O'REILLY®

Observability Engineering

Achieving Production Excellence

Early Release
Raw & Unedited

Sponsored by

honeycomb.io

Charity Majors,
Liz Fong-Jones &
George Miranda

Honeycomb

honeycomb.io

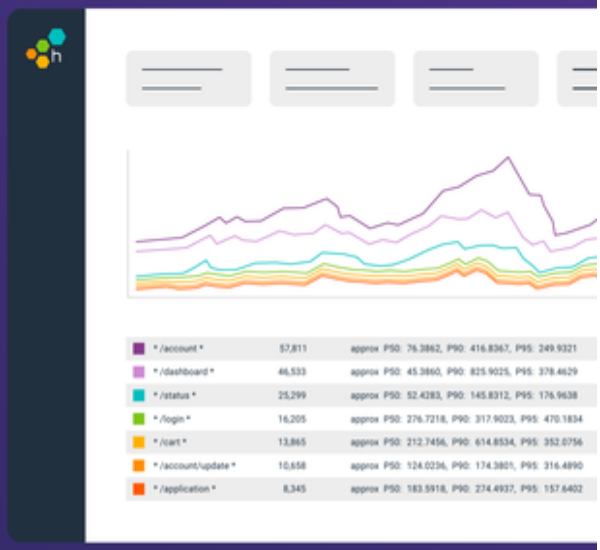
Understand Production

No matter how complex.

Solve problems faster.
Deploy without pain.
Build reliable services.

With Honeycomb, you guess less and know more. Get started with observability today!

GET STARTED



Send 20 Million Events Per Month For FREE

Unlimited users.
Unlimited services.

Unlimited data storage.
60-day retention.

No hidden fees.
EVER!

Observability Engineering

Achieving Production Excellence

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Charity Majors, Liz Fong-Jones, and George Miranda

Observability Engineering

by Charity Majors, Liz Fong-Jones, and George Miranda

Copyright © 2022 Hound Technology, Inc. All rights reserved.

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: John Devins

Development Editor: Virginia Wilson

Production Editor: Kate Galloway

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: O'Reilly Media, Inc.

January 2022: First Edition

Revision History for the Early Release

- 2020-08-17: First Release
- 2020-12-08: Second Release
- 2021-01-11: Third Release
- 2021-02-11: Fourth Release

- 2021-03-05: Fifth Release
- 2021-05-03: Sixth Release
- 2021-06-28: Seventh Release
- 2021-07-20: Eighth Release
- 2021-10-08: Ninth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492076445 for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Observability Engineering*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

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 Honeycomb. See our statement of editorial independence.

978-1-492-07637-7

Preface

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Thank you for picking up our book on observability engineering for modern software systems. Our goal is to help you understand how to develop a practice of observability within your engineering organization. This book is based on our experience as practitioners of observability, and as makers of observability tooling for users who want to improve their own observability practices.

As outspoken advocates for driving observability practices in software engineering, our hope is that this book can set a clear record of what observability means in the context of modern software systems. The term "observability" has seen quite a bit of recent uptake in the software development ecosystem. This book aims to help you separate facts from hype by providing a deep analysis of:

- What observability means in the context of software delivery and operations
- How to build the fundamental components that help you achieve observability
- The impact observability has on team dynamics

- Considerations for observability at scale
- Practical ways to build a culture of observability in your organization

Who this is for

Because observability predominantly focuses on achieving a better understanding of how software operates in the real world, this book is most useful for software engineers responsible for developing production applications. Anyone who supports the operation of software in production will also greatly benefit from the content in this book.

Additionally, managers of software delivery and operations teams who are interested in understanding how the practice of observability can benefit their organization will find value in this book, particularly in the chapters that focus on team dynamics, culture, and scale.

Anyone who helps teams deliver and operate production software and is curious about this new thing called "observability" and why people are talking about it should also find this book useful.

Why we wrote this book

Observability has become a popular topic that has quickly garnered a lot of interest and attention. With its rise in popularity, observability has been unfortunately mischaracterized as a synonym for monitoring or system telemetry. Observability is a characteristic of software systems. Further, it's a characteristic that can only be effectively utilized in production software systems when teams adopt new practices that support its ongoing development. Introducing observability into your systems is both a technical challenge and a cultural challenge.

We are particularly passionate and outspoken about the topic of observability. We are so passionate about it, that we started a company whose sole purpose is to bring the power of observability to all teams that

manage production software. We spearheaded a new category of observability tools, and other vendors have followed suit.

While we all work for Honeycomb, this book is not here to sell you on our tools. We have written this book to explain how and why we adapted the original concept of observability to managing modern software systems. You can achieve observability with different tools and in different ways. However, we believe that our dedication to advancing the practice of observability in the software industry makes us uniquely qualified to write a guide that describes, in great detail, the common challenges and effective solutions. You can apply the concepts in this book, regardless of your tool choices, to practice building production software systems with observability.

This book aims to give you a look at the various considerations, capabilities, and challenges associated with teams that practice using observability to manage their production software systems. At times, this book may provide a look at what Honeycomb does as an example of how a common challenge has been addressed. These are not intended as endorsements of Honeycomb, but rather as practical illustrations of abstract concepts. It is our goal to show you how to apply these same principles in other environments, regardless of the tools you use.

What you will learn

You will learn what observability is, how to identify an observable system, and why observability is best suited for managing modern software systems. You'll learn how observability differs from monitoring, as well as why and when a different approach is necessary. We will also cover why industry trends have helped popularize the need for observability and how that fits into emerging spaces, like the cloud-native ecosystem.

Next, we'll cover the fundamentals of observability. We'll examine why structured events are the building blocks of observable systems and how to stitch those events together into traces. Events are captured by telemetry built into your software and you will learn about open-source initiatives,

like OpenTelemetry, that help jumpstart the instrumentation process. Instrumentation exists to analyze system events, and you will learn both how that analysis works and how observability and monitoring can co-exist.

After an introduction to the technical concepts behind observability, you will learn about the social and cultural elements that often accompany the adoption of observability. Managing software in production is a team sport, and you will learn how observability should be used to help better shape team dynamics. You will learn about how observability fits into business processes, affects the software supply chain, and reveals hidden risks. And you will learn about the intersection between business objectives, engineering team needs, and user experience that is captured with Service-Level Objectives and their role in observable systems.

Observability presents further challenges for large-scale organizations. You will learn about the challenges of efficient data storage for system events, managing large quantities of data with pipelines, deciding when and how to introduce event sampling solutions, and the considerations to take into account when embarking down the path of building your own observability solution.

Finally, we look at organizational approaches to adopting a culture of observability. Beyond introducing observability to your team, you will learn practical ways to scale observability practices across an entire organization. You will learn how to identify and work with key stakeholders, use technical approaches to win allies, and how to make a business case for adopting observability practices.

Chapter 1. What is Observability?

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

In the software development industry, the subject of observability has garnered a lot of interest and is frequently found in lists of hot new topics. But with that level of surging interest in adoption, complex topics are often ripe for misunderstanding without a deeper look at the many nuances encapsulated by a simple topical label. This chapter looks at the mathematical origins of the term "observability" and examines how it has been adapted to describe characteristics of production software systems.

We also look at why the adaptation of observability for use in production software systems is necessary. Traditional practices for understanding the internal state of software applications rely on approaches that were designed for simpler legacy systems than those we typically manage today. As system architecture, infrastructure platforms, and user expectations have continued to evolve, the tools we use to reason about those components have not. Systems that only take aggregate measures into account don't provide the type of visibility needed to isolate very granular anomalies.

New methods for quickly finding needles buried in proverbial haystacks were born from necessity.

This chapter will help you understand what observability means, how to determine if a software system is observable, why observability is necessary, and how observability is used to find problems in ways that are not possible with other approaches.

The mathematical definition of observability

The term "observability" was coined by engineer Rudolf E. Kálmán in 1960. Since then, it has grown to mean many different things in different communities. Let's explore the landscape before turning to our own definition of observability for modern software systems.

In his 1960 paper,[1] Kálmán introduced a characterization he called *observability* to describe mathematical control systems. In control theory, *observability* is defined as a measure of how well internal states of a system can be inferred from knowledge of its external outputs.

This definition of observability would have you study observability and controllability as mathematical duals, along with sensors, linear algebra equations, and formal methods. This traditional definition of observability is the realm of mechanical engineers and those who manage physical systems with a specific end-state in mind. If you are looking for a mathematical and process engineering oriented textbook, you've come to the wrong place. Those books definitely exist: as any mechanical engineer or control systems engineer will inform you (usually passionately and at great length), observability has a formal meaning in traditional systems engineering terminology.

However, when adapted for use with squishier virtual software systems, that same concept opens up a radically different way of interacting with the code you write.

Applying observability to software systems

Kálmán's definition of observability can also be applied to modern software systems. When applying the concept of observability to software, we must also layer additional considerations that are specific to the software engineering domain.

For a software application to have observability, the following things must be true. You must be able to:

- Understand the inner workings of your application
- Understand any system state your application many have gotten itself into
- Understand the things above, solely by observing that with external tools
- Understand that state, no matter how extreme or unusual

A good litmus test for determining if those conditions are true is to ask yourself the following questions:

- Can you continually answer open-ended questions about the inner workings of your software to explain any anomalous values?
- Can you understand what any particular user of your software may be experiencing?
- Can you determine the things above even if you have never seen or debugged this particular state or failure before?
- Can you determine the things above even if this anomaly has never happened before?
- Can you ask arbitrary questions about your system and find answers without needing to predict what those anomalies would be in advance?

- And can you do these things without having to ship any new code to handle or describe that state (which would have implied that you needed to understand it first in order to ... understand it)?

Meeting all of the above criteria is a high bar for many software engineering organizations to clear. If you can clear that bar then you, no doubt, understand why observability has become such a popular topic for software engineering teams.

Put simply, our definition of observability for software systems is a measure of how well you can understand and explain any state your system can get into, no matter how novel or bizarre. You must be able to comparatively debug that bizarre or novel state across all dimensions of system state data, and combinations of dimensions, in an ad-hoc manner, without being required to define or predict those debugging needs in advance. If you can understand that bizarre or novel state *without shipping new code*, then you have observability.

We believe that adapting the traditional concept of observability for software systems in this way is a unique approach with additional nuances worth exploring.

For modern software systems, observability is not about the data types or inputs, nor is it about mathematical equations. It is about how people interact with and try to understand their complex systems. *Therefore, observability requires recognizing the interaction between both people and technology to understand how those complex systems work together.*

If you accept that definition, many additional questions emerge that demand answers:

- How does one gather that data and assemble it for inspection?
- What are the technical requirements for processing that data?
- What team capabilities are necessary to benefit from that data?

We will get to these questions and more throughout the course of this book. For now, let's put some additional context behind observability as it applies

to software.

The application of observability to software systems has much in common with its control theory roots. However, it is far less mathematical and much more practical. In part, that's because software engineering is a much younger and more rapidly evolving discipline than its more mature mechanical engineering predecessor. Production software systems are much less subject to formal proofs. That lack of rigor is, in part, a betrayal from the scars we, as an industry, have earned through operating the software code we write in production.

As engineers attempting to understand how to bridge the gap between theoretical practices encoded in clinical tests and the impacts of what happens when our code runs at scale, we did not go looking for a new term, definition, or functionality to describe how we got there. It was the circumstances of managing our systems and teams that led us to evolving our practices away from concepts, like monitoring, that simply no longer worked. As an industry, we need to move beyond the current gaps in tooling and terminology to get past the pain and suffering inflicted upon us by dealing with outages and a lack of more proactive solutions.

Observability is the solution to that gap.

Our complex production software systems are a mess for a variety of both technical and social reasons. So it will take both social and technical solutions to dig us out of this hole. Observability is not the entire solution to all of our software engineering problems. But it does help you clearly see what's happening in all the obscure corners of your software that you are stumbling around and trying to understand in the dark.

For instance, if you wake up in the morning and can't find your glasses, you're not going to see where you left your fork. And you certainly can't eat if you can't see your eggs on the table. So when it comes to solving practical problems in software engineering, observability is a darn good place to start.

Mischaracterizations of observability for software

Before proceeding, there's another definition of observability we need to address: the definition being promoted by SaaS developer tool vendors. These vendors are those who insist that "observability" has no special meaning whatsoever—that it is simply another synonym for telemetry, indistinguishable from monitoring. Proponents of this definition relegate observability to being another generic term for understanding how software operates. You will hear this contingent explain away observability as "three pillars" of things they can sell you that they already do today: metrics, logs, and traces.[2]

It is hard to decide which is more objectionable about this definition: its redundancy (why exactly do we *need* another synonym for telemetry?) or its epistemic confusion (why assemble a list of one data type, one anti-data type slash mess of strings, and one ... way of visualizing things in order by time?). Regardless, the logical flaw of this definition becomes clear when you realize its proponents have a vested interest in selling you the tools and mindsets built around the siloed collection and storage of data with their existing suite of metrics, logging, and tracing tools to sell. The proponents of this definition let their business models constrain how they think about future possibilities.

In fairness, we—the authors of this book—are also vendors in the observability space. However, this book is not created to sell you on our tools. We have written this book to explain how and why we adapted the original concept of observability to managing modern software systems. You can apply the concepts in this book, regardless of your tool choices, to practice building production software systems with observability. Observability is not achieved by gluing together disparate tools with marketing. There is not one specific tool you must adopt in order to get observability in your software systems. *Rather, we believe that observability requires evolving the essence of how we think about gathering the data*

needed to debug effectively. We believe that, as an industry, it is time to evolve the practices we use to manage modern software systems.

Why observability matters now

Now that we're on the same page about what observability does and doesn't mean in the context of modern software systems, let's talk about why this shift in approach matters now.

In short, the traditional approach of using metrics and monitoring of software to understand what it's doing falls drastically short. It's a fundamentally reactive approach. It may have served the industry well in the past, but modern systems demand a better approach.

For the past two or three decades, the space between hardware and their human operators has been regulated by a set of tools and conventions most call "monitoring." Practitioners have, by and large, inherited this set of tools and conventions and accepted it as the best approach for understanding that squishy virtual space between the physical and their code. And they have accepted this approach despite the knowledge that, in many cases, its inherent limitations have taken them hostage late into many sleepless nights of troubleshooting. Yet, they still grant it feelings of trust and maybe even affection because that captor is the best they have.

With monitoring, software developers can't fully see their systems. They squint at the system. They try, in vain, to size them up and try to predict all the myriad ways it could possibly fail. Then they watch—they monitor—for those known failure modes. They set performance thresholds and arbitrarily pronounce them "good" or "bad." Then they deploy a small robot army to check and re-check those thresholds on their behalf. They collect their findings into dashboards. They then organize themselves around those robots into teams, rotations, and escalations. When those robots tell them it's bad, they mobilize. Then, over time, they tend to those arbitrary thresholds like gardeners: pruning, tweaking, and fussing over the noisy signals they told the robots to stream them.

Is this really the best way?

For decades, that's how developers and operators have done it. Monitoring has been the de facto approach for so long, that they tend to think of it as *the only way* of understanding their systems, instead of just *one way* to understand them. Monitoring is such a default practice that it has become mostly *invisible*. They don't question whether *we should* do it, but *how*.

The practice of monitoring is grounded in many unspoken assumptions about systems (which we'll detail below). But as systems continue to evolve—as they become more abstract, more complex, and as their underlying components begin to matter less and less—those assumptions become less true. As developers and operators continue to adopt modern approaches to deploying software systems (SaaS dependencies, container orchestration platforms, distributed systems, etc.), the cracks in those assumptions become more evident.

As those assumptions become evident, more people find themselves slamming into the wall of inherent limitations and realizing that monitoring approaches simply do not work for the new modern worlds. With modern approaches, it has become evident that traditional monitoring practices are catastrophically ineffective for understanding systems. The assumptions of metrics and monitoring are now falling short.

To understand why they fail, it helps to examine their history and intended context.

Why are metrics and monitoring not enough?

In 1988, by way of SNMPv1 (Simple Network Management Protocol), the foundational substrate of monitoring was born: the metric. A metric is a single number, with tags optionally appended for grouping and searching those numbers. Metrics are, by their very nature, disposable and cheap. They have a predictable storage footprint. They're easy to aggregate along regular time series buckets. And, thus, the metric became the base unit for a

generation or two of telemetry—the data we gather from remote endpoints for automatic transmission to monitoring systems.

Many sophisticated apparatuses have been built atop the metric: time series databases, statistical analyses, graphing libraries, fancy dashboards, on-call rotations, ops teams, escalation policies, and a plethora of ways to digest and respond to what that small army of robots is telling you.

But there's a practical limit to where this model serves you.

If you've crossed that limit, realize the change is abrupt. Monitoring approaches seem to continually evolve, right up until they cease to work for you. What worked well enough last month simply does not work anymore. The inherent limitation becomes evident when you reach a tipping point in terms of complexity.

It's hard to quantify exactly when that tipping point is reached. Basically, what happens is that the sheer number of possible states the system could get itself into outstrips your team's ability to pattern-match based on prior outages. Your team can no longer guess which dashboards should be created to display any innumerable amount of failure modes. When that occurs, that's when the inherent assumptions about monitoring become clear. They cease to be hidden and they very much become the bane of your team's ability to understand what's happening.

The hidden assumptions of metrics based systems are that:

- Your application is monolithic in nature
- There is one stateful data store ("the database")
- Many low-level systems metrics are available and relevant (e.g., resident memory, CPU load average)
- The application runs on VMs or bare metal, giving you full access to system metrics
- You have a fairly static set of hosts to monitor
- Engineers examine systems for problems only after problems occur

- Dashboards and telemetry exist to serve the needs of operations engineers
- Monitoring examines “black-box” applications that are inaccessible
- Monitoring solely serves the purposes of operations
- The focus of monitoring is uptime and failure prevention
- Examination of correlation occurs across a limited (or small) number of dimensions

When compared to the reality of modern systems, it becomes clear that traditional monitoring approaches fall short in several ways. The reality of modern systems is that:

- There are many, many services to manage
- There is polyglot persistence (i.e., “many” databases)
- Infrastructure is extremely dynamic, with capacity flicking in and out of existence elastically
- Many far-flung and loosely coupled services are managed, many of which are not directly under your control
- Engineers actively check to see how changes to production code behave, in order to catch tiny issues early, before they create user impact
- Automatic instrumentation is insufficient for understanding what is happening in complex systems
- Software engineers own their own code in production and they are incentivized to proactively instrument their code and inspect the performance of new changes as they’re deployed
- The focus of reliability is instead how much tolerance to allow for constant and continuous failures, while building resiliency to user-

experienced failures by utilizing constructs like error budget, quality of service, and user experience

- Examination of correlation occurs across a virtually unlimited number of dimensions

The last point is important, because it describes the breakdown that occurs between the limits of correlated knowledge that one human can be reasonably expected to think about and the reality of modern system architectures. There are so many possible dimensions involved in discovering the underlying correlations behind performance issues that no human brain, and in fact no schema, can possibly contain them.

With observability, the ability to compare many high-cardinality dimensions, and many combinations of high-cardinality dimensions, becomes a critical component of being able to discover otherwise hidden issues buried in complex system architectures.

Debugging with metrics vs. observability

Beyond that tipping point of system complexity, it's no longer possible to fit a model of the system into your mental cache anymore. By the time you try to reason your way through its various components, your mental model is already likely to be out of date. As an engineer, you are probably used to debugging via intuition. To get to the source of a problem, it's likely you feel your way along a hunch or use a fleeting reminder of some outage long past to guide your investigation. However, the skills that served you well in the past are no longer applicable in this world. The intuitive approach only works as long as most of the problems you encounter are variations of the same few predictable themes you've encountered in the past.

Similarly, the metrics-based approach of monitoring relies on having encountered known failure modes in the past. Monitoring helps detect when systems are over or under predictable thresholds that someone has previously deemed means they're experiencing an anomaly. But what happens when you *don't know that type of anomaly is even possible*?

Historically, the majority of problems that software engineers encounter have been variants of somewhat predictable failure modes. Perhaps it wasn't known that your software could fail quite in the manner that it did, but if you reasoned about the situation and its components, it wasn't a logical leap to discover a novel bug or failure mode. It is a rare occasion for most software developers to encounter truly unpredictable leaps of logic because they haven't typically had to deal with the type of complexity that makes it commonplace (until now, most of the complexity for developers has been in the app bundle).

> *"Every application has an inherent amount of irreducible complexity. The only question is: Who will have to deal with it—the user, the application developer, or the platform developer?"*
>
> —- Larry Tesler

Modern distributed systems architectures notoriously fail in novel ways that no one is able to predict and that no one has experienced before. This condition happens often enough that an entire set of assertions has been coined about the false assumptions that programmers new to distributed computing often make.[3] Modern distributed systems are also made accessible to application developers as *abstracted infrastructure platforms*. As users of those platforms, application developers are now left to deal with an inherent amount of irreducible complexity that has landed squarely on their plates.

The previously submerged complexity of application code subroutines that interacted with each other inside the hidden random access memory internals of one physical machine have now surfaced as service requests between hosts. That newly exposed complexity then hops across multiple services, traversing across an unpredictable network many times over the course of a single function. When modern architectures started to favor decomposing monoliths into microservices, software engineers lost the ability to step through their code with traditional debuggers. Meanwhile, their tools have yet to come to grips with that seismic shift.

Examples of this seismic shift can be seen with the trend toward containerization, the rise of container-orchestration platforms, the shift to microservices, the common use of polyglot persistence, the introduction of the service mesh, the popularity of ephemeral auto-scaling instances, serverless computing, lambda functions, and any other myriad SaaS applications in a software developer's typical toolset. Stringing these various tools together into a modern system architecture means that a request may perform 20-30 hops after it reaches the edge of things you control (and likely multiply that by a factor of two if it includes database queries).

In modern cloud-native systems, the hardest thing about debugging is no longer understanding how the code runs but *finding where in your system* the code with the problem even lives. Good luck looking at a dashboard or a service map to see which node or service is slow because distributed requests in these systems often loop back on themselves. Finding performance bottlenecks in these systems is incredibly challenging. When something gets slow, *everything gets slow*. Even more challenging, because cloud-native systems typically operate as platforms, the code may live in a part of the system that this team doesn't even control.

In a modern world, debugging with metrics requires you to connect dozens of disconnected metrics that were recorded over the course of executing any one particular request, across any number of services or machines, to infer what might have occurred over the various hops needed for its fulfillment. The helpfulness of those dozens of clues depends entirely upon whether someone was able to predict, in advance, if that measurement was over or under the threshold that meant this action contributed to creating a previously unknown anomalous failure mode that had never been previously encountered.

By contrast, debugging with observability starts with a very different substrate: a deep context of what was happening when this action occurred. Debugging with observability is about preserving as much of the context around any given request as possible, so that you can reconstruct the

environment and circumstances that triggered the bug that led to a novel failure mode.

The role of cardinality

In the context of databases, cardinality refers to the uniqueness of data values contained in a set. Low cardinality means that a column has a lot of duplicate values in its set. High cardinality means that the column contains a large percentage of completely unique values. A column containing a single value will always be the lowest possible cardinality. A column containing unique IDs will always be the highest possible cardinality.

For example, if you had a collection of a hundred million user records, you can assume that userID numbers will have the highest possible cardinality. First name and last name will be high cardinality, though lower than userID because some names repeat. A field like gender would be fairly low-cardinality given the non-binary, but finite, choices it could have. A field like species would be the lowest possible cardinality, presuming all of your users are humans.

Cardinality matters for observability, because *high-cardinality information is the most useful data for debugging* or understanding a system. Consider the usefulness of sorting by fields like user IDs, shopping cart IDs, request IDs, or any other myriad IDs like instances, container, build number, spans, and so forth. Being able to query against unique IDs is the best way to pinpoint individual needles in any given haystack.

Unfortunately, metrics-based tooling systems can only deal with low-cardinality dimensions at any reasonable scale. Even if you only have merely hundreds of hosts to compare, with metrics-based systems, you can't use hostname as an identifying tag without hitting the limits of your cardinality key-space. These inherent limitations place unintended restrictions on the ways that data can be interrogated. When debugging with metrics, for every question you may want to ask of your data, you have to decide—in advance, before a bug occurs—what you need to inquire about so that its value can be recorded when that metric is written.

That inherent limitation has two big implications. First, if during the course of investigation, you decide that an additional question must be asked to discover the source of a potential problem, that cannot be done after the fact. You must first go set up the metrics that might answer that question and wait for the problem to happen again. Second, because it requires another set of metrics to answer that additional question, most metrics-based tooling vendors will charge you for recording that data. Your cost increases linearly with every new way you decide to interrogate your data to find hidden issues you could not have possibly predicted in advance.

Debugging with observability

Conversely, observability tools encourage developers to gather rich telemetry for every possible event that could occur, passing along the full context of any given request and storing it for possible use at some point down the line. Observability tools are specifically designed to query against high-cardinality data. What that means for debugging is that you can interrogate your event data in any number of arbitrary ways. You can ask new questions that you did not need to predict in advance and find answers to those questions or clues that will lead you to ask the next question. You repeat that pattern again and again until you find the needle in the proverbial haystack that you're looking for.

The ability to interrogate your event data in arbitrary ways means that you can ask any question about your system and inspect its corresponding internal state. That means you can investigate and eventually understand any state your system has gotten itself into—even if you have never seen that state before—without needing to predict what those states might be in advance.

Again, observability means that you can understand and explain any state your system can get into—no matter how novel or bizarre—*without shipping new code*.

The reason monitoring worked so well for so long is that systems tended to be simple enough that engineers could reason about exactly where they

might need to look for problems and how those problems might present themselves. For example, it's relatively simple to connect the dots that when sockets fill up, CPU would overload, and the solution is to add more capacity by scaling application node instances, or by tuning your database, or so forth. Engineers could, by and large, predict the majority of possible failure states up front and discover the rest the hard way once their applications were running in production.

However, monitoring creates a fundamentally reactive approach to system management. You can catch failure conditions that you predicted and knew to check for. If you know to expect it, you check for it. For every condition you don't know to look for, you have to see it first, deal with the unpleasant surprise, investigate it to the best of your abilities, possibly reach a dead end that requires you to see that same condition multiple times before properly diagnosing it, and then you can develop a check for it. In that model, engineers are perversely incentivized to have a strong aversion to situations that could cause unpredictable failures. This is partially why some teams are terrified of deploying new code (more on that topic later).

Observability is for modern systems

A production software system is observable to the extent that you can understand new internal system states without having to make arbitrary guesses, predict those failure modes in advance, or ship new code to understand that state. In this way, we extend the control theory concept of observability to the field of software engineering.

In its software engineering context, observability does provide benefits for more traditional architectures or monolithic systems. For example, it could certainly save teams from having to discover unpredictable failure modes in production the hard way. But the benefits of observability are absolutely critical when using modern distributed systems architectures.

In distributed systems, the ratio of somewhat predictable failure modes to novel and never-before-seen failure modes is heavily weighted toward the bizarre and unpredictable. Those unpredictable failure modes happen so

commonly, and repeat rarely enough, that they outpace the ability for most teams to set up the appropriate and relevant enough monitoring dashboards to easily show that state to the engineering teams responsible for ensuring the continuous uptime, reliability, and acceptable performance of their production applications.

This book is written with these types of modern systems in mind. Any system consisting of many components that are loosely coupled, dynamic in nature, and difficult to reason about are a good fit for realizing the benefits of observability vs. traditional management approaches. If you manage production software systems that fit that description, this book describes what observability can mean for you, your team, your customers, and your business. We will also focus on the human factors necessary to develop a practice of observability in key areas of your engineering processes.

Conclusion

Although the term *observability* has been defined for decades, its application to software systems is a new adaptation that brings with it several new considerations and characteristics. Compared to their simpler early counterparts, modern systems have introduced such additional complexity that failures are harder than ever to predict, detect, and troubleshoot.

To mitigate that complexity, engineering teams must now be able to constantly gather telemetry in flexible ways that allow them to debug issues without first needing to predict how failures may occur. Observability enables engineers to slice and dice that telemetry data in flexible ways that allow them to get to the root of any issues that occur in unparalleled ways.

Next, we'll examine how observability differs from the traditional systems monitoring approach.

---

1 "On the General Theory of Control Systems", R.E. Kálmán, 1960

2 Sometimes these claims include time spans to signify “discrete occurrences” as a fourth pillar of a generic synonym for telemetry.

3 See: *The Fallacies of Distributed Computing*

Chapter 2. How Observability Differs from Monitoring

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Monitoring systems collect, aggregate, and analyze metrics to sift through known patterns that indicate troubling trends are occurring. In the previous chapter, we covered the origins and common use of metrics for debugging. Now we'll examine how metrics are configured to trigger alerts that notify human operators when further investigation is needed. Dashboards—metrics data trends adapted for visual consumption by humans—are also used to complete the metrics-based troubleshooting toolset. These monitoring systems work well for identifying previously encountered known failures.

Observability tools take a fundamentally different exploratory approach to finding problems. In this chapter, we focus on understanding the limitations of monitoring-based troubleshooting. First, we unpack how monitoring tools are used within the context of troubleshooting software performance issues in production. Then we examine the behavioral ramifications

institutionalized by using monitoring-based approaches. Finally, we use that examination to draw distinctions between monitoring and observability.

How monitoring data is used

There are two main consumers for monitoring data: machines and humans.

Machines use monitoring data to make decisions about whether a detected condition should trigger an alert or if a recovery should be declared. A metric is a numerical representation of system state over the particular interval of time when it was recorded. Similar to looking at a physical gauge, at a glance, a metric might be able to convey whether a particular resource is over or under utilized at a particular moment in time. For example, CPU utilization might be at 90% right now.

But is that behavior changing? Is the measure shown on the gauge going up or going down? Metrics are typically more useful in aggregate. Understanding the trending values of metrics over time is what provides insights into system behaviors that affect software performance. Monitoring systems collect, aggregate, and analyze metrics to sift through known patterns that indicate trends their humans want to know about.

If CPU utilization continues to stay over 90% for the next two minutes, someone may have decided that's a condition they want to be alerted about. When monitoring systems detect that trend, an alert is sent. Similarly, if CPU utilization drops below 90% for a preconfigured timespan, the monitoring system will determine that the error condition for the triggered alert no longer applies and it, therefore, declares the system recovered.

It's a rudimentary system, yet so many of our troubleshooting capabilities rely on it.

How humans use that same data is a bit more interesting. Those numerical measurements are fed into time series databases, and a graphical interface uses that database to source graphical representations of data trends. Those graphs can be collected and assembled into progressively more complicated combinations, known as dashboards.

Static dashboards are commonly assembled one per service, and they're a useful starting point for an engineer to begin understanding particular aspects of the underlying system. This is what dashboards were originally intended for: to provide an overview into how a set of metrics are tracking and to surface noteworthy trends. However, dashboards are a poor choice for discovering new problems with debugging.

When dashboards were first built, there weren't many system metrics to worry about. So it was relatively easy to build a dashboard that showed the critical data anyone should know about for any given service. In modern times, storage is cheap, processing is powerful, and the data we can collect about a system seems virtually limitless. Modern services typically collect so many metrics that it's impossible to fit them all into the same dashboard. Yet that doesn't stop many engineering teams from trying to fit those all into a singular view. After all, that's the promise of the dashboard!

To make everything fit in a dashboard, metrics are often aggregated and averaged. These aggregate values no longer provide meaningful visibility into what's happening in their corresponding underlying systems. To mitigate that problem, some vendors have added filters and drill-downs to their dashboarding interfaces that allow you to dive deeper and narrow down visualizations in ways that improve their function as a debugging tool.

However, your ability to troubleshoot effectively using dashboards is limited by your ability to pre-declare conditions that describe what you might be looking for. In advance, you will need to specify that you want the ability to break down values along a certain small set of dimensions. That has to be done in advance so that your dashboarding tool can create the indexes necessary on those columns to allow the type of analysis you want. That indexing is also strictly bounded by groups of data with high cardinality.

Requiring the foresight to define necessary conditions puts the onus of data discovery on the user. Any efforts to discover new system insights

throughout the course of debugging are limited by conditions you would have had to predict prior to starting your investigation.

In that respect, using metrics to surface new system insights is an inherently reactive approach. Yet, as a whole, the software industry has seemingly been conditioned to rely on dashboards for debugging despite these limitations. That reactiveness is a logical consequence of metrics being the best troubleshooting tool the industry had available for many years. Because we're so accustomed to that limitation as the default way troubleshooting is done, the impact that has on our troubleshooting behavior may not be immediately clear at a glance.

Troubleshooting behaviors when using dashboards

The scenario below should be familiar to engineers responsible for managing production services. If you're such an engineer, put yourself in these shoes and use that perspective to examine the assumptions that you also make in your own work environment. If you're not an engineer that typically manages services in production, examine the following scenario for the types of limitations described in the previous section.

It's a new morning and your work day is just getting started. You walk over to your desk and one of the first things you do is glance at a collection of readily displayed dashboards. Your dashboarding system aspires to be a "single pane of glass," behind which you can quickly see every aspect of your application systems, its various components, and their health statuses. You also have a few dashboards that act as high-level summaries that convey important top-line business metrics so you can see, for example, whether your app is breaking any new traffic records, if any of your apps have been removed from the App Store overnight, and a variety of other critical conditions that require immediate attention.

You glance at these dashboards to seek familiar conditions and reassure yourself that you are free to start your day without the distraction of firefighting production emergencies. The dashboard has a collection of two to three dozen graphs displayed. You don't know what many of those

graphs actually show. Yet, over time, you've developed confidence in the predictive powers that these graphs give you. For example, if the graph at the bottom of this screen turns red, you should drop whatever you're doing and immediately start investigating before things get worse.

Perhaps you don't know what all the graphs actually measure, but they pretty reliably help you predict where the problems are happening in the production service you're intimately familiar with. When the graphs turn a certain way, you almost acquire a psychic power of prediction. If the left corner of the graph at the top has a dip while the right bottom is growing steadily, then that means there's a problem with your message queues. If the box in the center is spiking every five minutes and the background is a few shades redder than normal, then there's a database query that's acting up.

Just then, as you're glancing through the graphs, you notice that there's a problem with your caching layer. There's nothing on the dashboard that clearly says, "your primary caching server is getting hot." But you've gotten to know your system so well that by deciphering patterns on a screen, you can immediately leap into action to do things that are not at all clearly stated by the data available. You've seen this type of issue before and, based on those past scars, you know that this particular combination of measures indicates a caching problem.

Seeing that pattern, you quickly pull up the dashboard for the caching component of your system to confirm your suspicion. Your suspicion is confirmed and you jump right into fixing the problem. Similarly, there are more than a handful of patterns you can do this with. Over time, you've learned to divine the source of problems by reading the tea leaves of your particular production service.

The limitations of troubleshooting by intuition

That troubleshooting approach is one many engineers are intimately familiar with. Ask yourself, just how much intuition do you rely on when hopping around various components of your system throughout the course of investigating problems? Typically, as an industry, we value that intuition,

and it has provided us with much value throughout the years. Now ask yourself: if you were placed in front of those same dashboarding tools, but with an entirely different application, written in a different language, with a different architecture, could you divine those same answers? When the lower left corner turns blue, would you know what you were supposed to do or if it was even necessary to take action?

Clearly, the answer to that question is no. The expressions of various system problems as seen through dashboards is quite different from app stack to app stack. Yet, as an industry, this is the primary way of interacting with systems that many of us have ever experienced. Historically, engineers have relied on static dashboards that are densely populated with data that is an interpretive layer or two away from the data needed to make proper diagnoses when things go wrong. But we start seeing the limitations of their usefulness when we discover novel problems.

Example 1: Timestamp correlation

An engineer adds an index and wants to know if that achieved her goal of faster query times. She also wants to know whether there were any other unexpected consequences. She might want to ask the following questions:

- Is a particular query (which is known to be a pain point) scanning fewer rows than before?
- How often is the new index getting chosen by the query planner, and for which queries?
- Are write latencies up overall, on average, or at the 95th/99th percentiles?
- Are queries faster or slower when they use this index than their previous query plan?
- What other indexes are also used along with the new index (assuming index intersection)?

- Has this index made any other indexes obsolete, so we can drop those and reclaim some write capacity?

Those are just a few example questions, and she can ask more. However, what she has available is a dashboard and graphs for CPU load average, memory usage, index counters, and lots of other internal statistics for the host and running database. She cannot slice and dice or break down by user, query, destination or source IP, or anything like that. All she can do is eyeball the broad changes and hazard a sophisticated guess, mostly based on timestamps.

Example 2: Not drilling down

An engineer discovers a bug that inadvertently expires data and wants to know if it's affecting *all* the users or just some shards. Instead, what they have available in a dashboard is the ability to see disk space dropping suspiciously fast … on just one data shard. They briskly assume the problem is confined to that shard and move on, not realizing that disk space appeared to be holding steady on the other shard, thanks to a simultaneous import operation.

Example 3: Tool-hopping

An engineer sees a spike in errors at a particular time. They start paging through dashboards, looking for spikes in other metrics at the same time, and they find some, but they can't tell which are the cause of the error and which are the effects. So they jump over into their logging tool and start grepping for errors. Once they find the request ID of an error, they turn over to their tracing tool and copy-paste the error ID into the tracing tool. (If that request isn't traced, they repeat this over and over until they catch one that is.)

These monitoring tools can get better at detecting finer-grained problems over time—if you have a robust tradition of always running post-mortems after outages and adding custom metrics where possible in response.

Typically, the way this happens is that the on-call engineer figures it out or arrives at a reasonable hypothesis, and also figures out exactly which metric(s) would answer the question if it exists. They ship a change to create that metric and begin gathering it. Of course, it's too late now to see if your last change had the impact you're guessing it did—you can't go back in time and capture that custom metric a second time unless you can replay the whole exact scenario—but if it happens again, the story goes, next time you'll know for sure.

The engineers in the examples above might go back and add custom metrics for each query family, for expiration rates per collection, for error rates per shards, etc. (They might go nuts and add custom metrics for every single query family's lock usage, hits for each index, buckets for execution times, etc.—and then find out they doubled their entire monitoring budget the next billing period.)

