THE NEW STACK

Cloud Native Observability for DevOps Teams

The New Stack

Cloud Native Observability for DevOps Teams

Alex Williams, Founder and Publisher

Ebook Team and Guest Authors:

Catherine Paganini, Author

Danyel Fisher, Author

Franciss Espenido, Author

Gabriel H. Dinh, Executive Producer

Heather Joslyn, Ebook Editor

Jason Morgan, Author

Joab Jackson, Editor-in-Chief

Judy Williams, Copy Editor

Libby Clark, Vice President of Strategic Development

Peter Putz, Author

Steve Tidwell, Author

Susan Hall, Copy Editor

Supporting Team:

Benjamin Ball, Director of Sales and Account Management

Michelle Maher, Assistant Editor

Table of Contents

Sponsor

We are grateful for the support of our ebook sponsor:

logdna

LogDNA is a centralized log management solution that helps modern engineering teams be more productive in a DevOps-oriented world. It enables frictionless consumption and actionability of log data so developers can monitor, debug and troubleshoot their systems with ease.

Introduction

If you're reading this, you likely already work with cloud native applications and architecture, or your organization is embarking on a journey to the cloud. If so, you are already familiar with the overwhelming choices the cloud native landscape offers. So many tools and so many opportunities to make the wrong decisions.

In a word, so much complexity.

Even if your team is overseeing just one microservices cluster or just a few, those clusters may be deploying across more than one public cloud or a combination of cloud and on-premises servers. More complexity.

When an anomaly pops up—latency, a spike in application programming interface (API) calls, a sudden outage of an essential service—how do you know what's causing it? How do you know whether it's an isolated incident or a glitch that's going to crash everything?

The thing is, you can't know unless your whole team has full observability.

Nothing is more crucial to an organization's ability to not simply function but serve its customers, than observability. And nothing, perhaps, is more widely misunderstood.

Observability means inferring the internal state of a system from its external outputs. But it isn't just the ability to see what's going on in your systems. It's the ability to make sense of it all, to gather and analyze the information you need to prevent incidents from happening, and to trace their path when they do happen, despite every safeguard, to make sure they don't happen again.

Traditionally, observability has been the responsibility of operations engineers. But with the advent of DevOps teams, and more responsibility "shifting left" to developers, it's become every team member's job. If you're building an application, observability cannot be relegated to "add-on" status later in the application's life cycle. To think otherwise would be like building a car and leaving the speedometer, odometer and instrument lights for the dealership to install.

In a survey taken in January 2021 of more than 300 IT professionals by VMware Tanzu, 84% of participants said their cloud applications would have better availability and performance if more stakeholders, including developers, had visibility into their systems' overall infrastructure and performance metrics.

The current conversation about observability began before the introduction of game-changing cloud native tech like Kubernetes. In a much-cited 2013 blog post by Cory Watson, the tech world learned how engineers at Twitter sought ways to keep track of their systems as the company moved from a monolithic to a distributed architecture. At this time, as Watson described, Twitter focused its observability efforts on collecting and monitoring metrics, and on the visualizations generated from the data points it collected:

> **"Charts are often created ad hoc in order to quickly share information within a team during a deploy or an incident, but they can also be created and saved in dashboards. A command-line tool for dashboard creation, libraries of reusable components for common metrics, and an API for automation are available to engineers.**

Logging and tracing were addressed in a single paragraph, under the heading "Related Systems."

Twitter, Watson wrote, created a command-line tool to help its engineers create their own dashboards to keep track of their metrics-generated charts:

> **"The average dashboard at Twitter contains 47 charts. It's common to see these dashboards on big screens or on engineer's monitors if you stroll through the offices. Engineers at Twitter live in these dashboards!**

As the decade of cloud computing rolled on, more engineers began to live in their dashboards. It's not enough to merely monitor data points, they learned. And so the notion spread that observability didn't mean mere monitoring, but was based on three pillars, which became known as:

1. **Metrics:** measurement of various activities in a system.
2. **Tracing:** the path taken by a request as it moves through a distributed system.
3. **Logs:** records of activity within the system.

Increasingly, the conversation around observability is moving beyond the three pillars, taking a more nuanced view. There's greater awareness of how those three pillars fit together and a greater emphasis on analysis. DevOps teams are becoming more cognizant of the importance of measuring what truly matters to meet service-level objectives (SLOs).

And managers, struggling with high turnover and a relatively small pool of talent from which to hire, are trying to figure out how to alleviate the human cost of "pager fatigue" — the demand for operations engineers to respond to alerts, at all hours of the day or night, that may or may not signal a business-critical incident.

SLOs, a concept documented by Google's site reliability engineering team in its SRE book, can vary widely depending on each organization or even team's purpose, such as achieving a particular latency for a certain volume of requests or determining how many customers can make purchases in an online shopping cart application at once. Service-level indicators (SLIs) are the signals that illuminate a robust observability process and can show whether a team is on track to meet its SLOs or if a problem is brewing.

And, as stated previously, distributed systems and cloud native technologies add additional layers of complexity to observability. After all, Kubernetes runs everywhere, and "everywhere" can be tough to track.

In the VMware Tanzu survey, 90% of the IT professionals who participated said that distributed applications create monitoring challenges of an order of magnitude bigger than other applications.

More than 80% of participants in the survey said that legacy monitoring tools aren't sufficient to track modern cloud applications. And only 8% of respondents said they are "very satisfied" with their organization's current monitoring tools and processes.

Cloud technologies do not always lend themselves easily to observability. Plain vanilla Kubernetes, for instance, offers only very basic functions, through kubectl, for checking on the status of objects in a cluster and no full-fledged native logging solutions, as Franciss Espenido, LogDNA's senior technical partnerships program manager, writes in his chapter of this ebook.

But overcoming these challenges can pay off for businesses.

In the VMware Tanzu survey, 92% of respondents said observability drives better business decisions. One example of how observability is becoming embedded into the way businesses run involves Adidas, the sportswear retailer.

Adidas found that as it scaled up, it needed to make observability a lot easier, according to Rastko Vukasinovic, the company's director of solution architecture. So it built its own holistic monitoring system that allowed it to not only collect and watch technical metrics, but also business data.

Its worldwide DevOps teams now compile code more than 10,000 times a day. And Adidas' overall digital transformation has helped its e-commerce revenue soar from $47 million in 2012 to $4.7 billion in 2020.

For developers, having a greater knowledge of observability — and building secure, observable applications that easily lend themselves to meeting SLOs — means contributing more to overall business goals. Fifty-five percent of developers' time is spent maintaining and managing custom applications that serve current business needs, according to a 2019 survey by 451 Research; only 45 percent of time is spent building new applications to help the business differentiate itself from their company's competition.

According to 451 Research's 2020 report on observability, commissioned by Sumo Logic, greater focus on SLOs can help developers spend more time on applications that fuel new business:

> **"With visibility into key objectives that describe the performance that's important for end users, developers can prioritize the work they do on existing applications for the most important performance problems rather than, for instance, on infrastructure or application anomalies that have no negative impact on users.**

The story of cloud native observability is still adding new chapters. The tech world is watching OpenTelemetry — an open source project aimed at creating a standardized set of tools, APIs and software development kits (SDKs) — which is now a sandbox project at the Cloud Native Computing Foundation.

In this ebook, The New Stack has gathered some of its best articles on the current state of observability, with contributions from experts at LogDNA, Buoyant, Dynatrace and Honeycomb. It's aimed at every member of a DevOps team, making the case for full-stack involvement in making sure cloud native applications and systems run smoothly and keep customers satisfied. The days of throwing application code "over the wall" and letting operations engineers deal with the consequences are over.

— Heather Joslyn, The New Stack

Heather Joslyn is features editor for The New Stack. Previously, she was editor-in-chief of the cloud native consultancy Container Solutions and an editor/reporter at The Chronicle of Philanthropy and Baltimore City Paper. On Twitter, she's @ha_joslyn.

SECTION 01

Observability's Role in Cloud Native Applications

CHAPTER 01

Meet the Cloud Native Observability Stack

Original article published 17 May, 2021, by Catherine Paginini and Jason Morgan, Buoyant.

Editor's Note: *First things first. What tools are available to help a DevOps team improve their observability of cloud native applications? Which problems do they solve, and how do they solve them? Here, experts from Buoyant offer a birds-eye view of the Cloud Native Computing Foundation's cloud native landscape, focusing on the tools that can enable observability.*

Let's start by defining observability and analysis. Observability is a system characteristic describing the degree to which a system can be understood from its external outputs. Measured by central processing unit (CPU) time, memory, disk space, latency, errors, etc., computer systems can be more or less observable. Analysis, on the other hand, is an activity in which you look at this observable data and make sense of it.

To ensure there is no service disruption, you'll need to observe and analyze every aspect of your application so any anomaly gets detected and rectified right away. This is what this category of the Cloud Native Computing Foundation (CNCF) landscape is all about. It runs across and observes all layers, which is why it's on the side and not embedded in a specific layer.

Tools in this category are broken down into logging, monitoring, tracing and chaos engineering. Please note that while listed here, chaos engineering is rather a reliability than an observability or analysis tool.

