith, while the circles represent different functions or features within the monolith. In this example, the B function is experiencing problems, denoted by the service's gray shading. This may or may not result in issues with the A and C functions.

Types of Monitoring

There are two different types of monitoring. First, there is monitoring based on data exposed from the internals of the system. This means the application makes specific data available for external systems to gather. This type of monitoring is sometimes called *white box monitoring* because you can see into the system,[11] though a better name would be *internally provided monitoring*. Second, there is monitoring based on external behavior. This means the application does not make any data available beyond what is required for the application to function. As such, an external system must infer what an application is doing. This type of monitoring is sometimes called *black box monitoring* because you cannot see into the system,[12] though a better name would be *externally provided monitoring*.[13]

In many cases, application developers add instrumentation as necessary, including to measure performance and investigate issues during development and operations. Engineers responsible for monitoring the health and performance of these applications would typically send telemetry data to a monitoring platform. Based on this telemetry data, the engineer would then define alerts with static thresholds. To determine these thresholds, an engineer would need to know what problems to expect beforehand, thus enabling *proactive monitoring*; otherwise, new thresholds would have to be defined after an issue is identified, which is known as *reactive monitoring*. One way to think about monitoring is like a doctor who collects certain pieces of information from a person and compares that data against known baselines to understand the symptoms being experienced and to determine the health of the person. The monitoring of heart rate, blood pressure, and temperature in humans is like the monitoring of CPU (central processing unit), memory, and disk usage in applications.

While monitoring with static thresholds provides some awareness of potential system issues, it is not without its limitations. Take, for example, CPU utilization, which represents the rate at which an application is operating expressed as a percentage. If CPU utilization is very high, this could be a symptom of a system issue and, as such, something you want to be notified about. For example, you could define an alert when the CPU utilization exceeds 95 percent for some period

of time. In fact, such a definition is common in traditional monitoring applications. The problem is, such an alert may not indicate a problem but instead indicate that the application is using its resources efficiently. What is missing from this symptom is context, including how other related components are behaving, and correlation, including changes within the environment. Another limitation of traditional monitoring tools is the difficulty in alerting on issues that do not manifest as high resource consumption or latency.

The introduction of cloud native workloads made traditional monitoring even less effective. In this new world, workloads are run in the cloud and often consist of many small applications, called *microservices*, that are isolated to individual functionality. For example, an authentication service or a notification service. Microservices make it easier to deploy more instances, known as *scaling out* or *scaling horizontally*, and allow for specific components to be scaled as needed. These microservices typically run on immutable infrastructure using declarative APIs (application programming interfaces). In addition, they are run with DevOps practices and with the help of site reliability engineers (SREs).[14] Software release cycles are also more frequent and leverage continuous integration and continuous deployment or CI/CD pipelines. The decoupling and elasticity of applications enable developers to reduce duplicated efforts and scale to meet demand, but often at the cost of being able to troubleshoot the system and keep it available. In this era, it is the "unknown unknowns" that need to be addressed.

Due to the difficulty in troubleshooting microservice-based architectures, a popular meme was shared throughout the community:

> *"We replaced our monolith with micro services so that every outage could be more like a murder mystery." @honest_update*[15]

With cloud native workloads, problem isolation became a problem. This is because when one microservice has an issue, it could impact upstream or downstream services, causing them to have problems as well (see Figure 1.2). Using traditional monitoring, the net result is alert storms and the need to investigate every issue on every service in order to get to the root cause and remediation. Of course, there are other issues with cloud native workloads as well. For example, there is an inability to have complete visibility into the infrastructure as it is being managed by a third party and prone to dynamic changes.

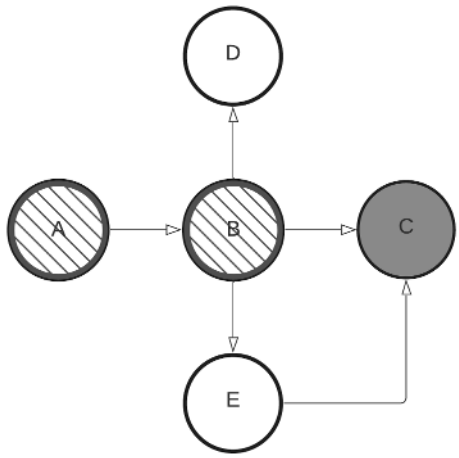


Figure 1.2
An example of a microservice-based architecture experiencing an issue. Each circle represents a different microservice. In this example, multiple microservices are experiencing an issue, denoted with gray lines, though one service is the root cause of the problem, denoted with gray shading. Note not all services called by the root cause service are impacted.

Going back to the doctor analogy, assume you have a large group of people who are all part of the same community, and multiple people become sick around the same time. While you may want to help everyone experiencing symptoms concurrently, it requires many doctors and resources. In addition, focusing on the symptoms of the patients does not address the root cause issue, which is that people are getting sick from something, and it is spreading instead of being contained. The sickness may cause other problems to arise as well. For example, doctors may become sick and thus become unable to care for patients, or businesses might need to shut down because they do not have enough employees to work. Without containment, an infectious disease can spread uncontrollably. This analogy is similar to the changes necessary due to the shift to cloud native workloads. For example, instead of paging all service owners during an outage and burning out engineers, the more sustainable approach is to contain the problem and page the root cause service. Observability helps with containment.

When dealing with complex systems, it is ideal when you can address things you are aware of and understand, referred to as *known knowns*, as well as things you are not aware of and do not understand, referred to as *unknown unknowns*, using the same solution. See Table 1.1 for different states of awareness and understanding. A goal of observability is to provide the ability to answer the "unknown unknowns." At the same time, it contains the building blocks necessary to address the "known knowns." As a result, observability may be considered a superset of monitoring.

Table 1.1: A 2×2 matrix showing states of awareness and understanding. Monitoring systems are optimized to address "known knowns" where observability systems can address all aspects but especially "unknown unknowns."

		Awareness Known	Unknown
Understanding	**Known**	Aware of and understand	Aware of but do not understand
	Unknown	Understand but not aware of	Neither aware of nor understand

Metadata

When you hear the term *observability*, you may initially think about data sources such as metrics, logs, and traces. These terms will be introduced in the next section, but something just as important as the data source information is metadata. While a fancy word, metadata is just data about other data. For example, if you generate and collect a metric, such as the total number of HTTP requests, it may also be helpful to know other information about that metric, such as which host it is running on or what HTTP status code was returned for that request. These additional pieces of information are known as *metadata* and are typically attached to traditional data source information, such as metrics, logs, and traces. Metadata may go by other names as well, including tags, labels, attributes, and resources.

Metadata is powerful because it provides additional information to data sources, which helps with problem isolation and remediation. This information may even contain context and correlation, topics explored in Chapter 8, "The Power of Context and Correlation." Without

metadata, observability is harder to achieve. Metadata is typically represented as a key-value pair, such as `foo="bar"`. The *key* is the name for the piece of metadata and is often referred to as a *dimension*. The *value* can be of various forms, including numbers or strings and the uniqueness of the values is referred to as *cardinality*. Other ways to represent metadata also exist. For example, in unstructured log records, metadata is sometimes presented as just a value where the name is inferred—an example is provided in the "Logs" section later in this chapter.