Traditional monitoring is fundamentally reactive

That approach is entirely reactive, yet many teams accept this as the normal state of operations—that is simply how troubleshooting is done. At best, it's a way of playing whack-a-mole with critical telemetry, always playing catchup after the fact. It's also quite costly, since metrics tools tend to price out custom metrics in a way that scales up linearly with each one. Many teams enthusiastically go all-in on custom metrics, then keel over when they see the bill, and end up going back to prune the majority of them.

Is your team one of those teams? To help determine that, watch for some of these indicators as you perform your work maintaining your production service throughout the week:

- When issues occur in production, are you determining where you need to investigate based on an actual visible trail of system information breadcrumbs? Or are you following your intuition of where those problems are? Are you looking in the place where you know you found the answer last time?

- Are you relying on your expert familiarity of this system and its past problems? When you use a troubleshooting tool to investigate a problem, are you exploratively looking for clues? Or are you trying to confirm a guess? For example, if latency is slow across the board and you have dozens of databases and queues that could be producing it, are you able to use data to determine where the latency is coming from? Or do you guess it must be your mysql database, per usual, and then go check your mysql graphs to confirm your hunch?
- Are your troubleshooting tools giving you precise answers to your questions and leading you to direct answers? Or are you performing translations based on system familiarity to arrive at the answer you actually need?
- How many times are you leaping from tool to tool, attempting to correlate patterns between observations, relying on yourself to carry the context between disparate sources?
- Most of all, is the best debugger on your team always the person who has been there the longest? This is a dead giveaway that most of your knowledge about your system comes not from a democratized method like a tool, but through personal hands-on experience alone.

Guesses aren't good enough. Correlation is not causation. There is often a vast disconnect between the specific questions you want to ask and dashboards available to you. You shouldn't have to make a leap of faith to connect cause and effect.

It gets even worse when you consider the gravity-warping impact confirmation bias can have. With a system like this, you can't find what you don't know to look for. You can't ask the questions you didn't predict you might need to ask, far in advance.

Historically, engineers have had to stitch together answers from various data sources, along with the intuitions they've developed about their

systems in order to diagnose issues. As an industry, we have accepted that as the normal state of operations. But as systems have grown in complexity, beyond the ability for any one person or team to intuitively understand its various moving parts, the need to grow beyond this reactive and limiting approach becomes clear.

How observability is different

As we've seen in the section above, monitoring is a reactive approach that is best suited for detecting known problems and previously identified patterns—the model centers around the concept of alerts and outages. Conversely, observability is best suited for interrogating systems to explicitly discover the source of any problem, along any dimension or combination of dimensions, without needing to first predict where and how that problem might be happening— this model centers around questioning and understanding.

Let's comparatively examine the differences between monitoring and observability across three axes: tribal knowledge, finding hidden issues, and confidence diagnosing production. We will give you more in-depth examples of how and why these differences occur in upcoming chapters. For now, we will make these comparative differences at a high level.

Tribal knowledge is unwritten information that may be known to some, but is not commonly known by others within an organization. With monitoring-based approaches, teams often orient themselves around the idea that seniority is the key to knowledge: the engineer who has been on the team longest is typically the best debugger on the team and the debugger of last resort. When debugging is tightly coupled to an individual's experience deciphering previously known patterns, that predilection should be unsurprising.

Conversely, teams that practice observability are inclined in a radically different direction. With observability tools, the best debugger on the team is typically the engineer who is most curious. Engineers practicing observability have the ability to interrogate their systems by asking

exploratory questions, using the answers discovered to lead them toward making further open-ended inquiries. Rather than valuing intimate knowledge of one particular system to provide an investigative hunch, observability instead rewards skilled investigative abilities that translate across different systems.

The impact of that shift becomes most evident when it comes to finding issues that are buried deep within complex systems.

A reactive monitoring-based approach that rewards intuition and hunches is also prone to obscuring the real source of issues with confirmation bias. When issues are detected, they're diagnosed based on how similar their behavioral pattern appears to previously known problems. That can lead to treating symptoms of a problem without ever getting to the actual source. Engineers guess at what might be occurring, jump to confirm that guess, and alleviate the symptom without ever fully investigating why it was happening in the first place. Worse, by introducing a fix for the symptom instead of its cause, teams will now have two problems to contend with instead of one.

Rather than leaning on expert foreknowledge, observability allows engineers to treat every investigation as new. When issues are detected, even if the triggering conditions appear similar to past problems, an engineer should be able to put one metaphorical foot in front of the other to follow the clues provided by breadcrumbs of system information. You can follow the data toward determining the correct answer—every time, step by step. That methodical approach means that any engineer can diagnose any issue without needing a vast level of system familiarity to reason about impossibly complex systems to intuitively divine a course of action. Further, the objectivity afforded by that methodology means engineers can get to the source of the specific problem they're trying to solve rather than treating the symptoms of similar problems in the past.

The shift toward objective and methodological investigation also serves to increase the confidence of entire teams to diagnose production issues.

In monitoring-based systems, humans are responsible for leaping from tool to tool and correlating observations between them. They must mentally carry the context when moving from looking at dashboards to reading logs. Then once more when moving from logs to looking at a trace, and back again. This context switching is error-prone, exhausting, and often impossible given the inherent incompatibilities and inconsistencies encountered when dealing with different sources of data and truth. For example, if you're responsible for drawing correlations between units like TCP/IP packets and HTTP errors experienced by your app, or resource starvation errors and high memory eviction rates, your investigation likely has such a high degree of conversion error built in that it might be equally effective to take a random guess.

Observability tools pull the context from different sources of information like logs, metrics, events, and traces into one central context where investigators can easily switch between the necessary views to find definitive answers. Engineers should be able to move through an investigation steadily and confidently, without the distraction of constant context switching. Further, by holding that context within one tool, implicit understandings that were often stitched together by experience and tribal knowledge instead become explicit data about your system. Observability allows critical knowledge to move out of the minds of the most experienced engineers and into a shared reality that can be explored by any engineer as needed.

You will see how these benefits are unlocked as we explore more detailed features of observability tools throughout this book.

Conclusion

The monitoring-based approach of using metrics and dashboards in tandem with expert knowledge to determine the source of problems in production is a practice that is very prevalent in the software industry. In the era of elementary application architectures with limited data collection, an investigative practice that leads with the experience and intuition of humans

to detect system issues made sense given the simplicity of legacy systems. With modern applications, the complexity and scale of their underlying systems quickly make that approach untenable and illustrate the need for a new approach.

Observability tools differ from traditional monitoring tools because they are designed to enable engineers to investigate any system, no matter how complex, without leaning on experience or intimate system knowledge to generate a hunch. With observability tools, engineers can approach the investigation of any problem methodically and objectively. By interrogating their systems in an open-ended manner, engineers practicing observability can find the source of deeply hidden problems and confidently diagnose issues in production, regardless of their prior exposure to any given system.

Next, let’s look at how you can make the shift from a monitoring-based troubleshooting approach to adopting a practice of observability.

Chapter 3. Lessons from Scaling Without Observability

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

So far, we've defined observability and how it differs from traditional monitoring. We've covered some of the limitations that traditional monitoring tools have when managing modern distributed systems and how observability solves them. But there's an evolutionary gap between the traditional and modern world. What happens when trying to scale modern systems without observability?

In this chapter, we look at a real example of slamming into the limitations of traditional monitoring and architectures, along with why different approaches are needed when scaling applications. Co-author Charity Majors shares her firsthand account on lessons learned from scaling without observability at her former company, Parse Inc. This story is told from her perspective.

An introduction to Parse

Hello, dear reader. I'm Charity, and I've been on call since I was 17 years old. Back then, I was racking servers and writing shell scripts at the University of Idaho. I remember the birth and spread of many notable monitoring systems: Big Brother, Nagios, RRDtool and Cacti, Ganglia, Zabbix, and Prometheus. I've used most—not quite all—of them. They were incredibly useful in their time. Once I got a handle on time series databases and their interfaces, every system problem suddenly looked like a nail for the time series hammer: set thresholds, monitor, rinse, and repeat.

During my career, my niche as an engineer has been coming in as the first—or one of the very early first—infrastructure engineer who was joining an existing team of software engineers in order to help mature their product to production readiness. I've made decisions around how best to understand what's happening in production systems many, many times.

That's what I did at Parse. Parse was a mobile backend as a service (MBaaS) platform, providing mobile app developers a way to link their apps to backend cloud storage systems and APIs to backend systems that enabled features like user management, push notifications, and integration with social networking services. In 2012, when I joined the team, Parse was still in beta. At that time, they were using a bit of AWS CloudWatch and, somehow, they were being alerted by five different systems. I switched us over to using Icinga/Nagios and Ganglia because those were the tools I knew best.

Parse was an interesting place to work because it was so ahead of its time in many ways (we would go on to be acquired by Facebook in 2013). We had a microservice architecture before we had the name "microservices" and long before that pattern became a movement. We were using MongoDB as our datastore and very much growing up alongside it: when we started, it was version 2.0 with a single lock per replica set. We were developing with Ruby on Rails and we had to monkeypatch Rails to support multiple database shards.

We had complex multi-tenant architectures with shared tenancy pools. In the early stages, we were optimizing for development speed, full stop.

I want to pause here to stress that optimizing for development speed was *the right thing to do*. With that decision, we made many early choices that we later had to undo and redo. But most startups don't fail because they made the wrong tooling choices. Let's be clear: most startups do fail. They fail because there's no demand for their product, or because they can't achieve product/market fit, or because customers don't love what they built, or any number of reasons where time is of the essence. Choosing a stack that used MongoDB and Ruby on Rails enabled us to get to market quickly enough that we delighted customers and they wanted a lot more of what we were selling.

Around the time Facebook acquired Parse, we were hosting over 60,000 mobile apps. Two and a half years later, when I left Facebook, we were hosting over a million mobile apps. But even at that earlier stage, the cracks were already starting to show.

Parse officially launched a couple of months after I joined. Our traffic doubled, then doubled, and then doubled again. We were the darling of Hacker News and every time a post about us showed up there, we'd get a spike of new signups. In August of 2012, for the first time, one of our hosted apps moved into a top 10 spot on the iTunes store. The app was for a death metal band from Norway. They used it to livestream broadcasts during their (local time) evenings (for Parse, it was our crack of dawn). Every time they live streamed a broadcast, Parse went down in a matter of seconds.

Then we had a scaling problem.

Scaling at Parse

At first, it was very difficult to figure out what was happening to our services. It took us about a week just to isolate the actual source of the problem. It was so difficult to diagnose because whenever one thing got slow, EVERYTHING became slow, even if it had nothing to do with whatever was causing the problem.

Solving that collective set of problems took a lot of tinkering with trial and error. We had to level up our MongoDB database administration skills. Then, finally, after a lot of work, we figured out the source of the problem. To mitigate that same problem again in the future, we wrote tooling that let us selectively throttle particular apps or rewrite/limit poorly structured queries on the fly. We also generated some custom Ganglia graphs for the user ID of this Norwegian death metal band so that, in the future, we could swiftly tell whether or not they were to blame for an outage.

It was tough to get there. But we got a handle on it. We all heaved a collective sigh of relief.

But that was only the beginning.

Parse made it easy for mobile app developers to spin up new apps and quickly ship them to the app store. It was a hit! Developers loved the experience. So, day after day, that's what they did. Soon, we were seeing new apps skyrocket to the top of the iTunes Store or within the Android ecosystem. We were hosting a new top app practically every week. Those apps would do things like use us to send push notifications. Millions of device notifications were dumped onto the Parse platform at any time of the day or night.

That's when we started to run into many of the fundamental flaws in the architecture we'd chosen, the languages we were using, and the tools we were using to understand those choices. I want to reiterate: this was *the right thing to do*. We got to market fast, we found a niche, and we delivered. Now, we had to figure out how to grow into the next phase.

I'll pause here to describe some of those decisions we made, and the impact they now had at this scale.

When incoming API requests came in, they were load balanced and handed off to a pool of Ruby on Rails HTTP workers known as unicorns. Our infrastructure was in AWS and all of our EC2 instances hosted a unicorn primary (née, master), which would fork a couple of dozen unicorn child processes; those then handled the API requests themselves. The unicorns were configured to hold a socket open to a number of backends—MySQL

for internal user data, Redis for push notifications, MongoDB for user-defined application data, Cloud Code (containers executing server-side code), Cassandra for Parse analytics, and so on.

It's important to note here that Ruby is not a threaded language. So that pool of API workers was a fixed pool. Whenever any one of the backends got just a little bit slower at fulfilling requests, the pool would rapidly fill itself up with pending requests to that backend. The impact was that whenever a backend became very slow (or became completely unresponsive), the pools would fill up within seconds—and all of Parse would go down.

At first, we attacked that problem by overprovisioning instances: our unicorns ran at 20% utilization during their normal steady state. That approach allowed us to survive some of the gentler slowdowns. But at the same time, we also made the painful decision to undergo a complete rewrite from Ruby on Rails to Golang. We realized that the only way out of this hell hole was to adopt a natively threaded language. It took us over two years to rewrite the code that had taken us one year to write. In the meantime, we were on the cutting edge of experiencing all the ways that traditional operational approaches were fundamentally incompatible with modern architectural problems.

This was a particularly brutal time all around Parse. We had an extremely experienced operations team doing all of the "right things." We were discovering that the best practices we all knew, that were born from using traditional approaches, simply weren't up to the task of dealing with tackling problems in the modern distributed microservices era.

At Parse, we were all-in on Infrastructure as Code. We had an elaborate system of Nagios checks, PagerDuty alerts, and Ganglia metrics. We had tens of thousands of Ganglia graphs and metrics. But those tools were failing us because they were only valuable when we already knew what the problem was going to be—when we knew which thresholds were good and where to check for problems.

For instance, time series database graphs were only valuable when we knew which dashboards to carefully curate and craft—if we could predict which custom metrics we would need in order to diagnose problems. Our logging tools were valuable when we had a pretty good idea of what we were looking for, if we'd remembered to log the right things in advance, and if we knew the right regex to search for. Our APM tools were great when problems manifested as one of the top 10 bad queries, bad users, or bad endpoints that they were looking for.

But we had a whole host of problems that those solutions couldn't help us solve. That previous generation of tools was falling short when:

- Every other day, we had a brand-new user skyrocketing into the Top 10 list for any of the mobile app stores.
- It wasn't load coming from any of the users identified in any of our top 10 lists that was causing our site to go down.
- The list of slow queries are all actually just symptoms of a problem and not the cause (e.g., when it's a bunch of tiny write queries).
- We needed help seeing that our overall site reliability was 99.9%, but that 0.1% wasn't evenly distributed across all users. That 0.1% meant that just one shard was 100% down and it just happened to be the shard holding all data that belonged to a famed multibillion dollar entertainment company with a mousey mascot.
- Every day, a new bot account might pop up and do things that would saturate the lock percentage on our MySQL primary.

These types of problems are all categorically different from the last generation of problems: the ones for which that set of tools was built. They were built for a world where predictability reigned. In those days, production software systems had "the app"—with all of its functionality and complexity contained in one place—and "the database." But now, scaling meant that we blew up those monolithic apps into many different

services. We blew up the database into a diverse range of many different storage systems.

At many companies, like at Parse, our business models turned our products into platforms. We invited users to run any code they saw fit to run on our hosted services. We invited them to run any query they felt like running against our hosted databases. And in doing so, suddenly, all of the control we had over our systems evaporated in a puff of market dollars to be won.

In the era of services as platforms, our customers love how powerful we make them. That drive has revolutionized the software industry. And for those of us running the underlying systems powering those platforms, it meant that everything became massively—and exponentially—harder to not just operate and administer, but also to *understand*.

How did we get here and when did the industry seemingly change overnight? Let's look at the various small iterations that created such a seismic shift.

The evolution toward modern systems

In the beginning—back in those early days of the dotcom era—there was "The App" and "The Database." They were simple to understand. Each of those were either UP or they were DOWN. They were either slow or they were not slow. What we had to do was monitor them for aliveness and acceptable responsiveness thresholds.

That task was not always simple. But, operationally, it was straightforward. For quite some time there, we even centered around a pinnacle of architectural simplicity. The most popular architectural pattern was the LAMP stack: Linux, Apache, Mysql, and PHP or Python. You've probably heard this before, and I feel obligated to say it again now: *if you can solve your problem with a LAMP stack (or equivalent), you probably should.*

When architecting a service, the first rule is to not add *unnecessary complexity*. You should carefully identify the problems you need to solve, and solve *only those problems*. Think carefully about the options you need

to keep open, and keep those options open. What that means is that, most of the time, you should choose boring technology. Don't confuse boring with bad. Boring technology simply means that its edge cases and failure scenarios are well understood by very many people.

However, with the demands of today's users, more of us are discovering that many of the problems we need to solve cannot be solved by the humble-yet-mighty LAMP stack. There is a multitude of reasons that could be the case. You might have higher reliability needs to make resiliency guarantees that a LAMP stack can't provide. Maybe you have too much data or data that's too complex to be well served by a LAMP stack. Usually, we reach the edges of that simple architectural model in service of scale, reliability, or speed—that's what drives us to shard, partition, or replicate our stacks.

How we manage our code also matters. In larger or more complex settings, the monorepo also creates organizational pressures that can drive technical change. Splitting up a code base can create clearer areas of ownership in ways that allow an organization to move more swiftly and autonomously than they could if everyone was contributing to "The One Big App."

These types of needs—and more—are part of what has driven modern shifts in systems architecture that are clear and pronounced. On a technical level, these shifts include a number of primary effects:

- The decomposition of everything, from one to many.
- The need for a variety of datastores, from one database to many storage systems.
- A migration away from large monolithic applications, toward many smaller microservices.
- A variety of infrastructure types, away from "big iron" servers, toward containers, functions, serverless, and other ephemeral, elastic resources.

These technical changes have also had powerful ripple effects at the human and organizational level: our systems are *sociotechnical*. The complexity introduced by these shifts at the social level (and their associated feedback loops) have driven further changes to the systems and how we think about them.

Consider some of the prevalent social qualities we think of in conjunction with the computing era during the height of popularity for the LAMP stack. There was a tall organizational barrier between operations and development teams, and from time to time, code would get lobbed over that wall. Ops teams notoriously resisted introducing changes to production systems, tending closely to their uptime—preventing developers from ever directly touching prod. In that type of "glass castle" approach, deploy freezes were a commonly used method for keeping production services stable.

In the LAMP stack era, that protective approach to production wasn't entirely wrong either. Back then, much of the chaos in production systems was indeed inflicted when introducing bad code. When new code was deployed, the effects were pretty much binary: the site was up or it was down (maybe, sometimes if you were unlucky, it had a third state: inexplicably slow). Very few architectural constructs (that are commonplace today) existed back then to control the side effects of dealing with bad code. There was no such thing as a graceful degradation of service. If developers introduced a bug to some small, seemingly trivial, system component (for example, export functionality), it might crash the entire system every time someone used it. There were no mechanisms available to temporarily disable those turns-out-not-so-trivial subcomponents. It wasn't possible to just fix that one broken subsystem while the rest of the service ran unaffected. Some teams would attempt to add that sort of granular control to their monoliths. But that path was so fraught and difficult that most didn't even try.

Regardless, many of those monolithic systems continued to get bigger. As more and more people tried to collaborate on the same paths, problems like lock contention, cost ceilings, and dynamic resource needs, etc., became showstoppers. The seismic shift from one-to-many became a necessity for

these teams. Monoliths were quickly decomposed into distributed systems and, with that shift, came a number of second-order effects:

- The binary simplicity of service availability as up or down shifted to complex availability heuristics to represent any number of partial failures.
- Code deployments to production that had relatively simple "flip a switch" type schemes shifted toward progressive delivery.
- Deployments that immediately changed code paths shifted to promotion schemes where code deployments were decoupled from feature releases.
- Applications that had one current version running in production shifted toward typically having multiple versions baking in production at any given time.
- Having an in-house operations team running your infrastructure shifted toward many critical infrastructure components being run by other teams or other companies in ways that may or may not even be accessible to you, the developer.
- Monitoring systems that worked well for alerting against and troubleshooting previously encountered known failures shifted to being woefully inadequate in a distributed world where unknown failures that had never been previously encountered (and may never be encountered again) were the new type of prevalent problem.

The tools and techniques needed to manage monolithic systems like the LAMP stack were radically ineffective for running modern systems. Systems where applications are deployed in one "big bang" release are managed rather differently than microservices. With microservices, applications are often rolled out piece by piece and code deployments don't necessarily release features because feature flags now enable or disable code paths with no deployment required. Similarly, in a distributed world,

staging systems have become less useful or reliable than they used to be. Even in a monolithic world, it was always difficult to replicate a production environment to staging. Now, in a distributed world, it's effectively impossible. That means debugging and inspection has become ineffective in staging, and shifted to a model where it has to be accomplished in production itself.

The evolution toward modern practices

The technical and social aspects of our systems are interrelated. Because our systems are sociotechnical, the emphasis on shifting our technology toward different performance models also means the models that define how our teams perform must also shift. These shifts are so interrelated that it's impossible to draw a clean boundary between them. In the last section, many of the technology shifts described influenced how teams had to reorganize and change practices in order to support them.

There is, however, one area where a radical shift in social behavior is absolutely clear. The evolution toward modern distributed systems also brings with it tertiary effects that change the relationship engineers must have with their production environment:

- User experience can no longer be generalized as being the same for all service users. It has shifted to a model where different users of a service may be routed through the system in different ways, using different components, providing experiences that can vary widely.
- Given the variety of all possible routes, interactions, and components involved, services no longer tend to break in the same few predictable ways over and over. That predictability has shifted to a model where any time a service is degraded, any number of impossibly large combinations of code paths could be involved, and it's likely that failure mode will be a completely new discovery for this system.

- Monitoring alerts that page engineers to respond when measures that look for edge cases in production when system conditions exceed known thresholds, generate a tremendous amount of false positives, false negatives, and meaningless noise. Alerting has shifted to a model where fewer alerts are triggered by only focusing on symptoms that directly impact user experience.
- Debuggers can no longer be attached to one specific runtime. Service requests have shifted to a model where fulfillment requires hopping across a network, spanning multiple runtimes, often multiple times per individual request.
- Known recurring failures that require manual remediation and can be defined in a playbook are now no longer the norm. Service failures have shifted from that model toward one where known recurring failures can be recovered automatically. Failures that cannot be automatically recovered, and therefore trigger an alert, likely mean the responding engineer will be facing a novel problem.

These tertiary shifts mean that a massive gravitational shift in focus is happening away from the importance of pre-production and toward the importance of being intimately familiar with production. Traditional efforts to harden code and ensure its safety *before* it goes to production are starting to be accepted as limiting and somewhat futile in nature. Test-driven development and running tests against staging environments still have use. But they can never replicate the wild and unpredictable nature of how that code will be used in production.

As developers, we all have a fixed amount of cycles we can devote to accomplishing our work. The limitation of the traditional approach is that it focuses on pre-production hardening first and foremost. Any leftover scraps of attention, if they even exist, are then given to focusing on production systems.

That ordering must be reversed.

In modern systems, we *must* focus the bulk of our engineering attention and tooling on production systems, first and foremost. The leftover cycles of attention should be applied to staging and pre-production systems. There is value in staging systems. But it is secondary in nature.

Staging systems are not production. They can never replicate what is happening in production. The sterile lab environment of pre-production systems can never mimic the same conditions under which real paying users of your services will test that code in the real world. Yet many teams still treat production as a glass castle.

Engineering teams must reprioritize the value of production and change their practices accordingly. By not shifting production systems to their primary focus, these teams will be relegated to toiling away on production systems with subpar tooling, visibility, and observability. They will continue to treat production as a fragile environment where they hesitate to tread because they lack the controls to make small tweaks, tune settings, gently degrade services, or progressively deploy changes in response to problems they intimately understand.

The technological evolution toward modern systems also brings with it a social evolution that means engineering teams must change their relationship with production. Pre-production systems can never replicate modern production systems. As that trend becomes emergent, architectures more complex, and these systems more operationally integrated ubiquitously, teams that do not change that relationship will more acutely suffer the pains of not doing so.

The sooner that production is no longer a glass castle in modern systems, the sooner the lives of the teams responsible for those systems—and the experience of the customers that use them—will improve.

Shifting practices at Parse

At Parse, we had to undergo those changes rather painfully as we worked to quickly scale. But those changes didn’t happen overnight. The traditional

practices we had were all we knew. Personally, I was following the well-known production system management path I had used at other companies many times before.

When novel problems happened, like with the Norwegian death metal band, I would dive in and do a lot of investigation until the source of the problem was discovered. My team would run a post-mortem to dissect the issue, write a playbook to instruct our future selves on how to deal with it, craft a custom dashboard (or two) that would surface that problem instantly next time, then move on and consider the problem resolved.

That pattern works very well for monoliths, where truly novel problems are rare. It worked very well in the earliest days at Parse. But that pattern is completely ineffective for modern systems, where truly novel problems are likely to be the bulk of all the problems encountered. Once we started encountering a barrage of categorically different problems on a daily basis, the futility of that approach became undeniable. In that setting, all the time we spent on post-mortems, creating playbooks, and crafting custom dashboards was little more than wasted time. We would never see those same problems again.

The reality of this traditional approach is that so much of it involved guessing. In other systems, I had gotten so good at guessing what might be wrong that it felt almost effortless. I was intuitively in tune with my systems. I could eyeball a complex set of dashboards and confidently tell my team, “The problem is Redis,” despite the fact that Redis was represented nowhere on that screen. I took pride in that sort of technical intuition. It was fun! I felt like a hero.

This is an important characteristic that I want to underline about hero culture in the software industry. In monolithic systems, with LAMP stack style operations, the debugger of last resort is always the person who has been there the longest. The engineer with the most seniority is the ultimate point of escalation. They have the most scar tissue, the largest catalog of outages in their mental inventory, and are the ones who inevitably swoop in to save the day.

What it means is that these heroes can also never take a real vacation. I remember being on my honeymoon in Hawaii when I was paged at 3:00 a.m. because MongoDB had somehow taken down the Parse API service. My boss, the CTO, was overwhelmingly apologetic. But the site was down. It had been down for over an hour and no one there could figure out why. So they paged me. Yes, I complained. But also, deep down, it secretly felt great. I was *needed*. I was necessary.

If you've ever been on-call to manage a production system, you might recognize that pattern. Hero culture is a terrible pattern. It's not good for the company. It's not good for the hero (it quickly leads to burnout). It's horribly depressing for every engineer who comes along later and has no chance of ever being the "best" debugger unless a more senior engineer leaves. That pattern is completely unnecessary. But most importantly, it *doesn't scale*.

At Parse, to address our scaling challenges, we needed to reinvest the precious time we were effectively wasting by using old approaches to novel problems. So many of the tools that we had at that time—and that the software industry still relies on—were geared toward pattern-matching. Specifically, they were geared toward helping the expert-level and experienced engineer pattern-match previously encountered problems to newer variants on the same theme.

I never questioned that because it was the best technology and process we had.

After Facebook acquired Parse in 2015, I came to discover Facebook's Scuba tool. Scuba is the data management system Facebook uses for most real-time analysis. It's a fast, scalable, distributed, in-memory database that ingest millions of rows (events) per second. It stores live data completely in memory and aggregates it across hundreds of servers when processing queries. My experience with it was rough. I thought Scuba had an aggressively ugly, and even hostile, user experience. But it did one thing extremely well that permanently changed my approach to troubleshooting

systems. It let me slice and dice data, in near real-time, on dimensions of infinitely high cardinality.

We started sending some of our application telemetry datasets into Scuba and began experimenting with its analytical capabilities. The time that it took us to discover the source of problems when we encountered novel problems dropped dramatically. Previously, with the traditional pattern-matching approach, it would take us days—or possibly never?—to understand what was happening. With real-time analytics that could be arbitrarily sliced and diced along any arbitrary dimension of infinitely high cardinality, that time dropped to mere minutes—possibly seconds.

We could start investigating a novel problem by starting with the symptoms and following a "trail of breadcrumbs" to the solution, no matter where it led, no matter if it was the first time we had experienced that problem or not. We didn't have to be familiar with the problem in order to solve it. Instead, we now had an analytical and repeatable method for asking a question, getting answers, and using those answers to ask the next question until we got to the source of a problem. Troubleshooting issues in production meant starting with the data and relentlessly taking one methodical step after another until we arrived at a solution.

That newfound ability irrevocably shifted our practices.

Rather than relying on intricate knowledge and a catalog of outages locked up in the mental inventory of individual engineers—inaccessible to their teammates—we now had data and methodology that was visible and accessible by everyone in a shared tool. For the team, that meant we could follow each other's footsteps, retrace each other's paths, and understand what we were each thinking. That ability freed us from the trap of relying on tools geared toward pattern-matching.

With metrics and traditional monitoring tools, we could easily see performance spikes or notice that a problem might be occurring. But those tools didn't allow us to arbitrarily slice, dice, and dig our way down our stack to identify the source of problems or see correlations between errors that we had no other possible way of discovering. Those tools also required

us to predict what data might be valuable in advance of investigating a novel problem (as if we knew which questions we would need to ask before the novel problem ever presented itself). Similarly, with logging tools, we had to remember to log anything useful in advance—and know exactly what to search for when that was relevant to the investigation. With our APM tools, we could quickly see problems that our tool vendor predicted would be the most useful to know, but that did very little when novel problems were the norm.

Once we made that paradigm shift, the best debugger was no longer the person who had been there the longest. The best debugger became whoever was the most curious, the most persistent, and the most literate with our new analytical tools. And that turned out to be anyone on my team at any given time. It democratized access to our system and the shared wisdom of its engineers.

By approaching every problem as novel, we could determine what was truly happening any time there was an issue in production instead of simply pattern-matching it to the nearest analogous previous problem. That made it possible to effectively troubleshoot systems that presented the biggest challenges with a traditional approach. We could now quickly and effectively diagnose problems with:

- Microservices and requests that spanned multiple runtimes
- Polyglot storage systems, without requiring expertise in each
- Multi-tenancy and running both server-side code and queries; we could easily drill down into individual user experiences to see exactly what was happening

Even with our few remaining monolithic systems that used boring technology and where all possible problems were pretty well understood, we experienced some gains. We didn't necessarily discover anything new, but we were able to ship software more swiftly and confidently as a result.

At Parse, what allowed us to scale our approach was learning how to work with a system that was *observable*. By gathering application telemetry, at

the right level of abstraction, from the perspective of the user, and having the ability to analyze it in real-time, we gained magical insights. We removed the limitations of our traditional tools once we had the capability to ask any question, trace any sequence of steps, and understand any internal system state, simply by observing the outputs of our applications.

We were able to modernize our practices once we had *observability*.

Conclusion

Charity's story, from her days at Parse, illustrates how and why organizations make a transition from traditional tools and monitoring approaches to scaling their practices with modern distributed systems and observability.

Charity departed Facebook in 2015, shortly before they announced the impending shutdown of the Parse Hosting Service. Since then, many of the problems she and her team faced in managing modern distributed systems have only become more common as the software industry has shifted toward adopting similar technologies. We believe that shift accounts for the enthusiasm behind, and the rise in popularity of, observability. Observability is the solution to problems that have become prevalent when scaling modern systems.

In further chapters, we'll explore the many facets, impacts, and benefits that observability delivers.

Chapter 4. How Observability Relates to DevOps, SRE, and Cloud Native

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

So far, we've referenced observability in the context of modern software systems. Therefore, it's important to unpack how observability fits into the landscape of other modern practices such as the DevOps, SRE (Site Reliability Engineering), and Cloud Native movements. This chapter examines how these movements have both influenced the need for observability and integrated it into their practices.

Observability does not exist in a vacuum; instead, it is both a consequence and an integral part of the DevOps, SRE, and Cloud Native movements. Like testability, observability is a property of these systems that improves understanding of them. Rather than being something that we add once, or having a one-size-fits-all solution, observability and testability require continuous investment. As they improve, benefits accrue for the developers and end users of our systems. By examining why the DevOps, SRE, and

Cloud Native movements created a need for observability and integrated its use, we can better understand why observability has become a mainstream topic and why increasingly diverse teams are adopting this practice.

Cloud Native DevOps, and SRE in a nutshell

In contrast with the monolithic and waterfall development approaches employed in the 1990s to early 2000s, modern software development and operations teams increasingly use Cloud Native and Agile methodologies. In particular, these methodologies enable teams to autonomously release features without tightly coupling their impact to other teams. That capability unlocks several key business benefits, including higher productivity, better profitability, and more[1]. For example, the ability to resize individual service components upon demand and pool resources across a large number of virtual and physical servers means the business benefits from better cost controls and scalability.

However, these benefits are not free. An often overlooked aspect of introducing these capabilities is that it also introduces a management cost. Abstracted systems with dynamic controls introduce new challenges of emergent complexity and non-hierarchical communications patterns. Older monolithic systems had less emergent complexity, and thus simpler monitoring approaches sufficed; we could easily reason about what was happening inside those systems and where unseen problems might be occurring. Today, running Cloud Native systems feasibly at scale demands more advanced sociotechnical practices like observability.

The Cloud Native Computing Foundation defines Cloud Native as "building and running scalable applications in modern, dynamic environments... [Cloud Native] techniques enable loosely coupled systems that are *resilient*, *manageable*, *and observable*. Combined with robust automation, they allow engineers to make high-impact changes *frequently and predictably with minimal toil*."[2] By minimizing toil[3] (repetitive manual human work) and emphasizing observability, Cloud Native systems

empower developers to be creative. This definition focuses not just on scalability, but also upon development velocity and operability as goals.

It's worth focusing on the fact that the shift to Cloud Native is not just one that requires adopting a complete set of new technologies; it also requires changing how people work. That shift is inherently sociotechnical. On the surface, using the toolchain itself has no explicit requirement to adopt new social practices. But to achieve the promised benefits of the technology, it becomes necessary to also change the way that people work. Although this should be evident from the stated definition and goals, it's not uncommon for teams to get several steps in before realizing that their old work habits do not help them address the management costs introduced by this new technology. That is why successful adoption of Cloud Native design patterns is inexorably tied to the need for observable systems and for DevOps and Site Reliability Engineering (SRE) practices.

Similarly, DevOps and SRE both highlight a desire to shorten feedback loops and reduce operational toil in their definitions and practices. DevOps provides "Better Value; Sooner, Safer, & Happier"[4] through culture and collaboration between development and operations groups. SRE joins together systems engineering and software skillsets to solve complex operational problems through developing software systems rather than manual toil[5]. As we'll explore in this chapter, the combination of Cloud Native technology, DevOps and SRE methodologies, and observability are stronger together than each of their individual parts.

Observability: Debugging Then vs. Now

The goal of observability is to provide a level of introspection that helps people reason about the internal state of their systems and applications. That state can be achieved a number of ways. For example, that could be done utilizing a combination of logs, metrics, and traces. But the goal of observability itself is agnostic to how it's accomplished.

For monolithic systems, where we could anticipate the potential areas of failure, one person in isolation could debug our systems and achieve

appropriate observability using verbose application logging, or coarse system-level metrics such as CPU/disk utilization combined with flashes of insight. However, these legacy tools and instinctual techniques no longer work for the new set of management challenges created by the opportunities of Cloud Native systems.

Among the example technologies mentioned in the Cloud Native definition are containers, service meshes, microservices, and immutable infrastructure. Compared to legacy technologies like virtual machines and monolithic architectures, containerized microservices inherently introduce new problems such as cognitive overload from interdependencies between components, transient state discarded after container restart, and incompatible versioning between separately released components. Immutable infrastructure means that it's no longer feasible to ssh into a host for debugging, as it may perturb the state on that host. Service meshes add an additional routing layer which provide a powerful way to collect information about how service calls are happening, but that data is of limited use without the ability to store it for later analysis.

Debugging anomalous issues requires a new set of capabilities to help engineers detect and understand problems from within their systems. Tools such as distributed tracing can help capture the state of system internals when specific events occurred. By adding wide context to each event, it's possible to create a rich view of what was happening in all the parts of a system that are typically hidden and impossible to reason about. For example, if engineers have the ability to systematically drill down into which hosts should be examined using nested sets of metrics aggregated at different levels (or exemplar wide events), then it no longer matters that logs are sharded across kubelets and ephemerally retained. If we can visualize each individual step in service request executions with distributed tracing, then it no longer matters that services have complex dependencies. If we can visualize the relationship between calling and receiving code paths and versions, then version skew between components is a solvable problem. Observability provides a shared context that enables teams to debug problems in a cohesive and rational way, regardless of how complex

a system might be, rather than relying upon the entire state of the system to fit within one person's mind.

Observability empowers DevOps and SRE practices

It's the job of DevOps and SRE teams to understand production systems and tame complexity, and thus it's natural for them to care about the observability of the systems they build and run. SRE focuses on the idea of managing services according to Service Level Objectives (SLOs) and Error Budgets. DevOps focuses on managing services through cross-functional practices where developers own running their code in production. Rather than starting with a plethora of alerts that enumerate potential causes of outages, mature DevOps and SRE teams both measure whether there are visible symptoms of user pain, then drill down into understanding the outage using observability tooling.

That shift away from cause-based monitoring and towards symptom-based monitoring means that a need exists to be able to explain the failures you see in practice, rather than the traditional approach of enumerating a growing list of known failure modes. Rather than burning a majority of their time responding to a slew of false alarms that have no bearing upon the end-user visible performance, teams can instead focus on systematically winnowing hypotheses and devising mitigations for actual systems failures. [for more on this, see the chapter on SLOs and Observability]

Beyond adoption of observability for break/fix use cases, forward-thinking DevOps and SRE teams use engineering techniques such as feature flagging, continuous verification, incident analysis. Observability supercharges these use cases by providing the data required to effectively practice them.

- *Chaos engineering* and continuous verification requires us to have observability to "detect when the system is normal and how it deviates from that steady-state as the experiment's method is executed."[6] We cannot meaningfully perform chaos experiments without the ability to understand the system's baseline state, to

predict expected behavior under test, and to explain deviations from expected behavior. “There is no point in doing chaos engineering when you actually don’t know how your system is behaving at your current state before you inject chaos.”[7]

- *Feature flagging* introduces novel combinations of flag states that we cannot test exhaustively. Thus, we need observability to understand the individual and collective impact of each feature flag, user by user. The notion of monitoring behavior component by component no longer holds when an endpoint can execute in multiple different ways depending upon what user is calling it with which flags enabled.
- *Progressive release* patterns such as canarying and blue/green deployment require observability to effectively know when to stop the release and to analyze whether the system’s deviations from expected are a result of the release.
- *Incident analysis* and blameless postmortems require us to construct clear models about our sociotechnical systems -- not just what was happening inside the technical system at fault, but also what the human operators *believed* to be happening during the incident. Thus, robust observability tooling facilitates performing excellent retrospectives by providing a post-facto paper trail and details to cue retrospective writers.

