Foreword

What is observability? It seems like this should be a settled question after over five years of sustained hype, millions of dollars of marketing, dozens of companies of various sizes selling observability products, and an expansive landscape of open source software in the field.

While academic definitions exist borrowing from safety studies, control theory, cognitive systems engineering, and more, there's still an identifiable gap between 'observability-as-defined' and 'observability-as-practiced'.

Part of the problem is that the definition and practice of observability depends on the domain; here we focus on observability of cloud-native production software applications. Beyond the question of domain, confusion about observability has been exacerbated by the voluminous marketing budgets of both incumbent and insurgent developer tooling companies. All of this presents a challenge: if observability feels so straightforward as we preach it, why does it often feel so unattainable in practice?

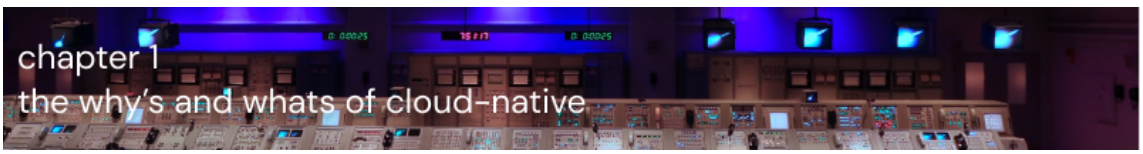
This project seeks to understand and answer that question, as well as several more. In lieu of being dictated, we're trying to find the answer to it together. To that end, you can think of what you're about to read as an 'open access' white paper. Our goal is to present an opinion on what observability -- specifically, observability for cloud-native systems -- is, and where it should go. Rather than accept the status quo definitions, we would like to build something new, and build it in a collaborative way.

Much of this content assumes some familiarity with the basics of cloud-native software and systems, although we try to keep it rather light. We're expressing a vision, not a blueprint, and visions aren't that useful if you can't interpret them. Important concepts are introduced and explained in-line where possible, although please refer to the glossary if there's something you don't understand.

Finally, this project isn't meant to just be a stand-alone report to be written and then rot. This is meant to be a collaborative exercise, created and iterated on in public. If you find something you disagree with, open an issue and let's discuss it. Do you have a personal story that aligns with something we're talking about? Let us know, add an appendix. We gladly welcome case studies and anecdotes about the success and failures of current observability practice.

I hope this artifact acts not only as an impetus and inspiration to the entire observability community, but that the way in which we build it can act as a model as well. Let's figure out where we want to go, then go out and build it. Please, contribute to this journey, and let's go together.

@austinlparker 3/25/22



The Why and What of Cloud-Native

Software touches nearly every aspect of our lives and has rewritten the playing field in nearly every industry. The fact that software is *strategic* is firmly "not news." That software deserves *massive levels of investment* isn't news, either.

What's less clear is *how* to successfully turn these massive investments into innovating and reliable software applications: it's easy enough to hire a few thousand developers, but how do we get thousands of developers to collaborate efficiently on a single piece of deployable software?

Here's the thing: we don't! Instead, **to make software development scale, we split these massive, strategic software applications into pieces – aka, "microservices" – and use cloud-native devops practices and cloud-native technology stacks to develop, deploy, and operate them.** At the same time, we want to get out of the business of procuring, maintaining, or even *thinking* about hardware, so all of the above happens in hosted infrastructure run by multiple cloud vendors.

What is 'The Cloud'?

The cloud is more than just "servers on demand", it represents a fundamental shift in how we operate software. It allows us to get out of the business of thinking about hardware by turning computing resources into a *utility*, similar to public electricity or water. These resources run the gamut -- servers on demand, yes, but also speech recognition, planet-scale databases, practically infinite storage, and much more.

What makes the cloud special is its elasticity and it's programmability. You can scale a cloud application from one, to hundreds, to hundreds of millions of users with little additional effort. You can also do this without relying on human beings to rack servers or push buttons via programmatic automation. That said, this potential requires new architectures and strategies to fully realize its potential, and that's where cloud-native software comes into play.

So what is "Cloud-Native" exactly?

Cloud-native software is not the same thing as "production software that's hosted by a public cloud provider" – cloud-native software is characterized by software architecture and development practices that can scale linearly, even with thousands of developers or billions of end-users.

The Cloud Native Computing Foundation defines Cloud-Native Technology as follows:

> *Cloud native technologies, also referred to as the cloud native stack, are the technologies used to build cloud native applications. These technologies enable organizations to build and run scalable applications in modern and dynamic environments such as public, private, and hybrid clouds, while leveraging cloud computing benefits to their fullest. They are designed from the ground up to exploit the capabilities of cloud computing and containers, service meshes, microservices, and immutable infrastructure exemplify this approach.*

Ironically, some cloud-native apps don't even run in the cloud! A Kubernetes-based app with hundreds of microservices running on premises is more "cloud-native" than a lift-and-shift Oracle business application running in AWS.

Why Cloud-Native Matters

Does this approach actually increase release velocity? Yes, and sometimes by factors of more than 100x. If software requires money and time, cloud-native can act as a force multiplier to one, or both, of these inputs.

We can leverage the implicit expertise of other developers and engineers who build these platforms, granting us greater reliability. Our money is more flexible, as we don't have to invest upfront capital in the care and feeding of servers to run our programs.

This benefit is so large that it extends across architectures -- analyst reports find cloud ROI to be over 4x self-hosted, which is pretty amazing all on its own. Beyond the simple math, cloud-native software bridges a solutions gap as well, giving you the means to leverage expertise in *many* fields to enhance what you can build. Need some text-to-speech? There's managed cloud services for that. Want to know what's going on with your software in production? A constellation of open source and software as a service tools are out there for your perusal.

Beyond the economic arguments, though, cloud-native paints a wonderfully collaborative picture of how software can benefit our lives and world. The consolidation of data centers has created a green revolution, where computing power has gone up but energy use has gone down. Open source software, once regarded as a land of free-speech absolutists and cranks, has been commercialized and funded by the world's largest companies to build even more cloud-native software, freely given to the next generation of developers to solve the challenges of today and tomorrow. The adaptive capacity and scaling benefits of the cloud were on display in full effect as a global workforce moved from offices to work-from-home, and Zoom became the lifeline for our personal and professional lives. It's hard to imagine the future without the cloud, but it's even harder to imagine the present.

End-users and Engineers, Transactions and Resources

Observability must be more than an arcane set of tools wielded by experts or the desire to view a bunch of system diagnostics. Our model of observability must reflect the human needs of the end-users and engineers involved in cloud-native systems, and reflect the transactions and resources that comprise those systems.

End-users

Cloud-native software applications ultimately exist **to serve their end-users.** This is why all engineers – or at least all *good* engineers – obsess over the quality of their end-users' experiences with that software. These end-users do *not* care about Kubernetes, they do *not* care about microservices, and they do *not* care about observability; rather, they care that the application does its job, quickly and without getting in the way. The end-users don't care about software, they care about what the software is doing for them.

Transactions

End-user interactions with cloud-native software ultimately boil down to requests that begin in a client (e.g., a mobile phone, a browser, or an SDK), propagate to the cloud-native services, do their work, then ultimately terminate, successfully or otherwise. These distinct and distributed journeys are called "transactions", and if all transactions are successful and expedient, the software is doing its job for the end-users.

The journeys that these transactions take are often extraordinarily elaborate in real-world cloud-native software. To make matters even more complicated, these transactions are not independent: they commingle and interfere with one another as they consume shared resources like databases, queueing systems (like Kafka), and even in the network itself.

The work the transactions do can be modeled at many different levels of detail: one could reasonably describe transactions in terms of nested OSI Layer 7 HTTP calls, as function calls, or even as individual CPU instructions. All of these are valid ways to think about transactions, though depending on the context they may not be viable ways to sufficiently answer questions about the transactions or the resources they consume along the way.

Addressing complexity and interference effects

If all transactions could be represented as a series of distinct, independent steps taken by a single, logical thread it'd be far easier to model them. However, there are several confounding variables that we must address -- scheduling, interference effects, and trust.

Non-blocking scheduling is common in cloud-native apps but complicates transaction tracing telemetry. Asynchronous dispatch of a transaction (consider multiple independent calls to a profile service, a products service, an order history service that powers a product purchase in a web client, and an async job to send a user email confirmation) leads to context growth, and requires adaptations to our transactional models. While to a user, this is one logical request -- "I want to view my dashboard" -- it's up to the developer of a system to decide how this should be represented in the telemetry itself. Further complicating this is the actual organization and how it's built -- if different teams, in different time zones, each own distinct segments of the transaction lifecycle, what's the best way to model it so that teams can understand how changes to other parts of the system affect theirs?

The inverse of this is also true; since async dispatch causes combinatorial growth, batching causes context collapse as multiple logical transactions fold into one. Batching is one of the most common ways to reduce the *resource* footprint under heavy transactional load, but as soon as transactions are batched, a trace-per-transaction model breaks down. This modeling issue with "coalescing effects" has been a noted problem since the original Dapper tracing paper, and remains problematic in the Span model.

Finally, any sufficiently complex transaction will involve the crossing of at least one, and potentially many, secure boundaries. Since telemetry generation happens close to where a the work of a transaction is being done, this also means that user data can cross those boundaries as well. Telemetry that generates a shared, single-transaction context (like a trace) starting from an untrusted client, moving into a trusted server, can leave purchase for potential vulnerabilities. In addition, different transactions in the same multi-tenant system can operate at differing trust levels; even authorized privilege escalation within a client can intermingle trusted and untrusted data in the same telemetry context. Addressing this requires not only data hygiene and defensive coding practices, but a strong understanding of security boundaries within a transaction.

Addressing these complexities requires innovation in telemetry generation and analysis. Our tools must be able to understand links between different types of telemetry, so that we can create traces and metrics that not only respect trust boundaries, but allow us to isolate potentially untrustworthy data. We need to be able to spatially visualize transactions as they fan-out or fan-in, without becoming unmanageably complex and difficult to read.

Resources

As these distributed transactions hop from service to service, enter and exit caches, make system calls, and generally do work, they consume *resources*. Resources are nearly always *shared* across multiple

transactions, both over time (i.e., in serial) and/or concurrently (i.e., in parallel). As with transactions above, resources also can be modeled at many different levels of detail: a mutex lock is a resource, CPU cycles are a resource, a microservice is a resource, a Kafka queue is a resource, and even an AWS quota can be thought of as a resource.

