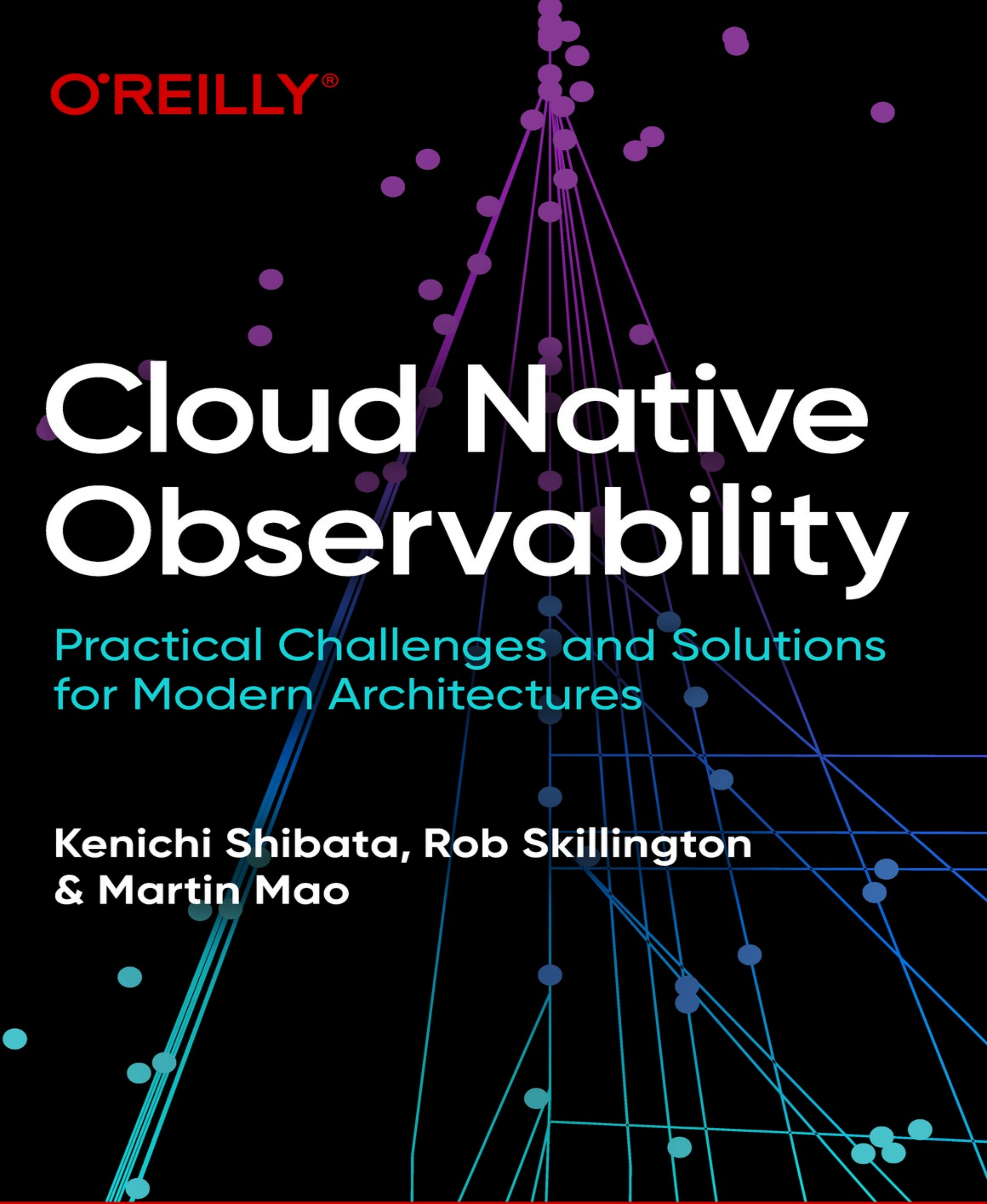
O'REILLY®



Cloud Native Observability

Practical Challenges and Solutions for Modern Architectures

Kenichi Shibata, Rob Skillington
& Martin Mao

REPORT

Cloud Native Observability

Practical Challenges and Solutions for Modern Architectures

Kenichi Shibata, Rob Skillington, and Martin Mao

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Cloud Native Observability

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*.

- Acquisitions Editor: Megan Laddusaw
- Development Editor: Gary O'Brien
- Production Editor: Gregory Hyman
- Copyeditor: nSight, Inc.
- Interior Designer: David Futato
- Cover Designer: Susan Thompson
- Illustrator: Kate Dullea
- February 2024: First Edition

Revision History for the First Edition

- 2024-02-20: First Release

The views expressed in this work are those of the authors and do not represent the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O’Reilly and Chronosphere. See our statement of editorial independence.

978-1-098-15894-1

[LSI]

Chapter 1. The Cloud Native Impact on Observability

Cloud native technologies have changed how people around the world work. They allow us to build scalable, resilient, and novel software architectures with idiomatic backend systems using an open source ecosystem and open governance.

The Cloud Native Computing Foundation (CNCF) defines "cloud native" as technologies that empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds.[1] Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

Our own in-the-trenches understanding of cloud native leads us to define cloud native technologies simply as technologies that are highly interlinked, flexible, and scalable using cloud technologies as first-class building blocks.

This then leads us to a discussion about cloud native observability. However, let's define what observability is first. According to the CNCF, *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."[2]

With this in mind, we are defining cloud native observability as the ability to measure how well you can understand the total state of your system with all of the complexities of a highly interlinked, flexible, and scalable system running in containers on a microservices architecture in the cloud.

With the above definition, it seems obvious that any existing principles, practices, or tools that apply to traditional observability would also apply to cloud native observability. However, as you soon will learn, this is not the case; many principles are similar but have unique challenges to overcome.

Challenges of Cloud Native Observability

What, then, is the difference between the observability of a traditional system and a cloud native system? Mostly it comes down to the three additional cloud native definitions we stated earlier:

- Cloud native systems are commonly interlinked (e.g., via inter-service calls directly or over a service mesh), causing most failures to be *cascading in nature* and making it difficult to observe the exact cause versus a symptom of a failure. For example, Slack experienced a major incident caused by a cascading failure on February 22, 2022.
- Cloud native systems are highly flexible and dynamic, making transient failures an expected state of the system, and therefore there is a greater need for *gracefully handling failures*, which create noise in observability systems. From October 28 to October 31 of 2021, Roblox experienced a significant systems failure because of unexpected failures in Consul (a software used for service discovery) that were not handled gracefully.
- Cloud native systems are typically decoupled and deployed in smaller units of compute for high scalability, causing a large amount of telemetry to be added to observability systems. A study by the analyst firm Enterprise Strategy Group (ESG) concluded that larger volumes of telemetry are one of the top three concerns when using or supporting observability solutions.

These three challenges not only require better tooling and principles but also practices when dealing with highly sophisticated spiderweb-like cloud native systems.

A more day-to-day analogy is that traditional observability is akin to a simple magnet and paper clip. You know that the magnet will always attract the paper clip; each time there is no change. Same magnetic force and the same metal.

However, cloud native observability is akin to observing air pressure in a tire, which is fixed in nature until you try to observe it using an air pressure gauge. Even the very act of observing the pressure (or in our case, cloud native services) changes it!

To conclude, cloud native observability is hard *because* cloud native services are highly scalable and dynamic. It's the advantages of these services that make our job harder than ever before.

Deep Dive into Observability Data

Let's talk about what we mean by "observability data" anyway. Let's take a look at an ecommerce system.

One part of the system is a cart service. Let's say you ordered a box of tissues from that ecommerce website. What kind of observability data does it produce? The cart service might produce an *event* that would take stock from a database inventory and subtract from it one box of tissues. This event firing is both observability data and a service transaction. Notice that we have not yet explicitly talked about metrics, logs, or traces. Events are the source of observability data, such as logs, traces, and metrics, which are derived from such events that occur within a system.

Observability Data Is Growing in Scale

We described that observability data in a cloud native landscape increases exponentially. But digging deeper, what is the main cause of this increase?

There are two main factors for this ongoing explosion of observability data. They are volume and cardinality. *Volume* is simple, the amount of data you collect. *Cardinality* is much more complicated; there are many different ways you can slice and dice your data. We will discuss cardinality in more detail shortly.

Since there are more interconnected services in a cloud native environment, a greater volume of data is captured per business transaction than non–cloud native environments. Also, containers are smaller in unit size than their predecessors, and each container emits unique telemetry separately. Finally, since there are more variations of these cloud native services, that data needs to be sliced using dimensions in a myriad of different ways.

Because of the increased volume and cardinalities in real-world use cases, we posit that cloud native environments emit 10 to 100 times more data than traditional virtual machine–based environments.

As a consequence of volume and cardinalities, to achieve good observability in a cloud native system, you will have to deal with large-scale data.

Understanding Cardinality and Dimensionality

Two important concepts you'll need to understand are cardinality and dimensionality. *Cardinality* is the number of possible groupings depending on the observability data's dimensions. *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 1-1 visualizes this example by looking at two dimensions: color and size.

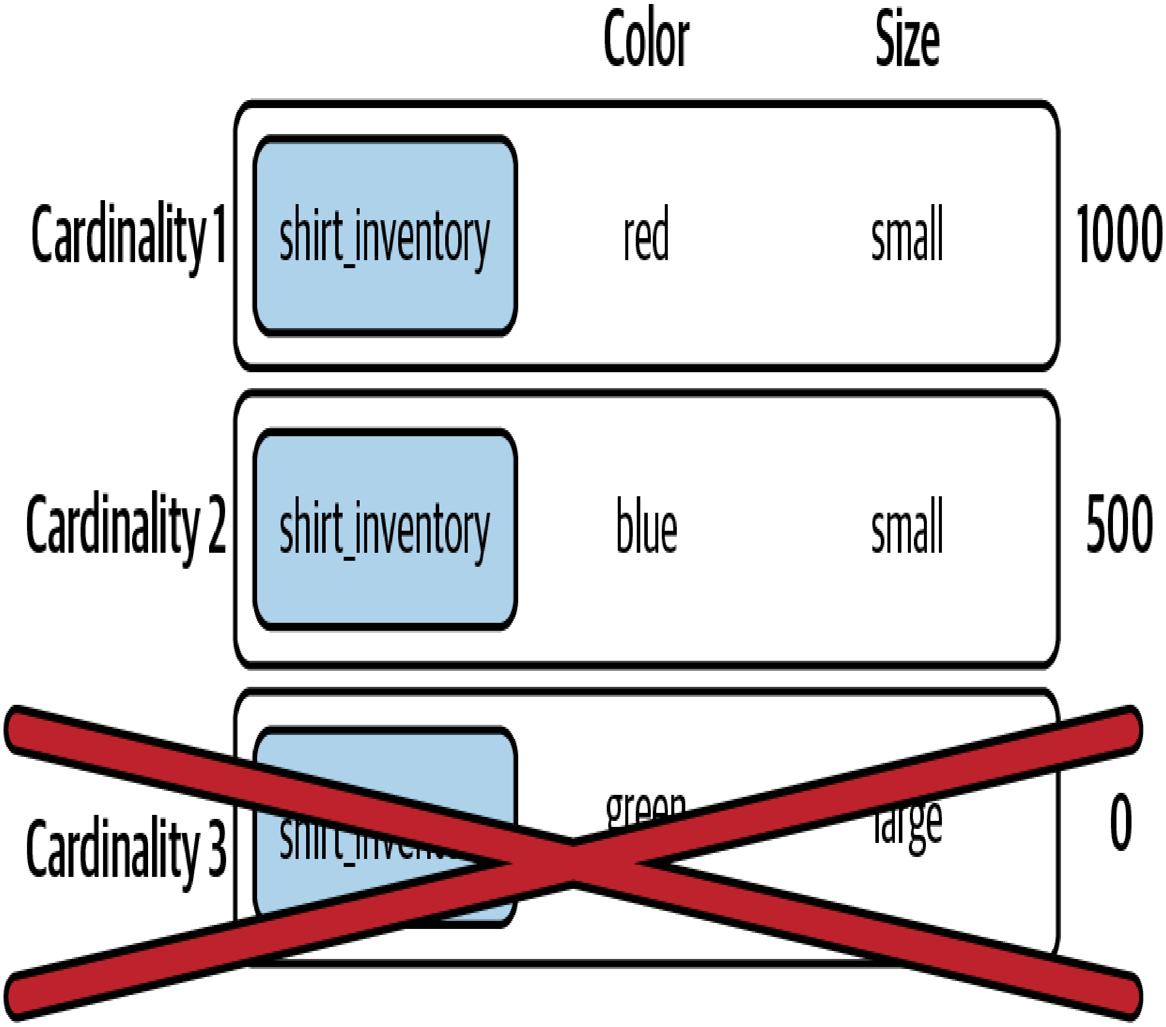


Figure 1-1. Shirt inventory with two cardinalities

There are only two cardinalities in Figure 1-1, 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.

Because events have dimensions, any data derived from events has cardinality. This includes derivatives such as logs, traces, and metrics.

Now let’s look at what happens if we increase the number of dimensions, and therefore possible cardinalities, by adding the type of shirt. Figure 1-2 shows how increasing the number of dimensions also increases the cardinality of the metrics.

