O'REILLY®

# Cloud Native Monitoring

## Practical Challenges and Solutions for Modern Architecture

Kenichi Shibata, Rob Skillington & Martin Mao

REPORT

# Cloud Native Monitoring

Practical Challenges and Solutions for Modern Architecture

**Kenichi Shibata, Rob Skillington, and Martin Mao**


Beijing • Boston • Farnham • Sebastopol • Tokyo

**Cloud Native Monitoring**

by Kenichi Shibata, Rob Skillington, and Martin Mao



Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisition Editor: John Devins
- Development Editor: Sarah Grey
- Production Editor: Gregory Hyman
- Copyeditor: nSight, Inc.
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- April 2022: First Edition

**Revision History for the First Edition**

- 2022-04-08: First Release

978-1-098-12690-2

[LSI]

# Chapter 1. The Three Phases of Observability: An Outcomes-Focused Approach

The cloud native ecosystem has changed how people around the world work. It allows us to build scalable, resilient, and novel software architectures with idiomatic backend systems by using the power of the open source ecosystem and open governance.

How does it do that? Distributed architectures. The introduction of containers made the cloud flexible, and empowered distributed systems. However, the ever-changing nature of these systems can cause them to fail in a multitude of ways. Distributed systems are inherently complex, and, as systems theorist Richard Cook notes, "Complex systems are intrinsically hazardous systems."[1]

Think about how many different hazards a container faces: it can be terminated, it can run out of memory, it can fail the readiness probes, or its pods can be evicted from a restarting node, to name a few. These additional complexities are a trade-off for highly flexible, scalable, and resilient distributed architectures.

Distributed systems have many more moving parts. The constant struggle for high availability means that, more than ever, we need *observability*: the ability to understand changes within a system.

Thanks in large part to Cindy Sridharan's concept of "three pillars of observability," introduced in her groundbreaking work *Distributed Systems Observability*,[2] many people think that if you have logs, traces, and metrics (Figure 1-1), you have observability. Let's look quickly at each of these:

*Logs*

Logs describe discrete events and transactions within a system. They consist of messages generated by your application over a precise period of time that can tell you a story about what's happening.

*Metrics*

Metrics consist of time-series data that describes a measurement of resource utilization or behavior. They are useful because they provide insights into the behavior and health of a system, especially when aggregated.

*Traces*

Traces use unique IDs to track down individual requests as they hop from one service to another. They can show you how a request travels from one end to the other.

Indeed, as Sridharan makes clear, these are powerful tools that, if understood well, can unlock the ability to build better systems.

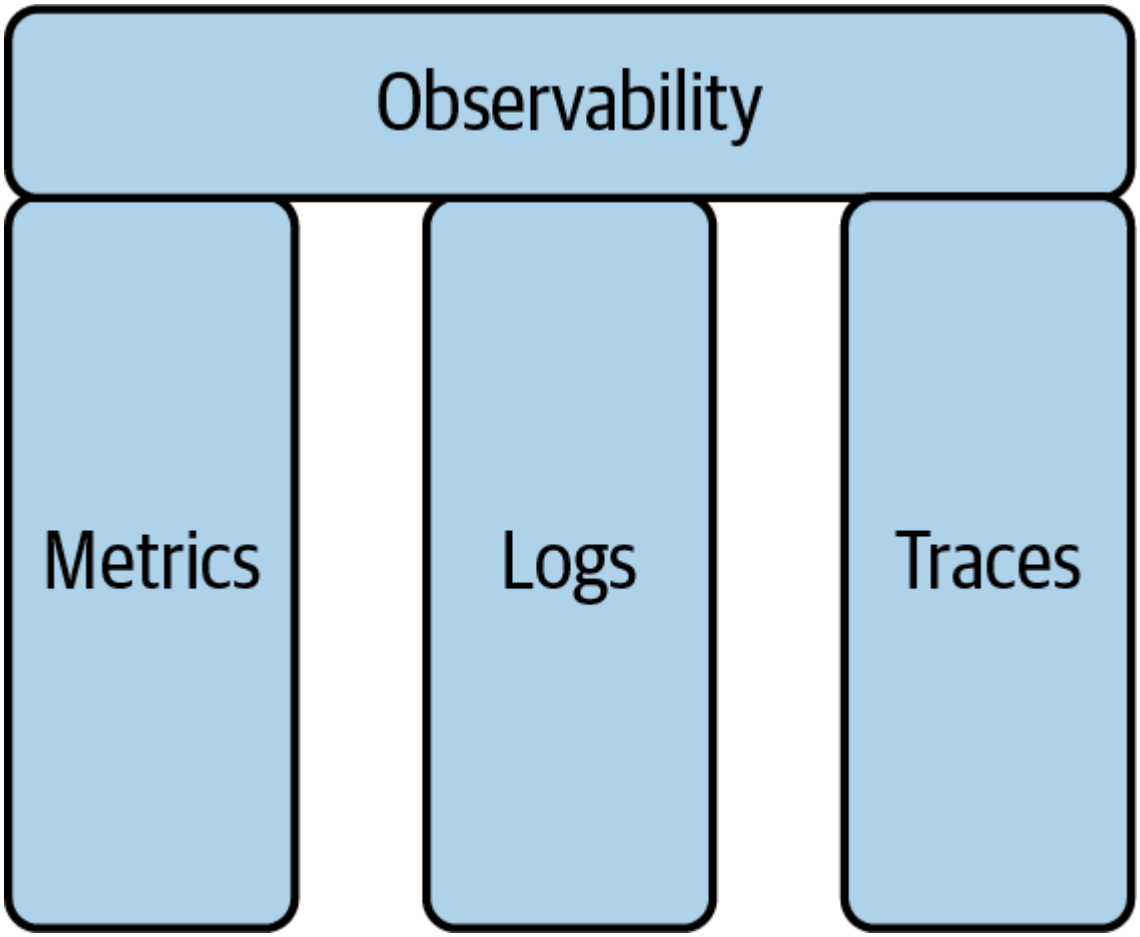


*Figure 1-1. The three pillars of observability*[3]

However, as Rob Skillington pointed out at 2021's SREcon,[4] simply adding more data (and more types of data) won't necessarily make observability more effective. After all, adding more data can easily create more noise and disorganization. Uber, he notes, initially used Graphite successfully with tens of microservices but found that it did not scale up to handle hundreds or thousands of microservices.

Martin Mao, along with Skillington, solved Uber's scaling problem by building M3, Uber's large-scale metrics platform. He points out that increasing your logs, metrics, and traces does not guarantee a better outcome either. Metrics, like logs and traces, are simply the inputs to observability, but having all three does not necessarily lead to better observability or even proper observability at all. Thus, in our opinion, metrics are the wrong thing to focus on.

If the three pillars of observability don't in themselves constitute observability, then how *do* we measure observability? In our view, one of the most impactful ways is to see how well your observability system helps you remediate an issue within the system efficiently. Our approach shifts the focus from what kind of *data* you have to what kind of *outcomes* you want to strive for. This is an outcomes-focused approach.

But let's take another step back and ask why we even want observability at all. What do we want to *do* with all this data we're producing? It's for a single, unchanging purpose: to remediate or prevent issues in the system.

As builders of that system, we want to measure what we know best. We tend to ask about what kinds of metrics we should produce in order to understand if something is wrong with the system and remediate it. Working backward from customer outcomes allows us to focus on where the heart of observability should be: *What is the best experience for the customer?*

In most cases, the customer (whether they are external or internal) wants to be able to do what they came to do: for example, buy the products they are looking for. They cannot do that if the payment processor isn't working. We can work backward from there: we don't want our customers to be unable to buy products, so if the payment processor goes down or becomes degraded, we want to know as soon as possible so we can remediate that issue. To do that, we need to ensure that we can detect payment processor downtime quickly, then triage to make sure we know the impact and the root cause, all while looking for opportunities to rapidly remediate, stopping the customer's pain.

Once you find the outcomes you are looking for, *then* the signals (metrics, logs, and traces) can play a role. If your customers need error-free payment processing, you can craft a way to measure and

troubleshoot that. When deciding on signals, then, we endorse starting from the outcomes you want.

In response to Sridharan, we call our approach the *three phases of observability* (Figure 1-2).

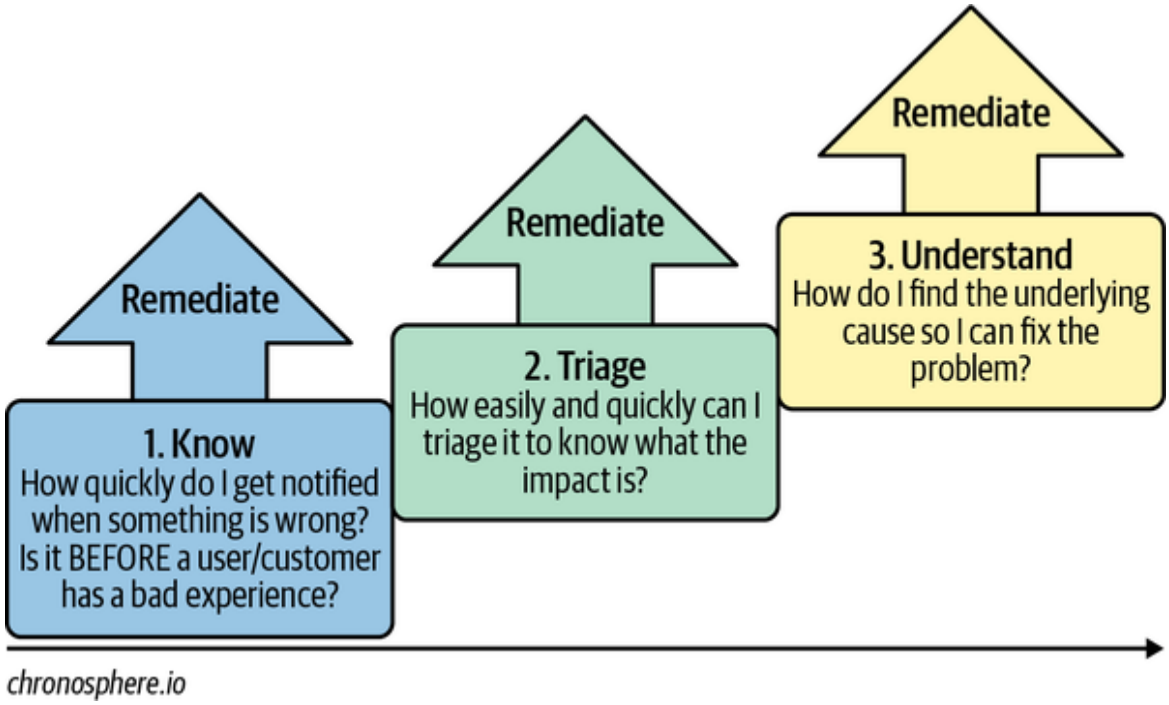


*Figure 1-2. The three phases of observability*[5]

As part of a remediation process, the three phases can be described in the following terms:

1. *Knowing* quickly within the team if something is wrong
2. *Triaging* the issue to understand the impact: identifying the urgency of the issues and deciding which ones to prioritize
3. *Understanding* and fixing the underlying problem after performing a root cause analysis

Some systems are easier to observe than others. The key is understanding the system in question.

Let's say you work for an ecommerce platform. It's the annual Black Friday sale, and millions of people are logged in simultaneously.

Here's how the three phases of observability might play out for you:

*Phase 1: Knowing*

> Suddenly, multiple alerts fire off to notify you of failures. You now know that requests are failing.

*Phase 2: Triaging*

> Next, you can triage the alerts to learn which failures are most urgent. You identify which teams you need to coordinate. Then learn if there is any customer impact. You scale up the infrastructure serving those requests and remediate the issue.

*Phase 3: Understanding*

> Later on, you and your team perform a postmortem investigation of the issue. You learn that one of the components in the payments processor system is scanning multiple users and causing CPU cycles to increase tenfold—far more than necessary. You determine that this increase was the root cause of the incident. You and the team proceed to fix the component permanently.

In this example, you resolved an issue using observability, even though you didn't use all three signals. Looking only at the metrics dashboard, you determined which systems were causing the issue and guided the infrastructure team in fixing it.

Just like in mathematics, there are multiple ways to arrive at the correct answer; the important thing is to do so quickly and efficiently. If you can remediate a problem by relying only on your previous knowledge of the system, without using metrics, logs, and traces, that is still a good outcome. You remediated the problem, and that's the real goal. And, of course, this is made easier with correct signals that are outcomes-based and can quickly validate any remediation assumptions!

## Remediating at Any Phase

Although we posit three phases, at any phase, your goal is always to remediate problems. If a single alert is firing off and you can remediate the issue by using only visibility (phase 1), you should do so. You don't have to triage or do a root cause analysis every time if these are unnecessary.

