O'REILLY®

Compliments of
Humio

Distributed Systems Observability

A Guide to Building Robust Systems

Cindy Sridharan

HUMIO

Limitless Logging for Developers and Operations Teams

Log Everything, Answer Anything:
Unlock the power of having all of your data at your fingertips instantly.

- On-Premises or Cloud
- 1 TB/day ingest on a single node
- Search without indexing
- Open API
- 10x compression at ingest

Try Humio for Free

"With Humio, anyone who's familiar with a Unix pipe can easily make the most of its querying abilities. So, Humio not only manages all our log data but also allows us to do real-time, complex queries on our log stream that wouldn't have been possible before."
David Højelsen, co-founder and CTO, Mono Solutions

humio.com @meethumio

THE LINUX FOUNDATION SILVER MEMBER

Distributed Systems Observability

A Guide to Building Robust Systems

Cindy Sridharan

Beijing · Boston · Farnham · Sebastopol · Tokyo
O'REILLY®

Distributed Systems Observability
by Cindy Sridharan

Copyright © 2018 O'Reilly Media. 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/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nikki McDonald
Development Editor: Virginia Wilson
Production Editor: Justin Billing
Copyeditor: Amanda Kersey
Proofreader: Sharon Wilkey

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest
Tech Reviewers: Jamie Wilkinson and Cory Watson

May 2018: First Edition

Revision History for the First Edition
2018-05-11: First Release

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

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author 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 Humio. See our *statement of editorial independence*.

978-1-492-03340-0

[LSI]

Table of Contents

1. **The Need for Observability. 1**
    What Is Observability? 2
    Observability Isn’t Purely an Operational Concern 3
    Conclusion 3

2. **Monitoring and Observability. 5**
    Alerting Based on Monitoring Data 6
    Best Practices for Alerting 7
    Conclusion 9

3. **Coding and Testing for Observability. 11**
    Coding for Failure 12
    Testing for Failure 13
    Conclusion 15

4. **The Three Pillars of Observability. 17**
    Event Logs 17
    Metrics 21
    Tracing 24
    The Challenges of Tracing 27
    Conclusion 28

5. **Conclusion. 29**

CHAPTER 1

The Need for Observability

Infrastructure software is in the midst of a paradigm shift. Containers, orchestrators, microservices architectures, service meshes, immutable infrastructure, and functions-as-a-service (also known as “serverless”) are incredibly promising ideas that fundamentally change the way software is built and operated. As a result of these advances, the systems being built across the board—at companies large and small—have become more distributed, and in the case of containerization, more ephemeral.

Systems are being built with different reliability targets, requirements, and guarantees. Soon enough, if not already, the network and underlying hardware failures will be robustly abstracted away from software developers. This leaves software development teams with the sole responsibility of ensuring that their applications are good enough to make capital out of the latest and greatest in networking and scheduling abstractions.

In other words, better resilience and failure tolerance from off-the-shelf components means that—assuming said off-the-shelf components have been understood and configured correctly—most failures not addressed by application layers within the callstack will arise from the complex interactions between various applications. Most organizations are at the stage of early adoption of cloud native technologies, with the failure modes of these new paradigms still remaining somewhat nebulous and not widely advertised. To successfully maneuver this brave new world, gaining visibility into the behavior of applications becomes more pressing than ever before for software development teams.

Monitoring of yore might have been the preserve of operations engineers, but observability isn’t purely an operational concern. This is a book authored by a software engineer, and the target audience is primarily other software developers, not solely operations engineers or site reliability engineers (SREs). This book introduces the idea of observability, explains how it’s different from traditional operations-centric monitoring and alerting, and most importantly, why it's so topical for software developers building distributed systems.

What Is Observability?

Observability might mean different things to different people. For some, it's about logs, metrics, and traces. For others, it's the old wine of monitoring in a new bottle. The overarching goal of various schools of thought on observability, however, remains the same—bringing better visibility into systems.

Observability Is Not Just About Logs, Metrics, and Traces

Logs, metrics, and traces are useful tools that help with testing, understanding, and debugging systems. However, it's important to note that plainly having logs, metrics, and traces does not result in observable systems.

In its most complete sense, observability is a property of a system that has been designed, built, tested, deployed, operated, monitored, maintained, and evolved in acknowledgment of the following facts:

- No complex system is ever fully healthy.
- Distributed systems are pathologically unpredictable.
- It's impossible to predict the myriad states of partial failure various parts of the system might end up in.
- Failure needs to be embraced at every phase, from system design to implementation, testing, deployment, and, finally, operation.
- Ease of debugging is a cornerstone for the maintenance and evolution of robust systems.

The Many Faces of Observability

The focus of this report is on logs, metrics, and traces. However, these aren't the only observability signals. Exception trackers like the open source Sentry can be invaluable, since they furnish information about thread-local variables and execution stack traces in addition to grouping and de-duplicating similar errors or exceptions in the UI.

Detailed profiles (such as CPU profiles or mutex contention profiles) of a process are sometimes required for debugging. This report does not cover techniques such as SystemTap or DTrace, which are of great utility for debugging standalone programs on a single machine, since such techniques often fall short while debugging distributed systems as a whole.

Also outside the scope of this report are formal laws of performance modeling such as universal scalability law, Amdahl's law, or concepts from queuing theory such as Little's law. Kernel-level instrumentation techniques, compiler inserted instrumentation points in binaries, and so forth are also outside the scope of this report.

Observability Isn't Purely an Operational Concern

An observable system isn't achieved by plainly having monitoring in place, nor is it achieved by having an SRE team carefully deploy and operate it.

Observability is a feature that needs to be enshrined into a system at the time of system design such that:

- A system can be built in a way that lends itself well to being tested in a realistic manner (which involves a certain degree of testing in production).
- A system can be tested in a manner such that any of the hard, actionable failure modes (the sort that often result in alerts once the system has been deployed) can be surfaced during the time of testing.
- A system can be deployed incrementally and in a manner such that a rollback (or roll forward) can be triggered if a key set of metrics deviate from the baseline.
- And finally, post-release, a system can be able to report enough data points about its health and behavior when serving real traffic, so that the system can be understood, debugged, and evolved.

None of these concerns are orthogonal; they all segue into each other. As such, observability isn't purely an operational concern.

Conclusion

Observability isn't the same as monitoring, but does that mean monitoring is dead? In the next chapter, we'll discuss why observability does *not* obviate the need for monitoring, as well as some best practices for monitoring.