Dimensionality

In observability, *Dimensionality* refers to the number of unique *keys* (sometimes called *names*) within a set. It is represented by attributes or labels associated with telemetry data, allowing for more granular and detailed analysis. Each piece of telemetry data can have multiple dimensions that provide context about the data. These dimensions enable the grouping, filtering, and slicing of data along various axes, which is crucial for deep analysis and troubleshooting. Examples of dimensions include:

- Time
- Application, such as `service.name` and `service.version`
- Host, such as `host.name` and `host.arch`
- User, such as `enduser.id` and `enduser.role`
- HTTP, such as `http.route` and `http.response.status_code`

These dimensions would allow you to ask your telemetry to show you data such as:

- All `502` errors in the last half hour for host `foo`
- All `403` requests against the `/export` endpoint made by user `bar`

Dimensionality, which may also be referred to as the width of telemetry data, is a foundational concept in observability that greatly enhances the depth and utility of telemetry data. It matters because it enables more detailed, contextualized, and actionable insights, which are essential for maintaining and improving the performance and reliability of modern distributed systems. In practice, dimensions are indexed by observability platforms to support capabilities, including auto-complete and real-time analysis of key-value pairs, that assist with troubleshooting.

Cardinality

In observability, *Cardinality* refers to the number of unique values for a given key within a set. *High cardinality* refers to a large number of unique values, whereas *low cardinality* indicates fewer unique values. For example, a dimension like HTTP status code, which includes values such as `404` or `500`, is bounded and has low cardinality, whereas a dimension like a user, session, or transaction ID is unbounded and is likely to have high cardinality. Monitoring and observability platforms care about cardinality. For example, if a platform supports indexing of keys, it likely needs to return values for those indexed keys quickly. For high cardinality metadata, this can prove challenging to visualize and very expensive to compute. In short, cardinality affects the storage, performance, and usability of telemetry data. High cardinality presents both opportunities for detailed insights and challenges in terms of resource consumption and data management. Effectively managing cardinality is essential for maintaining scalable, efficient, and actionable observability systems.

Semantic Conventions

Another concept you should be aware of is *semantic conventions*, or semconvs for short. These are standardized dimensions, or keys, for metadata and ensure consistency in how data is recorded, labeled, and interpreted across different systems and services. It may also contain standardized cardinality, or values for these dimensions. For example, it is common to have semconvs for HTTP-related data. An example of this may include the key for the HTTP route, such as `http.route`, or the response status code, such as `http.response.status_code`. Semconvs can be grouped into multiple different categories, such as the aforementioned HTTP. Other categories would include databases, exceptions, host metrics, function as a service, and messaging, to name a few. Each category would have multiple semconvs defined. Semconvs may be signal specific or apply to more than one signal type. Semconvs matter because they enable context and correlation and provide data portability. For example, if the same key is used to represent the same data, then it is easy to see its behavior across systems and environments. In addition, if keys are consistently named, they can be leveraged identically across different platforms.

Data Sensitivity

Metadata can contain sensitive information. For example, names or email addresses may be attached to data sources and leak personally identifiable information (PII). In addition, internal business logic, such as Internet protocol (IP) addresses or hostnames, may be considered sensitive information. This information would generally only be sent to the configured observability platforms, but that configuration can change over time. In addition, while only restricted users may have access to the platform data, for example, employees of a company authenticated via Security Assertion Markup Language (SAML), without proper data permissions, such as role-based access control (RBAC), it is possible that sensitive information is exposed to employees who should not have access to such information. Given that metadata can contain virtually anything, care must be taken to ensure proper data configuration, scrubbing, and access control.

Signals

The *three pillars of observability* is an industry phrase that you have likely encountered. The three pillars refer to metrics, logs, and traces. While these pillars are just data sources and do not inherently provide observability, they are recognized as fundamental types of telemetry data needed to understand the behavior and performance of systems. Another comparable term or acronym in the observability space is *MELT*, which stands for metrics, events, logs, and traces. These are the most common data sources, but they are far from exhaustive. Other examples include profiling and sessions. Data sources have a variety of names in the industry, including diagnostics, telemetry, signals, or data sources. For the purposes of this book, and in alignment with OpenTelemetry, the term *signals* will be used going forward. It is important to note that signals do not inherently provide observability, though they are necessary to enable it.

Metrics

A *metric*, sometimes referred to as a *metric record*, *measurement*, or *metric time series* (MTS), is a set of data points represented as a time series with metadata. A *time series* is a set of data points over

some period of time. To generate a time series, an instrument takes one or more measurements. For example, a speedometer measures speed, and a measurement could be taken every tenth of a second but recorded every minute. Metrics also have signal-specific metadata terms. For example, *attributes*, *dimensions*, *labels*, and *resources* are all terms used with metrics that refer to some kind of metadata.

A metric contains a name, value, timestamp, and optionally additional metadata. Note that multiple types of metric values exist. For example, it may be a single value, such as a counter, or a multi-value, such as a histogram. Here is an example of a metric from Prometheus, an open source metric solution that will be described in more detail later:

```
http_requests_total{method="post",code="200"}     1027     |  1395066363000
```

The example Prometheus metric is made up of various components, including:

- Name—`http_requests_total`
- Metadata—`{method="post",code="200"}`
- Value—`1027`
- Timestamp—`1395066363000`

Metrics are one of the primary data sources used to engage on-call engineers as well as troubleshoot availability and performance issues. It is pervasive for alerts and dashboards to be configured based on metric data. Generally, aggregated metrics, like those shown in Figure 1.3, provide the most value because they identify behaviors over time and can be used to determine anomalies. Some popular methods for analyzing aggregated metrics include:

- RED, which stands for requests, error, and duration and was popularized by Tom Wilkie.[16] The idea is for every object to monitor the number of requests, the number of those requests that result in an error, and the amount of time those requests take. In general, this information can be used to determine user experience.
- USE, which stands for utilization, saturation, and errors and was popularized by Brendan Gregg.[17] The idea is for every object to monitor the percentage of time the object was busy, the amount of work (queue size) for the object, and the number of errors. In general, this information can be used to determine object experience.
- Four golden signals, which include latency, traffic, errors, and saturation and was popularized by the Google SRE Handbook.[18] The idea is for every object to monitor the time it takes to service a request, the amount of demand placed on the object, the rate of requests that fail, and the fullness of the object. This is like RED but includes saturation.

In addition, metrics are used to define service-level indicators (SLIs) that measure the performance of applications. These SLIs are used to define and measure service-level objectives (SLOs) which determine whether applications are operating within acceptable bounds. Service-level agreements (SLAs) are also defined and calculated based on metrics to determine whether applications are meeting specified customer expectations.

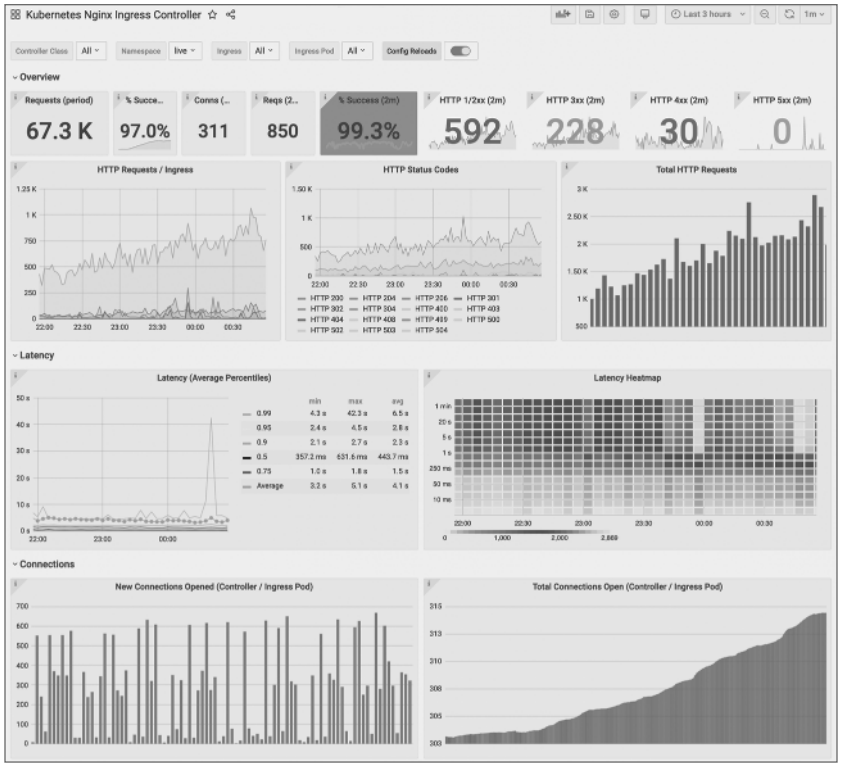


Figure 1.3
A Grafana dashboard displaying aggregate metric information.