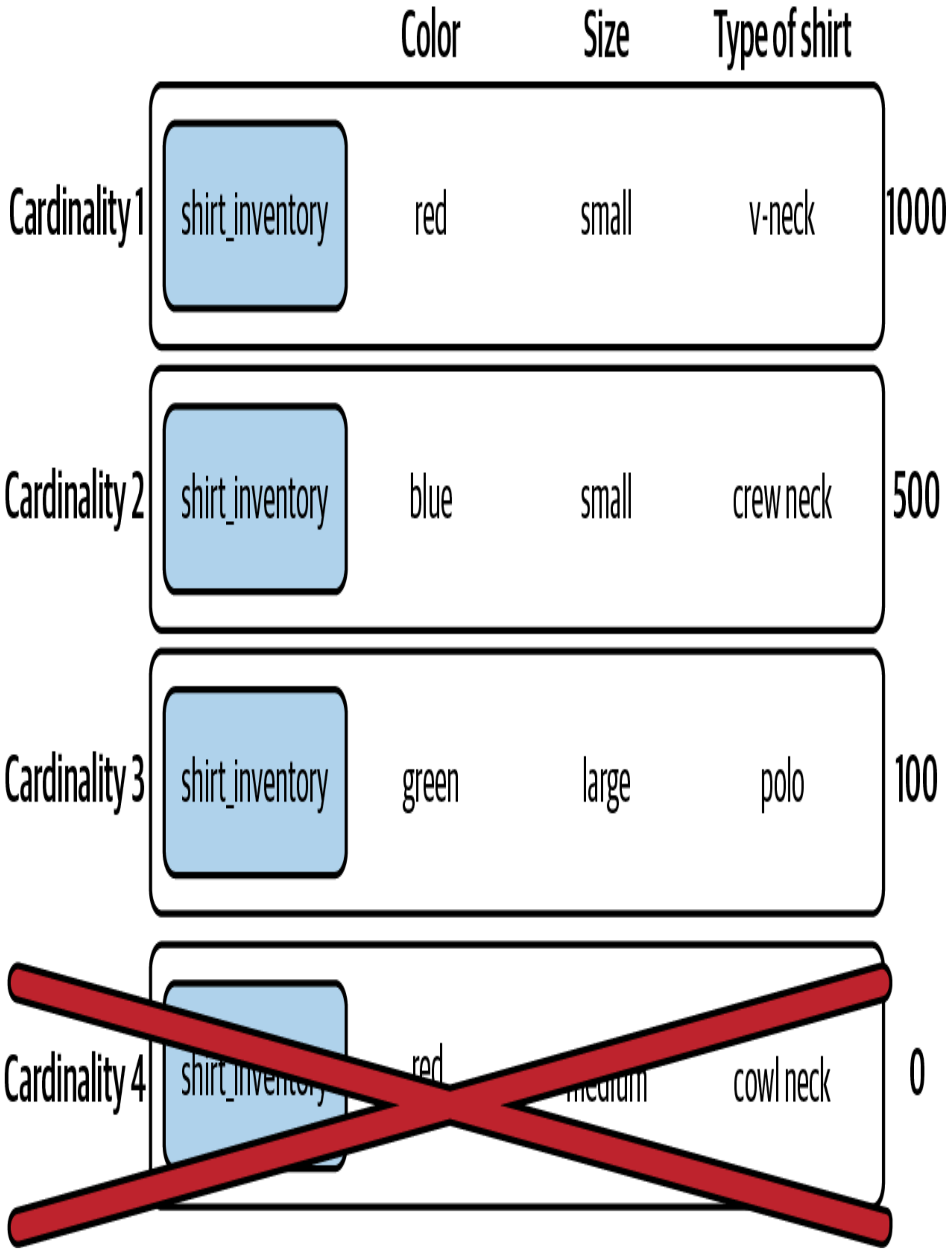


Figure 1-2. Shirt inventory with three cardinalities

What does this look like in practice? For example, assume you wanted to add a new dimension in an `HTTP_REQUEST_COUNT` metric. You want to know which specific HTTP status code created an `HTTP 5xx` error. This would allow you to better debug the error code path server side. To further debug issues introduced by certain client versions, you need to add something like a `CLIENT_GIT_REVISION` dimension in the above `HTTP` metric. Let's calculate how much it will theoretically add to the cardinality:

50 services/applications (including daemon agents/proxies/balancers/etc.)

× 20 average pods per service

× 20 average HTTP endpoints or gRPC methods per service

× 5 common status codes

× 30 histogram buckets

= **3 million unique time series**

However, precisely because it is multiplicative, removing one dimension will decrease the cardinality multiplicatively as well. Assume you do not need to determine exactly which pod is causing `HTTP 5xx` errors because you can track down the offending error code path using `CLIENT_GIT_REVISION` itself:

50 services/applications (including daemon agents/proxies/balancers/etc.)

~~× 20 average pods per service~~

× 20 average HTTP endpoints or gRPC methods per service

× 5 common status codes

× 30 histogram buckets

= **150,000 unique time series**

As Bastos and Araújo note, "Cardinality is multiplicative—each additional dimension will increase the number of produced time series by repeating the existing dimensions for each value of the new one.”[4] Choosing the right dimensions is key to keeping cardinality in check.

Cloud Native Systems Are Flexible and Ephemeral

“Containers are inherently ephemeral,” Lydia Parziale et al. 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, you must take specific actions to deal with the ephemeral behavior.”[5]

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 a few seconds 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.[6] The largest cohort—27%—are containers that churn roughly every 5 to 10 minutes. Eleven percent of containers churn in less than 10 seconds.

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 observability of data produced.

Another key change from traditional workloads is that data retention in cloud native environments is different. Retaining terabytes and petabytes of data is easier in cloud native workloads, as there is virtually unlimited storage. However, in cloud native architectures, especially in containers, workloads are inherently stateless and flexible. Retaining data is paramount to prevent loss of information when termination happens and new containers are spun up.

This is why data in cloud native architectures is constantly growing in scale and cardinality. The ephemeral nature of cloud native systems gives it the flexibility to scale up and down faster than ever before.

The Goldilocks Zone of Cloud Native Observability

Another factor in cloud native observability is that business outcomes of providing a reliable and scalable service are sometimes decreasing while observability data volumes are increasingly disproportionate to the value of the additional data collected.

Increasingly, we hear that cloud native observability systems cost more than traditional observability systems to maintain and run while counterintuitively delivering reduced business outcomes. But all is not lost! We believe you can intelligently shape your data using policies without having to micromanage your data to best utilize costly telemetry storage. This state is a good balance between cost and still keeping enough data to deliver better business outcomes. We call this theory the *Goldilocks zone* of cloud native observability. “Goldilocks zone” is used in astronomy to refer to the theoretically habitable area around a star where it is not too hot or too cold for liquid water to exist on surrounding planets.[7]

But why is this cost increasing? What are the factors that are driving this cost? How do we control this cost while increasing business outcomes? And more importantly, how do we keep our fellow observability practitioners from burning out?

We postulate a truism (or a paradox) in cloud native systems that cloud native adoption itself can feel self-defeating:

> *To help build better business outcomes in your cloud journey, your cloud native adoption needs to expand. As you adopt cloud native further, you need to build more microservices. And as you build more microservices, you accumulate complexity, which conversely causes lesser business outcomes.*

The complexity we are talking about here is the number of systems you have to support and the additional increase in failure modes that can lead to the growth of data each system produces. This could mean business data but also observability data.

Cloud Native Environments Emit Exponentially More Data Than Traditional Environments

Cloud native systems growing in scale to tackle complexity means the observability data they produce also grows. Based on a study by ESG, this growth is exponential in nature, far outpacing the growth of the business and its infrastructure (as shown in Figure 1-3). This rapid rate of growth causes problems: storing *all* of the data about everything (logs, metrics, and traces) would be prohibitively expensive in terms of cost and performance. ESG also found that 69% of the IT, DevOps, and application development professional respondents saw the growth rate of observability data as "alarming."[8]

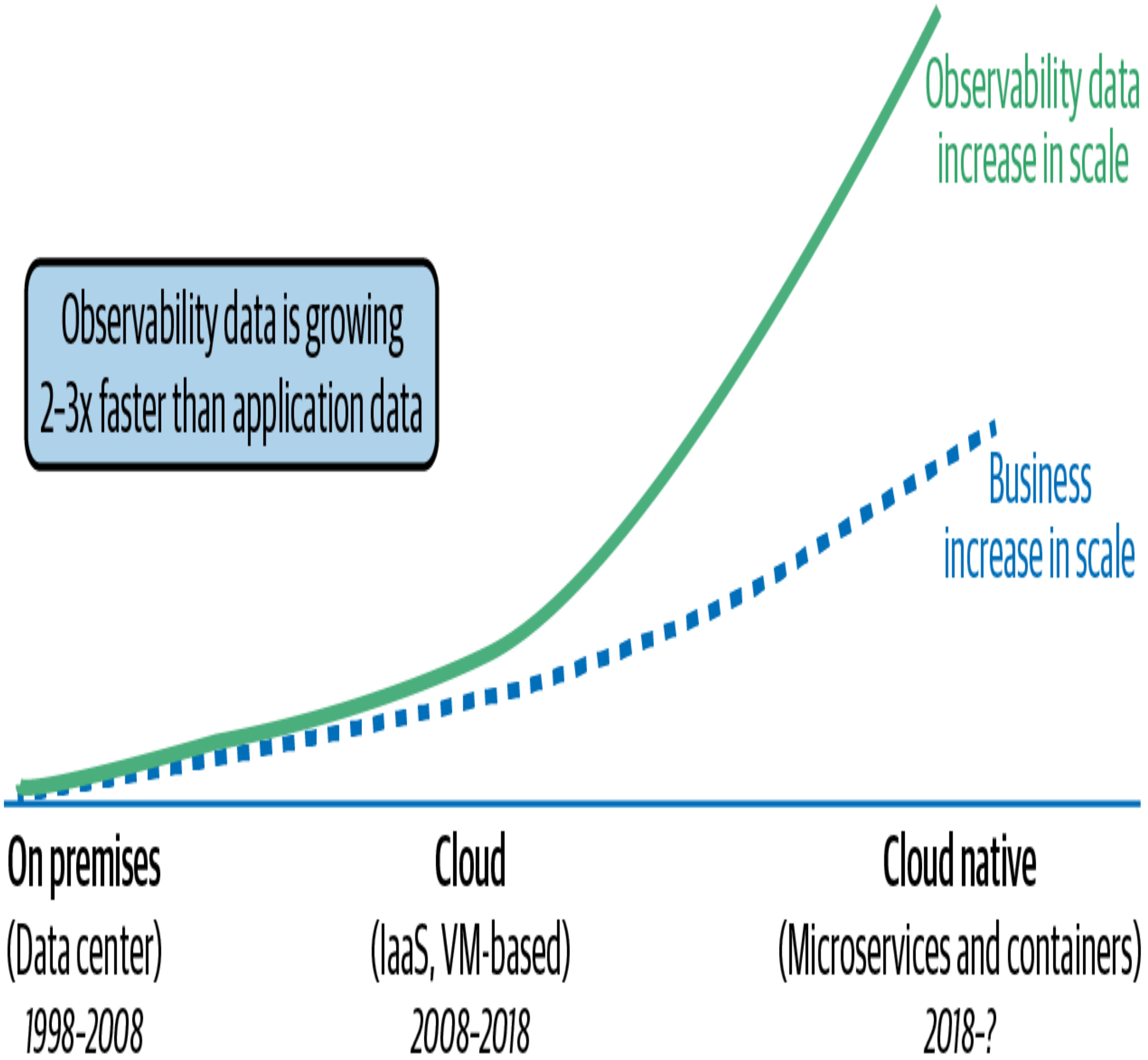


Figure 1-3. Cloud native's impact on observability data growth (adapted from an image by Chronosphere)

Why so much growth? The reasons include faster deployments, a shift to microservices architectures, the ephemerality of containers, and even the cardinality of the observability data itself to model the moving picture of this complex environment!

Delivering Reduced Business Outcomes

This moves us to the fact that a higher volume of observability data in cloud native is correlated with reduced business outcomes. Why is that?

In traditional observability, the main challenge for practitioners was increasing observability data due to higher transaction volume or system complexity to meet business needs. There was no standard way of outputting observability data from traditional systems; each vendor had their way of doing this (discussed in depth in Chapter 2).

What little observability data we had was directly related to the systems we wanted to observe. Since there was less interdependence, each data point was independently valuable and presented a slice of critical observation tied directly to obvious business outcomes.

With cloud native, we now have an abundance of observability data. Yet, when much of this data is too fine-grained and ambiguous without further context, its significance decreases and creates more noise. Instead, we need aggregate data that would be contextually useful and directly support better business outcomes.

In short, we're seeing reduced business outcomes due to too much observability data of low quality and irrelevant to the big picture. Increasingly we are seeing that practitioners find it difficult to deliver the same level of business outcomes using the same level of investments. As Michael Hausenblas explains, "With observability, it's just like that: you need to invest to gain something."[9]

Observability Practitioners Lose Focus

Practitioner teams tend to get bogged down in the weeds of instrumentation and lose sight of the fact that more telemetry data does not always equal better observability. As cloud native systems become more complex, so does the instrumentation needed to get telemetry data from these systems, shifting the focus from conducting data analysis to telemetry management. All of this creates undue stress and tends to cause burnout for practitioners.

This does not change the need for observability data in the cloud native world. It simply means it is now harder to extract positive business value and outcomes from this data. To refocus, we need to ask the right questions, such as:

- Do you instrument critical parts of your code sufficiently?
- Which data do you collect?
- What is your storage and retention strategy?
- How many ways can you slice and dice your data?
- Are there dashboards to visualize this data?

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

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?

We recommend only instrumenting the telemetry that matters to your organization, which allows you to focus on outcomes. Focusing on business outcomes helps practitioners be more connected to the overall goal of the business, reducing burnout. To achieve this focus, follow the three phases of observability (which we will discuss in more detail in "The Three Phases of Observability: An Outcome-Focused Approach") to refine your processes and tools iteratively, and be vigilant in measuring your mean time to remediate (MTTR) and mean time to detect (MTTD). Additionally, implementing internal service-level objectives (SLOs) and customer service-level agreements (SLAs) centers observability practices around crucial business outcomes, fostering a more outcome-oriented approach.