Logging

What It Is

Applications emit a steady stream of log messages describing what they are doing at any given time. These log messages capture various events happening in the system — such as failed or successful actions, audit information or health events. Logging tools collect, store and analyze these messages to track error reports and related data. Along with metrics and tracing, logging is one of the pillars of observability.

Problem It Addresses

Collecting, storing and analyzing logs is a crucial part of building a modern platform. Logging tools can help with, or perform one or all of those tasks. Some tools handle every aspect, from collection to analysis, while others focus on a single task like collection. All logging tools aim at helping organizations gain control over their log messages.

How It Helps

When collecting, storing and analyzing application log messages, you'll understand what an application was communicating at any given time. But note this: Logs represent messages that applications can deliberately emit. They won't necessarily pinpoint the root cause of a given issue.

That being said, collecting and retaining log messages over time is an extremely powerful capability and will help teams diagnose issues and meet regulatory and compliance requirements.

Technical 101

While collecting, storing and processing log messages is by no means a new problem, cloud native patterns and Kubernetes have caused significant changes in the way we handle logs. Traditional approaches to logging that were appropriate for virtual and

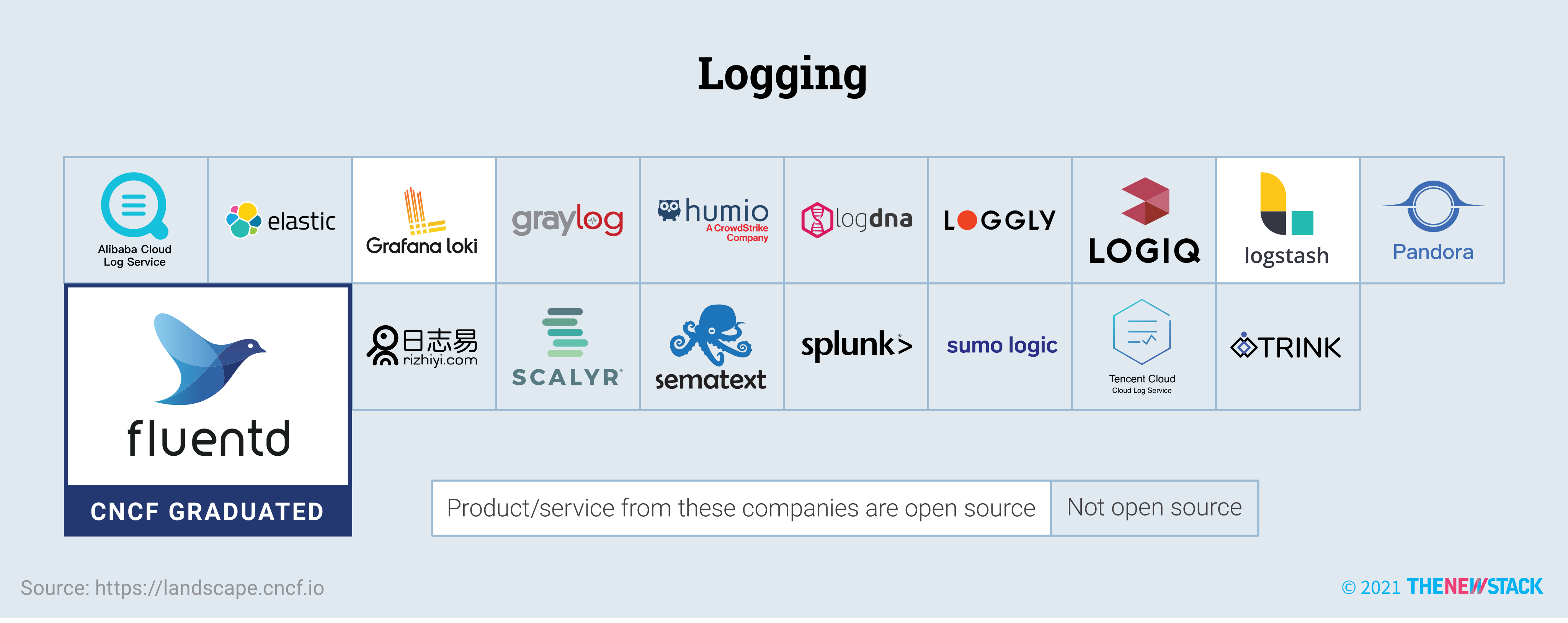


FIG 1.1: *Logging tools currently included in the CNCF landscape.*

physical machines, like writing logs to a file, are ill-suited to containerized applications where the file system doesn't outlast an application. In a cloud native environment, log-collection tools like Fluentd run alongside application containers and collect messages directly from the applications. Messages are then forwarded on to a central log store to be aggregated and analyzed. At this writing, Fluentd is the only CNCF project in this space.

- **Buzzwords:** Logging
- **Popular Projects:** Fluentd and Fluentbit, Elastic Logstash

Monitoring

What It Is

Monitoring refers to instrumenting an application to collect, aggregate and analyze logs and metrics to improve our understanding of its behavior. While logs describe specific events, metrics are a measurement of a system at a given point in time. They are two different things, but both are necessary to get the full picture of your system's health. Monitoring includes everything from watching disk space, CPU usage and memory consumption on individual nodes to doing detailed synthetic transactions to see whether a system or application is responding correctly and in a timely manner. There are a number of different approaches to monitoring systems and applications.

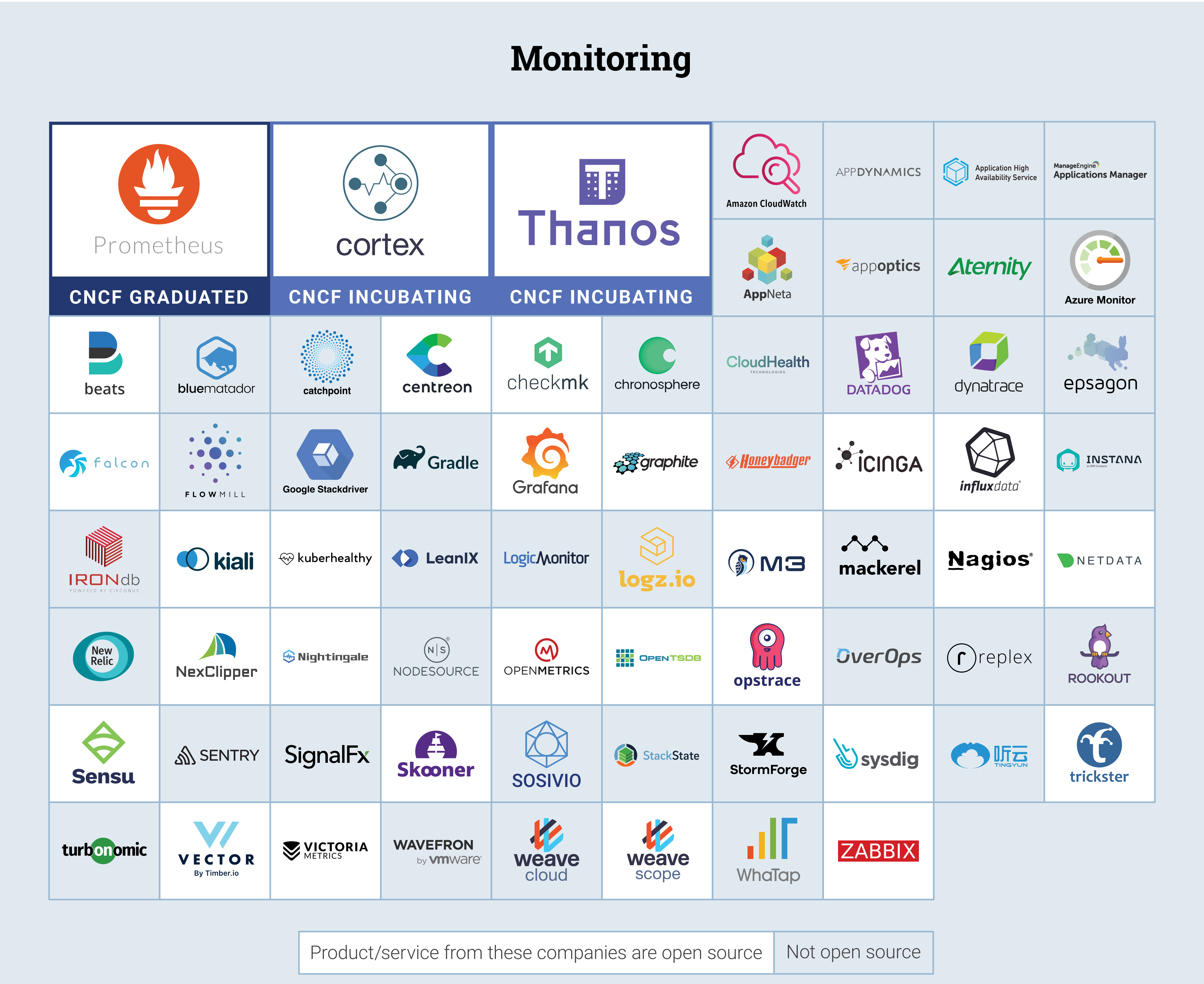
Problem It Addresses

When running an application or platform, you want it to accomplish a specific task as designed and ensure it's only accessed by authorized users. Monitoring allows you to know if it is working correctly, securely and cost-effectively; is only accessed by authorized users; and/or any other characteristic you may be tracking.