Learning More About SLIs, SLOs, and SLAs

SLIs, SLOs, and SLAs are critical topics that are outside the scope of this book. If you are looking to learn more about these concepts, be sure to read the *Google Site Reliability Engineering (SRE)* book, which is freely available online.[19]

Given the ever increasing number of objects in an environment and the need to collect more and more data, metric platforms need to be able to process and store a large number of metrics quickly. Various techniques are used to control the amount of data generated, processed, and stored. For example, the interval at which metrics are generated within the application or stored within an observability platform can be different from the resolution displayed in charts. Aggregation techniques are used to achieve these different granularities, including aggregation policies and rollups. In short, these strategies provide a summarized view of granular data over specific time intervals. Regardless of the techniques used, end users consume charts or alerts from this collected, analyzed, and queried data.

Several open source metric instrumentation frameworks and standards have become popular over the years. For example, the following solutions were popular in the monitoring era:

- StatsD (`https://github.com/statsd/statsd`)
- Graphite (`https://graphite.readthedocs.io/en/stable/overview.html`)

- Nagios (`https://www.nagios.org`)
- Telegraf (`https://www.influxdata.com/time-series-platform/telegraf`)
- Zabbix (`https://www.zabbix.com`)

In the observability era, the following projects have gained popularity:

- Grafana (`https://grafana.com/grafana`)
- OpenTelemetry (`https://opentelemetry.io`)
- Prometheus (`https://prometheus.io`)
- M3 (`https://m3db.io`)

Programming languages also have their own frameworks that can be leveraged to generate metric data. For example:

- Java Management Extensions (JMX) (`https://en.wikipedia.org/wiki/Java_Management_Extensions`) for Java
- `System.Diagnostics.Metrics` for .NET

There are also a variety of open source third-party frameworks, such as Micrometer (`https://micrometer.io`) for Java.

Logs

A *log*, sometimes called a *log record*, is a time-based event with metadata. A log typically contains a timestamp, severity, message, and optionally additional metadata. Logs have signal-specific metadata terms. For example, *attributes*, *fields*, and *resources* are all terms used with logs that refer to some kind of metadata.

A log can be either:

- Structured, meaning the message and other components are stated in a known regular syntax, making them easily recognizable and parsable. Structured logs are becoming the standard in cloud native workloads. Here is an example of a structured log in JavaScript Object Notation (JSON) format:

```
"@timestamp":"2024-07-01T10:07:13.425Z", "log.level": "INFO", "message":
  "Tomcat started on port(s): 8080 (http) with context path ''" "service.name":
  "springpetclinic","process.thread.name":"restartedMain","log.logger":
  "org.springframework.boot.web.embedded.tomcat.TomcatWebServer"}
```

or

- Unstructured, meaning the message is a string that could contain almost anything and whose metadata may be irregular and inconsistent requiring the parsed format to be dynamically inferred and often incomplete. Unstructured logs were more common before cloud native workloads, but they are still present given the broad adoption of unstructured syslog from legacy systems. Here is an example of an unstructured syslog message:

```
212.87.37.154 - - [01/Jul/2024:10:07:13 +0000] "GET /favicon.ico HTTP/1.1" 200
  3638 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36
  (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36"
```

In the preceding example, the `200` is an example of metadata where the name is missing and must be inferred. The value represents the HTTP response status code.

In addition to the format, logs can also be of different types. For example, events may be thought of as a unique signal but could also be thought of as a specific kind of log. Events contain the same information as a log record and could be structured or unstructured. What makes events unique is that they indicate that something happened. For example, a deployment to production could be denoted as an event and used to determine whether key performance indicators (KPIs) changed because of the deployment. Aligned with OpenTelemetry, events will be treated as a subtype of log records throughout this book.

Individual logs can be helpful in determining the root cause of issues as well as for security use cases, including providing an audit log of changes. Logs can also contain metric data. If properly parsed, aggregate data extracted from logs can be helpful to determine the health and behavior of a system. In either case, log payloads are significantly larger than metrics, and as such, more data needs to be processed and stored.

It is common for log data required for security purposes to have requirements about collecting all the data for a minimal period of time and even guaranteeing the data is not lost through, for example, a disk-based queue. In addition, keeping logs that identify root causes is also essential. As a result, collecting all logs or at least all logs at a certain severity level is common. *Severity* is a term used in logging to determine the type of logs to collect. The general severity levels are `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, and `FATAL`. Care should be taken when deciding which severity level to collect, as improper log collection can impact observability.

Logs have developed open standards over time thanks in part to syslog. Multiple Requests for Comments (RFCs)[20] have been created for syslog, including:

- RFC 3164 (`https://datatracker.ietf.org/doc/html/rfc3164`)
- RFC 5424 (`https://datatracker.ietf.org/doc/html/rfc5424`)
- RFC 5425 (`https://datatracker.ietf.org/doc/html/rfc5425`)
- RFC 5426 (`https://datatracker.ietf.org/doc/html/rfc5426`)
- RFC 6587 (`https://datatracker.ietf.org/doc/html/rfc6587`)

While these open standards helped define protocol and data model support, they were created in response to proprietary and commercial solutions. In turn, several open source solutions emerged, and commercial solutions were forced to support these open standards. Of course, other open standards exist beyond the aforementioned standards. Most notable are the OpenTelemetry Logs Data Model[21] and the Elastic Common Schema.[22] Additional open standards and their importance will be covered in Chapter 2.

Traces

A *trace* is a recording of a time-based transaction or end-to-end request with metadata. A trace contains a unique identifier (ID), a start and end time, one or more spans, and optionally additional metadata. A *span* is a single step in a transaction and typically captures service or function calls, making it easy to follow the progression of a transaction. It contains similar information to a trace, including a unique ID, two timestamps (start and end), and optionally additional metadata. In addition, spans contain a parent ID, which is null for the first span,

known as the *root span*. Spans have signal-specific metadata terms. For example, *attributes*, *tags*, and *resources* are all terms used with spans that refer to some kind of metadata.