Crucially, resources are *finite*: you can peg a CPU, you can run out of memory, and you can flood a Kafka queue. As anyone who's observed production software can attest, resource exhaustion can get very ugly very quickly for the transactions that are queueing up and/or failing as a result. We know these common symptoms well -- database queries that used to take moments now taking minutes, login services timing out that are normally bulletproof, pods crash-looping in OOMKilled state.

Often when we discuss 'resiliency' in the context of cloud-native systems, what we really mean is 'resource resilience', not 'transaction resilience'. Orchestration systems such as Kubernetes will blissfully attempt to restart failed pods due to application crashes and errors, for example. Some of this is due to an implicit bias towards 'ops' concerns in the SRE discipline (after all, it's "site reliability" not "app reliability"), but much of it is because our reliability model can't accurately capture or model the interactions between transactions and resources.[^reliabilitySidebar]

Engineers

Finally we come to the engineers. For most readers here, "we're talking about you."

Engineers care about their end-users, and thus they certainly *care* about transactions. That being said, *caring* about something is different than *controlling* something: engineers cannot and **should not** directly control the behavior of individual transactions, as nothing could be less scalable. **Engineers only control resources,** not transactions.

Engineers and End-Users: The Great Tragedy

And this brings us to the tragic relationship of well-meaning engineers and the end-users that depend on their cloud-native software application!

It's like this:

1. **End-users** only care about the **transactions**,
2. **Engineers** care about their **end-users**,
3. ... but engineers can only *control* their **resources** – not the transactions that flow through them.

This is why we see engineers who care deeply about end-users spend their time staring at dashboards showing CPU usage and pod restart rates. Observability's challenge is to not only help engineers understand transactions and the end-user experience, but to automate the extremely challenging task of understanding how those transactions and the many layers of resources interact.

Resources and Transactions: The Great Duality

The interdependencies between transactions and resources are manifold and often complex. Transactions can manipulate resources, resource faults can modify the flow of transactions. These aren't necessarily two frictionless, non-interactive spheres. Let's illustrate through an example.

Suppose we have a transaction that depends on resource like Kafka. Most of the time, a Kafka cluster will support multiple topics with a whole battery of consumers reading and producers writing to them. The transaction is, for the most part, ignorant about Kafka except through an abstraction layer buried several libraries deep; It simply writes messages and then polls for a response. This transaction is part of several

other, broader transactions, that extend all the way back to a command issued by an end-user through a CLI. Kafka itself is performing transactions constantly, as message brokers receive and dispatch messages to topics and requests come in for the replay of new statements on each topic.

Again, we have transactions and resources, layered upon each other. Based on our responsibilities, we might care more or less about certain parts of this entire ball of wax, but at the end of the day there's only one thing everyone can agree on -- someone is making a request, and if it doesn't work, they're not gonna be terribly thrilled.

Now, suppose this transaction begins to take longer than it usually does. From the end-user perspective, nothing's changed; They entered a command, and they're waiting for a response. Our sub-transactions as well don't see anything different, as they're operating more or less independently, writing and reading messages. However, on the resource side, Kafka's `client.consumer.lag` metric just spiked from a few milliseconds to nearly two minutes across all of our hosts for that topic.

Descriptively, it's easy enough to explain why -- if a resource which a transaction depends on starts suffering a performance anomaly, then our transaction will necessarily suffer. If we are only able to look at these two data sets independently, then that's where our investigation of this problem would end. The Kafka (Ops) team notices consumer lag has increased, posts a message to Slack that performance is degraded, and starts to comb through log files to try and understand why. The users, developers, and others who depend on that resource are left scratching their heads as to why performance is impacted unless they happen to be responsible for the abstraction that talks to Kafka, at which point they start to look at recent pushes to prod, to see if they broke something.

How do we solve this problem? If you're in a completely disconnected world, it's hard! Different teams own different parts of this problem, and maybe have entirely different monitoring stacks. Even if you have all of this different transactional and resource telemetry in the same tool, you've still got disconnected experiences where at best you can line up time windows with some deployment markers to try and narrow the search space.

Stepping back, this is really a bigger problem than it really seems! As your system grows to include more transaction and resource dependencies, the ownership of the health of those dependencies becomes more and more distributed. The cloud is resilient, but it isn't bulletproof -- think about what happens when us-east-1 suffers an adverse event. Heck, think about what happens when ancillary parts of the software development toolchain have problems -- GitHub failures, npm outages, or simply Greg (the guy who knows how to fix the build) being on a hike with no cell service. Not only do our transactions and resources have layered dependencies, our entire development lifecycle has them, and a lot of them are completely outside our control.

How do we make sense of this? How can we define reliability, quality, or even a basic understanding of what we require from each party involved in this whole mess? Our answer is in a couple of short acronyms.

SLIs and SLOs

The Service Level Indicator (SLI) and Service Level Objective (SLO) have been popularized by the Site Reliability Engineering (SRE) movement, a practice which seeks to professionalize the highly technical and often invisible work of building resilient systems.[^sre]

An SLI is some quantitative measure of a service -- like request latency, error rate, or throughput. They are also measurements of a *transaction*, mid-stream. Another way to phrase this is that the combination of all relevant SLIs for each sub-transaction in a transaction is the SLI of that transaction -- i.e., if my user-facing transaction has twelve steps and I sum the duration it takes for each step, I know the total duration. Indeed,

instead of thinking about transaction health as a trace, or a metric, or the lack of an error log, I can think of each transaction as an SLI unto itself.

If we extend that metaphor a bit, then the SLO describes the acceptable range for those transactional SLIs.

It follows that the SLO becomes a contract between transactions and resources to define acceptable performance bounds, and thus the SLO is a contract between the end-users and engineers themselves. Instead of thinking about observability as the ability to 'explore unknown unknowns' or 'quickly reduce search space during incidents', we can begin to think of it as a way to define and iterate on SLIs and use those to formalize SLOs with each other, setting and communicating expectations between apps and servers, transactions and resources, end-users and engineers, managers and employees, teams and organizations. The SLO becomes a mutually agreed upon instrument for talking about change, prioritizing work, or assessing reliability standards -- and observability becomes our way to monitor and understand these changes.

[^txnDisambiguation]: To be clear, 'transaction' is any interaction with a system and not just a 'database transaction'. [^sre]: You can read more about Site Reliability Engineering in the SRE Handbook from Google. [^reliabilitySidebar]: Chaos Engineering is perhaps the biggest counter-factual to this proposition, and many interesting examples of transaction/resource contention arise from its principles. For example, the LinkedOut failure injection pipeline that allows for individual faults to be bubbled into backend systems. That said, I would advance that Chaos Engineering is still somewhat niche outside of the most sophisticated teams.

The Anatomy of Observability

A big reason that the definition of observability has become so polluted is because popularized definitions are so broad as to be meaningless. Saying that you can "answer questions you don't know to ask yet", or "find a needle in a haystack" are -- for lack of a better phrase -- junky marketing. You know what I can put a bunch of data into and answer arbitrary questions with? MySQL.

Observability, then, isn't simply defined by the desired end state of "understanding a system by its outputs" (because there are no systems that can be controlled otherwise; a system without output is one that might as well not exist) -- it's a combination of tools and techniques, workflows and products, bound together with a shared language of SLIs and SLOs.

The building blocks of observability are pretty straightforward in this model -- telemetry, persistence, and workflows; Each building on the other. Telemetry is data and metadata about the behavior of our transactions and resources. Persistence is how we collect and store that data. Workflows are how we analyze and make sense of it. Let's look deeper into each, starting with telemetry!

Telemetry

In the prior chapter, we discussed three different telemetry signals that can be emitted by transactions and resources. These signals -- logs, metrics, and traces -- are sometimes referred to as the "three pillars" of observability. However, this is a misnomer borne from the rough way that the observability space has

matured backed up mostly by product marketing rather than practical reality. These signals are all interrelated, interconnected, and indispensible to observability tools.

Let's discuss each of them first, and how they can be used to understand transactions and resources. Then, we'll discuss how signal quality and richness impacts observability.

Logs, Metrics, and Traces

Logging is the practice of writing human-readable text data to the console or a file. This data can be structured into a predefined schema for processing by a log management system, or unstructured free text -- usually, both, and often inside the same component. Logs help engineers understand transaction or resource state, especially in exceptional cases such as process failure.

Metrics are compact, numerical, statistic data designed for easy and cheap transmission and storage. Metrics can be emitted by transactions or resources, and are usually visualized and analyzed as a time-series plot. Metrics help engineers understand aggregate system health, or track key indicators of transaction performance.

Traces are a special form of structured, machine-readable, logging. While different tracing systems can vary in their details, they all share the idea of a transaction-level context -- this means that a single trace is mapped to a single, logical request through a distributed system. A trace is comprised of multiple spans, where each span represents a logical unit of work executed as a part of the transaction.

All of these signals share a few commonalities -- they comprise some sort of message or value, and a collection of metadata. The message varies based on signal type; Logs emit a string of human-readable text, metrics contain a number, and traces emit the duration and name of work performed. The metadata tends to be similar in nature if not specifics -- everything from host names, source or destination IP addresses, customer identifiers, version numbers, and much more.

While transactions and resources can both emit any of these signals, there are some that make more sense than others. The following table illustrates a few examples of this:

	Logs	Metrics	Traces
Transactions	A stack trace dumped during process crash	A counter of concurrent user sessions in a given service	A span that wraps a database call
Resources	Information about processes that started or stopped	The amount of memory in use per-process	The work being done by a forked shell script

The challenge of cloud-native observability isn't simply in emitting each of these signals, however -- we need to understand why they fail in isolation, and how to relate and connect them to each other.

The Momentum Tax

As said in *Mass Effect*, Sir Issac Newton is the deadliest son-of-a-b*tch in the galaxy. The same is unfortunately true in the realm of telemetry. I would suggest that the overwhelming majority of complex systems in use today are built on a rats nest of variegated telemetry signals with vastly different and often conflicting users and stakeholders. If you believe this to be false, then I invite you to count how many crucial business processes relating to the software you maintain or build occur within Microsoft Excel.

The hard truth is that any observability practice is bound by the least observable component of any of its critical paths. What does it mean to be 'observable'? Broadly, to have accurate, correct, and descriptive

telemetry data produced reliability and quickly by your components. As we'll see, extant telemetry signals tend to fail at multiple parts of this test. As to why we still *use* these signals, well, look at the heading -- momentum. It's hard to justify replacing your telemetry for anything other than a massive perceived value, because the people who pay the tax are usually *engineers* rather than customers. Let's discuss some of the flaws of each signal as they're usually implemented.