CHAPTER 2

Monitoring and Observability

No discussion on observability is complete without contrasting it to monitoring. Observability isn't a substitute for monitoring, nor does it obviate the need for monitoring; they are complementary. The goals of monitoring and observability, as shown in Figure 2-1, are different.

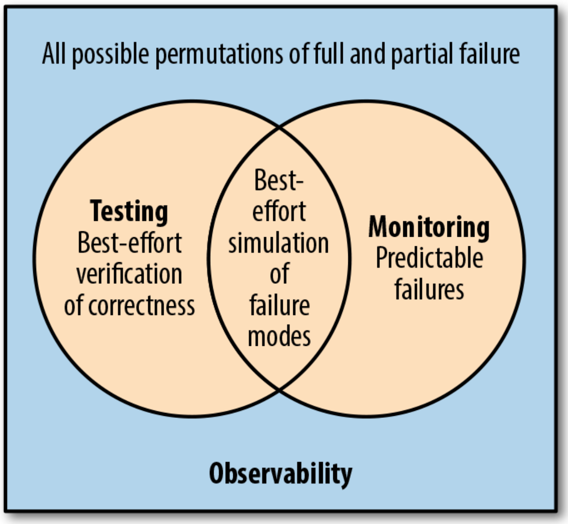
All possible permutations of full and partial failure
Testing
Best-effort verification of correctness
Best-effort simulation of failure modes
Monitoring
Predictable failures
Observability

Figure 2-1. Observability is a superset of both monitoring and testing; it provides information about unpredictable failure modes that couldn't be monitored for or tested

Observability is a superset of monitoring. It provides not only high-level overviews of the system's health but also highly granular insights into the implicit failure modes of the system. In addition, an observable system furnishes ample context about its inner workings, unlocking the ability to uncover deeper, systemic issues.

Monitoring, on the other hand, is best suited to report the overall health of systems and to derive alerts.

Alerting Based on Monitoring Data

Alerting is inherently both failure- and human-centric. In the past, it made sense to “monitor” for and alert on symptoms of system failure that:

- Were of the predictable nature
- Would seriously affect users
- Required human intervention to be remedied as soon as possible

Systems becoming more distributed has led to the advent of sophisticated tooling and platforms that abstract away several of the problems that human- and failure-centric monitoring of yore helped uncover. Health-checking, load balancing, and taking failed services out of rotation are features that platforms like Kubernetes provide out of the box, freeing operators from needing to be alerted on such failures.

Blackbox and Whitebox Monitoring

Traditionally, much of alerting was derived from blackbox monitoring. *Blackbox monitoring* refers to observing a system from the outside—think Nagios-style checks. This type of monitoring is useful in being able to identify the *symptoms* of a problem (e.g., “error rate is up” or “DNS is not resolving”), but not the triggers across various components of a distributed system that led to the symptoms.

Whitebox monitoring refers to techniques of reporting data from inside a system. For systems internal to an organization, alerts derived from blackbox monitoring techniques are slowly but surely falling out of favor, as the data reported by systems can result in far more meaningful and actionable alerts compared to alerts derived from external pings. However, blackbox monitoring still has its place, as some parts (or even all) of infrastructure are increasingly being outsourced to third-party software that can be monitored only from the outside.

However, there’s a paradox: even as infrastructure management has become more automated and requires less human elbow grease, understanding the lifecycle of applications is becoming harder. The failure modes now are of the sort that can be:

- *Tolerated*, owing to relaxed consistency guarantees with mechanisms like eventual consistency or aggressive multitiered caching

- *Alleviated* with graceful degradation mechanisms like applying backpressure, retries, timeouts, circuit breaking, and rate limiting
- *Triggered* deliberately with load shedding in the event of increased load that has the potential to take down a service entirely

Building systems on top of relaxed guarantees means that such systems are, by design, not necessarily going to be operating while 100% healthy at any given time. It becomes unnecessary to try to predict every possible way in which a system might be exercised that could degrade its functionality and alert a human operator. It's now possible to design systems where only a small sliver of the overall failure domain is of the hard, urgently human-actionable sort. Which begs the question: where does that leave alerting?

Best Practices for Alerting

Alerting should still be both hard failure–centric and human-centric. The goal of using monitoring data for alerting hasn't changed, even if the scope of alerting has shrunk.

Monitoring data should at all times provide a bird's-eye view of the overall health of a distributed system by recording and exposing high-level metrics over time across all components of the system (load balancers, caches, queues, databases, and stateless services). Monitoring data accompanying an alert should provide the ability to drill down into components and units of a system as a first port of call in any incident response to diagnose the scope and coarse nature of any fault.

Additionally, in the event of a failure, monitoring data should immediately be able to provide visibility into the impact of the failure as well as the effect of any fix deployed.

Lastly, for the on-call experience to be humane and sustainable, all alerts (and monitoring signals used to derive them) need to *actionable*.

What Monitoring Signals to Use for Alerting?

A good set of metrics used for monitoring purposes are the USE metrics and the RED metrics. In the book *Site Reliability Engineering* (O'Reilly), Rob Ewaschuk proposed the four golden signals (latency, errors, traffic, and saturation) as the minimum viable signals to monitor for alerting purposes.

The *USE methodology* for analyzing system performance was coined by Brendan Gregg. The USE method calls for measuring utilization, saturation, and errors of primarily system resources, such as available free memory (utilization), CPU run queue length (saturation), or device errors (errors).

The *RED method* was proposed by Tom Wilkie, who claims it was "100% based on what I learned as a Google SRE." The RED method calls for monitoring the request rate, error rate, and duration of request (generally represented via a histogram), and is necessary for monitoring request-driven, application-level metrics.

Debugging "Unmonitorable" Failures

The key to understanding the pathologies of distributed systems that exist in a constant state of elasticity and entropy is to be able to debug armed with evidence rather than conjecture or hypothesis. The degree of a system's observability is the degree to which it can be debugged.

Debugging is often an iterative process that involves the following:

- Starting with a high-level metric
- Being able to drill down by introspecting various fine-grained, contextual observations reported by various parts of the system
- Being able to make the right deductions
- Testing whether the theory holds water

Evidence cannot be conjured out of thin air, nor can it be extrapolated from aggregates, averages, percentiles, historic patterns, or any other forms of data primarily collected for monitoring.

An unobservable can prove to be impossible to debug when it fails in a way that one couldn't proactively monitor.