As the practices of DevOps and SRE continue to evolve, and as platform engineering grows as an umbrella discipline, inevitably more innovative engineering practices will emerge in your toolbelts. But all of those innovations will depend upon having observability as a core sense to understand our modern complex systems. The shift toward DevOps, SRE, and Cloud Native practices have created a need for a solution like observability. In turn, observability has also supercharged the capabilities of teams that have adopted its practice.

1 See, for instance, the Accelerate State of DevOps Report, 2019 by Forsgren et al.

2 https://github.com/cncf/toc/blob/master/DEFINITION.md

3 https://landing.google.com/sre/sre-book/chapters/eliminating-toil/

4 https://medium.com/sooner-safer-happier/want-to-do-an-agile-transformation-dont-focus-on-flow-quality-happiness-safety-and-value-11e01ee8f8f3

5 https://landing.google.com/sre/sre-book/chapters/introduction/

6 https://learning.oreilly.com/library/view/chaos-engineering-observability/9781492051046/ch01.html

7 https://www.heavybit.com/library/podcasts/o11ycast/ep-11-chaos-engineering-with-ana-medina-of-gremlin/

Chapter 5. Structured Events Are the Building Blocks of Observability

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 5th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

In the first section of this book, we examined the definition of observability, its necessity in modern systems, its evolution from traditional practices, and an introduction to how it is currently being used in practice. This section delves deeper into technical aspects, and details why particular requirements are necessary in observable systems. In later chapters, we'll detail how application telemetry must be batched and grouped in preparation for network transport to a persistence layer, how that data must be stored in that persistence layer, the types of analytical capabilities that must be present when that data is queried, and various options for the types of debugging data that must be captured in observable systems.

In this chapter, we'll start by examining the fundamental building block of observability: the structured event. If you accept our definition of observability—that it's about the unknown-unknowns, that it means being

able to ask any question, understand any inner system state, without anticipating or predicting it in advance—there are a number of technical prerequisites you must meet to fulfill this definition.

In order to ask *any* question, about any combination of telemetry details, this requires the ability to arbitrarily slice and dice data along any number of dimensions. It also requires that your telemetry must have been gathered while retaining the connective tissue of the event—not firing off metrics willy-nilly, but gathering them up per request/per service.

With old fashioned style metrics, you had to define custom metrics up front if you wanted to start asking a new question and gathering new telemetry to answer it. Metrics did not retain the connective event, so you could not ask new questions or look for new outliers in the existing dataset. With observability, you can ask new questions of your existing dataset at any time.

In an observable system, you must be able to iteratively explore system aspects ranging from high-level aggregate performance all the way down to the raw data used in individual requests. The data format used to gather that data—the arbitrarily-wide structured event—is what makes that possible. It is not optional, it is not an implementation detail. Instead, it is a requirement for what makes any level of analysis possible within that wide ranging view.

Debugging with structured events

First, let's start by defining an event. For purposes of understanding the impact to your production services, an event is a record of everything that occurred while one particular request interacted with your service. To create that record, you start by initializing an empty blob right at the beginning, when the request first enters your service. During the lifetime of that request, any interesting details about what occurred—unique ids, variable values, headers, every parameter passed by the request, execution time, any calls made to remote services, the execution time of those remote calls, or any other bit of context that may later be valuable in debugging—get

appended to that blob. Then, when the request is about to exit or error, that entire blob is captured as a rich record of what just happened. The data written to that blob is organized and formatted, as key-value pairs, so that it's easily searchable: in other words, the data should be structured. That's a structured event.

As you debug problems in your service, you will need to identify individual requests that are anomalous, then identify what those outliers have in common. Exploring those outliers requires filtering and grouping by different dimensions—and combinations of dimensions—contained within that event that might be relevant to your investigation.

The data that you feed into your observability system as structured events needs to capture the right level of abstraction to help observers determine the state of your applications, no matter how bizarre or novel. Information that is helpful for investigators may have runtime information not specific to any given request (such as container information, versioning information, etc), as well as information about each request that passes through the service (such as shopping cart id, user id, session token, etc). Both types of data are useful for debugging.

The sort of data that is relevant to debugging a request will vary, but it helps to think about that by comparing the type of data that would be useful using a conventional debugger. For example, in that setting, you might want to know the values of variables during the execution of the request, and to understand when function calls are happening. With distributed services, those function calls may happen as multiple calls to remote services. In that setting, any data about the values of variables during request execution can be thought of as the *context* of the request.

All of that data should be accessible for debugging and it should be captured with enough granularity that investigators can arbitrarily slice and dice it along any number of dimensions. That unit is the structured event. We'll delve further into the types of information that should be captured in structured events, but first let's compare how capturing system state with structured events differs from traditional debugging methods.

The limitations of metrics as a building block

First, let's define what we mean by "metrics." Unfortunately, we have to disambiguate this term because the word "metrics" is sometimes used as a generic synonym for telemetry—e.g. "all of our metrics are up and to the right, boss!" For clarity, as you may recall from Chapter 2, what we mean by "metrics" are those single numbers collected to represent system state, with tags optionally appended for grouping and searching those numbers. Metrics are the staple upon which traditional monitoring of software systems has been built.

However, the fundamental limitation of the metric is that it is a pre-aggregated measure. The numerical values generated by a metric reflect an aggregated report of system state over a predefined period of time. When that number is reported to a monitoring system, that pre-aggregated measure now becomes the lowest possible level of granularity for examining system state. That aggregation obscures many possible problems. Further, a request into your service may be represented by having a part in hundreds of metrics over the course of its execution. Those metrics are all distinct measures—disconnected from one another—that lack the sort of connective tissue and granularity necessary to reconstruct exactly which metrics belong to the same request.

For example, a metric for `page_load_time` might examine the average time it took for all active pages to load during the trailing 5s period. Another metric for `requests_per_second` might examine the number of HTTP connections any given service had open during the trailing 1s period. Metrics reflect system state, as expressed by numerical values derived by measuring any given property over a given period of time. The behavior of all requests that were active during that period are aggregated into one numerical value. In this example, investigating what occurred during the lifespan of any one particular request flowing through that system would provide a trail of breadcrumbs that leads to both of these metrics. However, the level of available granularity necessary to dig further is unavailable.

In an event-based approach, a webserver request could be instrumented to record each parameter submitted with the request (for example, `userid`), the intended subdomain (`www`, `docs`, `support`, `shop`, `cdn`, etc), total duration time, any dependent child requests, the various services involved (`web`, `database`, `auth`, `billing`) to fulfill those child requests, the duration of each child request, and more. Those events can then be arbitrarily sliced and diced in various ways, across any window of time, to present any view relevant to an investigation.

In contrast to metrics, events are snapshots of what happened at a particular point-in-time. One thousand discrete events may have occurred within that same trailing 5s period in the example above. If each event recorded its own `page_load_time`, when aggregated along with the other 999 events that happened in that same 5s period, you could still display the same average value as shown by the metric. Additionally, with the granularity provided by events, an investigator could also that same average over a 1s period, or subdivide queries by fields like user id or hostname to find correlations with `page_load_time` in ways that simply aren't possible when using the aggregated value provided by metrics.

An extreme counterargument to this analysis could be that, given enough metrics, the granularity needed to see system state on the level of individual requests could be achieved. Putting aside the wildly impractical nature of that approach, ignoring that concurrency could never completely be accounted for, and that investigators would still be required to stitch together which metrics were generated along the path of a request's execution, the fact remains that metrics-based monitoring systems are simply not designed to scale to the degree necessary to capture those measures. Regardless, many teams relying on metrics for debugging find themselves in an escalating arms race of continuously adding more and more metrics in an attempt to do just that.

As aggregate numerical representations of predefined relationships over predefined periods of time, metrics serve as only one narrow view of one system property. Their granularity is too large and their ability to present

system state in alternate views is too rigid to achieve observability. Metrics are too limiting to serve as the fundamental building block of observability.

The limitations of unstructured data as a building block

As seen earlier, structured data is clearly defined and searchable types of data. Unstructured data isn't organized in an easily searchable manner and is usually stored in its native format. When it comes to debugging production software systems, the most prevalent use of unstructured data is a construct older than the metric: log files.

Log files are essentially large blobs of unstructured text, designed to be readable by humans but difficult for machines to process. Log files are documents generated by applications and various underlying infrastructure systems that contain a record of all notable—as defined by a configuration file somewhere—events that have occurred. For decades, they have been an essential part of system debugging applications in any environment. Logs typically contain tons of useful information: a description of the event that happened, an associated timestamp, a severity of level type associated with that event, and a variety of other relevant metadata (user id, IP address, etc).

Traditional logs are unstructured because they were designed to be human readable. Unfortunately, for purposes of human readability, logs often separate the vivid details of one event into multiple lines of text, like so:

```
6:01:00 accepted connection on port 80 from 10.0.0.3:63349
6:01:03 basic authentication accepted for user foo
6:01:15 processing request for /super/slow/server
6:01:18 request succeeded, sent response code 200
6:01:19 closed connection to 10.0.0.3:63349
```

While that sort of narrative structure can be helpful when first learning the intricacies of a service in development, it generates huge volumes of noisy data that becomes slow and clunky in production. In prod, these chunks of narrative are often buried in millions upon millions of other lines of text.

Typically, they're useful in the course of debugging once a cause is already suspected and an investigator is verifying their hypothesis by digging through logs for verification.

However, modern systems no longer run at an easily comprehensible human scale. In traditional monolithic systems, human operators had a very small number of services to manage. Logs were written to the local disk of the machines where applications ran. In the modern era, logs are often streamed to a centralized aggregator, where they're dumped into very large storage backends.

Searching through millions of lines of unstructured logs can be accomplished by using some type of log file parser. Parsers split log data into chunks of information and attempt to group them in meaningful ways. However, with unstructured data, parsing gets complicated because there are different formatting rules (or no rules at all) for different types of log files. Logging tools are full of different approaches to solving this problem, with varying degrees of success, performance, and usability.

The solution is to instead create structured log data that is machine-parsable. From the example above, a structured version might instead look something like this:

```
time="6:01:00" msg="accepted connection" port="80"
authority="10.0.0.3:63349"
time="6:01:03" msg="basic authentication accepted" user="foo"
time="6:01:15" msg="processing request" path="/super/slow/server"
time="6:01:18" msg="sent response code" status="200"
time="6:01:19" msg="closed connection" authority="10.0.0.3:63349"
```

When connecting observability approaches to logging, it helps to think of an event as a unit of work within your systems. An event should contain information about what it took for a service to perform a unit of work. A unit of work can be seen as somewhat relative. For example, a unit of work could be downloading a single file, parsing it, and extracting specific pieces of information. Yet other times, it could mean processing an answer after extracting specific pieces of information from dozens of files. In the context of services, a unit of work could be accepting an HTTP request and doing

everything necessary to return a response. Yet other times, one HTTP request can generate many other events during its execution.

Ideally, an event should be scoped to contain everything about what it took to perform that unit of work. It should record the input necessary to perform the work, any attributes gathered—whether computed, resolved, or discovered—along the way, the conditions of the service as it was performing the work, and details about the result of the work performed.

Many logs are only portions of events, regardless of whether those logs are structured. It's common to see anywhere from a few to a few dozen log lines or entries that, when taken together, represent what could be considered one unit of work. So far, the example we've been using does just that: one unit of work (the handling of one connection) is represented by five separate log entries. Rather than being helpfully grouped into one event, the messages are spread out into many different messages. Sometimes, a common field—like a request id—might be present in each entry so that the separate entries can be stitched together. But, often, typically there won't be.

The log lines in our example could instead be rolled up into one singular event. Doing so could instead make it look like this:

```
time="2019-08-22T11:56:11-07:00" level=info msg="Served HTTP
request"
    authority="10.0.0.3:63349" duration_ms=123
path="/super/slow/server" port=80
    service_name="slowsvc" status=200 trace.trace_id=eafdf3123
user=foo
```

That representation can be formatted in different ways, including JSON. Most commonly this type of event information could appear as a JSON blog like so:

```
{
    "authority":"10.0.0.3:63349",
    "duration_ms":123,
    "level":"info",
    "msg":"Served HTTP request",
```

```
    "path":"/super/slow/server",
    "port":80,
    "service_name":"slowsvc",
    "status":"200",
    "time":"2019-08-22T11:57:03-07:00",
    "trace.trace_id":"eafdf3123","user":"foo"
}
```

The goal of observability is to enable an ability to interrogate your event data in arbitrary ways to understand the internal state of your systems. Any data used to do so must be machine-parsable in order to facilitate that goal. Unstructured data is simply unusable for that task. Log data can, however, be useful when redesigned to resemble a structured event: the fundamental building block of observability.

Properties of events that are useful in debugging

Earlier in this chapter, we defined an event as a record of everything that occurred while one particular request interacted with your service. When debugging modern distributed systems, that is approximately the right scope for a "unit of work." In order to create a system where an observer can understand any state of the system—no matter how novel or complex—an event must contain ample information that may be relevant to an investigation.

The backend datastore for an observable system must allow your events to be arbitrarily wide. When using structured events to understand service behavior, remember that the model is to initialize an empty blob and record anything that may be relevant for later debugging. That can mean pre-populating the blob with data known as the request enters your service: request parameters, environmental information, runtime internals, host/container statistics, etc. As the request is executing, other valuable information may be discovered: user ID, shopping cart ID, other remote services to be called to render a result, or any various bits of information that may help you find and identify this request in the future. As the request

exits your service, or returns an error, there are several bits of data about how the unit of work was performed: duration, response code, error messages, etc. It is not uncommon for events generated with mature instrumentation to contain 300 - 400 dimensions per event.

Debugging novel problems typically means discovering previously unknown failure modes (i.e. *unknown unknowns*) by searching for events with outlying conditions and finding patterns or correlating them. For example, if a spike of errors occur, you may need to slice across various event dimensions to find out what they all have in common. In order to make those correlations, your observability solution will need the ability to handle fields with high cardinality.

Cardinality refers to the number of unique elements in a set. Some dimensions in your events will have very low or very high cardinality. Any dimension containing unique identifiers will have the highest possible cardinality. For example, a field like a hash representing Social Security Number would be high cardinality (every person has a unique number). Fields like first name or last name would have slightly lower cardinality than Social Security numbers (there might be multiple “Jane” and “Smith” entries), but still high cardinality. A field like gender, though non-binary, would be low cardinality. And a field like species—presuming your users are all human—would have only one value: making it the lowest possible cardinality.

An observability tool must be able to support high-cardinality queries to be useful to an investigator. In modern systems, many of the dimensions that are most useful to debug novel problems are high cardinality. Investigation also often requires stringing those high cardinality dimensions together (i.e. high dimensionality) to find deeply hidden problems. Debugging problems is often the practice of trying to find a needle in a haystack. High cardinality and high dimensionality are the capabilities that enable you to find very fine-grained needles in deeply complex distributed system haystacks. For example, to get to the source of an issue, you may need to create a query that finds, “*all Canadian users, running iOS11 version 11.0.4, using the French language pack, who installed the app last Tuesday, running*

firmware version 1.4.101, who are storing photos on shard3 in region us-west-1." Every single one of those constraints is a high cardinality dimension.

To enable that functionality, it is important that events can be **arbitrarily wide**. That is to say that fields within an event cannot be limited to known or predictable fields. Limiting ingestible events to containing known fields may be an artificial constraint imposed by the backend storage system used by a particular analysis tool. Backend datastores may require data schemas. Using a schema requires that users of the tool be able to predict, in advance, which dimensions may need to be captured. Often, in the course of investigating a problem, users may discover that unexpected data should also be captured. Additional dimensions should be able to be added at any time. Using schemas or other strict limitations on data types, or shapes, are also anathema to the goals of observability.

Conclusion

The fundamental building block of observability is the structured event. Observability requires the ability to arbitrarily slice and dice data along any number of dimensions, in order to answer any question when stepping through the debugging process. Structured events should be scoped as everything required to accomplish one unit of work. For distributed services, that typically means scoping an event as the record of everything that happened during the lifetime of one individual service request.

Metrics aggregate system state over a predefined period of time. They serve as only one narrow view of one system property. Their granularity is too large and their ability to present system state in alternate views is too rigid to stitch them together to represent one unit of work for any particular service request. Metrics are too limiting to serve as the fundamental building block of observability.

Unstructured logs are human readable, but computationally difficult to use. At the scale of modern systems, logs must be machine-parsable to be useful for the goals of observability. Structured logs can be useful, presuming they

follow the same guidelines for capturing everything needed to perform one unit of work: in which case, they become analogous to structured events—the fundamental building block of observability.

Structured events are the fundamental building blocks, however observability tools must be able to support at least two distinct properties for those events: they must allow structured events to be arbitrarily wide, and queries against these events must support high cardinality and high dimensionality. In the next few chapters, we will further define the necessary capabilities for a tool to enable observability. We'll start by looking at how structured events can be stitched together to create traces.

Chapter 6. Stitching Events into Traces

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 6th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

In the previous chapter, we explored why events are the fundamental building blocks of an observable system. Within the last decade, distributed tracing has become an indispensable troubleshooting tool for distributed systems engineering teams. In this chapter, we demystify distributed tracing by examining its core components and why they are so useful for observable systems.

Distributed traces are simply an interrelated series of events. Distributed tracing systems provide packaged libraries that automagically create and manage the work of tracking those relationships. The concepts used to create and track the relationships between discrete events apply far beyond traditional tracing use cases. In order to further explore what's possible with observable systems, we must first explore the inner workings of tracing systems.

In this chapter, we'll cover more about what distributed tracing is and why it matters. We explain the components of a trace and use code examples to

illustrate the steps necessary to assemble a trace by hand and how those components work. We'll see examples of adding additional and relevant data to a trace event (or span) and why you may want that data. Finally, having seen how a trace is assembled manually, we'll apply those same techniques to non-traditional tracing use cases (like stitching together log events) that are possible with observable systems.

Distributed tracing and why it matters now

Tracing is a term used to describe a fundamental software debugging technique wherein various bits of information are logged throughout a program's execution for the purpose of diagnosing problems. Since the very first day that two computers were linked together to exchange information with one another, software engineers have been discovering the gremlins lurking within our programs and protocols. Those issues persist despite our best efforts and in an age where distributed systems are the norm, the debugging techniques we use must adapt to meet more complex needs.

Distributed tracing is a method of tracking the progression of a single request—called a "trace"—as it is handled by various services that comprise an application. Tracing in this sense is "distributed" because in order to fulfill its functions a singular request must often traverse process, machine, and network boundaries. The popularity of microservice architectures have led to a sharp increase for debugging techniques that pinpoint where failures occur along that route and what might be contributing to poor performance. But any time that a request traverses bounds—such as from on-premises to cloud infrastructure, or from infrastructure you control to SaaS services you don't and back again—distributed tracing can be incredibly useful to diagnose problems and optimize code.

The rise in popularity of distributed tracing also means that several approaches and competing standards for accomplishing that task have emerged. Distributed tracing first gained mainstream traction after Google's publication of the Dapper paper in 2010. Two notable open-source tracing

projects emerged shortly after: Twitter's Zipkin in 2012 and Uber's Jaeger in 2017, in addition to several commercially available solutions such as Honeycomb or Lightstep.

Despite the differences in implementation, the core methodology and the value it provides are the same. As explored in the first section of this book, modern distributed systems have a tendency to scale into a tangled knot of dependencies. Distributed tracing is valuable because it clearly shows the relationships between various services and components in a distributed system. In an observable system, a trace is simply a series of interconnected events.

To understand how traces relate to the fundamental building blocks of observability, let's start by looking at how traces are assembled.

The components of tracing

To understand the mechanics of how tracing works in practice, we'll use an example to illustrate the various components needed to collect the data necessary for a trace. First, we'll consider the outcome we want from a trace (to clearly see relationships between various services). Then we'll look at how we might modify our existing code to get that outcome.

Traces help you understand system interdependencies. Those interdependencies can obscure problems and make them particularly difficult to debug unless the relationships between them are clearly understood. For example, if a downstream database service experiences performance bottlenecks that latency can cumulatively stack up. By the time that latency is detected three or four layers upstream, it can be incredibly difficult to identify which component of the system is the root of the problem because now that same latency is being seen in dozens of other services.

To quickly understand where bottlenecks may be occurring, it's useful to have waterfall-style visualizations of a trace where each stage of a request is displayed as an individual chunk in relation to the start time and duration of a request being debugged.

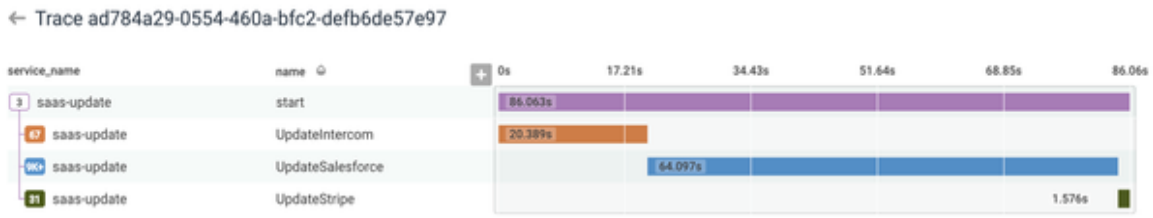


Figure 6-1. This waterfall style trace visualization displays four different trace spans during one request.

Each chunk of this waterfall is called a trace span, or **span** for short. Within any given trace, spans are either the **root span**—i.e. the first span in that trace—or they are nested within the root span. Spans nested within the root span may also have nested spans of their own. That relationship is sometimes referred to as **parent-child**. For example, if Service A calls Service B which calls Service C, then for that trace Span A is the parent of Span B which is the parent of Span C.

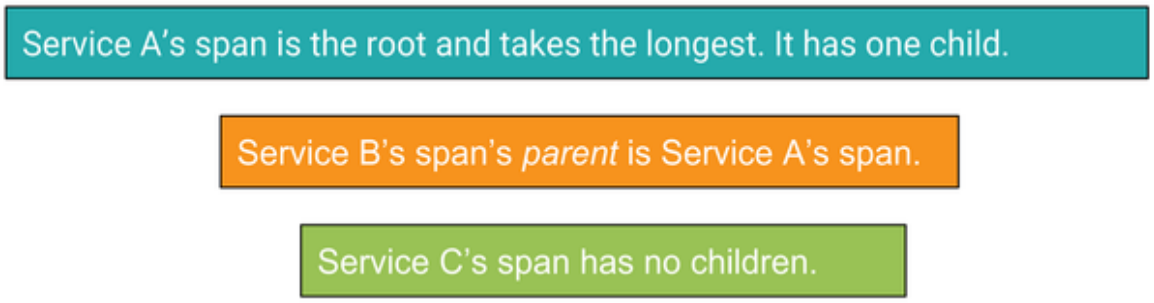


Figure 6-2. A trace that has two parent spans. Span A is the root span, and it is also the parent of Span B. Span B is a child of Span A, and it is also the parent of Span C. Span C is a child of Span B and has no child spans of its own.

Note that a service might be called and appear multiple times within a trace as separate spans, such as in the case of circular dependencies or intense calculations broken up into parallel functions within the same service hop. In practice, requests often traverse messy and unpredictable paths through a distributed system. In order to construct the view we want for any path taken, no matter how complex, there are five pieces of data we need to have for each component.

- **Trace ID** - We need a unique identifier for the trace we're about to create so that we can map it back to a particular request. This ID is

created by the root span and propagated throughout each step taken to fulfill the request.

- **Span ID** - We also need a unique identifier for each individual span created. Spans contain information captured while a unit of work occurred during a single trace. The unique ID allows us to refer back to this span whenever we need it.
- **Parent ID** - This field is used to properly define nesting relationships throughout the life of the trace. A Parent ID is absent in the root span (that's how we know it's the root).
- **Timestamp** - Each span must indicate when its work began.
- **Duration** - Each span must also record how long that work took to finish.

Those fields are absolutely required in order to assemble the structure of a trace. However, you will likely find a few other fields helpful when identifying what these spans are or how they relate to your system. Any additional data added to a span is essentially a series of tags. Common examples of helpful fields include things like:

- **Service Name** - For investigative purposes, you'll want to indicate things like the name of the service where this work occurred.
- **Span Name** - To understand the relevancy of each step, it's helpful to give each span a name that identifies or differentiates the work that was being done—e.g. names could be `intense_computation1` and `intense_computation2` if they represent different work streams within the same service or network hop.

With that data, we should be able to construct the type of waterfall visualization we want for any request in order to quickly diagnose any issues. Next, let's look at how we would instrument our code to generate that data.

Instrumenting a trace the hard way

To understand how the core components of a trace come together, we'll create a manual example of an overly simple tracing system. In any distributed tracing system, there's quite a bit more happening in order to make your data useful. For our purposes, we'll presume that a backend for collection of this data (see: Chapter 16) already exists and we will just focus on the client-side instrumentation necessary for tracing. We'll also presume that we can just send data to that system via HTTP.

If you wish to make an apple pie from scratch, you must first invent the universe.

—Carl Sagan

Let's say that we have a very simple web endpoint. For quick illustrative purposes, we will create this example with Go as our language. When we issue a GET request, it makes calls to two other services to retrieve data based on the payload of the request (such as whether the user is authorized to access the given endpoint) and then it returns the results:

```
func rootHandler(r *http.Request, w http.ResponseWriter) {
   authorized := callAuthService(r)
   name := callNameService(r)
   if authorized {
       w.Write([]byte(fmt.Sprintf(
       `{"message": "Waddup %s"}`,
       name)))
   } else {
       w.Write([]byte(
       `{"message": "Not cool dawg"}`
       ))
   }
}
```

The main purpose of distributed tracing is to follow a request as it traverses multiple services. In this example, because this request makes calls to two other services, we would expect to see a minimum of three spans[1] when making this request:

1. The originating request to rootHandler
2. The call to our authorization service (to authenticate the request)
3. The call to our name service (to get the user's name)

First, let's generate a **trace ID** so that we can group any subsequent spans back to the originating request. We'll use UUIDs to avoid any data duplication issues and store the attributes and data for this span in a map (we could then later serialize that data as JSON to send it to our data backend). We'll also generate a **span ID** that can be used as unique identifiers to later retrieve any particular span.

```
func rootHandler(...) {
   traceData := make(map[string]interface{})
   traceData["trace.trace_id"] = uuid.String()
   traceData["trace.span_id"] = uuid.String()

   // ... main work of request ...
}
```

Now that we have IDs that can be used to string our requests together, we'll also want to know when this span started and how long it took to execute. We will do that by capturing a **timestamp,** both when the request starts and when it ends. Noting the difference between those two timestamps, we will calculate **duration** in milliseconds.

```
func rootHandler(...) {
   // ... trace id setup from above ...

   startTime := time.Now()
   traceData["timestamp"] = startTime.Unix()
   // ... main work of request ...
   traceData["duration_ms"] = time.Now().Sub(startTime)
}
```

Finally, we'll add two descriptive fields: **service name** indicates which service the work occurred in and **span name** indicates the type of work we did. Additionally, we'll set up this portion of our code to send all of this data to our tracing backend via RPC once it's all complete.

```
func rootHandler(...) {
   // ... trace id and duration setup from above ...
   traceData["name"] = "/"
   traceData["service_name"] = "root"
   // ... main work of request ...

   sendSpan(traceData)
}
```

We have the portions of data we need for this one singular trace span. However, we don't yet have a way to relay any of this trace data to the other services that we're calling as part of our request. At the very least, they need to know which trace they're part of and which span called them and that data should be propagated throughout the life of the request.

The most common way that information is shared in distributed tracing systems is by setting it in HTTP headers on outbound requests. In our example, we could expand our helper functions `callAuthService` and `callNameService` to accept the `traceData` map and use it to set special HTTP headers on their outbound requests.

You could call those headers anything you want, as long as the programs on the receiving end understand those same names. Typically, HTTP headers will follow a particular standard (such as W3C or B3). For our example, we'll use the B3 standard. We would then send the following headers to ensure child spans are able to build and send their spans correctly:

- X-B3-TraceId - Contains the **trace ID** for the entire trace (from above)
- X-B3-ParentSpanId - Contains the current **span ID**, which will be set as the **parent id** in the child's generated span

```
func callAuthService(originalRequest *http.Request,
traceData map[string]interface{}) {
   req, _ = http.NewRequest("GET",
"http://authz/check_user", nil)
   req.Header.Set("X-B3-TraceId",
```

```
traceData["trace.trace_id"])
    req.Header.Set("X-B3-ParentSpanId",
traceData["trace.span_id"])
    // ... make the request ...
}
```

We would also make a similar change to our `callNameService` function. With that, when each service is called, it can pull the information from these headers and add them to `trace.trace_id` and `trace.parent_id` in their own generation of `traceData`. Each of those services would also send their generated spans to the tracing backend. On the backend, those traces are stitched together to create the waterfall type visualization we want to see.

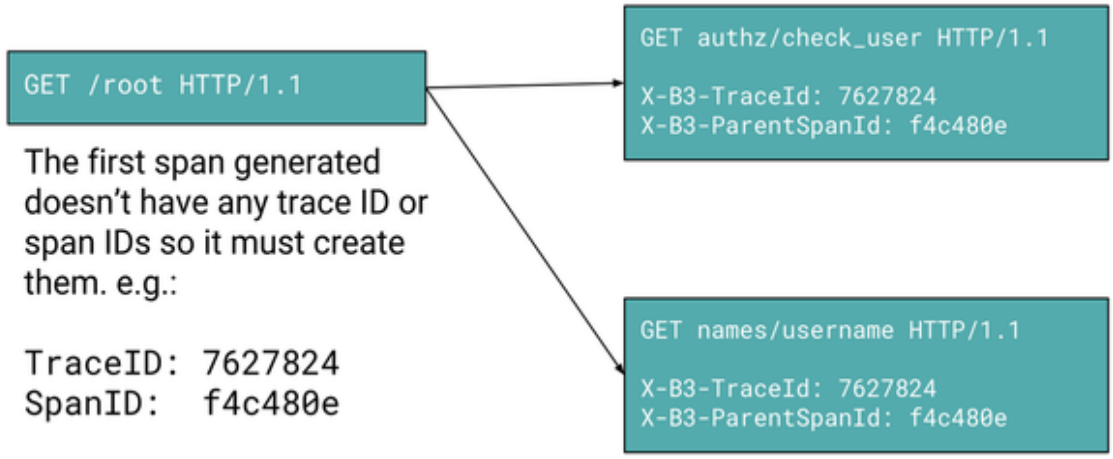


Figure 6-3. Our example app would now propagate traceID and parentID to each child span.

Now that we've seen what goes into instrumenting and creating a useful trace view, let's see what else we might want to add to our spans to make them more useful for debugging.

Adding custom fields into trace spans

Understanding parent-child relationships and execution duration is a good start. But there are other fields you may want to add in addition to the

necessary trace data to better understand what's happening in each span whose operation is typically buried deeply within your distributed systems.

For example, in addition to the service name it might be useful to know the exact host on which the work was executed and if it was related to a particular user. Let's modify our example to capture those details as part of our trace span by adding them as key value pairs.

```
hostname, _ := os.Hostname()
traceData["tags"] = make(map[string]interface{})
traceData["tags"]["hostname"] = hostname
traceData["tags"]["user_name"] = name
```

You could further extend this example to capture any other system information you might find relevant for debugging such as application build_id, instance_type, information about your runtime, or any plethora of details like any of the examples in the last chapter. For now, we'll keep it simple.

Putting this all together, our full example app that creates a trace from scratch would look something like this (the code is repetitive and verbose for clarity):

```
func rootHandler(r *http.Request, w http.ResponseWriter) {
   traceData["tags"] = make(map[string]interface{})
   hostname, _ := os.Hostname()
   traceData["tags"]["hostname"] = hostname
   startTime := time.Now()
   traceData["timestamp"] = startTime.Unix()
   traceData := make(map[string]interface{})
   traceData["trace.trace_id"] = uuid.String()
   traceData["trace.span_id"] = uuid.String()
   traceData["name"] = "/"
   traceData["service_name"] = "root"
   func callAuthService(originalRequest *http.Request, traceData
map[string]interface{}) {
      req, _ = http.NewRequest("GET", "http://authz/check_user",
nil)
      req.Header.Set("X-B3-TraceId", traceData["trace.trace_id"])
      req.Header.Set("X-B3-ParentSpanId",
traceData["trace.span_id"])
      // ... make the auth request ...
```

```
    }
    func callNameService(originalRequest *http.Request, traceData
map[string]interface{}) {
        req, _ = http.NewRequest("GET", "http://authz/check_user",
nil)
        req.Header.Set("X-B3-TraceId", traceData["trace.trace_id"])
        req.Header.Set("X-B3-ParentSpanId",
traceData["trace.span_id"])
        // ... make the name request ...
    }
    authorized := callAuthService(r)
    name := callNameService(r)
    if authorized {
        w.Write([]byte(fmt.Sprintf(
        `{"message": "Waddup %s"}`,
        name)))
    } else {
        w.Write([]byte(
        `{"message": "Not cool dawg"}`
        ))
    }
    traceData["duration_ms"] = time.Now().Sub(startTime)
    sendSpan(traceData)
}
```

This section is a bit contrived in order to illustrate how these concepts come together in practice. The good news is that you would typically not have to generate all of this code yourself. Distributed tracing systems—such as OpenTelemetry, covered in the next chapter—commonly have their own supporting libraries to do most of the boilerplate setup work we outlined in this example.

For example, Honeycomb provides instrumentation libraries (called "Beelines") for most languages. In the Honeycomb Go Beeline, the above example is much simpler to implement. After initializing the Beeline with your Honeycomb write key, you would simply wrap the Go HTTP muxer to create spans whenever an API call is received.

```
http.ListenAndServe(":8080", hnynethttp.WrapHandler(muxer))
```

In the case of Beelines, the custom fields we added above are not included out-of-the-box. But adding them is also simpler.

```
func rootHandler(r *http.Request, w http.ResponseWriter) {
   // ctx contains the Beeline-generated span
   ctx := r.Context()
   beeline.AddField(ctx, "hostname", hostname)
   beeline.AddField(ctx, "user_name", name)
}
```

Shared libraries like this are typically unique to the particular needs of the tracing solution you wish to use. In the next chapter, we'll look at the open-source OpenTelemetry project and how it enables you to instrument once for a wide variety of solutions rather than re-instrumenting each time you wish to try a different solution.

Now that we have a complete view of what goes into instrumenting and creating useful trace views, let's apply that to what we learned in the previous chapter to understand why tracing is such a key element in observable systems.

Stitching events into traces

In the last chapter, we examined the use of structured events as the building blocks of an observable system. We defined an event as a record of everything that occurred while one particular request interacted with your service. During the lifetime of that request, any interesting details about what occurred in order to return a result should be appended to the event.

In an observable system, traces are simply an interrelated series of events. In the example above, our functions were incredibly simple. In a real application, each service call made throughout the execution of a single request would have its own interesting details about what occurred: variable values, parameters passed, results returned, associated context, etc. Each event would capture those details along with the `traceData` that later allows you to see and debug the relationships between various services and components in your distributed systems.

In the example above—as well as in distributed tracing systems in general—the instrumentation we used was added at the remote-service-call level.

However, in an observable system, you could use the same approach to tie together any number of correlated events from different sources. For example, you could take a first step toward observability by migrating your current single-line logging solution toward a more cohesive view.

First, you could migrate from generating unstructured multi-line logs to generating structured logs (see: Chapter 5). You could then add the same required fields for `traceData`, to your structured logs. Having done so, you could generate the same waterfall-style view from your log data. We wouldn't recommend that as a long-term approach, but it can be useful especially when first getting started (see Chapter 9 for more tips on getting started).

A more common scenario for a non-traditional use of tracing is to do a chunk of work that is not distributed in nature, but that you want to split into its own span for a variety of reasons. For example, perhaps you find that your application is bottlenecked by JSON unmarshalling (or some other CPU-intensive operation) and you need to identify when that causes a problem.

One approach is to wrap these "hot blocks" of code into their own separate spans to get an even more detailed waterfall view (see: Chapter 15 for more examples). Building on our Go-based examples, using Beelines that can be accomplished using the context provided (or equivalent in other languages) to call `startSpan`, then send that span when it's all done.

```
ctx, span := beeline.StartSpan("slowCodeBlock", ctx)
if err := slowCodeBlock(ctx); err != nil {
   beeline.AddField(ctx, "error.detail", err)
}
span.Send()
```

That approach can be used to create traces in non-distributed (i.e. monolithic) or non-service-oriented programs. For example, you could create a span for every chunk of work in a batch job (e.g. every object uploaded to AWS S3) or for each distinct phase of an AWS Lambda-based pipeline.

In an observable system, any set of events can be stitched into traces. Tracing doesn't have to be limited to service-to-service calls. So far, we've focused only on gathering the data in these events and sending them to our observability backend. In later chapters, we'll look at the analysis methods that allow you to arbitrarily slice and dice that data to find patterns along any dimension you choose.

Conclusion

Events are the building blocks of observability and traces are simply an interrelated series of events. The concepts used to stitch together spans into a cohesive trace are useful in the setting of service-to-service communication. Those same concepts can also be applied beyond making remote procedure calls to any discrete events in your systems that are interrelated (like individual file uploads all created from the same batch job).

In this chapter, we instrumented a trace the hard way by coding each necessary step by hand. A more practical way to get started with tracing is to use an instrumentation framework. In the next chapter, we'll look at the open-source OpenTelemetry project as well as how and why you would use it to instrument your production applications.

---

1 You could see more than three spans in some scenarios. For additional context on why you might spawn additional spans, see Chapter 15.

Chapter 7. Analyzing Events to Achieve Observability

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 8th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

In the first two chapters of this section, you've learned about telemetry fundamentals that are necessary to create a data set that can be properly debugged with an observability tool. While having the right data is a fundamental requirement, observability is measured by what you can learn about your systems from that data. This chapter explores debugging techniques applied to observability data and what separates them from traditional techniques used to debug production systems.