Logging tends to suffer the most from the momentum tax, as it's the easiest thing to produce. The first thing most of us probably learned how to program was an application that wrote "Hello, World!" to the console, after all. Unfortunately, most developers education about how to log started and stopped at this point. It's one of those skills that we don't really train except through osmosis as we become more experienced engineers. It's a "damned if you do, damned if you don't" signal, because it truly is the lowest common denominator of telemetry; Everything can emit it, and everything can read it, if you put in the work. In addition, logging has many suitors -- security teams, auditors, finance and operations, developers, and many more.

Metrics suffer as well, but in a more pernicious way. Rather than a lack of suitability, metrics tend to suffer from a lack of precision due to the ever-present fear of cardinality explosions or confusing and misleading names or attributes. Metrics tend to be underused in transaction telemetry specifically -- the requisite variety required to produce highly detailed time series at scale is, for most metric systems, unsustainable. A question like "show me the latency of a particular endpoint by customer ID" is easy when you have ten customers, hard when you have ten thousand, and nearly impossible when you have ten million. Resource metrics suffer from similar scaling issues, especially thanks to container orchestration and horizontal scaling.

Traces, being a 'newer' signal type, suffer less from this momentum tax. Generally the biggest challenge experienced here is divergent or discordant trace specifications and implementations as different teams implement tracing on specific sub-systems without connecting them to each other. Additionally, APM (Application Performance Monitoring) and RUM (Real User Monitoring) tools have led many engineers to only understand tracing in the context of *profiling* a single process rather than holistic system health.

With all that said, what *should* it look like?

Cloud-Native Telemetry Generation

Our discussion of transactions and resources touched on a point -- transactions consume resources, resource exhaustion makes transactions suffer. You can also think of this in the inverse, that a collection of resources is a state machine which produces desired transactions. This is an important distinction, because it raises very real questions about where we *should* generate telemetry, and how we aggregate it.

I would suggest a rather facile response, which is that we do *both* at the same time through context mechanisms that allow for us to tie the telemetry signals generated by our transactions and resources together. This necessarily means we need tools that are capable of correlating signals regardless of their component, and aggregating them based on the needs of operators rather than on arbitrary distinctions driven by their source. Logs, metrics, and traces all have a place -- and it's the same place, as an interrelated and interconnected braid of data.

We'll address the exact mechanisms by which this data can be generated in the next chapter, but I want to bring it up here before we start talking about what *good* telemetry looks like. As mentioned above, it's not enough to just be correct or accurate or descriptive -- you need to achieve all of these simultaneously. It's certainly true that all of the telemetry signals we previously discussed are convertible -- you can turn a log file into a stream of metrics, for example. However, this fails the other part of our equation, as adding processing delays for telemetry conversion fails to make it available quickly to operators. Fundamentally, though, **these signal types are just telemetry sugar**. Traces are a structured form of logging, metrics can

be generated from aggregate trace analysis, and logs are just text-based metrics of state. Once we stop thinking of these signals as independent from each other, or as driving unique and siloed analysis experiences, then we can start to imagine different possibilities.

Later chapters will dive more deeply into this topic, but I want to preview them here in order to frame the rest of this discussion. Some readers may ask why we spend so much time on these individual signals but at the same time seem to eschew them by saying they're all 'the same' -- it's because without getting this part right, you're building on a foundation of sand. Here's some statements about what 'correct' telemetry generation can get you --

- Accurate histograms and summaries generated from actual customer data rather than synthetic testing.
- Telemetry points that are automatically correlated through associativity with their respective transactions rather than time-windowing.
- Resource telemetry that can identify transaction dependency chains, such as specific jobs on a queue that are blocked by upstream consumer lag.
- Performance telemetry linked to user analytics, security analytics and logs, or any other data source.

> *More information on telemetry resolution and other signals can be found in this appendix*

Telemetry Quality

Telemetry, in general, is the most important part of observability; If you don't have telemetry, you can't do much else. It stands to reason, then, that the quality and quantity of your telemetry data has a direct and measurable impact on your observability practice in general. This doesn't simply impact the software systems analyzing your telemetry, but the engineers trying to make sense of it -- and it's those engineers who define what "quality" ultimately means here.

For many of us, the quality of our telemetry data varies drastically, even within a given signal. Consider the amount of varying logging libraries or targets you're likely to find in a sufficiently large application, the variance in metrics produced by different operating systems or frameworks, and tracing data produced by different resources or half-implemented omnibus tracing efforts.

This puts us in a bit of a pickle; We're generating more telemetry than ever, but none of it in the same way. Even within the context of a single transaction, there's no guarantee that we can associate the telemetry from multiple components together except via time windows (or fairly rudimentary links across shared attributes). Furthermore, transactional dependencies may emit critical telemetry in differing formats that our own, necessitating ever-more complex ETL pipelines in order to translate this data into a common format. The result is a byzantine series of pipes, data lakes, and dashboards that introduce real latency to the comprehension and value-generation aspects of telemetry utilization.

A simple example -- how do you represent the host name of the server that ran a piece of code? Is it the `hostname`, `host.name`, `HostName`, `host_name`, or `hostName`? These are all linguistically valid, but if all of my telemetry can't agree, then suddenly it becomes very difficult to group and filter on this attribute. What about if the semantic meaning of `hostname` differs between components or regions? This causes real, constant pain for developers today, and it needs to be solved.

The solution here is fairly simple in theory, if not in practice -- but it's this: we need standards, and they need to be open. A common API that can be shared between LOB applications and their dependencies -- along with a common format that both transactions and resources can emit telemetry data in -- goes a long way towards solving this pain point. We'll discuss this more in the next chapter, because thankfully, this solution exists.

Persistence

We've come a long way from the "record everything and let God sort it out" mindset of the 2010's. It's true that storage is [cheaper than ever(https://blog.dshr.org/2012/02/cloud-storage-pricing-history.html)], the increase in data volume easily negates the reduction in persistence costs. Additionally, an increasingly complex legal and regulatory environment -- that only promises to get more complex -- demands sophistication in the question of not only what data we store, but where we store it, how we store it, and how we attribute changes to it.

This makes persistence a key consideration in application and system telemetry data, and there's a lot to consider. Observability systems tend to suffer from a mismatch between reads and writes; This is to say, the way that telemetry is presented to the storage system is orthogonal to the way you'd like to read it back from that system. When we ingest these signals, they're often batched and mixed with a wide variety of metadata that covers multiple transactions or resources; When we read them back, we usually only want information about one. The individual data points are very small, but queries that you'd be interested in might aggregate hundreds of thousands of points to return a result. This presents many challenges in the design and operation of a production observability system; You can't just take something off the shelf and go, it really does require a lot of thought and effort!

The solutions we've seen to this problem usually involve highly specialized databases for different signals. Projects such as Cortex, OpenSearch, and Cassandra are all popular time series or NoSQL databases for storing metrics, logs, and traces, respectively. This fails a test, though -- observability is about more than just having these signals available, it's about the *integration* and *correlation* between these signals. Keeping all of our data in separate buckets with incompatible query languages and APIs (or torturously adapting metrics queries to support traces, or trying to use SQL as a lingua franca...) means that we're tilting our head and squinting at observability without really grasping the point.

What observability requires is specialized persistance that natively supports multiple types of telemetry, but also allows that telemetry data to be queried in a consistent fashion regardless of type. This allows us to perform actual cross-cutting queries and comparisons between signals, have a shared syntax to express aggregations, groups, filters, and so forth, and reduce the cost and maintenance burden required to operate the storage layer. This cost shouldn't just be thought of as 'how much am I spending to keep this stuff operating', either -- controlling how much you're spending to keep this data around, and figuring out how useful it is, is key.

Is everything an event?

There's an alternative theory about telemetry that agrees in spirit with the above sentiment. In summary, it claims that metrics, logs, and spans can all be generally classified as "events" -- a generic form of telemetry that represents some distinct, observable occurrence in your system. The storage layer for these events can then be optimized for one type of data, rather than for each individually.

While spiritually similar to the argument above, the 'everything is an event' position presents challenges at the persistence layer. Certain signals contain implicit expectations about persistence that need to be addressed -- for example, most people wouldn't expect a single trace to arbitrarily be kept for over a month, but they would expect a time series to be (albeit in a compacted form). Metrics exist in an extremely compact format to begin with; 'Up-scaling' them into an event adds metadata that effectively increases their storage cost over time. This also adds a burden to users, who need to pay special attention to how much data is being emitted to the storage engine, as it doesn't necessarily know about the different types of signals and can't make intelligent decisions about how or where to store things (i.e., moving useful-but-not-critical log or trace data to cold storage rather than keeping it warm)

In practice, rather than trying to genericize telemetry signals, it can be more helpful to think of the association between them via shared context at the point of emission -- we don't need to turn everything into an event if we're able to ensure that each signal is emitted with the appropriate attributes to ensure we can correlate them later across types. This is something we'll touch on, again, in the next chapter -- stay tuned.

Workflows

Once you've got some data, and you've got it stored, you wanna do something with it. Traditionally this has meant you pull up some sort of query builder or data explorer and start running queries, then pulling those queries into a dashboard. A link to that dashboard would be added to a runbook or wiki page, and it'd mostly sit there unneeded until trouble came a-calling, at which point whoever was on call would open it up and realize half the graphs didn't work any more because someone renamed the metrics last month and didn't update the charts.

Realistically, everything we've talked about before accretes into some sort of actual benefit over the existing approach. Observability isn't about simply swapping out your storage layer with something smarter, or improving the data quality of your telemetry signals, even if both of those are necessary for it. Think back to our earlier chapter, where we talked about transactions and resources -- our goal is to improve our ability to understand change by monitoring critical signals. The workflows that observability enables are what actually helps us do just that!

Monitoring and Alerting

The practice of monitoring looks a lot like what we described in the introduction to this section -- a bunch of dashboards that attempt to record signals that describe every failure state of a system which you need to peruse and interpret to determine why things failed. There's a lot of problems with this; As I mentioned before, our underlying data can shift and change leading to dashboard staleness, or a lack of breadth in the dashboard itself requires us to click between multiple tabs and systems in order to get a complete picture of the system state.

A better way to think of this is to change what we're actually monitoring. Dashboards full of visualizations over a collection of time series shouldn't be our primary interaction method with our observability tools. Monitoring needs to very explicitly connect resource and transaction health with the business goals that they support. This changes our frame of reference to monitoring our SLOs rather than our specific SLIs.