To illustrate this point, let's say a scheduled deployment breaks your production environment. There is no need to triage or do root cause analysis here, since you already know that the deployment caused the breakage. Simply rolling back the deployment when errors become visible remediates the issue.

## The Three Phases Illustrated

In real life, if your system is crashing, you don't focus on the data. You focus on fixing the problem immediately. No one does root cause analysis without fixing the current issue and mitigating customers' pain.

Take, for example, a burning house (Figure 1-3).

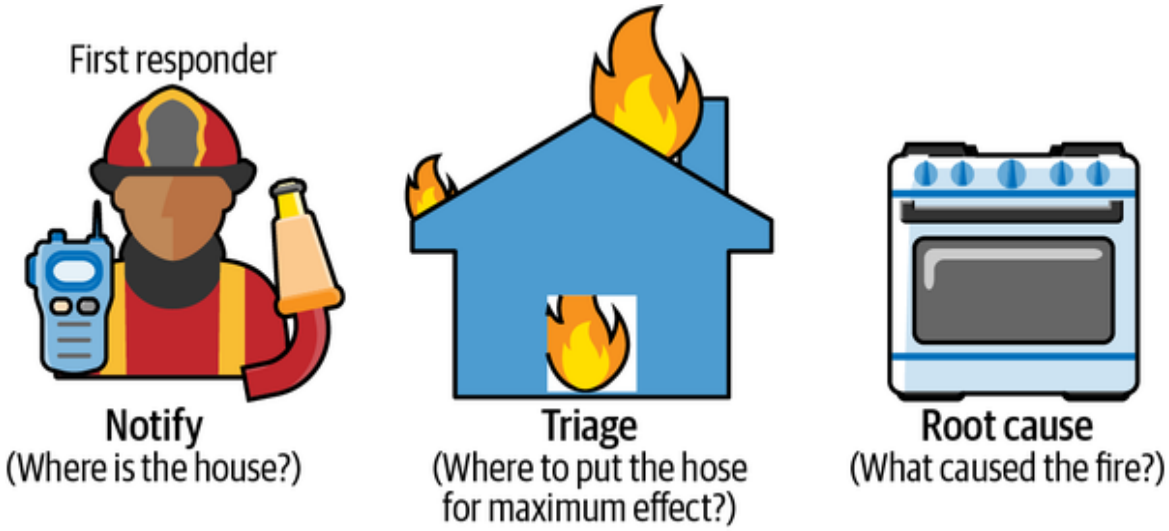


*Figure 1-3. Remediating a burning house: you should put out the fire before you start investigating the cause*

If your house is on fire, how do you know? Most likely, your smoke alarm goes off, emitting a loud, unmistakable noise that notifies you of the problem. That smoke alarm is the alert, triggered by sensors detecting smoke in the room. Metrics can tell you what the issue is and give you enough information to address it. This is surface-level detection but enough to continue investigating. That's phase 1. Metrics should give you a low *mean time to detect* (MTTD), so a sensitive fire alarm that goes off at the first sign of smoke or heat will be better than one that lets the fire spread for several minutes before notifying you—and better still than no alarm at all.

What now? You might jump out of bed and look around the house to see where the fire is, then get everyone out immediately and call emergency services. That's a temporary remediation (phase 2): you're all safe, but the house is still on fire. It's also triaging: you are choosing to prioritize safety over other things, like saving your favorite electronics.

The sooner you can call emergency services, the faster they will arrive to put the fire out. In observability, we call this interval *mean time to remediate* (MTTR). This, too, should be as low as possible: if the firefighters arrive quickly and start hosing down the house right away, part of the house could be saved. If anyone is injured, you'll want the paramedics to arrive quickly to help them: that is, you want a low MTTR.

The next morning, with everyone safe and the last embers extinguished, the fire marshal and insurance investigator examine the house to see what started the fire (a root cause analysis, phase 3). Perhaps they learn that a faulty cord on an appliance overheated. The appliance manufacturer might even recall the product to ensure that the faulty cords don't start any more fires, a still more permanent remediation that keeps even more people safe.

No one does an investigation during an active fire as there is still a threat of injury. Similarly, the worst time to do a deep dive on how exactly a system misbehaves is when there is an ongoing outage.

You do phases 1 and 2 immediately, before you try to figure out where the fire started or why. You focus on the *outcome* of keeping everyone safe. One way to fulfill that is to use metrics as your starting point. In this case, since smoke in the room is the metric, then once you smell smoke, you automatically evacuate the house.

---

1 Richard Cook, "How Complex Systems Fail," Cognitive Technologies Laboratory, 2000, *https://oreil.ly/zw73j*.

2 Cindy Sridharan, *Distributed Systems Observability* (O'Reilly Media, 2018), *https://oreil.ly/v8PUu*.

3 Sridharan, *Distributed Systems Observability*.

4 Rob Skillington, "SREcon21—Taking Control of Metrics Growth and Cardinality: Tips for Maximizing Your Observability," USENIX, October 14, 2021, YouTube video, 27:21, *https://oreil.ly/gvAq7*.

5 Adapted from an image in Rachel Dines, "Explain It Like I'm 5: The Three Phases of Observability," Chronosphere, August 10, 2021, *https://chronosphere.io/learn/explain-it-like-im-5-the-three-phases-of-observability*.

# Chapter 2. Why Do You Need Metrics?

You might have noticed that we're focusing specifically on metrics here rather than logs or traces. Why not logs first? Why metrics?

## Metrics as a Starting Point

Sridharan defines a metric as "a numeric representation of data measured over intervals of time," adding, "Metrics can harness the power of mathematical modeling and prediction to derive knowledge of the behavior of a system over intervals of time in the present and future."[1] Figure 2-1 shows an example of measuring HTTP requests as a metric.

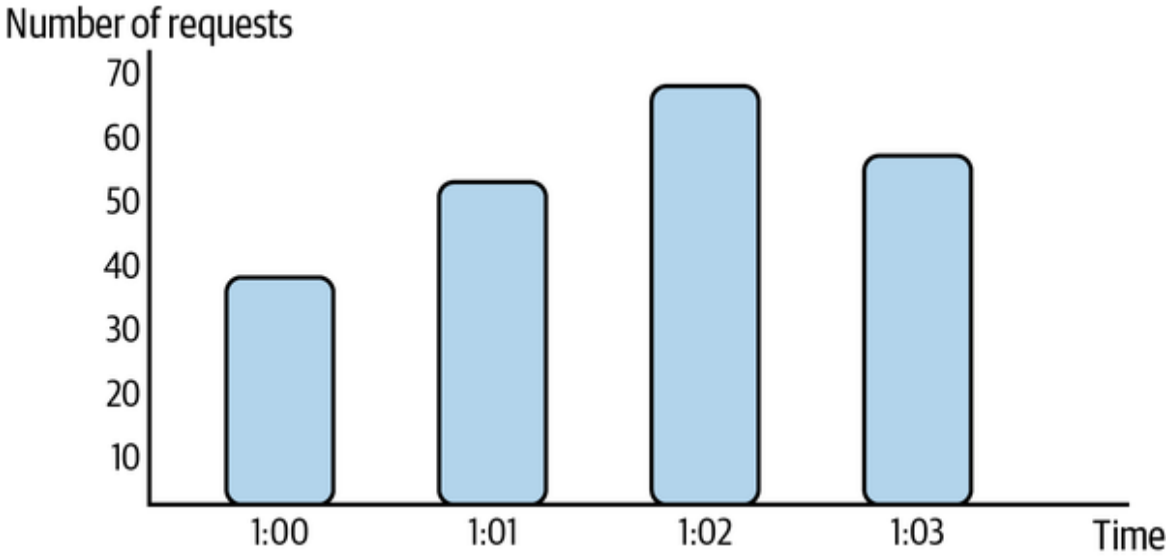


*Figure 2-1. An example metric*

## The Case for Metrics

If solving your problem requires a deep dive, you might need all three signals. Logs will tell you what happened in a specific period of time. Traces allow you to track a request from beginning to end.

However, when you are starting your investigation, you need a bird's-eye view. Starting with metrics is logical because it lets you move from the broadest view down to the narrowest. Metrics also can provide that perspective with what we call a *low-latency impact analysis*, which provides an efficient view of the system's current state. What's more, metrics are easy to implement and use, and they let you aggregate data quickly and compare it over time. Let's look at each of these factors in turn.

## Metrics Provide an Efficient Snapshot of the System

First, metrics allow you to understand the current state of the system efficiently. If you have some contextual knowledge, you can ask questions and prove or disprove assumptions so that you can start troubleshooting right away: *Is there an* `HTTP:503 Service Unavailable` *error? When did it happen?* Combined with context, metrics will get you those answers quickly—you don't need to do a JSON filter or a full-text search for all HTTP response codes.

One of the most popular metrics and monitoring platforms is Prometheus, which collects metrics by scraping metrics HTTP endpoints on monitored targets. (We'll look more closely at Prometheus in Chapter 3.)

Let's say you have an HTTP server that's producing an `HTTP:503 Service Unavailable` error. We'll show you how you can detect that quickly by graphing a metric in Prometheus.

In Figure 2-2, you can see that the `HTTP:503 Service Unavailable` errors started at 17:38 GMT, without even looking at the logs. However, it is not very simple to see if there is a new `HTTP:503 Service Unavailable`. If you want to alert on it, usually you will need to aggregate counters `rate()` or `irate()`.

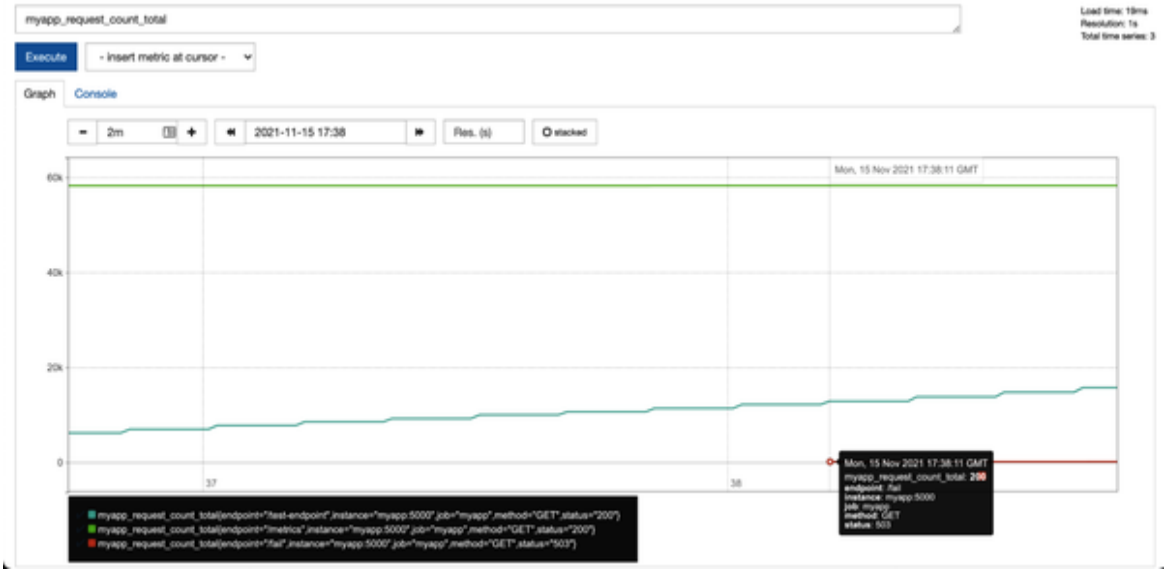


*Figure 2-2. Illustration of a Prometheus 503 error*

Here, the metric `myapp_request_count_total` is a Prometheus metrics type *counter*. There are four types of metrics in Prometheus:

*Counter*

> Counters are cumulative metrics that can only increase. In the preceding example, the `myapp_request_count_total` is a counter, and it either increases or does not move. As the name implies, counter metrics are best used for counting things like HTTP requests, RPC calls, or even business metrics like number of sales.

*Gauges*

> Gauges are singular metrics that can either increase or decrease. Examples of metrics measured in gauges include

temperature, speed, or memory usage. Gauges can also be used for metrics that decrease, such as concurrent HTTP requests.

*Histogram and summary*

Histograms and summaries are sample observation metrics. Examples of metrics expressed in histograms and summaries include request duration, request sizes, or ranking. Histograms and summaries are similar, but it is advisable to use a histogram for aggregation, since you can use quantiles. For more information about the difference between histograms and summaries, visit the Prometheus histogram documentation.

For more information about the types of Prometheus metrics, refer to the Prometheus documentation.

## Metrics Are Easy to Use

Metrics provide the most utility with the least amount of effort to set up. The open source tools you're already using likely support Prometheus exposition formats. In NGINX, PostgreSQL, and Kubernetes, for example, user communities have built Prometheus exporters that you can run using just a few lines of configuration.

## Metrics Let Us Aggregate Data Quickly

Metrics allow you to observe aggregated data over time, which is the fastest way to combine data to troubleshoot and fix your system. For example, you can find out how many bounces have happened in the last hour or the last day. You can see where the most bounces occur and even at which endpoints they happen.