Next, an example of a simplified trace (some metadata has been removed) is shown in JSON format. This example was produced using the OpenTelemetry protocol exporter. You will learn more about the OpenTelemetry protocol exporter and use the payload shown next in Chapter 5, "Managing the OpenTelemetry Collector." You can either create a file named `otlphttp-trace.json` and save the content now or access it from the book's GitHub repository when you reach Chapter 5.

```
{
  "resourceSpans": [
    {
      "resource": {
        "attributes": [
          {
            "key": "host.name",
            "value": { "stringValue": "web01.sflanders.net" }
          },
          {
            "key": "service.name",
            "value": { "stringValue": "frontend" }
          }
        ]
      },
      "scopeSpans": [
        {
          "scope": {
            "name": "io.opentelemetry.armeria-1.3",
            "version": "2.3.0-alpha"
          },
          "spans": [
            {
              "traceId": "80dcfdf0f3fe1a032e53facf9ae2ceca",
              "spanId": "f041a77c36ad23ae",
              "parentSpanId": "f4426a2e1f99e8a0",
              "flags": 1,
              "name": "GET /",
              "kind": 2,
              "startTimeUnixNano": "1718649061103266000",
              "endTimeUnixNano": "1718649061105462875",
              "attributes": [
                {
                  "key": "http.response.status_code",
                  "value": { "intValue": "200" }
                },
                {
                  "key": "http.request.method",
                  "value": { "stringValue": "GET" }
                },
```

(continued)


```
            {
              "key": "http.route",
              "value": { "stringValue": "/" }
            }
          ],
          "status": {}
        },
        {
          "traceId": "80dcfdf0f3fe1a032e53facf9ae2ceca",
          "spanId": "f4426a2e1f99e8a0",
          "parentSpanId": "",
          "flags": 1,
          "name": "GET",
          "kind": 3,
          "startTimeUnixNano": "1718649061098436000",
          "endTimeUnixNano": "1718649061105925208",
          "attributes": [
            {
              "key": "http.response.status_code",
              "value": { "intValue": "200" }
            },
            {
              "key": "http.request.method",
              "value": { "stringValue": "GET" }
            }
          ],
          "status": {}
        }
      ]
    }
  ],
  "schemaUrl": "https://opentelemetry.io/schemas/1.24.0"
 }
 ]
}
```


Looking at an example of a trace, you may notice that it looks like a structured log. Traces are like multiple structured logs that have been stitched together. The significant difference between a trace and a log is that a trace natively provides context and correlation. Context and correlation are possible by creating and passing an ID through a transaction. In fact, traces require that consistent context propagation be used end-to-end. This means all services in a transaction need to be instrumented and configured to use the same context propagation mechanism, and context must be allowed to flow between all services. Passing context, especially between disparate systems across various protocols, is difficult. As a result, trace instrumentation is not as prevalent as metrics and logs today.

While context passing is generally done via HTTP headers, the creation process is defined by the standard implemented. Several standards exist, with W_3C Trace Context[23] being the standard for cloud native workloads. Other standards also exist, including B3 from the open source Zipkin

project[24] and Amazon's trace ID for its proprietary X-Ray service[25]. The same trace ID standard must be used in order for tracing to work properly, and HTTP headers must be allowed to pass between all services. As a result, the decision on the context format is important.

Context Propagation Format Recommendation

In order for tracing to work properly, all services in a transaction must use the same context propagation format. If different context propagation formats are used across services then traces will be broken or disconnected. While continuing to use an existing context propagation format is acceptable, using W_3C Trace Context in greenfield environments, or those without existing tracing, is highly recommended. If a proprietary context propagation format is used, it is recommended to consider switching to an open standard, such as W_3C Trace Context, to future-proof the architecture and ensure data portability.

Trace data can be more easily consumed visually. Each trace can be represented as a waterfall, as shown in Figure 1.4.

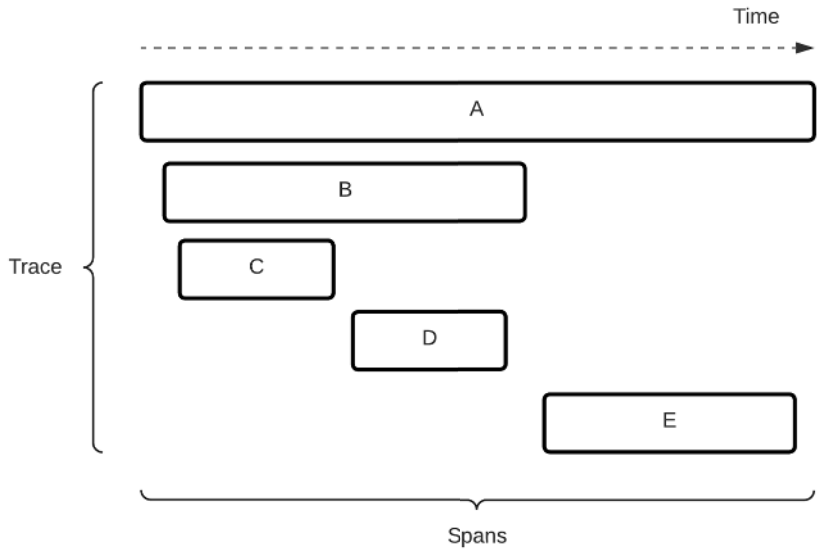


Figure 1.4
An example of a generic trace represented as a waterfall. This waterfall example could be further enhanced by denoting if an error occurred at any step in the transaction.

While example traces, often referred to as *exemplars*, can be useful, especially after problem isolation has occurred, aggregate trace information is helpful in determining system health and performance. In addition, aggregate data can be used to construct service graphs of an environment, as shown in Figure 1.5. Traces also contain other signals. At the very least, traces contain RED metrics. Metadata within a trace may also contain metric information, and spans can contain events. Events are essentially a specific type of log, as described in the "Logs" section earlier in this chapter. In addition, spans with errors can also contain logs.

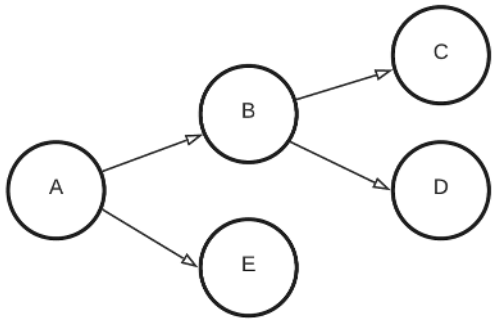


FIGURE 1.5
An example of a generic trace represented as a service map. As you may have noticed, this service map was constructed from the waterfall example shown previously. This service map example could be further enhanced to include RED metric data.

The concept of tracing has been around for a long time[26] and has been implemented in various ways. For example, logs customized to include and pass an ID have been used to provide tracing capabilities. Before the cloud native era, most tracing was proprietary, either developed in-house or provided by commercial vendors. Now, open source and open standards have changed this—a topic that will be explored in Chapter 2. In summary, OpenTelemetry has emerged as the tracing (metrics and logs too) standard, with W_3C Trace Context as the context format (made available to metrics and logs too).

Like logs, traces have the propensity to generate a lot of data. The number of traces generated depends on the number of requests through the instrumented paths of the architecture. For high request environments, a lot of traces are generated. In addition, depending on the amount of metadata, individual traces may be large in size. Different sampling techniques have been introduced since there is a cost associated with generating, processing, and storing trace data, and not all traces provide the same value. The two primary techniques are:

- Head-based—When a sampling decision is made with the initial request in a transaction (think the first span of the trace), as shown in Figure 1.6. Head-based sampling is easy to implement as the decision is made before the trace is created. However, not all information about the transaction is known in the initial request, so sampling here can result in large gaps in observability. For example, knowing whether an error will happen later in the transaction cannot be determined ahead of time.

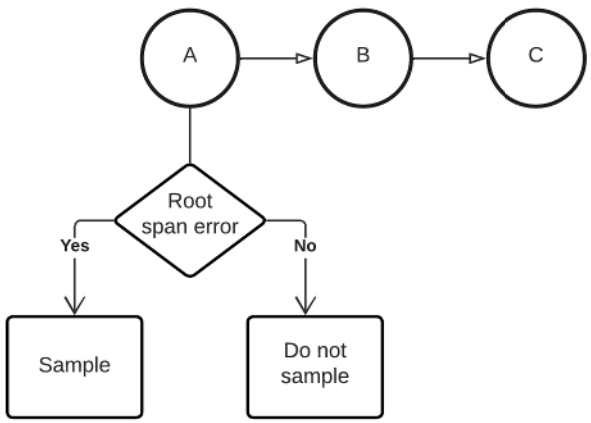


FIGURE 1.6
Example of head-based sampling. Each circle represents a microservice. The diamond represents a decision made within the service and the rectangles represent the action taken.