Increasing Cost of Observability Data

With both cloud native and traditional environments, as we increase the amount of data we increase the cost (Figure 1-4).

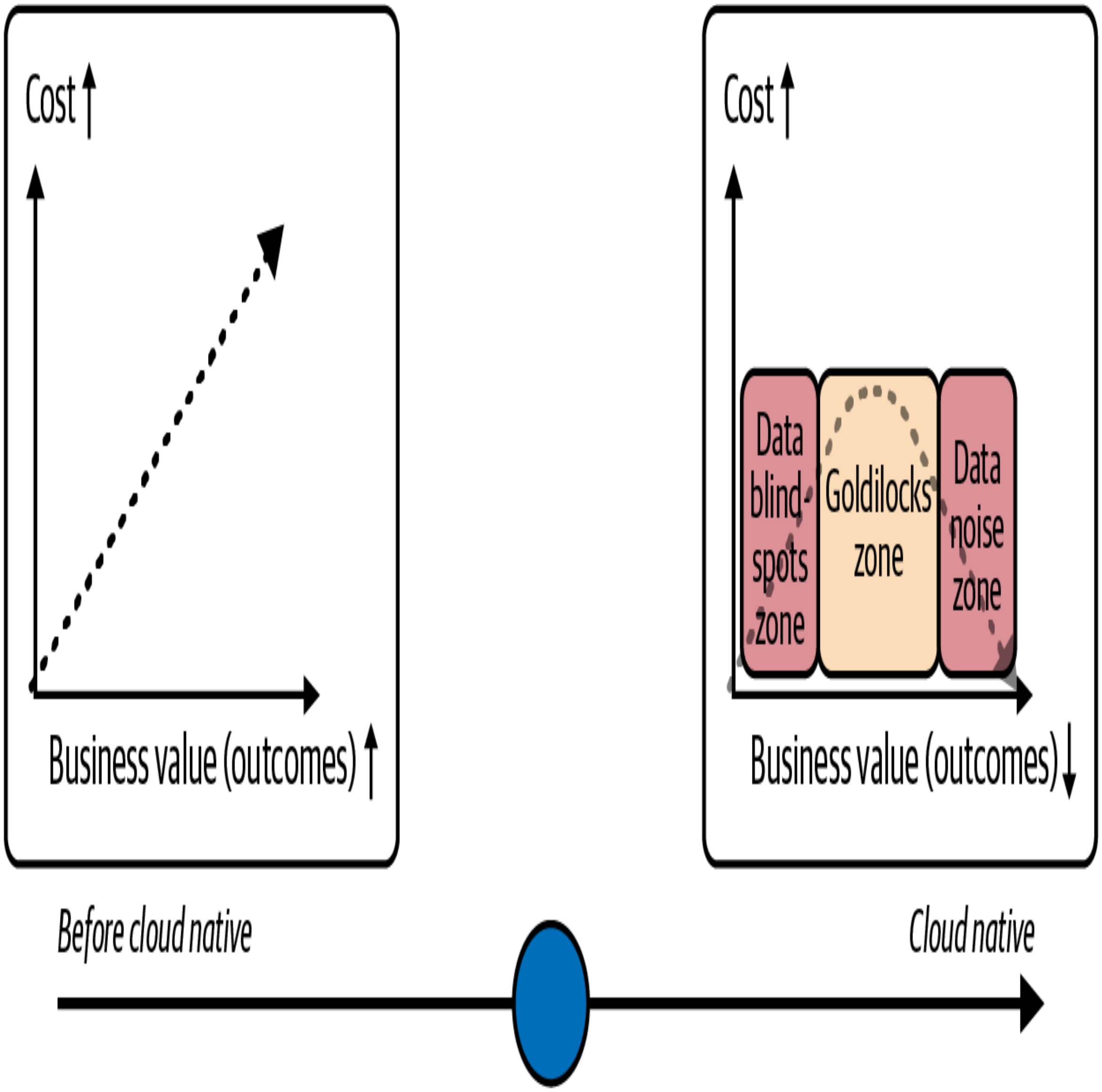


Figure 1-4. Value versus cost of observability tool before and after cloud native

The *Goldilocks zone* is where cloud native observability data is worth the cost. However, the business value decreases as you decrease or increase the observability data. Finding the Goldilocks zone means you gather the correct observability data and store the useful and effective slices of the synthesized observability data.

The Cloud Native Impact

According to a recent survey conducted by Gartner, cloud native adoption is growing.[10] By 2028, 95% of all global organizations will be running containers in production, an increase from fewer than 50% in 2023. The survey also indicates that 25% of all enterprise applications will be running in containers, from fewer than 15% in 2023.

What this shows us is that cloud native adoption has already crossed the chasm and that there is no going back.

Slower Troubleshooting

A separate survey found that engineers working on cloud native environments spend an average of 10 hours of their time per week troubleshooting,[11] 54% of participants feel stressed out, 45% don’t have time to innovate, and 29% want to quit due to burnout. Burnout bites twice because software engineers are not interchangeable and onboarding new engineers takes a while.

When people who have burned out leave, they take valuable institutional knowledge with them. The loss of institutional knowledge further deteriorates business outcomes.

Eighty-seven percent of participants agree that cloud native architectures make observability more challenging, with the same percentage agreeing that observability is essential to cloud native success.

Tools Become Unreliable

Tools become unreliable when they are unable to scale with the volume and complexity of data generated by cloud native systems or when they fail to adapt to the ephemeral nature of cloud native architectures.

Traditional observability systems were generally simpler because there were fewer ways a system could fail and the volumes of data were generally smaller. With cloud native, there are more ways for systems to fail, and there is also a growth in observability data due to distributed sets of interrelated data.

In a pre–cloud native world, tools were reliable because the architectures were mostly static and homogenous, allowing you to understand the failure modes. However, with the shift to cloud native architectures, these tools are no longer reliable. To increase the reliability of your observability system, you need to store the correct observability data.

The question is, how do you identify which observability data is worth storing?

Use Context to Troubleshoot Faster

To answer this question, you need to understand the major use cases for observability data in your organization.

For example, look at high-impact observability data like remote procedure call 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? Performing this analysis, at least on the present and shared dimensions across these metrics, can be a difficult but worthwhile exercise.

What does that exercise give your observability data? In a word, *context*. You are no longer getting and keeping all observability data but only observability data useful in the context of your cloud native architecture and applications.

The Three Phases of Observability: An Outcome-Focused Approach

A focus on context also becomes important when we keep its original and primary purpose in mind: 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 what metrics we should produce to understand if something is wrong with the system. To 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 end user?*

In most cases, a customer wants to be able to do what they came to do: for example, buy your products. You can work backward from there. You don't want your customers to be unable to buy products, so if the payment processor goes down or becomes degraded, you need to know as soon as possible to remediate the issue.

Once you find the customer outcomes you are looking for, *then* the primary signals of observability (metrics, logs, and traces…alongside other signals like events) 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, we endorse starting from the outcomes you want.

We call our approach the *three phases of observability* (Figure 1-5).

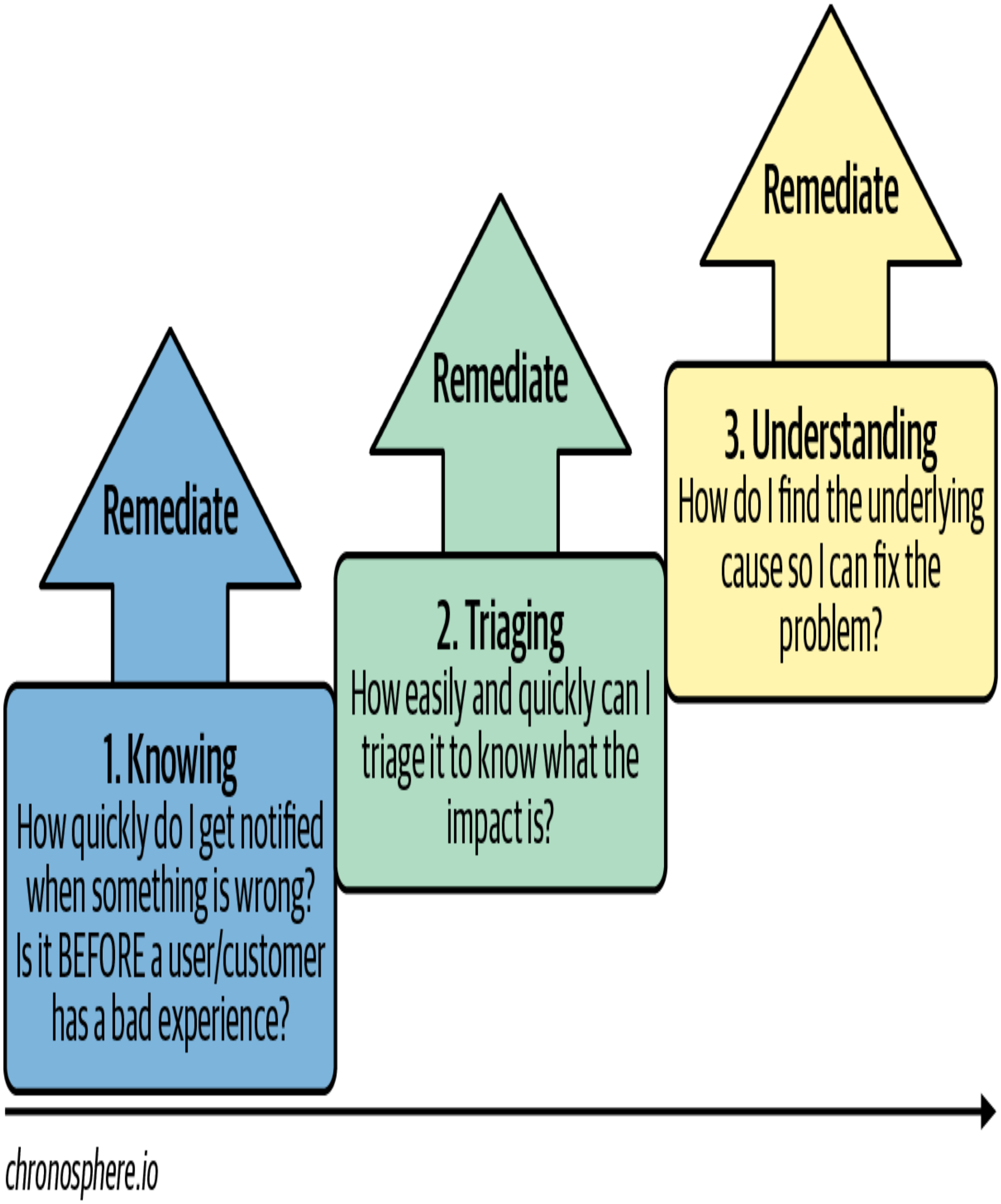

Remediate
Remediate
Remediate
1. Knowing
How quickly do I get notified when something is wrong? Is it BEFORE a user/customer has a bad experience?
2. Triaging
How easily and quickly can I triage it to know what the impact is?
3. Understanding
How do I find the underlying cause so I can fix the problem?
chronosphere.io

Figure 1-5. The three phases of observability (adapted from an image in Rachel Dines, "Explain It Like I'm 5: The Three Phases of Observability," Chronosphere, August 10, 2021, https://oreil.ly/f7dnZ)

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

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

Some systems are easier to observe than others. The key is being able to observe the system in question at the granularity that lets you make a decision quickly and decisively.

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 you 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 many unrelated users and causing CPU cycles to increase tenfold—far more than necessary.

You determine using the payments processor metrics dashboard that this increase was the root cause of the incident. The payments processor requires more CPU than you currently allocate, which bottlenecks the entire cart system. You and the team fix the component permanently by allocating more CPU and scaling the payments processor horizontally.

Remediating at Any Phase, with Any Signal

Although we posit three phases, at any phase your goal in practice is always to remediate problems quickly. If a single alert is firing off and you can take steps to remediate the issue as soon as you know about it (Phase 1), you should do so. You don't have to stop to triage or do a root cause analysis every time if these are unnecessary.

To illustrate this point, let's say a scheduled deployment immediately 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.

You can also resolve an issue using observability, even without using all of the traditional observability signals (metrics, logs, and traces). In the payment processing example in the previous section, we show one situation where just looking at the metrics dashboard could be used to determine which systems in what environments and code paths were causing the issue and provide enough information to allow for a quick and efficient fix.

Conclusion

In conclusion, the shift to cloud native systems created an issue of ballooning observability data, resulting in a loss of focus on customer outcomes, practitioner burnout, and an increase in the cost of observability systems.

Finding the *Goldilocks zone* in your observability strategy will allow you to manage the trade-off between cost and better business outcomes.

The growth of observability data in cloud native systems is caused by cardinality, the flexibility of cloud native architecture, and greater interdependence between cloud native services.

Finally, we recommend that you follow the three phases of observability to remediate issues and to ensure that you keep only useful observability data you need, using context as your guide.

[1] See CNCF Cloud Native Definition v1.0 for more information.

[2] See the CNCF's observability definition for more information.

[3] Rob Skillington, "What Is High Cardinality," Chronosphere, February 24, 2022, *https://oreil.ly/-wttH*.

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

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

[6] Eric Carter, *2018 Docker Usage Report* (Sysdig, 2018), *https://oreil.ly/ftZum*.

[7] National Aeronautics and Space Administration, "What Is the Habitable Zone or 'Goldilocks Zone'?" Exoplanet Exploration: Planets Beyond Our Solar System, *https://oreil.ly/hB0kG*.

[8] Rachel Dines, "New ESG Study Uncovers Top Observability Concerns in 2022," Chronosphere, February 22, 2022, *https://oreil.ly/gKLm8*.