How It Helps

Good monitoring allows operators to respond quickly and potentially automatically when an incident arises. It provides insights into the current health of a system and

FIG 1.2: *Monitoring tools currently included in the CNCF landscape.*



Source: https://landscape.cncf.io

watches for changes. Monitoring tracks everything from application health to user behavior and is an essential part of effectively running applications.

Technical 101

Monitoring in a cloud native context is generally similar to monitoring traditional applications. You need to track metrics, logs and events to understand the health of your applications. The main difference is that some of the managed objects are ephemeral, meaning they may not be long lasting, so tying your monitoring to auto-generated resource names won't be a good long-term strategy. There are a number of CNCF projects in this space that largely revolve around Prometheus, the CNCF-graduated project.

- **Buzzwords:** Monitoring, Time series, Alerting, Metrics
- **Popular Projects/Products:** Prometheus, Cortex, Thanos, Grafana

Tracing

What It Is

In a microservices world, services are constantly communicating with each other over the network. Tracing, a specialized use of logging, allows you to trace the path of a request as it moves through a distributed system.

Problem It Addresses

Understanding how a microservice application behaves at any given point in time is an extremely challenging task. While many tools provide deep insights into service behavior, it can be difficult to tie an action of an individual service to the broader understanding of how the entire application behaves.

How It Helps

Tracing solves this problem by adding a unique identifier to messages sent by the application. That unique identifier allows you to follow, or trace, individual

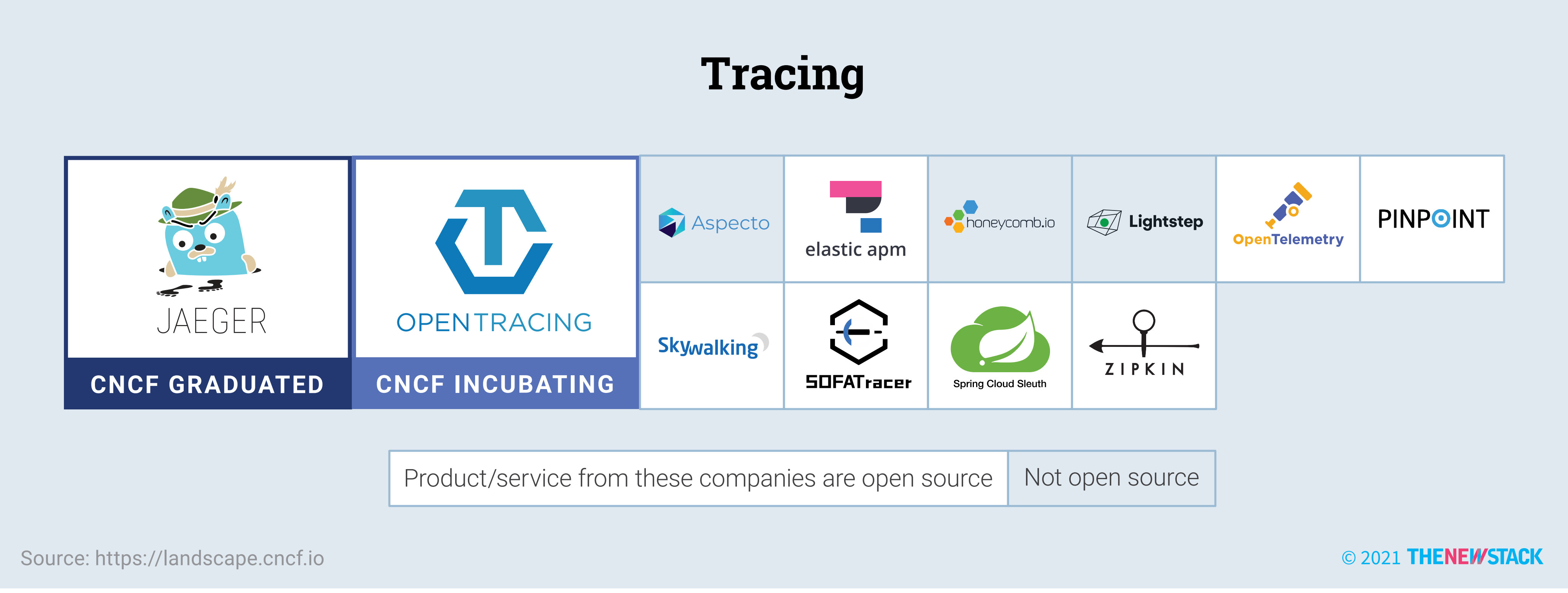


FIG 1.3: *Tracing tools currently included in the CNCF landscape.*

transactions as they move through your system. You can use this information to see the health of your application and also to debug problematic microservices or activities.

Technical 101

Tracing is a powerful debugging tool that allows you to troubleshoot and fine-tune the behavior of a distributed application. That power does come at a cost. Application code needs to be modified to emit tracing data, and any spans need to be propagated by infrastructure components in the data path of your application, specifically service meshes and their proxies. Jaeger and Open Tracing are CNCF projects in this space.

- **Buzzwords:** Span, Tracing
- **Popular Projects:** Jaeger, OpenTracing

Chaos Engineering

What It Is

Chaos engineering refers to the practice of intentionally introducing faults into a system to create more resilient applications and engineering teams. A chaos engineering tool will provide a controlled way to introduce faults and run specific experiments against a particular instance of an application.

Problem It Addresses

Complex systems fail. They fail for a host of reasons, and the consequences in a distributed system are typically hard to understand. Chaos engineering is embraced by organizations that accept that failures will occur and, instead of trying to prevent failures, practice recovering from them. This is referred to as optimizing for mean time to recovery, or MTTR.

Side note: The traditional approach to maintaining high availability for applications is referred to as optimizing for mean time between failures, or MTBF. You can observe this practice in organizations that use things like change review boards and long change freezes to keep an application environment stable by restricting changes. The authors of the book "Accelerate" suggest that high-performing IT organizations achieve high availability by optimizing for MTTR instead.

How It Helps

In a cloud native world, applications must dynamically adjust to failures, a relatively new concept. That means, when something fails, the system doesn't go down completely but gracefully degrades or recovers. Chaos engineering tools enable you to experiment on a software system in production to ensure they gracefully degrade or recover if a real failure occurs.

In short, you experiment with a system because you want to be confident that it can withstand turbulent and unexpected conditions. Instead of waiting for something to happen and finding out how your application fares, you put it through duress under controlled conditions to identify weaknesses and fix them before chance uncovers them.

Technical 101

Chaos engineering tools and practices are critical to achieving high availability for your applications. Distributed systems are often too complex to be fully understood by any one engineer, and no change process can fully predetermine the impact of

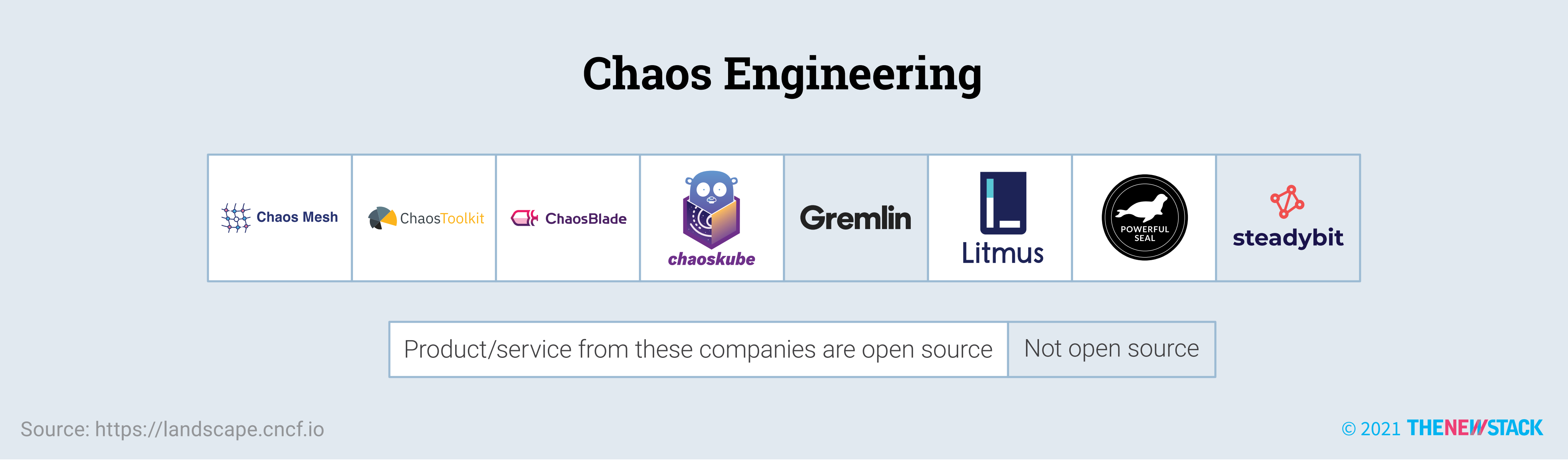


FIG 1.4: *Chaos engineering tools currently included in the CNCF landscape.*

changes on an environment. By introducing deliberate chaos engineering practices, teams are able to practice and automate recovering from failures. Chaos Mesh and Litmus Chaos are CNCF tools in this space, but there are many open source and proprietary options available.