Depending on the type of metrics, there are multiple ways to aggregate.

Here are a few examples of aggregated metric requests using *PromQL* (Prometheus Query Language) that can be used in *gauges*:

```
sum(metric_name)
max(metric_name)
count(metric_name)
```

Here are some examples of aggregation using *counters*:

```
sum(rate(myapp_request_count_total[1m]))
max(rate(myapp_request_count_total[1m]))
rate(myapp_request_count_total[1m])
increase(business_sales[1h])
rate(business_question_set_completed[1h])
```

Here are some examples of aggregation using *histograms*:

```
rate(http_request_duration_seconds_sum[5m])
rate(http_request_duration_seconds_count[5m])
```

This example from the Prometheus documentation calculates the average request duration during the last five minutes from a histogram `http_request_duration_seconds`.

In many cases, you still want to retain or remove the cardinality of these metrics while aggregating. To do so, you will need to use `by` to aggregate based on a dimension or `without` to aggregate all dimensions except for the one specified. For example:

```
sum by (status) (metric_name),
sum without (status) (metric_name),
```

`Sum` adds all the requests and aggregates the metric. `Max` takes the highest value of the metric and ignores the other value. `Count` gives the metric's cardinality (we'll discuss cardinality in the next section). These aggregations are useful when setting up advanced detection.

Metrics are a numerical summary or aggregation of data, and because of that, they are far more efficient to query than detailed logging or tracing data. For example, storing the number of requests as a metric is far more efficient than counting the individual log messages for every request. Even if there is no metrics instrumentation, it's often more efficient and recommended to create metrics from logs and traces for notification and triage purposes.

Want to see only the requests that produced errors? You can query your metrics to filter out the successful requests. Here, we count only the maximum number of requests that are successful:

```
max(rate(myapp_request_count_total{status="200"}[1m]))
```

## Metrics Allow Us to Create Alerts

In *Site Reliability Engineering*, Rob Ewaschuk defines *alerts* as "notifications intended to be read by a human and...pushed to a system such as a bug or a ticket queue, an email alias, or a pager."[2] As the name suggests, alerts let you know when there's a problem. They're meant to tell you when the system reaches a certain threshold that you've set, such as a number of HTTP requests, a certain error being generated, or even (in our fire example from the last chapter) a smoke alarm sensor picking up a certain level of smoke in the air. Querying metrics is a very efficient way to generate the alerts you need before you even go to individual logs.

Using the same Prometheus request you just saw, for example, allows you to alert on any occurrence of `HTTP:503 Service Unavailable` simply by using the same query and adding an evaluation greater than 0:

```
rate(myapp_request_count_total{status="503"}[1m]) > 0
```

This efficiency makes metrics the most axiomatic of the signals as a data source.

## Metric Data Is Growing in Scale

The massive shift of systems around the world from monoliths to the cloud has resulted in an ongoing explosion of metric data in terms of both volume and cardinality. This is especially prevalent in cloud native systems. To achieve good observability in a cloud native system, you will have to deal with large-scale data.

We describe this explosion of data as a metric *scale*, or the number of metrics produced during instrumentation. How many things are you measuring, and how much data does that measurement produce?

Say you're working with MyApp (a fictional app). You start by using a metric called `myapp_request_count_total`, which counts the number of HTTP requests MyApp receives:

```
myapp_request_count_total{endpoint="/test"} 226
```

Now you can add more metric data, like all created requests, all requests, and all bounced requests:

```
myapp_request_count_total{endpoint="/test"} 226
myapp_request_count_created{endpoint="/test"} 163
myapp_request_bounce_total{endpoint="/test"} 440
```

You can see how fast the volume of data can increase when you add more metrics. The volume of logs and traces doesn't change very much, as a rule, but metrics are a different story.

## Understanding Cardinality and Dimensionality

Each increase in cardinality multiplicatively increases the volume of metrics, which requires more system resources for Prometheus. High cardinality therefore degrades Prometheus's performance. As you gather more dimensions for each metric, you add observability context—at the cost of Prometheus performance.

Before we go any further, let's take a quick detour to discuss these two important concepts that you'll need to understand: cardinality and dimensionality. *Cardinality* is the number of possible groupings depending on the dimensions the metrics have. *Dimensions* are the different properties of your data, as Rob Skillington explains.[3] Think of the labels on a shirt on a store shelf. Each label (in this simplified example) contains three dimensions: color, size, and type.

Each dimension increases the amount of information we have about that shirt. You could slice that information into many shapes, based on how many dimensions you use to sort it. For example, you could look by just color and size, size and type, or all three. *Dimensionality* means being able to slice the metrics into multiple shapes.

Increased dimensionality can greatly increase cardinality. Cardinality, in this example, would be the total number of possible labels you could get by combining those dimensions *from the shirts in your inventory*. Figure 2-3 visualizes this example by looking at two dimensions: color and size.

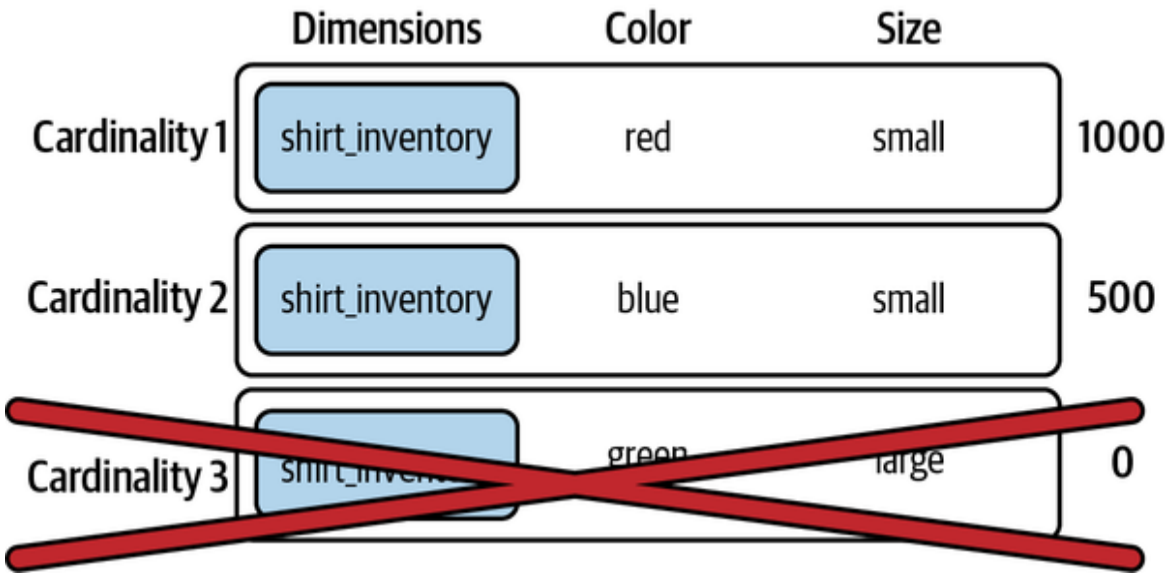


*Figure 2-3. Shirt inventory with two cardinalities*

There are only two cardinalities represented in Figure 2-3, even though there are three possibilities. That's because the last combination, while theoretically possible, is not in the inventory and therefore is not emitted to the metrics platform.

The term *metric cardinality* refers to how many unique combinations of metric data "are produced by a combination of metric names and their associated label [dimension] names/values,"[4] while the total number of combinations with data that exists are cardinalities.

Now let's look at what happens if we increase the number of dimensions, and therefore of possible cardinalities, by adding the type of shirt. Figure 2-4 shows how increasing the number of dimensions also increases the cardinality of the metrics.

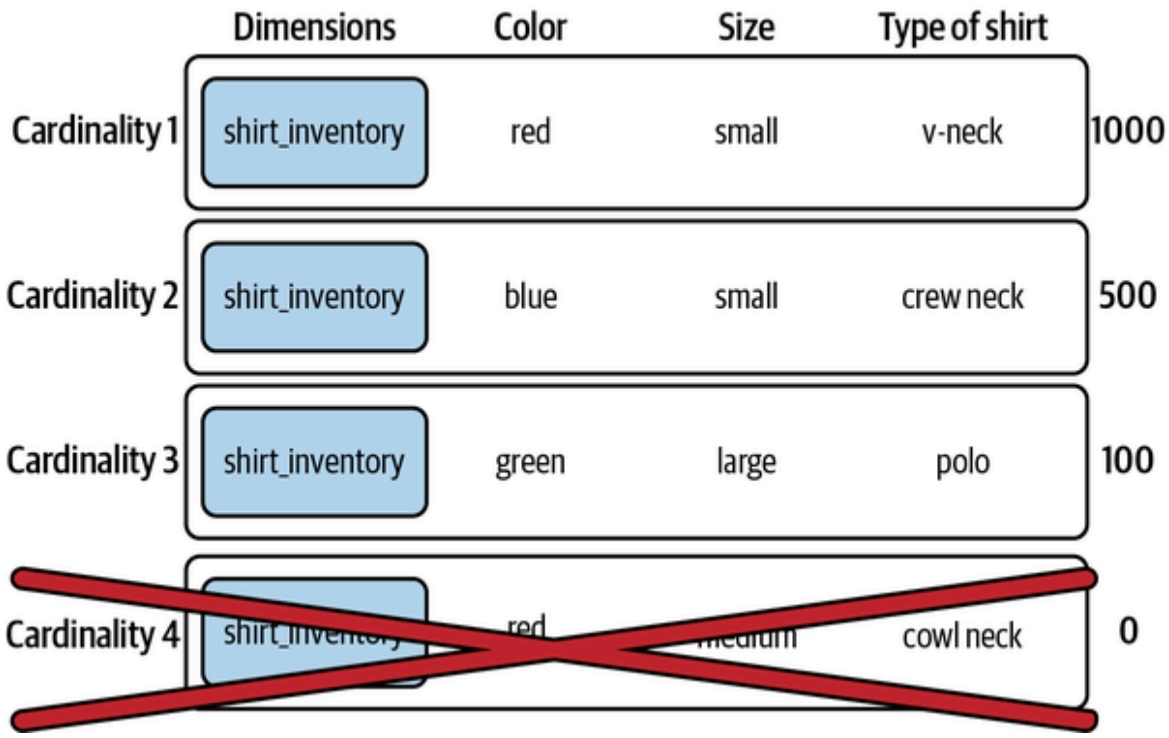


*Figure 2-4. Shirt inventory with three cardinalities*

Let's take another example, this time from the software world. Say we want to count the total number of HTTP requests in an API. We use the metric `api_http_requests_total` with two dimensions, `method` and `handler`. Working again in Prometheus, we could run:

```
api_http_requests_total{method="POST", handler="/messages"}
60
```

The total number of requests, in this case, is 60.

If we decide to track the HTTP status code as well, this would add a third dimension:

```
api_http_requests_total{method="POST", handler="/messages",
\
    status="200"} 30
api_http_requests_total{method="POST", handler="/messages",
\
    status="503"} 30
```

As Bastos and Araujo note, "Cardinality is multiplicative—each additional dimension will increase the number of produced time series [metric data] by repeating the existing dimensions for each value of the new one."[5]

## Cloud Native Systems Are Flexible and Ephemeral

"Containers are inherently ephemeral," Lydia Parziale and Zach Burns write in *Getting Started with z/OS Container Extensions and Docker*. "They are routinely destroyed and rebuilt from a previously pushed application image. Keep in mind that after a container is removed, all container data is gone. With containers, it is necessary that you take specific actions to deal with the ephemeral behavior."[6]

The other effect of using containers and cloud native architecture is that distributed systems are more flexible and more ephemeral than monolithic systems. This is because containers are faster to spin up and close down. Containers make observing an ephemeral system difficult, since they come and go quickly: a container that spun up just a second ago could be terminated before we get a chance to observe it.

According to a survey by Sysdig, 95% of containers live less than a week.[7] The largest cohort—27%—are containers that churn roughly every 5 to 10 minutes. Eleven percent of containers churn in less than ten seconds.

For example, let's consider a container that processes datafiles. This container runs Python code in a database called Bronze that deletes data that's more than one day old. It also creates a metric called `job_data_processed` with two dimensions, `pod` and `database`, to count the amount of data it deletes. The metric emitted by a single pod looks like this:

```
job_data_processed{database="bronze", pod="pod1"} 1000
```

This metric shows that this single pod processed 1,000 datafiles.