One way to think about this is to borrow from the field of industrial control and safety thinking[^safetyWhitepaper]. Safety-I focuses around incidents, and what goes wrong. Safety-II is far more concerned with what goes *right* in a system, and learning from that. Observability workflows should, on the balance, be more concerned with reporting and measuring acceptable outcomes in order to shape our thinking about our system, and how to build it in a more resilient way (aka 'model formation'). Traditional alerting and monitoring approaches are caught in a Safety-I mindset, but SLOs are much more useful to drive Safety-II approaches because they help you focus on what *should* be happening rather than what *is* happening.

SLOs provide a convenient framework for alerting, as well. Traditionally, we'd know 'something's wrong' because we would set alerts on our indicators around things like 'is this value too high or low for too long', or 'is the change from period to period greater than some arbitrary threshold'. While this can tell you there's smoke, it doesn't necessarily tell you if there's fire. What's worse in my mind is that these kind of alerts don't actually give you any sort of context or rationale for how they're impacting things. Service owners can set up a variety of alerts based on assumptions that were made in the past that may not have any real connection to how the service is being used today or it's current usage patterns.

SLO-based alerting, though, solves this problem neatly. An SLO is already going to have a built-in threshold at which you should start caring about it (when you're in violation) *and* logically consistent breakpoints at which you should be notified about changes (when you're burning down, when you've run out of error budget, when you're not using enough budget, etc.) that don't really change based on the context of the service you're using. An SLO is an SLO is an SLO, after all.

SLO-based monitoring and alerting also can make a huge difference in how teams outside of engineering interact with observability and reliability. SLOs are a natural way to communicate between engineering and business stakeholders on the tradeoffs between feature development and 'reliability engineering', or about the benefits of architectural and structural improvements to a codebase. Consistent alerting based on SLOs reduces on-call handoff errors and anxiety, as there's a consistent set of things you'll be alerted on.

Of course, this is only half of the equation -- once you've been alerted that you're burning down an SLO, you gotta figure out *why*.

Investigating Change and Difference

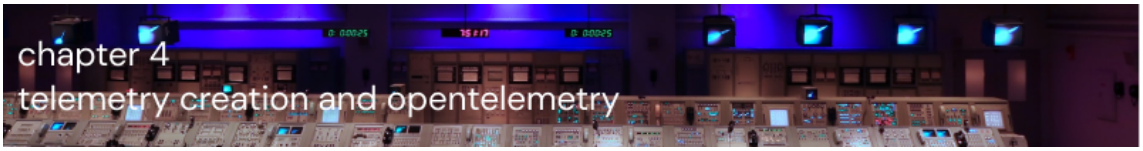
I'm gonna say that there's no outage or incident that ever existed without something changing to cause it. Maybe that change wasn't something *you* did -- it could be a customer sending malformed payloads, or a dependent service suddenly changing its API, or an underlying dependency being updated by a well-meaning security team, or an unexpected cloud failure.

There's two types of change that we should care about -- intentional and unintentional. These intentional changes are produced as we develop and push code out to our systems. They're the result of our CI/CD process, or dependency updates, or changes to our team structure. Observability acts as a flywheel on all of these, making us able to move forward more quickly and more safely, by ensuring we can identify the results of these deliberate changes. Did we actually fix that bug? Are we actually getting more sign-ups? Did we reduce how long it takes to load the page? While it's nice to muse about the high-minded pursuit of 'data exploration', most of us don't want to become Magellan in order to see the effect of merging a PR.

Unintentional changes are more pernicious, but they're equally important to understand. These unintentional changes can be the result of accidental dependency upgrades, emergent user behavior, certificate expiry, exercise of dormant code paths, and much more. Helpfully, the actual process of detecting change works about the same for both. What is different about unintentional changes is that our context differs greatly. We know when we've deployed code and are watching it roll out; We usually have a theory we're trying to prove or disprove. Unintentional change happens when we least expect it, from vectors we're unlikely to predict. In these cases, we need workflows that are deliberately crafted in to quickly present accurate hypotheses, present relevant correlations, and let us compare differences in broad system state across arbitrary time windows using a blend of telemetry signals.

It's actually very straightforward if you think about it -- the big story about observability has a lot more to do with how you think about monitoring than anything else. It's not just about storing a bunch of different data, it's not just about having traces and metrics and logs, it's not even about just using SLOs. The promise of observability is a radically different approach to how you interact with your telemetry data -- it turns you from being a statistician into a troubleshooter. It democratizes both data, and outcomes, so that everyone involved in a software business can understand how software health and business outcomes are interrelated.

[^safetyWhitepaper]: For a more comprehensive discussion of Safety-I and Safety-II, this whitepaper provides a decent overview.



Telemetry Creation and OpenTelemetry

The foundation of observability is high quality, ubiquitous, telemetry data. You need high quality signals that contain rich metadata, tooling that aids in the collection and processing of those signals, and APIs that are flexible enough to integrate into a wide variety of resources.

While this might sound straightforward, it's been a strictly aspirational goal for many years. Historically, everyone just kinda does whatever makes sense based on whatever tools have the most acclaim for their language or tech stack. There's some fairly broad 'standards' -- Apache Logging and syslog for logs, statsd and Prometheus for metrics, Zipkin for traces. Commercial vendors have generally reacted to this by creating agent-based pipelines that accept a variety of formats, then translate them to a vendor-specific dialect.

The status quo presents multiple challenges that need to be overcome. An agent-based model where telemetry data is hydrated with additional metadata after it's emitted is going to lack specific transactional context (as this context is generally lost after a transaction concludes), even if it can be correlated with resource metadata. Time-window correlation can be helpful, but in highly concurrent or asynchronous transactions, it usually isn't specific enough to narrow down your search space. Furthermore, the reliance on external agents and tools to translate this data into a common format locks us in to a specific commercial vendor to handle that data. This also stymies implementors of open source libraries, managed cloud services, and other resources that we might wish to collect telemetry signals from -- they're forced to use the lowest common denominator of signals to maximize compatibility with their end-users telemetry pipelines. Finally, all of these additional translation layers has a cost not only in implementation complexity, but in simple time -- if all of your telemetry data is five minutes old by the time you see it because it had to be written to logs, translated into metrics, then batched up with a bunch of other reports and fed into a general database, you're gonna have a bad time actually trying to fight fires as you'll always be a few steps behind whatever's actually happening -- or, worse, you're left wondering that a sudden lack of data is due to an outage or simply slow processing.

Enter OpenTelemetry, a broadly-supported project backed by the cloud native community and a host of monitoring and observability vendors. OpenTelemetry has a simple goal -- make observability a built-in feature of cloud-native software. It achieves this by providing a consistent API and SDK to generate telemetry signals, all backed by a shared context that allows you to connect specific signals to specific transactions. It provides a standardized exposition format, and semantic conventions for telemetry data, enabling better analytical and persistence options for end-users. In short, OpenTelemetry is the foundation for cloud-native observability practices.

> *Further Reading*
>
> *A more complete discussion of OpenTelemetry, along with more depth on the topics in this chapter can be found in The Future of Observability with OpenTelemetry.*

Independent of OpenTelemetry, however, there's the actual act of instrumentation itself, and how telemetry is created in the first place. Let's step back to talk about how we should think about instrumentation in general, before we summarize the specific role of OpenTelemetry in the cloud-native landscape.

Instrumentation and Granularity

Cloud-native observability requires high quality, *thoughtful*, instrumentation of software. This is a challenging ask, and it's probably the reason that so many teams fail to achieve their observability goals. Much existing instrumentation is either up-scaled (i.e., hydrating existing logging or metrics with additional metadata from an orchestration layer, like Kubernetes) or down-scaled (i.e., fine-grained function or line level stack traces and metrics being compacted or aggregated).

There's two maxims we should abide by, here -- first, white-box instrumentation is going to always be of a higher quality than black-box instrumentation, and second, your telemetry instruments should live one step away from what you're trying to understand. The first is self-explanatory; If your instrumentation is able to emit signals that are aware of the transactional context of your application, then you're going to have access to more useful data and metadata about that transaction. This means that telemetry creation should be something you think about as part of development rather than something that gets added in later. Imagine a world where you define the expected state of a transaction by implementing it as a trace, then you know you're done when the expected trace and the actual one match! White-box instrumentation allows for engineers to define their intended system state then validate that production matches up with this definition.

The second point requires a bit more nuance. What does 'one step away' mean? In practical terms, it means your telemetry code shouldn't necessarily live with your business logic. Imagine a simple HTTP server responding to API requests; What part of this code should be instrumented with tracing, logs, and metrics? The function handler that defines a route? The underlying HTTP library that handles connection management? The runtime or system libraries that are being called to open and close sockets? None of these are necessarily the *wrong* answer, but the right answer depends a great deal on what your workflows can support. If you mostly can control the code that you've written, then instrumenting at the library makes a lot of sense. If you can control the code and the libraries, then another layer at the runtime makes sense. If you control all of that, then system-level instrumentation makes a lot of sense.

These differing granularity's need to also be configurable, either through a rules engine, or heuristic-based algorithms (i.e., when *x* error rate is high, increase collection rate/depth, when *y* metadata is present disable collection). We'll talk more about it in a later chapter, but high ROI observability requires high granularity, but not in perpetuity.

What should be avoided, however, is over-indexing on a specific type of telemetry. Metrics, logs, and traces are all *required* to represent the work being done by transactions, and the resources that they consume. It's not three pipes, it's a single braid of unified, contextual data.

OpenTelemetry and Commodity Telemetry

How do we achieve this nirvana of granular, omnipresent telemetry data? As we mentioned above, OpenTelemetry exists as an open-source standard for creating and representing metrics, logs, and traces. The real goal of OpenTelemetry isn't necessarily to supplant all existing telemetry formats, but to provide a *lingua franca* for how that data is collected and expressed, to provide a shared context for associating transactional telemetry together, and to ensure that high-quality data is a built-in feature of the cloud-native ecosystem.

The status quo sucks, this much is true. Proprietary vendors spend millions on writing commodity integrations with popular open source libraries and packages, or on translation layers between existing telemetry formats and their own. If you look at the actual code, though, most of this is all the same stuff! There's only so many ways to get a trace or a metric from a SQL client or a HTTP library. OpenTelemetry promises to change this, by providing open source maintainers a single, well-supported, and un-opinionated

standard for writing their own instrumentation code *or* translating their existing instrumentation into the OpenTelemetry format.