- Tail-based—When a sampling decision is made at the end of the transaction (think after the last span of a trace), as shown in Figure 1.7. Tail-based sampling requires generating and storing (typically in-memory) all information about the trace until the transaction is complete (typically done by waiting some period of time) and then making a sample decision. As such, tail-based sampling is significantly more complicated to implement and can consume substantially more resources, especially at a large scale. With that said, it has all the data, so it can make better sampling decisions than head-based sampling.

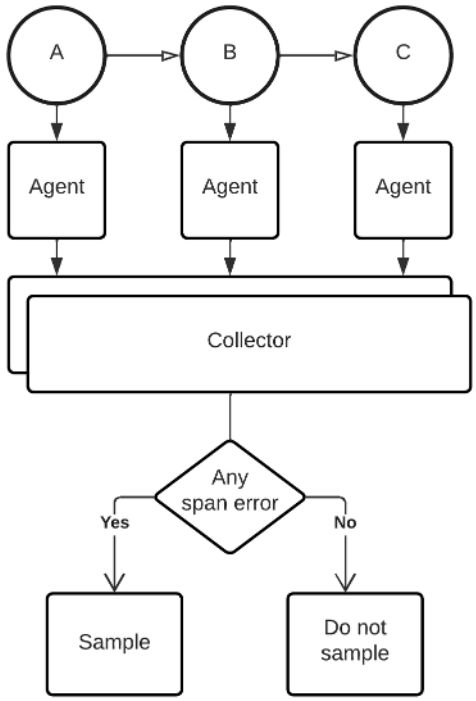


Figure 1.7
Example of tail-based sampling. Each circle represents a microservice. All microservices send data to a local agent, which sends data to the same node in a gateway cluster. The diamond represents a decision made by the gateway nodes, and the rectangles represent the action taken.

It is worth noting that sampling and filtering are similar concepts but not the same. For example, configuring tail-based sampling to collect all traces for which at least one span has an error is an example of filtering. Alternatively, collecting 10 percent of traces for which at least one span has an error is an example of sampling. While sampling can be used to reduce the amount of data stored, analyzing all trace data is critical to achieving observability. For instance, regardless of whether a trace is sampled, the aggregate RED metrics provided by the spans in the trace are necessary to achieve observability. To collect these metrics, processing of all spans is required. In addition, sampling can impact your ability to achieve observability if not properly configured. In short, trace sampling requires careful consideration and is not a magic solution.

Other Signals

While understanding and collecting the traditional three pillars is critical to achieving observability, other data sources are also important and should be considered. Some to be aware of include:

- Baggage, which is metadata passed between spans that must be explicitly added to signals. An example use case for this signal is adding metadata from further up the stack to telemetry that happens much later. While this concept is common in tracing, it may not be familiar to all readers. It is also a different type of signal in that it must exist in the presence of another signal, such as metadata on a different signal. Baggage will be covered in Chapter 4, "Understanding the OpenTelemetry Specification," and Chapter 8, "The Power of Context and Correlation."
- Sessions, which are used in Real User Monitoring (RUM) to analyze user experience (UX). The traditional three pillars provide a large amount of the data required to get visibility into applications that are being operated or depended on. Still, they do not provide complete, end-to-end visibility. What is missing is the user side. The end user connects to applications through either a browser or a mobile device. It is essential to understand what the end user is doing and experiencing, and that is where RUM helps. Sessions are signals that are collected from end-user devices, and they are similar to traces. They contain a unique identifier, are time-based, and optionally have additional metadata. Another concept in RUM is session replay,[27] which captures a snapshot of what the user did and experienced from a UI perspective. Session replay captures different data but attaches to and complements sessions. Finally, crash analytics is important and something that can be collected and attached to session data.
- Profiles, which are used to describe an application's execution and to understand code behavior from a CPU and memory perspective. Even if you can get to problem isolation with cloud native workloads, sometimes it can be hard to know why an application is behaving the way it is. Profiling[28] is like tracing within an application and provides deeper insights into how code is behaving in an application. The collection of profiling data is done via call stacks. A call stack is like a trace in that it contains a unique identifier, one or more calls (think spans), and optionally additional metadata. The difference is that the calls are explicitly only function calls in the application code.

Collecting Signals

As you can see, the three pillar signals are similar in terms of the general structure containing time-based information that can be analyzed. Traces provide a documented path of a transaction, similar to contact tracing in the medical field. They help identify where an issue occurred (problem isolation) as well as its impact. Traces are useful in answering at least the *where* question. Metrics typically collect symptoms, like a doctor checking vitals, and help identify what happened and/or changed. Metrics are useful in answering at least the *what* question. Logs can provide security information as well as the specific reasons behind a system's behavior, and they are like a medical record containing an audit of what has been done and who did it. Logs are useful in answering at least the *why* and *who* questions. While each of these signals is powerful, they are all necessary to have full observability.

Given that all three signals are similar, why were three signals created instead of one? To answer this question, consider some of the differences:

- Metrics emit small payloads at very frequent intervals. As a result, performance is critical. Metrics are usually not collected nor sent anywhere by default. While it is easy to add metrics and many frameworks exist, most metrics do not contain context or correlation information.
- Logs can contain richer information than metrics but, as a result, have larger payloads, and parsing requires proper formatting. Logs are usually written to a destination like a disk or a remote solution. While frameworks exist to add logs to applications, developers add most logging manually. Like metrics, most logs do not contain context or correlation.
- Traces are similar to logs. While they are easier to parse, it requires assembling an entire trace to realize the full potential. Traces require passing a context (typically a header) between requests. In addition, adding trace instrumentation is often significantly more challenging than metrics or logs. Trace payloads are as big, if not bigger, than logs and are frequently sampled.

Instrumentation

Adding something to an application to generate, process, and emit signals is known as *instrumentation*. Instrumentation, which will be covered extensively in Chapter 6, "Leveraging OpenTelemetry Instrumentation," may be signal specific or support more than one signal type. Applications can be instrumented in the following ways:

- Manual, sometimes called *code-based*, meaning a developer needs to add it directly into the code. Manual instrumentation offers a lot of flexibility but comes at the cost of developer cycles to add and maintain.
- Automatic, sometimes called *zero-code*, meaning it can be injected into the application, most commonly at runtime. Automatic instrumentation is convenient and is often easier to use when getting started. Still, it can only truly instrument known frameworks, may not provide as much detail as desired, and is not as customizable as manual instrumentation.
- Via an instrumentation library, sometimes called *programmatic*, meaning a library or framework comes with built-in instrumentation. While still a form of manual instrumentation, this approach requires less instrumentation knowledge and code changes. In addition, it handles aspects such as context propagation by default. With that said, an instrumentation library only instruments a known library.
- A combination of approaches, meaning more than one way. For example, manual and automatic or manual and programmatic can be used together.