If we divide the thousand-file workload across 100 pods and use the same metrics, the amount of metric data produced will be multiplied by 100:

```
job_data_processed{database="bronze", pod="pod1"} 10
job_data_processed{database="bronze", pod="pod2"} 10
...
job_data_processed{database="bronze", pod="pod100"} 10
```

Now let's say we add more dimensions to the metric data. Some of the datafiles are in `json` format, others in `csv`. This adds 100 additional metric data points—and you'll get a further 100 more for each dimension you add:

```
job_data_processed{database="bronze", pod="pod1", \
  type="json"} 5
job_data_processed{database="bronze", pod="pod2", \
  type="json"} 5
...
job_data_processed{database="bronze", pod="pod100", \
  type="json"}
job_data_processed{database="bronze", pod="pod1", \
  type="csv"} 5
job_data_processed{database="bronze", pod="pod2", \
  type="csv"} 5
...
job_data_processed{database="bronze", pod="pod100", \
  type="csv"} 5
```

This is why data in cloud native architectures is constantly growing in scale and cardinality.

The flexibility of cloud native architectures allows for increased scalability and performance. You can easily take a pod away or increase it a hundredfold. You can even add labels to increase the context of the metrics. This has fundamentally increased the metric data produced.

## Cloud Native Services and Systems Have Greater Interdependencies

The DevOps culture in the cloud native world has allowed developers greater control over the entire lifecycle of their applications, including interdependencies with other applications.

Before cloud native, developers built entire suites of different business needs in the same application. This often made it nearly impossible to scale those applications independently.

Today, however, using microservices architecture, we can build a new microservice for each business need and orchestrate them all to work together. This allows for greater flexibility: we can scale some highly utilized parts of the architecture faster and in greater quantities than other parts. We can even reuse some of the more generic services in other architectures. This is a powerful feature of microservices that offers not only greater flexibility but greater scalability as well.

However, as we say in the architecture space, everything is a trade-off. In this case, we increase scalability, but the trade-off is that monitoring the individual services is more difficult. Even so, metrics let us track individual data points to be allocated to multiple services using dimensions.

Take the same container request you saw in the last section, but this time it's been smartly cut using dimensions.

Here is the original metric:

```
api_http_requests_total{method="POST", handler="/messages"
\
  pod="pod2"} 600
```

And here is the new metric:

```
api_http_requests_total{method="POST", handler="/messages"
\
  pod="pod2" from="frontendservice"} 300
api_http_requests_total{method="POST", handler="/messages"
\
  pod="pod2" from="backendservice"} 300
```

We can now determine how many requests came from the `backendservice` and how many from the `frontendservice`.

## The Risk of Losing Focus on Outcomes

As you've seen, metric data can grow very quickly. Even just instrumenting metrics creates a high cognitive load for any site reliability engineering (SRE) team. Because of this, paradoxically, it's not uncommon for SRE teams to lose sight of why they're instrumenting metrics in the first place. SRE teams' performance is often graded on how "well instrumented" systems are, which then leads to increasing the number of data points they instrument. Further, this constricts their focus on the data being collected rather than on the outcomes. They might instrument too much without knowing what to do with all that data.

This approach tends to involve asking questions like:

- Do you instrument your code?
- Which data do you collect?
- How many ways can you slice and dice your data?
- Are there dashboards to visualize this data?

Worse yet, they may even instrument the *wrong* data and present it to the dashboard and the alerting system.

You don't have to instrument literally everything, just the metrics that matter in your organization and that allow you to focus on

outcomes.

The loss of focus is even more apparent if we improve the questions:

- Do you get alerted appropriately when there is an issue?
- Does the alert give you a good place to start your investigation?
- Are the alerts too noisy?
- How do you visualize the data you collect?
- Do you even use it at all during incidents?
- Can you use the dimensions of the metrics to help triage and scope the impact of the issue?
- Are the alerts useful and helpful before and after incidents?

Again, we recommend always focusing on outcomes. To achieve this focus, follow the three phases of observability to refine your processes and tools iteratively, and be vigilant in measuring your MTTR and MTTD.

Metrics are an efficient and powerful tool for all three phases of observability. But how do we solve the problem of scale and data growth?

---

1 Sridharan, *Distributed Systems Observability*.

2 Rob Ewaschuk, "Monitoring Distributed Systems," chap. 6 in *Site Reliability Engineering*, ed. Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy (O'Reilly Media, 2016), *https://oreil.ly/3xCYp*.

3 Rob Skillington, "What Is High Cardinality," Chronosphere, February 24, 2022, *https://chronosphere.io/learn/what-is-high-cardinality*.

4 Joel Bastos and Pedro Araujo, "Cardinality," in *Hands-On Infrastructure Monitoring with Prometheus* (Packt, 2019), *https://oreil.ly/vmk4Z*.

5 Bastos and Araujo, "Cardinality."

6 Lydia Parziale, et al., Chapter 10, in *Getting Started with z/OS Container Extensions and Docker* (Redbooks, 2021), *https://oreil.ly/e21Wc*.

7 Eric Carter, *2018 Docker Usage Report* (Sysdig, May 29, 2018), *https://oreil.ly/ftZum*.

# Chapter 3. The Rise of Open Source Metrics

Before the industry standardized on Prometheus, many companies were forced to use proprietary metrics solutions, such as AppDynamics, Dynatrace, or New Relic. These vendors control the instrumentation and aggregation of metrics using *agents*, which are software processes that run alongside an application to collect data and then send it to an external server. If you are running an AppDynamics observability setup, you have no choice but to use the AppDynamics agent to send metrics to their system, as shown in Figure 3-1. This is called *agent-based* application instrumentation. Applications typically need to install a software library or software development kit to run these agents and send the data back to the aggregation server.

Agents are also largely noninteroperable. This means that if you rely on AppDynamics, the same agents cannot easily aggregate those same metrics into New Relic's system.

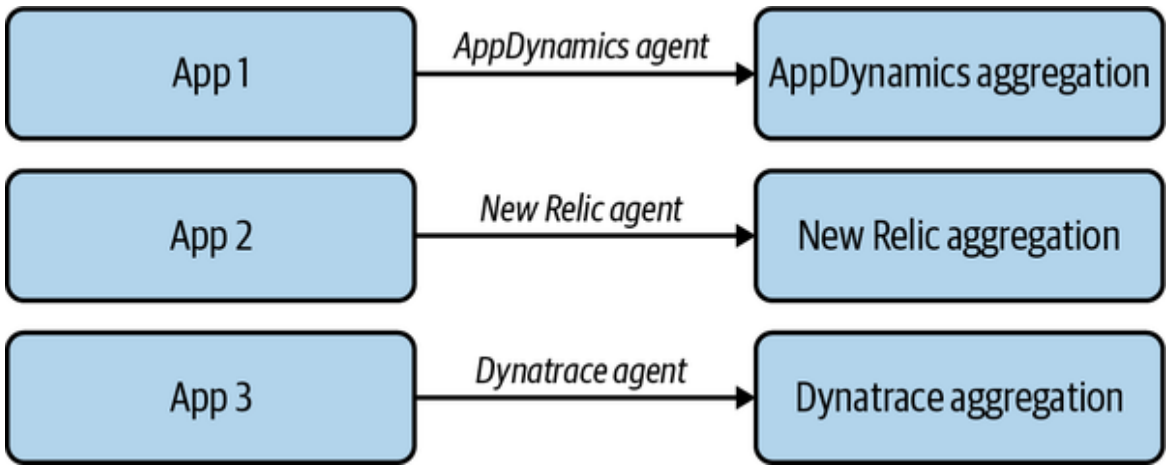


*Figure 3-1. Applications instrumented using agents*

Agent-based systems use the same system resources as the application and in some cases can slow down or even crash applications. SRE teams can't observe when agents cause performance issues since, after all, they are using the same agents to send the data back to the aggregation servers.

## User-Controlled Metric Data Collection

To solve the performance issues, the SRE team at ecommerce site Etsy wrote a new type of agent, which they called StatsD. StatsD, writes Etsy engineer Ian Malpass, is a simple open source agent that "listens for messages on a UDP [User Datagram Protocol] port."[1] It improves the performance of agent-based monitoring by sending messages over a "fire-and-forget" protocol, which minimizes system resource use and does not wait for a response from the aggregation server, ultimately improving performance.

As StatsD grew in popularity, the financial services company Stripe adopted it. Once they began using it, however, Stripe found two flaws in StatsD: StatsD uses UDP, which is inherently unreliable and does not work with metric types that rely on timer.

Stripe's engineers decided to start an open source project of their own: a metrics sink called Veneur. *Metrics sinks* send metric data to vendors or aggregators in various formats, as shown in Figure 3-2.

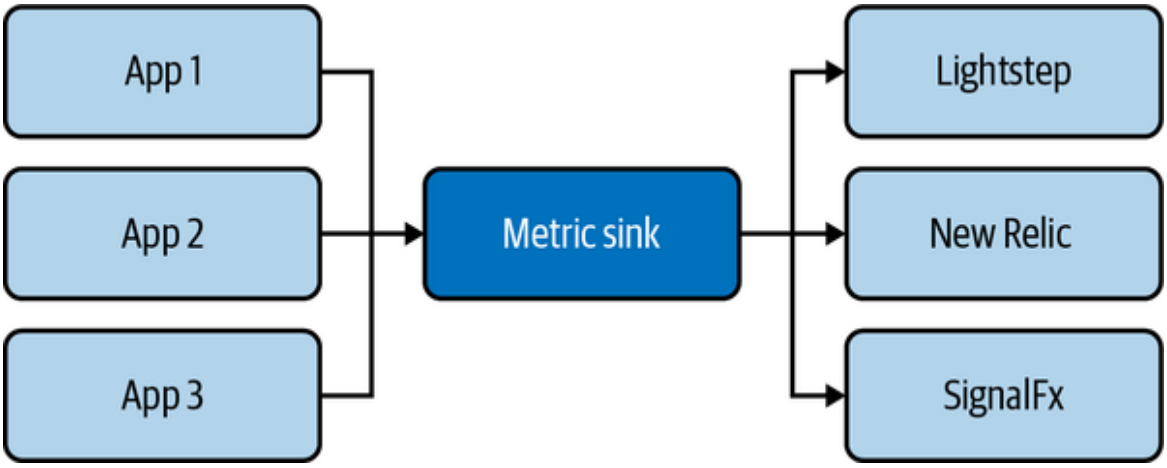


*Figure 3-2. A metrics sink sending data to three different vendors*

The problem with metrics sinks, however, is that every vendor needs a unique sink implementation. Vendors want their clients to use their native agent, so they have little incentive to write a sink implementation. Sinks make it all too easy for clients to migrate out of the vendor's product and toward the competition. Indeed, as of late 2021, Dynatrace and AppDynamics still do not have a sink implementation in Veneur.

In 2012, while most organizations were making the switch to microservices architecture, Soundcloud ran into a set of similar challenges while scaling their existing monitoring system. To solve these challenges, SoundCloud created Prometheus: a way to instrument once and output everywhere.

Prometheus is inspired by Google's Borgmon monitoring system. Instead of using a sink that pushes data to an aggregator system, Prometheus instrumentation exposes a metric endpoint (usually an HTTP endpoint in `/metrics`). The Prometheus server scrapes the metric endpoint. While most other systems are push-based, pushing data out toward an aggregator, Prometheus is pull-based. This represents a major innovation: because push-based systems must wait for servers to respond to requests, they can cause delays and performance degradation. *Pull-based* systems expose data by using a broadcast system, "listening" to and then broadcasting data

without affecting or even notifying the system producing the data. This eliminates the need for agents, and for most applications, its impact on performance is almost negligible. Figure 3-3 shows agentless metrics in Prometheus.

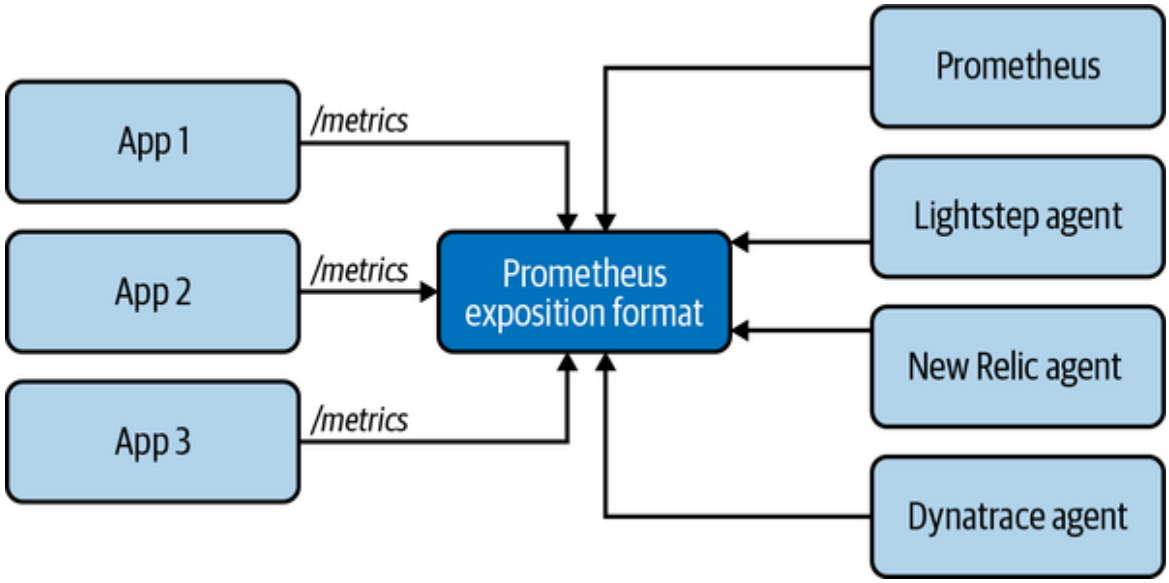


*Figure 3-3. Prometheus's exposition format, supported by vendors*

The shift to pull-based metrics collection has allowed SRE teams to better control the metrics they collect. The caveat is that for a pull-based system to be effective, it needs a standard data format to eliminate the need for conversion. Similar to Borg, Prometheus created its own exposition format, Prometheus Exposition (followed by OpenMetrics), then wrote clients that use it to expose metrics simply.