To reiterate, because as of this writing, we don't think it's well understood -- OpenTelemetry is **not really an end-user API**. The core API and SDK is really intended to be integrated by upstream library or service authors. If you're a user of telemetry data, the goal is for it to simply be *present*, available on a specific endpoint, quietly waiting for you to connect a compatible analysis tool and begin inspecting it. If you think about it from this lens, many of the decisions the project has made makes more sense. Will it always be this way? Probably not -- as the project stabilizes and matures, more work will go into the end-user experience, and it'll become easier to use for non-integrators as well. That said, the original goal is still the goal; OpenTelemetry should be a checkbox, something that just exists in the background.

> ***Is OpenTelemetry Right For Me?***
>
> *Some readers may come upon the prior section and question if OpenTelemetry is the right answer for smaller projects, or toy ones. I'm not sure this is the right question to ask, necessarily. OpenTelemetry isn't doing a lot of stuff that's entirely new, after all -- the signal primitives it exposes have been used for many years. If I had to compare it to something, it'd be LXC Containers in terms of maturity -- it combines a lot of things that make sense to have together in a single place in the hope to spur adoption of the underlying format and concepts. Perhaps OpenTelemetry is waiting for its Docker equivalent, a friendlier API for using and deploying it? I would suggest that learning OpenTelemetry and implementing it for even smaller applications is useful and worthwhile as it helps you learn thoughtful practices around telemetry creation and annotation.*

With this in mind, it's easier to see how OpenTelemetry solves the instrumentation challenges posed in the last section. If it simply exists in your libraries, runtime environments, and kernels then the actual burden of instrumenting (or re-instrumenting) is greatly reduced. If it exists at multiple layers, then dynamically controlling the rate of telemetry generation isn't that hard -- you just start listening for more data, or for different data. If you're getting metrics, traces, and logs in a unified, context-aware format, then you aren't going to have to try and force everything to be represented as just a metric, or just a log, or just a trace, or some combination -- you'll have it all, intertwined, and cross-referenced.

Effective Monitoring

Traditional monitoring looks a lot like 'Safety-I', a model of system reliability that focuses on measuring and preventing *errors*. This helps explain why most existing monitoring practices are characterized by an "error-centric" design -- we don't have a great way to quantify what we're monitoring, exactly, without looking at errors. You can see this manifest in the acronyms and statements we make about reliability -- mean time to resolution, root cause, etc. These are all error-centric, in that they focus on errors (and the resolution of those errors) as the primary thing we should care about.

Cloud-native observability recognizes that errors and faults aren't something you can prevent, because they're a normal part of system operations. Instead, it seeks to redefine not only what we monitor, but *why* we perform monitoring. Monitoring isn't something we can throw away in its entirety, human operators will always need the ability to inspect system state. What we need to change is the instruments we use to monitor that state, and the processes we use to make sense of it.

Effective Dashboards

Current approaches to dashboards suffer from many issues, but most analysis of these systems focus on the symptomatic impacts rather than the root cause of why these issues occur in the first place. It's useful to think of the problem as a disconnect between the loan-concept of a 'dashboard' and what we actually get. In a vehicle, a dashboard displays crucial operator-controllable gauges and alarms to aid in the operation of said vehicle. This includes fuel status, speed, engine tachometer, temperature, and a cluster of lights to indicate subsystem status. Comparatively, a dashboard for a piece of software like Kafka can include over a dozen line graphs, several large counters, and a handful of alarms.

The key distinction where these models break down is that everything on a vehicle dashboard is designed to impart statuses that the operator is fully in control over, or ones that they require in order to shift their operation in response to. For example, the speedometer is something you can control as a driver, the tachometer is critical if you're driving a manual transmission, and the status of subsystems such as traction control allows you to make an educated decision on how to control the vehicle in response to external, uncontrollable events (such as the weather or road condition). Kafka[^dashboardOther] dashboards, on the other hand, are completely disconnected from the transactions that they are being used for. As a shared resource, the dashboard for Kafka will invariably intermingle data from multiple transactions, making it difficult to pinpoint failures or give actionable feedback to operators on *why* things are happening.

> ***Design and Dashboards***
>
> *Another crucial distinction between vehicular (plane, automobile, etc.) dashboards and software ones is that the vehicle dashboards are designed very specifically to be used while the vehicle is in motion. Items are displayed using distinct pictograms, with consistent color coding and placement (such as using yellow or red to impart meaning -- 'check engine' is a yellow alarm, 'parking brake' is a red one), and very little continuous wear information. A vehicle's TPMS (Tire Pressure Monitoring System) is usually in a sub-menu or on an infotainment display, for example. Explicitly, these dashboards are oriented around what you need to know to operate the vehicle safely at any given moment. This fundamental design distinction separates them from software dashboards, which often intermingle continuous health or state reporting with active status indicators, forcing operators to familiarize themselves with "what matters in an emergency" vs. "what matters when things sound weird". There's a difference between knowing if something is broken, and figuring out why it's broken, and our dashboards usually don't support this.*

This fundamental mismatch isn't something that can necessarily be corrected by 'better' dashboards, it requires rethinking what we actually use them for, and what we should be looking at. Currently, software dashboards look a lot like the internal diagnostics display for a vehicle -- crammed full of hundreds or thousands of data points that are both difficult to interpret without expertise *and* lacking the context and high-level guidance required for operators to quickly and accurately identify the source of system failures.

Cloud-native observability would suggest that in lieu of these detailed, diagnostic dashboards, our primary dashboard becomes one that's full of Service Level Objectives. These SLOs provide useful and actionable feedback to operators on *what's happening*, based on the actual business goals that our software is trying to achieve. With dashboards of SLOs, we can immediately understand the actual health of our transactions and understand how service health is impacting wider objectives. These dashboards make it immediately obvious as to the state of a system, not to mention letting us know how much room we have for error.

Effective Alerting

"But wait, how do I know when something broke?" you might be asking yourself, after reading the prior section. Alerts are, traditionally, how we are informed of system state interruptions that require human

intervention. The lived reality of most engineers is rather different -- alerts are set up at some point based on recommendations or projections and then they sit stagnant until they annoy you enough to modify them. This isn't to say alerts are useful or useless, they're important -- but the amount of time we spend on grooming alert thresholds is, frankly, ridiculous. If you don't do it, it's probably because someone else is -- the invisible labor of 'DevOps'.

It follows from our earlier discussion that the most important thing to alert on are SLO violations and status. What alerts should we focus on, outside of those? Not a lot, actually -- alerts should be weighed against a three-part test, and discarded if they don't stack up.

Alerts should be:

- Actionable (I can make a change to resolve this alert)
- Specific (This alert is tied to a logical resource or transaction, or aggregation thereof)
- Permanent (Without intervention, this state will not improve)

Non-actionable alerts are the bane of on-call rotations (especially transient, non-actionable alerts). Being paged because something's broken that you can't fix is an exercise in frustration! If the alert is tied to a resource or transaction out of your control that does result in service impact, then that status is better off tied to your SLO targets. Non-specific alerts are ones that are too broad; They don't give you enough information to make a determination about what changed or needs to be changed to resolve them. Non-permanent (or transient) alerts are the type that resolve themselves without intervention; Containers crashing (but not crash-looping), intermittent API failures, things like that.

If your alerts don't pass all three tests, then you should reconsider them. Often, a 'bad' alert can be made 'good' through better targeting/scoping, but you should also be fairly liberal with turning off or muting alerts[^mutingAlerts] and seeing how things go.

Fantastic SLOs and Where To Find Them

Both of the above sections are written with the implicit and explicit idea that you're using SLOs as the primary method of understanding reliability. There's a reason for this -- they're an incredibly powerful and effective tool for not only quantifying reliability, but also for communicating that reliability to others. SLOs encapsulate another important property, and that's an acceptance of errors. As discussed in the introduction to this chapter, traditional monitoring approaches view errors as an exceptional case to be avoided -- this outlook doesn't reflect reality. Even if you assume your system to be perfect and free of bugs, there's things you can't control which can introduce errors through emergent behavior or unexpected interactions. SLO-based monitoring frees us up to treat errors as room for experimentation and improvement.

SLOs are a complex topic, with more nuance and detail than have been levied here. They can appear as somewhat scary, I think -- you'll start to run into a lot of fancy-looking math which can be a turn-off. If you'd like a much more in-depth discussion, Alex Hidalgo's book is a rather comprehensive introduction and analysis of the topic. That said, we can summarize the benefits of SLOs here, and how they fit into the bigger picture of cloud-native observability.

A good SLO means something, and it's a way to distill multiple functional aspects of a software system into a convenient shorthand. The reason you should be monitoring SLOs is because the SLO ties *meaningful user experiences* back to your team. What's another way to phrase a 'meaningful user experience'? It's a transaction, of course! This is why SLOs map so well to observability, because the thing we actually care about measuring is the same between them. A SLO should measure the aggregate health of a particular type of transaction and give you at-a-glance knowledge about how that transaction is operating in

production. This dramatically simplifies your monitoring workflow, as you're freed up to focus on what matters instead of slapping down flapping alerts or staring at stale dashboards.

Even better, since SLOs tie these transactions to business goals and objectives, they're powerful tools to democratize reliability across an organization and its customers. Incidents, especially a glut of them, tend to shake confidence in products -- even amongst coworkers. Broadly communicated SLOs help people outside of engineering not only understand the actual performance and reliability of your service, but they can help build confidence in products by demonstrating how the software relates to specific business objectives. As the business grows and changes, so too do your SLOs, always evolving to meet new challenges and opportunities.

For more reading on SLOs, see Effective SLOs

[^dashboardOther]: Or 'Kubernetes', or any other sufficiently complex resource.

[^mutingAlerts]: This presupposes that you have defined SLOs for your services. Please don't actually start disabling production alerts arbitrarily.

Effective Investigation

There's a common aphorism, 'the only constant in life is change'. This is especially true of a software system. The magnitude and frequency of these changes differ based on a variety of factors both internal (team size, composition, development methodology) and external (user base, seasons, internet connectivity interruptions) but they all can have an impact. Identifying, understanding, and mitigating the adverse impacts of these changes is in many ways the 'primary use case' of observability tools.