- **Buzzwords:** Chaos engineering
- **Popular Projects:** Chaos Mesh, Litmus Chaos

Conclusion

As we've seen, the observability and analysis layer is all about understanding the health of your system and ensuring it stays operational even under tough conditions. Logging tools capture event messages emitted by applications, monitoring watches logs and metrics, and tracing follows the path of individual requests. When combined, these tools ideally provide a 360-degree view of what's going on within your system. Chaos engineering is a little different. It provides a safe way to verify that the system can withstand unexpected events, basically ensuring it stays healthy.

Jason Morgan is a technical evangelist for Linkerd at Buoyant. He's responsible for helping to educate engineers on Linkerd, the original service mesh.

Catherine Paginini is head of marketing at Buoyant, the creator of Linkerd. A marketing leader turned cloud native evangelist, she is passionate about educating business leaders on the new stack and the critical flexibility it provides.

CHAPTER 02

Observability Beyond Metrics, Logs and Traces

Original article published 6 August, 2020, by Danyel Fisher, Honeycomb.

Editor's Note: *By themselves, the three pillars of observability — metrics, tracing and logs — generate data, but that data is of limited value without analysis. Here, the author, principal design researcher at Honeycomb, walks us through what it means to use the perspective offered by each of the pillars to get a comprehensive view of what's going on in a system.*

Observability is a relatively new concept being applied to production software systems. In an attempt to explain what it means and how to achieve it, some have tried to break it into three parts with the "three pillars of observability," splitting it into metrics, tracing and logs. But I think that misses a critical point of what it means to observe a system.

Observability measures how well you can understand your system's internal state based on signals and externally visible output. The term describes one cohesive capability. The goal of observability is to help you clearly see the state of your entire system.

When you unify these three "pillars" into one cohesive approach, a new ability to understand the full state of your system in several new ways also emerges. That new understanding doesn't exist within the pillars on their own; it's the collective sum of the individual parts when you take a unified approach. Simply defining observability by its individual components misses the bigger picture.

Considering Capabilities on Their Own

A good analogy for describing how this system comes together is one of using color-filtering lenses. Each lens removes some wavelengths of information in exchange for emphasizing others. When you need to focus on particular shades of red, using a red lens can help, and it's the tool you should reach for. But to see the big picture, you need to be able to see all the vivid colors as they come together in real time.

For example, let's say that you notice that a production service seems to be acting up. Let's look at how different lenses see that problem.

Monitoring and Metrics

An alert has been triggered, notifying you that the number of incoming connections is higher than a specified threshold. Does that mean users are having a poor experience in production? That's unclear. But this first lens, monitoring tools (i.e., metrics), can tell us some very important things we may need to know.

The metrics lens reflects the state of the system as a time series of numbers that are essentially used as gauges. At any given time, is each measure performing over or under thresholds we care about? That's certainly important information. Metrics become even more important when we can slice down your time series and use a different one for each machine or each endpoint.

"Simply defining observability by its individual components misses the bigger picture."

— **Danyel Fisher,** Honeycomb

A metrics monitoring tool such as Prometheus can be used to trigger an alert when the metric exceeds specified thresholds for a predefined length of time. Most metrics tools let you aggregate performance across a small number of labels. For your triggered alert, that might help you simply see which service is experiencing a problem or on which machines it's happening.

FIG 2.1: *The view through a blue lens.*

To narrow down more precisely, though, you need to see your metrics with high cardinality to track hundreds or thousands of different values. This can let you create views that compare enough time series that you can infer more granular insights. You can pin down which particular service endpoints are experiencing errors or perhaps information about which users are actively using them.

For the analogy, metrics can be represented as a magnifying glass with a blue lens on your data. In which case, your view of what's happening in production would look a bit like the image above.

Tracing

We still have a problematic service in production. With high-cardinality metrics, we were able to pin down what is failing and which users are affected, but the reasons why are still unclear. The next lens, tracing, can tell us some very important things we may need to know to find out. Traces help you look at individual system calls and understand the individual underlying steps taken to return a result.

Tracing tools, such as Jaeger, provide wonderful visibility into what's happening with underlying components. Traces allow you to see things like which component took the longest or shortest amount of time to execute or whether specific underlying functions resulted in errors. Traces can be a useful way to more deeply investigate an alert triggered by your metrics system.

For example, you might be able to use the information found from your metrics dashboard to look for a trace that hits the same endpoint as the one experiencing problems. The trace might visually show us that the slowest part of a problematic request being issued, or an individual trace span, was a system call to a particular database. That span is taking significantly more time than usual.

But, already, we've hit a snag in continuity. The process of getting from a metrics tool to a tracing tool is bumpy. They're two different types of tools collecting

FIG 2.2: *The view through a red lens.*

different types of data. To understand that the span is taking significantly longer than usual, you would need to somehow correlate information between both tools. That process can be time-consuming and error-prone, often not providing the fast accuracy needed to debug production issues. The key data you need to correlate between the two might not even be available in both systems.

For example, your metric may indicate that page load times are spiking on a particular page for some users. But unless our metrics system and tracing system share the same underlying data, we may not be able to find traces that reflect that particular change in load times. We may have to search an entirely different system to find sample traces that will illustrate the problem we're trying to hunt down.

In our analogy, tracing can be represented as a magnifying glass with a red lens. From this lens, the picture looks pretty different, but there are enough common attributes that we can successfully identify some parts in common and can often stay oriented between the images. Under this lens, some parts stand out more than others, while some aspects of detail entirely disappear.

Logging

Our work with tracing showed us where the underlying problem occurred: in one particular call to a certain database. But why did the database calls become slow? To continue to get to the root of the issue, you will need to examine what happened at the system or software level. The third lens, logs, provides the necessary insights into raw system information that can help us figure out what happened to your database.

For example, scrolling around in logs for that database, you might find some warnings issued by the database to show that it was overloaded at the time or you may identify particularly slow queries or you may discover that the event queue became too long. Logs help shed light on an issue once you know where to look.

But that microlevel view is extremely limited. For example, if you wanted to know

FIG 2.3: *The view through a green lens.*

how often this particular problem had occurred, you'd need to go back to the metrics tool to examine the history of that particular database queue, presuming you had metrics for that.

Like the other lenses, the process of switching between tools (in this case, from tracing to logging,) requires a new set of searches, a new set of interactions — and, of course, more time.

For the analogy, we can think of logs as a green lens.

Putting the Lenses Together

Unfortunately, some observability solutions simply lump the views from these lenses together. They do that as separate capabilities, and it's up to the observer to determine the differences.

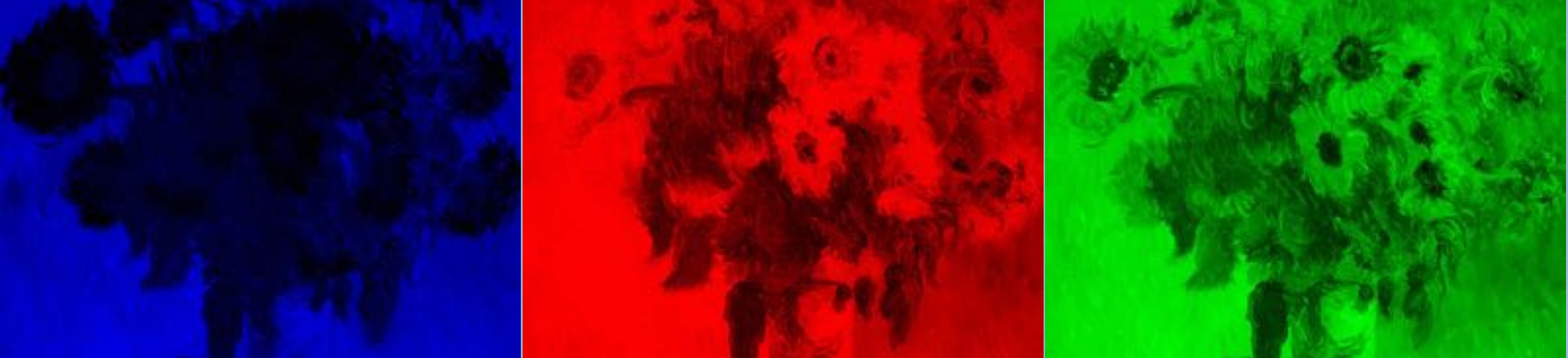


FIG 2.4: *A comparison of what each lens reveals.*

That's not a bad start. As we can see, some attributes are completely invisible in one view, but easy to see in others. Side-by-side comparisons can help alleviate those gaps. For this example, each image brings different aspects more clearly into view: The blue image shows the outline of the flowers best; the red shows the detail in the florets; and the green seems to best highlight shading and depth.

But these three separate lenses have inherent limitations. Observability isn't just the ability to see each piece at a time, it's also the ability to understand the broader picture and to see how these pieces combine to show you the state of your system.

The Greater Sum of the Parts

The truth of your systems is that the aspects highlighted by each lens don't exist separately in a vacuum. There is only one underlying system running with all its rich attributes. If we separate out these dimensions — if we collect metrics monitoring separately from log and traces — then we can lose sight of the fact that this data reflects the state of one single underlying system.