Additionally, Prometheus scrapes application endpoints asynchronously with labels included! This makes it much simpler to detect when an application is down. It also makes the metrics reliable and as scalable as the application itself. This also allows for metrics collection with more dimensions, which is a better fit for microservices-based architectures.

Metrics instrumentation becomes part of the application rather than a separate process, as shown in Figure 3-4.

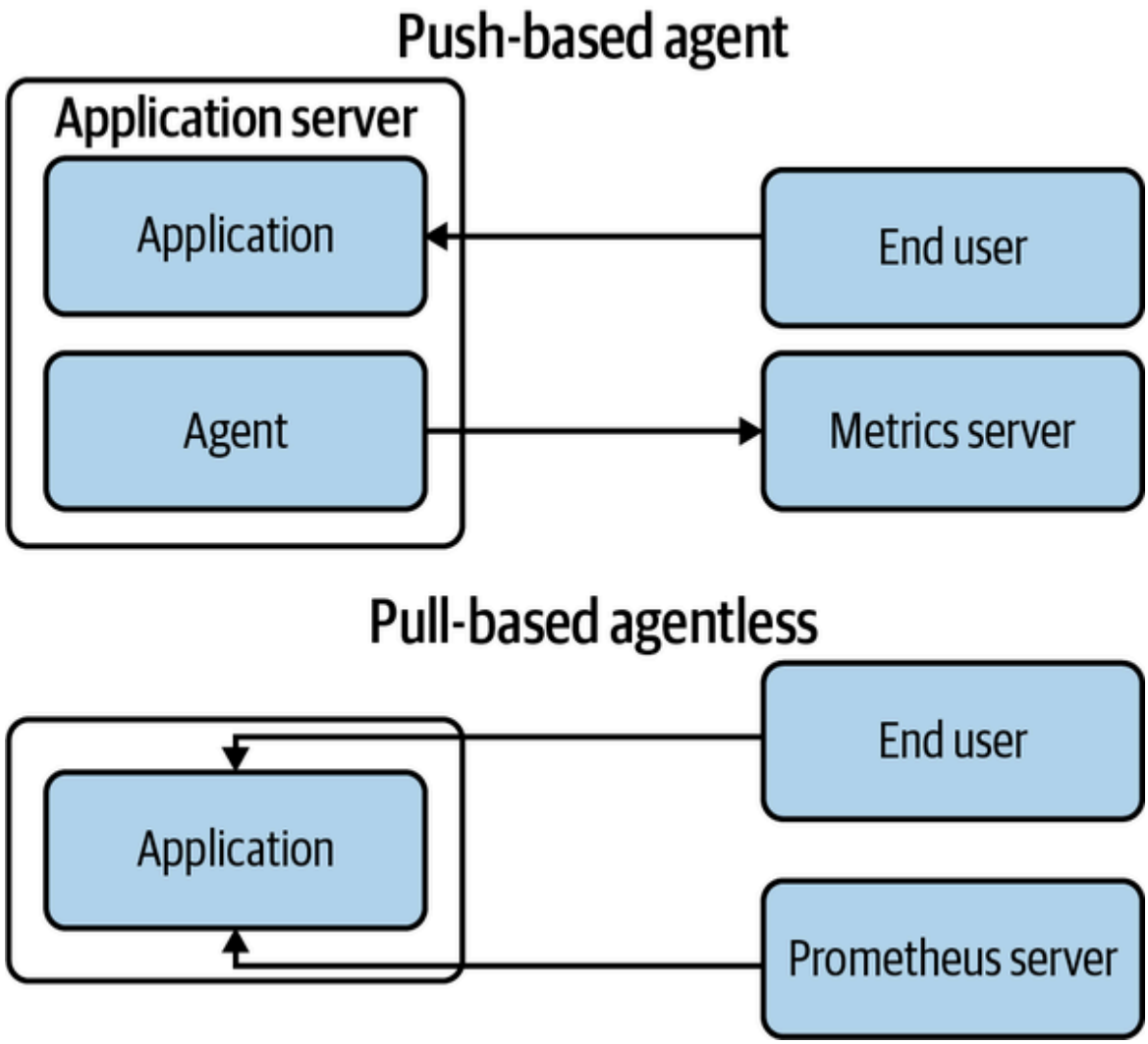


*Figure 3-4. Push-based agent instrumentation versus pull-based agentless instrumentation*

Another contributor to Prometheus's reliability is the nature of the pull model. The pull model is inherently reliable because if the collector is down then it simply waits longer to pull the metric, while the push model will fail.

Prometheus thus solved the two big problems Stripe had identified: reliability and collection scalability. It has since been adopted so widely that most open source tools in the cloud native ecosystem support Prometheus metrics exposition.

Its pervasiveness[2] became especially evident when cloud native ecosystems started to build tools and standards on top of

Prometheus. Of these, the most important is Open Metrics, a format standard that is 100% compatible with the Prometheus exposition format. This means that any toolsets or vendors that are compatible with Prometheus are now *forced to be interoperable* with each other.

Both metrics sinks and Prometheus tools and standards, then, give SRE teams greater control over their metrics instrumentation. What impact has this had on the cloud native ecosystem?

## Prometheus: The Good and the Not-So-Good

For good or bad, the industry is adopting Prometheus. After Kubernetes, it was the second project to attain "graduated" status from the Cloud Native Computing Foundation, which requires meeting stringent criteria.

Prometheus itself has many advantages, the foremost of which are:

*Dimensional metric data models*

> Prometheus uses a dimensional metric data model that allows flexibility when labeling metric data. You can use these dimensions to query metrics using the PromQL language.

*Service Discovery*

> Prometheus can use Service Discovery native to the system Prometheus is monitoring. For example, Prometheus can self-discover pods endpoints using Kubernetes's own Service Discovery APIs.

*Deep integration between PromQL and alerting*

> Prometheus has a built-in Alertmanager subsystem that can push to paging systems like PagerDuty and Slack. Alertmanager uses PromQL to build alerts and thresholds.

However, with all good systems, it has disadvantages as well:

*Generic use case*

The use case for Prometheus is too generic: it isn't built for any one type of application, so you have to configure it for your specific system, including creating metadata labels for each metric type. The relabeling configuration becomes complex as you collect more metrics.

*Annotation leads to complexity*

The more dimensions your metrics have, the more complicated it gets to configure Prometheus scraping. You can solve this problem easily by using tools like PromLens and taking care to annotate metrics only when absolutely necessary.

*Hard to operate reliably*

Prometheus is hard to operate reliably. Prometheus runs as a single binary, which means it's easy to stand up but harder to keep running on unexpected errors. Having Prometheus run in production means tweaking and fine-tuning to keep Prometheus reliable. You end up spending time on Prometheus that you could (and should!) be spending on your core business applications instead. The only exception to this rule is if your business runs Prometheus full time, as with the fully managed options we will discuss later in this chapter.

*Horizontal scalability*

The biggest disadvantage of Prometheus is that its server uses vertical scaling.

We'll discuss horizontal scaling in more depth later, but first you need to understand how scaling works, so let's take a quick detour. In general, there are two types of scaling: horizontal and vertical.

*Horizontal scaling*, also sometimes called *fan-out scaling*, is based on multiple servers, while *vertical scaling* is based on the resources of one server. Most distributed systems are scaled horizontally because it is faster and more cost-effective.

Prometheus does not support horizontal scaling out of the box. Most Prometheus servers instead scale vertically. For big Prometheus deployments, then, you need a giant server with lots of CPU and memory. This is bad for many reasons; let's look at four of them:

*Single point of failure*

> If there is an outage in the Prometheus server's region or data center, that server becomes a single point of failure. Compare that with cloud native systems, which are built on the assumption that networks are unreliable and compute is ephemeral.

*Easily overloaded*

> The Prometheus server tends to get overloaded with tasks. As more applications go online, it scrapes metric data more slowly.

*Hard to automate*

> Vertical scalability means that the Prometheus server must be treated like a pet, not cattle, as the famous analogy goes. Patches and configurations cannot be automated easily—and should not be—in case of any version incompatibility.[3]

*Limited scalability*

> In the cloud, there is a high ceiling of compute types available for machines, but this is by no means infinite. At some point, vertically scaling Prometheus is no longer an option.

That said, there is a way to scale Prometheus servers horizontally.

## Horizontal Scalability

A Prometheus server is itself an intricate web of tools. Take Alertmanager for example, a component of the Prometheus server that generates alerts on the back of PromQL-based queries. Each component has to be scaled separately. The same is true for most other Prometheus components, like HTTP Service Discovery, Prometheus Storage, Prometheus Querying API, and Prometheus WebUI.

## Self-Managed Remote Storage Options

There are two main options for horizontally scaling Prometheus: self-managed remote storage and fully managed SaaS solutions. We'll look at three self-managed options first, then move to four fully managed options.

### Thanos

Thanos adheres to two classic Unix philosophies: "do one thing well" and "keep it simple." This is important because its architecture can get complicated fast.

Thanos does not divide the Prometheus server into multiple functionality components. Instead, it uses its own APIs to extend the Prometheus server. You might say Thanos runs as a sidecar attached to a Prometheus pod.

The biggest advantage of this is that you can pick and choose which functionalities you need while running your Prometheus server similarly to how you would run a single server. Persistence is handled by your cloud provider's blob storage, and there is no need for sharding individual Prometheus components.

According to Thanos documentation, "The only explicitly scalable components are query nodes, which are stateless and can be scaled

out arbitrarily. Scaling of storage capacity is ensured by relying on an external object storage system."[4]

In short, the only parts of Thanos you need to run are the sidecar and the blob storage. Everything else is optional. The only additional cost is the price of storing and querying data.

However, the disadvantage is that blob storage performance is worse than reading data in file storage, making queries slower. File storage is not used to cache data in chunks; only index data is cached. This means that each time you run queries that would need blob storage retrieval, the performance suffers.

### Cortex

The first project that tried to horizontally scale Prometheus using remote storage was Cortex, famously dubbed the "Frankenstein project" because it stitched together a big cobweb of functionalities, illustrating the difficulty of this task. It has since compacted some of these disparate functionalities into a smaller set of logical units to run.

Cortex has a different approach from Thanos's minimalism: it designed a set of microservices that individually reuse codes and shards from Prometheus libraries. Its architecture (Figure 3-5) essentially reimagines Prometheus, recreating various components to work as if Prometheus had originally been written with horizontal scalability in mind.

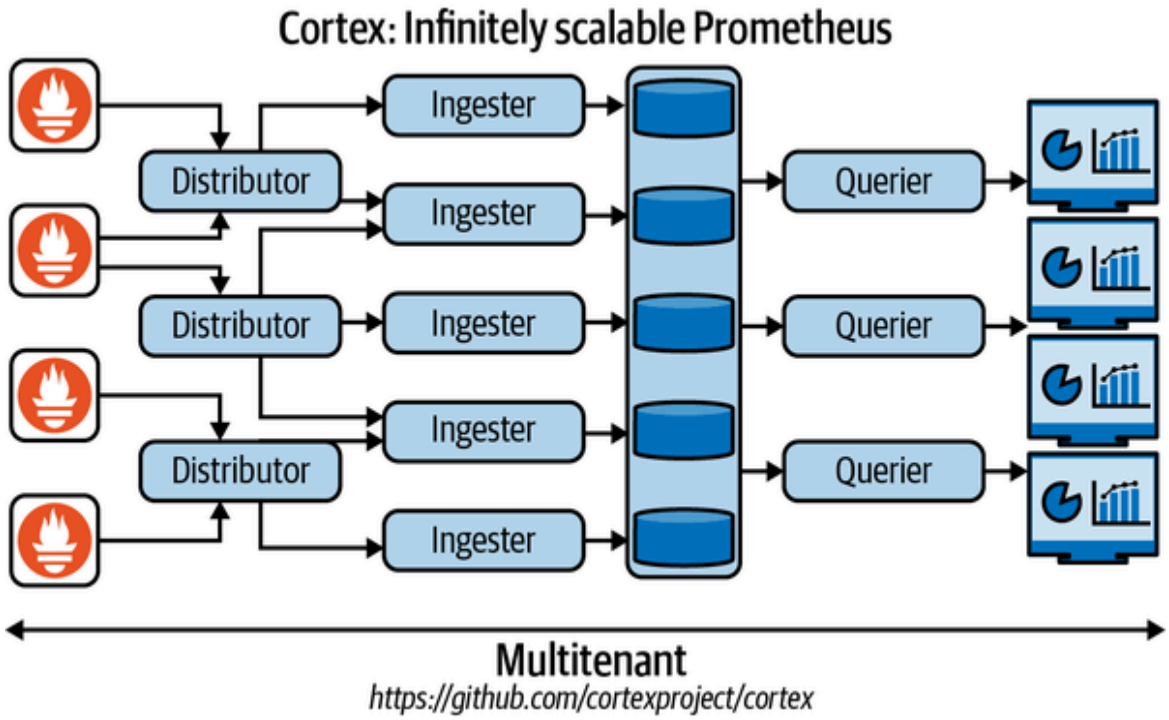


*Figure 3-5. High-level concept diagram of Cortex's architecture*[5]

Cortex separates reads from writes. The read part of the architecture uses Cortex Querier to create a horizontally scalable way to query Prometheus remote storage. Querier heavily relies on caching query results, queueing large queries, and splitting large queries into different Queriers to produce quick results.