Observability Isn't a Panacea

Brian Kernighan famously wrote in the book *Unix for Beginners* in 1979:

> The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

When debugging a single process running on a single machine, tools like GDB helped one observe the state of the application given its inputs. When it comes to distributed systems, in the absence of a distributed debugger, observability data from the various components of the system is required to be able to effectively debug such systems.

It's important to state that observability doesn't obviate the need for careful thought. Observability data points can lead a developer to answers. However, the process of knowing what information to expose and how to examine the evidence (observations) at hand—to deduce likely answers behind a system's idiosyncrasies in production—still requires a good understanding of the system and domain, as well as a good sense of intuition.

More importantly, the dire need for higher-level abstractions (such as good visualization tooling) to make sense of the mountain of disparate data points from various sources cannot be overstated.

Conclusion

Observability isn't the same as monitoring. Observability also isn't a purely operational concern. In the next chapter, we'll explore how to incorporate observability into a system at the time of system design, coding, and testing.

CHAPTER 3

Coding and Testing for Observability

Historically, testing has been something that referred to a pre-production or pre-release activity. Some companies employed—and continue to employ—dedicated teams of testers or QA engineers to perform manual or automated tests for the software built by development teams. Once a piece of software passed QA, it was handed over to the operations team to run (in the case of services) or shipped as a product release (in the case of desktop software or games).

This model is slowly but surely being phased out (at least as far as *services* go). Development teams are now responsible for testing as well as operating the services they author. This new model is incredibly powerful. It truly allows development teams to think about the scope, goal, trade-offs, and payoffs of the entire spectrum of testing in a manner that's realistic as well as sustainable. To craft a holistic strategy for understanding how services function and to gain confidence in their correctness before issues surface in production, it becomes salient to be able to pick and choose the right subset of testing techniques given the availability, reliability, and correctness requirements of the service.

Software developers are acclimatized to the status quo of upholding production as sacrosanct and not to be fiddled around with, even if that means they always verify in environments that are, at best, a pale imitation of the genuine article (production). Verifying in environments kept as identical to production as possible is akin to a dress rehearsal; while there are some benefits to this, it's not quite the same as performing in front of a full house.

Pre-production testing is something ingrained in software engineers from the very beginning of their careers. The idea of experimenting with live traffic is either seen as the preserve of operations engineers or is something that's met with alarm. Pushing some amount of regression testing to post-production monitoring requires not just a change in mindset and a certain appetite for risk, but more importantly an overhaul in system design, along with a solid investment in good release engineering practices and tooling.

In other words, it involves not just architecting for failure, but, in essence, coding and testing for failure when the default was coding (and testing) for success.

Coding for Failure

Coding for failure entails acknowledging that systems will fail, being able to debug such failures is of paramount importance, and enshrining debuggability into the system from the ground up. It boils down to three things:

- Understanding the operational semantics of the application
- Understanding the operational characteristics of the dependencies
- Writing code that's debuggable

Operational Semantics of the Application

Focusing on the operational semantics of an application requires developers and SREs to consider:

- How a service is deployed and with what tooling
- Whether the service is binding to port 0 or to a standard port
- How an application handles signals
- How process starts on a given host
- How it registers with service discovery
- How it discovers upstreams
- How the service is drained off connections when it's about to exit
- How graceful (or not) the restarts are
- How configuration—both static and dynamic—is fed to the process
- The concurrency model of the application (multithreaded, purely single threaded and event driven, actor based, or a hybrid model)
- The way the reverse proxy in front of the application handles connections (pre-forked, versus threaded, versus process based)

Many organizations see these questions as something that's best abstracted away from developers with the help of either platforms or standardized tooling. Personally, I believe having at least a baseline understanding of these concepts can greatly help software engineers.

Operational Characteristics of the Dependencies

We build on top of increasingly leaky abstractions with failure modes that are not well understood. Here are some examples of such characteristics I've had to be conversant with in the last several years:

- The default read consistency mode of the Consul client library (the default is usually "strongly consistent," which isn't something you necessarily want for service discovery)
- The caching guarantees offered by an RPC client or the default TTLs
- The threading model of the official Confluent Python Kafka client and the ramifications of using it in a single-threaded Python server
- The default connection pool size setting for pgbouncer, how connections are reused (the default is LIFO), and whether that default is the best option for the given Postgres installation topology

Understanding such dependencies better has sometimes meant changing only a single line of configuration somewhere or overriding the default provided by a library, but the reliability gains from changes have been immense.

Debuggable Code

Writing debuggable code involves being able to ask questions in the future, which in turn involves the following:

- Having an understanding of the instrumentation format of choice (be it metrics or logs or exception trackers or traces or a combination of these) and its pros and cons
- Being able to pick the best instrumentation format given the requirements of the given service and the operational quirks of the dependencies

This can be daunting, and coding accordingly (and more crucially, being able to *test* accordingly) entails being able to appreciate these challenges and to address them head-on at the time of writing code.

Testing for Failure

It's important to understand that testing is a best-effort verification of the correctness of a system as well as a best-effort simulation of failure modes. It's impossible to predict every possible way in which a service might fail and write a regression test case for it, as E. W. Dijkstra has pointed out:

> The first moral of the story is that program testing can be used very effectively to show the presence of bugs but never to show their absence.

Unit tests only ever test the behavior of a system against a specified set of inputs. Furthermore, tests are conducted in very controlled (often heavily mocked) environments. While the few who do fuzz their code benefit from having their code tested against a set of randomly generated input, fuzzing can comprehensively test against the set of inputs to only *one* service. End-to-end testing might allow for some degree of holistic testing of the system, but complex systems fail in complex ways, and there is no testing under the sun that enables one to predict every last vector that could contribute toward a failure.

This isn't to suggest that testing is useless. If nothing else, testing enables one to write better, maintainable code. More importantly, research has shown that something as simple as "testing error handling code could have prevented 58% of catastrophic failures" in many distributed systems. The renaissance of tooling aimed to understand the behavior of our services in production does not obviate the need for pre-production testing.

However, it's becoming increasingly clear that a sole reliance on pre-production testing is largely ineffective in surfacing even the known-unknowns of a system. Testing for failure, as such, involves acknowledging that certain types of failures can only be surfaced in the production environment.

Testing in production has a certain stigma and negative connotations linked to cowboy programming, insufficient or absent unit and integration testing, as well as a certain recklessness or lack of care for the end-user experience.

