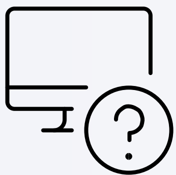
WHITE PAPER

Balancing cardinality and cost to optimize the observability platform



Observability, what is it good for?

Back in the day, monitoring was simple. Proprietary monitoring agents sent infrastructure-level metrics like CPU-usage and another agent sent logs to a syslog server to trawl for error messages.

However, as applications moved from virtual machine-based monoliths to cloud native microservice architectures running in the cloud, understanding the application and system state became much more difficult, requiring a more flexible approach to monitoring. Prometheus, the CNCF-backed open source metrics platform, has grown into the de-facto standard for monitoring cloud native environments, and we've moved from proprietary agents to open standards with many software components being pre-instrumented for quick and easy integration. Its popularity is, among other things, due to its inherent flexibility to explore data at will, arbitrarily querying the data to show different slices and views of data as teams investigate issues.

This ubiquity of metrics data allows teams to look at their applications and underlying systems without a fixed set of limited metrics. Instead teams are able to dive into any misbehaving part of a system, investigating complex emergent behaviors and the longer tail of failure modes in complex cloud native microservice architectures as they happen and solving problems caused by ephemeral combinations of corner-case conditions and bottlenecks.

You need a system that lets you arbitrarily look at different parts of the system in a much more dynamic, flexible, and ephemeral way than you viewed a production application. This requires more than just infrastructure-level metrics, or just logs, or just traces of user requests across the system. Legacy systems that provide them are not enough to be able to dive into those emergent behaviors and figure out what's going on in your application stack.

As companies adopt container-based and cloud native application architectures, observability becomes harder and harder to manage due to the exponential growth of metrics data, as not only the number of monitored services increases, but also because each service is instrumented too broadly enough to observe it adequately. Observability teams are scrambling to cope with this growth, while trying to continue to provide meaningful insights from the observability data to teams quickly and accurately.

The problem is, as the amount of metrics data being produced grows, the pressure on the observability platform grows, increasing cost and complexity to a point where the value of the platform diminishes. This is especially true with Prometheus, due to its inherent limitations to operate at scale. Simply put, Prometheus can only handle so much data before it starts to break down or forces you to stand up a second (or third, fourth… you get the idea) instance, preventing teams from gaining visibility and insights from metrics data.

So how do observability teams take control over the growth of the platform's cost and complexity, without dialing down the usefulness of the platform? In this whitepaper, we'll talk about optimizing the cost and value of your observability without compromising insights.

Observability's place in the landscape

But what are the insights derived from an observability platform? What value does the platform bring? Let's dive into the fundamentals of observability, first.

DevOps teams use feedback loops to understand the production application's operational status. Observability, in modern, cloud native software development scenarios, is the practice of being able to understand emergent behaviors of the application in production to increase reliability, security, and performance. Observability helps teams get real-time visibility into how live users interact with the production system and use that insight to develop fixes and improvements.

And this is important, because modern, cloud native applications rarely fail outright. Usually, a failure affects only a part of the application and cascades across dependencies in unpredictable ways, creating issues with performance and reliability in unique ways that will likely only happen once.

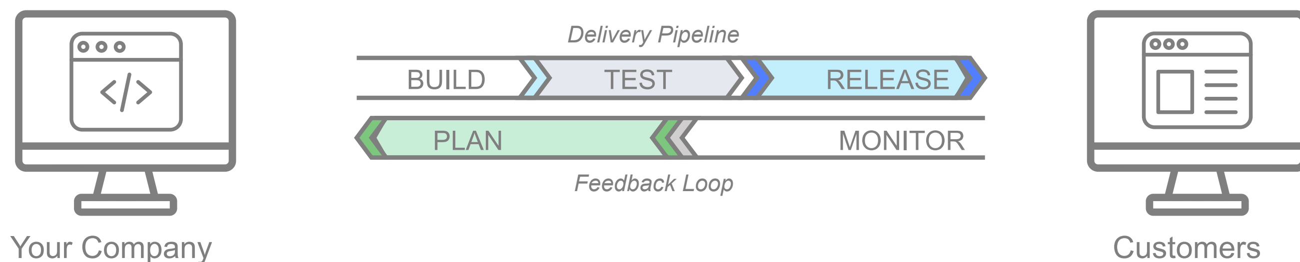


Image 1: the DevOps loop

The business value of observability is thus immediately clear: to provide clarity and insights into a system's state by monitoring signals from the application's execution, behavior, internal states, communication between services, and components. The goal is to optimize the application's business (and technical) performance, cost, reliability, security, and more. Without adequate observability of their application stacks, teams are flying blind and won't be able to respond to issues in real time, creating down-time, loss of revenue, or damage to reputation.