The write part of Cortex is much more complex. Cortex Distributor handles incoming data from Prometheus scraping, ensuring it is valid and not duplicated. It uses a hash ring algorithm with a key value database (like *etcd* or *consul*) to ensure high availability.

Storage is done through the Cortex ingesters, which are also highly available and semistateful. They achieve high availability by splitting the storage using a hash ring algorithm, similar to writes. The Cortex ingester retains the data in memory until a batch job is run, then puts the data into long-term object storage.

You may have noticed some parallels with Thanos. Cortex is much more complicated to run than Thanos, but also generally more

scalable since it shards individual components and runs them as microservices.

In March 2022, Grafana Labs announced they forked Cortex and relicensed it as Mimir under the AGPLv3 license. As a result, they will no longer contribute to Cortex. Since Grafana Labs employees were the primary maintainers of Cortex, its future remains uncertain as of this report's publication. For the time being, it continues to be a project licensed under Apache 2 as part of the CNCF, and new maintainers will need to maintain the project independently from Grafana's fork, Mimir. As a fork, Mimir reuses similar components and architectural principles as Cortex.

### M3

M3 provides a turnkey, scalable, and configurable multitenant store by using M3DB, its scalable backend-storage solution, as a Prometheus remote storage backend. M3 Coordinator acts as a Prometheus sidecar to horizontally scale writes and reads.

Not only do M3DB instances scale Prometheus's storage horizontally, but they can also improve its performance. Because M3 reads and writes to disk rather than blob storage, reads and writes can be much faster, as they are with Thanos or Cortex. The trade-off is that M3's storage is not infinitely scalable. In practice, this is less of a concern when running in the cloud, where you can request block storage and nodes on demand. Additionally, M3 and Cortex are more complicated to run than Thanos because they use sharded architecture for individual components.

Both Thanos and Cortex have larger communities than M3DB does. At a production-ready scale, however, M3DB has proven to be a good fit for big companies like Walmart and Uber. M3 also allows for more specific granularity in retaining metric data: for example, all metrics tagged with `keep_30days` can be kept for 30 days with a simple configuration. You can also downsample metric data in long-term storage to minimize its size.

In a review of M3DB for Prometheus-as-a-service, Logz.io notes that its capabilities are "not as robust as one would expect for a production-level database."[6] Be sure to run your own tests to assess whether it fits your use case.

### Choosing the right self-managed remote storage option

The primary drawback to self-managed remote storage is that it is complicated to build and run with high levels of availability. The lifecycles and configurations of these three systems (M3, Thanos, Cortex) require specialized knowledge of the inner workings of both that system and Prometheus.

How much do ease of configuration and performance matter to you? If you need high-performance reads and writes with easy configuration and proven production scale, running M3 may be preferable. If cost matters more, Thanos might be the right option, since it utilizes cheaper blob-storage services. If, however, community-wide adoption and ease of scaling matter most to you and you have a dedicated team that is not afraid of complexity, then Cortex is likely your best choice.

## Fully Managed SaaS Options

Having a good understanding of your organizational structure, goals, objectives, and tech stack is key to making any well-informed decision. When it comes to choosing between fully managed SaaS options, however, we think that *heuristics*—hands-on exploration—are most useful. We encourage you to look at four specific heuristics:

*Prometheus conformance*

> Prometheus conformance measures what it takes to lift and shift metrics from your old agent-based system to Prometheus. The more your option conforms to Prometheus, the more flexibility you get to move between vendors and open source. You also get

backward compatibility to any version of Prometheus in the open source ecosystem, so you can access all the benefits of both the open source and proprietary tools.

*Operational integration*

It is much easier to implement a system that has native integration with your existing system; otherwise, you have to build translation APIs. Having a good level of integration means you can easily get up and running and start to realize a return on your investment.

*Full observability feature sets*

Feature sets let you use existing functionalities in your fully managed option rather than building things on top of that option. This saves time and allows you to focus on building observability tailored to your organization's needs.

*Reliability*

In our view, the most important heuristic of any observability system is reliability. After all, if you are outsourcing your monitoring system to a fully managed system, the last thing you want is unreliable monitoring in the middle of an incident. Reliability heuristics means that your monitoring solution should allow you to monitor your systems consistently without fail, even during massive internet outages, *force majeure* events, and cybersecurity incidents, and with all the scalability you require to understand your system.

All of the fully managed services here, with the exception of Chronosphere and Google, use Cortex as their base platform. Chronosphere uses M3.

### Amazon Managed Service for Prometheus and Google Cloud

### Managed Service for Prometheus

Amazon Managed Service for Prometheus and Google Cloud Managed Service Prometheus are both good choices if you want full to high conformance with Prometheus. Both have good native integration with existing Prometheus setups on their respective cloud environments. Both are also very reliable, since they are running highly scalable cloud environments.

The downside is that neither offering is a full observability platform built on top of Prometheus; rather, they function as a managed remote storage solution for Prometheus. This means that platform features like Prometheus Alertmanager (for alerting based on Prometheus queries) and Grafana (for dashboarding) still have to be built separately. It will take time for your SRE team to add the functionality needed to extend into a full platform.

Another downside to using either of these fully managed services is that if it is hosted in the same region as your production cluster, you will not get alerts at all if your production applications are down.

### Grafana Cloud

Grafana Cloud is a full observability platform offering some of the features SRE teams use most, like dashboarding, synthetic monitoring, support alerting with integrations, and OnCall functionalities. Grafana Cloud uses Mimir (a fork of Cortex) behind the scenes, allowing low-cost storage and infinite scalability, and is fully compliant with Prometheus.

Integration from Prometheus Open Source to Grafana Cloud should be a seamless switch; switching from Prometheus to Grafana Cloud will be a straightforward and familiar experience for most SRE teams.

As for reliability, Grafana Cloud has experienced multihour outages on its shared infrastructure in 2021 and 2022.[7] They attribute these outages to the complexity of running Cortex/Mimir.

### Chronosphere

Chronosphere is a full observability platform built on top of M3 and is fully conformant with Prometheus. It integrates natively with both Prometheus-based systems and StatsD systems like Graphite.

It includes feature sets that make SRE teams' lives easier, like a Query Builder to facilitate writing PromQL. Dashboarding is available via a Grafana-compatible user interface and supports both StatsD and Prometheus metrics. The tight linking between traces and metrics also allows for more contextual visibility.

Operationally, Chronosphere allows for control mechanisms like rate limiting, or limiting the number of metrics each team is allowed per second. Profiling metrics describe how much volume is coming from individual Prometheus labels. In addition, Chronosphere's performance reacts to changes in the metric data being queried or stored. However, it does not yet support logs out of the box.

Chronosphere is three years old, but there have been no documented outages as of this writing.

## Choosing Between Self-Managed and Fully Managed

As discussed previously, horizontally scaling self-managed Prometheus is difficult. There are only two scenarios where we recommend running self-managed. The first is if your main business is running observability systems—that is, if you are a company like Chronosphere, Grafana Labs, or New Relic. The second scenario is if your requirements for scaling are sufficiently big that any of the fully managed options would be too small for your organization. This would only apply to large enterprises on the scale of Amazon, Google, Facebook, or Apple.

Otherwise, our recommendation is to go with fully managed options. In our view, any cost savings you could achieve by running

self-managed Prometheus would be offset by the high total cost of ownership when horizontally scaling.

---

1 Ian Malpass, "Measure Anything, Measure Everything," Etsy, February 15, 2011, *https://oreil.ly/GQpxT*.

2 Anne McCrory, "Ubiquitous? Pervasive? Sorry, They Don't Compute," *Computerworld*, March 20, 2000, *https://oreil.ly/juHHV*.

3 "What [would] happen if several of your servers went offline right now?" Viktor Farcic asks. "If they are pets, such a situation will cause significant disruption for your users. If they are cattle, such an outcome will go unnoticed. Since you are running multiple instances of [a] service distributed across multiple nodes, failure of a single server (or a couple of them) would not result in a failure of all replicas. The only immediate effect would be that some services would run fewer instances and would have a higher load." See Farcic, *The DevOps 2.1 Toolkit: Docker Swarm* (Packt, 2017).

4 "Architecture," Thanos, accessed March 16, 2022, *https://oreil.ly/XPHpF*.

5 Adapted from an image in Tom Wilkie, "Grafana Labs at KubeCon: The Latest on Cortex," Grafana Labs, May 21, 2019, *https://oreil.ly/QMYZ3*.

6 Jonah Kowall, "Why We Chose the M3DB Data Store for Logz.io Prometheus-as-a-Service," Logz.io, March 24, 2021, *https://oreil.ly/JqtLQ*.

7 Tom Wilkie, "How We Responded to a 2-Hour Outage in Our Grafana Cloud-Hosted Prometheus Service," Grafana Labs, March 26, 2021, *https://oreil.ly/TX0hk*; "Grafana Cloud Intermittent Loadbalancer Errors," Grafana Labs, accessed March 16, 2022, *https://oreil.ly/d4HiG*.

# Chapter 4. Strategies for Controlling Metric Data Growth

Metrics are growing in scale, and that means the data they produce also grows exponentially, far outpacing the growth of the business and its infrastructure (as shown in Figure 4-1). Growth this fast causes problems: storing *all* of the data your metrics produce about everything (logs, metrics, and traces) would be prohibitively expensive in terms of both cost and performance. In a survey of 357 IT, DevOps, and application development professionals by ESG, 71% saw the growth rate of observability data as "alarming."[1]

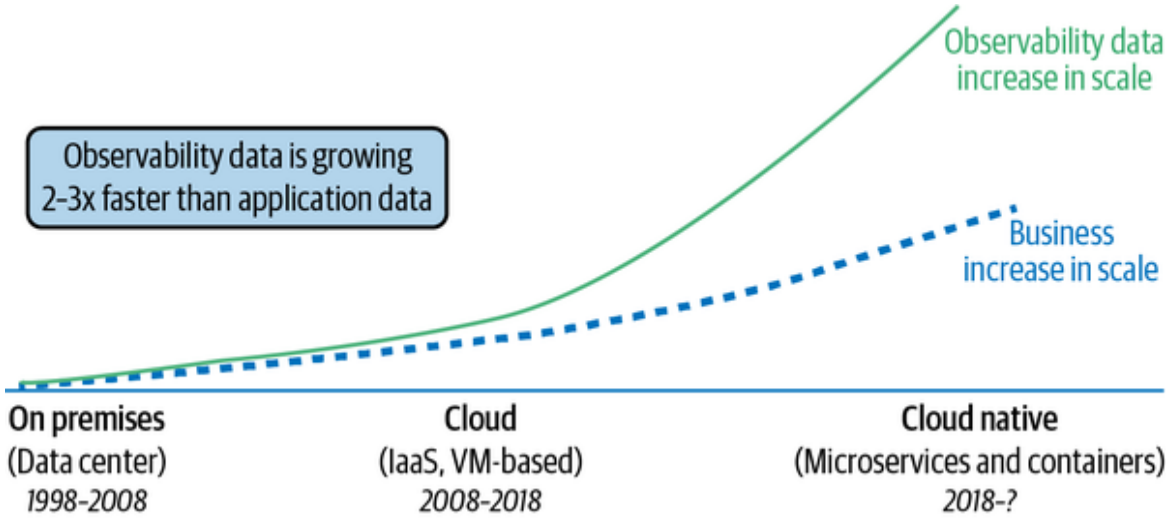


*Figure 4-1. Cloud native's impact on observability data growth*[2]

Why so much growth? The reasons include faster deployments, a shift to microservices architectures, the ephemerality of containers, and even the cardinality of metric data itself. This causes a dilemma: how should you identify which metric data is worth storing?