[9] Michael Hausenblas, "Return on Investment Driven Observability," March 24, 2023, *https://oreil.ly/n2Aax*.

[10] Gartner, "Gartner Says Cloud Will Be the Centerpiece of New Digital Experiences," November 10, 2021, *https://oreil.ly/Ney9y*.

[11] Chronosphere, “Cloud Native Without Observability Is Like a Flightless Bird,” *https://oreil.ly/UmYrM*.

Chapter 2. Cloud Native Challenges in the Real World

Observability in a cloud native world is difficult; gathering data from a single output source and correctly inferring a view of the world about that cloud native service is impossible. We are now in a world where cloud native observability needs to be correlated and processed in myriad ways for a single assumption to be proven true or false.

In a survey titled “Filling the Observability Gap” conducted by O’Reilly about observability, respondents revealed three main challenges: lack of observability data, high costs related to tools and training, and difficulties coordinating the teams that were trying to solve system and network problems.[1]

This chapter delves into real-world scenarios highlighting the performance, cost, and reliability issues associated with observability data. We will explore case studies from companies that illustrate practical approaches and possible solutions to these challenges. Finally, we will try to come up with a reusable solution.

While the overarching challenges of cloud native observability are clear, one of the most immediate impacts is seen in system performance. Let’s explore how uncontrolled data growth can significantly strain our systems.

Impact of Uncontrolled Data Growth on System Performance

A key factor contributing to this uncontrolled data growth is automatic instrumentation. Consider the example of NGINX:

installing the NGINX ingress controller in a cluster is straightforward, and enabling observability data generation is as simple as activating a configuration setting. However, this ease of use comes with a downside. The activation of automatic instrumentation often results in the creation of numerous unnecessary metrics, leading to additional clutter in the observability system.

The combination of automatic instrumentation and horizontal scaling exacerbates the problem. We began accumulating so much data in the observability system that it created an extensive data set. Querying this extensive data set presented a substantial challenge in maintaining efficient system performance.

As we have seen, unchecked data growth can severely hinder system performance. But beyond performance, there lies a critical concern of cost management.

Controlling Cost

The ideal solution to control cost is to downsample or downright drop data before any of it is persisted. Anecdotally, we figured out that this is not enough on its own.

As we soon learned, controlling costs and making cloud native observability efficient is not a milestone you can reach but a whole journey you must undertake. Think about it this way: you can delete unused data today, but tomorrow another set of unused data will be populated and you will be in an unending game of whack-a-mole. You need a strategic plan to effectively target priority issues and maintain repeatable outcomes.

Having discussed the theory behind cost control and performance, let’s examine how these principles are applied in real-world scenarios. Our first case study provides a concrete example of tackling these challenges.

Case Study 1: Improving Performance While Gaining Huge Cost Savings

The Challenge

Our first example is one of the world's largest fintech companies, servicing close to 80% of the US ecommerce market. Mere seconds of downtime directly impact top-line revenue, affecting not just them but also their customers. For example, selling BNPL (buy now, pay later) financial products means that reliability and performance are paramount. If an end customer can't secure financing at checkout, it's a loss for the company in terms of revenue. Furthermore, the end customer might forgo purchasing the product from the partner business, leading to a compounded loss of revenue. All in all, it's not a good experience for anybody. Not for the end customers, the company, or its customers.

They needed a way to detect right away if there are any issues and if performance was impacted in subseconds. Not only that, but they also needed to implement solutions that would recover from issues in record time. The criticality of these services meant that observability performance is a top concern.

Approach

> *We had significant cost savings while we were actually able to send more data over the system. With big load increases during our Black Friday event—up to 10x—and Chronosphere had no issues handling that.... It was a win-win overall for us.*
>
> —Former senior principal engineer at fintech company

The solution was a multifaceted approach, focusing on proactive issue detection and swift resolution. This started with a sophisticated data analysis system capable of evaluating observability data granularly before it was even persisted. By analyzing data at the ingestion stage, they identified potential issues or performance bottlenecks early on, thereby preventing them from escalating into customer-impacting problems.

Refining the persisted data was another critical step. This involved optimizing data to ensure that only relevant and necessary data is retained, thereby reducing storage costs and improving data processing efficiency. Employing advanced data aggregation and downsampling techniques played a significant role in this step, leading to significant cost savings without compromising on data quality or accessibility.

Operating a highly performant observability solution required continuous improvement of use cases as well. Imagine creating a solution tailored to each use case that allows you to generate insights into your usage costs. Another aspect was a highly optimized query that enabled the creation of performant dashboards for each team, allowing them to be alerted in subseconds if something goes wrong in a specific customer journey.

Lastly, it was important to follow the three phases of observability to sort through alerts effectively, distinguishing between real issues and minor glitches or “noise.” This way, the team can focus on what truly matters and ensure a smooth experience for their customers.

The company’s approach to tackling performance and cost issues provided valuable insights into handling data growth, a pervasive challenge in cloud native observability. This leads us to the next critical aspect: the impact of this data growth on the reliability of observability systems.

Impact of Uncontrolled Data Growth on Observability Reliability

The unrestrained expansion of data within observability systems poses a threat to their reliability. As the volume of data grows, the ability of observability systems to effectively monitor and report on applications is compromised. The main issue is the overload of information. When observability systems are flooded with more data than they can efficiently handle, indexing the data becomes very difficult, and important signals get lost. Worse, it can obscure key indicators, making it difficult to identify and analyze issues.

The surge in data not only complicates the analysis process but also puts a significant strain on the infrastructure supporting the observability systems. As data volumes grow, the infrastructure is strained, struggling to store, process, and retrieve the vast quantities of data efficiently. This strain can lead to increased latency in data processing or, in more severe cases, cause system outages or slowdowns.

While we have seen how uncontrolled data growth can strain our system's reliability, another significant aspect is its impact on the developers who build and maintain these systems. Let's delve into how poor observability data can lead to a poor developer experience.

Poor Developer Experience Caused by Poor Observability Data

The challenges extend beyond uncontrolled data growth; capturing inadequate or sometimes outright not useful data also has a substantial impact. Inadequate observability data can significantly impede a developer's understanding and control over the software they build. This lack of clarity not only diminishes visibility into the application's performance but also hinders the developer's ability to take full ownership of their work. Often, when faced with unreliable or nonexistent centralized observability solutions, developers resort to creating or adopting alternative observability methods themselves.

In this context, there is a need for a centralized observability team to exist. The primary goal of that centralized observability team should be to provide service owners and developers with high-quality observability tools. These tools are essential for enabling teams to manage their services effectively and efficiently. An important aspect of this is the team's ability to empower developers to provide good developer experience and allow them autonomy within the observability system's framework. This ensures that developers have the necessary resources to quickly implement new features and effectively resolve incidents, streamlining the development process.

Understanding the struggles developers face when observability systems are unreliable and observability data is inadequate sets the stage for our next case study. We will be examining how a social network is facing similar challenges aimed at improving both the reliability of their observability and their developers' experience.

Case Study 2: Increased Observability Reliability and Improved Developer Experience

The Challenge

One social media company has grown massively, serving hundreds of millions of daily users worldwide. As they scaled, their in-house open source observability solution was getting expensive and time-consuming for engineers to realistically manage as the company saw massive user growth.

To be competitive in a crowded social media market and delight the millions of customers they serve, they must deliver best-in-class availability and performance to customers worldwide. However, the company’s previous observability setup wasn’t meeting expectations:

Availability

> The previous open source software system faced stability issues and was constantly failing. Every time the system crashed, engineers had to manually step in and bring systems back online—which was time-consuming and costly. Furthermore, the system had performance issues, which meant that dashboards and queries would load slowly or not at all, and engineers couldn’t respond quickly to customer-facing issues.

Scalability

> Their observability system was at its limits and couldn’t keep up with the amount of daily ingested data. This meant that they were only able to ingest and retain a fraction of the metrics they needed for troubleshooting purposes. As they moved to a cloud native architecture, they knew they would need a system that could provide all the critical metrics at a reasonable cost.

Usability

> They found that their self-managed Graphite instances were difficult to use, time-consuming to manage, and negatively affected developer experience. It was also costly to run in terms of infrastructure costs and people hours.

Approach

The approach for them involved creating the ability to analyze their most useful observability data. The company has millions of customers, so they needed to ruthlessly prioritize only the observability data that has the most value. However, as the observability data changes depending on the context, it was prudent to have a real-time view of data flowing into their system.

A real-time view of the data allowed them to rank and label observability data and how it contributed to the overall growth of observability data, thus improving reliability. Additionally, the ability to filter signal from noise when dealing with the data of millions of customers became paramount. Analysis on this level accelerated issue triage and significantly curtailed cost.

The analysis described allowed them to have a highly available and scalable solution with a justifiable cost. However, to further improve the developer experience, providing a highly refined set of observability data was crucial. By dropping irrelevant data, aggregating useful data, setting appropriate data retention periods, and downsampling data effectively, they ensured that developers are not overwhelmed with unnecessary information.

This refinement step aids in maintaining a balance between data volume and utility, making the observability system more efficient and easier to use. Developers can then focus on meaningful data that directly impacts their work, enhancing their productivity and improving their ability to swiftly address issues. The result was a developer-friendly environment where the focus is on actionable insights.

The strategic approach they adopted, which balanced cost efficiency and high-quality data for developers, naturally led into the realm of fostering rapid innovation. As we shift our focus we will explore how observability practitioners can enable a fast-paced innovation environment where stability and innovation coexist within well-defined guardrails.

Making Way for Fast-Paced Innovation

A critical challenge for observability practitioners is the balancing act of enabling developers to rapidly innovate while providing stable and resilient platforms for them to quickly experiment without causing disruption to the entire observability system or causing noisy neighbor issues. Developers need to have autonomy to innovate effectively; however, most innovations are disruptive in nature. The observability practitioners need to be able to set guardrails to provide stability while keeping developers as autonomous as possible.

Aside from stability, the observability practitioners should also empower organizations to adapt and evolve by providing in-depth insights. The key therefore lies in establishing robust guardrails that provide a safe environment for experimentation and innovation without compromising observability system stability.

As we delve into the intricacies of fostering innovation, visibility of usage becomes crucial. This visibility will not only allow for tracking usage but also for enabling informed decisions when dealing with cost allocation. Which team uses which observability data, and for what reasons? This would allow a centralized observability team to provide a consistent approach when building guardrails.

Navigating from the need for fast-paced innovation and the implementation of effective guardrails, we transition into another pivotal aspect of cloud native observability: adhering to regulatory requirements.

Regulatory Requirements

In highly regulated industries like finance and healthcare, observability is crucial not only for organizational efficiency but also for compliance with legal mandates. These sectors demand a high level of transparency, especially for sensitive financial transactions, making observability integral to regulatory adherence.

Service-level agreements (SLAs) are vital in these regulated environments, setting benchmarks for minimum uptime of essential services. SLAs are more than operational metrics; they represent a commitment to reliable customer service and are often scrutinized by regulatory bodies. However, as businesses grow and become more complex, maintaining these SLAs can be challenging. Increased demand can strain observability systems, leading to potential service disruptions or performance degradation, which in turn risks noncompliance with regulatory SLAs.

This challenge of scaling observability systems in line with business growth and maintaining compliance with SLAs is not just theoretical. It's a real-world issue that many companies face, particularly in the fast-evolving fintech sector.

Case Study 3: Navigating Observability Challenges in Balancing Rapid Fintech Growth and SLA Compliance

The Challenge

In the competitive and fast-evolving fintech sector, companies are compelled to accelerate their pace of innovation, embracing cutting-edge technologies like artificial intelligence/machine learning, predictive analytics, and modern application development. This drive for innovation, while crucial for staying ahead, introduces complex systems that demand vigilant monitoring to ensure seamless functionality. One leading name in the fintech world exemplifies this scenario.

For them, the core challenge is twofold: maintaining high reliability and performance for its rapidly expanding user base and simultaneously sustaining its swift pace of innovation. This must be achieved within the framework of strict regulatory compliance, adding a layer of complexity to their operational strategy.

Approach

> *When money and regulatory bodies are involved, the reliability stakes are even higher—we needed to eliminate all barriers for customers to trade on our platform.*
>
> —Senior staff engineer at fintech company

For this rapidly evolving company, the primary approach revolved around aligning with the business pace and ensuring SLA compliance. This began with a backward analysis to identify traffic and usage patterns most critically affecting SLA fulfillment.

A key aspect of this analysis involved understanding developers' needs regarding observability data to accelerate innovation. Would additional observability data about product features that customers use help developers, or would it be a case of understanding the important observability data and getting it more granularly? These kinds of questions need to be answered during the analysis.

Refining the existing data becomes crucial as they continue to innovate. With each new feature or capability added, the impact on existing features is considered; for example, how a new feature A can influence existing feature B positively or negatively. Therefore, the ability to aggregate is a critical consideration in developing effective observability solutions.

Lastly, there is a commitment to continual improvement, adjusting for efficiency and expanding coverage and visibility. Possibly even creating custom instrumentation to handle business-related metrics that you cannot get out of the box from open source tools. This ongoing optimization ensures that observability systems remain aligned with their SLA.

Conclusion

As we synthesize insights for all of these use cases, it is evident that the ability to centrally drive the agenda for observability is imperative. Which parts of the observability systems should we improve? What kinds of data are we missing? Which data can we drop, aggregate, or downsample? How much budget do we need to allocate, and for what use cases? These considerations underscore the necessity of centralized observability governance. Such governance would not only steer the observability strategy but also act as an enabler and consultant to developer teams, aiding in scalability.

In these three cases, we found the need for three common steps to tackle observability reliability, performance, cost issues, and developer experience. A notable gap in all cases is the insufficient analysis of the incoming data at a granular level, leading to substantial data growth.