When done poorly or haphazardly, "testing in production" does, in fact, very much live up to this reputation. Testing in production is by no means a substitute for pre-production testing, nor is it, by any stretch, easy. Being able to successfully and safely test in production requires a significant amount of diligence and rigor, a firm understanding of best practices, as well as systems designed from the ground up to lend themselves well to this form of testing.

In order to be able to craft a holistic and safe process to effectively test services in production, it becomes salient to not treat "testing in production" as a broad umbrella term to refer to a ragbag of tools and techniques.

"Testing in production" encompasses the entire gamut of techniques across three distinct phases: deploy, release, and post-release (Figure 3-1).

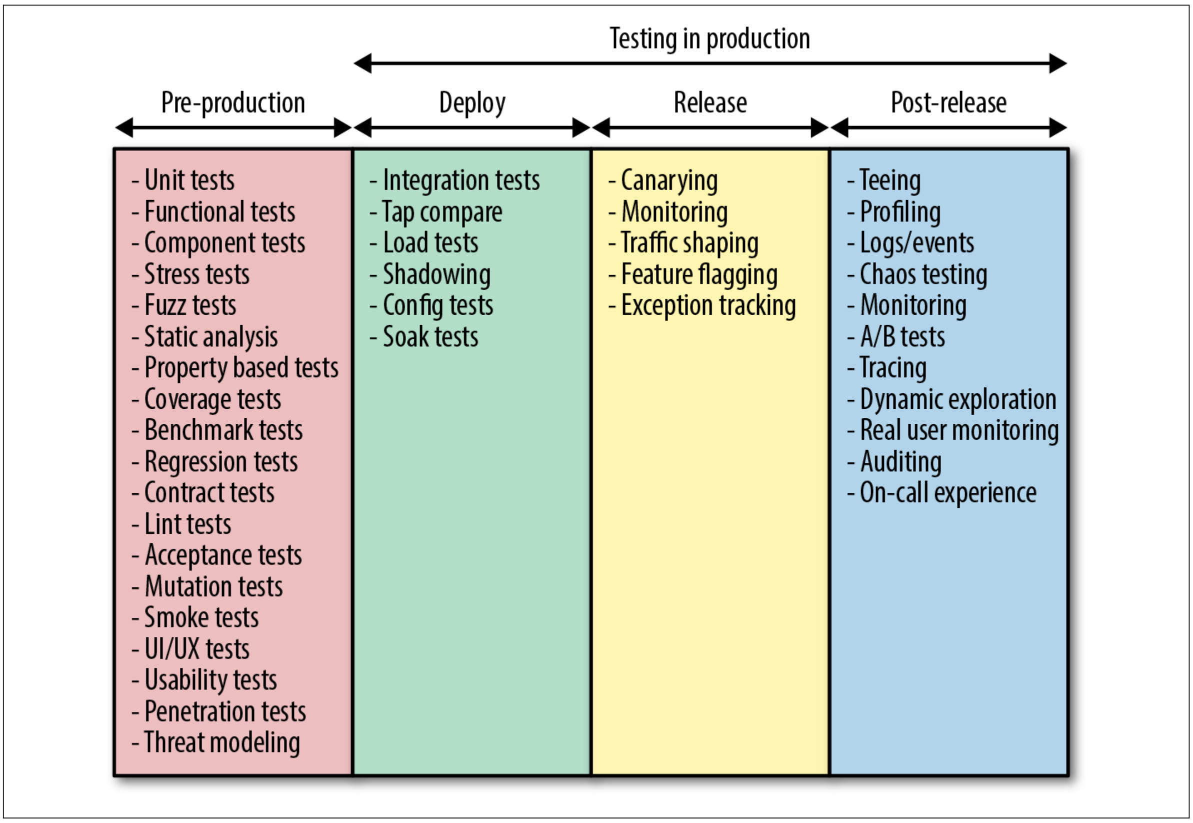
Testing in production
Pre-production
Deploy
Release
Post-release
- Unit tests
- Functional tests
- Component tests
- Stress tests
- Fuzz tests
- Static analysis
- Property based tests
- Coverage tests
- Benchmark tests
- Regression tests
- Contract tests
- Lint tests
- Acceptance tests
- Mutation tests
- Smoke tests
- UI/UX tests
- Usability tests
- Penetration tests
- Threat modeling
- Integration tests
- Tap compare
- Load tests
- Shadowing
- Config tests
- Soak tests
- Canarying
- Monitoring
- Traffic shaping
- Feature flagging
- Exception tracking
- Teeing
- Profiling
- Logs/events
- Chaos testing
- Monitoring
- A/B tests
- Tracing
- Dynamic exploration
- Real user monitoring
- Auditing
- On-call experience

Figure 3-1. The three phases of testing in production

The scope of this report doesn't permit me to delve deeper into the topic of testing in production, but for those interested in learning more, I've written a detailed post about the topic.

The overarching fact about testing in production is that it's impossible to do so without measuring how the system under test is performing in production. Being able to test in production requires that testing be halted if the need arises. This in turn means that one can test in production only if one has the following:

- A quick feedback loop about the behavior of the system under test
- The ability to be on the lookout for changes to key performance indicators of the system

For an HTTP service, this could mean attributes like error rate and latencies of key endpoints. For a user-facing service, this could additionally mean a change in user engagement. Put differently, testing in production essentially means proactively "monitoring" the test that's happening in production.

Conclusion

Testing in production might seem daunting, way above the pay grade of most engineering organizations. The goal of testing in production isn't to eliminate all manner of system failures. It's also not something one gets for free. While not easy or risk-free, undertaken meticulously, testing in production can greatly build confidence in the reliability of complex distributed systems.

CHAPTER 4

The Three Pillars of Observability

Logs, metrics, and traces are often known as the three pillars of observability. While plainly having access to logs, metrics, and traces doesn't necessarily make systems more observable, these are powerful tools that, if understood well, can unlock the ability to build better systems.

Event Logs

An *event log* is an immutable, timestamped record of discrete events that happened over time. Event logs in general come in three forms but are fundamentally the same: a timestamp and a payload of some context. The three forms are:

Plaintext
: A log record might be free-form text. This is also the most common format of logs.

Structured
: Much evangelized and advocated for in recent days. Typically, these logs are emitted in the JSON format.

Binary
: Think logs in the Protobuf format, MySQL binlogs used for replication and point-in-time recovery, systemd journal logs, the `pflog` format used by the BSD firewall `pf` that often serves as a frontend to `tcpdump`.