To answer this question, you need to understand the major use cases for metrics in your organization. Look at RPC traffic, request/response rates, and latency, ideally as they enter your system. If you have, for example, 100 microservices, how many dimensions should you add to your metrics? Should you capture all data as it comes in for each metric?

Metrics cardinalities can generally be classified into three types, as Chronosphere's John Potocny notes:[3]

*High-value cardinality*

> These are the dimensions we need to measure to understand our systems, and they are always (or at least often) preserved when consuming metrics in alerts and dashboards.

*Low-value or incidental cardinality*

> The value of these dimensions is more questionable. They may be an unintentional by-product of how you collect metrics instead of dimensions that you purposefully collected.

*Useless or harmful cardinality*

> Collecting useless or harmful dimensions is essentially an antipattern, to be avoided at all costs. Including such dimensions can explode the amount of data you collect, resulting in serious consequences for your metric system's health and significant problems querying metrics.

To determine which type of cardinality you're dealing with, look back to the principle we laid out in the beginning of this report: *take an outcomes-based approach.* Is the data you're getting useful for remediation? How does it affect your customers' outcomes? Is it really necessary to capture this data through a metric, or could you capture it through traces or logs instead? If you're not getting what you need to solve your problems, what's missing? If you're getting

metric data that you don't or can't use, can you identify what isn't helpful?

Next, we'll look at the three key strategies of an outcomes-based approach: retention, resolution, and aggregation. They are useful whether your approach is fully managed or self-managed. Note that Thanos and Cortex do not have special resolution and aggregation functionalities like M3 does; they use Prometheus's default capabilities instead.

## Retention

How long are you keeping your data? Prometheus's default retention period is 15 days. However, most organizations we've worked with retain all metric data for 13 months, whether it relates to production, staging, or even development environments! Do you actually need or use 13 months' worth of metric data?

Let's say you're collecting metrics for development environments and retaining them for 13 months. Is that useful if the development environment gets recycled every week? What if you retained those development metrics for a few weeks instead?

Don't just stick with the default: base your retention periods for different kinds of data on the outcomes that you can gain by retaining it. If you reduce the retention period for data that you do not need, the overall volume will grow at a much more reasonable rate.

In Prometheus, you can configure retention globally by using `--storage.tsdb.retention.size` and `--storage.tsdb.retention.time` flags at startup. The self-managed remote storage systems Thanos and Cortex use blob storage as an ultimate backend store, suggesting the possibility of infinite retention. However, M3 has a different approach, which we'll examine in a moment.

## Resolution

How *often* do you collect metric data? Can you improve your outcomes by collecting data more frequently? The frequency of data collection is called *resolution*—more data points, just like more pixels in a photograph, would mean a higher resolution.

Let's say your development environments are deploying multiple times a day. Do you need per-minute metrics for all of them? Perhaps for some applications that would be helpful. But other environments or applications might need only metrics every 10 seconds or every minute. Some might not need metrics at all! Collecting at a lower resolution for certain environments can drastically reduce your metric data's growth.

Using Prometheus, you can resolve individual scrape jobs using `scrape_interval`. This example below resolves metrics every minute:

```
- job_name: nginx_ingress
  scrape_interval: 1m
  scrape_timeout: 10s
  metrics_path: /metrics
  scheme: https
```

Feel free to make the `scrape_interval` as granular as possible, since it is tied to individual scrape jobs.

## Applying Resolution and Retention in M3

If you are using M3, you can apply both resolution and retention strategies with the *mapping rules* feature. A helpful doc by M3 examines the following rule:[4]

```
downsample:
  rules:
    mappingRules:
```

```
- name: "mysql metrics"
  filter: "app:mysql*"
  aggregations: ["Last"]
  storagePolicies:
    - resolution: 1m
      retention: 48h
- name: "nginx metrics"
  filter: "app:nginx*"
  aggregations: ["Last"]
  storagePolicies:
    - resolution: 30s
      retention: 24h
    - resolution: 1m
      retention: 48h
```

In this rule, the authors note, "We have two mapping rules configured—one for `mysql` metrics and one for `nginx` metrics. The filter determines what metrics each rule applies to. The `mysql metrics` rule will apply to any metrics where the `app` tag contains `mysql*` as the value (`*` being a wildcard). Similarly, the `nginx metrics` rule will apply to all metrics where the `app` tag contains `nginx*` as the value."

M3's metric data storage policies can define dynamic settings that allow for a mix of retention and resolution, unlike those of Thanos and Cortex, which only has a single resolution and retention setting for all the metrics when scraping metrics, supported by default in Prometheus.

## Aggregation

Aggregation is perhaps the most effective of these three strategies. Most applications that generate metrics generate high cardinality and volume by default.

Imagine you have a web application that is running behind NGINX Proxy. NGINX produces very granular HTTP metrics: when you start

scraping metrics, by default, it's all or nothing. Do you really need data for *every* dimension? If you're trying to measure the latency of your HTTP responses, will it be useful to know the NGINX version or which data center it's running on?

Combining the right dimensions and dropping any that aren't useful is important in controlling the growth of your metric data.

Using the aggregation strategy is about paying attention to what you're capturing, choosing the metrics and dimensions you really need and combining them, and dropping the rest to keep metrics growth under control. Additionally, the aggregation strategy can be used in conjunction with resolution and retention strategies.

In Prometheus, you can aggregate metrics via *recording rules*, which periodically compute expensive queries in the background, such as aggregating and dropping a high-cardinality dimension.

While recording rules can help improve performance by computing aggregates, they do not make it possible to drop the original data; doing so requires a federated setup and is quite cumbersome to manage.

To successfully reduce metric data through aggregation in Prometheus, you have to federate Prometheus instances and then combine and drop metrics via *recording rules*, as in Figure 4-2.

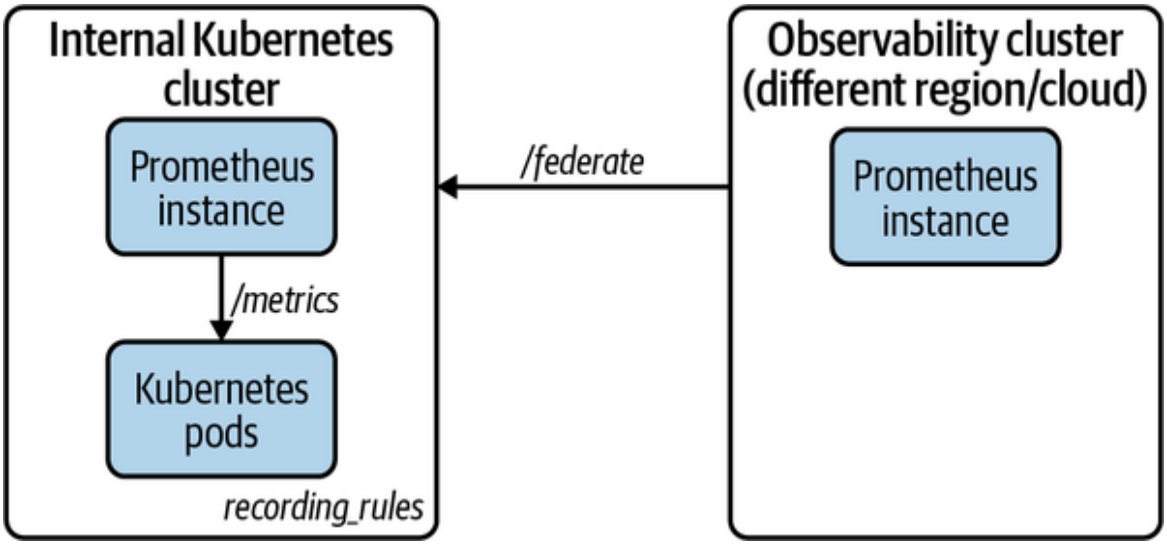


*Figure 4-2. Showing a combination of federation and recording rules to aggregate metrics*

The reason we aggregate then federate is to work around the limitation of having to store data in Prometheus before we can use recording rules to aggregate it. In this setup, we store, then aggregate, then forward only the aggregated data to another Prometheus instance. This allows us to realize a net reduction in metric data at our final storage location.

Here is an example of a *recording rule* for aggregation:

```
groups:
 - name: node
   rules:
    - record: job:process_cpu_seconds:rate5m
      expr: >
        sum without(instance)(
          rate(process_cpu_seconds_total{job="node"}[5m])
        )
```

This runs the `rate(process_cpu_seconds_total{job="node"}[5m])` query, dropping the dimension instance, then creates a new metric, `job:process_cpu_seconds:rate5m`. This new metric has a lower cardinality than the original `process_cpu_seconds` metric.

## Applying Aggregation in M3

Applying aggregation in M3 has a similar effect as in Prometheus. The main difference is that M3 has functionalities called *mapping rules* and *rollup rules*, which, when combined, drop unnecessary metrics efficiently by performing aggregation before anything is written and choosing what to keep. This makes aggregating cost-effective. These functionalities also allow M3 to do the same aggregation as Prometheus—without needing to create a federated setup. Consider an example from M3:[5]

```
downsample:
  rules:
    mappingRules:
      - name: "http_request latency by route \
          and git_sha drop raw"
        filter: "__name__:http_request_bucket k8s_pod:* \
          le:* git_sha:* route:*"
        filter: "__name__:http_request_bucket k8s_pod:* \
          le:* git_sha:* route:*"
      - name: "http_request latency by route \
          and git_sha without pod"
        filter: "__name__:http_request_bucket k8s_pod:* \
          le:* git_sha:* route:*"
      - name: "http_request latency by route \
          and git_sha without pod"
        filter: "__name__:http_request_bucket k8s_pod:* \
          le:* git_sha:* route:*"
            metricName: "http_request_bucket" \
              # metric name doesn't change
            groupBy: ["le", "git_sha", "route", \
              "status_code", "region"]
        filter: "__name__:http_request_bucket k8s_pod:* \
          le:* git_sha:* route:*"
            metricName: "http_request_bucket" \
              # metric name doesn't change
            groupBy: ["le", "git_sha", "route", \
              "status_code", "region"]
        filter: "__name__:http_request_bucket k8s_pod:* \
          le:* git_sha:* route:*"
```

```
    metricName: "http_request_bucket" \
      # metric name doesn't change
    groupBy: ["le", "git_sha", "route", \
      "status_code", "region"]
    type: "Increase"
- rollup:
    metricName: "http_request_bucket" \
      # metric name doesn't change
    groupBy: ["le", "git_sha", "route", \
      "status_code", "region"]
    aggregations: ["Sum"]
- transform:
    type: "Add"
storagePolicies:
- resolution: 30s
  retention: 720h
```

The *rollup rule* above eliminates the `k8s_pod` label for the `http_request_bucket` metric that we're matching against. To do this, we add up the `http_request_bucket` metric grouped by the other dimensions it has that we want to keep. In addition, we pair it with a mapping rule that drops the original data, which allows us to retain the original `http_request_bucket` metric name rather than creating a new metric name for the aggregate.

## Conclusion

Explosive data growth does not equate to better observability—there is so much more to it. Finding the right balance between too much information and not enough is key. The metrics you capture and retain should be useful to your business goals and outcomes and should measure crucial business and application benchmarks. These three key strategies—resolution, retention, and aggregation—can help you control the growth of your metric data, so you're only getting and keeping what counts. Remember, it's all about outcomes.

1 Rachel Dines, "New ESG Study Uncovers Top Observability Concerns in 2022," Chronosphere, February 22, 2022, *https://chronosphere.io/learn/new-study-uncovers-top-observability-concerns-in-2022*.

2 Adapted from an image by *Chronosphere*.

3 John Potocny, "Classifying Types of Metric Cardinality," Chronosphere, February 15, 2022, *https://chronosphere.io/learn/classifying-types-of-metric-cardinality*.

4 "Mapping Rules," M3, accessed March 16, 2022, *https://oreil.ly/xJ5wT*.

5 "Rollup Rules," M3, accessed March 16, 2022, *https://oreil.ly/Oz5eT*.

# Chapter 5. Building Great Metrics Functions

To give you more ideas on how to build a great metrics function, we've compiled some tips and tricks in this final chapter.

## Out-of-the-Box Standard Instrumentation and Dashboarding

Life is increasingly complex in the DevOps world. Software engineers are expected to do much more than write code: we productionize applications, architect and build systems, make everything compatible across the organization, and ensure that everything we make follows standards.

Thanks to open source tools like Envoy and NGINX, though, SRE teams and software engineers can enable standardized metrics and dashboards right out of the box. For example, if you're collecting Prometheus metrics with Envoy or NGINX, you can give any HTTP-based application a Grafana dashboard. If you need organization-specific metrics, such as for sales on a specific API, you can build a sales dashboard for use by any team that implements the API. You can even create a RED (Request *R*ate, Request *E*rror, Request *D*uration) metrics dashboard out of the box, so that your software engineers can observe and monitor these customized, prebuilt applications without needing to learn PromQL or add intrusive instrumentation to their code.