A unified approach collects and preserves that richness and dimensionality. To see the full picture, we need to move through the data smoothly, precisely, efficiently and cohesively. When these lenses are unified into one tool, we can do things like quickly discover where a trace contains an anomaly that is repeatedly occurring in other traces, including finding out where and how often.

Observability tools don't just need these essential lenses, they also need to maintain a single set of telemetry data and storage that retains enough rich context that we can view system events from the perspective of metrics, tracing and logs all at once. Monitoring, tracing and logs shouldn't be different sets of data. Rather, they should be different views of the same cohesive picture. These lenses aren't meant to view separate pictures. Each one simply brings certain aspects into a sharper focus. Observability tools should make it so that any point on a metrics line chart or a heatmap connects to corresponding traces; any trace span should be able to query for views into log data.

The power of observability doesn't just come from not having to switch contexts. Having a single data store also unlocks the ability to do things like visually slice across your data. In other words, you can look at how two sets of events differ from each other across all their various dimensions (i.e., fields). That's the sort of analysis that isolated metrics systems simply can't show because they don't store the individual events or that would be prohibitively exhausting for a logging system.

Manually connecting these lenses means needing to make correlations on your own to find non-obvious problems. For our analogy, that would be like seeing an area

FIG 2.5: *Claude Monet, "Bouquet of Sunflowers," 1881*

that appears to be light-colored in both the green and red lens and intuiting that the common color must actually be yellow. Some artists may be able to infer that, a system could do the math for you, or you could flip back and forth between images staring at where bits contrast.

Observability lets you see the beautiful and complete picture that is your production software systems. You don’t need the skill and experience to combine those lenses in your mind to divine which shades of yellow might be a problem. All you need are parts that aren’t smaller than the sum of the whole.

Danyel Fisher is principal design researcher at Honeycomb. His work centers on data analytics and information visualization. He focuses on how users can better make sense of their data. He supports data analysts, end-users, and people who just happen to have had a lot of information dumped in their laps. His research perspective starts from a background in human-computer interaction and brings together cutting-edge data management techniques with new visualization algorithms.

CHAPTER 03

How Observability Enables DevOps

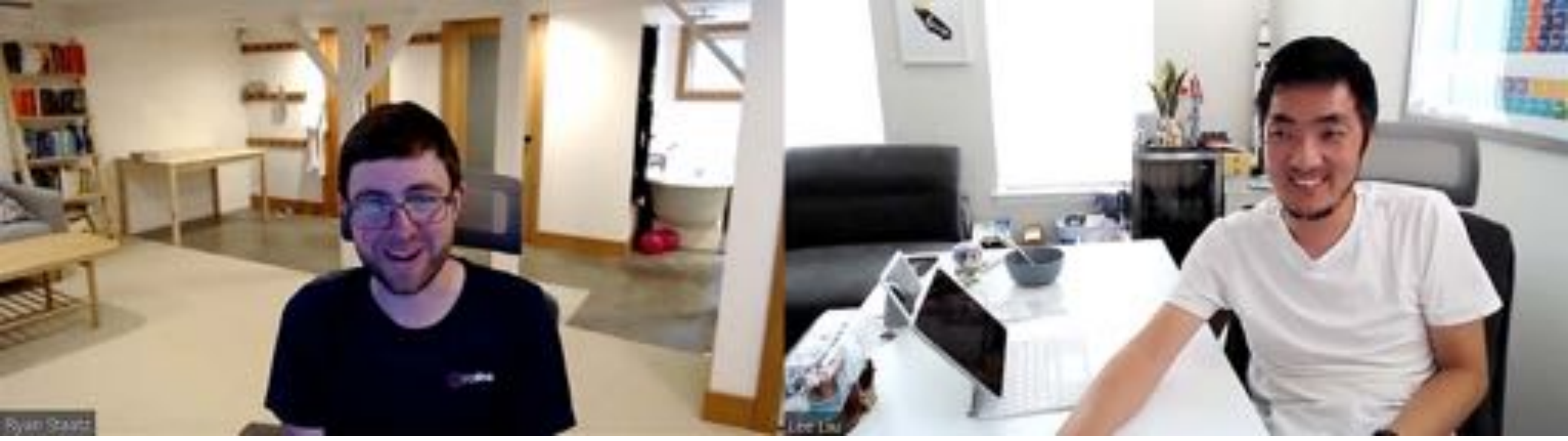
Original article published 15 July, 2021, by Joab Jackson, The New Stack.

Editor's Note: *What does it mean in practical terms if everyone on the team can take full advantage of observability? In this Q&A session, two experts from LogDNA — Lee Lui, the company's co-founder, and Ryan Staatz, a systems architect — talked to The New Stack's Editor-in-Chief Joab Jackson about how the right tools and organizational culture can help DevOps teams not only deliver more efficiently, but also gain greater understanding of each others' jobs and challenges.*

For the folks at LogDNA, DevOps is all about empathy, in getting beyond the "hot potato" mindset of not worrying about something because it is not your problem, and instead getting the whole crew working together on a shared challenge. LogDNA's own roots are in solving a problem that traditional log management software couldn't: ingesting unstructured logs and structuring them so that the developer or administrator doesn't have to do it by hand.

The company has taken this holistic approach to its own product, building out a platform that gives both the developer and the site reliability engineer (SRE) access to the information they need to build and maintain software.

We interviewed longtime LogDNA System Architect Ryan Staatz and company co-founder Lee Liu about how a company can embrace DevOps best practices even as it dramatically increases in size, as well as the importance of observability for DevOps and how the best observability tools are made for multiples types of users.



The interview has been edited for brevity and clarity.

What is your definition of DevOps?

Ryan Staatz: DevOps is a nebulous term. It's somewhere between developer and operations. If you look at Wikipedia, it'll say it's a very specific set of methodology and set of practices, which is the formal definition. But like most things, once the industry gets ahold of it, it takes on a definition of its own.

And, at least for me, DevOps is figuring out how you build that cohesive life cycle for the life span of the application, where you go from developing in code to testing [continuous integration/continuous delivery], to pushing it out to production, and getting feedback, that whole loop. I feel like a lot of that loop has gaps, depending on what stage of maturity your organization is in.

When you're really small, and you only have a few people, it's not hard to have one or two people just own the entire life cycle. Communication is really fast. As you grow, you have more teams and more people available to you to run faster. But you have these gaps now that start appearing in your life cycle where an app might be after the developer hands it off, or it might be between releases, or maybe you don't have a release team.

DevOps is the idea that you're owning those gaps, you want that whole life cycle to work. It's this idea that development and operations, while they're largely seen as separate entities doing separate things, have some of the same goals. We've seen many security and operations responsibilities shift left in the life cycle. Now, we're seeing responsibilities that were traditionally owned by dev and product, like aligning business objectives, shift right. Ultimately, we all have to be in sync in order to deliver awesome products.

What are some of the gaps between Dev and Ops in many organizations?

Staatz: The age-old gripe that I hear about Dev and Ops teams is that it is this hot potato that keeps getting tossed over the fence. That's an unhealthy interaction that you want to avoid. In DevOps, you have a shared goal that you're working toward.

The attitude that you'll see in some places is, "How do I make this not my problem?" And that's, I think, just an unfortunate part of being at a company that may not have all the resources set aside or may not have all the right processes in place. And no company is perfect.

Some of it just has to do with the fact that as you grow as a company, you have more strictly defined roles. And when you get to areas that are somewhat shared, it can be harder to basically say, "Well, who owns that?" Well, the answer is nobody really.

There is no magic fix, you know? It's usually more along the lines of encouraging people to think about what other teams are doing, think about why they're doing it. Understand their goals, maybe look at their tools. So, it takes a little bit of effort, honestly.

Lee Liu: One of the challenges we've seen come up when Dev and Ops teams aren't aligned is risk adversity. In the airline industry, you can see this problem exacerbated over decades. You look into a cockpit, and you see things that are really old, like that computer that looks like it belonged on my dad's desk, right? Everything else is very modern about the planes, like the seatback TVs, and the like. But the actual technology seems really old.

And I think that that could be one of the things that comes out of risk adversity, where the Ops team got burned one too many times. So now there's PTSD that's developed over the years. And the Ops team is like, "Just don't make too many big changes, and then things will be more stable.”

What the airline industry faces is when airplanes crash, people die. So, they are not eager to change the software. Software may have been in use since 1990, but it doesn’t need to change. It will fly the plane, and it will land the plane. That’s all it needs to do. It doesn’t need to be modernized.

What ends up happening is now [you’re] using 20-year-old technology, and I don’t think that is the right approach for software startups at least. There is always risk in deploying new code, for sure. But it’s up to us using the tools, especially observability tools, to help facilitate that [so] when problems happen, we know how to fix them.

What could observability bring to DevOps?

Staatz: Part of the whole life cycle is figuring out where things are breaking down, right? Conversely, where in this life cycle can I improve things, so things don’t break down?

At the end of the day, observability is all about getting [the] details around what’s going on.

There are lots of different tools out there that help do different things and often in different parts of that life cycle. It can be specific to some section of your infrastructure. It might be from something in your application code that is sending data somewhere. You can have things that can be logged. It could be metrics. It could be something that keeps an eye on the state of your environment.