Hopefully, it does not come as a surprise that adding instrumentation to applications in order to generate, process, and emit signals is not "free." While instrumentation is optimized to minimize the resources required, it will still consume some amount of CPU time and memory. How much overhead do they introduce? Like many things in life, it really depends. It depends on factors including:

- Where instrumentation is added—For example, in a frequently called or latency-sensitive path, sometimes called a *hot path*

- What is required to generate it—For example, blocking or asynchronous calls
- What processing is done—For example, regular expression matching
- If the data requires transformation—For example, aggregations or renaming
- How and where the data is exported—For example, buffer and retry logic

While tests can be performed and results published, they depend on various factors that may or may not be applicable to your environment. In general, automatic instrumentation may have more overhead than properly implemented manual instrumentation. In addition, automatic instrumentation may impact the startup time of applications, including Java-based applications. It is strongly recommended that you test the overhead of any instrumentation you add to your applications before deploying it to production. If you experience performance problems, check the documentation and configuration for potential issues, upgrade to the latest version to see if the issue has been resolved, and review release notes or known issues to see if others have reported it. If you are unable to find anything, then be sure to clearly document everything necessary to reproduce the issue and file an issue.

Real World Scenario

Watchwhale Pushes APM

One of Riley's first tasks was to meet with the Watchwhale vendor and determine the adoption plan. The vendor account team was encouraging Riley and Jupiterian to manually instrument their microservices with Watchwhale instrumentation so that engineers could start taking advantage of application performance monitoring (APM) capabilities. They believed that introducing APM, a technology Jupiterian was not currently leveraging, would provide better visibility into the environment and allow the on-call team to be more proactive. The instrumentation was open source but developed and managed by Watchwhale.

Riley understood the importance of instrumenting applications to generate, process, and export telemetry data, but she was concerned about using proprietary instrumentation. For example, if the leadership team at Jupiterian changed and decided to bring in a different observability vendor in the future, then this instrumentation would need to be ripped out and replaced. While Watchwhale did offer some automatic instrumentation, Riley knew that she would eventually need to work with developers to add manual instrumentation for custom code and enrich the telemetry with metadata. Instead, she proposed ingesting the current observability data, namely Prometheus metrics, into Watchwhale and later dealing with instrumentation and APM.

Push Versus Pull Collection

In general, signals can be collected via push or pull mechanisms. Push means that the application sends the signal to a configured destination. The push mechanism is ubiquitous for logs and traces. Examples include a log being written to a log file or spans being sent to an agent or directly to an observability platform. The alternative is pull, where applications make their signal data available, and some other system collects this data. For cloud native workloads, the pull

mechanism is most common for metrics. Examples include a collector configured to gather data from an endpoint such as the Kubernetes (K8s) API or scraping a Prometheus endpoint for metrics. Of course, metrics can also be pushed and at least logs could be pulled.

While either mechanism is acceptable to use, it is important to note the differences. One of the biggest issues is that pull-based mechanisms require the data to be exposed to a different system. As a result, proper security measures must be in place. Beyond this, whichever mechanism is used needs to be able to account for issues in collecting/sending data, including buffering and retry logic, and introduces the potential for overhead that must be accounted for.

Data Collection

You can collect signal data in a variety of ways. In addition, several data collection architectural topologies can be adopted. Each architecture depends on a multitude of factors, including security, cost, and configuration. Perhaps the most straightforward architecture is to have applications emit their signal data directly to an observability platform for processing and storing, as shown in Figure 1.8. There are a variety of reasons why you might consider this model, including:

- Simplicity—This architecture may be chosen when getting started or for small or noncomplex environments. It reduces the number of hops between telemetry generation and persistence.
- Efficiency—For some environments, an agent may consume too many resources or provide capabilities that are not required for the use case. Examples of such environments may include serverless functions or Internet of Things (IoT) devices.

Some of the drawbacks include the potential overhead to applications to handle buffer and retry logic as well as concerns around token management, processing configuration, and data loss risk (due to small retry buffers).

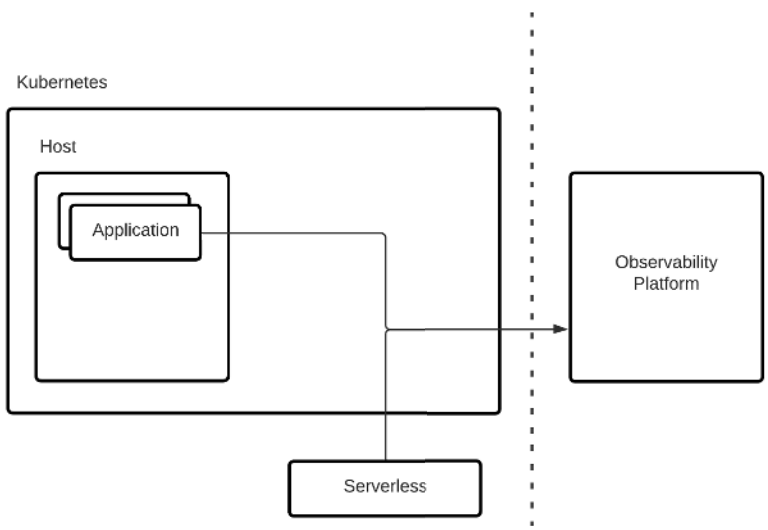


Figure 1.8
An application emitting telemetry data directly to a vendor-based observability platform. Everything to the left of the dashed line is within the customer's environment, while the vendor provides everything to the right.

Another common model is to deploy an agent close to the application, as shown in Figure 1.9. "Close" may mean directly in the application as a separate binary, as a sidecar to the application, or on the host the application is running on. There are a variety of reasons why you might consider this model, including:

- Middle ground—Host-level agents are common as they offer a reasonable trade-off between the deciding factors. They also separate the responsibility of telemetry generation from processing and transmission.
- Consistency—Agents make it easier to handle security concerns and centralize configuration. This is especially important in polyglot environments.

On the flip side, agents need to be appropriately sized to support the amount of data received, processed, and exported, which can be challenging. In addition, agents introduce another hop and configuration point between the telemetry data and the observability platform.

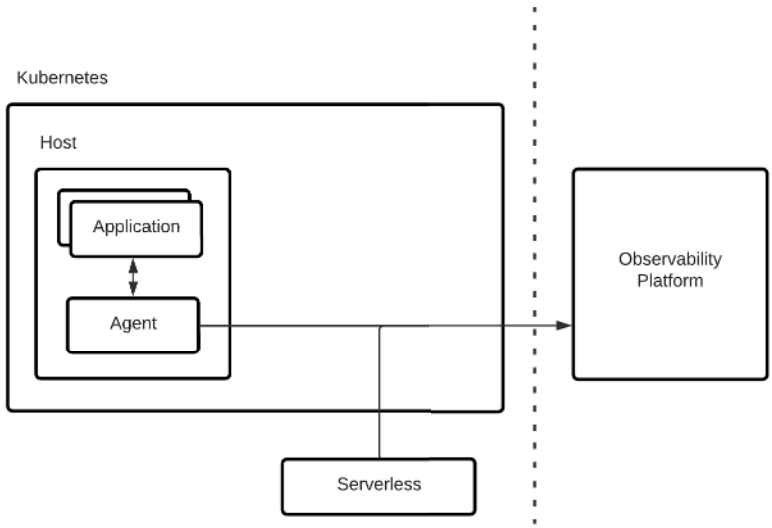


FIGURE 1.9
An application that emits its telemetry data to an agent running nearby, and the agent sends the data to a vendor-based observability platform. Everything to the left of the dashed line is within the customer's environment, while the vendor provides everything to the right.

A third model would be to introduce a collection tier between the agent and the observability platform, as shown in Figure 1.10. This tier goes by various terms, including *aggregation*, *collection*, *edge*, or *gateway*. You can think of this as an agent sending to another agent, though that may not always be the case, such as serverless functions sending directly to a collection tier. There are a variety of reasons why you might consider this model, including:

- Security—For example, network access
- Business requirements—For example, data lake
- Specific functionality—For example, tail-based sampling

While this architecture is more complex, it offers the greatest flexibility and can handle a variety of business requirements.