Rely on your software engineers or SRE/observability teams to help you build and create standardized dashboards. They know the organizational context better than any vendor. For example, if your organization relies heavily on Remote Procedure Calls (RPCs), you'll

want to create RPC dashboards and alerts. You can collect these metrics using appropriate middleware (such as the Java or Go Prometheus gRPC middleware libraries) or metrics exposed by an RPC proxy. (Envoy, for instance, supports proxying gRPC traffic and exposes the corresponding metrics). If your main event bus management system is Kafka, you could start collecting metrics with the Kafka metrics exporter for Prometheus, then create dashboards and alerts that monitor all Kafka topics for each application. What do your colleagues need to know to achieve their desired outcomes?

## Adding Business Context to Standardized Metrics

Once you have a respectable toolset of standardized metrics and dashboards, it's time to tie them into your business context.

Perhaps you decide to enrich your metrics by adding a new label, `client_name`, to the standard out-of-the-box instrumentation. You can use `client_name` to understand how traffic is being served between different clients. For example, if `client_name` ACMECorp is creating more requests to your services, then you can scale the servers hosting ACMECorp and possibly send an email to ACMECorp about the increased usage.

Another example of adding business context is customizing how you route alerts. You might add the names of the app and the team that owns it to the alert, so it's always routed to the right people. You can even go one step further by adding dependent applications. This way, the alerts go not only to the team that owns the app but also to the teams downstream from that app. This allows the organization to detect a cascading failure and pinpoint where it starts.

Tiering applications is another common use case for adding business context to metrics. Some businesses have systems that should *never* stop running, under *any* circumstances—if they did, it would cost the business more than any other applications. You can add labels with tiers to differentiate the most crucial applications, then route alerts differently based on tier. This means that if a system labeled tier=1 goes down, you can alert not only to your technical staff but also sales, PR, and the executive team, allowing everyone to act quickly and get ahead of potential problems.

## Creating SLOs from Standardized Instrumentation

Service levels, which you can derive from your metrics, are a great way to align site reliability with your business goals. Stavros Foteinopoulos of Mattermost lays out three concepts that are key to understanding service levels: *service level indicators* (SLIs), *service level objectives* (SLOs), and *service level agreements* (SLAs).[1]

Foteinopoulos defines the SLI as a "carefully defined quantitative measure of some aspect of the level of service provided"—in short, a metric. An SLO is "a target value or range of values for a service level that is measured by an SLI," or what you want your metric's value(s) to be. An SLA is a contract that promises users specific values for their SLOs (such as a certain availability percentage) and lays out the consequences if you don't meet those targets (SLOs).

Let's say your company builds an API that provides memes about cats for app developers to use. Your SLI is the percentage of time your API is available to all downstream external customers. If your SLO is 99% availability, that means you've promised your customers no more than 3.65 days of downtime per year. This is measured by an error rate formula, since down time is at its core a kind of error. The formula, as Foteinopoulos notes, is:

*Error Rate = Error Requests / Total Requests*

If downtime errors exceed the limit specified in the SLO, your SLA states that you agree to donate $1 to an animal shelter per additional minute of downtime.

The beauty of SLAs is that they are not restricted to external customers. Some companies use internal SLAs to assign employees an "error budget," then rotate staff between software engineering and SRE based on whether their error counts stay within the specified values. Steven Thurgood of Google writes in *The Site Reliability Workbook* that error budgets "are the tool SRE uses to balance service reliability with the pace of innovation."[2]

If you use availability as an SLI, you've likely already instrumented a set of standardized metrics, like the Prometheus RED metrics. You can use those metrics to build a dashboard, which will make it fairly straightforward to create realistic SLOs based on your performance. Sloth by Slok (Xavier Gallego) is a tool for onboarding service levels. It lets you create Prometheus rules and Grafana dashboards quickly and easily, in a way that stays up-to-date as you add new applications to your system over time.

In addition to using standardized metrics like the RED metrics, it's also important to standardize across the organization what each SLI means. For example, what does 99% availability mean? Does that mean if any error occurs in a time window, it's not available? Does it mean the error rate has to be above 99%? What is the time window we're checking? Every minute, every hour? The answer to this varies, but the rule of thumb is that it needs to be consistent across an organization.

## Monitor the Monitor

Who monitors the monitoring system? Do you even know? What happens when it goes down?

For the sake of reliability, your monitoring should not live in the same region where you run the infrastructure. If it does and the region is misconfigured or runs out of resources, then you can no longer monitor the infrastructure within that region.

If possible, use multiple regions or a region separate from where your application lives. This way you can monitor the cluster and applications even when there is a regional data center outage. This matters whether you run your self-managed monitoring system in the cloud or data center or even if you are utilizing a fully managed option.

Ideally, you'll want to use a different cloud provider from the one you use for production workloads. This ensures you will still have a monitoring system even if an entire cloud service goes down. SaaS fully managed service providers usually use cloud providers as well. Ensure that their system does not utilize the same provider or the same region where your production workloads are running.

Finally, use an external probe on the internet (if it is a publicly facing application) or an internal probe in a different region but within your network. This gives your engineers real-time user monitoring: they can see what your end users see. End users' experiences, whether they are external or corporate, should always be your top priority when monitoring applications.

## Write and Read Limits

Because metric cardinalities are multiplicative, one engineer can write one query, but that query can read metrics with a cardinality in the order of millions or, in the absolute worst case, billions. Imagine, for example, that a software engineer accidentally lists *all* of the cardinalities within Prometheus by using a wildcard star (*).

No matter how powerful your backend system is, whether you are writing metrics or reading them, you cannot safely guarantee that

your system will not be overloaded. Build a detection system that allows you to understand if one of your queries or writes will cause an outage.

For write use cases, show your developers the cost of each metric write they add by allowing them to monitor their publishing rate. Once they can view this, they can see their usage relative to other applications and teams in their organization, and it is easy to show them when they are using more than their fair share of resources. This will help them assess and think twice about whether they really need a metric before adding it to the observability system. Furthermore, once they can view writes, you can start blocking these writes.

For read use cases, remember that every query uses the same set of resources. Users should only query for the data they actually need and make sure the cardinality matches their appropriate use. For instance, if you never query the P99 latency on a per container or pod basis, then there is no need to query the high cardinality metrics from your dashboards or alerts. Instead, query aggregated metrics that don't have the pod or container granularity (either using recording rules or an intermediate aggregator).

## Safe Ways to Experiment and Iterate

Let's say you have a basic monitoring system. You want to keep it modern and highly upgradeable, but the observability tool's landscape changes too fast for you to keep up. What's more, SREs are in high demand and tend to have high turnover. This can create a sense of fear that anything you touch will break the observability system and lead SREs to conclude that it is much more prudent to leave things as they are.

The problem with that is that you lose the ability to experiment and iterate and to learn new technologies and tools without breaking the system observability tech stack that your team is used to

running. This causes a downward spiral of being stuck but also responsible for innovating. This can feel like trying to run a marathon with your shoelaces tied together.

How do we get away from this dilemma? A lot of it is about making it safe to create a whole new set of monitoring data (or a subset of it) in another observability system. Simply spin up another observability system like Prometheus that can scrape data from the `/metric` endpoint every 10 seconds instead of every second (see Figure 5-1). Try to get 10% of all production observability data onto the new system without your developers doing anything additional.

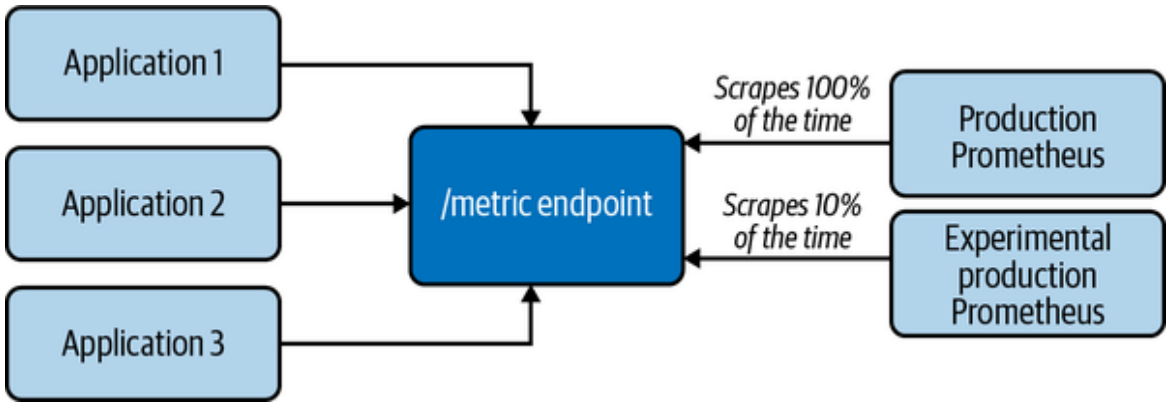


*Figure 5-1. Production Prometheus and experimental production Prometheus with 10% of production data*

You can use this different set of observability systems to experiment with things like new relabel rules, upgrading your Prometheus version, creating a new aggregation across different types of metrics, ingesting a new set of data from a completely different type of tech stack, and measuring the daily metric size.

You can also use it to iterate safely on existing tasks, like adding new cardinality to existing metrics. You can even destroy and recreate your observability stack to teach new SRE engineers how its system works. The idea is to make it safe for your SRE team to iterate and experiment without your developer needing to change or redeploy code. This removes the element of fear and empowers

your teams to learn more about your observability system and your entire application suite.

---

1 Stavros Foteinopoulos, "How We Use Sloth to Do SLO Monitoring and Alerting with Prometheus," Mattermost, October 26, 2021, *https://oreil.ly/e35u8*.

2 Steven Thurgood, "Example Error Budget Policy," in *The Site Reliability Workbook* (O'Reilly Media, 2018), *https://oreil.ly/yEg2b*.

## Conclusion

The monitoring and observability landscape has changed greatly over the past three to five years. System architectures are sufficiently different from their pre–cloud native counterparts to demand a new paradigm, born from radically rethinking, as an industry, how we build and implement monitoring systems.

This kind of radical rethinking allows new ideas to emerge. One such idea, as you've learned, is Sridharan's "three pillars of observability." Although it's a flawed paradigm that focuses on the data rather than the outcome, it's also an inspiring glimpse into what's possible.

To refine our focus and make a discernible impact, our thinking about cloud native monitoring must pivot away from the "three pillars" and toward the "three phases of observability" we've outlined in this report. These three phases allow for a goal-driven, pragmatic approach to cloud native monitoring that emphasizes remediating problems and improving business outcomes.

Open source solutions continue to drive innovation, even as more enterprises adopt cloud native monitoring. The companies encouraging them to do so, like Chronosphere, Grafana Labs, Weaveworks, Amazon, and Google, are adopting more and more open source technologies, as well as creating and using frameworks that allow cross-compatibility between various technologies and organizations.

We've emphasized metrics in this report because they can provide a high-level overview of everything happening in your system, at a fraction of the cost of logs and traces. Mathematical modeling, together with metrics aggregation, can provide sufficient alerting to allow you to remediate issues quickly and efficiently. It's not an exaggeration to call metrics the keystone of your cloud native observability solution. However, as your systems grow more

complex, the metric data you collect will also grow exponentially—and this can present problems when it comes to cost and storage.

We recommend fully managed monitoring solutions over self-managed, because implementing and maintaining the latter can be complex and costly. Ultimately, though, either type of solution can make it easier to collect, aggregate, and analyze metrics and can help you manage metric data growth while optimizing for different strategies based on your needs.

Building a great metrics function is all about finding the right strategy. Keep your eyes on the prize by focusing on outcomes, and your cloud native monitoring journey will be off to an exceptional start.

## About the Authors

**Kenichi Shibata** is a cloud native architect at esure, specializing in cloud migration and cloud native microservices implementation using infrastructure as code, container orchestration, host configuration management, and CI/CD. He has production experience in Kubernetes in a highly scalable and highly regulated environment.

**Rob Skillington** is the cofounder and CTO of Chronosphere. He was previously at Uber, where he was the technical lead of the observability team and creator of M3DB, the time-series database at the core of M3. He has worked in both large engineering organizations such as Microsoft and Groupon and a handful of startups. He and his family are based in New York City, where he mainly spends weekends exploring all of New York's playgrounds and also following his wife's jazz adventures.

**Martin Mao** is the cofounder and CEO of Chronosphere. He was previously at Uber, where he led the development and SRE teams that created and operated M3. Prior to that, he was a technical lead on the EC2 team at AWS and has also worked for Microsoft and Google. He and his family are based in Seattle, and he enjoys playing soccer and eating meat pies in his spare time.