We'll start by closely examining common techniques for debugging issues with traditional monitoring and application performance monitoring tools. As highlighted in previous chapters, traditional approaches presume a fair amount of familiarity with previously known failure modes. That approach is unpacked so that it can then be contrasted with debugging approaches that don't require the same degree of system familiarity to identify issues. We'll then look at how those debugging techniques can be automated and consider the roles that both humans and computers play in creating effective

debugging workflows. When combining those factors, you'll understand how observability tools help you analyze telemetry data to identify issues that are impossible to detect with traditional tools.

Debugging from known conditions

Prior to observability, system and application debugging mostly occurred by building upon what you know about a system. This can be observed when looking at how the most senior members of an engineering team approach troubleshooting. It can seem downright magical when they know which questions are the right ones to ask and how they instinctively know where the right place is to look. That magic is born from intimate familiarity with their application and systems.

To capture this magic, managers urge their senior engineers to write detailed runbooks in an attempt to capture and solve every possible "root cause" they might encounter out in the wild. In Chapter 2, we covered the escalating arms race of dashboard creation embarked upon to create just the right view that identifies a newly encountered problem. But that time spent creating runbooks and dashboards is largely wasted, because modern systems rarely fail in precisely the same way twice. And in the cases where they do, it's increasingly common to configure an automated remediation that can correct that failure until someone can investigate it properly.

Anyone who has ever written or used a runbook can tell you a story about just how woefully inadequate they are. Perhaps they work to temporarily address technical debt: there's one recurring issue and the runbook tells other engineers how to mitigate the problem until the upcoming sprint when it can finally be resolved. But more often, especially with distributed systems, a long thin tail of problems that almost never happen are what will be responsible for cascading failures in production. Or, five seemingly impossible conditions will align just right to create a large-scale service failure in ways that might only happen once every few years.

Yet engineers typically embrace that dynamic as just the way that troubleshooting is done. Because that is how the act of debugging has

worked for decades. First, you must intimately understand all parts of the system—whether through direct exposure and experience, documentation, or a runbook. Then you must reason your way through possible places and ways that the discrete part you're concerned with could have met a known failure mode.

Even after instrumenting your applications to emit observability data, you might still be debugging from known conditions. For example, you could take that stream of arbitrarily wide events and pipe it to tail -f and grep it for known strings, like troubleshooting with unstructured logs is done today. Or you could take query results and stream them to a series of infinite dashboards, like troubleshooting with metrics is done today.

It's not the fact that you're now collecting event data that gives you an ability to debug unknown conditions. It's how you approach the act of debugging.

At Honeycomb, it took us quite some time to figure this out, even after we'd built our own observability tool. Our natural inclination as engineers is to jump straight to what we know about our systems. In Honeycomb's early days, before we learned how to break away from debugging from known conditions, what observability helped us do was to just jump to the right high-cardinality questions to ask.

If we'd just shipped a new front-end feature and were worried about performance, we'd ask, "how much did it change our css and js asset sizes?" Having just written the instrumentation ourselves, we would know to figure that out by calculating maximum *css_asset_size* and *js_asset_size*, then breaking down performance by *build_id*. If we were worried about a new customer who was starting to use our services, we'd ask, "are their queries fast?" Then we would just know to filter by *team_id* and calculate *p95 response time*.

But what happens when you don't know what's wrong or where to start looking, and haven't the faintest idea of what could be happening? When debugging conditions are completely unknown to you, then you must instead debug from first principles.

Debugging from first principles

As laid out in the previous two chapters, gathering telemetry data as events is the first step. Achieving observability also requires that you unlock a new understanding of your system by analyzing that data in powerful and objective ways. Observability enables you to debug your applications from first principles.

A first principle[1] is a basic assumption about a system that was not deduced from another assumption. In philosophy, a first principle is defined as the first basis from which a thing is known. To "debug from first principles" is basically a methodology to follow in order to understand a system scientifically. Proper science requires that you do not assume anything. You must start by questioning what has been proven and what you are absolutely sure is true. Then, based on those principles, you must form a hypothesis and validate or invalidate it based on observations about the system. Debugging from first principles is a core capability of observability.

In Chapter 2 and Chapter 3, you saw examples of elusive issues that were incredibly difficult to diagnose. What happens when you don't know a system's architecture like the back of your hands? Or what happens when you don't even know what data is being collected about this system? What if the source of an issue is multi-causal: you're wondering what went wrong and the answer is 13 different things?

The real power of observability is that you shouldn't have to know so much in advance of needing to debug an issue. You should be able to systematically and scientifically take one step after another, to methodically follow the clues to find the answer, even when you are unfamiliar with the system. The magic of instantly jumping to the right conclusion by inferring an unspoken signal, relying on past scar tissue, or making some leap of familiar brilliance is instead replaced by methodical, repeatable, verifiable process.

Putting that approach into practice is demonstrated with the core analysis loop.

The core analysis loop

Debugging from first principles begins when you are made aware of the fact that something is wrong (we'll look at alerting approaches that are compatible with observability in Chapter 11). Perhaps you received an alert, but this could also be something as simple as receiving a customer complaint: you know that something is slow, but you do not know what is wrong.

The core analysis loop works like this:

1. Start with an overall view of what prompted your investigation: what did the customer or alert tell you? Start there.
2. Verify if what you know so far is true: is there a notable change in performance happening somewhere in this system? Data visualizations can help you identify changes of behavior as a change in curve somewhere in the graph.
3. Now, search for dimensions that might drive that change in performance. Approaches to accomplish that might include:
    a. Examining sample rows from the area that shows the change: are there any outliers in the columns that might give you a clue?
    b. Slicing those rows across various dimensions looking for patterns: do any of those views highlight distinct behavior across one or more dimensions?
    c. Filtering for particular dimensions or values within those rows to better expose potential outliers.
4. Do you now know enough about what might be occurring? If so, you're done! If not, filter your view to isolate this area of performance as your next starting point. Then return to step 3.

This is the basis of debugging from first principles. You can use this loop as a brute force method to cycle through all available dimensions to identify

which ones explain or correlate with the outlier graph in question, with no prior knowledge or wisdom about the system required.

Of course, that brute force method could take an inordinate amount of time and make such an approach impractical to leave in the hands of human operators alone. An observability tool should automate as much of that brute force analysis for you as possible.

Automating the brute force portion of the core analysis loop

The core analysis loop is a method to objectively find a signal that matters within a sea of otherwise normal system noise. Leveraging the computational power of machines becomes necessary in order to swiftly get down to the bottom of issues.

When debugging slow system performance, the core analysis loop has you isolate a particular area of system performance that you care about. Rather than manually searching across rows and columns to coax out patterns, an automated approach would be to retrieve the values of all dimensions both inside the isolated area (the anomaly) and outside the area (the system baseline), diff them, and then sort by difference. Very quickly, that lets you see a list of things that are different between what your investigation is concerned with and everything else.

For example, you might isolate a spike in request latency and, when automating the core analysis loop, get back a sorted list of dimensions and how often they appear within this area. You might see that:

- request.endpoint with value batch is in 100% of requests in the isolated area, but only in 20% of the baseline area
- handler_route with value /1/markers/ is in 100% of requests in the isolated area, but only 10% of the baseline area
- request.header.user_agent is populated in 97% of requests in the isolated area, but 100% of the baseline area

- etc

At a glance, this tells you that the events in this specific area of performance you care about are different from the rest of the system in all of these ways, whether that be one deviation or dozens. Let's look at a more concrete example of the core analysis loop using Honeycomb's BubbleUp feature.

With Honeycomb, you start by visualizing a heatmap to isolate a particular area of performance you care about. The way Honeycomb automates the core analysis loop is with the BubbleUp feature: you point and click at the spike or anomalous shape in the graph that concerns you, and draw a box around it. BubbleUp then computes the values of *all* dimensions both inside the box (the anomaly you care about and want to explain) and outside the box (the baseline), then it compares the two and sorts the resulting view by percent of differences.

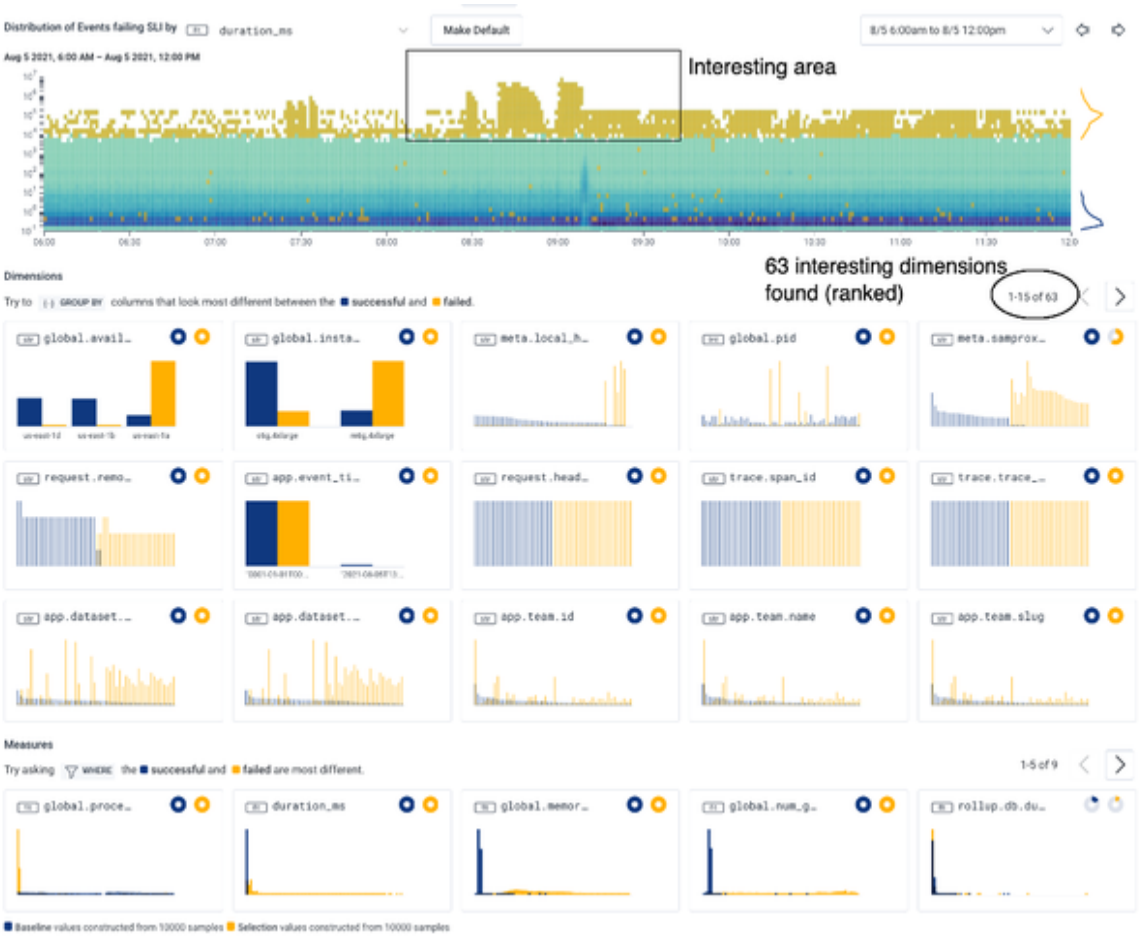


Figure 7-1. shows a Honeycomb heatmap of event performance during a real incident. BubbleUp surfaces results for 63 interesting dimensions and ranks the results by largest percentage difference.

In this example, we're looking at an application with high-dimensionality instrumentation that BubbleUp can compute and compare. The results of the computation are shown in histograms using two primary colors: blue for baseline dimensions and orange for dimensions in the selected anomaly area. In the top left-corner (the top results in the sort operation), we see a field named global.availability_zone with a value of us-east-1a only showing up in 17% of baseline events, but showing up in 98% of anomalous events in the selected area.

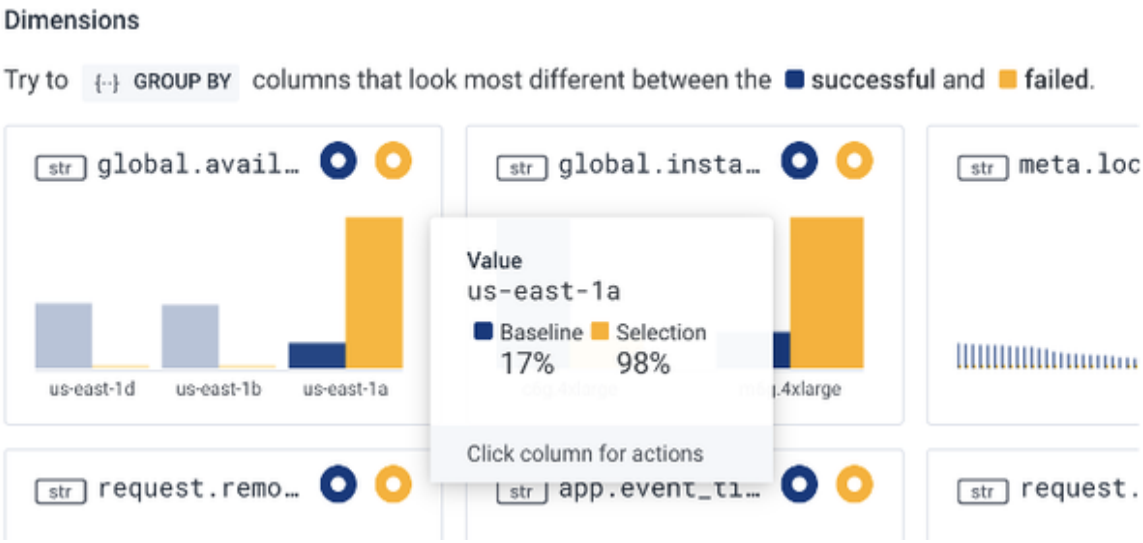


Figure 7-2. shows a close look at the results from the figure above. The orange bar shows that global.availability_zone appears as us-east-1a in 98% of events in the selected area, vs. the blue bar which shows that's only the case in 17% of baseline events.

In this example, you can quickly see that slow performing events are mostly originating from one particular availability zone from our cloud infrastructure provider. The other information automatically surfaced also points out one particular virtual machine instance type that appears to be more affected than others. Other dimensions surfaced, but in this example the differences tended to be less stark; indicating that they were perhaps not as relevant to our investigation.

This information has been tremendously helpful: we now know what conditions appear to be triggering slow performance. A particular type of instance in one particular availability zone is much more prone to very slow performance than other infrastructure we care about. In that particular situation, the glaring difference pointed to what turned out to be an underlying network issue with our cloud provider's entire availability zone.

Not all issues are as immediately obvious as this underlying infrastructure issue. Often, you may need to look at other surfaced clues to triage code related issues. The core analysis loop remains the same and you may need to slice and dice across different dimensions until one clear signal emerges, similar to what is seen in the example above. In this case, we contacted our cloud provider and were also able to independently verify the unreported

availability issue when our customers also reported similar issues in the same zone. If this had instead been a code related issue, then we might decide to reach out to those users, or figure out what path they followed through the UI to see those errors, and fix the interface, or the underlying system.

The core analysis loop can be done manually to uncover these interesting dimensions. But in a modern era where computing resources are cheap, an observability tool should automate that investigation for you. Because debugging from first principles does not require prior familiarity with the system; that automation can be done simply and methodically, without the need to seed "intelligence" about the application being debugged.

Note that the core analysis loop is something that can only be achieved using the baseline building blocks of observability, which is to say arbitrarily-wide structured events. You cannot achieve this with metrics -- they lack the broad context to let you slice and dice and dive up or down in the data. You cannot achieve this with logs unless you have correctly appended all the request id, trace id, and other headers, and then done a great deal of postprocessing to reconstruct them into events -- and then added the ability to aggregate them at read time and perform complex custom querying.

In other words: yes, an observability tool should automate much of the number crunching for you, but even the manual core analysis loop is unattainable without the basic building blocks of observability.

This misleading promise of AIOps

Since Gartner coined the term in 2017, Artificial Intelligence for Operations (or AIOps) has generated lots of interest from companies seeking to somehow automate common operational tasks. Delving deeply into the misleading promise of AIOps is beyond the scope of this book. But there are two operational considerations where AIOps intersects observability: reducing alert noise and anomaly detection.

In Chapter 11 (Using Service Level Objectives for reliability), we unpack a simpler approach to alerting that makes the need for algorithms to reduce alert noise a moot point. The second intersection, anomaly detection, is the focus of this chapter so we'll unpack that in this section.

As seen earlier in this chapter, the concept behind using algorithms to detect anomalies is to select a baseline of "normal" events and compare those to "abnormal" events not contained within the baseline window. Selecting the window in which that's done can be incredibly challenging for automated algorithms.

Similar to using BubbleUp to draw a box around the area you care about, AI must decide where to draw its own box. If a system behaved consistently over time, anomalies would be worrisome unusual cases that could be easily detected. In an innovative environment where system behavior changes frequently, it's more likely that AI will select the wrong size area around which to draw that box. Either the box will be too small and identify a great deal of perfectly normal behavior as anomalies, or it will be too large and miscategorize anomalies as normal behavior. In practice, both types of mistakes will be made and detection will be far too noisy or far too quiet.

This book is about the engineering principles needed to manage running production software on modern architectures with modern practices. Any reasonably competitive company in today's world will have engineering teams frequently deploying changes to production. New feature deployments introduce changes in performance that didn't previously exist: that's an anomaly. Fixing a broken build: that's an anomaly. Introducing service optimizations in production that change the performance curve: that's an anomaly too.

AI technology isn't magic. AI can only help if there are clearly discernible patterns and if it can be trained to use ever changing baselines to model its predictions: a training pattern that, so far, has yet to emerge in the AIOps world.

In the meantime, there is an intelligence designed to reliably adapt its pattern recognition to ever changing baselines by applying real-time context to a new problem set: human intelligence. Human intelligence and contextual awareness of a problem to be solved can fill in the gaps where AIOps techniques fall short. Similarly, that adaptive and contextual human intelligence lacks the processing speed achieved by applying algorithms over billions of rows of data.

It's in observability and automating the core analysis loop that both human and machine intelligence merge to get the best of both worlds. Let computers do what they do best: churn through massive sets of data to identify patterns that might be interesting. Let the humans do what they do best: add cognitive context to those potentially interesting patterns that reliably sifts out the signals that matter from the noises that don't.

Humans alone can't solve today's most complex software performance issues. But neither can computers or vendors touting AIOps as a magical silver bullet. Leveraging the strengths of humans and machines, in a combined approach, is the best solution in today's world. Automating the core analysis loop is a prime example of how that can be done.

Conclusion

Collecting the right telemetry data is only the first step in the journey toward observability. That data must be analyzed according to first principles in order to objectively and correctly identify application issues in complex environments. The Core Analysis Loop is an effective technique for fast fault localization. However, that work can be time consuming for humans to conduct methodically as a system becomes increasingly complex.

When looking for the sources of anomalies, you can leverage compute resources to quickly sift through very large datasets to identify interesting patterns. Surfacing those patterns to a human operator, who can put them into the necessary context and then further direct the investigation, strikes an effective balance that best utilizes the strengths of machines and humans

to quickly drive system debugging. Observability systems are built to apply this type of analysis pattern to the event data you've learned how to collect in the previous chapters.

Now that you understand both the types of data needed and the practice of analyzing that data to get fast answers, the next chapter looks at how you can begin introducing observability practices within your organization.

---

1 https://en.wikipedia.org/wiki/First_principle

Chapter 8. How Observability and Monitoring Come Together

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 9th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

So far in this book we've examined the differentiating capabilities of observable systems, the technological components necessary for observability, and how observability fits into the technical landscape. Observability is fundamentally distinct from monitoring, and both serve different purposes. In this chapter, we examine how they fit together and what the considerations are to determine how both coexist within your organization.

Most organizations have years—if not decades—of accumulated metrics data and monitoring expertise set up around their production software systems. As covered in earlier chapters, traditional monitoring approaches are adequate for traditional systems. But when managing modern systems, does that mean you should throw that all away and start fresh with observability tools? Doing that would be a rather cavalier and brash approach. The truth for most organizations is that their approach should be dictated by their adopted responsibilities.

This chapter explores how observability and monitoring come together by examining the strengths of each, the domains to where they are best suited, and how they complement one another. Every organization is different and a recipe for how observability coexists with monitoring cannot be universally prescribed. But by considering your workloads, you can figure out how the two come together best for you.

Where Monitoring Fits

In Chapter 2, we examined how observability differs from monitoring. That chapter mostly focuses on the shortcomings of monitoring systems and how observability fills in those gaps. But monitoring systems still continue to provide valuable insights. Let's start by examining where monitoring is the right tool for the job.

The traditional monitoring approach to understanding system state is a very mature and well-developed process. Decades of iterative improvement have evolved monitoring tools beyond their humble beginnings with simple metrics and round-robin databases (RRDs) toward time-series databases (TSDBs) and elaborate tagging systems. A wealth of sophisticated options also exist to provide this service; from open source software solutions, to startups, to publicly traded companies.

Monitoring practices are well-known and widely understood beyond the communities of specialists that form around specific tools. Across the software industry, there are monitoring best practices that anyone who has operated software in production can likely agree upon.

For example, a widely accepted core tenet is that a human doesn't need to sit around watching graphs all day: the system should proactively inform its users when something is wrong. However, this is the exact *opposite* of its counterpart best practice for observability, which states that engineers should always watch their code as it is deployed, and should spend time every day exploring their code in production, watching users use it, and looking around for outliers and curious trails to follow. Different best practices, different tools; different purposes.

Monitoring systems and metrics have evolved to optimize themselves for the job to which they're very well suited. They automatically report whether known failure conditions are occurring or about to occur. In other words, they are optimized for reporting on unknown conditions about known failure modes (known unknowns). In contrast, observability is centered around discovering if and why previously unknown failure modes may be occurring: in other words, to discover unknown unknowns.

Those distinctions matter when matching those alternating approaches to their appropriate workloads. The optimization of monitoring systems to find known unknowns means it's a best fit for understanding the state of your infrastructure, which changes less frequently and in more predictable ways. The optimization of observability to find unknown unknowns means it's a best fit for understanding the state of your software, which changes every day and in far less predictable ways.

Infrastructure Considerations vs. Software Considerations

In traditional systems, the distinction between infrastructure and software was clear: bare-metal systems are the infrastructure and everything running inside those systems was the software. Modern systems, and their many higher-order abstractions, have made that distinction less clear for some. Let's start by defining what is considered infrastructure and what is considered software.

For these purposes, "software" is the code you are actively developing that runs a production service which delivers value to your customers. Software is what your business *wants* to run to solve a market problem. "Infrastructure" is an umbrella term for everything else about the underlying runtime that is necessary in order to run that service. Infrastructure is what your business *needs* to run in order to support the software it *wants* to run.

By that definition, infrastructure includes everything from databases (e.g., mysql, mongodb, etc) to compute and storage (containers, virtual machines, etc) to anything and everything else that had to be provisioned and set up before you could deploy and run your software.

The world of cloud computing has potentially made these definitions somewhat difficult to nail down, so let's drill down further. Let's say that in order to run your software, you need to run underlying components like kafka, postfix, haproxy, memcached, or even something like Jira. If you're buying access to those components as a service, then it doesn't count as infrastructure for this definition: you're essentially paying someone else to run it for you. However, if your team is responsible for installing, configuring, occasionally upgrading, and troubleshooting the performance of those components, then that's infrastructure you need to worry about.

Compared to software, infrastructure is a commodity layer that changes infrequently, is focused on a different set of users, and provides a different value. Software—the code you write for your customers to use—is a core differentiator for your business: it is the very reason your company exists today. Software, therefore, has a different set of considerations for how it should be managed.

	Infrastructure	Your Software
Rate of change	Package updates (monthly)	Repo commits (daily)
Predictability	High (stable)	Low (many new features)
Value to your business	Low (cost center)	High (revenue generator)
Number of users	Few (internal teams)	Many (your customers)
Core concern	Is the system or service healthy?	Can each request acquire the resources it needs for end-to-end execution in a timely and reliable manner?
Evaluation perspective	The system	Your customers
Evaluation criteria	Low-level kernel and hardware device drivers	Variables and API endpoint
Functional responsibility	Infrastructure operations	Software Development
Method for understanding	**Monitoring**	**Observability**

With infrastructure, only one perspective really matters: the perspective of the team responsible for its management. The important question to ask with infrastructure is whether the service it provides is essentially healthy. If it's not healthy, then that team must quickly take action to restore it to a healthy condition. The system may be running out of capacity, or an underlying failure may have occurred: a human should be alerted and respond to take action. The conditions that affect infrastructure health change infrequently and they are relatively easier to predict. In fact, well-established practices exist to predict (e.g. capacity planning) and automatically remediate (e.g. auto-scaling) these types of issues. Due to its relatively predictable and slowly changing nature, aggregated metrics are perfectly acceptable to monitor and alert for infrastructure problems.

With software, the perspective that matters most is that of your customers. The underlying infrastructure may be essentially healthy, yet user requests may still be failing for any number of reasons. As covered in earlier chapters, distributed systems make these types of problems harder to detect and understand. Suddenly, the ability to use high-cardinality fields (user id, shopping cart id, etc) as a way to observe a specific customer's experience becomes critical. Especially in the modern world of continuous delivery, where new versions of your code are constantly being deployed, software concerns are always shifting and changing. Observability provides a way to ask appropriate questions that address those concerns in real time.

These two approaches are not mutually exclusive. Every organization will have considerations that fall more into one category than the other. Next, let's look at different ways those two approaches may co-exist, depending on the needs of your organization.

Assessing Your Organizational Needs

Just as infrastructure and software are complementary, so too are the methods for understanding how each behaves. Monitoring best helps engineers understand infrastructure concerns. Observability best helps engineers understand software concerns. Assessing your own organizational

needs means understanding which concerns are most critical to your business.

Observability will help you deeply understand how software you develop and ship is performing when serving your customers. Code that is well-instrumented for observability allows you to answer complex questions about user performance, see the inner workings of your software in production, and gives you the ability to identify and swiftly remediate issues that are easy to miss when only examining aggregate performance.

If your company writes and ships software as part of its core business strategy, you need an observability solution. If, in addition to providing an overall level of acceptable aggregate performance, your business strategy also relies on providing excellent service to a particular subset of high-profile customers, then your need for observability is especially emphasized.

Monitoring will help you understand how well the infrastructure you run in support of that software is doing its job. Metrics-based systems and their associated alerts help you see when capacity limits or known error conditions of underlying systems are being reached.

If your company provides infrastructure to its customers as part of its core business strategy (e.g. an IaaS provider), you will need a substantial amount of monitoring -- low level DNS counters, disk statistics, etc. Infrastructure is business-critical for these organizations, and they need to be expert in the low level systems they surface to customers. However, if providing infrastructure is not a core differentiator for your business, monitoring becomes less critical and you will need to monitor only the high-level services and end-to-end checks, for the most part. There are several considerations for determining just how much less monitoring your business needs.

Companies that run a significant portion of their own infrastructure will need more monitoring. Whether running systems on-premises or with a cloud provider, this consideration is less about where that infrastructure lives and more about operational responsibility. Whether you provision

virtual machines in the cloud or DBA your own databases on-premises, the key factor is whether your team takes on the burden of ensuring infrastructure availability and performance.

Organizations that take on the responsibility of running their own bare-metal systems will need monitoring that examines low-level hardware performance. They will need monitoring to inspect counters for ethernet ports, statistics on hard drive performance, and versions of system firmware. Organizations that outsource hardware-level operations to an IaaS provider won't need metrics and aggregates that perform at that level.

And so it goes, further up the stack. As more operational responsibility is shifted to a third-party, so too are infrastructure monitoring concerns.

Companies that outsource most of their infrastructure to higher-level PaaS providers can likely get away with very little, if any, traditional monitoring solutions. Platforms as a Service (such as Heroku, AWS Lambda, and others) essentially let you pay someone else to do the job of ensuring the availability and performance of the infrastructure that your business *needs to run*, so that it can instead focus on the software it *wants to run*.

Today, your mileage may vary, depending on the robustness of your cloud provider. Presumably, the abstractions are clean enough and high level enough that the experience of removing your dependence on infrastructure monitoring wouldn't be terribly frustrating. But, in theory, all providers are moving to a model that enables that shift to occur.

Exceptions: Infrastructure Monitoring That Can't Be Ignored

There are a few exceptions to this very neat dividing line between monitoring for systems and observability for software. As mentioned earlier, the evaluation perspective for determining how well your software performs is customer experience. If your software is performing slowly, your customers are experiencing it poorly. Therefore, a primary concern for evaluating customer experience is understanding anything that can cause performance bottlenecks. The exceptions to that neat dividing line are any

metrics that directly indicate how your software is interacting with its underlying infrastructure.

From a software perspective, there's very little—if any—value in seeing the thousands of graphs for variables discovered in the /proc filesystem by every common monitoring tool. Metrics about power management and kernel drivers might be useful for understanding low-level infrastructure details, but it gets routinely and blissfully ignored (as it should) by software developers because it indicates little useful information about impact on software performance.

However, higher-order infrastructure metrics like CPU usage, memory consumption, and disk activity are indicative of physical performance limitations. As a software engineer, you should be closely watching these indicators because they can be early warning signals of problems triggered by your code. For instance, you want to know if the deploy you just pushed caused resident memory usage to triple within minutes. Being able to see sudden changes like a jump to twice as much CPU consumption or a spike in disk write activity right after a new feature is introduced can quickly alert you to problematic code changes.

Higher-order infrastructure metrics may or may not be available depending on how abstracted your underlying infrastructure has become. But if they are, you will certainly want to capture them as part of your approach to observability.

Real World Examples

While observability is still a nascent category, there are a few emergent patterns for how monitoring and observability coexist. The examples cited in this section are selections that represent the patterns we've commonly seen among our customers or within the larger observability community, but they are by no means exhaustive or definitive. These approaches are included to illustrate how the concepts described in this chapter play are applied in the real world.

Our first example customer had a rich ecosystem of tools they were using to understand the behavior of their production systems. Prior to making a switch to observability, teams were using a combination of Prometheus for traditional monitoring, Jaeger for distributed tracing, and NewRelic for application performance monitoring. They were looking to improve their incident response times by simplifying their existing multi-tool approach that required making correlations between data captured in three disparate systems. Switching to an observability based approach meant that they were able to consolidate needs and reduce their footprint to a monitoring system and an observability system that co-exist. Software engineering teams at this organization report primarily using observability to understand and debug their software in production. The central operations team still uses Prometheus to monitor their infrastructure. However, software engineers report that they can still refer back to Prometheus when they have questions about the resource usage impacts of their code. They also report that need is infrequent and that they rarely need to use Prometheus to troubleshoot application bottlenecks.

Our second example customer is a relatively newer company that was able to build a greenfield application stack. Their production services primarily leverage serverless functions and SaaS platforms to power their applications and they run almost no infrastructure of their own. Never having had any real infrastructure to begin with, they never started down the path of trying to make monitoring solutions work for their environment. They rely on application instrumentation and observability to understand and debug their software in production. They also export some of that data for longer-term aggregation and warehousing.

Lastly, our third example customer is a mature financial services company undergoing a digital transformation initiative. They have a large heterogeneous mix of legacy infrastructure and applications as well as greenfield applications that are managed across a variety of business units and engineering teams. Many of the older applications are still operating but the teams that originally built and maintained them have long since disbanded or been reorganized into other parts of the company. Many

applications are managed with a mix of metrics-based monitoring approaches and dashboarding capabilities provided by Datadog and various logging tools to search their unstructured logs. The business would not realize much, if any, value from ripping out, rearchitecting, and replacing monitoring approaches that work well for stable and working services. Instead, greenfield applications are being developed for observability instead of using the former approach requiring a mix of monitoring, dashboards, and logging. When new applications make use of company infrastructure, software engineering teams also have access to infrastructure metrics to monitor resource usage impacts. However, some software engineering teams are starting to capture infrastructure metrics in their events in order to reduce their need to use a different system to correlate resource usage with application issues.

Conclusion

The guiding principle for determining how observability and monitoring coexist within your organization should be dictated by the software and infrastructure responsibilities adopted within its walls. Monitoring is best suited to evaluating the health of infrastructure. Observability is best suited to evaluating the health of your software. Exactly how much of each solution will be necessary in any given organization depends on how much management of that underlying infrastructure has been outsourced to third party (aka “cloud”) providers.

The most notable exceptions to that neat dividing line are higher-order infrastructure metrics on physical devices that directly impact software performance like CPU, memory, and disk. Metrics that indicate consumption of these physical infrastructure constraints are critical to understand the boundaries imposed by underlying infrastructure. If these metrics are available from your cloud infrastructure provider, they should be included as part of your approach to observability.

By illustrating a few common approaches to balancing monitoring and observability in complementary ways, you can see how the considerations

outlined throughout this chapter are implemented in the real world by different teams.

Now that we've covered the fundamentals of observability in-depth, the next section of this book goes beyond technology considerations to also explore the cultural changes necessary for successfully adopting observability practices and driving that adoption across different teams.

Chapter 9. Applying observability practices in your team

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 10th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

In the last section of this book, we looked at the fundamentals of observability from a technology perspective: how events form the building blocks, stitching events into traces, the basics of instrumentation, methods for analyzing event data, and how observability and monitoring play complementary roles. This section delves into implementation challenges with a focus on processes that evolve team practices to further observability adoption. In later chapters, we'll examine modern software development patterns unlocked by observability, new ways to achieve effective alerting and reduce noise, deployment patterns that improve with observability, and go deeper into techniques for risk mitigation via instrumentation.

This chapter starts by focusing on tips to help you get started with observability practices. If you're in a leadership role within your engineering team—such as a team lead, a manager, or maybe the resident

observability fangirl/champion—the hardest thing after getting management approval to figure out an observability implementation strategy is knowing where to start.

For us, this is a particularly tricky chapter to write. Having helped many different teams get started down this path, we know that there's no universal recipe for success. How and where you get started will always depend on many factors. As unsatisfying as "it depends" can be for an answer, the truth is that your journey with observability depends on particulars like the problems most pertinent to you and your team, the gaps in your existing tooling, the level of support and buy-in from the rest of your organization, the size of your team, and other such considerations.

Whatever approach works best for you is, by definition, not wrong. The advice in this chapter is not intended to suggest that this is the one true way to get started with observability (there is no singular path!). That said, we have seen a few emergent patterns and, if you are struggling with where to begin, some of these suggestions may be helpful to you. Feel free to pick and choose from any of the tips in this chapter.

Join a community group

Observability is an emerging practice and the approaches are still relatively young, with plenty of exploration left to do. Whenever the practices and technology behind our sociotechnical systems are rapidly evolving, one of the best ways to learn and improve is by participating in a community of people that are struggling with variations on the same themes as you. Community groups connect you with other professionals that can quickly become a helpful network of friends and acquaintances.

As you and your community face similar challenges, you'll have an opportunity to learn so much very quickly just by hanging out in Slack groups and talking to other people that are banging against the same types of problems. Community groups allow you to connect with people beyond your normal circles from a variety of backgrounds. By actively participating and understanding how other teams handle some of the same challenges as

you, you'll make comparative observations and learn from the experiences of others.

Over time, you'll also discover other community members with common similarities in tech stack, team size, organizational dynamics, and so forth. Those connections will give you someone to turn to as a sounding board, for background, or personal experiences with solutions or approaches you might also be considering. Having that type of shared context before you pull the trigger on new experiments can be invaluable. Actively participating in a community group will save you a ton of time and heartbreak.

Participating in a community will also keep you attuned to developments you may have otherwise missed. Different providers of observability tools will participate in different communities to better understand user challenges, gather feedback on new ideas, or just generally get a pulse on what's happening. Participating in a community specific to your observability tool of choice can also give you a pulse on what's happening as it happens.

When joining a community, remember that community relationships are a two-way street. Don't forget to do your share of chopping wood and carrying water: show up and start contributing by helping others first. Being a good community citizen means participating and helping the group for a while before dropping any heavy asks for help. In other words, don't only speak up when you need something from others. Communities are only as strong as you make them.

Start with the biggest pain points

Introducing any new technology can be risky. As a result, new technology initiatives often target small and inconspicuous services as a place to start. Counterintuitively, when it comes to getting started with observability, starting small is one of the bigger mistakes we often see people make.

Observability tools are designed to help you quickly find elusive problems. Starting with an unobtrusive and relatively unimportant service will have the exact opposite effect of proving the value of observability. If you start with a service that already works relatively well, your team will experience all of the work it takes to get started with observability and get none of the benefit.

When spearheading a new initiative, it's important to get points on the board relatively quickly. Demonstrate value, pique curiosity, and garner interest by solving hard or elusive problems right off the bat. Is there a flaky and flappy service that has been waking people up for weeks, yet nobody can figure out the right fix? Start there. Do you have a problem with constant database congestion, yet no one can figure out the cause? Start there. Are you running a service bogged down by inexplicable load that's being generated by a yet-to-be-identified user? That's your best place to start.

Quickly demonstrating value will win over naysayers, create additional support, and further drive observability adoption. Don't pick an easy problem to start with. Pick a hard problem that observability is designed to knock out of the park. Start with that service. Instrument the code, deploy it to production, explore with great curiosity, figure out how to find the answer, and then socialize that success. Show off your solution during your weekly team meeting. Write up your findings and methodologies, then share them with the company. Make sure whoever is on-call for that service knows how to find that solution.

The fastest way to drive adoption is to solve the biggest pain points for teams responsible for managing their production services. Target those pains. Resist the urge to start small.

Buy instead of build

Similar to starting with the biggest pain points, the decision to build your own observability tooling vs. buying a commercially available solution comes down to proving ROI quickly. Later, in Chapter 19, we'll cover the

nuances of making the buy vs. build decision in greater detail. For now, we'll frame this decision by advising that your predilection be toward putting in the least amount of effort to prove the greatest amount of value.

A few unfortunate realities play into this advice. First, there isn't much by way of open-source observability tooling for you to run yourself even if you wanted to. The three leading candidates you'll come across when googling options are Prometheus, the ELK stack (Elasticsearch, Logstash, and Kibana), or Jaeger. These tools each provide a specific valuable solution, but individually they don't meet our definition of observability (see: Chapter 1).