A focus on aggregation and retention plays a key part in fully refining existing data. Also, there's a distinct lack of refinement in how observability data is stored, highlighting the importance of downsampling and dropping redundant data, particularly in automatic instrumentation.

The operation of observability systems should be geared toward continuous improvement, focusing on expanding visibility and integrating custom business-level observability data. In many implementations, however, observability systems are treated more as an afterthought, a bolt-on to the developer's toolkit, rather than as a strategic capability.

In response to these findings, we've developed a new observability model, the Observability Data Optimization Cycle (O11y DOC). This model is structured around three cyclical steps: Analyze, Refine, and Operate. Each step is critical and should be continuously revisited for each optimization effort. O11y DOC is not just a framework; it's a strategic approach that integrates observability into the heart of organizational operations. In the next chapter, we will delve into the intricacies of O11y DOC.

[1] Andy Oram, *Filling the Observability Gap* (O'Reilly, 2021), *https://oreil.ly/HWxUl*.

Chapter 3. Strategies for Controlling Observability Data Growth and Complexity

In this chapter, we tackle the escalating challenges of data growth and complexity in cloud native observability. We will introduce a new framework to streamline and manage observability data, ensuring systems remain efficient and manageable in the dynamic cloud native landscape.

Transitioning to our new framework, we bridge the gap between the overwhelming data influx and the need for meaningful insights, aiming to achieve the ideal balance in cloud native observability.

Emerging Solution Using a Repeatable Framework

In Chapter 1, we delved into the complexities of cloud native observability, highlighting a paradoxical increase in costs alongside diminishing business outcomes. We introduced the concept of the Goldilocks zone in cloud native observability, an optimal state where costs are controlled while maximizing business value. This zone represents the sweet spot between excessive data that overwhelms systems and insufficient data that hinders insightful decision making.

Chapter 2 further explored the practical challenges in achieving this balance, emphasizing the need for a strategic approach to manage the deluge of observability data without compromising on system performance and reliability.

We have developed a repeatable, standardized, and vendor-agnostic framework based on these insights. This framework is inspired by the principles of the FinOps framework, renowned for its effectiveness in the financial management of cloud services. Our framework is designed to systematically address the key challenges of cloud native observability discussed in Chapter 1.

Using FinOps as an Inspiration

FinOps is a cultural practice. It's a way for teams to manage their cloud costs, where everyone takes ownership of their cloud usage, supported by a central best practices group. Cross-functional teams in engineering, finance, product, etc., work together to enable faster product delivery, while at the same time gaining more financial control and predictability.[1]

In this new world of cloud native, FinOps was developed to rein in costs. However, we do not have such a model for the observability space. The idea is simple: we want the cost of observability data not to exceed the value it is providing to the organization. Most practitioners think the observability data they are gathering will be useful later on. However, more often than not this is not the case, and the observability data is forgotten.

Moreover, it's hard to identify which teams or observability data is causing the spikes in cost and decreased signal-to-noise ratio. Our strategic approach will help you avoid the endless game of whack-a-mole to decrease cost and improve performance.

It is analogous to cleaning out your garage after you have already accumulated all the tools (observability data). The difficulty comes with teams not thinking about the allocation in observability capacity, mostly because they are not acutely aware of the cost associated with storing and querying such observability data.

The problem starts at scale. When people are just starting with cloud native observability, the data is not that complex, and they have less of it. However, most mature organizations do not understand what observability data they have, and they lack control over how much data they have or where it is being used.

As shown in Figure 3-1, using the FinOps framework as an inspiration, we devised a new approach that we call the *Observability Data Optimization Cycle* (O11y DOC).

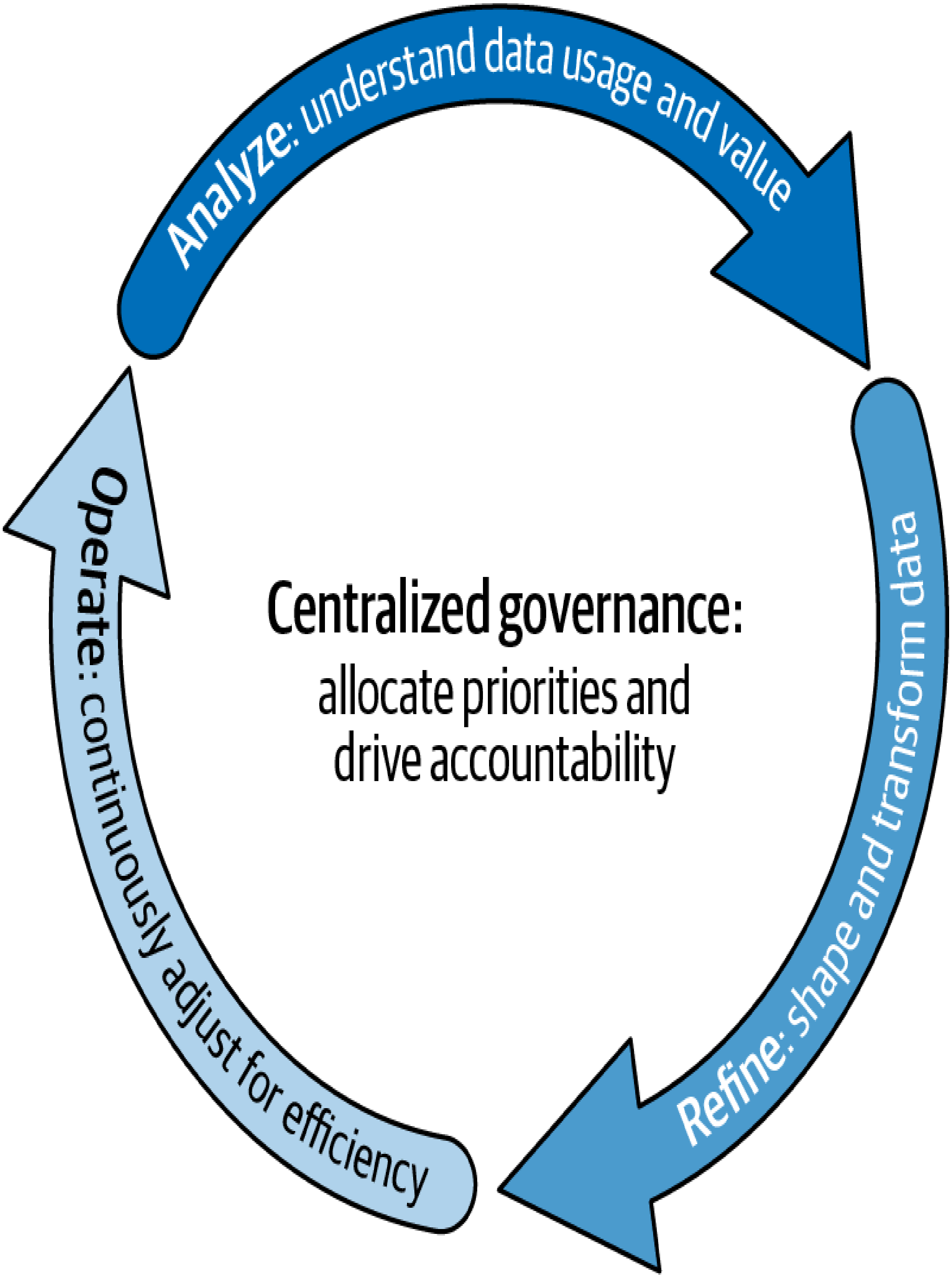

Analyze: understand data usage and value
Refine: shape and transform data
Operate: continuously adjust for efficiency
Centralized governance:
allocate priorities and
drive accountability

Figure 3-1. The Observability Data Optimization Cycle

Observability Data Optimization Cycle

A key consideration before beginning your O11y DOC is buy-in. You need a *centralized governance* structure that drives accountability within individual development teams and allocates priorities across workloads.

Observability data costs are becoming a significant budget item and increasingly unpredictable. Everyone needs to understand their usage and take ownership of their allocation of observability capacity. We propose that a centralized governance structure works best to address these issues.

Step 0: Centralized Governance

A centralized governance removes confusion on ownership and aligns incentives. As you go through the process of O11y DOC, more and more decisions need to be made. A clear centralized governance structure allows you to make informed decisions balancing trade-offs such as cost, telemetry capacity, and use cases.

We encourage the team consuming these services to communicate what kind of capacity it needs to improve its outcomes. After all, the team building the services, henceforth known as the *independent team*, knows how best to observe the services it is building.

Autonomy and Allocations to Increase Responsibility and Improve Responsiveness

Historically, developers have not had insight into the costs of observability systems. However, until the data explosion mentioned in Chapter 1, this lack of knowledge was not a dire problem.

Independent teams can deploy asynchronously and create high volumes of observability data. The two big implications are cost and performance for the observability system as a whole. Since the compute, storage, and cost of the system is a common resource, if one team uses too much of it, there is a risk that other teams then get crowded out and can no longer use the observability system.

Making teams aware of the impact of their decisions allows them to be empowered to make localized optimizations. The teams will get allocations that they can manage and use.

The carrot is that the better the teams use their allocation, the more allocation we can give them. The stick is the actual allocation: if a team perpetually breaks the quota, then the team might be using more than they should.

In short, there should be autonomy for the independent team to make decisions while informing the centralized governance of their requirements, thus empowering each team to carry out their responsibilities and get the support they need.

Usable Capacity by Allocation to Optimize Use Cases

Fundamentally, the question is how much can you have and how much can you do with it. As we previously defined via analogy, use cases are the activities the independent team uses for its allocation.

For example, let’s say you have a team—we’ll call them team 1—that supports cart platform services (Figure 3-2). Team 1 currently supports three use cases: order db monitoring, cart frontend monitoring, and inventory backend monitoring. However, to fully support the end customers, team 1 requires two additional use cases: recommendation backend monitoring and shopping analytics event monitoring.

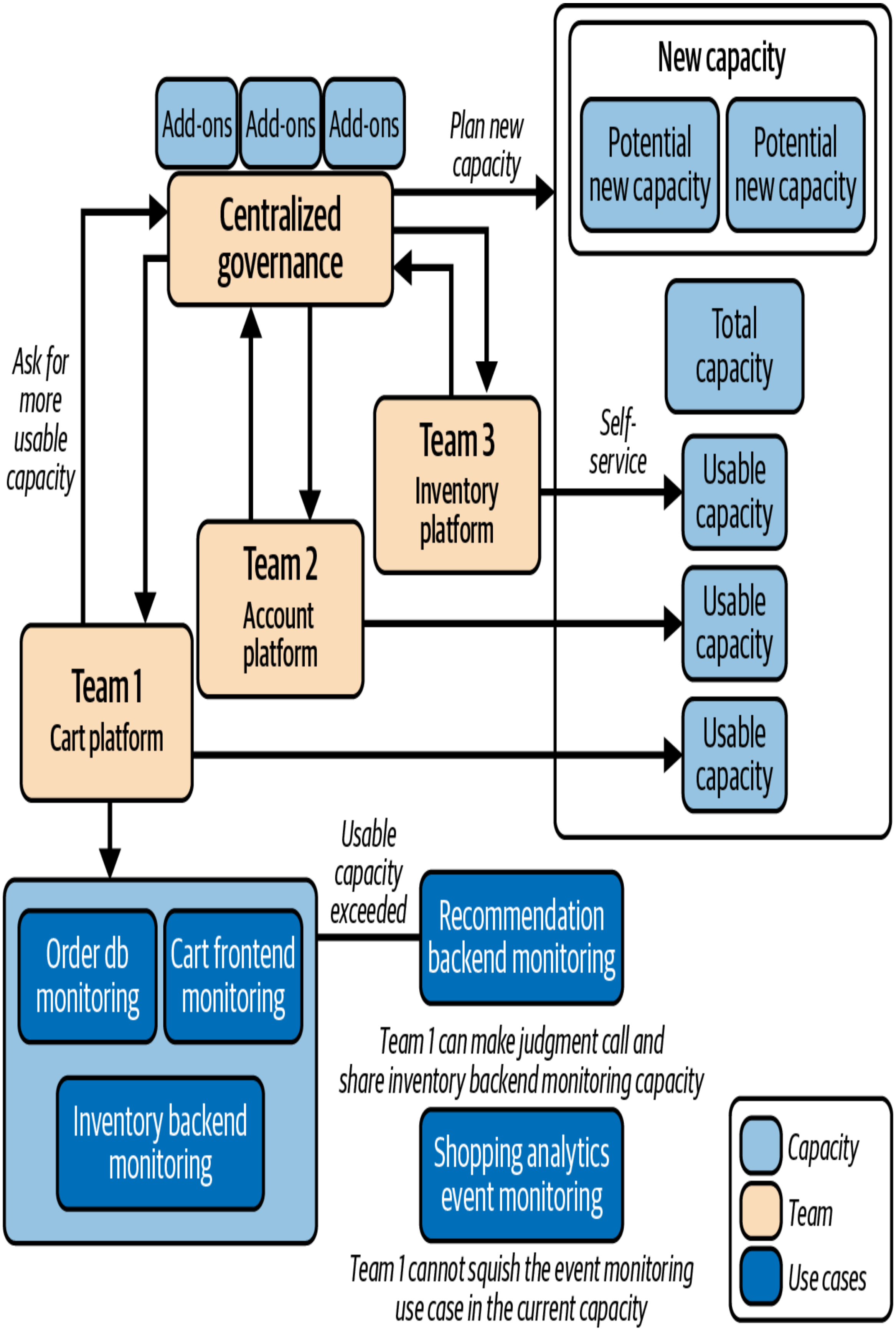

New capacity
Potential new capacity
Potential new capacity
Add-ons
Add-ons
Add-ons
Plan new capacity
Centralized governance
Total capacity
Ask for more usable capacity
Team 3
Inventory platform
Self-service
Usable capacity
Team 2
Account platform
Usable capacity
Team 1
Cart platform
Usable capacity
Usable capacity exceeded
Order db monitoring
Cart frontend monitoring
Inventory backend monitoring
Recommendation backend monitoring
Team 1 can make judgment call and share inventory backend monitoring capacity
Shopping analytics event monitoring
Team 1 cannot squish the event monitoring use case in the current capacity
Capacity
Team
Use cases

Figure 3-2. Example of a centralized governance structure

You want to empower the teams to self-service because they know best how to optimize usable capacity for their individual use cases. To further use the above example, team 1 can squish the recommendation backend monitoring and share the capacity with the inventory backend monitoring. Meanwhile, team 1 can raise a request to create the shopping event analytics monitoring to get an additional capacity.