In microservices-based, container-based cloud native architectures, the amount of services working together to create a single user-facing application can be staggering, and the amount of dependencies are too much for any human being to fathom. This creates inherent complexities that observability platforms have to deal with, and have to unravel, to provide any value to its users. The amount of data produced by these dynamic, ephemeral, and ever-changing landscapes of services can be so large, it becomes nearly impossible to analyze and derive valuable insights from it. Simultaneously, these complexities result in a greater need to correlate and interconnect infrastructure, applications, and business metrics.

A common misconception of collecting telemetry is that 'more is better'. It is not.

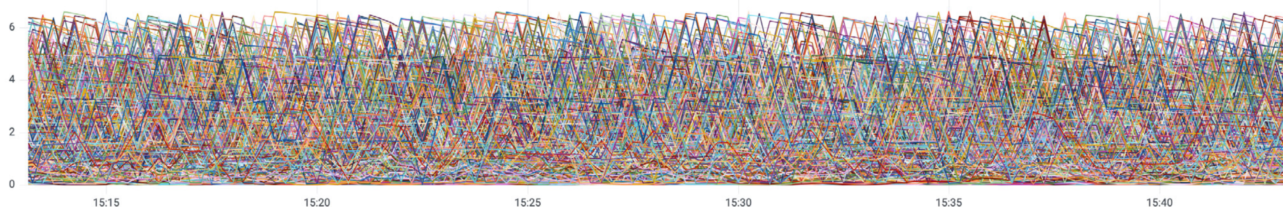


Image 2: Signal is lost in the noise

As image 2 shows: the signal is lost in the noise completely. For anyone triaging a problem, this amount of data does not meaningfully contribute to finding the root cause, and likely has the inverse effect. It evidently shows that while the underlying data is important, teams need the ability to separate signal from noise. By combining different types of data and correlating data, a pattern emerges that helps build a picture of the state of the system, and helps engineers to see a path towards resolution of the issue at hand.

While teams may take the path of least resistance and collect lots of data on their service, the team responsible for the observability platform has to balance the cost of collecting and storing an amount of data with the value of the insights derived from it, seeking the optimal balance between the two.

And given that this optimal balance varies between different services due to business value, scale, and other factors, the platform team has to account for these differences across services, and help teams responsible for each of the services find that balance. Different teams have different needs, with some wanting to monitor different aspects of their application or, requiring different levels of detail and insights.

Luckily, the platform team has a few knobs to turn up or down, the collective of which helps teams find the optimal balance — and find the answers to how they can control what metrics data our observability platform ingests, as well how they can limit the impact of querying the metrics data.

Cardinality, resolution, and granularity

Taking back control over the amount of data being collected is not an easy task. Teams running apps in production often err on the side of caution, collecting more data than they think they need, in case they do need it. Or they forget that they started collecting a specific metric or label in the past, and it's being stored, but not used. Or, they made false assumptions about how much data a metric would generate.

Often, these teams don't feel the pain of collecting too much data, as they're not the ones running the observability platform. However, they are the ones that feel the impact of too much telemetry, as the observability platform's performance trends downhill and either their queries slow down, or the ingestion of data slows down to a point where it takes too long for ingested data to be available for querying.

Either way, performance is key, and it's the observability team's responsibility to keep performance of the platform at adequate levels. This simply means they must balance the amount of data collected with the cost and value of that data to drive insights for those same app teams.

So what can teams do to limit the amount of data collected, without impacting their ability to answer questions about the state of their systems? In order for us to answer these questions, we must understand these key concepts:

Dimensions are the attributes or properties that you are collecting data on. Metrics can have an arbitrary number of dimensions: Customer ID, Shopping Basket Size, Transaction ID Pod, Region, Service Name, and other items of interest to your business. Essentially, these are the keys in key-value pairs. The smaller, or more granular, the dimension is, the more instances of that dimension we need to collect data on. For instance, do we measure memory usage on a cluster level (just one dimension), on each of the individual cluster nodes (one dimension per cluster node), or on each individual pod running on each individual cluster node (many dimensions per cluster node)?

For every key (also called labels and tags throughout this paper) that you add to the observability platform, the amount of data collected increases multiplicatively; each additional dimension increases the number of data by repeating the existing dimensions for each value of the new dimension.

Cardinality is the uniqueness of the values of the dimensions you're collecting data on. Having two dimensions of 10 values each (for instance, an HTTP error dimension with ten error codes) isn't much of an issue. Even using a single dimension with lots of possible values (like cluster name) isn't an issue. But having many dimensions, some with many possible values, quickly does become an issue.

It's the multiplicative effect of the total number of combinations of all the dimensions' values that defines high cardinality. In addition to cardinality within any given metric (defined by how many combinations of its dimensions we can make), the same concept applies to the total number of combinations across all metrics.