Observability allows you to understand and explain any state your system can get into, no matter how novel or bizarre. You must be able to comparatively debug that bizarre or novel state across all dimensions of system state data, and combinations of dimensions, in an ad hoc manner, without being required to define or predict those debugging needs in advance.

Prometheus is a time series database for metrics monitoring. While Prometheus is arguably one of the most advanced metrics monitoring systems with a vibrant development community, it still operates solely in the world of metrics-based monitoring solutions. It carries with it the inherent limitations of trying to use coarse measures to discover finer grained problems (see: Chapter 2).

The ELK stack focuses on providing a log storage and querying solution. Log storage backends are optimized for plaintext search at the expense of other types of searching and aggregation. While useful when searching for known errors, plaintext search becomes impractical when searching for answers to compound questions like "who is seeing this problem and when are they seeing it?" The analytical capabilities to identify relevant patterns amongst a flood of plaintext logs critical for observability are challenging in an entirely log-based solution (see: Chapter 8).

Jaeger is an event-based distributed tracing tool. Jaeger is arguably one of the most advanced open source distributed tracing tools available today. As

discussed in Chapter 6, trace events are a fundamental building block for observable systems. However, a necessary component is also the analytical capability to determine which trace events are of interest during your investigation. Jaeger has some support for filtering certain data, but it lacks sophisticated capabilities for analyzing and segmenting all of your trace data (see: Chapter 8).

Each of these tools provides different parts of a system view that can be used to achieve observability. The challenges in using them today is either running disparate systems that place the burden of carrying context between them into the minds of their operators, or building your own bespoke solution for gluing those individual components together. Today, no open source tool exists to provide observability capabilities in one out-of-the-box solution. Hopefully, that will not be true forever. But it is today's reality.

Second, engineering cycles are scarce. It takes time to understand how to roll your own observability solution or write enough glue code to piece together individual components to provide the right level of observability for your applications. Unless that is a core business problem your organization is trying to solve, spending time to build a bespoke observability solution ultimately steals time away from using engineering cycles to build core differentiators that provide real business value.

Lastly, whichever direction you decide to take, make sure that what you end up is actually providing you with observability. Don't just take the word of whatever is on the packaging. One unfortunate result of the rise in observability's popularity is misleading labelling. Observability requires an ability to debug system state across all high-cardinality fields, with high dimensionality, in interactive and performant real-time exploration.

The key to getting started with observability is to move quickly and demonstrate value early in the process. Choosing to buy a solution will keep your team focused on solving problems with observability tooling rather than on building their own.

Flesh out your instrumentation iteratively

Properly instrumenting your applications takes time. The automatic instrumentation included with projects like OpenTelemetry are a good place to start. But the highest value instrumentation will be specific to the needs of your individual application. Start with as much useful instrumentation as you can, but plan to develop your instrumentation as you go.

One of the best strategies for rolling out observability across an entire organization is to instrument a painful service or two as you first get started. Use that instrumentation exercise as a reference point and learning exercise for the rest of the pilot team. Once the pilot team is familiar with the new tooling, use any new debugging situation as a way to introduce more instrumentation.

Whenever an on-call engineer is paged about a problem in production, the first thing they should do is instrument problem areas of your application using the new tooling. Use the new instrumentation to figure out where issues are occurring. After the second or third time people take this approach, they usually catch on to how much easier and less time consuming it is to debug issues by introducing instrumentation first. Debugging from instrumentation first allows you to actually see what's really happening.

Once that pilot team is up to speed, they can help others learn. They can provide coaching on creating helpful instrumentation, suggest helpful queries, or point them toward other examples of helpful troubleshooting patterns. Each new debugging issue can be used to build out the instrumentation you need. You don't need to have a fully developed set of instrumentation to get immediate value with observability.

Look for opportunities to leverage existing efforts

One of the biggest barriers to adoption of any new technology is the sunk-cost fallacy. Individuals and organizations commit the sunk cost fallacy when they continue a behavior or endeavor as a result of previously

invested time, money, of effort.[1] How much time, money, and effort has your organization already invested in traditional approaches that are no longer serving your needs?

Resistance to fundamental change often hits a roadblock when the perception of wasted resources creeps in. What about all those years invested into understanding and instrumenting for the old solutions? Although it's a logical fallacy, the feelings behind it are very real and they'll stop your efforts dead in their tracks if you don't do anything about them.

Always be on the lookout for and leap onto any chance to forklift other work into your observability initiatives. As examples, if there's an existing stream of data you can tee to a secondary destination or critical data that can be seen in another way, jump on the chance to ship that data into your observability solution. Some examples of situations you could use to do this include:

- If you're using a ELK stack—or even just the Logstash part—it's trivial to add a snippet of code to fork the output of a source stream to a secondary destination. Send that stream to your observability tool. Invite users to compare the experience.
- If you're already using structured logs, all you need to do is add a unique ID to log events as they propagate throughout your entire stack. You can keep those logs in your existing log analysis tool, while also sending them as trace events to your observability tool.
- Try running client side observability instrumentation (for example, Honeycomb's Beelines or OpenTelemetry) alongside your existing APM solution. Invite users to compare and contrast the experience.
- If you're using Ganglia, you can leverage that data by parsing the XML dump it puts into /var/tmp with a once a minute cronjob that shovels that data into your observability tool as events. That's a less than optimal use of observability, but it certainly creates familiarity for Ganglia users.

- Re-create the most useful of your old monitoring dashboards as easily referenceable queries within your new observability tool. While dashboards certainly have their shortcomings (see: Chapter 2), this gives new users a landing spot where they can understand the system performance they care about at a glance, and also gives them an opportunity to explore and know more.

Anything you can do to blend worlds will help lower that barrier to adoption. Other people need to understand how their concerns map into the new solution. Help them see their current world in the new world you're creating for them. It's okay if this sneak peak isn't perfect. Even if the experience is pretty rough, what you're shooting for is familiarity. The use of the new tooling in this approach might be terrible, but if the names of things are familiar and the data is something they know, it will still invite people to interact with it more than a completely scratch dataset.

The last push is the hardest to complete

Using the strategies above to tackle the biggest pain points first and adopt an iterative approach can help you make fast progress as you're getting started, but they don't account for one of the hardest parts of implementing an observability solution: crossing the finish line. Now that you've gotten some momentum going, you also need a strategy for polishing off the remaining work.

Depending on the scope of work and size of your team, the technique of rolling out new instrumentation iteratively as part of your on-call approach can typically get most teams to a point where they've done about half to two-thirds of the work required to introduce observability into every part of the stack they intend. Inevitably, most teams discover that some parts of their stack are under less active development than others. For those rarely touched parts of the stack, you'll need a solid completion plan or your implementation efforts are likely to lag.

Even with the best of project management intentions, as some of the pain that was driving observability adoption begins to ease, so to can the urgency of completing the implementation work. The reality most teams live in is that engineering cycles are scarce, demands are always competing, and there's always another pain to address waiting around the corner once they've dealt with the one in front of them.

The goal of a complete implementation is to have built a reliable go-to debugging solution that can be used to fully understand the state of your production applications whenever anything goes wrong. Before you get to that end state, you will likely have different bits of tooling that are best suited to solving different problems. During the implementation phase, that disconnect can be tolerable because you're working toward a more cohesive future. But in the long-term, the risk of not completing your implementation is that it will create a drain of time, cognitive capacity, and attention from your teams.

That's when you need to make a timeline to chug through the rest quickly. Your target milestone should be to accomplish the remaining instrumentation work necessary so that your team can use your observability tool as their go-to option for debugging issues in production. Consider setting up a special push to get to the finish line, like a hack week culminating in a party with prizes (or something equally silly and fun) to bring the project over the top to get it done.

During this phase, it's worth noting that your team should strive to genericize instrumentation as often as possible so that it can be reused in other applications or by other teams within your organization. A common strategy here is to avoid repeating the initial implementation work by creating generic observability libraries that allow you to swap out underlying solutions without getting into code internals, similar to the approach taken by OpenTelemety (see: Chapter 7).

Conclusion

Knowing exactly where and how to start your observability journey depends on the particulars of your team. Hopefully, these general recommendations are useful to help you figure out places where you can get started. Actively participating in a community of peers can be invaluable as your first place to dig in. As you get started, focus on solving the biggest pain points rather than starting in places that already operate smoothly enough. Throughout your implementation journey, remember to keep an inclination toward moving fast, demonstrating high value and ROI, and tackling work iteratively. Find opportunities to include as many parts of your organization as possible. And don't forget to plan for completing the work in one big final push to get your implementation project over the finish line.

The tips in this chapter can help get you complete the work it takes to get started with observability. Once that work is complete, using observability on a daily basis helps unlock other new ways of working by default. The rest of this section will explore those in detail. In the next chapter, we'll examine how observability-driven development can revolutionize your understanding of how new code behaves in production.

---

1 Arkes, H. R., & Blumer, C. (1985), The psychology of sunk costs. Organizational Behavior and Human Decision Processes, 35, 124-140.

Chapter 10. Observability-Driven Development

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 11th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Observability is a practice that fundamentally helps engineers improve how their software runs in production. That practice isn't applicable only after software is released to production. Observability can, and should, be a part of the software development life cycle itself. This chapter focuses on the practice of observability-driven development and how to shift observability left.[1]

Test-driven development

Today's gold standard for testing software prior to its release in production is Test-Driven Development (TDD).[2] Within the last two decades, TDD is arguably one of the more successful practices to take hold across the software development industry. TDD has provided a very useful framework for shift-left testing that catches, and prevents, many potential problems long before they reach production. Widely adopted across wide-swaths of

the software development industry, TDD should be credited with having uplifted the quality level for code running production services.

TDD is a powerful practice that provides engineers with a clear way to think about software operability. Applications are defined by a deterministic set of repeatable tests that can be run hundreds of times per day. If these repeatable tests pass, then the application must be running as expected. Before changes to the application are actually produced, they start as a set of new tests that exist to verify that change would work as expected. A developer can then begin to write new code in order to ensure the test passes.

TDD is particularly powerful because tests run the same way every time. Data typically doesn't persist between test runs, it gets dropped; erased and recreated from scratch for each run. Responses from underlying or remote systems are stubbed or mocked. With TDD, developers are tasked with creating a specification that precisely defines the expected behaviors for an application in a controlled state. The role of tests is to identify any unexpected deviations from that controlled state and deal with them immediately. In doing so, TDD removes guesswork and provides consistency.

But that very consistency and isolation also limits what TDD can reveal about what is happening with your software in production. Running isolated tests doesn't reveal whether or not customers are having a good experience with your service. Nor does passing those tests mean that any errors or regressions could be quickly and directly isolated and fixed before releasing that code back into production.

Any reasonably experienced engineer responsible for managing software running in production can tell you that production environments are anything but consistent. Production is full of interesting deviations that your code might encounter out in the wild, but that have been excised from tests because they're not repeatable, they don't quite fit the specification, or they don't go according to plan. While the consistency and isolation of TDD makes your code tractable, it does not prepare your code for the interesting

anomalies that should be surfaced, watched, stressed, and tested because they ultimately shape how your software behaves when real people start interacting with it.

Observability can help you write and ship better code even before it lands in source control—because it's part of the set of tools, processes, and culture that allow engineers to find bugs in their code quickly.

Observability in the development cycle

Catching bugs cleanly, resolving them swiftly, and preventing them from becoming a backlog of technical debt that weighs down the development process relies on a team's ability to find those bugs quickly. Yet, software development teams often hinder their ability to do so for a variety of reasons.

For example, consider organizations where software engineers aren't responsible for operating their software in production. Engineers merge their code into master, cross their fingers hoping that this change won't be one that breaks prod, and they essentially wait to get paged if a problem occurs. Sometimes they get paged soon after deployment. The deployment is then rolled back and the triggering changes can be examined for bugs. But more likely, problems wouldn't be detected for hours, days, weeks, or months after that code had been merged. By that time, it becomes extremely difficult to pick out the origin of the bug, remember the context, or decipher the original intent behind why that code was written or why it shipped.

Resolving bugs quickly critically depends on being able to examine the problem *while the original intent is still fresh in the original author's head*. It will never again be as easy to debug a problem as it was right after it was written and shipped. It only gets harder from there: speed is key. At first glance, the links between observability and writing better software may not be clear. But it is this need for debugging quickly that deeply intertwines the two.

Determining where to debug

Newcomers to observability often make the mistake of thinking that observability is a way to debug your code, similar to using highly verbose logging. While it's possible to debug your code using observability tools, that is not the primary purpose of observability. Observability operates on the order of *systems*, not on the order of *functions*. Emitting enough detail at the lines level to reliably debug code would emit so much output that it would swamp most observability systems with an obscene amount of storage and scale. It would simply be impractical to pay for a system capable of doing that because it would likely cost somewhere in the ballpark of 1X-10X as much as your system itself.

Observability is not for debugging your code logic. *Observability is for figuring out where in your systems to find the code you need to debug*. Observability tools help you swiftly narrowing down where problems may be occuring. From which component did an error originate? Where is latency being introduced? Where did a piece of this data get munged? Which hop is taking up the most processing time? Is that wait time evenly distributed across all users, or is it only experienced by a subset thereof? Observability helps your investigation of problems pinpoint likely sources.

Often, observability will also give you a good idea of what might be happening in or around an affected component, what the bug might be, or even provide hints as to where the bug is actually happening: your code, the platform's code, or a higher-level architectural object.

Once you've identified where the bug lives and some qualities about how it arises, then observability's job is done. From there, if you want to dive deeper into the code itself, the tool you want is a good old-fashioned debugger (for example, gdb). Once you suspect how to reproduce the problem, you can spin up a local instance of the code, copy over the full context from the service, and continue your investigation. While they are related, the difference between an observability tool and a debugger is an order of scale; like a telescope and a microscope, they may have some overlapping use cases, but they are primarily designed for different things.

This is an example of different paths you might take with debugging vs observability:

- You see a spike in latency. You start at the edge; break down by endpoint, calculate average, 90th, and 99th percentile latency; identify some cadre of slow requests; trace one of them; it shows the timeouts begin at service3. You copy the context from the traced request into your local copy of service3 binary and attempt to reproduce it in the debugger.
- You see a spike in latency. You start at the edge; break down by endpoint, calculate average, 90th, and 99th percentile latency; notice that only endpoints which are write-only are the ones that are suddenly slower. You break down by db destination host, and note that it is distributed across some, but not all, of your db primaries. For those db primaries, this is only happening to ones of a certain instance type or in a particular AZ. You conclude the problem is not a code problem, but one of infrastructure.

Debugging in the time of microservices

When viewed through this lens, it becomes very clear why the rise of microservices is tied so strongly to the rise of observability. Software systems used to have fewer components, which meant they were easier to reason about. An engineer could think their way through all possible problem areas using only low cardinality tagging. Then, to understand their code logic, they simply always used a debugger or IDE. But once monoliths started being decomposed into many distributed microservices, the debugger no longer worked as well because it couldn't hop the network.[3]

Once service requests started traversing networks to fulfill their functions, all kinds of additional operational, architectural, infrastructural, and other assorted categories of complexity became irrevocably intertwined with the logic bugs we unintentionally shipped and inflicted on ourselves.

In monolithic systems, it's very obvious to your debugger if a certain function slows down immediately after code is shipped that modified that function. Now, such a change could manifest in several ways. For example, you might notice:

- That a particular service is getting slower
- That dependent services upstream of that service are also getting slower
- Downstream dependent services called by the first service are also getting slower
- All of the above

Furthermore, regardless of which of those manifestations you encounter, it might still be incredibly unclear if that slowness is being caused by:

- A bug in your code
- A particular user changing their usage pattern
- A database overflowing its capacity
- Network connection limits
- A misconfigured load balancer
- Issues with service registration or service discovery
- Some combination of the above

WIthout observability, all you may see is that all of the performance graphs are either spiking or dipping at the same time.

How instrumentation drives observability

Observability helps pinpoint where problems originate, common outlier conditions, which half a dozen or more things must all be true for the error to occur, and so forth. Observability is also ideal for swiftly identifying

whether problems are restricted to a particular build ID, set of hosts, instance types, container versions, kernel patch versions, database secondaries, or any number of other architectural details.

In order for that to be true, a necessary component of observability is focusing on creating useful instrumentation. *Good instrumentation drives observability*. One way to think about how instrumentation is useful is to consider it in the context of pull requests. Pull requests should never be submitted or accepted without first asking yourself the question, "How will I know if this change is working as intended or not?"

A helpful goal when developing instrumentation is to create reinforcement mechanisms and shorter feedback loops. In other words, tighten the loop between shipping code and feeling the consequences of errors. This is also known as "putting the software engineers on call." One way to achieve this is to automatically page the person who just merged the code that is being shipped. For a brief period of time, maybe 30 minutes to 1 hour, if an alert is triggered in production it gets routed to them. When an engineer experiences their own code in production, their ability (and motivation) to instrument their code for faster isolation and resolution of issues naturally increases. This feedback loop is not punishment; rather, it is *essential* to code ownership. We cannot develop the instincts and practices needed to ship quality code if we are insulated from the feedback of our errors.

Every engineer should be expected to instrument their code such that they can answer these questions as soon as it's deployed:

- Is your code doing what you expected it to do?
- How does it compare to the previous version?
- Are users actively using your code?
- Are there any emerging abnormal conditions?

A more advanced approach is to enable engineers to test their code against a small subset of production traffic. With sufficient instrumentation, the best way to understand how a proposed change will work in production is to

measure how it will work by deploying it to production. That can be done in several controlled ways. For example, that can happen by deploying new features behind a feature flag and only exposing it to a subset of users. Alternatively, a feature could also be deployed directly to production and have only select requests from particular users routed to the new feature. These types of approaches shorten feedback loops to mere seconds or minutes, rather than what are usually substantially longer periods of time waiting for a full release cycle.

If you are capturing sufficient instrumentation detail in the context of your requests, you can systematically start at the edge of any problem and work your way to the correct answer every single time, with no guessing, intuition, or prior knowledge needed. This is one revolutionary advance observability has over monitoring systems, and does a lot to move operations engineering into the realm of science, not magic.

Shifting observability left

While test-driven development ensures developed software adheres to an isolated specification, observability-driven development ensures that software works in the messy reality that is a production environment; strewn across a complex infrastructure, at any particular point in time, experiencing fluctuating workloads, with particular users.

Building instrumentation into our software early in the development lifecycle allows engineers to more readily consider and more quickly see the impact that small changes truly have in production. By focusing on just adherence to an isolated spec, teams inadvertently create conditions where they cease to have visibility into the chaotic playground where that software comes into contact with messy and unpredictable people problems. As we've seen in previous chapters, traditional monitoring approaches only reveal an aggregate view of measures that were developed in response to triggering alerts for known issues. Traditional tools provide very little ability to accurately reason about what happens in complex modern software systems.

As a result, teams will often approach production as a glass castle; a beautiful monument to their collective design over time, but one they're afraid to touch because any unforeseen movement could shatter the entire structure. Developing the engineering skills to write, deploy, and use good telemetry to understand behaviors in production, teams become empowered to reach further and further into the development lifecycle to consider the nuances of detecting unforeseen anomalies that could be roaming around their castles unseen.

Observability-driven development is what allows engineering teams to turn their glass castles into playgrounds. Production environments aren't immutable, they're full of action, and engineers should be empowered to confidently walk into any game and score a win. But that only happens when observability isn't considered solely the domain of SREs, infrastructure engineers, or operations teams. Software engineers must adopt observability and work it into their development practices in order to unwind the cycle of fear they've developed over making any changes to production.

---

1 https://en.wikipedia.org/wiki/Shift-left_testing

2 https://en.wikipedia.org/wiki/Test-driven_development

3 As a historical aside, the phrase "strace for microservices" was an early attempt to describe this new style of understanding system internals before the word "observability" was adapted to fit the needs of introspecting production software systems.

Chapter 11. Using Service Level Objectives for Reliability

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 12th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

This chapter introduces why and how to use Service Level Objectives (SLOs) for ensuring reliability in observable systems. You will learn about the common problems created by traditional monitoring approaches, how those problems are exacerbated in the world of distributed systems, and how an SLO-based approach to monitoring can solve those problems. In the next chapter, we'll examine why observability makes for actionable alerts when using an SLO-based approach.

In this chapter, we cover an introductory look at SLO-based alerting, how observability is used to respond to those alerts, and a real-world use case of shifting engineering culture to adopt SLOs for monitoring.

Introduction to Service Level Objectives

Service Level Objectives (SLOs) are internal goals for measurement of service health. Popularized by the Google SRE book, SLOs are a key part

of setting external service level agreements between service providers and their customers. These internal measures are typically more stringent than externally facing agreements or availability commitments. With that added stringency, SLOs can provide a safety net that helps teams identify and remediate issues before external user experience reaches unacceptable levels.

Much has been written about the use of SLOs for service reliability (we recommend Alex Hidalgo's book, *Implementing Service Level Objectives* for a deeper understanding of the subject) and they are not unique to the world of observability. However, using an SLO-based approach to monitoring service health requires having observability built into your applications. SLOs can be, and sometimes are, implemented in systems without observability. But the ramifications of doing so can have severe unintended consequences. To examine why that's true, let's start by looking at the characteristics of traditional monitoring alerts.

Traditional Monitoring Approaches Create Dangerous Alert Fatigue

In monitoring-based approaches, alerts often measure the things that are easiest to measure. Metrics are used to track simplistic system states that might indicate a service's underlying process(es) may be running poorly or may be a leading indicator of troubles ahead: trigger an alert if CPU is above 80%, or if available memory is below 10%, or if disk space is nearly full, or if more than X many threads are running, or any set of other simplistic measures of underlying system conditions.

While those measures are easy to collect, they don't turn out to be useful measures for triggering alerts. Those simplistic measures may also be indicators that a backup process is running, or a garbage collector is doing its cleanup job, or that any number of other things may be happening on a system. In other words, those conditions may reflect any number of system factors, not just the problematic ones we really care about. Triggering alerts from these simplistic measures creates a high percentage of false positives.

Experienced engineering teams that own the operation of their software in production will often learn to tune out, or even suppress, these types of alerts because they're so unreliable. Teams that do so regularly adopt phrases like, "don't worry about that alert: we know the process runs out of memory from time to time."

Becoming accustomed to alerts that are prone to false positives is a known problem and a dangerous practice. In other industries, that problem is known as "normalization of deviance": a term coined during investigation of the Challenger disaster. When individuals in an organization regularly shut off alarms or fail to take action when alarms occur, they eventually become so desensitized to the fact that this practice deviates from the expected response that it no longer feels wrong to them. Failures that are "normal" and disregarded are, at best, simply background noise. At worst, they create oversights that eventually enable cascading system failures with disastrous results.

In the software industry, the poor signal-to-noise ratio of monitoring-based alerting often leads to alert fatigue — and to gradually paying less attention to all alerts, because so many of them are false-alarms, inactionable, or simply not useful. Unfortunately, with monitoring-based alerting, that problem is often compounded when incidents occur. Post-incident reviews often generate action-items that create new, more important alerts that, presumably, would not have not gone ignored. That leads to an even larger set of alerts generated during the next incident. That pattern of alert escalation creates an ever-increasing flood of alerts and an ever-increasing cognitive load on responding engineers to determine which alerts matter and which don't.

That type of dysfunction is so common in the software industry that many monitoring and incident response tool vendors proudly offer various solutions labeled "AIOps" to group, suppress, or otherwise try to process that alert load for you. Engineering teams have become so accustomed to alert noise that this pattern is now seen as normal. If the future of running software in production is doomed to generate so much noise that it must be artificially processed, it's safe to say the situation has gone well beyond a

normalization of deviance: industry vendors have now productized that deviance and will happily sell you a solution for its management.

Again, we believe that type of dysfunction exists because of the limitations imposed by the fact that metrics and monitoring tools were the best choice we previously had available to understand the state of production systems. As an industry, we now suffer a collective Stockholm Syndrome that is just the tip of the dysfunction iceberg when it comes to the many problems we encounter today as a result. The complexity of modern system architectures along with the higher demands for their resilience have pushed the state of these dysfunctional affairs away from tolerable toward no longer acceptable. Distributed systems have demanded an alternative approach.

Distributed Systems Exacerbate the Alerting Problem

In self-contained systems, it is much easier to anticipate failure modes. So it seems logical to add monitoring that expects each of those known failure modes to occur. Operations teams supporting self-contained production systems can write very specific monitoring checks for each precise state. However, as systems become more complex, they create a combinatorial explosion of potential failure modes. Ops teams accustomed to working with less complex systems will rely on their intuition, best guesses, and memories of past outages in order to predict any possible system failures they can imagine. While that approach with traditional monitoring may work at smaller scales, the explosion of complexity in modern systems requires teams to write and maintain hundreds, or even thousands, of precise state checks to look for every conceivable scenario.

That approach isn't sustainable. The checks aren't maintainable, often the knowledge isn't transferable, and past historical behavior doesn't necessarily predict failures likely to occur in the future. The traditional monitoring mindset is often about preventing failure. But in a distributed system with hundreds or thousands of components serving production traffic, *failure is inevitable*.

In modern systems, engineering teams regularly distribute load across multiple disparate systems. They split up, shard, horizontally partition, and replicate data across geographically distributed systems. While these architectural decisions optimize for performance, resiliency, and scalability, they can also make systemic failures impossible to detect or predict. Emergent failure modes can impact your users long before the coarse synthetic probes traditionally used as indicators will tell you. Traditional measurements for service health suddenly have little value because they're so irrelevant to the behavior of the system as a whole.

Traditional system metrics often miss unexpected failure modes in distributed systems. An abnormal number of running threads on one component might indicate garbage collection is in progress, or it might also indicate slow response times might be imminent in an upstream service. It's also possible that the condition detected via system metrics might be entirely unrelated to service slowness. Still, that system operator receiving an alert about that abnormal number of threads in the middle of the night won't know which of those conditions is true and it's up to them to divine that correlation.

Further, distributed systems design for resiliency with loosely coupled components. With modern infrastructure tooling, it's possible to automatically remediate many common issues that used to require waking up an engineer. You probably already use some of the current and commonplace methods for building resilience into your systems: autoscaling, load balancing, failover, and so forth. Failures that get automatically remediated should not trigger alarms. (Anecdotally, some teams may do just that in an attempt to build valuable "intuition" about the inner workings of a service. On the contrary, that noise only serves to starve the team of time they could be building intuition about the inner workings of service components that don't have auto-remediation.)

That's not to say that you shouldn't debug auto-remediated failures. You should absolutely debug those failures *during normal business hours*. The entire point of alerts is to bring attention to an emergency situation that simply cannot wait. Triggering alerts that wake up engineers in the middle

of the night to notify them of transient failures simply creates noise and leads to burnout. Thus, we need a strategy to define the urgency of problems.

In a time of complex and interdependent systems, it's easy for teams to reach fatigue from the deluge of alerts that may, but probably don't, reliably indicate a problem exists with the way customers are currently using the services your business relies on. Alerts for conditions that aren't tied directly to customer experience will quickly become nothing more than background noise. Those alerts are no longer serving their intended purpose and, more nefariously, they actually serve the opposite purpose. They distract your team from paying attention to the alerts that really do matter.

Unreliable alerts must be removed if you expect to run a reliable service.

Yet, even if you believe in your heart that assertion is true, many teams still fail to remove those unnecessary distractions. It can be a challenging transition to get rid of the old alerts associated with traditional monitoring. Often, the prevailing concern is that by removing alerts, teams will have no way of learning about service degradation. But it's important to realize that these types of traditional alerts are only helping you detect *known* unknowns: problems you know could happen, but are unsure may be happening at any given time. That alert coverage provides a false sense of security because it does nothing to prepare you for dealing with new and novel failures, or the *unknown unknowns*.

How many of you have the luxury of operating systems where only a fixed set of failures can ever occur? While that state might be closer to achievable for legacy self-contained software systems with very few components, that's certainly not the case with modern systems. Yet still, there's hesitation to remove many of the old alerts associated with traditional monitoring. Later in this chapter, we'll look at ways to feel confident when removing unhelpful alerts.

For now, let's define the criteria for an alert to be considered helpful. First, it must be a reliable indicator that the user experience of your service is in a degraded state. Second, the alert must be actionable. There must be a

methodical way to take action in response to the alert that does not require a responder to divine the right course of action. If those two conditions are not true, any alert you have configured is no longer serving its intended purpose.

Static Thresholds Can't Reliably Indicate Degraded User Experience

How then, do you focus on setting up alerts to detect failures that impact user experience? This is where a departure from traditional monitoring approaches becomes necessary. The traditional metrics-based monitoring approach relies on using static thresholds to define optimal system conditions. Yet the performance of modern systems — even at the infrastructure level — often changes shape dynamically under different workloads. Static thresholds simply aren't up to the task of monitoring impact to user experience.

Setting up traditional alerting mechanisms to monitor user experience means that system engineers must choose arbitrary constants that predict when that experience is poor. For example, these alerts might be implemented to trigger when "10 users have experienced slow page load times," or "when the 95th percentile of requests have persisted above a certain number of milliseconds." In a metrics-based approach, system engineers are required to divine which exact static measures indicate unacceptable problems are occurring.

Yet system performance varies significantly throughout the day as different users, in different timezones, interact with your services in different ways. During slow traffic times, when you may have hundreds of concurrent sessions, 10 users experiencing slow page load times might be a significant metric. But that significance drops sharply at peak load times when you may have tens of thousands of concurrent sessions running.

Remember that in distributed systems, failure is inevitable. Small transient failures are always occurring without you necessarily noticing. Common examples include situations where one failed request will later succeed on a

retry, a critical process might initially fail but its completion is merely delayed until it gets routed to a newly provisioned host, or a service that becomes unresponsive until its requests are routed to a backup service. The additional latency introduced by these types of transient failures might blend into normal operations during peak traffic, but p95 response times would be more sensitive to individual data points during periods of low traffic.

In a similar vein, that example also illustrates the coarseness of time-based metrics. Let's say p95 response time is being gauged in five minute intervals. Every five minutes, performance over the trailing five minute interval reports a value that triggers an alert if it exceeds a static threshold. If that value exceeds the threshold, the entire five minute interval is considered bad (and conversely, any five minute interval that didn't exceed the threshold is considered good). Alerting on that type of metric results in high rates of both false positives and false negatives. It also has a level of granularity that is insufficient to diagnose when and where exactly a problem may have occurred.

Static thresholds are too rigid and coarse to reliably indicate degraded user experience in a dynamic environment. They lack context. Reliable alerts need a finer level of both granularity and reliability.

Reliable Alerting with SLOs

Service Level Objectives quantify an agreed upon target for service availability, based on critical end-user journeys rather than system metrics. That target is measured using Service Level Indicators (SLIs) which categorize the system state as good or bad. There are two kinds of SLIs: time-based measures (such as "99th percentile latency less than 300ms over each 5 minute window"), and event-based measures (such as "proportion of events that took less than 300ms during a given rolling time window"). We recommend setting SLIs that use event-based measures, as opposed to time-based measures because event-based measures provide both a more reliable and a more granular way to quantify the state of a service. Remember that your first criteria for helpful alerts is that they must be a reliable indicator

that customer experience for your service is in a degraded state. Let's look at an example of how that works with an event-based SLI.

As an example, you may define a good customer experience as being a state when, "a user should be able to successfully load your home page and see a result quickly." Expressing that with an SLI means qualifying events and then determining if they meet our conditions. In this example, your SLI would:

- Look for any event where the request path is "/home"
- Screen qualifying events for conditions where the event duration < 100 ms
    - If event duration < 100 ms *and* was served successfully, consider it OK
    - If event duration > 100 ms, consider that event an error even if it returned a success code.

Any event that is an error would detract from your SLO's error budget. We'll closely examine patterns for managing SLO error budgets and triggering alerts in the next chapter. For now, we'll summarize by saying that given enough errors, an alert could be triggered.

In this approach, the granularity of the alert is per-event and the scope of time over which you want that considered can be arbitrary. The flexibility of that measurement isn't sensitive to dynamic changes like traffic dips, transient failures, or other unpredictable events. The focus is also narrowed to only consider symptoms that impact what the users of our service experience. If some underlying condition is occurring that impacts "a user loading our home page and seeing it quickly," then an alert should be triggered because someone needs to investigate why.

Note that in this approach, there is no correlation as to why the service might be degraded. We simply know what is wrong. In contrast, traditional monitoring relies on a cause-based approach: a previously known cause is detected (e.g. an abnormal number of threads), signaling that users might

experience undesirable symptoms (slow page load times). That approach fuses the "what" and "why" of a situation in an attempt to help pinpoint where investigation should begin.

But, as we've seen, there's a lot more to your services than just the known states of UP, DOWN, or even SLOW. Emergent failure modes can impact your users long before coarse synthetic probes will tell you. Decoupling "what" versus "why" is one of the most important distinctions in writing good monitoring with maximum signal and minimum noise.

The second criteria for an alert to be helpful is that it must be actionable. SLO-based alerts are symptom-based: they tell you that something is wrong. They are actionable because now it is up to the responder to determine the why. In order for your SLO-based alerts to be actionable, your production systems must be sufficiently debuggable. That's where a shift to observability becomes essential: it must be safe and natural to debug in production.

In an SLO-based world, you need observability: the ability to ask novel questions of your systems without having to add new instrumentation. As you've seen throughout this book, observability allows you to debug from first principles. With rich telemetry you can start wide and then filter to reduce the search space. That approach means you can respond to determine the source of any problem, regardless of how novel or emergent the failure may be.

In self-contained, unchanging systems with a long-lived alert set, it's possible to have a large collection of alerts that correspond to all known failures. Given the resources and time, it's also theoretically possible to mitigate and even prevent all of those known failures (so why haven't more teams done that already?). The reality is that no matter how many known failures you automatically fix, the emergent failure modes that come with modern distributed systems can't be predicted. There's no going back in time to add metrics around the parts of your system that happened to break unexpectedly. When anything can break at any time, you need data for everything. Note, that's "*data* for everything," and not, "*alerts* for

everything." It's not feasible to have alerts for everything. As we've seen, the software industry as a whole is already drowning in the noise.

Observability is a requirement for responding to novel, emergent failure modes. With observability, you can methodically interrogate your systems by taking one step after another: ask one question and examine the result, then ask another, and so on. No longer are you limited by traditional monitoring alerts and dashboards. Instead, you can improvise and adapt solutions to find any problem in your system.

Instrumentation that provides rich and meaningful telemetry is the basis for that approach. Being able to quickly analyze that telemetry data to validate or falsify hypotheses is what empowers you to feel confidence in removing noisy, unhelpful alerts. Decoupling the "what" versus "why" in your alerting is possible when using SLO-based alerts in tandem with observability.

When most alerts are not actionable, that quickly leads to alert fatigue. In order to eliminate that problem, it's time to delete all of your unactionable alerts.

Still not sure you can convince your team to remove all of those unhelpful alerts? Let's look at a real example of what it takes to drive that sort of culture change.

Changing Culture Toward SLO-Based Alerts: A Case Study

Just having queryable telemetry with rich context, on its own, might not be enough to feel confident deleting all of those existing unhelpful alerts. That was our experience at Honeycomb. We'd implemented SLOs, but our team didn't quite fully trust them yet. SLO alerts were being routed to a low-priority inbox, while the team continued to rely on traditional monitoring alerts. That trust wasn't established until we had a few incidents where SLO-based alerts flagged issues *long before* traditional alerts provided any useful signal.

To illustrate how this change occurred, let's examine an incident from the end of 2019. In that incident, Liz had developed the SLO feature and was paying attention to SLO alerts while the rest of the team was focused on traditional alerts. In this outage, the Shepherd service that ingests all of our incoming customer data had its SLO begin to burn, and sent an alert to the SLO test channel. The service recovered fairly quickly: there had been a 1.5% brownout for 20 minutes. The SLO error budget had taken a ding because that burned most of the 30 day budget. However, the problem appeared to go away by itself.

Our on-call engineer was woken up by the SLO alert at 1:29 a.m. his time. Seeing that the service was working, he blearily decided it was probably just a blip. The SLO alert triggered, but traditional monitoring which required consecutive probes in a row to fail didn't detect a problem. When the SLO alert triggered a fourth time, at 9:55am, this was undeniably not a coincidence. At that time, the engineering team was willing to declare an incident even though traditional monitoring had still not detected a problem.

While investigating the incident, another engineer thought to check the process uptime. In doing so, they discovered a process had a memory leak. Each machine in the cluster had been running out of memory, failing in synchrony, and restarting. Once that problem was identified, it was quickly correlated with a new deploy, and we were able to roll back the error. The incident was declared resolved at 10:32 a.m.

At this point, we have to acknowledge that many engineers who are reading this story might challenge that traditional cause-based alerting would have worked well enough in this case. Why not alert on memory, and therefore get alerts when the system runs out of memory (OOMed)? There are two points to consider when addressing that challenge.

First, at Honeycomb, our engineers had long since been trained out of tracking OOMs. Caching, garbage collection, and backup processes all opportunistically used — and occasionally used up — system memory. 'Ran out of memory' turned out to be very common for our applications. Given our architecture, even having a process crash from time to time

turned out not to be fatal, so long as all didn't crash at once. For our purposes, tracking those individual failures had not been useful at all. We were more concerned with the availability of the cluster as a whole.

Given that scenario, traditional monitoring alerts did not—and would never have—noticed this gradual degradation at all. Those simple coarse synthetic probes could only detect a total outage, not one out of 50 probes failing and then recovering. In that state, there were always machines available, so the service was up and most data was making it through.

Second, even if were somehow possible to introduce enough complicated logic to trigger alerts when only some notable number of specific types of OOMs were detected, we would have needed to predict this exact failure mode, well in advance of this one bespoke issue ever occurring, in order to devise the right incantation of conditions on which to trigger a useful alert. That theoretical incantation might have detected this one bespoke issue this one time, but would likely most often just exist to generate noise and propagate alert fatigue.