In effect, you created an optimization by empowering the independent team to make localized decisions. An equation about cost would look like this:

$$\text{total cost} \ = \ \text{total capacity}$$

However, because team 1 has optimized for the cost by grouping use cases, the equation would be as follows:

$$\text{optimized cost} = \text{total capacity} - \left(\frac{\text{total capacity}}{\text{grouped use cases}}\right)$$

Centralizing governance while providing use case autonomy will allow for an optimized cost by empowering the individual independent team to make decisions.

Using Observability Team as Consultants Instead of as Bottlenecks

While the teams are empowered, they still have the support of the observability team. The observability team acts as the consultants or centers of excellence that can guide best practices and make recommendations to increase allocation based on data. Whereas before the observability team was seen as a bottleneck before going live, giving teams autonomy will change the perception from a bottleneck to an enabler.

Autonomy changes the observability problem from a centralized problem to a decentralized problem distributed across the teams. Teams can optimize locally for their use cases, rather than continually push the observability team to make changes. This culture of autonomy is a huge shift in the approach that most teams work with, but one that the O11y DOC concept needs to succeed.

In conclusion, step 0 is to find the correct balance between three axes—cost, use cases, and capacity—by a centralized governance structure. If you buy more capacity, you increase the cost, which you can allocate to use cases. On the other hand, if you buy less capacity, you minimize cost. Finally, you need your teams to be more empowered to allocate use cases economically by grouping useful data.

Framework Components

The O11y DOC is a framework to deliver the best possible observability outcomes at scale while controlling costs. The following are the defined stages of the O11y DOC:

1. Analyze

> Identify key cost drivers, data sources, and areas of inefficiency in cloud native observability to optimize data handling, aligning cost to value and understanding the utility of the data.

2. Refine

> Give teams tools like shaping tools and transforming tools to reduce the amount of data; this increases the signal-to-noise ratio and decreases cost.

3. Operate

> Develop real-time mechanisms to detect anomalies, inefficiencies, and data quality issues in observability data

streams.

Step 1: Analyze

The first step of the O11Y DOC cycle is Analyze. Two key areas you need insights on are the flow of observability data traffic into your system and usage.

Understanding your observability data traffic volume and usage will allow you to make intelligent, data-driven decisions about how to Refine (step 2) your data.

Traffic Analysis

For effective traffic analysis, you need to be able to analyze your observability data and all its dimensions in real time. Having a real-time view of the data flowing into your system helps you understand how often applications emit data, troubleshoot sudden spikes in ingest rates, and ensure all the data you want to collect is being collected.

Using metrics as an example, you should be able to view all the labels (or tags) along with:

- The number of unique values for each label
- The percentage of metrics you're viewing that have the matching label
- The metrics that contain the label
- The volume of data that these metrics contribute to storage

This information allows you to rank and group labels and metrics to understand how they contribute to cost and where they are coming from. For example, ranking labels based on the number of unique values from highest to lowest allows you to quickly identify your high-cardinality labels and their associated metrics.

Usage Analysis

Usage analysis is about understanding your observability data's cost and utility. It should give you insights into how the data is being used, including which dashboards and alerts it shows up in, which specific users are querying that data, when, and how often.

Combining Traffic and Usage Analysis to Make Decisions

When you combine traffic analysis with usage analysis, you can now start to understand the value your observability data delivers. There are few hard and fast rules when it comes to determining value. Outside of identifying unused data, which provides zero value, it's a function of your budget and the insights the data delivers, and every organization will be different.

You will need to think of this on a spectrum, as depicted in Figure 3-3. You may be able to assign values depending on what is important to your organization and the tools you are using. Still, as we said, it will vary depending on the organization.

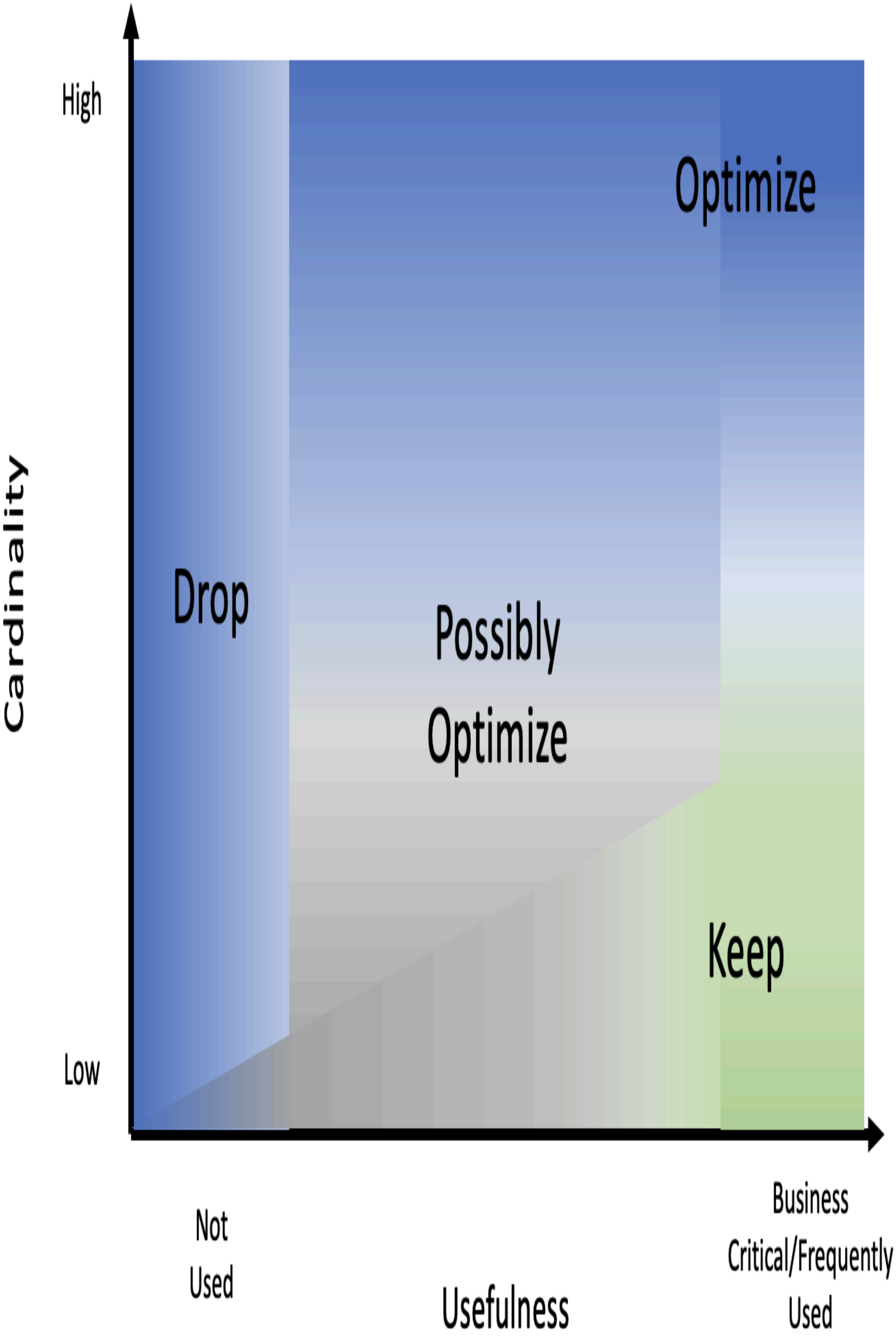

High
Optimize
Cardinality
Drop
Possibly Optimize
Keep
Low
Not Used
Usefulness
Business Critical/Frequently Used

Figure 3-3. Usefulness versus cardinality spectrum analysis (source: courtesy of Chronosphere)

At the extreme end, it's pretty easy to decide what to do. You will want to drop data that has not been used. Business critical data or frequently used data you will want to optimize using aggregations. But there will always be a middle ground where you will want to spend most of your time analyzing the cost versus value that your observability data delivers.

Output of Analyze Step

Once you have identified the data you want to drop and the data you want to optimize, it's time to look at the tools available to refine the data to align the cost-to-value ratio.

In Chapter 2, we talked about how one organization analyzed their observability system to proactively detect issues that align with the O11y DOC's principles.

Step 2: Refine

Refining observability is all about shaping and transforming the observability data. While Analyze will allow you to understand your data's quantified cost and value, the Refine step is all about aligning those costs to value.

There are multiple ways to refine the observability data, the easiest being simply dropping the data you do not need; other options include aggregating data, setting a retention period, and downsampling data.

Dropping

Dropping can mean dropping entire metrics, but it can also mean keeping only dimensions of your observability data that are useful to you and aggregating the rest in place. More often than not, even if a metric itself is very useful it will still contain dimensions that are not useful.

Let's assume that we have a metric that has dimensions `regulatory_body_name`, `status_code`, and `pod_name`, such that the metric looks like:

```
api_request_to_regulatory_body{status_code=200, \
  pod_name=pod-xxx-1, regulatory_body_name="sec"} 100
api_request_to_regulatory_body{status_code=200, \
  pod_name=pod-xxx-2, regulatory_body_name="sec"} 100
api_request_to_regulatory_body{status_code=200, \
  pod_name=pod-xxx-3, regulatory_body_name="sec"} 100
```

If you create a policy that keeps `status_code` and `regulatory_body_name` while dropping all `pod_name`s in metrics such as this, you will divide the cardinality by three. The metric will look like this:

```
api_request_to_regulatory_body{status_code=200, \
  regulatory_body_name="sec"} 300
```

The result will be a single cardinality metric instead of three cardinalities, thereby improving the overall shape of the metric.

Retention

Retention refers to the duration of storage of your observability data. Simply put, it answers the question: how long are you keeping your 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?

Base your retention periods for different kinds of data on the outcomes you can gain by retaining it. If you reduce the retention period for data you do not need, the overall volume will grow much more reasonably.

Resolution

Resolution in observability refers to the frequency at which data points are collected and recorded within a system. High-resolution observability implies that data points are gathered more frequently, providing a detailed, granular view of the system’s behavior over time. This is analogous to having a high pixel density in a photograph, where more data points translate to a clearer, more detailed image. High resolution allows for a deeper analysis of system trends and anomalies but comes with increased storage and processing requirements.

As systems grow and the volume of data skyrockets, storing and processing high-resolution data can become both cost-prohibitive and technically challenging. To navigate this, one can use techniques like downsampling or aggregation.

Downsampling

Downsampling is a method employed to manage the challenges posed by high-resolution data, particularly in large-scale systems. It involves selectively reducing the frequency at which data points are recorded, thus effectively decreasing the resolution. This process involves choosing representative data points or averaging out the data over longer intervals. For instance, if data resolution is every 15 seconds, downsampling to a 30-second interval would reduce the data volume by half. This technique helps in managing storage and processing loads but must be balanced carefully to keep the essential context and accuracy of the observability data. Downsampling is particularly useful when the high frequency of data collection does not significantly contribute to a better understanding of the system’s behavior.

In essence, while resolution is about the initial frequency of data collection, providing a detailed view of the system, downsampling is a subsequent step to optimize this data for efficient storage and processing, with a focus on retaining critical information while reducing the overall data load.

Aggregation

Aggregation refers to the process of summarizing and transforming observability data into useful observability insights. This practice optimizes storage, accelerates query performance, and provides clarity to the information presented to teams.

Retention and resolution mostly allow you to optimize cost; however, aggregation does the most when it comes to increasing the signal-to-noise ratio of your observability system. Instead of querying two or more observability cardinalities, you only need to query one aggregated observability data that would immediately give you insights.

Aggregation reduces observability set sizes by stripping high-cardinality dimensions and merging them to make more useful observability data. This is especially true for dimensions that are not valuable.

For example, suppose you are monitoring an HTTP web service. The web service generates observability data with a variety of attributes: timestamp, user agent, Internet Protocol (IP) address, request path, response code, response time, and more.

If you only care about generating insights about the divide of users between mobile and desktop, many of these attributes, such as IP address, request path, and response time, might not be immediately relevant. Therefore, you can perform aggregation based on the user agent.

Output of Refine Step

Using a combination of dropping, retention, resolution, and aggregation to curate the observability data allows for the creation of shaping rules. Implementing these shaping rules is expected to reduce costs and enhance the performance of the observability system. These rules can be one-off rules or integrated to a wider strategy to shape data. However, as the data set grows, the performance of the observability system can decline. It’s crucial not to lose the context of the data being captured. A well-defined shaping rule strategy should automatically drop unnecessary metrics or dimensions, set retention periods, and aggregate and downsample observability data.

The art lies in retaining the context. While it’s tempting to strip away as much data as possible for the sake of efficiency, it’s crucial that the resultant data set still tells a coherent story of the system’s behavior.

Shaping rules, therefore, should be dynamic and adaptive. They should set automated retention timelines, define resolutions based on the criticality of data, aggregate where meaningful, and downsample without losing the bigger picture. This holistic approach ensures that while the raw volume of data might be reduced, its informative value remains potent.