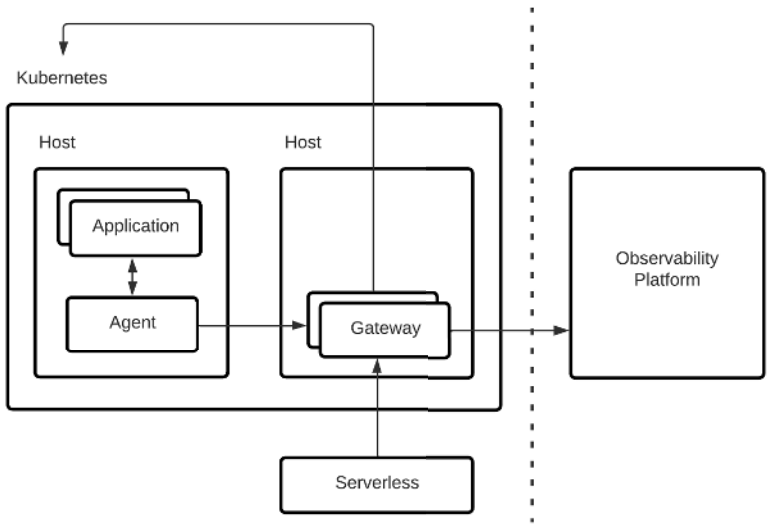


Figure 1.10
An application that emits its telemetry data to an agent running nearby, the agent sends the data to an edge cluster (gateway), and the edge cluster sends the data to a vendor-based observability platform. Everything to the left of the dashed line is within the customer's environment, while the vendor provides everything to the right.

As discussed previously in the "Signals" section, one consideration is the amount of signal data generated. One technique to reduce the amount of signal data sent to an observability platform is to filter it at the collection level. Of course, filtering is also useful for other reasons, including security, such as filtering out PII data. Filtering at the collection level reduces additional overhead on the application and makes centralized configuration easier. Data collection and processing will be covered extensively in Chapter 5.

Real World Scenario

Riley Avoids Vendor Lock-in Again

With the focus on consuming Prometheus metrics, the vendor suggested deploying the Watchwhale agent between the applications and the Prometheus platform. They stated their agent supported sending data to their platform and Prometheus concurrently. They also noted the Watchwhale agent collected critical information about the infrastructure and used it to add metadata to the telemetry data being collected beyond what Prometheus provided. They claimed this metadata was essential to achieving observability and being able to ask any question of the data.

Riley asked if the Watchwhale agent supported federated Prometheus server scraping, as this would allow her to easily tap into the existing architecture and try out the Watchwhale solution. Alternatively, she would need to deploy another agent, which would require a design document, architecture review, security review, and a migration plan, not to mention ongoing maintenance. The vendor stated that the Watchwhale agent was not designed to scrape a Prometheus cluster as large as the one run at Jupiterian. Riley told the vendor she would run some tests and get back to them regarding the data collection architecture.

Sampling Signals

While sampling was discussed in the earlier "Traces" section, its use extends beyond tracing. As may have been evident upon reading the earlier "Signals" section, the amount of data generated could be significant. For example, consider a system whose instrumentation generates a trace for every request. If the system has a million requests per second (RPS), then that means a million traces per second are generated. If all one million of these requests are successful and completed quickly, then retaining all the trace information may not be necessary. However, at least the RED metrics from the traces may be quite useful. In addition, with traces, you will not know if the entire transaction was successful or performed well until all calls have completed. This means the trace data needs to be generated before checking the status. Alternatively, if all one million traces returned the exact same error, collecting all of them again may not make sense. Ultimately, decisions need to be made on how much overhead is acceptable, how much data to generate, what to do with generated data and budgetary constraints. Chapters 5 and 6 will cover data collection and instrumentation tasks respectively, while Chapters 10, "Observability Antipatterns and Pitfalls," and 11, "Observability at Scale," will provide considerations guidance to ensure successful implementation.

Filtering and Sampling

Two terms you will hear regarding processing signal data are *filtering* and *sampling*. Filtering allows you to remove a subset of data from a dataset. For example, maybe you only want to collect data that has or does not have a particular piece of metadata. You can think of filters as a gate allowing only specific data to pass through based on predefined rules. Sampling can offer the same capability. For example, you can sample traces where at least one span has an error denoted through metadata semantic conventions. Sampling can also be configured to perform additional operations. For example, you could configure sampling only to collect 5 percent of traces for a given transaction, known as *probabilistic sampling*. In short, sampling can be thought of as a superset of filtering.

The concept of sampling exists to provide data flexibility. Sampling allows a subset of data to be generated or collected based on defined rules. For example, with tracing, you could choose to sample only initial requests for a specific operation or only if any span in a trace contains an error. Sampling decisions made at the initial request can be made via head-based sampling, whereas decisions after the initial request can be made via tail-based sampling. Similar scenarios exist for other signals as well. For example, collection interval, chart resolution, and exemplars, described earlier in the "Metrics" section, are all forms of sampling. Another example is profiling, which has the notion of temporal collection sampling or collection for a fixed period of time. Sampling rules are flexible and often involve metadata, further highlighting the need for proper metadata enrichment of signal data. What is important to remember is that sampling is a trade-off decision between observability, price, and complexity.

Real World Scenario

Riley Stays Focused on Metrics

The vendor also mentioned that Jupiterian would want to deploy the Watchwhale reducer when APM instrumentation is configured. This component offered tail-based sampling, which they stated helped reduce the amount of non-important telemetry data collected to minimize noise and cost. They noted that a single instance could easily be scaled up and was enough for most environments.

They also mentioned that multiple instances could be clustered and used with Redis. Riley understood the value proposition of tail-sampling but was worried about the availability issues of a single instance and the complexity of managing a distributed system of reducers. Luckily, she did not need to worry about this piece, as the focus was on consuming Prometheus metrics.

Observability

Once you generate, collect, and process telemetry data, you need to decide where to store the data and how to analyze it in a way that provides observability. There are many solutions in the market today that offer various capabilities. You need to determine what your requirements are. Considerations may include:

- Scalability—Is the solution able to scale to your needs? Ingesting the data is only one part of the story, the time-to-alert, ability to index, and query response time all also matter.
- Reliability—Is the solution able to meet your availability requirements? What happens if it goes down?
- User experience—Can you easily navigate across signals and how easily can you programmatically configure or access your data?
- Ease of use—How difficult is it to ask questions about your observability data?
- Performance—How quickly are results returned for queries?
- Security—What about compliance requirements such as SOC 2 Type II, PCI, and HIPAA?
- Cost—How much does the product cost and what options does the solution provide to manage expenses?
- Lock-in—Are you able to leverage open standards or proprietary solutions?

Platforms

While signals are critical to achieving observability, so is the platform to store, analyze, query, chart, and alert on the signal data. While observability platforms will be covered in Chapter 9, "Choosing an Observability Platform," it is important to know that the market has a wide range of open source and commercial solutions available. Here are some of the most popular open source observability platforms:

- Prometheus (`https://prometheus.io`) is a CNCF project that provides a platform to store, alert, and query metric data. Grafana (`https://grafana.com/oss/grafana`) can be configured to display Prometheus data in the form of charts and dashboards. Note that Grafana is open source but not part of the CNCF. It is primarily maintained by Grafana Labs, a for-profit company. In addition, there is Thanos (`https://thanos.io`), another CNCF project that can be used to make Prometheus highly available with long-term storage capabilities.
- Jaeger (`https://jaegertracing.io`) is a CNCF project that provides a platform to store, query, and visualize trace data.