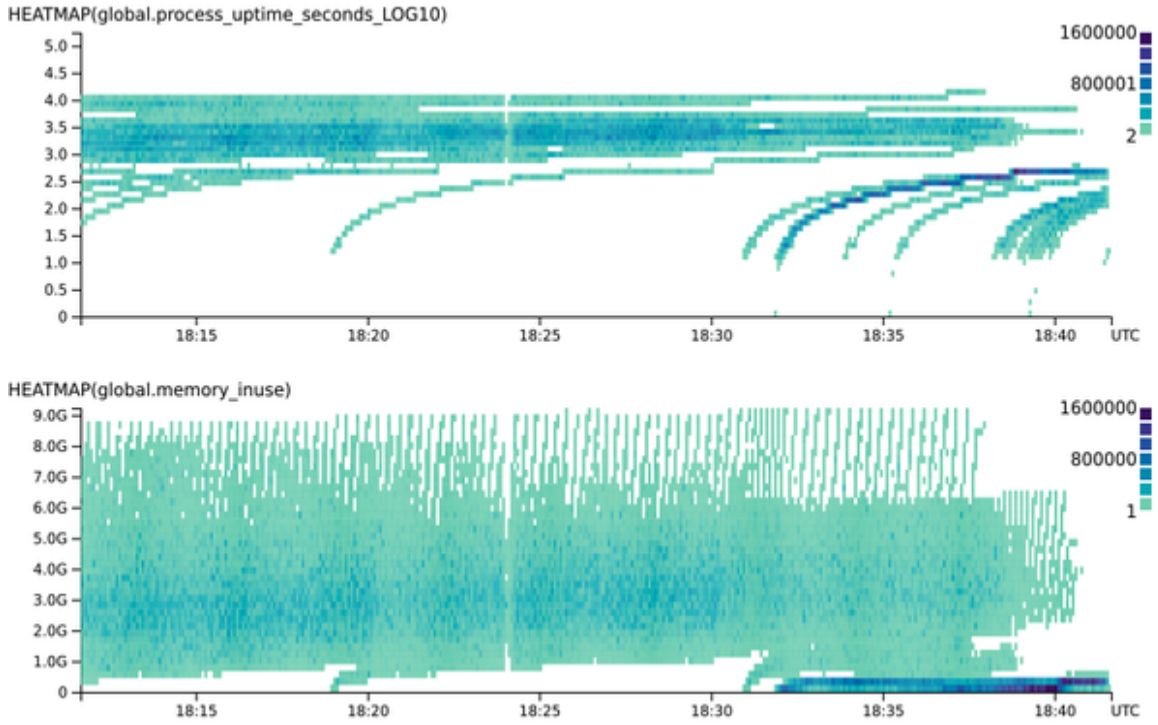


Figure 11-1. A heatmap showing uptime of a crashing fleet, restarting together around 18:30; and their memory profile.

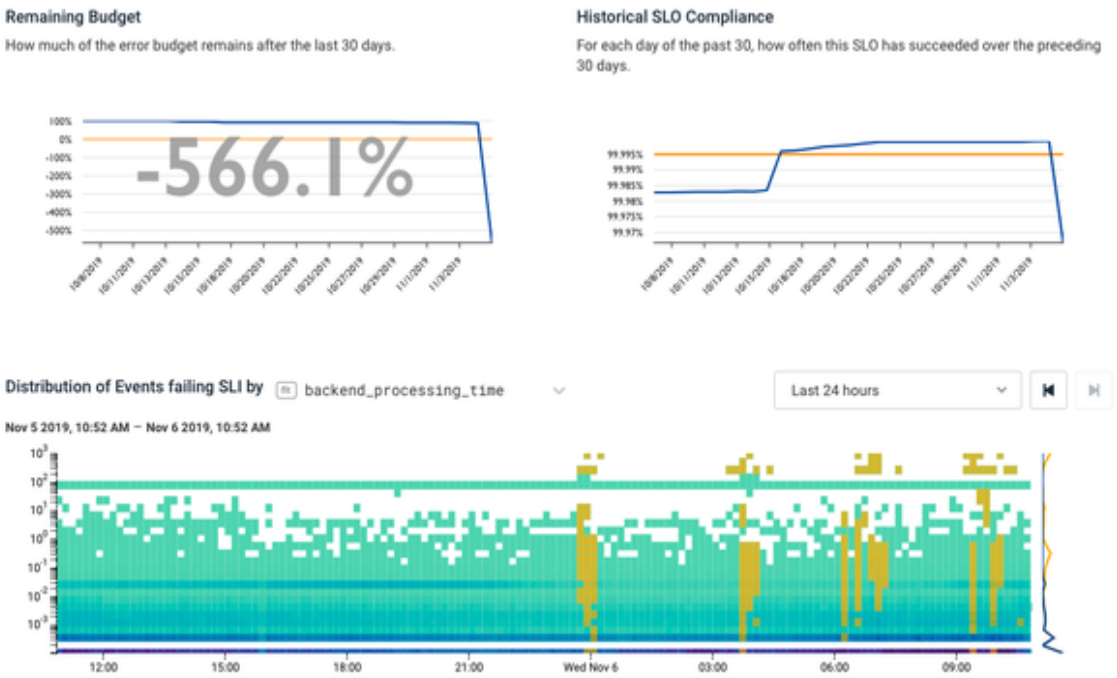


Figure 11-2. The SLO error budget burned -566% by the time the incident was over; compliance dropped to 99.97% (from the target of 99.995%). The yellow marks on the timeline show events marked as bad—and how they occurred fairly uniformly at fairly regular intervals until corrected.

Although traditional monitoring never detected the gradual degradation, the SLO hadn't lied: users were seeing a real effect. Some incoming customer data had been dropped. It burned our SLO budget almost entirely from the start.

By that stage, if the team had started treating SLO-based alerts as first-class citizens, we would have been less likely to look for external and transient explanations. We would have instead moved to actually fix the issue, or at least roll back the latest deploy. SLOs proved their ability to detect brownouts and prompt the appropriate response.

That incident changed our culture. Once SLO burn alerts had proven their value, our engineering team had as much respect for SLO-based alerts as they did for traditional alerts. After a bit more time relying on SLO-based alerts, our team became increasingly comfortable with the reliability of alerting purely on SLO data.

At that point, we deleted all of our traditional monitoring alerts and we now rely solely on SLO-based alerts.

Conclusion

This was a high-level overview of SLOs and the opportunities they provide for more effective alerting strategies. Alert fatigue is prevalent in the software industry and it's enabled by the cause-based approach taken by traditional monitoring solutions.

Alert fatigue can be solved by focusing on only creating "helpful" alerts that meet two criteria. First, they must only trigger as reliable indicators that the user experience of your service is in a degraded state. Second, they must be actionable. Any alert that does not meet that criteria is no longer serving its purpose and it should be deleted.

SLOs decouple the "what" and "why" behind incident alerting. Focusing on symptom-based alerts means that SLOs can be reliable indicators of customer experience. When SLOs are driven by event-based measures, they have a far lower degree of false positives and false negatives. Therefore, SLO-based alerts can be a productive way to make alerting less disruptive, more actionable, and more timely. They can help triage between systemic problems, and occasional stochastic failures.

But on their own, SLO-based alerts only reveal symptoms. In order for your SLO-based alerts to be actionable, your production systems must be sufficiently debuggable. Having observability in your systems is critical for success when using SLOs.

In the next chapter, we'll go in-depth into the inner workings of SLO burn budgets and examine how they're used with more technical detail.

Chapter 12. Using observability data to model actionable SLOs

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 13th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

In the previous chapter, we introduced Service-Level Objectives (SLOs) and how an SLO-based approach to monitoring makes for more effective alerting. This chapter closely examines the SLO error budget and the mechanisms available to create alerts when adopting the patterns previously described. We'll look at what an SLO error budget is and how it works, which forecasting calculations are available to predict that your SLO error budget will be exhausted, and show why it is necessary to use event-based observability data rather than time-based metrics to make reliable calculations.

Alerting before your error budget is empty

An error budget represents that maximum amount of system unavailability that your business is willing to tolerate. If your service-level objective is to ensure 99.9% of requests are successful, a time-based calculation would

state that your system could be unavailable for no more than 8 hours, 45 minutes, 57 seconds in one standard year (or 43 minutes, 50 seconds per month). As shown in the previous chapter, an event-based calculation considers each individual event against qualification criteria and keeps a running tally of "*good*" events vs. "*bad*" (or errored) events.

Since availability targets are represented as percentages, that means an SLO's error budget is always fixed. For any given period of time, only so many errors can be tolerated. A system is out of compliance with its SLO once its entire error budget has been spent. Subtract the number of errored queries from your available error budget and that is known colloquially as your "error budget burn."

To proactively manage SLO compliance, you need to become aware of and resolve application and system issues long before your entire error budget is burned. Time is of the essence. In order for you to take corrective action that averts the burn down, you need to know if you are on a trajectory to consume that entire budget well before it happens. The higher your SLO target, the less time you have to react.

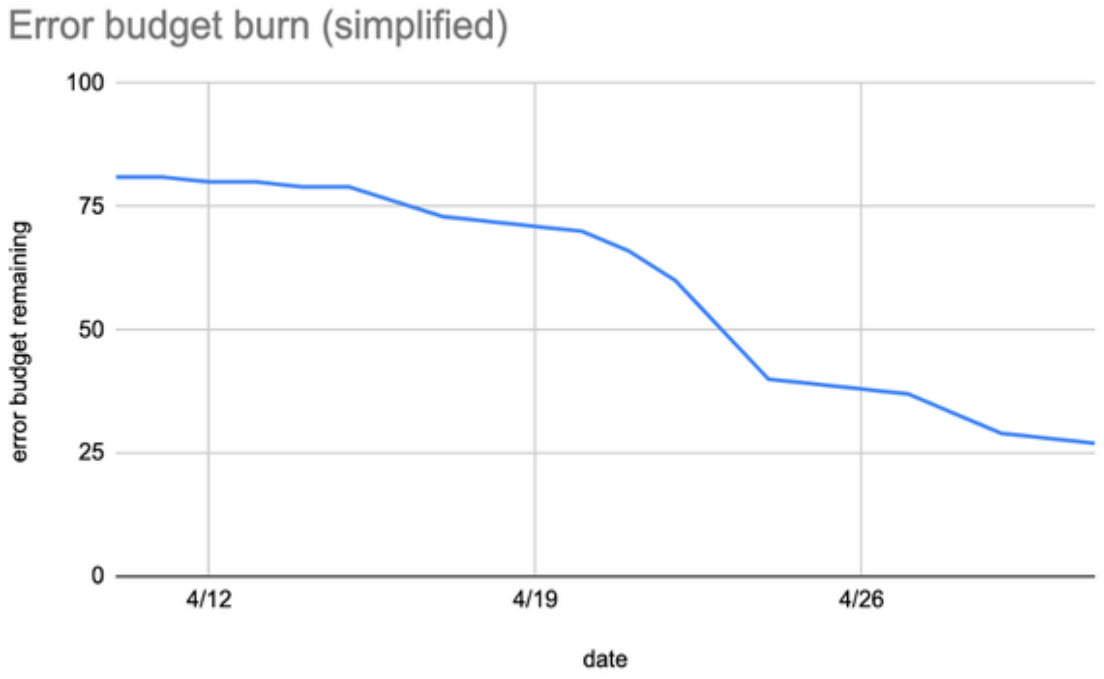


Figure 12-1. A simple graph showing how error budget burn over the course of approximately three weeks

For SLO targets up to about 99.95%—where up to 21 minutes, 54 seconds of full downtime per month are tolerable within a month—a reasonable enough time period exists where your team could be alerted to potential issues and still have enough time to act proactively before the entire error budget is burned.

Error budget burn alerts are designed to provide early warning about future SLO violations that would occur if the current burn rate continues.

Therefore, effective burn alerts must forecast the amount of error budget that your system will have burned at some future point in time (anywhere from several minutes to days from now). There are several ways to make that calculation and decide when to trigger alerts on error budget burn rate. The rest of this section will closely examine and contrast various approaches to making effective calculations to trigger error budget burn alerts.

Before proceeding, it's worth noting that these types of preemptive calculations work best to prevent violations for SLOs with targets up to

99.95% (like in the previous example). For SLOs with targets exceeding 99.95%, they work less preventatively but can still be used to report on and warn about system degradation.

Let's examine what it takes to get an error budget burn alert working. First, you must start by setting a frame for considering the all too relative dimension of time.

Framing time as a sliding window

The first choice is whether to set time as a fixed window or a sliding window. An example of a fixed window would be one that follows the calendar—starting on the 1st day of the month and ends on the 30th. A counterexample would be a sliding window that looks at any trailing 30-day period.

You might choose a fixed window to start with, but in practice fixed window availability targets don't match the expectations of your customers. You might issue a customer a refund for a particular bad outage that happens on the 31st of the month, but that does not wave a magic wand that suddenly makes them tolerant of a subsequent outage on the 2nd of the next month—even if that second outage is legally within a different window.

Service-level objectives should be used as a way to measure customer experience and satisfaction, not legal constraints. The better SLO is one that accounts for human emotions, memories, and recency bias. Human emotions don't magically reset at the end of a calendar month.

If you use a fixed window to set error budgets, it means that those budgets reset with dramatic results. Any error burns away a sliver of the budget with a cumulative effect gradually counting down toward zero. That gradual degradation continually chips away at how much time your team has to proactively respond to issues. Then, suddenly, on the 1st day of a new month everything is reset and you start the cycle again.

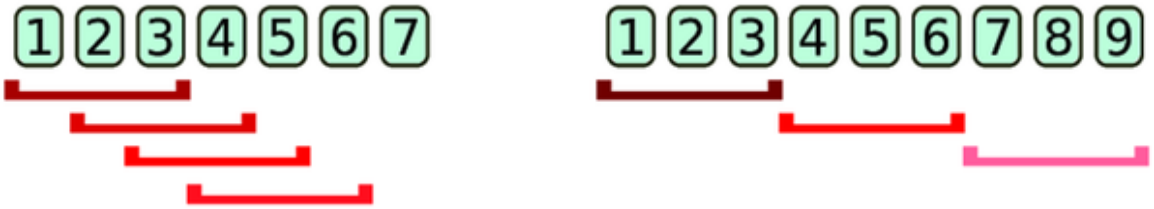


Figure 12-2. The illustration on the left shows a rolling 3 day rolling window. On the right, it shows a 3 day resetting window.

In contrast, using a sliding window to track how much error budget you've burned over a trailing period offers a smoother experience that more closely resembles human behavior. At every interval, the error budget can be burned a little or restored a little. A constant and low level of burn never completely exhausts the error budget and even a medium-sized drop will gradually be paid off, after a sufficiently stable period of time.

The correct first choice to make when calculating burn trajectories is to frame time as a sliding window, rather than a static fixed window. Otherwise, there isn't enough data after a window reset to make meaningful decisions.

Forecast models to create a predictive burn alert

With a time frame selected, you can now set up a trigger to alert you about error budget conditions you care about. The easiest alert to set is a *zero-level* alert—one that would trigger when your entire error budget is exhausted.

WHAT HAPPENS WHEN YOU EXHAUST YOUR ERROR BUDGET

When you transition from having a positive error budget to a negative one, a necessary practice is to shift away from prioritizing work on new features and toward work that prioritizes service stability. It's beyond the scope of this chapter to cover exactly what happens after your error budget is exhausted. But an important concept to include here is that your team's goal should be to prevent the error budget from being entirely spent. Error-budget over-expenditures translate into stringent actions such as long periods of feature freezes in production. The SLO model creates incentives to minimize actions that jeopardize service stability once the budget is exhausted. For an in-depth analysis at how engineering practices should change in these situations and how to set up appropriate policies, we recommend reading *Implementing Service Level Objectives* by Alex Hidalgo.

A trickier alert to configure is one that is preemptive. If you can foresee the emptying of your error budget, you have an opportunity to take action and introduce fixes that prevent it from happening. By fixing the most egregious sources of error sooner, you can forestall decisions to drop any new feature work in favor of reliability improvements. Planning and forecasting are better methods than heroism when it comes to sustainably ensuring team morale and stability.

Therefore, a solution is to track your error budget burn rate and watch for drastic changes that threaten to consume the entire budget. There are at least two models for triggering burn alerts above the zero-level mark. The first is to pick a non-zero threshold on which to alert. For example, you could alert when your remaining error budget dips below 30%.

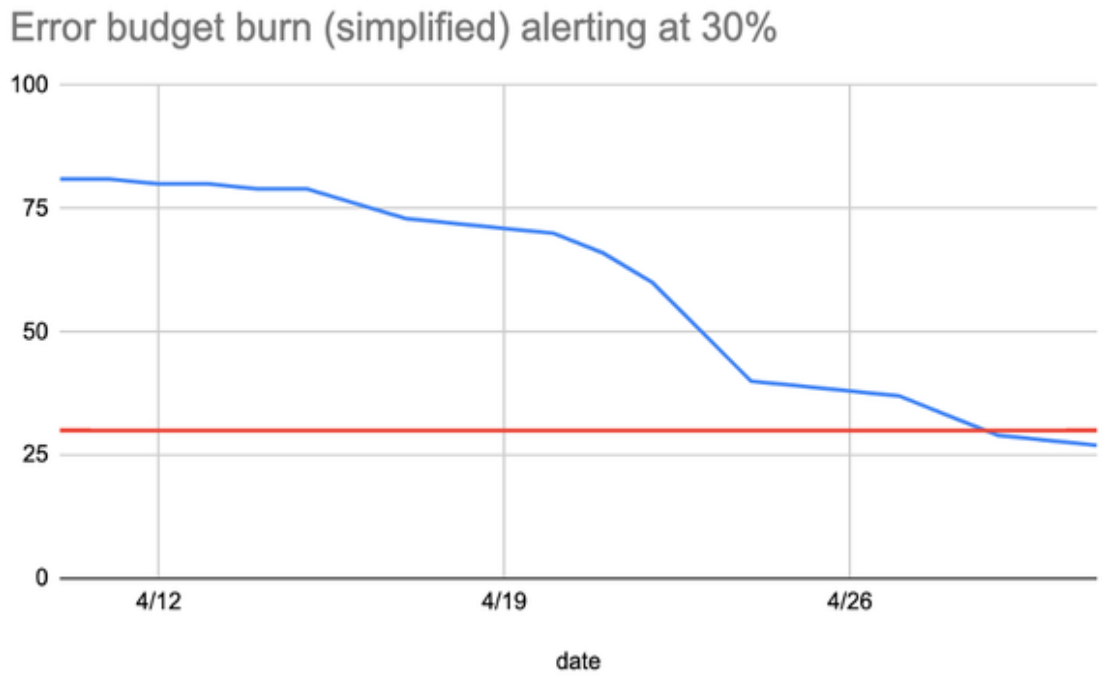


Figure 12-3. In this model, an alert would trigger when the remaining error budget (in blue) dips below the selected threshold (in red).

A challenge with this model is that it effectively just moves the goalpost by setting a different empty threshold. This type of "early warning" system can be somewhat effective, but it is crude. In practice, what happens is that after the threshold is crossed, your team acts as if the entire error budget had been spent. This model optimizes to ensure a slight bit of headroom so that your team meets its objectives. But that comes at the cost of forfeiting additional time that you could have spent delivering new features. Instead your team sits in a feature freeze while waiting for the remaining error budget to climb back up above the arbitrary threshold.

A second model for triggering alerts above the zero-level mark is to create *predictive* burn alerts. Predictive alerts forecast whether current conditions will result in burning your entire error budget.

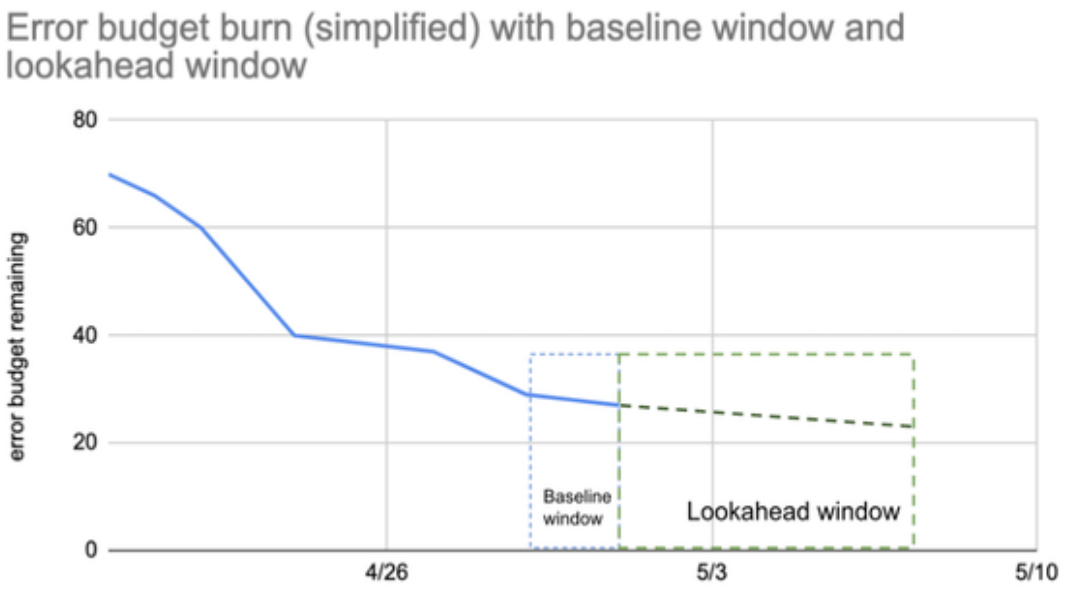


Figure 12-4. For predictive burn alerts, you need to a baseline window of recent past data to use in your model and a lookahead window that determines how far into the future your forecast extends

When using predictive burn alerts, you will need to consider the *lookahead* window and the *baseline* (or lookback) window: how far into the future are you modeling your forecast, and how much recent data should you be using to make that prediction? Let's start by considering the lookahead window since it is simpler to consider.

The lookahead window

In the predictive model, we can see that not every error that affects your error budget requires waking someone up. For example, let's say you have introduced a performance regression such that your service, with a 99.9% SLO target, is now on track to instead deliver 99.88% reliability one month from now. That is not an emergency situation requiring immediate attention. You could wait for the next business day for someone to investigate and correct the trajectory back above 99.9% reliability overall for the full month.

Conversely, if your service experiences a significant fraction of its requests failing such that you are on track to reach 98% within one hour, that should require paging the on-call engineer. If left uncorrected, such an outage

could hemorrhage your error budget for the entire month, quarter, or year within a matter of hours.

What both of these examples illustrate is that what is important to know is if your error budget will become exhausted at some point in the future, based on current trends. In the first example that happens in days and in the second that happens in minutes or hours. But what exactly does “based on current trends” mean in this context?

The scope for “current trends” depends upon how far into the future we want to forecast. On a macroscopic scale, most production traffic patterns exhibit both cyclical behavior and smoothed changes in their utilization curves. Cyclical patterns can occur in either minutely and hourly cycles. For example, a periodic cron job without jitter can influence traffic in predictable ways that repeat every few minutes. Cyclical patterns can also occur by day, week, or year. For example, in the retail industry there are seasonal shopping patterns with notable spikes around major holidays

In practice, that cyclical behavior means that some baseline windows are more appropriate to use than others. The past thirty minutes or hour are somewhat representative of the next few hours. But small minute-by-minute variations can be smoothed out when zooming out to consider daily views. Attempting to use the micro scale of past performance for the last thirty minutes or hour to extrapolate what could happen in the macro scale of performance for the next few days runs the risk of becoming flappy.

Similarly, the inverse situation is also dangerous. When a new error condition occurs, it would be impractical to wait for a full day of past performance data to become available before predicting what might happen in the next several minutes. Your error budget for the entire year could be blown by the time you make a prediction. Therefore, your baseline window should be about the same order of magnitude as your lookahead window.

In practice, we’ve found that a given baseline window can linearly predict forward by a factor of four at most without needing to add compensation for seasonality (e.g. peak/off-peak hours of day, weekday vs weekend, or end/beginning of month). That means you can get an accurate enough

prediction of whether you'll exhaust your error budget in four hours by extrapolating the past one hour of observed performance four times over. That extrapolation mechanism will be discussed in more detail in a later section of this chapter.

Extrapolating the future from current burn rate

Calculating what may happen in the lookahead window is a straightforward computation, at least for your first guess at burn rate. As you saw in the last section, the approach is to extrapolate future results based on the current burn rate: how long will it take before we exhaust our error budget?

Now that you know the approximate order of magnitude to use as a baseline, you can extrapolate results to predict the future.

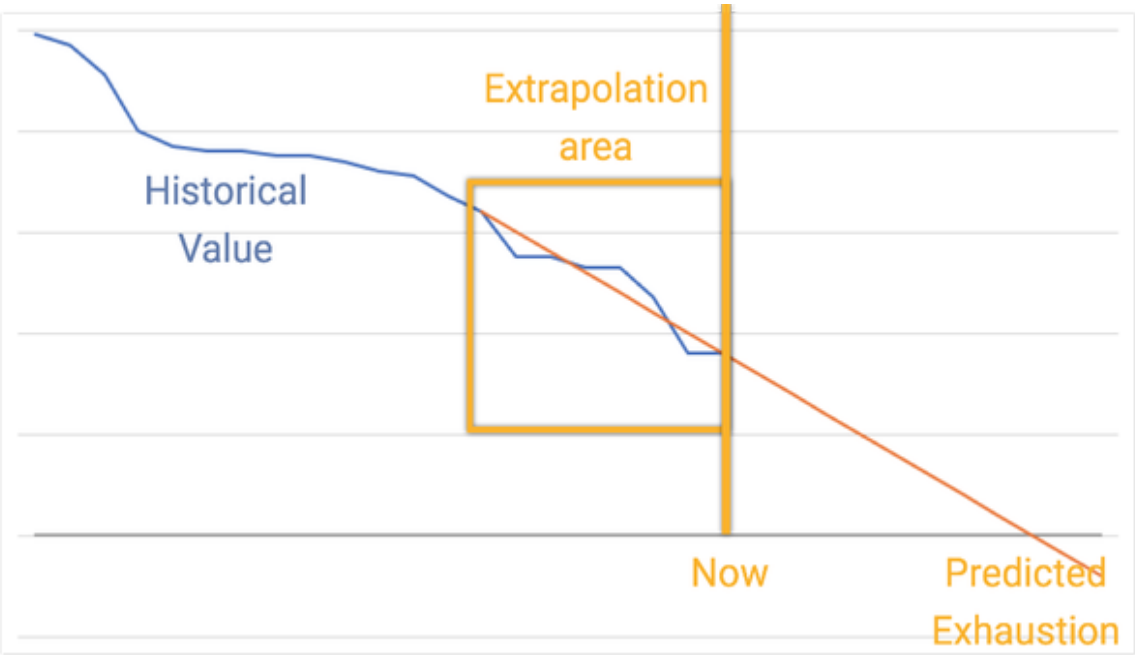


Figure 12-5. This graph shows the level of error budget (Y-axis) remaining over time (X-axis). Based on an extrapolation of a baseline window, you can predict the moment when the error budget empties.

The above graph illustrates how a trajectory can be calculated from your selected baseline to determine when your entire error budget would be completely exhausted. This is a common technique that should be familiar to anyone accustomed to making forecasts in areas like capacity planning or

project management. For example, if you use a weighted system to estimate task length in your ticketing system, you will have likely used this same approach to extrapolate when a feature might be delivered during your future sprint planning. With SLOs and error budget burn alerts, a similar logic is being applied to help prioritize production issues that require immediate attention.

Calculating a first guess is relatively straightforward. However, now is when you must weigh additional nuances when forecasting predictive burn alerts that determine the quality and accuracy of those future predictions.

In practice, there are two approaches to calculating the trajectory of predictive burn alerts. **Short-term burn alerts** extrapolate trajectories using only baseline data from the most recent time period and nothing else. **Context-aware burn alerts** take historical performance into account and use the total number of successful and failed events for the SLO's entire trailing window to make calculations.

The decision to use one method or the other typically hinges on two factors. The first, is a tradeoff between computational cost and sensitivity or specificity. Context-aware burn alerts are computationally more expensive than short-term burn alerts. However, the second factor is a philosophical stance on whether the total amount of error budget remaining should influence how responsive you are to service degradation. If you see a need to treat resolving a significant error when only 10% of your burn budget remains as having more urgency than resolving a significant error when 90% of your burn budget remains, then you may favor context-aware burn alerts.

> **NOTE**
>
> In the next two sections, we will use the word "unit" when referring to the granular building block on which SLO burn alert calculations are made. These units can be comprised of time-series data like metrics. Coarse measures like metrics have an aggregated granularity that marks a unit of time (like a minute or a second) as being either good or bad. These units can also be comprised of event-data where each individual event corresponding to a user transaction can be marked as either good or bad. In a later section, we'll examine the ramifications of which type of data is used. For now, the examples will be agnostic to data types by using arbitrarily specific "units" that correspond to "one datapoint per minute."

Let's look at examples of how decisions are made using each of those approaches.

Short-term burn alerts

When using short-term, or ahistorical, burn alerts you would record the actual number of failed service-level indicator (SLI) units along with the total number of SLI-eligible units observed over the baseline window.

You use that data to extrapolate forward, and compute how many minutes, hours, or weeks it would take to exhaust the error budget, assuming there had been no errors prior to the events observed in the baseline window, while assuming the typical number of measurement units that the SLO typically sees. Let's work an example:

Let's say you have a service with an SLO target where 99% of units will succeed over a moving 30-day window. In a typical month, the service sees 43,800 units. In the past 24 hours, the service has seen 1440 units, 50 of which failed. In the past 6 hours, the service has seen 360 units, 5 of which failed. To achieve the 99% target, only 1% of units are allowed to fail per month. Based on typical traffic volume (43,800 units) only 438 units are allowed to fail (your error budget). You want to know if, at this rate, you will burn your error budget in the next 24 hours.

A very simple short-term burn alert calculates that in the past 6 hours, you only burned 5 units. Therefore, in the next 24 hours you will burn another

20 units, totalling 25 units. You can reasonably project that you will not exhaust your error budget in 24 hours because 25 units is far less than your budget of 438 units.

A very simple short-term burn alert calculation would also consider that in the past 1 day, you've burned 50 units. Therefore, in the next 8 days you will burn another 400 units, totalling 450 units. You can reasonably project that you will exhaust your error budget in 8 days because 450 units is more than your error budget of 438 units.

The simple math above is used to illustrate projection mechanics. In practice, the situation is almost certainly more nuanced since your total amount of eligible traffic would typically fluctuate throughout the day, week, or month. In other words, you can't reliably estimate that 1 error over the past 6 hours means 4 errors over the next 24 hours if that error happens overnight, over a weekend, or whenever your service is receiving one-tenth of its median traffic levels. If it were the case that this error happened while you saw a tenth of the traffic, you might instead expect to see something closer to 30 errors over the next 24 hours since that curve would smooth with rising traffic levels.

So far, in this example, we've been using linear extrapolation for simplicity. But proportional extrapolation is far more useful in production. Consider this next change to the example situation above.

Let's say you have a service with an SLO target where 99% of units will succeed over a moving 30-day window. In a typical month, the service sees 43,800 units. In the past 24 hours, the service has seen 1440 units, 50 of which failed. **In the past 6 hours, the service has seen 50 units, 25 of which failed**. To achieve the 99% target, only 1% of units are allowed to fail per month. Based on typical traffic volume (43,800 units) only 438 units are allowed to fail (your error budget). You want to know if, at this rate, you will burn your error budget in the next 24 hours.

Extrapolating linearly, you would calculate that 25 failures in the past 6 hours mean 100 failures in the next 24 hours, totalling 105 failures which is far less than 438.

Using proportional extrapolation, you would calculate that in any given 24 hour period you would expect to see 43,800 units / 30 days, or 1440 units. In the last 6 hours, 25 of 50 units (50%) failed. If that ***proportional failure rate*** of 50% continues for the next 24 hours, you will burn 50% of 1440 units, or 720 units: far more than your budget of 438 units. With this proportional calculation, you can reasonably project that your error budget will be exhausted in about half a day. An alert should trigger to notify an on-call engineer to investigate immediately.

Context-aware burn alerts

When using context-aware, or historical, burn alerts you would keep a rolling total of the number of good and bad events that have happened over the entire window of the SLO, rather than just the baseline window. This section unpacks the various calculations you need to make for effective burn alerts. But we should note that you will want to tread carefully when practicing these techniques. The computational expense of calculating these values at each evaluation interval can quickly become financially expensive as well. At Honeycomb, we found this out the hard way when small SLO datasets suddenly started to rack up over $5,000 of AWS Lambda costs per day. :-)

To see how the considerations are different for context-aware burn alerts, let's work an example:

Let's say you have a service with an SLO target where 99% of units will succeed over a moving 30-day window. In a typical month, the service sees 43,800 units. **In the previous 26 days, you have already failed 285 units out of 37,960. In the past 24 hours, the service has seen 1460 units, 130 of which failed.** To achieve the 99% target, only 1% of units are allowed to fail per month. Based on typical traffic volume (43,800 units) only 438 units are allowed to fail (your error budget). You want to know if, at this rate, you will burn your error budget in the next four days.

In this example, you want to project forward on a scale of days. Using the maximum practical extrapolation factor of 4 (see note above), you set a

baseline window that examines the last 1 day's worth of data to extrapolate forward 4 days from now.

You must also consider the impact your chosen scale has on your sliding window. If your SLO is a sliding 30 day window, then your adjusted lookback window would be 26 days: 26 look back days + 4 extrapolated days = your 30 day sliding window.

Figure 12-6. Shows an adjusted SLO 30-day sliding window. When projecting forward 4 days, the lookback period to consider must be shortened to 26 days before today. The projection is made by replicating results from the baseline window for the next four days and adding those to the adjusted sliding window.

With those adjusted timeframes defined, you can now calculate how the future looks four days from now. To do that calculation you would:

- Examine every entry in the map of SLO events that has occurred in the past 26 days
- Store both the total number of events in the past 26 days and the total number of errors
- Re-examine map entries that occurred within that last 1 day to determine the baseline window failure rate

- Extrapolate the next 4 days of performance by presuming they will behave similarly to the 1-day baseline window
- Calculate the adjusted SLO 30-day sliding window as it would appear 4 days from now
- Trigger an alert if the error budget would be exhausted by then

Using a Go-based code example, you could accomplish this task if you had a given timeseries map containing containing the number of failures and successes in each window:

```
func isBurnViolation(now time.Time, ba *BurnAlertConfig, slo
*SLO, tm *timeseriesMap) bool {
        // For each entry in the map, if it's in our adjusted
window, add its totals.
        // If it's in our projection use window, store the rate.
        // Then project forward and do the SLO calculation.
        // Then fire off alerts as appropriate.
        // Compute the window we will use to do the projection,
with the projection offset as the earliest bound.
        pOffset :=
time.Duration(ba.ExhaustionMinutes/lookbackRatio) * time.Minute
        pWindow := now.Add(-pOffset)
        // Set the end of the total window to the beginning of
the SLO time period, plus ExhaustionMinutes.
        tWindow := now.AddDate(0, 0, -
slo.TimePeriodDays).Add(time.Duration(ba.ExhaustionMinutes) *
time.Minute)
        var runningTotal, runningFails int64
        var projectedTotal, projectedFails int64
        for i := len(tm.Timestamps) - 1; i >= 0; i-- {
                t := tm.Timestamps[i]
                // We can stop scanning backwards if we've run
off the end.
                if t.Before(tWindow) {
                        break
                }
                runningTotal += tm.Total[t]
                runningFails += tm.Fails[t]
                // If we're within the projection window, use
this value to project forward, counting it an extra lookbackRatio
times.
                if t.After(pWindow) {
                        projectedTotal += lookbackRatio *
```

```
tm.Total[t]
                        projectedFails += lookbackRatio *
tm.Fails[t]
                }
        }
        projectedTotal = projectedTotal + runningTotal
        projectedFails = projectedFails + runningFails
        allowedFails := projectedTotal * int64(slo.BudgetPPM) /
int64(1e6)
        return projectedFails != 0 && projectedFails >=
allowedFails
}
```

To calculate an answer to the example problem, you note that in the past 26 days there have already been 37,960 units and 285 failed. The baseline window of 1 day will be used to project forward 4 days. At the start of the baseline window, you are at day 25 of the data set (1 day ago).

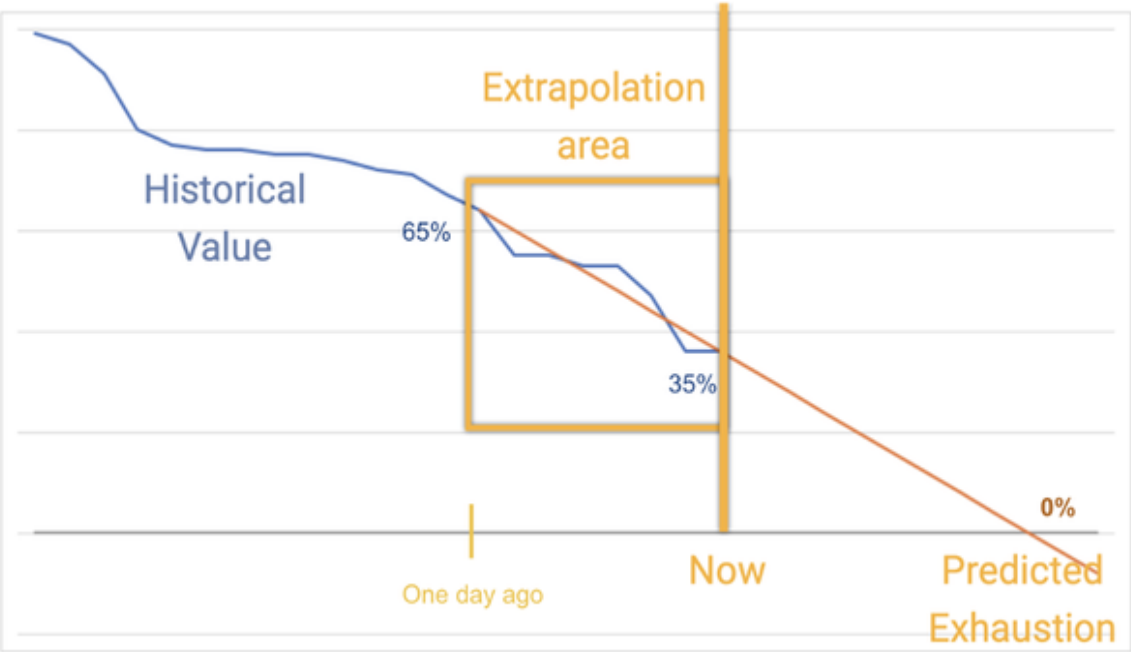


Figure 12-7. shows that one day ago, 65% of the error budget remained. Now, only 35% of the error budget remains. At the current rate, the error budget will be exhausted in less than two days.