In essence, shaping rules transform raw observability data into a well-structured narrative, shedding the redundant while highlighting the essential, decreasing cost, and increasing context. As Martin Mao says: “It’s not about having less data; it’s about having more meaningful data.”

In Chapter 2 we talked about how a social network effectively used the Refine principles to demonstrate effective use of aggregation and downsampling.

Step 3: Operate

Operating observability revolves around regular optimization—continuously validating and understanding the insights generated from your established shaping rules. Over time, circumstances and requirements change. It then becomes crucial to ensure that the shaping rules remain relevant and continue delivering the anticipated value.

During this step, look for opportunities to innovate and experiment. A fundamental question to continually address is: can the existing shaping rules be further optimized? Refining your observability system and freeing up capacity should allow you to do more experiments and innovation.

In the Operate step, you also need to check whether a new shaping rule is needed. Find out if the new shaping rule will inadvertently affect observability system performance. Forecasting and testing these outcomes can avert unintended consequences, ensuring that your observability remains robust and reliable.

Expanding Visibility and Coverage

After you’ve adjusted your data to be more efficient, you can look at more areas. At first, you might cut down on data, but later you can use the saved space to see more. If you wanted detailed data before but it was too expensive, now you might be able to afford it. Remember, it’s all about balance. If you manage your data well, you can do more with it.

Freeing Up More of the Observability Team’s

Time to Tackle Strategic Projects

The observability team will be more of a consultant rather than having an adversarial role with other teams. Freeing them up and helping them allows them to put on guardrails, making the observability team a strategic team rather than a tactical whack-a-mole team.

Conclusion

In a cloud native world, mastering observability data is crucial for maintaining a competitive edge. Optimal observability outcomes go beyond quick issue resolution; they involve leveraging observability data for business innovation and enhancing customer experiences. This chapter’s Observability Data Optimization Cycle (O11y DOC) advocates for a systematic approach, encompassing Analyze, Refine, and Operate, to efficiently manage costs and enhance system performance. It emphasizes the need for cultural change, centralized governance, and team autonomy in managing observability. Finally, the O11y DOC framework outlines what is necessary to control costs while improving the performance of your observability systems.

[1] “What Is FinOps?” The FinOps Foundation, *https://www.finops.org*.

Chapter 4. Open Source Telemetry Standards: Prometheus, OpenTelemetry, and Beyond

In the previous chapters we discussed how observability data has been growing in scale while delivering diminishing business outcomes. We delved into real-life use cases on how well-known companies have solved observability data issues. Finally, we introduced a new framework for reliably solving the same issue, distilling the core principles that we have gathered from our experience solving those real-world use cases.

In this chapter, we will explore implementations using open source software and how open source instrumentation has increasingly become the de facto standard for monitoring and observability in cloud native environments. We'll trace the evolution of this trend, highlighting the pivotal roles of Prometheus and OpenTelemetry (OTel) in shaping the landscape. These tools have simplified the collection and analysis of vast amounts of telemetry data and established new benchmarks for flexibility, scalability, and community-driven development in observability.

Instrumentation Before Prometheus and OTel

Before the industry standardized on Prometheus and OTel, many companies were forced to use proprietary collection solutions, such as AppDynamics, Dynatrace, or New Relic. These vendors control the instrumentation and aggregation of telemetry 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 telemetry to their system, as shown in Figure 4-1. This is called *agent-based* application instrumentation*.* Applications typically need to install a software library or software development kit (SDK) to run these agents and send the data back to the aggregation server.

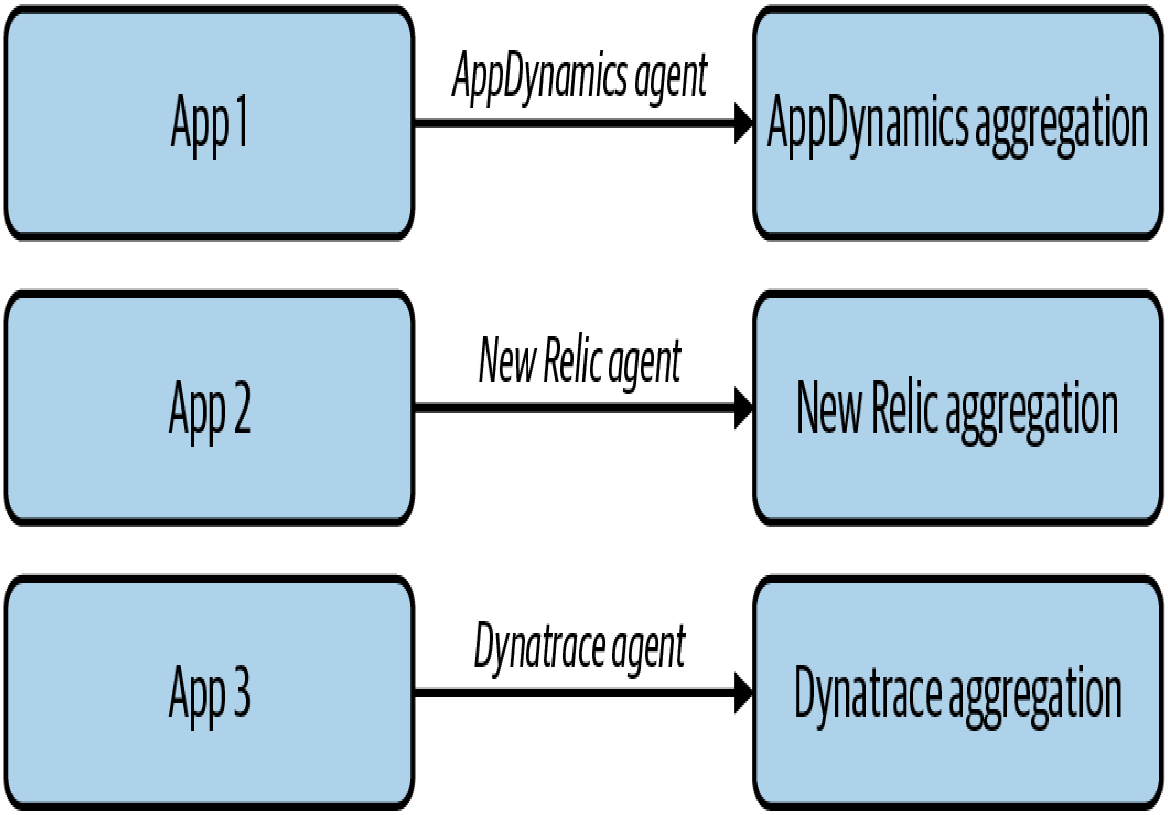


Figure 4-1. Applications instrumented using agents

In addition to proprietary agents, the data formats used by the agents are also proprietary. These proprietary data formats mean that, for all intents and purposes, your data is locked into the vendor. For example, you cannot easily migrate from one vendor to another without losing all dashboards and alerts that were built by the original vendor, making migrations labor-intensive and wasteful.

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.

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

Data Collection Is Controlled by Users

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

By August 2018, Prometheus graduated as a CNCF official project.[1] An open source ecosystem was built around Prometheus largely because of Kubernetes and its increasing ubiquity in the cloud native space.

Because of Prometheus, most organizations running in cloud native architectures today no longer have to deal with a myriad of tools and agents to instrument their applications. Effectively, this moved the data collection from being controlled by vendors to being controlled by cloud native observability practitioners.

Prometheus

Prometheus is inspired by Google's Borgmon monitoring system (Borg). 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.

Interoperability Between Different Observability Tools

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 4-2 shows agentless metrics in Prometheus.

The shift to pull-based metrics collection has allowed SRE teams to better control the metrics they collect. Further, pull-based collection allowed *interoperability* between different observability tools.

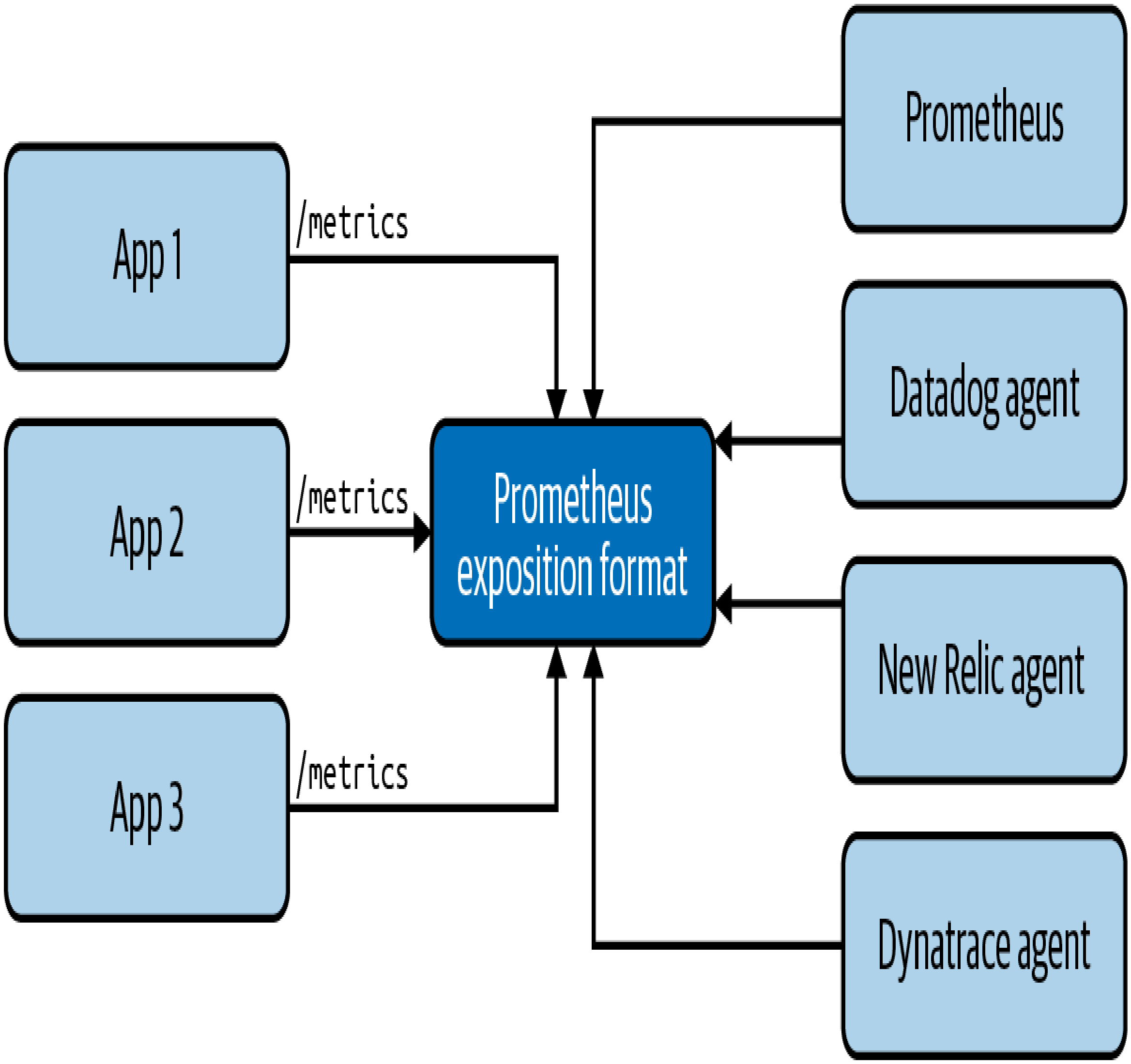


Figure 4-2. Prometheus's exposition format, supported by vendors

Standardization to Prometheus

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 exposition format, Prometheus exposition, then wrote clients that use it to expose metrics simply.

Since Prometheus shifted the responsibility to clients outputting a standard data format, it created a system where whoever supports that format can use the data. This created a cottage industry of every software that outputs data supporting Prometheus's exposition format, resulting in massive adoption and standardization around Prometheus.

In essence, you write once, and you can output anywhere that supports the Prometheus exposition format!

Prometheus Reliability

With Prometheus, metrics instrumentation is part of the application rather than a separate process, as shown in Figure 4-3. This contributes to greater reliability.