"That seems obvious", you might say, but you wouldn't know it based on most existing monitoring and observability tools. The overwhelming majority of existing tools are focused around monitoring and dashboards, with analytical features being difficult to use or of little value. Why is this the case? Three reasons come immediately to mind; The difficulty of building good general-purpose analytical workflows, inconsistent data quality preventing intelligent analysis workflows, and a lack of equity in telemetry data access or interest amongst engineering staff.

This is why our observability tools all kinda look the same if you turn your head and squint; The market forces optimization around 'table stakes' features such as query builders and visualizations, creating a dilemma for new entrants -- buyers don't want different unless you can prove that different will save them money in the short term. Users are also trapped by this, as existing tooling requires a great deal of intuition and 'muscle memory' gained through repeated use. Change becomes scary, as it asks you to throw away that learned expertise and control. This fear doesn't just stagnate innovation in products, but the entire toolchain; Innovating around new query languages, workflows, and signals has become leashed to short-term financial incentives to the benefit of nobody.

Cloud-native observability asks us to rethink this and to separate out investigation from monitoring, to really consider it as a standalone function. Investigating change should be about more than being able to bang out some really good queries or time-window correlations between different telemetry types. It calls for general-purpose analytical workflows that let us compare and contrast system state across transactions

and resources at different times, in different conditions, and at infinite layers of granularity in order to quickly identify what changed, why it changed, and how it's impacting reliability and performance.

Context -- What Ties Everything Together

The most challenging thing in any post-hoc (or real-time) investigation of an incident is establishing context. We need to know the status of tens, hundreds, or thousands of dependent and independent variables in order to accurately reconstruct a mental model or reproducible sample of our failure. This is the internal context (or state) of our transactions and resources. More difficult to capture, but also important to understand, is the *external* context of our system which can be altered by everything from emergent user behavior, penetration testers exercising underutilized code paths, multiple teams making changes to shared resources or data structures, and much more.

This presents several challenges in not only capturing a sufficiently detailed snapshot of system state and storing that snapshot for post-hoc analysis, but also in analyzing them in order to discover the specific changes that lead to an incident. Conventionally, this has meant making decisions about several tradeoffs between snapshot breadth, depth, and fidelity.

This shouldn't be as hard as we've made it. Transactions contain within themselves a sufficient picture of their internal state to aid in hypothesis generation and disqualification, linked together by a shared request context. Additionally, point-in-time or periodic measurement of a resource's instrumentation carry enough data to allow for effective correlation between a specific (or aggregate) transaction and the underlying resource state. Conveniently enough, capturing this internal context allows us to reasonably disambiguate and filter out external contexts that don't matter at the moment -- and in many cases, to quantify those external events so that we can build an understanding of states that tend towards failure.

What's been missing here is a *lingua franca* for associating transactional and resource telemetry together. OpenTelemetry solves this through it's context and baggage concepts; Combined, they're a generalizable way to propagate transaction-specific information to other processes or services involved in a transaction. In concert with other OpenTelemetry tooling, this context and baggage can be associated with resource telemetry such as logs and metrics, all of which can then be emitted to an analysis tool capable of understanding it.

This contextual information opens up a new world. Rather than relying on manual association through pre-shared tag values or simple time-window correlation, shared context means that tools can build more useful and impactful visualizations and analytical features. Rather than just looking at a single transaction with an error, we can instead look at a graph of all services involved in a transaction and compare those graphs against each other. Resource consumption and limits can be overlaid on these graphs, quickly giving us visual feedback on resource exhaustion or oversubscription, unusual usage patterns, and so forth.

With enough time, these tools can mature and begin to use machine learning technologies to dynamically perform these comparisons and issue suggestions or commands to a system, such as modifying throttling for specific clients to ease pressure on a congested API that's approaching a SLO violation.

Guided Analysis vs. Data Exploration

While the future is full of possibility, there's decisions we have to make today in this landscape. The fundamental tension in observability tooling can be broadly summarized as "do you want a really good query/visualization tool, or do you want something else?" It's not hard to understand why we often opt for the former -- there's a strong sense of familiarity and control when presented with this model of data exploration.

However, this model is insufficient to satisfy the needs of a cloud-native observability program. Even extremely powerful and intuitive query builders and data visualizers require a lot of buy-in from operators and developers. What happens quite often is a bimodal distribution of usage, where people are simply reading charts that someone else built or they're heavily building and caring for those charts for others. This sucks for everyone -- it silos knowledge into the hands of the observability sherpas, who quickly become single points of failure when they're unavailable or after they leave a team; It stunts growth and velocity as newer or more junior team members are unable to self-serve to answer questions.

We posit that data exploration is necessary but insufficient to satisfy the needs of cloud-native observability. It isn't an "either-or", but the primary way you interact with an observability tool shouldn't necessarily be a query editor! It's certainly familiar, and it certainly has its place, but it sets us up to think in a certain way -- we see there's a problem, and we start digging. In lieu of this, we propose other 'observability primitives' such as service graphs, aggregate trace comparisons, and histograms or heat maps be where we start, then drilling down and winnowing out possibilities from there.

The advantage of this winnowing process is that we don't have to rely on other people so much -- people that might be half a world away, or asleep, or busy with any number of other things. If we present users with rich visualizations that present service health and status via their relationships (i.e., seeing a service graph with an indicator about the volume of traffic or errors that are being reported as part of a transaction, or a diff tool that shows us the change in two bundles of traces) then we free ourselves from having to know what we're interested in finding out before we actually can start figuring things out. This democratizes problem-solving, as we don't need to be an expert to see where things are wrong, we can quickly see if it's a problem with our service or someone else's and go from there -- usually getting enough information in the process to find the right person or team and tell them about it!

Tagging and Cardinality

So if it's that simple, why don't we do it? Good question. As stated throughout this document, data quality is one of the biggest things holding us back; The sort of hand-held analysis mentioned earlier is extremely difficult to achieve without significant amounts of instrumentation written in a standardized way, embedded throughout a complex application from its earliest days and maintained over the course of months or years.

There's hope here, in the form of OpenTelemetry -- its semantic conventions promise to make it easy to create standardized, high-quality telemetry data. In a more general sense, however, we need to approach this question in a more holistic fashion. What is both necessary and sufficient for the systems we want to build and run?

To address this, we need to talk about cardinality and tagging. There's a lot you can read about cardinality[^cardinalityDef] online, but to briefly review what it is, it's the amount of unique values associated with a single attribute on some piece of telemetry data like a log, metric, or span. High cardinality is the bane of observability systems, as it either makes things very slow (such as searching for a single unique value in a set of millions or billions), very expensive (such as metrics 'cardinality explosions' where high cardinality attributes can cause 10x, 100x, 1000x, or greater combinations of time-series to be created), very expensive (such as the cost for indexes on high-cardinality structured logs), and usually all of the above. It's an unavoidable complication. It's also a requirement for observability tools, so this tension needs to be relieved somehow.

We'll talk more about it in our section on telemetry ROI, but to the end user, cardinality shouldn't really be a concern. What we can do, however, is blunt its impact through improved tooling throughout the observability pipeline. Traces, logs, and metrics should all contain high-cardinality data, but those situations shouldn't ever cause significant impacts to your observability system, as it should be able to dynamically

adjust the rate of collection and intelligently pre-aggregate and bucket data to avoid cardinality explosions hitting your actual system and causing failures there.

As we stated earlier, the cornerstone of effective investigation is that you can capture as complete of a snapshot of system state as possible, to make it feasible to perform post-hoc analysis of everything that was going on at the time of a failure or fault. There's still some missing pieces to this formula, though -- controlling costs, for one. That said, the key takeaway is that performing an effective investigation is about a lot more than nice visualizations and a great query experience, it's about workflows that guide you up and down the relationship tree between your transactions and resources, giving you the ability to quickly prove (or disprove) hypotheses and answer questions about what caused a given change.

[^cardinalityDef]: A discussion of cardinality can be found here

Effective SLOs and You

There's a lot of ways to communicate reliability, but the SLO is the way that 'fits' into cloud-native observability. An artifact of the SRE Book, SLOs are the simple and sane way to quantify things that are usually extremely difficult to quantify. That doesn't mean they're *easy*, but they're essential because they firmly orient everyone on your team around end-user experience, regardless of who that end-user is.

What makes an effective SLO, though? Like, a *really* effective SLO? The association of SLOs with engineering teams and SRE practitioners has, possibly unsurprisingly, made them feel somewhat unapproachable to people outside those spheres. In this section, we'll discuss not how to make an SLO, but how to use them as bridges between different parts of a business and use them to drive organizational change by reevaluating how we think about incidents and incident reporting.

Counting the Uncountable

The concept of 'data-driven' organizations has become rather en vogue over the past years. Driven in no small part by a bunch of businesses selling you data shovels, we're encouraged to collect and quantify increasingly fine-grained data about everything that happens in the course of doing business. Contacts with end-users, logins and logouts, session lengths, what links people click in email, who downloads reports or responds to targeted advertising, how often someone waits before asking for help, etc. This data obsession has also turned inwards, as companies track ever finer details of their development, marketing, production, staffing, and every other business process you can think of. This data all gets sucked up into a variety of databases and spit back out into reports, digested by managers and executives, and finally translated into broad pronouncements about the health of the business or a particular initiative.

Tayloristic management practices were said to have fallen out of style, but it's hard to really tell when you look at KPIs, OKRs, and the myriad of goal-setting or objective-tracking systems that have become commonplace in organizations. In fairness, there's a legitimate need to understand what's going on in the workplace, in order to better allocate and assign resources (like money) and people (like... people) to projects and initiatives.

Let's compare that, then, to how we report on reliability to the end-users of a service. It looks a lot like this:

Service Name	Status
Foobar	Up
Baz	Down

There's a pretty big disconnect between the level of granularity we expect out of goals that, frankly, are ill-defined and unquantifiable (such as "Ensure we're the best place to work!") and systems that have an absolute wealth of fine-grained performance and reliability data to work with.[^notAllDashboards]

Why does this disconnect exist? It's usually due to cultural reasons. There's an oft-repeated, possibly apocryphal, anecdote that marking certain AWS services as 'down' on a public dashboard requires VP-level approval. Not all incidents are created equal, after all, and if nobody was impacted (that you know of) by an issue, well, did it really ever happen?

SLOs avoid this trap by being directly tied to specific end-user experiences and communicating *expected* availability ahead of time. This is something that everyone should understand; KPIs, OKRs, these all have built-in 'pressure relief' valves that state "hey this is the goal, but we don't expect it to be 100%" -- OKRs, famously, are suggested that you target 30%+ over a 'normal' result in order to push your team. SLOs understand that binary success or failure isn't a useful way to communicate reliability.