Visualizing results as a graph, you see that when the one-day baseline window started you still had 65% of your error budget remaining. Right now, you only have 35% of your error budget remaining. In one day, you've burned 30% of your error budget. If you continue to burn your error budget

at that rate, it will be exhausted in far less than 4 days. You should trigger an alarm.

The baseline window

Having considered the lookahead window and seeing how it's used, let's now shift our attention back to the baseline window.

There's an interesting question of how *much* of a baseline to use when extrapolating forward. If the performance of a production service suddenly turns downhill, you want to be aware of that fairly quickly. At the same time, you also don't want to set such a small window that creates noisy alerts. If you used a baseline of fifteen minutes to extrapolate three days of performance, a small blip in errors could trigger a false alarm. Conversely, if you used a baseline of three days to extrapolate forward an hour, you'd be unlikely to notice a critical failure until it's too late.

An earlier sidebar describes a practical choice as a factor of four for the timeline being considered. You may find that other more sophisticated algorithms can be used, but our experience has indicated that the factor of four is generally reliable. The baseline multiplier of 4 also gives us a handy heuristic: a 24 hour alarm is based on the last six hours of data, a 4 hour alarm is based on the last one hours of data, and so forth. It's a good place to start when first approaching SLO burn alerts.

There's also an odd implication worth mentioning when setting baseline windows proportionally: it becomes possible for alarms to trigger in seemingly illogical order. For example, a system issue might trigger a burn alert that says you'll exhaust your budget within two hours, but it won't trigger a burn alert that says you'll exhaust it within a day. That's because the extrapolation for those two alerts comes from different baselines. In other words, "if you keep burning at the rate from the last 30 minutes, you'll exhaust in two hours—but if you keep burning at the rate from the last day, you'll be fine for the next four days."

For these reasons, you should measure both and you should act whenever either projection triggers an alert. Otherwise, you risk potentially waiting

hours to surface a problem on the burndown timescale of one day—when a burn alert with hour-level granularity means you could find out sooner. Launching corrective action at that point could also avert triggering the one-day projection alert. Conversely, while an hourly window allows teams to respond to sudden changes in burn rate, it's far too noisy to facilitate task prioritization on a longer timescale, like sprint planning.

That last point is worth dwelling on for a moment. To understand why both timescales should be considered, let's look at what actions should be taken when an SLO burn alert is triggered.

Acting on SLO burn alerts

In general, when a burn alert is triggered teams should initiate an investigative response. For a more in-depth look at various ways to respond, refer to Alex Hidalgo's *Implementing Service Level Objectives*. For this section, we'll examine a few broader questions you should ask when considering response to a burn alert.

First, you should diagnose the type of burn alert you received. Is it a new and unexpected type of burn happening? Or is it a more gradual and expected burn? Let's first look at different patterns for how error budgets can be consumed.

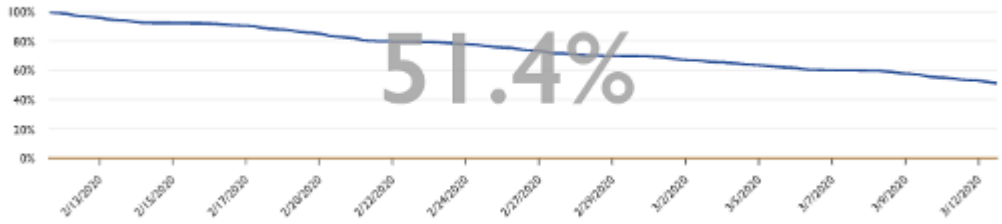


Figure 12-8. Shows a gradual error budget burn rate, with 51.4% remaining.

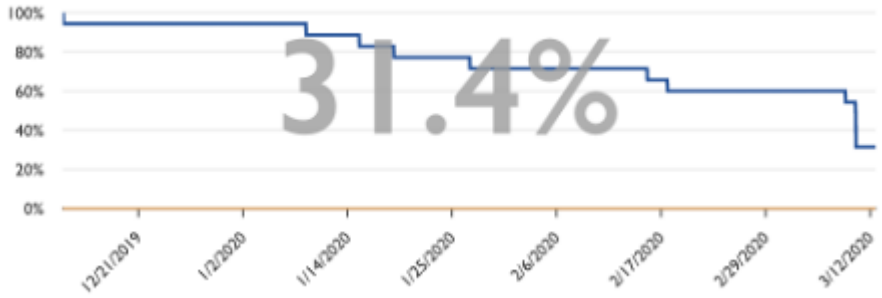


Figure 12-9. Shows an error budget of 31.4% remaining, that is burned in bursts.

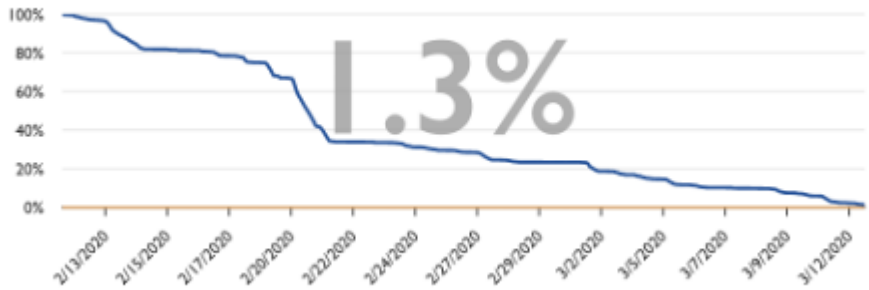


Figure 12-10. Shows a system that usually burns slowly, but had an incident that burned a significant portion of the error budget all at once, with only 1.3% remaining.

The charts above all display **error budget remaining** over **time**, for the duration of the SLO window. Gradual error budget burns, that occur at a slow but steady pace, are a characteristic of most modern production systems. You expect a certain threshold of performance, and a small number of exceptions fall out of that. In some situations, there may be period disturbances that cause exceptions to occur in bursts.

When a burn alert is triggered, you should assess whether it is part of a burst condition or whether it is an incident that could burn a significant portion of your error budget all at once. Comparing the current situation to historical rates can add helpful context for triaging its importance.

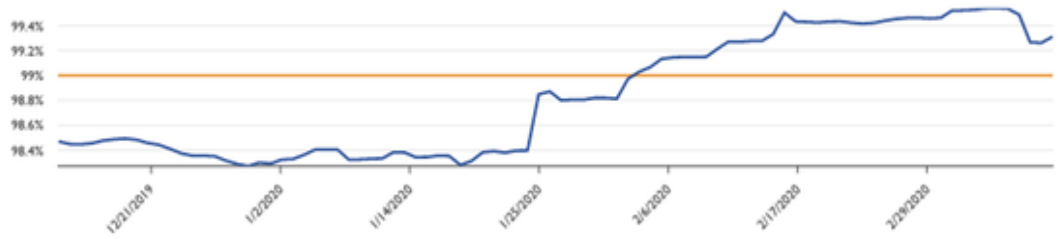


Figure 12-11. Shows a 90-day sliding window for an SLO that performed below a 99% target before recovering toward the start of February.

Instead of showing the instantaneous 31.4% remaining and how we got there over the trailing 30 days as we did in *Figure 12-9*, in *Figure 12-11* we zoom out to examine the 30-day-cumulative state of the SLO for each day in the past 90 days. Around the beginning of February, this SLO started to recover above its target threshold. Likely, in part, because a large dip in performance aged out of the 90-day window.

Understanding the general trend of the SLO can also answer questions about how urgent the incident feels—and can give a hint of how to solve it. Burning budget all at once suggests a different sort of failure than burning budget slowly over time.

Observability data for SLOs vs. time series data

Using time series data for SLOs introduces a few complications. In the examples in earlier sections, we used a generic "unit" to consider success or failure. In an event-based world, that unit is an individual user request: did this request succeed or fail? In a time series world, that unit is a particular slice of time: did the entire system succeed or fail during this slice of time? When using time series data, an aggregate view of system performance is what gets measured.

For stringent SLOs—those with 99.99% availability targets and higher—error budgets can be exhausted within minutes or seconds. In that scenario, every second especially matters and several mitigation steps (for example,

automated remediations) must be enacted in order to achieve that target. The difference between finding out about critical error budget burn in one second vs one minute won't matter much if it still takes a responding human three minutes to acknowledge the alert, unlock their laptop, log in, and manually run the first remediation they conjur (or even more time if the issue is complex and requires further debugging).

Another mitigation can come by way of using event data for SLOs rather than time series data. Consider how time series evaluation functions in the context of stringent SLOs. For example, let's say you have an SLI that specifies requests must have a p95 less than 300 milliseconds. Error budget calculation then needs to classify short time periods as either good or bad. In the case of evaluating on a per-minute basis, the calculation must wait until the end of the elapsed minute to declare it either good or bad. If the minute is declared bad, you've already lost that minute's worth of response time. This is a non-starter for initiating a fast alerting workflow when error budgets can be depleted within seconds or minutes.

The granularity of "good minute" or "bad minute" is simply not enough to measure a four-9s SLO. In the example above, if only 94% of requests were less than 300ms, the entire minute is declared bad. That one evaluation is sufficient enough to burn 25% of your 4.32 minute monthly error budget in one shot. And a more common 5-minute good/bad evaluation approach can only fail one single evaluation window before you are in breach of your SLO target.

In systems that use time series data for SLO calculation, a mitigation approach might look something like setting up a system where once three synthetic probes fail, automatic remediations are deployed and humans are then expected to double-check the results afterward. Another approach can also be measuring with metrics the instantaneous percentage of requests that are failing in a shorter window of time, given a sufficiently high volume of traffic.

Rather than making aggregate decisions in the good minute / bad minute scenario, using event data to calculate SLOs gives you request-level

granularity to evaluate system health. Using the same example SLI scenario above, any single request with duration lower than 300ms is considered good and any with duration greater than 300ms is considered bad. In a scenario where 94% of requests were less than 300ms, only 6% of those requests are subtracted from your SLO error budget, rather than the 100% that would be subtracted using an aggregate time series measure like p95.

In modern distributed systems, where 100% total failure blackouts are less common than partial failure brownouts, event based calculations are much more useful. Consider that a 1% brownout of a 99.99% reliability system is similar to measuring a 100% outage of a 99% reliability system. In both cases, you have 7+ hours to remediate before the monthly error budget is exhausted.

Observability data that traces actual user experience with your services is a more accurate representation of system state than what is shown by coarsely aggregated time series data. When deciding which data to use for actionable alerting, using observability data enables teams to focus on conditions that are much closer to the truth of what the business cares about: the overall customer experience that is occurring at this moment.

It's important to align your reliability goals with what's feasible for your team. For very stringent SLOs—such as a five-9s system—you must deploy every mitigation strategy available such as automatic remediation based on extremely granular and accurate measurements that bypass traditional human-involved fixes that may take minutes or hours. However, even for teams with less stringent SLOs, using granular event-based measures can create more response time buffers than are typically available with time-series-based measures. That additional buffer ensures reliability goals are much more feasible, regardless of your team's SLO targets.

Conclusion

We've examined the role error budgets play and the mechanisms available to trigger alerts when using SLOs. There are several forecasting methods available that can be used to predict when your error budget will be burned.

Each method has its own considerations and tradeoffs and the hope is that this chapter shows you which method to take to best meet the needs of your specific organization.

While not specific to observability, using SLOs to drive alerting can be a productive way to make alerting less noisy and more actionable. SLIs can be defined to measure how customers experience a service in ways that directly align with business objectives. Error budgets set clear expectations between business stakeholders and engineering teams. Error budget burn alerts enable teams to ensure a high degree of customer satisfaction, align with business goals, and initiate an appropriate response to production issues without the kind of cacophony that exists in the world of symptom-based alerting, where an excessive alert storm is the norm.

What is specific to observability is the additional power that event data adds to the SLO model. When calculating error budget burn rates, events provide a more accurate assessment of the actual state of production services. Additionally, merely knowing that an SLO is in danger of breach does not necessarily provide you with the insight you need to determine which users are impacted, which dependent services are affected, or which combinations of behavior by users are triggering errors in your service. Coupling observability data with SLOs helps you see where and when failures happened after a burn budget alert is triggered.

Using SLOs with observability data is an important component of both the SRE approach and the observability-driven-development approach. As seen in previous chapters, analyzing events that fail can give rich and detailed information about what is going wrong, and why. It can help triage between systemic problems, and occasional stochastic failures. In the next chapter, we'll more closely examine the observability debugging loop and how it enables debugging from first principles.

Chapter 13. Cheap and Accurate Enough: Sampling

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 18th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

So far in this section, we've covered the challenges of routing and storing large quantities of observability data when operating at scale. In this section, we cover strategies for reducing the amount of data that needs to be stored or routed. At a large enough scale, the resources necessary to retain and process every single event can become prohibitive and impractical. Sampling events can mitigate the tradeoffs between resource consumption and data fidelity.

This chapter examines why sampling is useful (even at a smaller scale), the various strategies typically used to sample data, and tradeoffs between those strategies. We use code-based examples to illustrate how these strategies are implemented and progressively introduce concepts that build upon previous examples. The chapter starts with simpler sampling schemes applied to single events as a conceptual introduction to using a statistical representation of data when sampling. We then build toward more complex sampling strategies as they are applied to a series of related events (i.e. trace

events) and how to propagate the information needed to reconstruct your data after sampling.

Sampling to refine your data collection

Past a certain scale, the cost to collect, process, and save every log entry, every event, and every trace that your systems generate dramatically outweighs the benefits. At a large enough scale, it is simply not feasible to run an observability infrastructure that is the same size as your production infrastructure. When observability events quickly become a flood of data, the challenge pivots to a tradeoff between scaling back how much data is kept and potentially losing the crucial information your engineering team needs to troubleshoot and understand your system's production behaviors.

The reality of most applications is that many of their events are virtually identical and successful. The core function of debugging is searching for emergent patterns or examining failed events during an outage. Through that lens, it is then less useful to transmit 100% of all events to your observability data backend. Certain events can be selected as representative examples of what occurred, and those sample events can be transmitted along with metadata your observability backend needs to reconstruct what actually occurred among the events that weren't sampled.

In order to debug effectively, what's needed is a representative sample of successful or "good" events, against which to compare the "bad" events. Using representative events to reconstruct your observability data enables you to reduce the overhead of transmitting every single event, while also faithfully recovering the original shape of that data. Sampling events can help you accomplish your observability goals at a fraction of the resource cost. It is a way to refine the observability process at scale.

Historically in the software industry, when facing resource constraints in reporting high-volume system state, the standard approach to surfacing the signal from the noise has been to generate aggregated metrics containing a limited number of tags. As covered in chapter 2, aggregated views of system state are far too coarse to troubleshoot the needs of modern

distributed systems. Pre-aggregating data before it arrives in your debugging tool means that you can't dig further past the granularity of the aggregated values.

With observability, you can sample events using the strategies outlined in this chapter and still provide granular visibility into system state. Sampling gives you the ability to decide which events are useful to transmit and which ones are not. Unlike pre-aggregated metrics that collapse all events into one coarse representation of system state over a given period of time, sampling allows you to make informed decisions about which events can help you surface unusual behavior, while still optimizing for resource constraints.

At scale, the need to refine your data set to optimize for resource costs becomes critical. But even at a smaller scale, where the need to shave resources is less pressing, refining the data you decide to keep can still provide valuable cost savings. First, let's start by looking at the strategies that can be used to decide which data is worth sampling. Then, we'll look at when and how that decision can be made when handling trace events.

Different approaches to sampling

Sampling is a common approach when solving for resource constraints. Unfortunately, the term "sampling" is used broadly as a one-size-fits-all label applied to the many different types of approaches that might be implemented. We need to disambiguate the term by naming each sampling technique and distinguishing how they are different, and similar, to each other.

Constant-probability sampling

Because of its understandable and easy-to-implement approach, constant-probability sampling is what most people think of when they think of sampling: a constant percentage of data is kept vs discarded (e.g. keep one out of every ten requests).

When performing analysis of sampled data, you will need to transform the data to reconstruct the original distribution of requests. Suppose that your service is instrumented with both events and with metrics, and receives 100,000 requests. It is misleading for your telemetry systems to report receiving only 100 events, if each received event represents approximately 1,000 similar events. Other telemetry systems such as metrics will record incremented counters for each of the 100,000 requests that your service has received. Only a fraction of those requests will have been sampled, so your system will need to adjust the aggregation of events to return data that is approximately correct. For a fixed sampling rate system, you can multiply each event by the sampling rate in effect to get the estimated count of total requests and sum of their latency. Scalar distribution properties such as the p99 and median do not need to be adjusted for a constant probability sampling, as they are not distorted by the sampling process.

The basic idea of constant sampling is that, if you have enough volume, any error that comes up will happen again. If that error is happening enough to matter, then you'll see it. However, if you have a moderate volume of data, constant sampling does not maintain the statistical likelihood that you still see what you need to see. Constant sampling is not effective if:

- You care a lot about error cases and not very much about success cases
- Some customers send orders of magnitude more traffic than others, and you want all customers to have a good experience
- You want to ensure that a huge increase in traffic on your servers can't overwhelm your analytics backend

For an observable system, a more sophisticated approach ensures enough data is captured to understand the state of any given service at any given time.

Sampling on recent traffic volume

Instead of using a fixed probability, you can dynamically adjust the rate at which your system samples events. Sampling probability can be adjusted upwards if less total traffic was received recently, or decreased if a traffic surge is likely to overwhelm the backend of your observability tool. However, this approach adds a new layer of complexity: without a constant sampling rate, you can no longer multiply out each event by a constant factor when reconstructing the distribution of your data.

Instead, your telemetry system will need to use a weighted algorithm that accounts for the sampling probability in effect at the time the event was collected. If one event represents 1,000 similar events, it should not be directly averaged with another event that represents 100 similar events. Thus, for calculating a count, your system must add together the number of *represented* events, not just the number of collected events. For aggregating distribution properties such as median or p99, your system must expand out each event into many when calculating the total number of events and where the percentile values lie. [to work on converting diagram into words, or just work an actual example]

[1111115555555599]

Sampling based on event content (keys)

This dynamic sampling approach involves tuning the sample rate based on event payload. At a high level, that means choosing one or more fields in the set of events and designating a sample rate when a certain combination of values is seen. For example, you could partition events based on HTTP response codes, and assign sample rates to each response code. Doing that allows you to specify sampling conditions such as:

- Events with *errors* are more important than those with *successes*
- Events for *newly-placed orders* are more important than those checking on *order status*
- Events affecting *paying customers* are more important to keep than those for customers using the *free tier*

With this approach, you can make the keys as simple (e.g. HTTP method) or complicated (e.g. concatenating the HTTP method, request size, and user-agent) as needed to select samples that can provide the most useful view possible into service traffic.

Sampling at a constant percentage rate based on event content alone works well when the key space is small (e.g. there are a finite number of HTTP methods: GET, POST, HEAD, etc.) *and* when the relative rates of given keys stay consistent — for example, when you can assume errors are less frequent than successes.

Combining per-key and historical methods

When the content of traffic is harder to predict, instead you can continue to identify a key (or combination of keys) for each incoming event, and then *also* dynamically adjust the sample rate for each key based on the volume of traffic recently seen for that key. For example, base the sample rate on the number of times a given key combination (such as [customer ID, dataset ID, error code]) is seen in the last 30 seconds. If a specific combination is seen many times in that time, it's less interesting than combinations that were seen less often. A configuration like that allows proportionally fewer of the events to be propagated verbatim until that rate of traffic changes and it adjusts the sample rate again.

Choosing dynamic sampling options

To decide which sampling strategy to use, it helps to look at the traffic flowing through a service, as well as the variety of queries hitting that service. Are you dealing with a front page app where 90% of the requests hitting it are nearly indistinguishable from one-another? The needs of that situation will differ substantially from dealing with a proxy fronting a database where many query patterns are repeated. A backend behind a read-through cache, where each request is mostly unique (with the cache already having stripped all the boring ones away) will have different needs from

those two. Each of these situations benefit from a slightly different sampling strategy that optimizes for their needs.

When to make a sampling decision for traces

So far, each of the strategies above has been considered *what* criteria to use when selecting samples. For events involving an individual service, that decision solely depends on the criteria above. For trace events, *when* a sampling decision gets made is also important.

Trace spans are collected across multiple services; with each service, potentially, employing its own unique sample strategy and rate. The probability that every span necessary to complete a trace will be the event that each service chooses to sample is relatively low. In order to ensure every span in a trace is captured, special care must be taken depending on when the decision on whether to sample is made.

As covered earlier, one strategy is to use a property of the event itself — such as the return status, latency, endpoint, or a high cardinality field like customer ID — to decide if it is worth sampling. Some properties within the event, such as endpoint or customer ID, are static and known at the start of the event. In "*head-based*" sampling (or, "upfront" sampling), a sampling decision is made when the trace event is initiated. That decision is then propagated further downstream (e.g. by inserting a "require sampling" header bit) to ensure every span necessary to complete the trace is sampled.

Some fields, like return status or latency, are known only in retrospect after event execution has completed. If a sampling decision relies on dynamic fields, by the time those are determined, each underlying service will have already independently chosen whether or not to sample other span events. At best, you may end up keeping the downstream spans deemed interesting outliers, but none of the other context. Properly making a decision on values known only at the end of a request requires "*tail-based*" sampling.

To collect full traces in the tail-based approach, all spans must first be collected in a buffer and then, retrospectively, a sampling decision can be made. That buffered sampling technique is computationally expensive and

not feasible in practice entirely from within the instrumented code. Buffered sampling techniques typically require external collector-side logic.

Additional nuances exist for determining the *what* and *when* of sampling decisions. But at this point, those are best explained using code-based examples.

Translating sampling strategies into code

So far, we've covered sampling strategies on a conceptual level. Let's look at how these strategies are implemented in code. This pedagogical example uses Go to illustrate implementation details, but the examples would be straightforward to port into any language that supports hashes/dicts/maps, pseudorandom number generation, and concurrency/timers.

The base case

Let's suppose you would like to instrument a high-volume handler that calls a downstream service, performs some internal work, then returns a result and unconditionally records an event to your instrumentation sink:

```
func handler(resp http.ResponseWriter, req *http.Request) {
        start := time.Now()
        i, err := callAnotherService()
        resp.Write(i)
        RecordEvent(req, start, err)
}
```

At scale, this instrumentation approach is unnecessarily noisy and would result in sky-high resource consumption. Let's look at alternate ways of sampling the events this handler would send.

Fixed-rate sampling

A naive approach might be probabilistic sampling using a fixed rate, by randomly choosing to send 1 in 1000 events.

```
var sampleRate = flag.Int("sampleRate", 1000, "Static sample
rate")
func handler(resp http.ResponseWriter, req *http.Request) {
        start := time.Now()
        i, err := callAnotherService()
        resp.Write(i)
        r := rand.Float64()
        if r < 1.0 / *sampleRate {
                RecordEvent(req, start, err)
        }
}
```

Every 1000th event would be kept, regardless of its relevance, as representative of the other 999 events discarded. To reconstruct your data on the backend, you would need to remember that each event stood for `sampleRate` events and multiply out all counter values accordingly on the receiving end at the instrumentation collector. Otherwise your tooling would misreport the total number of events actually encountered during that time period.

Recording the sample rate

In the clunky example above, you would need to manually remember and set the sample rate at the receiving end. What if you need to change the sample rate value at some point in the future? The instrumentation collector wouldn't know exactly when the value changed. A better practice is to explicitly pass the current `sampleRate` when sending a sampled event — indicating the event statistically represents `sampleRate` similar events. Note that sample rates can not only vary between services, but also vary within a single service as well.

Status Code	Time	Sample-Rate
ok	1:00	100
ok	1:00	100
err	1:00	1
ok	1:01	80
err	1:01	1
ok	1:01	80
ok	1:01	80

Figure 13-1. Different events may be sampled at different rates

Recording the sample rate within an event can look like this:

```
var sampleRate = flag.Int("sampleRate", 1000, "Service's sample
rate")
func handler(resp http.ResponseWriter, req *http.Request) {
        start := time.Now()
        i, err := callAnotherService()
        resp.Write(i)
        r := rand.Float64()
        if r < 1.0 / *sampleRate {
                RecordEvent(req, *sampleRate, start, err)
        }
}
```

With this approach, you can keep track of the sampling rate in effect when each sampled event was recorded. That gives you the data necessary to accurately calculate values when reconstructing your data, even if the sampling rate dynamically changes. For example, if you were trying to calculate the total number of events meeting a filter such as “err != nil”, you would multiply the count of seen events with “err != nil” by each one’s `sampleRate`. And, if you were trying to calculate the sum of `durationMs`, you would need to weight each sampled event’s `durationMs` and multiply it by `sampleRate` before adding up the weighted figures.

Status Code	Time	COUNT (reweighted)
ok	1:00	200
err	1:00	1
ok	1:01	240
err	1:01	1

Figure 13-2. Total events calculated using weighted numbers

This example is simplistic and contrived. Already, you may be seeing flaws in this approach when it comes to handling trace events. In the next section, let's look at additional considerations for making dynamic sampling rates and tracing work well together.

Consistent sampling

So far in our code, we've looked at *how* a sampling decision is made. But we have yet to consider *when* a sampling decision gets made in the case of sampling trace events. The strategy of using head-based, tail-based, or buffered sampling matters when considering how sampling interacts with tracing. We'll cover how those decisions get implemented toward the end of the chapter. For now, let's examine how to propagate context to downstream handlers in order to (later) make that decision.

To properly manage trace events, you should use a centrally generated "sampling/tracing ID" propagated to all downstream handlers instead of independently generating a sampling decision inside of each one,. Doing so lets you make consistent sampling decisions between different manifestations of the same end user's request. It would be unfortunate to discover that you have sampled an error far downstream for which the upstream context is missing because it was dropped because of how your sampling strategy was implemented.

HASH(TraceId)	SAMPLED?
8464143	
9976727	
2046000	YES
8697994	
1983000	YES
3427217	
6152331	
4919000	YES
6122453	

Figure 13-3. Sampled events containing a TraceId

Consistent sampling guarantees that if a 1:100 sampling occurs, a 1:99, 1:98, etc. sampling preceding or following it also preserves the execution context. And half of the events chosen by a 1:100 sampling will be present under a 1:200 sampling.

Let's modify the previous code sample to read a value for sampling probability from the TraceID/Sampling-ID, instead of generating a random value at each step.

```
var sampleRate = flag.Int("sampleRate", 1000, "Service's sample
rate")
func handler(resp http.ResponseWriter, req *http.Request) {
        // Use an upstream-generated random sampling ID if it
exists.
        // otherwise we're a root span. generate & pass down a
random ID.
        var r float64
        if r, err := floatFromHexBytes(req.Header.Get("Sampling-
ID")); err != nil {
                r = rand.Float64()
        }
        start := time.Now()
        // Propagate the Sampling-ID when creating a child span
        i, err := callAnotherService(r)
        resp.Write(i)
```

```
        if r < 1.0 / *sampleRate {
                RecordEvent(req, *sampleRate, start, err)
        }
}
```

Now, by changing the `sampleRate` feature flag to cause a different proportion of traces to be sampled, you have support for adjusting the sample rate without recompiling, including at runtime. However, if you adopt the technique we'll discuss next, Target Rate Sampling, you won't need to manually adjust the rate.

Target Rate Sampling

You don't need to manually flag-adjust the sampling rates for each of your services as traffic swells and sags; instead, you can automate this by tracking the incoming request rate that you're receiving.

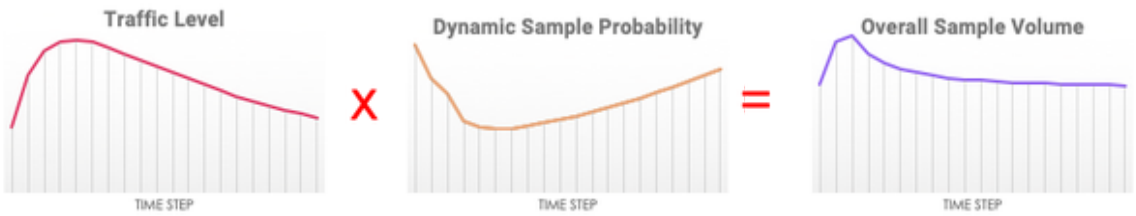


Figure 13-4. Automating the calculation of overall sample volume

Let's see how that is done in our code example.

```
var targetEventsPerSec = flag.Int("targetEventsPerSec", 5, "The
target number of requests per second to sample from this
service.")
// Note: sampleRate can be a float! doesn't have to be an
integer.
var sampleRate float64 = 1.0
// Track requests from previous minute to decide sampling rate
for the next minute.
var requestsInPastMinute *int
func main() {
        // Initialize counters.
        rc := 0
        requestsInPastMinute = &rc
        go func() {
                for {
```

```
                        time.Sleep(time.Minute)
                        newSampleRate = *requestsInPastMinute /
(60 * *targetEventsPerSec)
                        if newSampleRate < 1 {
                                sampleRate = 1.0
                        } else {
                                sampleRate = newSampleRate
                        }
                        newRequestCounter := 0
                        // Real production code would do
something less prone to race conditions
                        requestsInPastMinute = &newRequestCounter
                }
        }()
        http.Handle("/", handler)
        [...]
}
func handler(resp http.ResponseWriter, req *http.Request) {
        var r float64
        if r, err := floatFromHexBytes(req.Header.Get("Sampling-
ID")); err != nil {
                r = rand.Float64()
        }
        start := time.Now()
        *requestsInPastMinute++
        i, err := callAnotherService(r)
        resp.Write(i)
        if r < 1.0 / sampleRate {
                RecordEvent(req, sampleRate, start, err)
        }
}
```

This example provides predictable experience in terms of resource cost. However, the technique still lacks flexibility for sampling at variable rates depending upon the volume of each key.

Having more than one static sample rate

If the sampling rate is high, whether due to being dynamically or statically set high, you need to consider that you could miss long tail events — for instance, errors or high latency events — because the chance that a 99.9th percentile outlier event will be chosen for random sampling is slim. Likewise, you may want to have at least some data for each of your distinct

sources, rather than have the high-volume sources drown out the low-volume ones.

A remedy for that scenario is to set more than one sample rate. Let's start by varying the sample rates by key. Here, the example code samples any baseline (non-outlier) events at a rate of 1 in 1000 and chooses to tail sample any errors or slow queries at a rate of 1 in 1 and 1 in 5, respectively.

```
var sampleRate = flag.Int("sampleRate", 1000, "Service's sample
rate")
var outlierSampleRate = flag.Int("outlierSampleRate", 5, "Outlier
sample rate")
func handler(resp http.ResponseWriter, req *http.Request) {
        start := time.Now()
        i, err := callAnotherService(r)
        resp.Write(i)
        r := rand.Float64()
        if err != nil || time.Since(start) > 500*time.Millisecond
{
                if r < 1.0 / *outlierSampleRate {
                        RecordEvent(req, *outlierSampleRate,
start, err)
                }
        } else {
                if r < 1.0 / *sampleRate {
                        RecordEvent(req, *sampleRate, start, err)
                }
        }
}
```

While a good example of using multiple static sample rates, that approach is still susceptible to spikes of instrumentation traffic. If the application experiences a spike in the rate of errors, every single error gets sampled. Next, we will address that shortcoming with Target Rate sampling.

Sampling by key and target rate

Putting two previous techniques together, let's extend what we've already done to target specific rates of instrumentation. If a request is anomalous (for example, has latency above 500ms or, is an error), it can be designated

for tail sampling at its own guaranteed rate, while rate-limiting the other requests to fit within a budget of sampled requests per second.

```
var targetEventsPerSec = flag.Int("targetEventsPerSec", 4, "The
target number of ordinary requests per second to sample from this
service.")
var outlierEventsPerSec = flag.Int("outlierEventsPerSec", 1, "The
target number of outlier requests per second to sample from this
service.")
var sampleRate float64 = 1.0
var requestsInPastMinute *int
var outlierSampleRate float64 = 1.0
var outliersInPastMinute *int
func main() {
        // Initialize counters.
        rc := 0
        requestsInPastMinute = &rc
        oc := 0
        outliersInPastMinute = &oc
        go func() {
                for {
                        time.Sleep(time.Minute)
                        newSampleRate = *requestsInPastMinute /
(60 * *targetEventsPerSec)
                        if newSampleRate < 1 {
                                sampleRate = 1.0
                        } else {
                                sampleRate = newSampleRate
                        }
                        newRequestCounter := 0
                        requestsInPastMinute = &newRequestCounter
                        newOutlierRate = outliersInPastMinute /
(60 * *outlierEventsPerSec)
                        if newOutlierRate < 1 {
                                outlierSampleRate = 1.0
                        } else {
                                outlierSampleRate =
newOutlierRate
                        }
                        newOutlierCounter := 0
                        outliersInPastMinute = &newOutlierCounter
                }
        }()
        http.Handle("/", handler)
        [...]
}
```

```
func handler(resp http.ResponseWriter, req *http.Request) {
        var r float64
        if r, err := floatFromHexBytes(req.Header.Get("Sampling-
ID")); err != nil {
                r = rand.Float64()
        }
        start := time.Now()
        i, err := callAnotherService(r)
        resp.Write(i)
        if err != nil || time.Since(start) > 500*time.Millisecond
{
                *outliersInPastMinute++
                if r < 1.0 / outlierSampleRate {
                        RecordEvent(req, outlierSampleRate,
start, err)
                }
        } else {
                *requestsInPastMinute++
                if r < 1.0 / sampleRate {
                        RecordEvent(req, sampleRate, start, err)
                }
        }
}
```

That extremely verbose example uses chunks of duplicate code, but is presented in that manner for clarity. If this example were to support a third category of request, it would make more sense to refactor the code to allow setting sampling rates across an arbitrary number of keys.

Sampling with dynamic rates on arbitrarily many keys

In practice, it is likely that you will not be able to predict a finite set of request quotas that you may want to set. In the last example, our example code had many duplicate blocks and we were designating target rates manually for each case (“error/latency” vs. “normal”).

A more realistic approach is to refactor the code to use a map for each key’s target rate and the number of seen events. The code would then look up each key to make sampling decisions. Doing so modifies our example code like so:

```
var counts map[SampleKey]int
var sampleRates map[SampleKey]float64
var targetRates map[SampleKey]int
func neverSample(k SampleKey) bool {
        // Left to your imagination. Could be a situation where
we know request is a keepalive we never want to record, etc.
        return false
}
// Boilerplate main() and goroutine init to overwrite maps and
roll them over every interval goes here.
type SampleKey struct {
        ErrMsg        string
        BackendShard  int
        LatencyBucket int
}
// This might compute for each k: newRate[k] = counts[k] /
(interval * targetRates[k]), for instance.
func checkSampleRate(resp http.ResponseWriter, start time.Time,
err error, sr map[interface{}]float64, c map[interface{}]int)
float64 {
        msg := ""
        if err != nil {
                msg = err.Error()
        }
        roundedLatency := 100 *(time.Since(start) /
(100*time.Millisecond))
        k := SampleKey {
                ErrMsg:       msg,
                BackendShard: resp.Header().Get("Backend-Shard"),
                LatencyBucket: roundedLatency,
        }
        if neverSample(k) {
                return -1.0
        }
        c[k]++
        if r, ok := sr[k]; ok {
                return r
        } else {
                return 1.0
        }
}
func handler(resp http.ResponseWriter, req *http.Request) {
        var r float64
        if r, err := floatFromHexBytes(req.Header.Get("Sampling-
ID")); err != nil {
                r = rand.Float64()
        }
        start := time.Now()
```

```
        i, err := callAnotherService(r)
        resp.Write(i)
        sampleRate := checkSampleRate(resp, start, err,
sampleRates, counts)
        if sampleRate > 0 && r < 1.0 / sampleRate {
                RecordEvent(req, sampleRate, start, err)
        }
}
```

At this point, our code example is becoming quite large and it still lacks more sophisticated techniques. Our example has been used to illustrate how sampling concepts are implemented.

Luckily, there are existing code libraries to handle this type of complex sampling logic. For Go, the *dynsample-go* library exists and it maintains a map over any number of sampling keys, allocating a fair share of sampling to each key as long as it is novel. That library also contains more advanced techniques of computing sample rates, either based on target rates or without explicit target rates at all.

For this chapter, we're close to having put together a complete introductory tour of how sampling concepts are applied. Before concluding, let's make one last improvement by combining the tail-based sampling you've done so far with head-based sampling that can request tracing be sampled by all downstream services.

Putting it all together: head and tail per-key target rate sampling

Earlier in this chapter, we noted that head-based sampling requires setting a header to propagating a sampling decision downstream. For the code example we've been iterating, that means the parent span must pass both the head sampling decision and its corresponding rate to all child spans. Doing so forces sampling to occur for all child spans, even if the dynamic sampling rate at that level would not have chosen to sample the request.

```
var headCounts, tailCounts map[interface{}]int
var headSampleRates, tailSampleRates map[interface{}]float64
// Boilerplate main() and goroutine init to overwrite maps and
```

```
roll them over every interval goes here. checkSampleRate() etc.
from above as well
func handler(resp http.ResponseWriter, req *http.Request) {
        var r, upstreamSampleRate float64
        if r, err := floatFromHexBytes(req.Header.Get("Sampling-
ID")); err != nil {
                r = rand.Float64()
        }
        // Check if we have an non-negative upstream sample rate;
if so, use it.
        if upstreamSampleRate, err :=
floatFromHexBytes(req.Header.Get("Upstream-Sample-Rate")); err ==
nil && upstreamSampleRate > 1.0 {
                headSampleRate = upstreamSampleRate
        } else {
                headSampleRate := checkHeadSampleRate(req,
headSampleRates, headCounts)
                if headSampleRate > 0 && r < 1.0 / headSampleRate
{
                        // We'll sample this when recording event
below; propagate the decision downstream though.
                } else {
                        // clear out headSampleRate as this event
didn't qualify for sampling.
                        headSampleRate = -1.0
                }
        }
        start := time.Now()
        i, err := callAnotherService(r, headSampleRate)
        resp.Write(i)
        if headSampleRate > 0 {
                RecordEvent(req, headSampleRate, start, err)
        } else {
                // Same as for head sampling, except here we make
a tail sampling decision we can't propagate downstream.
                tailSampleRate := checkTailSampleRate(resp,
start, err, tailSampleRates, tailCounts)
                if tailSampleRate > 0 && r < 1.0 / tailSampleRate
{
                        RecordEvent(req, tailSampleRate, start,
err)
                }
        }
}
```

At this point, our code example is rather complicated. However, even at this level, it illustrates a powerful example of the flexibility that sampling can provide to capture all the necessary context needed to debug your code. In high-throughput modern distributed systems, it may be necessary to get even more granular and employ more sophisticated sampling techniques.