Debugging rare or infrequent pathologies of systems often entails debugging at a very fine level of granularity. Event logs, in particular, shine when it comes to providing valuable insight along with ample context into the long tail that averages and percentiles don't surface. As such, event logs are especially helpful for uncovering emergent and unpredictable behaviors exhibited by components of a distributed system.

Failures in complex distributed systems rarely arise because of one specific event happening in one specific component of the system. Often, various possible triggers across a highly interconnected graph of components are involved. By simply looking at discrete events that occurred in any given system at some point in time, it becomes impossible to determine all such triggers. To nail down the different triggers, one needs to be able to do the following:

- Start with a symptom pinpointed by a high-level metric or a log event in a specific system
- Infer the request lifecycle across different components of the distributed architecture
- Iteratively ask questions about interactions among various parts of the system

In addition to inferring the fate of a request throughout its lifecycle (which is usually short lived), it also becomes necessary to be able to infer the fate of a system as a whole (measured over a duration that is orders of magnitudes longer than the lifecycle of a single request).

Traces and metrics are an abstraction built on top of logs that pre-process and encode information along two orthogonal axes, one being request-centric (trace), the other being system-centric (metric).

The Pros and Cons of Logs

Logs are, by far, the easiest to generate. The fact that a log is just a string or a blob of JSON or typed key-value pairs makes it easy to represent any data in the form of a log line. Most languages, application frameworks, and libraries come with support for logging. Logs are also easy to instrument, since adding a log line is as trivial as adding a print statement. Logs perform really well in terms of surfacing highly granular information pregnant with rich local context, so long as the search space is localized to events that occurred in a single service.

The utility of logs, unfortunately, ends right there. While log *generation* might be easy, the performance idiosyncrasies of various popular logging libraries leave a lot to be desired. Most performant logging libraries allocate very little, if any, and are extremely fast. However, the default logging libraries of many languages and frameworks are not the cream of the crop, which means the application as a whole becomes susceptible to suboptimal performance due to the overhead of logging. Additionally, log messages can also be lost unless one uses a protocol like RELP to guarantee reliable delivery of messages. This becomes especially important when log data is used for billing or payment purposes.

RELP Isn't a Silver Bullet

RELP is a protocol that uses a command-response model (the command and the response is called a RELP *transaction*). The RELP client issues commands, and the RELP server responds to these commands.

The RELP server is designed to throttle the number of outstanding commands to conserve resources. Opting to use RELP means making the choice to apply backpressure and block the producers if the server can't process the commands being issued fast enough.

While such stringent requirements might apply to scenarios when every log line is critical or is legally required for auditing purposes, monitoring and debugging rarely, if ever, calls for such strict guarantees and the attendant complexity.

Last, unless the logging library can dynamically sample logs, logging excessively has the capability to adversely affect application performance as a whole. This is exacerbated when the logging isn't asynchronous and request processing is blocked while writing a log line to disk or `stdout`.

To Sample, or Not To Sample?

An antidote often proposed to the cost overhead of logging is to sample intelligently. *Sampling* is the technique of picking a small subset of the total population of event logs generated to be processed and stored. This subset is expected to be a microcosm of the corpus of events generated in a system.

Sampling isn't without its fair share of issues. For one, the efficacy of the sampled dataset is contingent on the chosen keys or features of the dataset based on which the sampling decision is made. Furthermore, for most online services, it becomes necessary to determine how to dynamically sample so that the sample rate is self-adjusting based on the shape of the incoming traffic. Many latency-sensitive systems have stringent bounds on, for instance, the amount of CPU time that can be spent on emitting observability data. In such scenarios, sampling can prove to be computationally expensive.

No talk on sampling is complete without mentioning probabilistic data structures capable of storing a summary of the entire dataset. In-depth discussion of these techniques is outside the scope of this report, but there are good O'Reilly resources for those curious to learn more.

On the processing side, raw logs are almost always normalized, filtered, and processed by a tool like Logstash, fluentd, Scribe, or Heka before they're persisted in a data store like Elasticsearch or BigQuery. If an application generates a large volume of logs, then the logs might require further buffering in a broker like Kafka before they can be processed by Logstash. Hosted solutions like BigQuery have quotas one cannot exceed.

On the storage side, while Elasticsearch might be a fantastic search engine, running it carries a real operational cost. Even if an organization is staffed with a team of operations engineers who are experts in operating Elasticsearch, other drawbacks may exist. Case in point: it's not uncommon to see a sharp downward slope in the graphs in Kibana, not because traffic to the service is dropping, but because Elasticsearch cannot keep up with the indexing of the sheer volume of data being thrown at it. Even if log ingestion processing isn't an issue with Elasticsearch, no one I know of seems to have fully figured out how to use Kibana's UI, let alone *enjoy* using it.

Logging as a Stream Processing Problem

Event data isn't used exclusively for application performance and debugging use cases. It also forms the source of all analytics data. This data is often of tremendous utility from a business intelligence perspective, and usually businesses are willing to pay for both the technology and the personnel required to make sense of this data in order to make better product decisions.

The interesting aspect here is that there are striking similarities between questions a business might want answered and questions software engineers and SREs might want answered during debugging. For example, here is a question that might be of business importance:

> Filter to outlier countries from where users viewed this article fewer than 100 times in total.

Whereas, from a debugging perspective, the question might look more like this:

> Filter to outlier page loads that performed more than 100 database queries.
>
> Show me only page loads from France that took more than 10 seconds to load.

Both these queries are made possible by events. Events are structured (optionally typed) key-value pairs. Marrying business information along with information about the lifetime of the request (timers, durations, and so forth) makes it possible to repurpose analytics tooling for observability purposes.

Log processing neatly fits into the bill of Online Analytics Processing (OLAP). Information derived from OLAP systems is not very different from information derived for debugging or performance analysis or anomaly detection at the edge of the system. One way to circumvent the issue with ingest delay in Elasticsearch —or indexing-based stores in general—is by treating log processing as a stream processing problem to deal with large data volumes by using minimal indexing.

Most analytics pipelines use Kafka as an event bus. Sending enriched event data to Kafka allows one to search in real time over streams with KSQL, a streaming SQL engine for Kafka.

Enriching business events that go into Kafka with additional timing and other metadata required for observability use cases can be helpful when repurposing existing stream processing infrastructures. A further benefit this pattern provides is that this data can be expired from the Kafka log regularly. Most event data required for debugging purposes are valuable only for a relatively short period of time after the event has been generated, unlike any business-centric information that is evaluated and persisted by an ETL job. Of course, this makes sense only when Kafka already is an integral part of an organization. Introducing Kafka into a stack purely for real-time log analytics is a bit of an overkill, especially in non-JVM shops without any significant JVM operational expertise.