Pushing The Envelope

Cloud-native observability posits that SLOs are woefully under-adopted and underappreciated in software organizations. Much of what you 'do' can be expressed as an SLO, and those SLOs (or aggregations of a group of them) should be the primary way reliability is communicated outward from engineering teams to the rest of the organization, and to your end-users directly.

Expressing reliability via SLOs as the default is rather revolutionary from a mindset perspective. You quickly realize that you were defaulting to a binary model of 'up' or 'down', because that's the only real language you had to communicate performance. SLOs break this, as they introduce not only the concept of safe failure, but also *acceptable* failure. This acceptable failure is where you can improve and iterate -- pushing new features, or cleaning up technical debt. It's also an affirmative signal to the rest of the organization that this kind of acceptable failure is expected, and to think about failure not as a terminal state, but as something that can be managed through process, prioritization, and effective coping strategies.

That's not just hyperbole, either -- "move fast, break things" is a hyperbolic expression of this! We accept a certain amount of failure, we even encourage it in our planning and iteration processes, everywhere except our software. If we can move towards an SLO-based understanding of reliability, then we can apply the same level of psychological safety that these mantras try to instill to people's feelings about our software and products themselves.

SLOs aren't just a way to keep track of how well you're meeting end user needs, they're a radically transparent way to set boundaries and expectations about the reliability of your software, your team, and your organization as a whole.

[^notAllDashboards]: It should be noted that there are organizations that communicate more fine-grained reliability data (such as availability over trailing 30 days) through their public status pages. We salute them.

Telemetry ROI -- The Elephant In The Room

If everything in the prior seven chapters has sounded like a beautiful crystalline pipe dream, this is where it all comes crashing down to earth. The explosive cost of telemetry and observability tooling, storage, and analysis is staggering. Organizations spend hundreds of millions of dollars a year (at the high end) on collecting and storing telemetry data, much of which is never even seen by a human being nor influences a decision.

The status quo is harmful to observability practice, practitioners, and end-users alike. Curiously, it's also something we don't talk about much. While there's a generalized acceptance that we pay too much for too little, there's also a generalized malaise that this is simply the way the world works and we must accept it.

In this section, we'll dive into a discussion about why telemetry costs keep rising, and what to do about it. We'll also discuss the tradeoffs between different cost management strategies, and where they should be applied.

Costs Go Up, ROI Goes Down -- You Can Explain That

You only have to look at financial reports from major monitoring vendors to see that the people selling shovels are doing pretty well indeed. Organizations pay more than ever to collect even greater amounts of telemetry data each year. If this cost was linked to a linear scaling function -- as in, you pay more because you have more users or more traffic -- then it would at least make sense. However, costs tend to increase even if usage is flat. Why is that the case?

Increases in system complexity throw a wrench into this logic, unfortunately. The product of new orchestration systems or managed services, new feature work, and sustaining engineering efforts causes a combinatorial explosion in the amount of telemetry emitted by a system. This telemetry tends to bias towards resources rather than transactions, as new foundational work (migrations to Kubernetes or the cloud in general) can emit significantly larger tranches of new telemetry data that requires more care and feeding -- more dashboards, more point solutions for monitoring or understanding specific types of cloud systems, and so forth.

This leaves developers in a bit of a lurch. The telemetry they need in order to understand and diagnose end-user issues tends to be high cardinality and multi-variate, but the monitoring budget is being doubly-spent on things like cloud metrics being fed into a different (or even multiple different) monitoring systems. We've all got stories about not being able to add a new attribute to a metric, because we're arbitrarily limited on the amount of custom attributes for contractual purposes or because we can't afford to scale up Prometheus to handle the potential cardinality.

What we're left with, then, is a curious condition. We're spending more than ever, but getting less value from every dollar we spend. It's not just the bias towards resource telemetry that causes this, it's everything we've touched on throughout each chapter. The implementation wall of improving data quality, the maintenance burden of creating and maintaining dashboards and queries, the morale and communication overhead of inconsistent or unclear alerts and reliability metrics, etcetera and so forth.

Compounding this problem is that we use data volume as a proxy for data quality. As discussed in the SLO chapter, modern organizations collect and capture unthinkable volumes of data about their processes and systems for analysis and decision-making. Different stakeholders each have their own telemetry desires -- security teams demand access and flow logging, analysts demand usage and journey analytics, SREs need high-fidelity resource and transaction telemetry. The data stores and tools each of these roles require are often disparate and nonintegrated, leading to a need to either warehouse vast quantities of data or

duplicate it to various endpoints. Managing this stream requires no small investment in time and money, and often overlays complex bureaucracy into what should be trivial asks.

In summary, we've arrived at a situation where we're generating more telemetry from more places than ever before, sending it to more destinations with more advanced features than we've previously had access to, but all of this has failed to appreciably improve outcomes. We're spending more money, and time, to eke out marginal reliability gains while suffering from a *perception* that software is less reliable than ever. Something's gotta give.

The Predictive Value of Telemetry

The key to changing this negative feedback loop is to rethink the value proposition of system telemetry. Rather than viewing telemetry as the input to primarily reactive processes (failure recovery, etc.) we should model it as a *proactive* input into predictive processes.

A predictive model of telemetry seeks to thoroughly instrument and collect telemetry from our systems in order to generate forward-looking insights into the reliability of not only our system itself, but the organizational support for iterating on that system. As mentioned back in the introductory chapters, we don't just want to be able to build fast, we *need* to be able to build fast. Rapid iteration isn't just smart, it's safe. Lengthy verification cycles in pre-production systems don't necessarily translate to higher quality features or less disruptive deployments due to the inherent drift between production and non-production environments. Observability can change this balance, as it allows us the ability to safely monitor continuous deployments into production and roll back if things aren't working out like we hope. Telemetry is the input into this prediction algorithm -- and calling it an algorithm is really just being fancy, this is basically describing a canary rollout. Traditional canaries look at limited amounts of telemetry, though -- simple "is it up? is it fast?" questions. Judicious use of metadata can give us more fine-grained options for canary-ing, allowing us to see if we're actually improving our SLO adherence or achieving specific business goals as part of a rollout.

Predictive telemetry also helps in capacity planning, not just for resources, but also for teams and organizations. Being able to predict future usage by analyzing historical patterns is one example of this, but by unifying our telemetry and our business data, we're able to improve our ability to forecast on-call rotations, hiring, and even the cadence of feature work. This may come as a non-intuitive realization, as "slow is smooth, smooth is fast" is an oft-repeated canard, but this is a tactical vs. strategic distinction. Slower deployments and organizational agility introduces a higher likelihood of bugs because not only does the underlying technical system shift out from under you, so too does the *mental model* of practitioners drift from reality. Being able to realize work in more granular chunks is the difference between plucking out a single Jenga brick from the tower versus grabbing a handful. As stated above, it's not just about technical insights. Democratizing our telemetry allows for tighter feedback loops across the entire organization.

Cost Reduction Strategies

Even if we wish to use our telemetry data in a more predictive, rather than reactive, fashion, we still need to control how much we send and spend on it. Traditional methods of managing this spend tend to be proactive -- for example, reducing the amount of telemetry generated by a given process, or not adding additional attributes due to cost concerns.

This is, again, an anti-pattern. Telemetry generation at the process level does have an overhead, but it makes a lot more sense to have this overhead be relatively fixed rather than dynamic. That is to say, if you know that every transaction will have a given amount of telemetry generated for it, then you can plan for the amount of resources required to support that overhead.

This isn't necessarily something that has a one-size-fits-all strategy to overcome. There's a world of difference between the amount of telemetry generated by planet-scale applications (such as Google Search) and even large enterprise applications (such as a large financial firm's electronic banking app). Different systems will have different requirements. How do we figure out what we need to keep and what we need to discard -- and how do we discard it, anyway?

SLOs and Deep Telemetry

Determining how much data to generate and keep should be a derived function of the SLOs we set for a given transaction. This does mean that telemetry data requirements can fluctuate over time; After all, SLOs aren't set in stone, they can change and grow. That said, the amount of telemetry we need should always be in support of measurable semantic structures like an SLO. Don't collect data 'just because' -- if you ain't using it, you don't need to keep it! Identify the critical transactional and resource telemetry required to achieve SLO monitoring and investigative workflows, then discard the rest.

That said, what do we do when things get weird? There's always going to be circumstances where the amount of data you need to solve a problem is greater than the amount of data you keep on-hand. This is where 'deep' telemetry comes in. We use this to refer to any number of process or system level dumps of telemetry data that wouldn't normally be collected; Think `pprof` profiles, kernel dumps, or network flow logs. The role of your observability system should be to make transitioning across the boundary (from steady-state collection of high-level diagnostic information to low-level system information) painless. The balance we should strive for is "just enough" information in our usual telemetry to solve 80% of our problems (or more), with enough data to help us narrow our search space for that last 20%.

Sampling

Even if we know what data we need, and how to generate it, it's foolish to suggest that we collect and keep all of it. Systems that are in a steady state don't really have anything interesting to say in detail. If everything is good, or everything is bad, then you're probably not getting any useful insights from detailed telemetry.

Most of the time, though, your system will be somewhere between these extremes. This is why sampling is useful -- it's a way to ensure a representative amount of telemetry is collected about your system, smoothed out over peaks and valleys of demand. Traditional sampling methodologies rely heavily on "head-based" or client-based sampling rates, where originators of transactions define a given ratio of transactions to be traced or logged (such as 1-in-10, 1-in-100, etc.) and only recording transactions that meet that demand. This approach leaves something to be desired, as outliers or deep errors can be easily missed. Further developments have introduced heuristic-based approaches, that override sampling rates based on indicators of state, such as prioritizing collection of errors or other faults in the final sample.

In general, cloud-native observability places more value on dynamic sampling approaches rather than static ones. The key difference is where this sampling is configured, and how it's adjusted. Traditional methods of dynamic sampling require bidirectional communication between telemetry clients to adjust head-based sampling rates based on other factors such as operator intent or independent signals, or by cache-and-forward approaches where a pool of collector processes store transactional telemetry in its entirety to make a decision about a group.