- OpenSearch (`https://opensearch.org`) provides a platform to store, alert, and query log data, as well as a frontend called OpenSearch Dashboards. Fluentd (`https://www.fluentd.org`) and Fluent Bit (`https://fluentbit.io`) are CNCF projects that provide a log agent that can send data to platforms, such as OpenSearch. Note that Elasticsearch (`https://github.com/elastic/elasticsearch`) and Kibana (`https://github.com/elastic/kibana`) are also popular, though were not considered open source for a period of time after Elastic, a for-profit company, moved the projects to the Server Side Public License (SSPL).[29] Elastic has since changed its license again adding AGPL and allowing it to be called open source again.[30]

Application Performance Monitoring

In the cloud native world, application performance monitoring, or APM,[31] has become more mainstream. It existed in the previous generation, but given that problem isolation was usually easier, profiling was more commonly used when needed. APM relies on the traces signal, and one of its most significant benefits is that it provides native context and correlation—something not typically found in the other major signals. Observability is sometimes conflated to be the same as APM. Given that one of the major pain points around cloud native monitoring is problem isolation, this may not be surprising. With that said, APM adoption is not nearly as prolific as metrics and logs in part because instrumentation is a challenging problem, and most people have become comfortable troubleshooting applications using primarily metrics and logs. In addition, just adding APM does not inherently give you observability. Beyond the signals, it is the correlation of the data and the ability to ask and answer unknown questions about your environment with that data that gives you true observability. In 2022, Gartner renamed the APM Magic Quadrant to the APM and Observability Magic Quadrant to help eliminate this confusion.[32] In 2024, Gartner renamed it again to the Magic Quadrant for Observability Platforms.[33] Speaking of the Magic Quadrant, feel free to check it out for commercially available observability platforms. The CNCF Landscape[34] is another good resource for commercial and open source observability solutions. This book will focus primarily on open source solutions.

The Bottom Line

Differentiate between monitoring and observability. Monitoring and observability, while often used interchangeably, serve distinct purposes in the realm of system management. Monitoring involves the regular collection and analysis of predefined metrics and logs to ensure that systems are operating within expected parameters. It is a reactive approach, focusing on known issues and performance thresholds. Observability, on the other hand, is a proactive and comprehensive practice that goes beyond monitoring. It involves instrumenting systems to provide deep insights into their internal states, enabling teams to uncover and diagnose previously unknown issues and anomalies. While monitoring tells you when something is wrong, observability helps you understand why.

> **Master It** What is the difference between a "known known" and an "unknown unknown"?

Explain the importance of metadata. Raw telemetry data is not sufficient to achieve observability; it must be enriched. Metadata is a crucial piece of enrichment because it

provides context to the raw telemetry data collected from various systems. Metadata can include information about the environment, version, location, and other attributes that help in identifying and contextualizing the data points. This enriched context allows for more accurate analysis, correlation, and troubleshooting, making it easier to pinpoint the root causes of issues and understand the behavior of the system under different conditions.

> **Master It** What are the differences between dimensionality, cardinality, and semantic conventions?

Identify the differences between telemetry signals. Telemetry signals, the data components that power observability, typically include at least metrics, logs, and traces. Metrics are measured data, most commonly numerical representations, collected over intervals, such as CPU usage or request latency, providing a quantifiable view of system performance. Logs are timestamped records of discrete events that have occurred within the system, offering a detailed and chronological narrative of actions and states. Traces represent the flow of requests through various services in a distributed system, helping to visualize and understand the interactions and dependencies between components. Each signal type provides a unique perspective, and together, they offer a holistic view of the system's health and performance.

> **Master It** Why are there at least three separate ways to collect telemetry data from applications?

Distinguish between instrumentation and data collection. Instrumentation and data collection are foundational activities in achieving observability. Instrumentation involves integrating code within applications to generate telemetry data, including metrics, logs, and traces, that reflect the internal state and activities of the system. This can be done through manual coding or using libraries and frameworks like OpenTelemetry. On the other hand, data collection involves gathering this telemetry data from various sources, processing it, and transporting it to observability platforms for storage, analysis, and visualization. Effective instrumentation ensures that the correct data is generated, while robust data collection ensures that this data is reliably transmitted and available for analysis.

> **Master It** Given instrumentation, why is data collection necessary?

Analyze the requirements for choosing an observability platform. Selecting the right observability platform requires careful consideration of several factors. Scalability is crucial to handle the growing volume and velocity of telemetry data in modern systems. Integration capabilities are important to ensure the platform can seamlessly work with existing tools and technologies. The platform should support comprehensive data ingestion and processing capabilities for metrics, logs, and traces. It should offer advanced analytics and visualization tools to make sense of the data. Security and compliance features are also essential to protect sensitive data and meet regulatory requirements. Cost-effectiveness and ease of use are also critical factors, ensuring that the platform delivers value without excessive complexity or expense. By carefully evaluating these requirements, organizations can choose an observability platform that effectively supports their operational needs and strategic goals.

> **Master It** How are observability platforms different from APM?

Notes

1 https://en.wikipedia.org/wiki/There_are_unknown_unknowns
2 https://glossary.cncf.io/observability
3 https://landscape.cncf.io
4 https://opentelemetry.io/docs/concepts/observability-primer
5 https://www.sciencedirect.com/science/article/pii/S1474667017700948
6 https://en.wikipedia.org/wiki/Observability
7 https://www.cncf.io
8 https://www.merriam-webster.com/dictionary/monitor
9 https://www.merriam-webster.com/dictionary/observe
10 https://en.wikipedia.org/wiki/First_principle
11 https://en.wikipedia.org/wiki/White_box_(software_engineering)
12 https://en.wikipedia.org/wiki/Black_box
13 https://sre.google/sre-book/monitoring-distributed-systems
14 https://sre.google/workbook/how-sre-relates
15 https://twitter.com/honest_update/status/651897353889259520
16 https://grafana.com/files/grafanacon_eu_2018/Tom_Wilkie_GrafanaCon_EU_2018.pdf
17 https://www.brendangregg.com/usemethod.html
18 https://sre.google/sre-book/monitoring-distributed-systems
19 https://sre.google.com/sre-book/service-level-objectives
20 https://en.wikipedia.org/wiki/Request_for_Comments
21 https://opentelemetry.io/docs/specs/otel/logs/data-model
22 https://www.elastic.co/guide/en/ecs/current/ecs-reference.html
23 https://www.w3.org/TR/trace-context
24 https://github.com/openzipkin/b3-propagation/tree/master
25 https://docs.aws.amazon.com/xray/latest/devguide/xray-concepts.html
26 https://netman.aiops.org/~peidan/ANM2019/7.TraceAnomalyDetection/ReadingList/2010Google_Dapper.pdf
27 https://en.wikipedia.org/wiki/Session_replay
28 https://en.wikipedia.org/wiki/Profiling_(computer_programming)
29 https://www.elastic.co/blog/elasticsearch-is-open-source-again. Open source license changes are becoming more common, especially for popular projects (https://www.infoworld.com/article/3486307/the-open-source-community-strikes-back.html)
30 https://blog.opensource.org/the-sspl-is-not-an-open source-license
31 https://en.wikipedia.org/wiki/Application_performance_management
32 https://www.gartner.com/reviews/market/application-performance-monitoring-and-observability
33 https://www.gartner.com/reviews/market/observability-platforms
34 https://cncf.landscape2.io