An alternative is Humio, a hosted and on-premises solution that treats log processing as a stream processing problem. Log data can be streamed from each machine directly into Humio without any pre-aggregation. Humio uses sophisticated compression algorithms to effectively compress and retrieve the log data. Instead of *a priori* indexing, Humio allows for real-time, complex queries on event stream data. Since Humio supports text-based logs (the format that the vast majority of developers are used to grepping), ad hoc schema on reads allows users to iteratively and interactively query log data. Yet another alternative is Honeycomb, a hosted solution based on Facebook's Scuba that takes an opinionated view of accepting only structured events, but allows for read-time aggregation and blazing fast real-time queries over millions of events.

Metrics

Metrics are a numeric representation of data measured over intervals of time. Metrics can harness the power of mathematical modeling and prediction to derive knowledge of the behavior of a system over intervals of time in the present and future.

Since numbers are optimized for storage, processing, compression, and retrieval, metrics enable longer retention of data as well as easier querying. This makes metrics perfectly suited to building dashboards that reflect historical trends. Metrics also allow for gradual reduction of data resolution. After a certain period of time, data can be aggregated into daily or weekly frequency.

The Anatomy of a Modern Metric

One of the biggest drawbacks of historical time-series databases has been the *identification* of metrics that didn't lend itself very well to exploratory analysis or filtering.

The hierarchical metric model and the lack of tags or labels in older versions of Graphite especially hurt. Modern monitoring systems like Prometheus and newer versions of Graphite represent every time series using a metric name as well as additional key-value pairs called *labels*. This allows for a high degree of dimensionality in the data model.

A metric in Prometheus, as shown in Figure 4-1, is identified using both the metric name and the labels. The actual data stored in the time series is called a *sample*, and it consists of two components: a *float64* value and a millisecond precision timestamp.

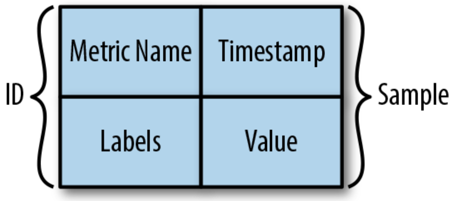

Figure 4-1. A Prometheus metric sample

It's important to bear in mind that metrics in Prometheus are immutable. Changing the name of the metric or adding or removing a label will result in a new time series.

Advantages of Metrics over Event Logs

By and large, the biggest advantage of metrics-based monitoring over logs is that unlike log generation and storage, metrics transfer and storage has a constant overhead. Unlike logs, the cost of metrics doesn't increase in lockstep with user traffic or any other system activity that could result in a sharp uptick in data.

With metrics, an increase in traffic to an application will not incur a significant increase in disk utilization, processing complexity, speed of visualization, and operational costs the way logs do. Metrics storage increases with more permutations of label values (e.g., when more hosts or containers are spun up, or when new services get added or when existing services get instrumented more), but client-side aggregation can ensure that metric traffic doesn't increase proportionally with user traffic.

NOTE

Client libraries of systems like Prometheus aggregate time-series samples in-process and submit them to the Prometheus server upon a successful scrape (which by default happens once every few seconds and can be configured). This is unlike statsd clients that send a UDP packet every time a metric is recorded to the statsd daemon (resulting in a directly proportional increase in the number of metrics being submitted to statsd compared to the traffic being reported on!).

Metrics, once collected, are more malleable to mathematical, probabilistic, and statistical transformations such as sampling, aggregation, summarization, and correlation. These characteristics make metrics better suited to report the overall health of a system.

Metrics are also better suited to trigger alerts, since running queries against an in-memory, time-series database is far more efficient, not to mention more reliable, than running a query against a distributed system like Elasticsearch and then aggregating the results before deciding if an alert needs to be triggered. Of course, systems that strictly query only in-memory structured event data for alerting might be a little less expensive than Elasticsearch. The downside here is that the operational overhead of running a large, clustered, in-memory database, even if it were open source, isn't something worth the operational trouble for most organizations, especially when there are far easier ways to derive equally actionable alerts. Metrics are best suited to furnish this information.

The Drawbacks of Metrics

The biggest drawback with both application logs and application metrics is that they are *system* scoped, making it hard to understand anything else other than what's happening inside a particular system. Sure, metrics can also be request scoped, but that entails a concomitant increase in label fan-out, which results in an increase in metric storage.

With logs without fancy joins, a single line doesn't give much information about what happened to a request across all components of a system. While it's possible to construct a system that correlates metrics and logs across the address space or RPC boundaries, such systems require a metric to carry a UID as a label.

Using high cardinality values like UIDs as metric labels can overwhelm time-series databases. Although the new Prometheus storage engine has been optimized to handle time-series churn, longer time-range queries will still be slow. Prometheus was just an example. All popular existing time-series database solutions suffer performance under high cardinality labeling.

When used optimally, logs and metrics give us complete omniscience into a silo, but nothing more. While these might be sufficient for understanding the performance and behavior of individual systems, both stateful and stateless, they aren't sufficient to understand the lifetime of a request that traverses multiple systems.

Distributed tracing is a technique that addresses the problem of bringing visibility into the lifetime of a request across several systems.

Tracing

A *trace* is a representation of a series of causally related distributed events that encode the end-to-end request flow through a distributed system.

Traces are a representation of logs; the data structure of traces looks almost like that of an event log. A single trace can provide visibility into both the path traversed by a request as well as the structure of a request. The path of a request allows software engineers and SREs to understand the different services involved in the path of a request, and the structure of a request helps one understand the junctures and effects of asynchrony in the execution of a request.

Although discussions about tracing tend to pivot around its utility in a microservices environment, it's fair to suggest that any sufficiently complex application that interacts with—or rather, contends for—resources such as the network, disk, or a mutex in a nontrivial manner can benefit from the advantages tracing provides.

The basic idea behind tracing is straightforward—identify specific points (function calls or RPC boundaries or segments of concurrency such as threads, continuations, or queues) in an application, proxy, framework, library, runtime, middleware, and anything else in the path of a request that represents the following:

- Forks in execution flow (OS thread or a green thread)
- A hop or a fan out across network or process boundaries

Traces are used to identify the amount of work done at each layer while preserving causality by using *happens-before* semantics. Figure 4-2 shows the flow of a single request through a distributed system. The trace representation of this request flow is shown in Figure 4-3. A trace is a directed acyclic graph (DAG) of *spans*, where the edges between spans are called *references*.