While these approaches have value, there's still a lot of confusion and further work needed. Static sampling is too proactive, dynamic sampling can be too reactive. The key innovation required is moving sampling 'up' a layer, from generation or collection to ingestion through observability pipelines. If sampling decisions become less about 'what's generated' and more about 'what's persisted', then these decisions can be made in the full context of not only a transaction, but also overall resource state. This means the knobs and levers for controlling sampling will need to move as well, away from low-level configuration in code, and towards management planes and analysis tools that can control and report on the current level of sampling at a

process, cluster, or system level. Much of the work that exists in this field is highly domain-specific and has only been implemented as part of platform engineering efforts at large software companies.

Aggregation

The other major lever to control steady-state telemetry costs is the aggregation of telemetry data. Sampling controls the flow of data into our observability pipeline, and thus our egress costs -- Aggregation is our lever to control long-term storage costs. For the most part, solutions here are rather non-controversial, but many of them aren't in wide use.

Metrics, logs, and traces all benefit from aggregation at storage and query-time, but the method of aggregation differs slightly between them. Rather than dwell on the strategies, we'll summarize them here and discuss how they impact cost controls. Metric aggregation occurs at both ingest and query time, as points are smoothed into regular buckets. Trace aggregation tends to occur mostly during ingest, as attributes and metadata are converted into frequencies and de-duplicated. Log aggregation is another ingest and query-time concern, as messages are de-duplicated, with variable fields clipped out and stored separately.

All of these existing approaches are useful, but there's some nuances we should be aware of. What benefits cloud-native observability is a shared set of aggregation formats and visualizations that allow us to use these disparate data types in similar ways, and to not re-aggregate data that we don't need to. Observability platforms that are aware of multiple forms of telemetry can, in response to operator requests or heuristic-based algorithms, dynamically adjust aggregation and persistence rates and methodologies based on desired recall times, storage constraints, or numerous other factors.

Intelligent Sampling in Example

Our discussion of ROI has been long on theory and, perhaps, short on solutions. To ameliorate this, we present a proposed model and architecture for dynamic sampling and aggregation. This model makes the following assumptions:

- Telemetry data that has been used in the past will continue to be useful in the future.
- Certain types of signals are more valuable than others in certain situations, and we can teach a computer what those situations are.
- The true value of any signal or instrument can only be discovered through refinement.

With these assumptions in mind, how would we implement a system to achieve them? Let's take our first example, a naive sampler that retains signals based on how often they're queried. A potential implementation of this sampler follows:

```
stateDiagram-v2
    Service --> Signal
    Signal --> Collector
    Sampler --> LRUCache
    topKSearches --> LRUCache
    LRUCache --> Sampler
    Collector --> Sampler
    Sampler --> Output
```

As signals are batched and processed by our collector, sampler configuration is dynamically updated by requesting the most frequent queries from our query engine and using those to fill a least-recently-used cache. Signals with attributes that are queried more often will be preserved, ones that are queried less often or never are deleted.

Further refinements to this process could result in the application of ML to this feedback loop, allowing for new telemetry attributes and signal 'shape' to be compared to operator feedback about signals that "worked" in investigative workflows, and training our sampler off that. This requires a rather deep integration between our investigative workflows and our observability pipeline itself.

Organizational Concerns of Observability

The previous chapters have more or less spelled out a vision of a new form of observability; One that is built around new primitives, like SLOs, and supported by centralized, multi-telemetry-aware platforms, built on open source observability libraries. We understand this vision might be a hard sell -- after all, it doesn't really exist in one place, anywhere.

This chapter will address a mixed bag of institutional concerns about the feasibility of this vision, objections raised by orthodox practitioners, and the requirements for successful implementation of cloud-native observability.

Combating the 'Three Pillars' ideology

Objects in motion tend to stay in motion, this much is known. The momentum of traditional 'three-pillars' monitoring is difficult to understate. An immense, incalculable amount of development time and money have gone into platforms that operate based on this ideology. Existing systems, instrumented for discrete metrics/tracing/logging experiences, are loathe to change without provable, exponential improvements in experience.

We argue that although change is hard, it is necessary, and it is also inevitable. Three pillars isn't a hard and fast law of the universe, it's a historical accident spurred by the existence of monitoring vendors. Many of these vendors only rose to preeminence due to a lack of first-party tooling for monitoring cloud workloads! A fish might rot from the head, but ideology is reinforced from the bottoms-up, and the primary way three pillars approaches are institutionalized is through telemetry generation. If you rely on a proprietary agent to create telemetry and send it off somewhere, you're going to get the telemetry that is given to you. OpenTelemetry combats this directly, by recognizing the heavily commodified nature of these agents, and moving telemetry generation away from them. It favors direct integration with libraries, managed services, orchestration platforms, and so forth.

As OpenTelemetry becomes a widely adopted standard through usage in cloud-native tools and platforms, the newly found freedom of developers to port their data to other analytical tools will act as a forcing function on the industry to modify their products. This should lead to a period of intense competition and consolidation, as new entrants and incumbents alike are no longer 'safe' due to vendor lock-in.

Build, Buy, and Everything Between

In this brave new OpenTelemetry world, should you buy or build? The answer depends on several factors. If we keep in mind that our telemetry data will be highly portable, then we see the formation of three general camps:

Pure Builders: Highly specialized applications of telemetry will always require highly specialized tooling that general-purpose tools won't provide. This group will leverage customized, off-the-shelf databases with tailor-made analysis tools in order to solve challenging, domain-specific problems.

Mixed Use: A larger group will choose to leverage one, or more, point observability solutions in conjunction with managed long-term storage in order to achieve a 'best of both worlds' approach. Different teams with different needs may elect to use different paid or free tools, drawing down from a centralized telemetry repository.

Pure Buyers: Possibly as large as the mixed-use group, these organizations will leverage paid solutions for their overall telemetry needs. Generally, they'll either be very small organizations that need to outsource expertise to a vendor, or extremely large ones that prefer OpEx over CapEx.

Your organization's adherence to one of these camps should not be thought of as static. As your organization and observability practice matures, it would not be surprising to see pure buyers become mixed use, or to have platform teams embedded within them that act as pure builders. Taking a longer view, it's likely that the product landscape for observability will open up as well and you'll see more vendors offering point solutions to specific problems that mixed use or builders will encounter. While this behavior does exist today -- products such as Cribl or Edge Delta exist as pipeline middleware, for the most part -- I believe it's an emerging part of the observability marketplace.

Organizing Effective Observability Through Centralization

Successful adoption of observability requires some level of ownership in the organization. Implementing standards for telemetry quality, determining spend thresholds and optimizing costs, and managing access and control of potentially sensitive data are all better served through a centralized owner rather than letting everyone fend for themselves.

It may be helpful to think of an observability team as a federalized role; Making rules and guidelines and providing cross-cutting infrastructure that provides a minimum guarantee for everyone else, rather than trying to embed in a variety of disparate teams and dictate terms. Centralized teams can ensure that tooling and processes exist to give service owners easy integration into the observability stack, guarantees around data quality, dashboards and SLOs to plug into, and overall visibility into the greater system. In an existing organization, the observability team can perform the crucial act of creating uniform metadata mapping and standardization, as they will have the necessary visibility into everyone else's systems. Additionally, this standards-making activity is a natural starting point for them to discover pain points being experienced by the organization, informing what their next project(s) should be.

In some organizations, this might be in the office of a CIO or CTO -- in others, a cloud center of excellence. Regardless of where it sits, there is a lot of value in having executive sponsorship for observability initiatives, as the most successful ones tend to be mandated from the top-down. Larger organizations, with more rigid service boundaries, require a lot more cross-cutting work to do the heavy lifting of getting an observability framework mandated and implemented in enough foundational resources to provide basic transaction-level visibility.

Glossary of Terms

This page is intended to be a quick reference to terms that we use throughout this project.

Distributed System: A system that's characterized by its components being split up across multiple distinct computers on a network.

Distributed Tracing: The technique of following – or "tracing" – requests as they traverse distributed systems, representing them as telemetry, and reassembling and analyzing models of the distributed requests.

Instrument: A logical object that emits a specific piece of named telemetry. For example, an instrument might measure the current number of requests being handled by a process, or the amount of free disk space.

Instrumentation: Code that exists to emit telemetry signals to some external system or endpoint. For example, a logging statement or a metric instrument.

Logs: Structured, or unstructured, output from a process to aid in debugging or insight into a service or system. Usually optimized for human-readability.

Metrics: Statistical data about resources and transactions that is gathered periodically, typically stored in a time-series database, and used as part of a broader observability solution.

Monitoring: An aspect of observability that tracks the health of individual resources and components, relative to the health of the entire system.

Observability: The ability to both understand and reason about the behavior of a software system without needing to make **changes** to that software system. In practice, observability requires both high-quality telemetry and high-quality observability tooling.

Resource: A finite and durable unit of computation, storage, or other work that is typically shared across many distinct transactions. See Resources for more information.

Microservice: A service in a distributed system that generally handles a well defined and tightly scoped amount of work. An example would be an authorization and authentication service.

SLA: A Service Level Agreement, a contractual instrument between parties about the performance and availability of a service.

SLI: A Service Level Indicator, a statistic about a system that corresponds to the health of a transaction or resource.

SLO: A Service Level Objective, a statistic about a system that ties SLIs to business objectives and goals.

Structured Logs: A type of log message that comports to a schema, usually intended to be read and processed by a machine.

Telemetry: Data, either human- or machine-readable, that describes the internal state of software services or infrastructure.

Transaction: The input, output, and path of a single client's request in a cloud-native application, consuming resources along the way. See Transactions for more information.

Appendices and Further Reading

This page contains a variety of short appendices; If a given section becomes too large, we should consider expanding it into a full chapter.

Telemetry Resolution

In our discussion of telemetry signals, we focus on logs, metrics, and traces to the exclusion of other types. Other authors in this space will often add a fourth or fifth, such as 'profiles' or 'events'.

The reason we focus on these three is because they generally sit at the 'sweet spot' of resolution to make it easy enough to understand the overall state of transaction and resource health in aggregate, while preserving enough resolution to 'zoom in' to a single fault (or a group of like faults) with sufficient metadata to help operators understand why that fault is happening. This is roughly what you need in order to set and monitor SLOs, or to generate hypotheses for further debugging.

As a specific example, profiling is a useful way to inspect line or function-level detail about a given process. Profile dumps of service code in production can be invaluable in understanding memory usage, garbage collection, or logic errors. The ability for engineers to generate and collect these dumps is an important part of a resilient system. That said, it's not a pre-requisite for *observability*.

It's Only A Trace If It's From The Dapper Region Of Google, Otherwise It's Just Sparkling Logs With Context