And so, there becomes this sort of interesting intersection, like we talked about earlier, where different teams overlap in certain spaces, and it becomes unclear as to who owns what in those spaces. If you have observability tools that help both of those teams achieve their goals, and hopefully their goals are somewhat aligned, then that can be a huge help.

"One of the best things you can do is being a good Samaritan toward other people who are either developing code or maintaining your code."

— Ryan Staatz, LogDNA

We have a product feature we developed called Kubernetes Enrichment. It displays information about the state of the Kubernetes cluster relevant to that log line in that application to give you a sense of if there is something going wrong in the environment at this time. And seeing that information is really helpful for two different teams, this overlap of development and infrastructure folks. And so it helps clarify an area that's somewhat vague: what might be going on at that point in time.

What would somebody do if they didn't have Kubernetes Enrichment? How would they get their Kubernetes logs? Do they even have access to that information?

Staatz: There are some complications that are sort of exacerbated by that gap that I talked about earlier, where there are a lot of environments, especially production, that many developers may not be allowed to access, and so getting the information about those pods might involve logging into another tool, assuming you have that set up. In our case, that'd be something like Sysdig. But it's limited in the types of metrics that are useful for developers.

What you normally do without any tooling is, you just basically say, "Hey, SRE friend, could you go look at the environment for me?" And at that point, it might be too late to figure out what's going on. So that's a lot of steps that need to happen across teams, which often can be difficult.

That kind of gets into the whole question of if you have other tooling, does the developer know how to use it? If it's not used generally by developers, this gets back to the same problem of how much of this is individual motivation — "I need to really sort of dig deep and do more" — and how much is these organizational processes, and how much of this is just choosing the right tool?

How are the observability demands of developers different from those of system administrators or site reliability engineers (SREs)?

Liu: I would say that, at least from a logging perspective, we try to make a tool that can be used by both parties. It's never perfect. And so, you can only cater so much to one audience versus another, because they're fundamentally looking for different things, and they understand different contexts.

Developers don't really care about other people's apps. They care about the ones they've written and the ones they need to debug. They understand their app much more intimately than they would know about what else is running on the system and plaguing the system that their app is running on.

The system administrators and SREs are looking to make sure that the system as a whole is stable. It's more of a macro-versus-micro level. Everything needs to behave cohesively in order for this to work.

The tools that we can build can try to have different things so that everyone has a piece of data they're looking for. But it's definitely challenging to have tools that work for both parties.

logdna

SUMMARY
Dev, Ops and security teams have different goals for observability. Tools like centralized log management can help them align those goals and provide context for how each team's work affects the others.

KEY PRODUCT
LogDNA is a DevOps log management solution that gives teams control of their data and allows them to gain valuable insights from their logs.

KEY PARTNERS
IBM Cloud, Heroku, Render, PagerDuty, NGINX

GITHUB
https://github.com/logdna

What challenges does Kubernetes present in terms of observability?

Staatz: Kubernetes did a lot of cool stuff, a lot of built-in tooling [and] CLI things that are wonderful, even a dashboard you can use. That being said, tracking down metrics and logs around microservices that run on hundreds of nodes can be really hard, like, even if you are a Kubernetes wiz, and you know all the label selectors and logs.

Kubernetes is fairly new, and it will mature over time, but it can be hard to track everything down. And not everyone wants to use command-line interface tools constantly even if they are technical. Kubectl is great, but at a certain point, you want to go to a tool that's a bit more user friendly.

Lee, what was the technical issue that spurred you and Chris Nguyen to focus LogDNA on logging? What was frustrating you about other logging tools?

Liu: We built our entire backend on Elasticsearch. Something that I did not like about Elasticsearch, though it wasn't actually Elasticsearch itself, was the ingestion portion. Elastic used Logstash to ingest the logs. The problem with Logstash was you need to tell it what type of logs you're ingesting so it can do the regex filtering and to enrich data and all that stuff.

If you have one app, that's not hard, right? But if you also use Mongo or Redis, Logstash becomes a little bit hard to maintain because you have different log sources. So, we wrote our own ingestor that basically would take the data that the logs are coming in and will auto-detect what type of file it was and auto-parse that.

It doesn't matter what kind of logs I sent it, it would have handled some things generically, and some things are very specific. If it detected a web log, it would do these things for the web logs. If it looks like a MongoDB log, it'd do these things that are MongoDB-related.

Some of the other companies that we talked to about Elasticsearch, that was their struggle, the lack of auto-detection. They had to do all those things manually. And they didn't want to do it manually.

Why is cross-team empathy so important when there's not a dedicated DevOps person or DevOps team for a company? And how can logs help with that?

Staatz: Some of the challenges that I face now are not so much solving technical problems, but more along the lines of having to solve this technical problem organizationally. It's a different set of problems, because I can't simply go in and fix the product. That's not how that works. So, a lot of it is figuring out what team does this.

And as you start going into this, you get into this sort of routine of asking, "Hey, what are you guys working on?" You inadvertently develop empathy for these different teams as you try and get your work done, because, you know, maybe the work encompasses more than just one team at this point.

Production is a big, scary place that you just ship your apps to, right? And you're like, "Oh, my app goes there. And I hope it works. Magic, right?" Well, it's not magic. There's somebody on the other end that has to clean up your mess or gets you to clean up your mess if there's a problem. And I know, it's kind of a negative way of thinking about it. But one of the best things you can do is being a good Samaritan toward other people who are either developing code or maintaining your code.

And as you become a bit more experienced as a developer, you start thinking about some of the same things that infrastructure deals with every day. And so having logs and observability that cover both of those things, and can be collaborated on by both, can go a long way. Anything that can help build those bridges and can build that trust and keep people on the same page, it's great.

Joab Jackson is editor-in-chief at The New Stack. He has logged more than 25 years in infrastructure IT journalism, including stints at IDG and Government Computer News. He started in 1995 as the first internet columnist for a U.S. newspaper, the Baltimore City Paper, and later worked for a U.S. Defense Department contractor to help explain the virtues of commercializing technology originally developed for the U.S. missile defense system.

SECTION 02

Kubernetes Observability in Practice

CHAPTER 04

What and How to Log in Kubernetes

Original article published 26 August, 2020, by Franciss Espenido, LogDNA.

Editor's Note: *How can Kubernetes logs improve observability of your applications? In this chapter, we take an in-depth look at what Kubernetes offers in terms of logging capabilities, along with its limitations. This practical guide can help you identify what you need in a logging solution if you're using Kubernetes.*

No matter what your Kubernetes environment looks like, logging and monitoring are among the first major challenges you'll need to address as you begin your Kubernetes journey. Whether you have a single-node cluster running locally on your PC, a small collection of nodes that host a development environment or a large-scale, multimaster cluster hosting production applications, it's critical to be able to access monitoring data and logs in order to troubleshoot problems and optimize performance.

Given that Kubernetes consists of so many different parts, knowing where to look for that data and how to interpret it can be tricky. Indeed, Kubernetes logging and monitoring require working with multiple sources of data and multiple tools, because Kubernetes generates logs in multiple ways.

This chapter provides an overview of logging for Kubernetes, including which types of log data are available in Kubernetes and how to access that data. There are multiple access methods available for most types of Kubernetes log data. We'll explain how to assess your logging needs and devise a logging strategy suited to them.

"Given that Kubernetes consists of so many different parts, knowing where to look for data and how to interpret it can be tricky."

— Franciss Espenido, LogDNA

Most of the following information applies to any type of Kubernetes environment. However, to ground the discussion, we'll reference IBM Cloud Kubernetes Service (IKS) where relevant to explain how the tools and practices we discuss can apply to a real-world production Kubernetes service.

What Is Kubernetes?

If you're reading this chapter, you probably already know what Kubernetes is. However, it's worth briefly spelling out what it does, because understanding what Kubernetes can and can't do is the first step in extending your logging strategy to support it.

Kubernetes offers several key types of functionality:

- **Application hosting:** Kubernetes' first and foremost feature is hosting applications. Typically, those applications are hosted in containers, although it's possible to run other types of workloads, such as virtual machines, with Kubernetes.
- **Load balancing:** Kubernetes automatically distributes traffic between different application instances to optimize performance and availability.
- **Storage management:** Kubernetes can manage access to storage pools that applications use to store stateful data.
- **Self-healing:** When something goes wrong, such as an application failure,

Kubernetes attempts to fix it automatically. It doesn't always succeed, however, which is one reason why Kubernetes logging is so important.

Kubernetes does other things too, but these are the core areas of functionality it offers.

Kubernetes and Logging: It's Complicated

You'll notice that logging and monitoring aren't on the list of core Kubernetes features. That's not because Kubernetes doesn't offer any kind of logging and monitoring functionality. It does, but it's complicated.

On the one hand, Kubernetes offers some very basic functions through kubectl for checking on the status of objects in a cluster, which we'll discuss below. It also creates logs for certain types of data, and it exposes other types of data in ways that make it available for collection through third-party logging tools.

On the other hand, Kubernetes offers no full-fledged, native logging solution. Unlike, say, Amazon Web Services, which has a built-in logging solution in the form of CloudWatch, or OpenStack, which has its own comprehensive logging solution, stock Kubernetes doesn't have a complete native logging service, or even a preferred third-party logging method. Instead, it expects you to use external tools to collect and interpret log data.