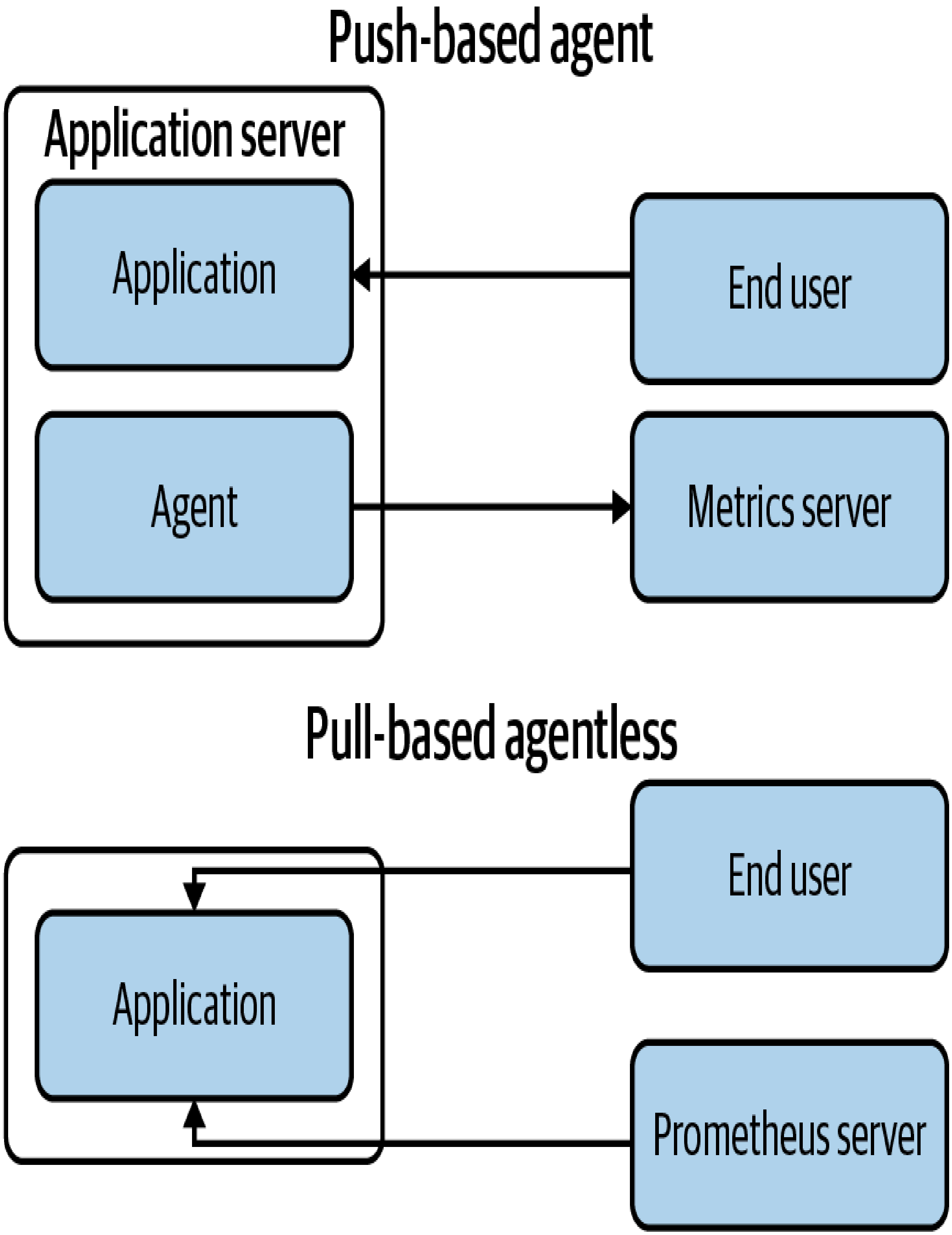


Figure 4-3. Push-based agent instrumentation versus pull-based agentless instrumentation

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, Prometheus simply waits longer to pull the metric, while the push model will fail.

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

Its pervasiveness became especially evident when cloud native ecosystems started to build tools and standards on top of Prometheus.[2] This means that any tool sets or vendors that are compatible with Prometheus are now *forced to be interoperable* with each other.

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

Prometheus: The Good

For good or ill, the industry is adopting Prometheus and it has become the de facto standard for cloud native observability for metrics. After Kubernetes, it was the second project to attain "graduated" status from the CNCF, which requires meeting stringent criteria.

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

Dimensional metric data model

> 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, contrasting with StatsD, which primarily employs a simplistic model focusing on counters and timers without inherent support for dimensional data or a specialized query language.

Service discovery

Prometheus can use service discovery native to the system Prometheus is monitoring. For example, Prometheus can self-discover pod 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.

Mature specification

Prometheus has reached a level of maturity that makes it a stable and reliable solution for many organizations.

Prometheus: The Not-So-Good

However, as with all good systems, Prometheus 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 because you have to coordinate collection between multiple instances. You can solve this problem easily by using tools like PromLens and annotating metrics only when necessary.

Hard to operate

Prometheus is hard to operate. 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 it running. You end up spending time on Prometheus that you could (and should!) be spending on your core business applications instead.

Horizontal scalability

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

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, by default, lacks horizontal scaling capabilities, leading to reliance on vertical scaling for large deployments. This approach necessitates the use of powerful servers with extensive CPU and memory resources. The approach poses another significant challenge as well: it creates a single point of failure, as an outage in the server’s region can disrupt the entire system. This is particularly problematic in cloud native environments where reliability is paramount. Additionally, managing such a setup is complex and resists automation, making it more akin to treating the server as a “pet” rather than “cattle,” as per the popular cloud analogy. Finally, the scalability of Prometheus is inherently limited; even in the cloud with its vast array of compute resources, there’s a ceiling to how much a single server can be scaled vertically.

> That said, there are ways to scale Prometheus servers horizontally. Projects such as Thanos, Cortex, and Mimir aim to add horizontal scalability to Prometheus. However, once you reach the point where you need to scale Prometheus horizontally, we suggest you look into fully managed options. The complexity of running horizontally scaling Prometheus usually outweighs the benefits of maintaining these systems, with very few exceptions.

OpenTelemetry

As more organizations and practitioners standardized to Prometheus for their metrics, another question arose. What about logs and traces? This leads to the challenge of how to deal with the fragmentation of tools to generate telemetry for logs, metrics, and traces.

Many tools and solutions were crafted to solve this challenge; the most well-known ones were OpenCensus and OpenTracing. OpenTracing focused on telemetry for tracing, while OpenCensus focused on telemetry for both metrics and tracing.

By 2019, a committee was formed that aimed to combine the efforts of OpenCensus and OpenTracing into building a standardized and unified set of tools, which was dubbed OpenTelemetry.

What Is OTel?

OTel is an observability framework and toolkit designed to create and manage *telemetry data* such as traces, metrics, and logs. Crucially, OTel is vendor- and tool-agnostic, meaning that it can be used with a wide variety of observability backends.

OTel generates, collects, processes, and exports telemetry. However, OTel is not a backend system for logs, metrics, or traces;

you still need a system to send the telemetry generated by OTel for further analysis or safekeeping.

OTel is not one system like Prometheus; it's an umbrella project that combines the effort of building multiple subsystems to generate high-quality, ubiquitous, and portable telemetry to enable effective observability.

The OTel Specification

Unlike Prometheus, which inadvertently built a standard, OTel is deliberately building a standard, the *OTel specification*, that can be used for any implementation.

OTel SDK

OTel SDK, also known by engineers as client libraries, allows us to create telemetry that we can install depending on which programming languages we are writing our application in. The client libraries can either generate telemetry automatically or manually.

Libraries have built-in automatic instrumentation. For example, HTTP metrics, gRPC tracing, and even Express.js metrics are automatically generated when you install these libraries and set them up in your JavaScript application. However, there are edge cases; not every library would automatically generate metrics, and you would need to configure them.

Manual instrumentation uses primitives that the client libraries will allow you to use to generate specific signals about your application or to add contextual metadata to the metrics, spans, or logs emitted.

OpenTelemetry Collector

The OpenTelemetry Collector functions as an intermediary for telemetry data, equipped with three core components: an ingestion endpoint that receives data and also translates incoming telemetry into OTel formats (OpenTelemetry Protocol [OTLP] over HTTP, OTLP over gRPC); a processor that handles filtering, batching, and transforming the data; and an exporter that transmits the processed telemetry data to various backends.

There are also multiple vendor exporters that you can use depending on where you want your telemetry data to end up. Additionally, there is a growing list of exporter, collector, receiver, and client instrumentation libraries in the OpenTelemetry Registry.

OTel: The Promise

OTel can be used for instrumenting logs, metrics, and traces to emit telemetry via a standard format. It promises a single unified standard for observability, simplifying the telemetry process, and supports multiple vendors and open source software (OSS) with no vendor lock-in. Further, it allows extensibility. Developers can build upon the specifications to extend OTel to fit their specific needs.

The willingness of popular vendors, libraries, and languages to support OTel means it's easier for developers to emit telemetry in OTel format.

Another promise of OTel is the ability to correlate signals from multiple sources, like logs to metrics correlation, metrics to traces, and even logs and metrics to traces, using a standard specification. Imagine jumping into one correlation ID for a failed HTTP request and finding all the downstream logs, metrics, and even traces!

OTel: The Reality

The learning curve to fully understand all the components of OTel and effectively use it in production can be steep, especially for practitioners who are used to working with proprietary observability systems.

Another important difference to note is that, unlike Prometheus, which uses a pull system, OTel uses a push system with a collector.

Limitations of maturity

Being an umbrella project, OTel has multiple levels of maturity depending on which programming language you are using and what types of signal you want to emit.

For example, as of this writing in November 2023, the Python trace and metric client libraries are stable, but logs are experimental. Golang traces are stable, but metrics are mixed, and logs have not yet been implemented. To get the full level of maturity, visit the OpenTelemetry status page.

The current limitations of maturity mean that fully adopting OTel across telemetry types will be an ongoing project until all the languages and frameworks your organization supports are stable.

Backend support

OTel is vendor neutral; however, different backends offer varying levels of support for OTel. Backends may not fully use OTel's capabilities.

Where to Start with OTel

Because of the increased complexity, we suggest those interested in adopting OTel begin by just running the collector. Simply running the collector will give you a good feel for how the rest of the OTel ecosystem works.

Having the collector will allow you to start utilizing telemetry prewritten by tools you are already familiar with. For example, if you are running NGINX Ingress Controller, you can follow the Kubernetes NGINX Ingress OpenTelemetry guide to start sending telemetry to your collector.

Once you run a collector, you will want to try your hand at configuring auto-instrumentation in the collector to see what telemetry you can get from your system out of the box. Additionally, we suggest you try to do Chronosphere's walkthrough of OTel using JavaScript and automatic instrumentation or view a practical demonstration of OTel in action.

Implications of OTel's Approach

For bigger organizations with a greater need for flexibility in their telemetry systems, OTel is a better way than proprietary or vendor-specific collectors to handle telemetry. OTel provides a standard, flexible, and interoperable way to generate telemetry.

Being a standard across the industry, OTel allows practitioners to better their portability of skills when moving across different organizations or divisions of a bigger organization. OTel lock-in becomes less of a concern for practitioners.

The ability to correlate data using OTel is perhaps its greatest advantage. There is no other system in the cloud native observability space that has that potential out of the box.

But as with any new standard, there is an adoption curve. The trick is to understand when OTel is stable enough for your organization and when the complexity of adoption is minimal enough for adopting OTel.

As more and more systems adopt OTel, it will become an indispensable project that allows all practitioners to better organize and standardize telemetry systems.

A key weakness in the adoption of OTel is perhaps the erratic support for logs, where Fluent Bit can come in to fill the gap.

Fluent Bit

Fluent Bit is a vendor-neutral, open source solution that enables organizations to connect any data source to any destination. Organizations leverage Fluent Bit to create *observability pipelines* that can collect, process, and route data. It has a fully pluggable architecture that allows users to connect telemetry sources with various other destinations and perform many different types of processing (such as filtering, parsing, etc.) on the data while in flight.

Fluent Bit began as an outgrowth of the Fluentd project, which was created by Sadayuki Furuhashi in 2011 as an open source data collector that lets users unify log data collection and consumption. Fluent Bit was created in 2014 as a more lightweight, performant version for resource-constrained environments.

With over 12 billion downloads, the Fluent Bit project is one of the most widely adopted solutions to address logging challenges in cloud native environments. It includes support for OTel and Prometheus as both an input and an output, supports connectors that allow it to integrate with hundreds of other systems, and allows extensibility via plugins written in WebAssembly and Golang. The synergy between the broad telemetry capabilities of projects like OTel and Fluent Bit's specialized log-processing abilities allows for a solution that works for any scale of organization—from a lightweight system for transforming log entries to structured metric data to large-scale processing of logs via backends like Kafka or OpenSearch. Fluent Bit is available as a default logging option in the environments of most cloud service providers.

[1] "Prometheus Graduates Within CNCF," Cloud Native Computing Foundation, August 9, 2018, *https://oreil.ly/qt-vt*.

[2] Anne McCrory, “Ubiquitous? Pervasive? Sorry, They Don’t Compute,” *Computerworld*, March 20, 2000, *https://oreil.ly/juHHV*.

Conclusion

Open source projects have transformed the way we standardize the emission and storage of observability signals.

With their introduction, the landscape of cloud native observability has evolved. The challenge is no longer solely about data capture; emitting telemetry has become more straightforward than ever.

These projects aren’t silver bullets that magically address every observability concern. Instead, they serve as standards, enabling us to harness these observability signals efficiently. The overarching goal is to utilize these tools to foster improved business results, though the path to mastery may come with a pronounced learning curve.

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

To refine our focus and make a discernible impact, our thinking about cloud native observability must pivot away from the “three pillars” we discussed in our previous report, *Cloud Native Monitoring*, 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 observability that emphasizes remediating problems and improving business outcomes.

The three phases of observability help us focus on what matters most. However, it is indeed difficult to discuss observability data without noting that such data in the cloud has grown exponentially. This growth has led to higher costs while potentially decreasing business outcomes.

In response, we’ve defined the Goldilocks zone of observability, where we retain the essential observability data—not too much and not too little—to fully understand our systems. However, achieving the Goldilocks zone of observability is challenging without a proper framework. That's why we introduced the O11y DOC. This framework helps us delve into the nuances of implementation, covering technical details, aspects of governance, and process.

Using O11y DOC, we can maximize the value of our data while keeping costs low. We've discussed examples and case studies where large organizations have implemented O11y DOC with the support of fully managed platforms such as Chronosphere.

We recommend fully managed monitoring solutions over self-managed ones because the latter can be complex and costly. With platforms that natively support and promote the principles in O11y DOC, we can accelerate optimization using O11y.

Building an effective observability function is all about strategy. Keep your eyes on the desired outcomes, and your cloud native observability journey will have a promising start.

About the Authors

Kenichi Shibata is a cloud native architect at esure, specializing in cloud observability, security, cloud migration, and cloud native microservices implementation and architecture. He has worked in multiple global industries ranging from banking and insurance to media and retailing across Europe and Asia. He and his family are based in the United Kingdom.

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 site reliability engineering (SRE) teams that created and operated M3. Prior to that, he was a technical lead on the EC2 team at Amazon Web Services 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.