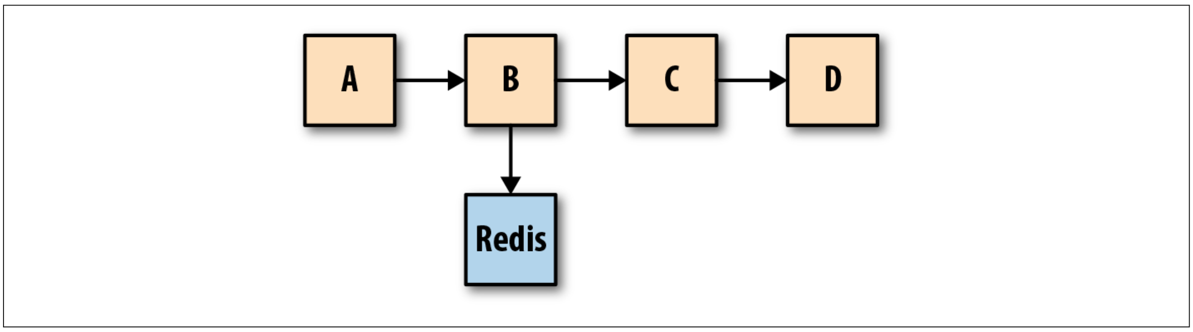

Figure 4-2. A sample request flow diagram

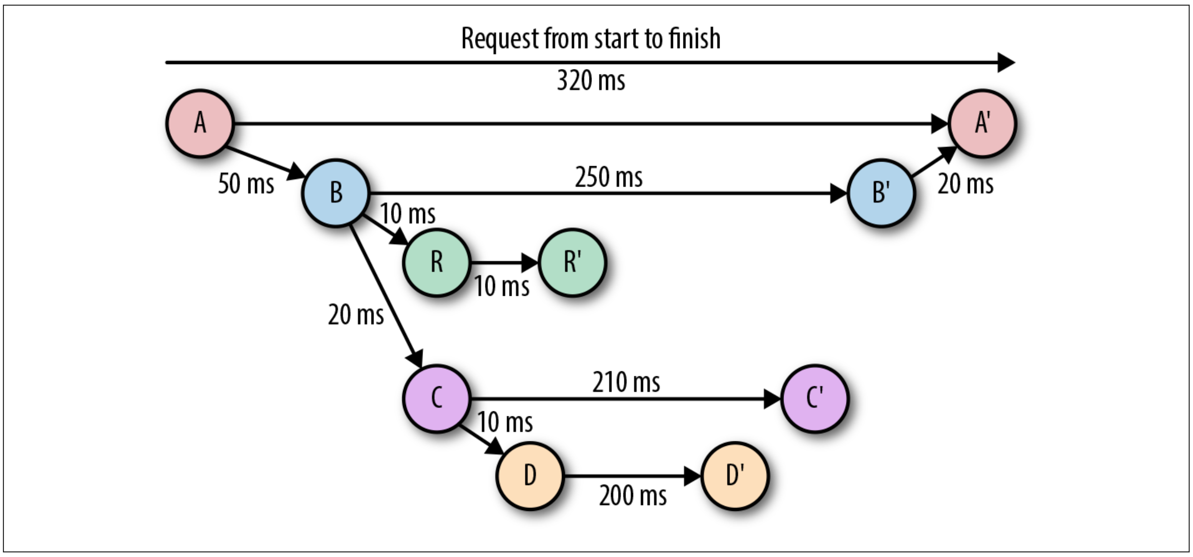
Request from start to finish
320 ms
A
A'
50 ms
B
250 ms
B'
20 ms
10 ms
R
R'
10 ms
20 ms
C
210 ms
C'
10 ms
D
200 ms
D'

Figure 4-3. The various components of a distributed system touched during the lifecycle of a request, represented as a directed acyclic graph

When a request begins, it's assigned a globally unique ID, which is then propagated throughout the request path so that each point of instrumentation is able to insert or enrich metadata before passing the ID around to the next hop in the meandering flow of a request. Each hop along the flow is represented as a span (Figure 4-4). When the execution flow reaches the instrumented point at one of these services, a record is emitted along with metadata. These records are usually asynchronously logged to disk before being submitted out of band to a collector, which then can reconstruct the flow of execution based on different records emitted by different parts of the system.