That said, certain Kubernetes distributions do come with built-in logging extensions based on third-party tooling, or at least a preferred logging method that they support. For example, as we'll see below, IBM Cloud Kubernetes Service (IKS) and IBM Log Analysis with LogDNA integrate to collect Kubernetes log data and enable real-time analysis and log management using LogDNA.

In most cases, it's possible to use an alternative logging method, even on a Kubernetes distribution that has a preferred or natively integrated logging solution. However, the vendor-supported approach is usually simpler to implement.

What to Log in Kubernetes

No matter which logging option you choose, there are several log data types that you can collect in Kubernetes.

Application Logs

First and foremost are the logs from the applications that run on Kubernetes. The data stored in these logs consists of the information that your applications output as they run. Typically, this data is written to stdout (standard output) inside the container where the application runs.

We'll look at how to access this data in the "Viewing Application Logs" section that follows.

Kubernetes Cluster Logs

Several of the components that form Kubernetes itself generate their own logs:

- Kube-apiserver.
- Kube-scheduler.
- Etcd.
- Kube-proxy.
- Kubelet.

These logs are usually stored in files under the /var/log directory of the server on which the service runs. For most services, that server is the Kubernetes master node. Kubelet, however, runs on worker nodes.

If you're experiencing a cluster-level problem, as opposed to one that affects just a certain container or pod, these logs are a good place to look for insight. For example, if your applications are having trouble accessing configuration data, you could look at Etcd logs to see if the problem lies with Etcd. If a worker node is failing to come online as expected, its Kubelet log could provide insights.

Kubernetes Events

Kubernetes keeps track of what it calls "events," which can be normal changes to the state of an object in a cluster, such as a container being created or starting, or errors, such as the exhaustion of resources.

Events provide only limited context and visibility. They tell you that something happened, but not much about why it happened. They are still a useful way of getting quick information about the state of various objects within your cluster.

Kubernetes Audit Logs

Kubernetes can be configured to log requests to the Kube-apiserver. These include requests made by humans, such as requesting a list of running pods, and Kubernetes resources, such as a container requesting access to storage.

Audit logs record who or what issued the request, what the request was for and the result. If you need to troubleshoot a problem related to an API request, audit logs provide a great deal of visibility. They are also useful for detecting unusual behavior by looking for requests that are out of the ordinary, like repeated failed attempts by a user to access different resources in the cluster, which could signal attempted abuse by someone who is looking for improperly secured resources. (It could also reflect a problem with your authentication configuration or certificates.)

How to Access Kubernetes Log Data

The various types of log data described above can be accessed in different ways.

Viewing Application Logs

There are two main ways to interact with application log data. The first is to run a command like

```
kubectl logs pod-name
```

where "pod-name" is the name of the pod that hosts the application whose logs you want to access.

The kubectl method is useful for a quick look at log data. Suppose you want to store logs persistently and analyze them systematically. In that case, you're better served by using an external logging tool like LogDNA to collect and interpret the logs. The easiest way to go about this is to run a so-called sidecar container, which runs alongside the application, collects its logs and makes them available to an external logging tool.

On IKS, you can set up a LogDNA instance to perform this function for application logs, as well as for logs associated with Kubernetes itself, from the command line or by following a few steps in the IKS Web Console. For full instructions, check out the IBM Cloud documentation.

Viewing Cluster Logs

There are multiple ways of viewing cluster logs. You can simply log into the server that hosts the log you want to view (as noted earlier, that's the Kubernetes master node server in most cases) and open the individual log files directly in `less, cat` — a favorite text editor — or whatever command-line tool you prefer. Or, you can use journalctl to retrieve and display logs of a given type for you.

The most user-friendly solution is again to use an external logging tool like LogDNA. As noted earlier in this chapter, IBM Cloud's integration with LogDNA makes it easy to collect Kubernetes cluster and application logs and analyze them through a centralized interface without having to worry about the tedious process of collecting individual logs from each of your nodes through the command line.

Viewing Events

You can view Kubernetes event data through kubectl with a command like

```
kubectl get events -n default
```

where the "-n" flag specifies the namespace whose events you want to view (default in the example above). The command

```
kubectl describe my-pod
```

will show you events data for a specific pod.

Because the context of events data is limited, you may not find it very useful to log all events. However, you can always redirect the CLI output from kubectl into a log file and then analyze it with a log analysis tool.

Viewing Audit Logs

To a greater extent than is the case for other types of Kubernetes log data, the way you view and manage audit logs varies significantly depending on which Kubernetes distribution and which log collector you use to collect these logs. There is no generic and straightforward way to collect audit logs directly from kubectl.

On IKS, audit events are routed to a webhook URL. From there, you can collect logging data with IKS' native LogDNA integration by following these instructions.

How to Build a Kubernetes Logging Solution

As we saw above, there are multiple types of log data available in Kubernetes, but there are also various approaches for accessing it. Devising the Kubernetes logging strategy and toolset that work best for you requires weighing several factors:

- **Which types of Kubernetes logs do you need to collect?** Some types of data may be more or less important to you. For example, if you run simple applications that don't generate meaningful monitoring data, application logs may be less important than Kubernetes cluster logs.
- **What are your logging goals?** If you just want to view a log file quickly, kubectl (or journalctl, in certain cases) will do the job. But, if you need longer-term log management, you'll want an external log collector.
- **Do you need visualizations?** When you access a Kubernetes log file, are you

looking just for specific pieces of information, like the source of an API request? Or do you want to be able to visualize data to recognize trends or compare logs? In the latter case, an external tool or combination of tools that provide log collection and visualization is needed.

- **Do you need to aggregate logs?** Is your goal to access individual log files, or do you want to aggregate multiple logs together and analyze them collectively? You need external tools to do the latter.
- **How long do you need to retain logs?** Most of the log data stored within Kubernetes is deleted after a certain period of time, which varies depending on the type of log and your logging configurations. Therefore, if you want to hold onto historical log data for the long term, you'll need to export it to an external logging platform.

Conclusion

There is a lot of nuance surrounding logging in Kubernetes. Although Kubernetes offers some basic built-in logging and monitoring functionality, it's a far cry from a full-fledged logging solution. To get the most out of Kubernetes logging, you'll need an external log collection, analysis and management tool.

Franciss Espenido is a senior technical partnerships program manager at LogDNA, where he focuses on technical enablement for observability offerings on IBM Cloud. He previously served as a technical support engineer for the company, where he developed a deep understanding of modern logging practice and challenges in developer workflows.

CHAPTER 05

Strategies for Kubernetes Observability

Original article published 6 May, 2021, by Peter Putz, Dynatrace.

Editor's Note: *Distributed systems, particularly those in which applications deploy to multiple clouds and environments, make observability more complicated. Here are some of the best strategies for maintaining observability when your infrastructure runs on Kubernetes.*

Adopting microservices, containers and other cloud native technologies allows teams to build new digital services and capabilities faster so they can adapt to rapidly evolving business needs and continue driving customer success. Yet, maintaining visibility into these cloud native environments can be a real challenge.

Kubernetes, the popular application and innovation platform, is great at automating and managing containerized workloads and applications. However, the dynamic abstraction layer that makes it so flexible and portable across environments can lead to new types of errors that are difficult to find, troubleshoot and prevent.

Connecting the complex web of data that monitoring tools generate back to business outcomes is even more difficult. Research shows over two-thirds of chief information officers (CIOs) believe the rise of Kubernetes has resulted in too many moving parts for IT teams to manage and that they need a radically different approach to IT and cloud operations management.

Here are three key reasons why gaining observability into Kubernetes environments is so difficult, along with some ways organizations can overcome these challenges.

1. Kubernetes Is Highly Dynamic, so AIOps and Automation Are Essential

While distributed platforms such as Kubernetes enable faster innovation and better scalability, they are also highly dynamic and complex. Clusters, nodes and pods change continuously, so there's no time to manually configure and instrument monitoring capabilities. IT teams are left scrambling to gain insight into the health of their applications and keep up with the rate at which their Kubernetes environments are changing, time that could be spent launching new services that drive business success.

The only way to maintain visibility into such a dynamic environment is for teams to have the ability to automatically discover services as new ones come online and existing ones scale, and instrument them on the fly. Harnessing continuous automation assisted by AIOps enables platform and application teams to operate large-scale environments with millions of changes in real time and constantly monitor the full stack for system degradation and performance anomalies.

Not only does this give teams a full view of their Kubernetes environments, but it also enables them to better prioritize tasks by determining which technical changes will have the greatest business impact. With this insight, teams can prevent issues that affect user experience before they occur and refocus on continually optimizing services to deliver the best outcomes for the business and its customers.

2. Kubernetes Runs in Many Places, So a Full-Stack Approach Is Key

In addition to keeping track of microservices and workloads that are constantly changing, the challenge of maintaining observability becomes even more complicated when you consider that organizations often deploy Kubernetes across multiple environments.

This is because Kubernetes can run on any cloud, giving organizations the flexibility of deploying their microservices across many platforms and through managed services such as Amazon Elastic Kubernetes Service (Amazon EKS), Azure Kubernetes Service (AKS) and Google Kubernetes Engine (GKE), as well as their own on-premises servers. As such, many organizations use different monitoring tools and cloud platform metrics to manage their Kubernetes environments.