For example, you may want to change the sampleRate of head-based samples for increased probability whenever a downstream tail-based heuristic captures an error in the response. In that example, collector-side buffered sampling is the mechanism that would allow deferring a sampling decision until after an entire trace has been buffered; bringing together the advantages of head-based sampling to properties only known at the tail.

Conclusion

Sampling is a useful technique for refining your observability data. While sampling is necessary when running at scale, it can be useful in a variety of circumstances even at smaller scales. The code-based examples illustrate how different sampling strategies are implemented. Increasingly, it's common for open source instrumentation libraries — such as OpenTelemetry — to implement that type of sampling logic for you. As those libraries become the standard for generating application telemetry data, it should become less likely that you would need to re-implement these sampling strategies in your own code.

However, even if you rely on third-party libraries to manage that strategy for you, it is essential that you understand the underpinnings of how sampling is implemented. Understanding how those strategies like static vs. dynamic, head vs. tail, or some combination thereof, work in practice enables you to use them in a wise manner to achieve data fidelity while also optimizing for resource constraints.

Similar to deciding what and how to instrument your code, deciding *what*, *when*, and *how* to sample is best defined by your unique organizational needs. The fields in your events that influence how interesting they are to

sample largely depend on how useful they are to understanding the state of your environment and their impact on achieving your business goals.

In the next chapter, we'll examine the impact that deciding to build your own vs. buying an existing observability solution, and the ROI you can expect, has on your business goals.

Chapter 14. The Business Case for Observability

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 20th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

In the last section of this book, we examined some of the challenges an organization must tackle in order to have an effective observability practice at scale. This section examines the social practices teams can use to spread a culture of observability within their organizations, regardless of scale. Observability often starts within one particular team or business unit in an organization. In order to spread a culture of observability, teams will need support from various stakeholders across the business. This section breaks down how that support comes together to help proliferate an observability practice.

In this chapter, we'll start by laying out the business case for observability. Some organizations adopt observability practices in response to overcoming dire challenges that cannot be addressed by traditional approaches. Others may need a more proactive approach to changing traditional practices. Regardless of where in your observability journey you may be, this chapter

aims to help you understand how to make a business case for observability within your own company.

We start by looking at both the reactive and proactive approaches to instituting change. We'll examine non-emergency situations to identify a set of circumstances that can point to a critical need to adopt observability outside the context of catastrophic service outages. Then we'll cover the steps needed to support creation of an observability practice, how to evaluate different tools, and how to know when your organization has achieved a state of observability that is "good enough" to shift your focus to other initiatives.

The reactive approach to introducing change

Change is hard. Many organizations tend to follow the path of least resistance. Why fix the things that aren't (perceived to be, anyway) broken? Historically, production systems have operated just fine for decades without observability. Why rock the boat now? Simpler systems could be reasoned about by engineers intimately familiar with the finer points of their architectures. As seen in Chapter 3 of this book, it isn't until traditional approaches suddenly and drastically fall short that some organizations realize their now-critical need for observability.

Introducing fundamental change into an organization in reactionary knee-jerk ways can have unintended consequences. The rush to fix mission-critical business problems often leads to oversimplified approaches that rarely lead to useful outcomes. Consider the case of reactionary change introduced as the result of critical service outages.

For example, an organization might perform a root-cause analysis to determine why an outage occurred, and the analysis might point to a singular reason. In mission-critical situations, it's often tempting for executives to leverage that reason as a driver for simplified remediations that demonstrate the problem has been swiftly dealt with. When the smoking gun for an outage can be pointed to as the line in the root cause analysis that says "we didn't have backups," that can be used to justify

demoting the employee who deleted the important file, engaging consultants to introduce a new backup strategy, and the business breathing a sigh of relief once they believe the appropriate gap has been closed.

While that approach might seem to offer a sense of security, it's ultimately false. Why was that one file able to create a cascading system failure? Why was a file that critical so easily deletable? Could the situation have been better mitigated with more immutable infrastructure? Any number of approaches in this hypothetical scenario might better treat the underlying causes rather than the most obvious symptoms. In a rush to fix problems quickly, often the oversimplified approach is the most tempting to take.

Unfortunately, waiting for a catastrophic failure to occur and then remediating the situation with an oversimplified fix is how some organizations go about making the business case for change. Taking a more holistic approach requires building up a business case for change proactively by examining the state of your systems and their impact on business goals.

The proactive approach to introducing change

A lack of observability in your systems can silently gum up the ability of your software engineering and operations teams to deliver innovative work. In non-observable systems, incidents with identical symptoms (and underlying causes) are often repeated. Without observability, teams are often left operating in the dark when attempting to find effective solutions that mitigate the source(s) of a problem.

In non-observable systems, repeated outages will occur due to a slew of unexplained system behaviors and near-miss scenarios. With no clear understanding of the underlying cause, and no foreseeable effective solution, the human toll from these repeated incidents rises. Customers lose trust in your services and abandon their transactions. Your engineering teams start experiencing alert fatigue that leads to burnout and eventually

churn: costing the business both lost expertise amongst their staff and the time it takes to rebuild it. That lack of understanding undermines your team's confidence when making changes to production. It creates more fragile systems, which in turn require more time to maintain and slows the delivery of new features that provide business value.

Unfortunately, business leaders unaccustomed to having observable systems often accept these hurdles as the normal state of operations. They introduce processes that they believe help mitigate these problems, such as Change Advisory Boards or rules prohibiting their team from deploying code changes on a Friday. They expect on-call rotations to burn out engineers from time to time, so they allow on-call exemptions for their star engineers. Many toxic cultural practices in engineering teams can be traced back to situations that start with a fundamental lack of understanding of their production systems.

Signs that your business may be hitting a breaking point without observability in its systems include—but are not limited to—some of the following scenarios:

- Customers discover and report critical bugs in production services long before they are detected and addressed internally.
- When minor incidents occur, it often takes so long to detect and recover them that they escalate into prolonged service outages.
- The backlog of investigation necessary to troubleshoot incidents and bugs continues to grow because new problems pile up faster than they can be retrospected or triaged.
- The amount of time spent on break/fix operational work exceeds the amount of time your teams spend on delivering new features.
- Customer satisfaction with your services is low due to repeated poor performance that your support teams cannot verify, replicate, or resolve.

- New features are delayed by weeks or months due to engineering teams dealing with disproportionately large amounts of unexpected work necessary to figure out how various services are all interacting with one another.

There may be other contributing factors to the above scenarios that may require additional mitigation approaches. However, teams experiencing a multitude of these symptoms more than likely need to address a systemic lack of observability in their systems. Teams operating in these ways display a fundamental lack of understanding how their production systems behave such that it negatively impacts their ability to deliver against business goals.

A proactive approach to introducing change is to recognize these symptoms as abnormal and preventable. The business case for introducing observability into your systems is to reduce both the time to detect (TTD) and time to resolve (TTR) issues within your services. Observability provides your teams a granular ability to investigate performance issues as experienced by individual users of your services. More than half of mobile users will abandon transactions after 3 seconds of load time.[1] The business case for observability is customer retention.

The upstream impact of detecting and resolving issues faster is that it reduces the amount of unexpected break/fix operational work for your teams. They can reduce the backlog of triaging issues and resolving bugs and increase the amount of time they spend delivering new features. By detecting and resolving service issues quickly, your teams can fix issues before they're detected by your customers leading to more reliable services with greater customer satisfaction.

If the outcomes above matter to your business, then you have a business case for introducing observability into your organization. Rather than waiting for a series of catastrophic failures that prompt your business to address the symptoms of non-observable systems, the proactive approach is to start introducing observability into your sociotechnical systems with small achievable steps that have big impacts.

Introducing observability as a practice

Similar to introducing security or testability into your applications, observability is an ongoing practice that is a responsibility shared by anyone responsible for developing and running a production service. Building effective observable systems is not a one-time effort. You cannot simply take a checkbox approach to introducing technical capabilities and declare that your organization has "achieved" observability any more than you can do that with security or testability. Observability must be introduced as a practice.

Observability begins as a capability that can be measured as a technical attribute of a system: can your system be observed or not? As highlighted several times throughout this book, production systems are sociotechnical. Once a system has observability as a technical attribute, the next step is measured by how well your teams and the system operate together. Just because a system can be observed, it does not mean that it is being observed effectively.

The goal of observability is to provide engineering teams the capability to develop, operate, thoroughly debug, and report on their systems. Teams must be empowered to explore their curiosity by asking arbitrary questions about their system to better understand its behavior. They must be incentivized to interrogate their systems proactively, both by their tools and with management support. A sophisticated analytics platform is useless if the team using it feels overwhelmed by the interface or is discouraged from querying for fear of running up a large bill.

A well-functioning observability practice not only empowers engineers to ask questions that help detect and resolve issues in production, it should also encourage them to begin answering business intelligence questions in real-time. If nobody is using the new feature that the engineering team has built, or if one customer is at risk of churning because they are persistently experiencing issues, that is a risk to the health of your business. Practicing observability should encourage engineers to adopt a cross-functional

approach to measuring service health beyond its performance and availability.

As the DevOps movement continues to gain mainstream traction, forward-thinking engineering leadership teams remove barriers between engineering and operations teams. Removing these artificial barriers empowers teams to take more ownership of the development and operation of their software. Observability helps engineers lacking on-call experience better understand where failures are occurring and how to mitigate them, eroding the artificial wall between software development and operations. Similarly, observability erodes the artificial wall between software development, operations, and business outcomes. Observability gives software engineering teams the appropriate tools to debug and understand how their systems are being used. It helps them shed their reliance on functional handoffs, excessive manual work, playbooks, guesswork, and external views of system health measures that impact business goals.

It is beyond the scope of this chapter to outline all of the practices and traits commonly shared by high-performing engineering teams. The DevOps Research and Assessment (DORA) State of DevOps Report describes many of the essential traits that separate elite teams from their low-performing counterparts.[2] Similarly, teams introducing observability benefit from many of the practices described in the report.

When introducing an observability practice, engineering leaders should first ensure that they are creating a culture of psychological safety. Blameless[3] culture fosters a psychologically safe environment that supports experimentation and rewards curious collaboration. Encouraging experimentation is necessary to evolve traditional practices. DORA's year-over-year reporting demonstrates both the benefits of blameless culture and its inextricable link with high-performing teams.

With a blameless culture in practice, business leaders should also ensure that a clear scope of work exists when introducing observability; such as happening entirely within one introductory team or line of business. Baseline performance measures for TTD and TTR should be used as a

benchmark to measure improvement within that scope. The infrastructure and platform work required should be identified, allocated, and budgeted in support of this effort. Only then should the technical work of instrumentation and analysis of that team's software begin.

Using the appropriate tools

Although observability is primarily a cultural practice, it does require engineering teams to possess the technical capability to instrument their code, to store the emitted telemetry data, and to analyze that data in response to their questions. A large portion of the initial technical effort to introduce observability will require setting up tooling and instrumentation.

At this point, some teams attempt to roll their own observability solutions. As seen in Chapter 20, the ROI on building a bespoke observability platform that does not align with your company's core competencies is almost never worthwhile. Most organizations will find that approach to be prohibitively difficult, time consuming, and expensive. Instead, there are a wide range of solutions with various tradeoffs to consider, such as commercial vs. open-source or on-premises vs. hosted.

Instrumentation

The first step to consider is how your applications will emit telemetry data. Traditionally, this consideration was intertwined with the needs of your backend datastore: teams were bound to using instrumentation frameworks that locked them into using the specific vendor of choice.

For instrumentation of both frameworks and application code, OpenTelemetry is the emerging standard. It supports every open source metric and trace analytics platform, and it is supported by almost every commercial vendor in the space. There is no longer a reason to lock into one specific vendor's instrumentation framework, nor roll your own.

With OpenTelemetry's pluggable exporters, you can configure your instrumentation to send data to the analytics tool of your choice. By using a

common standard, it's possible to easily demo the capabilities of any analytics tool by simply sending your instrumentation data to multiple backends at the same time.

When considering the data that your team must analyze, it's an oversimplification to simply break observability down into categories like metrics, logging, and tracing. While those can be valid categories of observability data, what's important to achieve observability is how those data types interact to give your teams an appropriate view of their systems. While messaging that describes observability as three pillars is useful as a marketing headline, it misses the big picture. At this point, it is more useful to instead think about what data type or types are best suited to your use case, and which can be generated on-demand from the others.

Data storage and analytics

Once you have telemetry data, you need to consider how it's stored and analyzed. Data storage and analytics are often bundled into the same solution, but that depends on whether you decide to use open source or proprietary solutions.

Commercial vendors typically bundle storage and analytics. Each vendor has differentiating features for storage and analytics and you should consider which of those best help your teams reach their observability goals. Vendors of proprietary all-in-one solutions include Honeycomb, Lightstep, New Relic, Splunk, Datadog, and others.

Open source solutions typically require separate approaches to data storage and analytics. These open source frontends include solutions like Grafana, Prometheus, or Jaeger. While they handle analytics, they all require a separate datastore in order to scale. Popular open source data storage layers include Cassandra, Elastic, M3, or InfluxDB.

It's great to have so many options. But you must also carefully consider and be wary of the operational load incurred by running your own data storage cluster. For example, the ELK stack (Elasticsearch, Logstash, and Kibana) is popular because it fulfills needs in the log management and analytics

space. But end-users frequently report that their maintenance and care of their ELK cluster gobbles up systems engineering time and grows quickly in associated management and infrastructure costs. As a result, you'll find that there is a competitive market for managed open source telemetry data storage (e.g. ELK-as-a-Service).

When considering data storage, we would also caution against finding separate solutions for each category (or "pillar") of observability data you need. Similarly, attempting to bolt-on modern observability functionality to a traditional monitoring system is likely to be fraught with peril. Since observability arises from how your engineers interact with your data to answer questions, it is better to have one cohesive solution that works seamlessly rather than maintaining three or four separate systems. Using disjointed systems for analysis places the burden of carrying context and translation between those systems on engineers and creates a poor usability and troubleshooting experience.

Rolling out tools to your teams

When considering tooling options, it's important to ensure that you are investing precious engineering cycles on differentiators core to your business needs. Consider whether your choice of tools is providing more innovation capacity or draining that capacity into managing bespoke solutions. Does your choice of tooling require creating a larger and separate team for management? The goal of observability isn't to create bespoke work within your engineering organization, it's to save your business time and money while increasing quality.

That's not to say that certain organizations should not create observability teams. Like security and testability, observability is a practice that should be in use by every engineer. However, especially in larger organizations, a good observability team will focus on helping each product team achieve observability in their platform or partner with them through the initial integration process. After evaluating which platform best fits the needs of your pilot team, an observability team can help make the same solutions more accessible to your engineering teams as a whole.

Knowing when you have enough observability

Like security and testability, there's always more work to be done with observability. Business leaders may struggle with knowing when to make investing in observability a priority and when observability is "good enough" that other concerns can take precedence. Knowing when you have enough observability is a useful checkpoint for determining the success of a pilot project.

If the symptom of teams flying blind without observability is excessive rework, then teams with sufficient observability should have predictable delivery and sufficient reliability. Let's examine how to recognize that milestone both in terms of cultural practices and key results.

Once observability practices have become a foundational practice within a team, little outside intervention should be required to maintain a system with excellent observability. Just as a team wouldn't think to check in new code without associated tests, so too should teams practicing observability think about associated instrumentation as part of any code review process. Instead of merging code and shutting down their laptops at the end of the day, it should be second nature for teams practicing observability to see how their code behaves as it reaches each stage of deployment.

Instead of code behavior in production being "someone else's problem", teams with enough observability should be excited to see how real users benefit from the features they are delivering. Every code review should consider if the telemetry bundled with the change is appropriate to understand the impact this change will have in production. Observability should also not just be limited to engineers: bundled telemetry should empower product managers and customer success representatives to answer their own questions about production. Two useful measures to determine if enough observability is present is seeing a marked improvement in self-serve fulfilment for one-off data requests about production behavior and a reduction in product management guesswork.

As teams reap the benefits of observability, their confidence level for understanding and operating in production should rise. The proportion of unresolved “mystery” incidents should decrease, and time to detect and resolve incidents will decrease across the organization. However, it is worth calling out that a frequent mistake for measuring success at this point is over-indexing on shallow metrics such as the overall number of incidents detected. Finding more incidents and comfortably digging into near-misses is a positive step as your teams gain an increased understanding of how production behaves. That often means previously undetected problems are now being more fully understood. You’ll know you’ve reached an equilibrium when your engineering teams live within their modern observability tooling to understand problems and disjointed legacy tooling is no longer a primary troubleshooting method.

Whenever your teams encounter a new problem that poses questions your data cannot answer, they will find it easier to take the time to fill in that telemetry gap rather than attempting to guess at what might be wrong. For example, if there is a mystery trace span that is taking too long for inexplicable reasons, they will add subspans to capture smaller units of work within it, or add attributes to understand what is triggering the slow behavior. Observability always requires some care and feeding as integrations are added or the surface area of your code changes. But even so, the right choice of observability platform will still drastically reduce your overall operational burdens and total cost of ownership.

Conclusion

The need for observability is recognized within teams for a variety of reasons. Whether that need arises reactively in response to a critical outage, or proactively by realizing how its absence is stifling innovation on your teams, it’s critical to create a business case in support of your observability initiative. Similar to security and testability, observability must be approached as an ongoing practice. Teams practicing observability must make a habit of ensuring any changes to code are bundled with proper

instrumentation, just as they're bundled with tests. Code reviews should ensure the instrumentation for new code achieves proper observability standards, just as they ensure it also meets security standards. Observability requires ongoing care and maintenance, but you'll know that observability has been achieved well enough by looking for the cultural behaviors and key results outlined in this chapter.

In the next chapter, we'll look at how engineering teams can create alliances with other internal teams to help accelerate the adoption of observabilty culture.

---

1 https://developer.akamai.com/blog/2016/09/14/mobile-load-time-user-abandonment

2 https://services.google.com/fh/files/misc/state-of-devops-2019.pdf

3 https://postmortems.pagerduty.com/

Chapter 15. An Observability Maturity Model

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 22nd chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Spreading a culture of observability is best achieved by having a plan that measures progress and prioritizes target areas of investment. In this chapter, we go beyond the benefits of observability and its tangible technical steps by using the maturity model approach as a way to benchmark and measure progress. You will learn about the key capabilities an organization can measure and prioritize as a way of driving observability adoption.

A foreword about maturity models

In the early 1990s, the Software Engineering Institute at Carnegie Mellon University popularized the Capability Maturity Model as a way to evaluate the ability of various vendors to deliver effectively against software development projects. The model defines progressive stages of maturity and a classification system used to assign scores based on how well a particular

vendor's process matches each stage. Those scores are then used to influence purchasing decisions, engagement models, and other activities.

Since then, maturity models have become somewhat of a darling for the software marketing industry. Going well beyond scoping purchasing decisions, maturity models are now used as a generic way to model organizational practices. To their credit, maturity models can be helpful for an organization to profile its capabilities against its peers, or to target a set of desired practices. However, maturity models are not without their limitations.

When it comes to measuring organizational practices, there is no upper bound for the performance level of a software engineering team. Practices, as opposed to procedures, are an evolving and continuously improving state of the art. They're never done and perfect, as reaching the highest level of a maturity model seems to imply. Further, that end state is a static snapshot of an ideal future reflecting only what was known when the model was created, often with the biases of its authors baked into its many assumptions. Objectives shift, priorities change, better approaches are discovered, and — more to the point — each approach is unique to individual organizations and cannot be universally scored.

When looking at a maturity model, it's important to always remember that there is no one-size-fits-all model applicable to every organization. Maturity models can, however, be useful as starting points against which you can critically and methodically weigh your own needs and desired outcomes to create an approach that's right for you. Maturity models can help you identify and quantify tangible and measurable objectives that are useful in driving long-term initiatives. It is critical that you create hypotheses to test the assumptions of any maturity model within your own organization and evaluate which paths are and aren't viable given your particular constraints. Those hypotheses, and the maturity model itself, should be continuously improved over time as more data becomes available.

Why observability needs a maturity model

When it comes to developing and operating software, practices that drive engineering teams toward a highly-productive state have rarely been formalized and documented. Instead, they are often passed along informally from senior engineers to junior engineers based on the peculiar history of individual company cultures. That tribal knowledge sharing has given rise to certain identifiable strains of engineering philosophy or collections of habits with monikers that tag their origin (e.g. “the Etsy way”, “the Googley way”, etc). Those philosophies are familiar to others with the same heritage and have become heretical to those without.

Collectively, the authors of this book have over six decades of experience watching engineering teams fail and succeed, and fail all over again. We’ve seen organizations of all shapes, sizes, and locations succeed in forming high-performing teams given they adopt the right culture and focus on essential techniques. Software engineering success is not limited to those fortunate enough to live in Silicon Valley or those who have worked at FAANG companies. It shouldn’t matter where teams are located, or what companies their employees have worked at previously.

As mentioned throughout this book, production software systems present sociotechnical challenges. While tooling for observability can address the technical challenges in software systems, we need to consider other factors beyond the technical. The observability maturity model considers the context of engineering organizations, their constraints, and their goals. Technical and social features of observability contribute to each of the identified team characteristics and capabilities in the model.

To make a maturity model broadly applicable, it must be agnostic to an organization’s pedigree or the tools they use. Instead of mentioning any specific software solutions or technical implementation details, it must instead focus on the costliness and benefits of the journey in human terms. “How will you know if you are weak in this area?” “How will you know if you’re doing well in this area and should prioritize improvements elsewhere?”

Based on our experiences watching teams adopt observability, we'd seen common qualitative trends like their increased confidence interacting with production and ability to spend more time working on new product features. To quantify those perceptions, we surveyed a spectrum of teams in various phases of observability adoption and teams who hadn't yet started or were not adopting observability. What we found was that teams who adopt observability were three times as likely to feel confident in their ability to ensure software quality in production, compared to teams that had not adopted observability.[1] Additionally, those teams that hadn't adopted observability spent over half their time toiling away on work that did not result in releasing new product features.

Those patterns are emergent properties of today's modern and complex sociotechnical systems. Analyzing capabilities like ensuring software quality in production and time spent innovating on features exposes both the pathologies of group behavior and their solutions. Adopting a practice of observability can help teams find solutions to problems that can't be solved by individuals "just writing better code," or, "just doing a better job." Creating a maturity model for observability ties together the various capabilities outlined throughout this book and can serve as a starting point for teams to model their own outcome-oriented goals that guide adoption.

About the Observability Maturity Model

The Observability Maturity Model (OMM) was developed by Charity Majors and Liz Fong-Jones based on the following goals for an engineering organization:

Sustainable systems and quality of life for engineers

> This goal may seem aspirational to some, but the reality is that engineer quality of life and the sustainability of systems are closely entwined. Systems that are observable are easier to own and maintain, improving the quality of life for an engineer who owns said systems. Engineers spending more than half of their time working on things that don't deliver customer value, i.e. toil, report higher rates of burnout, apathy,

and lower engineering team morale. Observable systems reduce toil and that, in turn, creates higher rates of employee retention and reduces the time and money teams need to spend on finding and training new engineers.

Delivering against business needs by increasing customer satisfaction

Observability enables engineering teams to better understand how customers interact with the services they develop. That understanding allows engineers to hone in on customer needs and deliver the performance, stability, and functionality that will delight their customers. Ultimately, observability is about operating your business successfully.

The framework described here is a starting point. With it, organizations have the structure and tools to begin asking themselves questions, and the context to interpret and describe their own situation—both where they are now, and what they should aim for.

The quality of your observability practice depends upon both technical and social factors. Observability is not a property of the computer system alone or the people alone. Too often, discussions of observability are focused only on the technicalities of instrumentation, storage, and querying, and not upon what they can enable a team to do. The OMM approaches improving software delivery and operations as a sociotechnical problem.

If teams feel unsafe applying their tooling to solve problems, they won't be able to achieve results. Tooling quality depends upon factors such as the ease of adding instrumentation, the granularity of data ingested, and whether it can answer any arbitrary questions that humans pose. The same tooling need not be used to address each capability, nor does strength of tooling in addressing one capability necessarily translate to an aptitude in addressing all other suggested capabilities.

Capabilities referenced in the OMM

The following list of capabilities are directly impacted by the quality of your observability practice. The list is non-exhaustive, but is intended to represent the breadth of potential business needs. The capabilities listed and their associated business outcomes overlap with many of the principles necessary to create production excellence[2].

There is no singular and correct order or prescriptive way of doing these things. Instead, every organization faces an array of potential journeys. At each step, focus on what you're hoping to achieve. Make sure you will get appropriate business impact from making progress in that area right now, as opposed to doing it later.

It's also important to understand that building up these capabilities is a pursuit that is never "done." There's always room for continuous improvement. Pragmatically speaking, however, once organizational muscle memory exists such that these capabilities are second nature, and they are systematically supported as part of your culture, that's a good indication of having reached the upper levels of maturity. For example, prior to Continuous Integration systems, code was often checked in without much thought paid to bundling in tests. Now, engineers in any modern organization practicing CI/CD would never think of checking in code without bundled tests. Similarly, observability practices must become second nature for development teams.

Respond to system failure with resilience

Resilience is the adaptive capacity of a team, together with the system it supports, that enables it to restore service and minimize impact to users. Resilience refers not only to the capabilities of an isolated operations team, or to the robustness and fault tolerance in its software.[3] Resilience must also measure both the technical outcomes and social outcomes of your emergency response process in order to measure its maturity.

Measuring technical outcomes can, at first approximation, take the form of examining how long it takes to restore service and how many people become involved when the system experiences a failure. For example, the

2018 Accelerate State of DevOps Report defines elite performers as those whose mean time to recover (MTTR) is less than 1 hour and low performers as those with a MTTR that is between 1 week and 1 month.[4]

Emergency response is a necessary part of running a scalable, reliable service. But emergency response may have different meanings to different teams. One team might consider a satisfactory emergency response to mean "power cycle the box", while another might interpret that to mean "understand exactly how the auto-remediation that restores redundancy in data striped across multiple disks broke, and mitigate future risk." There are three distinct areas to measure: how long it takes to detect issues, how long it takes to initially mitigate those issues, and how long it takes to fully understand what happened and address those risks.

But the more important dimension that team managers need to focus on is the people operating that service. Is the on-call rotation sustainable for your team so that staff remain attentive, engaged, and retained? Is there a systematic plan to educate and involve everyone responsible for production in an orderly and safe manner, or is every reponse an all-hands-on-deck emergency, no matter the experience level? If your service requires many different people to be on-call or context switching to handle break-fix scenarios, that's time and energy they are not spent generating business value by delivering new features. Over time, team morale will inevitably suffer if a majority of engineering time is devoted toward toil, including break-fix work.

If your team is doing well

- System uptime meets your business goals, and is improving.
- On-call response to alerts is efficient, alerts are not ignored.
- On-call is not excessively stressful, and engineers are not hesitant to take additional shifts as needed.
- Engineers can handle incident workload without working extra hours or feeling unduly stressed.

If your team is doing poorly

- The organization is spending a lot of additional time and money staffing on-call rotations.
- Incidents are frequent and prolonged.
- Those on call suffer from alert fatigue, or they aren't alerted about real failures.
- Incident responders cannot easily diagnose issues.
- Some team members are disproportionately pulled into emergencies.

How observability is related

Alerts are relevant, focused, and actionable (thereby reducing alert fatigue). There is a clear relationship between the error budget and customer needs. When incident investigators respond, context-rich events make it possible to effectively troubleshoot incidents when they occur. The ability to drill into high-cardinality data and quickly aggregate results on the fly supports pinpointing error sources and faster incident resolution. Preparing incident responders with the tools they need to effectively debug complex systems reduces the stress and drudgery of being on-call. Democratization of troubleshooting techniques by easily sharing past investigation paths helps distribute incident resolution skills across a team, so that anyone can effectively respond to incidents as they occur.

Deliver high quality code

High quality code is measured by more than how well it is understood, maintained, or how often bugs are discovered in a sterile lab environment (e.g. a CI test suite). While code readability and traditional validation techniques are useful, they do nothing to validate how that code actually behaves during the chaotic conditions inherent to running in production systems. It must be possible to adapt to changing business needs, rather than brittle and fixed in features. Thus, code quality must be measured by

validating its operation and extensibility where it matters to your customers and your business.

If your team is doing well

- Code is stable, there are fewer bugs discovered in production and fewer outages.
- After code is deployed to production, your team focuses on customer solutions rather than support.
- Engineers find it intuitive to debug problems at any stage, from writing code in development to troubleshooting incidents in production at full release scale.
- Isolated issues that occur can typically be fixed without triggering cascading failures.

If your team is doing poorly

- Customer support costs are high.
- A high percentage of engineering time is spent fixing bugs vs working on new features.
- Team members are often reluctant to deploy new features because of perceived risk.
- It takes a long time to identify an issue, construct a way to reproduce the failure case, and repair it.
- Developers have low confidence in their code's reliability after it has shipped.

How observability is related

Well-monitored and tracked code makes it easy to see when and how a process is failing, and easy to identify and fix vulnerable spots. High quality observability allows using the same tooling to debug code on one machine as on 10,000. A high level of relevant, context-rich telemetry means

engineers can watch code in action during deploys, be alerted rapidly, and repair issues before they become user-visible. When bugs do appear, it is easy to validate that they have been fixed.

Manage complexity and technical debt

Technical debt is not necessarily a bad thing. Engineering organizations are constantly faced with choices between short-term gain and longer-term outcomes. Sometimes the short-term win is the right decision if there is also a specific plan to address the debt, or to otherwise mitigate the negative aspects of the choice. With that in mind, code with high technical debt is code in which quick solutions have been chosen over more architecturally stable options. When unmanaged, these choices lead to longer-term costs, as maintenance becomes expensive and future revisions become dependent on costs.

If your team is doing well

- Engineers spend the majority of their time making forward progress on core business goals.
- Bug fixing and other reactive work takes up a minority of the team's time.
- Engineers spend very little time disoriented or trying to find where in the codebase they need to plumb through changes.

If your team is doing poorly

- Engineering time is wasted rebuilding things when their scaling limits are reached or edge cases are hit.
- Teams are distracted by fixing the wrong thing or picking the wrong way to fix something.
- Engineers frequently experience uncontrollable ripple effects from a localized change.

- People are afraid to make changes to the code, aka the "haunted graveyard"[5] effect.

How observability is related

Observability enables teams to understand the end-to-end performance of their systems and debug failures and slownesses without wasting time. Troubleshooters can find the right breadcrumbs when exploring an unknown part of their system. Tracing behavior becomes easily possible. Engineers can identify the right part of the system to optimize rather than taking random guesses of where to look and change code when attempting to find performance bottlenecks.

Release on a predictable cadence

The value of software development only reaches users once new features and optimizations are released. The process begins when a developer commits a change set to the repository, includes testing and validation and delivery, and ends when the release is deemed sufficiently stable and mature to move on. Many people think of continuous integration and deployment as the nirvana end-stage of releasing. But CI/CD tools and processes are just the basic building blocks needed to develop a robust release cycle. Every business needs a predictable, stable, frequent release cadence to meet customer demands and remain competitive in their market.[6]

If your team is doing well

- The release cadence matches business needs and customer expectations.
- Code gets into production shortly after being written. Engineers can trigger deployment of their own code once it's been peer reviewed, satisfies controls, and is checked in.
- Code paths can be enabled or disabled instantly, without needing a deploy.

- Deploys and rollbacks are fast.

If your team is doing poorly

- Releases are infrequent and require lots of human intervention.
- Lots of changes are shipped at once.
- Releases have to happen in a particular order.
- Sales has to gate promises on a particular release train.
- Teams avoid doing deploys on certain days or times of year. They are hesitant because poorly managed release cycles have frequently interfered with quality of life during non-business hours.

How observability is related

Observability is how you understand the build pipeline as well as production. It shows you any performance degradation in tests or errors during the build and release process. Instrumentation is how you know if the build is good or not, if the feature you added is doing what you expected it to, if anything else looks weird, and lets you gather the context you need to reproduce any error.

Observability and instrumentation are also how you gain confidence in your release. If properly instrumented, you should be able to break down by old and new build ID and examine them side by side. You can validate consistent and smooth production performance between deployments or you can see if your new code is having its intended impact and if anything else looks suspicious. You can also drill down into specific events, for example to see what dimensions or values a spike of errors all have in common.

Understand user behavior

Product managers, product engineers, and systems engineers all need to understand the impact that their software has upon users. It's how we reach

product-market fit as well as how we feel purpose and impact as engineers. When users have a bad experience with a product, it's important to understand both what they were trying to do and what the outcome was.

If your team is doing well

- Instrumentation is easy to add and augment.
- Developers have easy access to KPIs for customer outcomes and system utilization/cost, and can visualize them side by side.
- Feature flagging or similar makes it possible to iterate rapidly with a small subset of users before fully launching.
- Product managers can get a useful view of customer feedback and behavior.
- Product-market fit is easier to achieve.

If your team is doing poorly

- Product managers don't have enough data to make good decisions about what to build next.
- Developers feel that their work doesn't have impact.
- Product features grow to excessive scope, are designed by committee, or don't receive customer feedback until late in the cycle.
- Product-market fit is not achieved.

How observability is related

Effective product management requires access to relevant data. Observability is about generating the necessary data, encouraging teams to ask open-ended questions, and enabling them to iterate. With the level of visibility offered by event-driven data analysis and the predictable cadence of releases both enabled by observability, product managers can investigate

and iterate on feature direction with a true understanding of how well their changes are meeting business goals.

Using the OMM for your organization

The Observability Maturity Model can be a useful tool for reviewing your organization's capabilities when it comes to utilizing observability effectively. It provides a starting point for measuring where your team capabilities are lacking and where they excel. When creating a plan for your organization to adopt and spread a culture of observability, it is useful to prioritize capabilities that most directly impact the bottom line for your business and improve your performance.

It's important to remember that creating a mature observability practice is not a linear progression and that these capabilities do not exist in a vacuum. Observability is entwined in each capability and improvements in one capability can sometimes contribute to results in others. How that process unfolds is unique to the needs of each organization and where to start depends on your current areas of expertise.

Wardley mapping is a technique that can help you figure out how these capabilities relate in priority and interdependency with respect to your current organizational abilities. Understanding which capabilities are most critical to your business can help prioritize and unblock the steps necessary to further your observability adoption journey.

As you review and prioritize each capability, you should identify clear owners responsible for driving this change within your teams. Review those initiatives with those owners and ensure you develop clear outcome-oriented measures that are relevant to the needs of your particular organization. It's difficult to make progress unless you have clear ownership, accountability, and sponsorship both in terms of financing and time. Without executive sponsorship, a team may be able to make some small incremental improvements within their own silo. However, it is impossible to reach a mature high-performing state if the entire organization

is unable to demonstrate these capabilities and instead it relies on a few key individuals, no matter how advanced and talented those individuals may be.

Conclusion

The Observability Maturity Model provides a starting point against which your organization can measure its desired outcomes and create its own customized adoption path. The key capabilities that drive high performing teams who have matured their observability practice are measured along these axes:

- How they respond to system failure with resilience
- How easily they can deliver high quality code
- How well they manage complexity and technical debt
- How predictable their software release cadence is
- How well they can understand user behavior

The Observability Maturity Model is a synthesis of qualitative trends we'd noticed across organizations adopting observability paired with quantitative analysis from surveying software engineering professionals. The conclusions represented in this chapter reflect research studies conducted in 2020[7] and 2021.[8] It's important to remember that maturity models are static snapshots of an ideal future generalized well enough to be applicable across an entire industry. The maturity model itself will evolve as observability adoption continues to spread.

Similarly, observability practices will also evolve and the path toward maturity will be unique to your particular organization. However, this chapter provides the basis for your organization to create its own pragmatic approach. In the next chapter, we'll conclude with a few additional tips on where to go from here.

1 The Honeycomb Observability Maturity Report, 2020 edition

2 https://www.infoq.com/articles/production-excellence-sustainable-operations-complex-systems/

3 https://www.infoq.com/news/2019/04/allspaw-resilience-engineering/

4 https://cloudplatformonline.com/2018-state-of-devops.html

5 https://www.usenix.org/system/files/login/articles/login_fall16_08_beyer.pdf

6 https://www.intercom.com/blog/shipping-is-your-companys-heartbeat/

7 https://www.honeycomb.io/wp-content/uploads/2020/04/observability-maturity-report-4-2-2020-1-1-1.pdf

8 https://www.honeycomb.io/wp-content/uploads/2021/06/Observability_Maturity_Report.pdf

About the Authors

Charity Majors is a cofounder and engineer at Honeycomb.io, a startup that blends the speed of time series with the raw power of rich events to give you interactive, iterative debugging for complex systems. She has worked at companies like Facebook, Parse, and Linden Lab, as a systems engineer and engineering manager, but always seems to end up responsible for the databases too.

Liz Fong-Jones is a developer advocate, labor and ethics organizer, and Site Reliability Engineer (SRE) with 15+ years of experience. She is an advocate at Honeycomb.io for the SRE and Observability communities, and previously was an SRE working on products ranging from the Google Cloud Load Balancer to Google Flights.

George Miranda is a former engineer turned product marketer at Honeycomb.io. He spent 15+ years building large scale distributed systems in the finance and video game industries. He discovered his knack for storytelling and now works to shape the tools, practices, and culture that help improve the lives of people responsible for managing production systems.