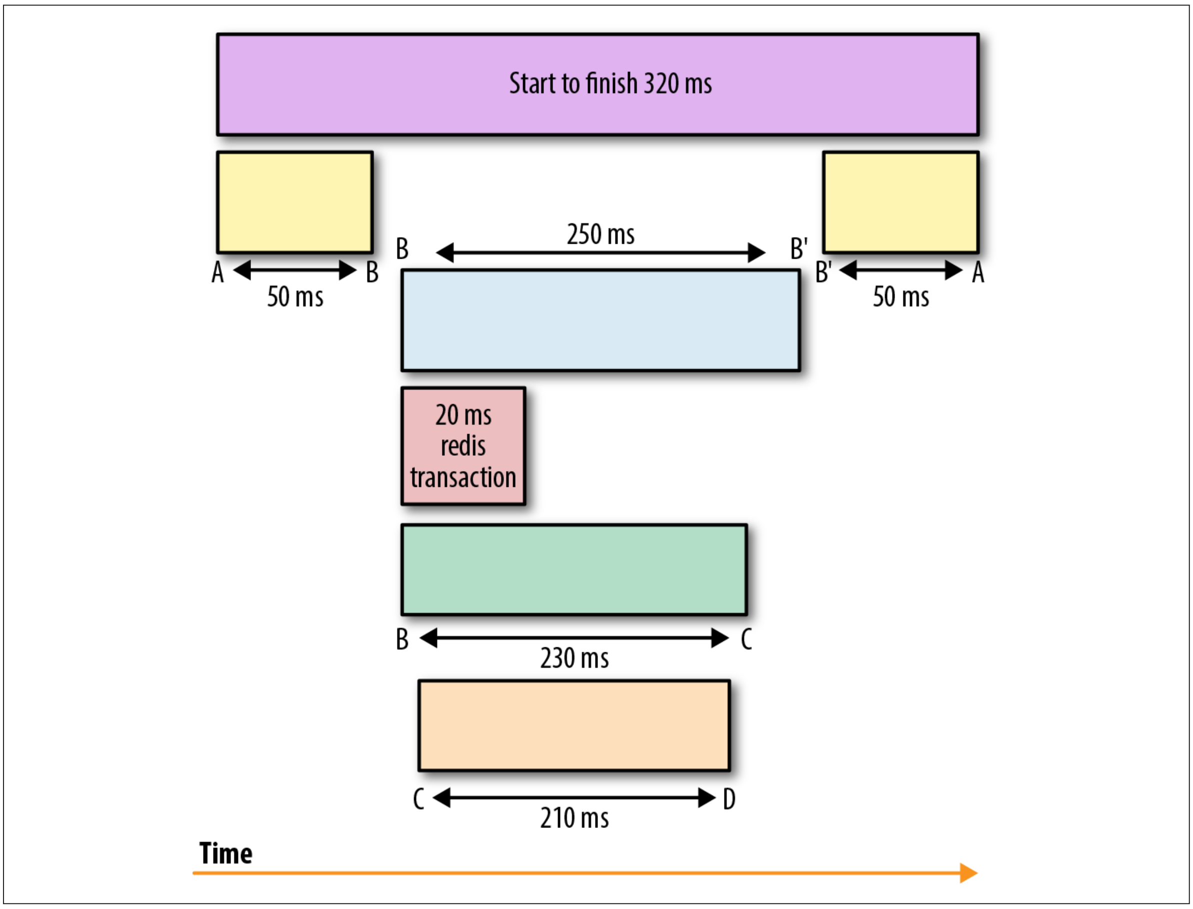

Figure 4-4. A trace represented as spans: span A is the root span, span B is a child of span A

Collecting this information and reconstructing the flow of execution while preserving causality for retrospective analysis and troubleshooting enables one to better understand the lifecycle of a request.

Most importantly, having an understanding of the entire request lifecycle makes it possible to debug requests spanning multiple services to pinpoint the source of increased latency or resource utilization. For example, Figure 4-4 indicates that the interaction between service C and service D was what took the longest. Traces, as such, largely help one understand the *which* and sometimes even the *why* (e.g., *which* component of a system is even touched during the lifecycle of a request and is slowing the response?).

The use cases of distributed tracing are myriad. While used primarily for inter service dependency analysis, distributed profiling, and debugging steady-state problems, tracing can also help with chargeback and capacity planning.

Zipkin and Jaeger are two of the most popular OpenTracing-compliant open source distributed tracing solutions. (OpenTracing is a vendor-neutral spec and instrumentation libraries for distributed tracing APIs.)

The Challenges of Tracing

Tracing is, by far, the hardest to retrofit into an existing infrastructure, because for tracing to be truly effective, every component in the path of a request needs to be modified to propagate tracing information. Depending on whom you ask, you'd either be told that having gaps in the flow of a request doesn't outweigh the cons (since adding tracing piecemeal is seen as better than having no tracing at all, as having partial tracing helps eke out nuggets of knowledge from the fog of war) or be told that these gaps are blind spots that make debugging harder.

The second problem with tracing instrumentation is that it's not sufficient for developers to instrument their code alone. A large number of applications in the wild are built using open source frameworks or libraries that might require additional instrumentation. This becomes all the more challenging at places with polyglot architectures, since every language, framework, and wire protocol with widely disparate concurrency patterns and guarantees needs to cooperate. Indeed, tracing is most successfully deployed in organizations that use a core set of languages and frameworks uniformly across the company.

The cost of tracing isn't quite as catastrophic as that of logging, mainly because traces are almost always sampled heavily to reduce runtime overhead as well as storage costs. Sampling decisions can be made:

- At the start of a request before any traces are generated
- At the end, after all participating systems have recorded the traces for the entire course of the request execution
- Midway through the request flow, when only downstream services would then report the trace

All approaches have their own pros and cons, and one might even want to use them all.

Service Meshes: A New Hope for the Future?

While tracing has been difficult to implement, the rise of service meshes make integrating tracing functionality almost effortless. Data planes of service meshes implement tracing and stats collections at the proxy level, which allows one to treat individual services as blackboxes but still get uniform and thorough observability into the mesh as a whole. Applications that are a part of the mesh will still need to forward headers to the next hop in the mesh, but no additional instrumentation is necessary.

Lyft famously got tracing support for every last one of its services by adopting the service mesh pattern, and the only change required at the application layer was to forward certain headers. This pattern is incredibly useful for retrofitting tracing into existing infrastructures with the least amount of code change.

Conclusion

Logs, metrics, and traces serve their own unique purpose and are complementary. In unison, they provide maximum visibility into the behavior of distributed systems. For example, it makes sense to have the following:

- A counter and log at every major entry and exit point of a request
- A log and trace at every decision point of a request

It also makes sense to have all three semantically linked such that it becomes possible at the time of debugging:

- To reconstruct the codepath taken by reading a trace
- To dervive request or error ratios from any single point in the codepath

Sampling exemplars of traces or events and correlating to metrics unlocks the ability to click through a metric, see examples of traces, and inspect the request flow through various systems. Such insights gleaned from a combination of different observability signals becomes a must-have to truly be able to debug distributed systems.

CHAPTER 5

Conclusion

As my friend Brian Knox, who manages the Observability team at DigitalOcean, said,

> The goal of an Observability team is not to collect logs, metrics, or traces. It is to build a culture of engineering based on facts and feedback, and then spread that culture within the broader organization.

The same can be said about observability itself, in that it's not about logs, metrics, or traces, but about being data driven during debugging and using the feedback to iterate on and improve the product.

The *value* of the observability of a system primarily stems from the business and organizational value derived from it. Being able to debug and diagnose production issues quickly not only makes for a great end-user experience, but also paves the way toward the humane and sustainable operability of a service, including the on-call experience. A sustainable on-call is possible only if the engineers building the system place primacy on designing reliability into a system. Reliability isn't birthed in an on-call shift.

For many, if not most, businesses, having a good alerting strategy and time-series based "monitoring" is probably all that's required to be able to deliver on these goals. For others, being able to debug needle-in-a-haystack types of problems might be what's needed to generate the most value.

Observability, as such, isn't an absolute. Pick your own observability target based on the requirements of your services.

About the Author

Cindy Sridharan is a distributed systems engineer who works on building and operating distributed systems. She likes thinking about building resilient and maintainable systems and maintains a blog where she writes about her experience building systems.