However, manually collecting and correlating the observability data from all these sources, to get the bigger picture and full context, is very time-consuming. Siloed teams with point-monitoring solutions further obstruct this and can break down cross-team collaboration.

An effective approach to observability should foster collaboration across the organization by helping to break down silos between teams. As such, it needs to unify all Kubernetes metrics, logs and traces into a single platform with a common data model. It also needs to include data from the traditional services and technology stacks that run alongside Kubernetes deployments to ensure platform and application teams have a unified view across their entire environment. This end-to-end approach to observability provides greater context that these teams can use to optimize Kubernetes workloads and applications more successfully.

3. Seeing Data in Context of the User Is Critical

It's also important to remember that observability is not just about accessing more data, it's also about how organizations can use that data to identify areas of their technology stack that need improving. Metrics, logs and traces are important, but they don't tell the whole story and indeed often limit developers and application owners by only allowing them to gain a backend perspective.

To understand the effect of Kubernetes performance on business outcomes, organizations need the ability to connect the dots between the code they push into production, the underlying cloud platform on the back end and user experience on the front end. This means they need to combine Kubernetes monitoring data with real-time business metrics such as user experience insights and conversion rates.

Teams can achieve these insights more easily with application topology mapping capabilities that automatically visualize all relationships and dependencies within a Kubernetes environment and across the wider cloud technology stack, including user experience data, in real time. Mapping dependencies vertically between clusters, hosts, pods and workloads — as well as horizontally between data centers, applications and services — allows IT teams to identify which issues are having the greatest overall impact on the business. Correlating user experience with backend performance in this way gives business leaders and digital teams the information they need to make better decisions about how to optimize their systems and where to further invest in their digital infrastructure to improve services and deliver better user experiences.

Ultimately, the most effective way to tackle the observability challenges that arise with a Kubernetes architecture is to embrace the benefits of automation and artificial intelligence (AI). Combining observability with AIOps and automation allows teams to extend their insight beyond metrics, logs and traces, and incorporate other valuable data, such as user behavior and business key performance indicators (KPIs). By rethinking their approach to Kubernetes monitoring in this way, organizations can eliminate silos so their teams spend less time troubleshooting and more time optimizing services to drive better business outcomes.

Peter Putz is a technology advocate at Dynatrace. He has more than 15 years of experience in leading international software project teams of scientists and engineers and managing the full life cycle of complex, innovative IT solutions. Prior to joining Dynatrace, he was a senior scientist at the NASA Ames Research Center.

CHAPTER 06

Why Developers Should Learn Kubernetes

Original article published 14 July, 2021, by Steve Tidwell, LogDNA.

Editor's Note: *We've mentioned the "shift left" movement in software development throughout this ebook. The idea is that developers take a bigger role in making sure their applications perform well once they are sent to production. But what does it mean to take on that greater responsibility? In this chapter, the author spells out the things developers in particular need to learn to more fully participate as a DevOps team member.*

Driven by the rise of container-based workflows for software development, Kubernetes has seen a surge in adoption in recent years as the platform of choice to deploy these containers.

Since 2016, the Cloud Native Computing Foundation has conducted a yearly survey to assess the adoption of containers and cloud native technologies by various engineering organizations. According to the CNCF 2020 survey respondents, 92% of companies are running containers in production and 83% of them are using Kubernetes as their orchestration tool.

At the same time, many organizations are also adopting DevOps and site reliability engineering (SRE) best practices to improve their applications' reliability and the time it takes to deliver new application features.

Due to this adoption, engineering teams are seeing the benefits of merging operational and development best practices. Operations teams are becoming more service- and software-oriented, and development teams are learning about the platforms and environments on which they deploy their applications.

Much of the focus in recent years has been on software development as applied to operational best practices, which has resulted in significant improvements in delivery and reliability. Sometimes though, there is a gap where development teams don't always have the operational skills needed to work with their applications outside of a development environment efficiently.

While building and shipping containerized applications may eliminate the need to maintain unique development environments, it is crucial that developers understand how containers work at runtime, especially when using an orchestrator like Kubernetes and even more so in production. It may not be necessary for developers to have a complete operations skill set, but they need to learn enough about Kubernetes and the production environment to be a reliable participant in a DevOps team.

Building Skills

Knowing about the platform that your application lives on is critical. Successful engineering organizations work hard to ensure development and operations teams avoid working in silos. Instead, they aim to collaborate earlier in the software development life cycle so that coding, building, testing and deployments are all well understood by all teams involved in the process.

Developers don't need to be experts in Kubernetes, but they should be proficient in skills that have an impact on the performance of their applications. Skills such as continuous integration/continuous delivery, deployments to production, monitoring, and understanding CPU, memory, and cluster and pod health are vital pieces of the application puzzle.

Understanding some basics about the application platform and tools that an organization uses goes a long way toward making both development and operations more efficient. Having these skills helps developers respond to incidents quickly and more effectively, thereby troubleshooting issues without escalating to another team when something goes wrong.

What Developers and Operations Teams Need to Know

Both developers and operations engineers need to understand a few things about what their peers know:

- They need to understand the peculiarities of the various services and features of their chosen cloud provider compared to other providers. This knowledge should apply whether the cloud is public, private or hybrid.
- They need to be cognizant of the financial impact of resourcing their applications and understand how to reduce costs and eliminate waste as it applies from a developer's perspective. It's straightforward to spin up a new environment and infrastructure in the cloud, and that also means it's easy to forget how quickly those costs can add up if we mismanage resources. For example, it's a good idea to consider auto-scaling policies and their effect on costs when those policies aren't set correctly.
- They need to know application performance management, especially the tools and techniques used to analyze and improve application performance.
- They need to know the proper incident response techniques to deal with incidents when they happen and escalate them when appropriate. One of the fundamental tenets of DevOps is to accept and find ways to mitigate failure, so efficiently and effectively dealing with incidents when they come up is critical.
- They need to establish feedback loops on both sides of the development and operations fence so that all teams know of any deficiencies in their tools or applications and how the developers might correct them. Shared ownership of tools and environments is an excellent way to encourage this.

So what should developers specifically learn about Kubernetes?

- How their organization's CI/CD system works, from concept to production, from code check-in to building, testing and deployment.
- Kubernetes pods and their relationship to containers.
- How applications interact with Kubernetes networking, including services, domain name system (DNS) and load balancing.
- Knowledge about commonly used tools for local testing of their deployment and for modeling how their application is deployed, such as minikube, kubectl, Helm, kind and the Kubernetes dashboard.
- Monitoring, logging and debugging clusters and containers for when things go wrong.

Of course, there's much more that both operations and development teams could understand about how they all can work together, but these are a great place to start.

Steve Tidwell has been working in the tech industry for over two decades and has done everything from end-user support to scaling a global data ingestion and analysis platform to handle data analysis for some of the largest streaming events on the web. He has worked for a number of companies, helping to improve their operations and automate their infrastructure. At the moment, Steve is plotting to take over the world with cloud-based technologies from his corner of the office.

Wrapping It Up

In our latest ebook, we've made a case for making observability a full-stack responsibility. We've offered an overview of the cloud native tools available to help improve visibility of metrics, tracing and logs, and also offered some practical strategies and tactics for improving observability in Kubernetes environments.

The key takeaway is that observability is an essential part of any DevOps team's work — to help improve application performance and resilience, to keep systems running smoothly, and most importantly, to maintain quality, reliable service to your organization's users and customers.

Cloud native applications and architecture remain complex, and as new tools emerge and systems grow ever more distributed, the challenges engineers and developers must face aren't likely to get easier. But greater observability, a shared responsibility between devs and ops, can prevent that complexity from getting in the way of robust service.

— Heather Joslyn

Disclosure

The following companies are sponsors of The New Stack:

Accurics, anynines, Armory, Aspen Mesh, Amazon Web Services (AWS), Bridgecrew, CAST AI, Check Point, CircleCI, Circonus, Citrix, Cloud Foundry Foundation, CloudBees, Cloud Native Computing Foundation (CNCF), Confluent, Cox, Cribl, DataStax, Dell Technologies, Dynatrace, Equinix, Fairwinds, GitLab, Harness, HashiCorp, Honeycomb, Hewlett Packard Enterprise (HPE), IBM Cloud, InfluxData, Infoblox, JFrog, Kasten, LaunchDarkly, Lightstep, LogDNA, MayaData, MinIO, Mirantis, MongoDB, New Relic, NGINX, Okta, Oracle, PagerDuty, Palo Alto Networks, Pure Storage, Puppet, Red Hat, Redis Labs, Rezilion, Rookout, Sentry, Shipa, Snowflake, Solo.io, Sonatype, StepZen, StorageOS, Storj, StormForge, Styra, Synopsys, Teleport, Tetrate, The Linux Foundation, Thundra, Tigera, Torq, Tricentis, TriggerMesh, Vates, VMware, Weaveworks and WSO2.

The New Stack is a wholly owned subsidiary of Insight Partners. TNS owner Insight Partners is an investor in the following companies, which are also sponsors of The New Stack: Armory, Jfrog, Kasten, MayaData, Mirantis, StormForge, Tigera and Tricentis.

thenewstack.io