Resolution is the interval of the measurement; how often a measurement is taken. This is important, because a longer interval often smooths out peaks and troughs in measurements, so that they don't even show up in the data at all; time precision is an important aspect of catching transient and spiky behaviors.

Retention, finally, is how long high-precision measurements are kept before being aggregated and downsampled into longer term trend data. By summarizing and collating, resolution is reduced, trading off storage and performance with less-accurate data.

But why is cardinality specifically such a problem in this case? Well, it's due to the fact that most, if not all modern metrics systems use a time-series database, which stores a unique time series for each and every metric that you add. And for every possible value for each tag or label added to a metric, another series is stored to disk. This leads to an explosion of data collected for every tag or label added, often without anyone consciously knowing or realizing why. For many engineers, it can be hard to understand that adding a single dimension to a metric can result in millions of additional time series and datapoint writes. Higher cardinality is also more useful for troubleshooting and being able to understand emergent

behaviors of the application in production to increase reliability, security, and performance. We need higher cardinality for the monitoring system in order to be useful. Without that sufficient level of detail, teams can't get to the bottom of the issue they're investigating.

So if we're monitoring the consumption of potato chips over time, we keep tally of them every, say, 5 seconds. This produces one time series. However, if we want to also keep track of what flavor each chip is, we add a tag 'flavor'. This creates a new time series metric on disk for every one of our, say, fifteen flavors (natural, barbecue, sour cream & onions, and so on). Now imagine that each of these flavors can also be one of two styles (straight-cut and wavy) and three health options (oven-baked, lightly-salted, and regular). We're now storing not one, but 15 x 2 x 3 = 90 time series on disk. In a time-series dataset like this, the cardinality is defined by the cross-product of the cardinality of each key-value pair, multiplying them 'key times value'. The more dimensions added, the more granular we can slice and dice the data by looking at a single dimension, a combination of a few dimensions (like straight-cut lightly salted but all flavors) or all dimensions.

This multiplication effect grows stronger as we add more tags and labels, especially when the number of unique values for a dimension is large. And if we want to track averages or percentiles as well, the number multiplies, again. The right indexes will help us find the flamin' hot wavy oven-baked potato chips quickly. With multiple indexed columns, the number of unique values starts to matter – a lot. The cross product quickly becomes very large, very quickly, even if only one dimension has a high cardinality. This is why cardinality is an often-cited bottleneck for observability platforms, and one of the largest tradeoffs between the cost of an observability platform, and the insights it helps teams gather. And isn't the whole point of the observability platform to drive insights that are more valuable than the cost and effort put into the platform?

All of the mentioned key aspects of keeping an observability platform under control are sliding scales without a single, fixed optimal cut-off point. Each has an impact on the cost and complexity of the platform, and the precision of the insights derived from the data. Simply put: more dimensions, with higher cardinality and better resolution means deeper insights, but at a monetary and operational cost to manage the complexity and volume of data. In an ideal world, our observability platforms can cope effortlessly with the explosion of telemetry. In reality, teams are forced to sample, approximate, to estimate, and to choose which parts of their landscape to observe with a fine level of precision due to the immense cost and complexity associated with those high levels of precision. Without adequate levels of precision, however, decision-making is flawed and has a risk of leading to wrong conclusions due to imprecise data.

"Cardinality is an often-cited bottleneck for observability platforms, and one of the largest tradeoffs between the cost of an observability platform, and the insights it helps teams gather."

And with cloud native architectures, the amount of dimensions, and the cardinality of those dimensions, is a few orders of magnitude different than with monolithic, VM-based systems.

This simple (albeit overly simplistic) example shows that moving from 300 VMs to 12,000 pods is a 40x increase of collected data, all with a single, relatively small change from VMs to containers. The good news is that by slicing it up more granularly, teams can dive into more detail and pinpoint issues at a more granular level more quickly.

However, in practice, cardinality works exactly in this way, quickly multiplying the data points per time interval, for example, for the number of (identical) instances of a Kubernetes service running on different pods: if there are 20 instances of a service running on 20 different Kubernetes pods, Prometheus (and pretty much every other time-series database) includes the 20 unique values and increases the cardinality and data points per second by 20 times.

And often, those that add dimensions to the platform, aren't the ones responsible for the performance and operations of the platform itself, creating a tension between consumers of the data and the observability platform team. And while the choice of time-series database has a definite impact on the level at which cardinality performance starts to suffer, dealing with cardinality, ingestion rate, and other exponential data growth challenges are the key operational priority for the observability team to ensure adequate service levels of the platform. The compute and storage needed to process all this data quickly is just not cost-effective at this scale, creating bottlenecks for reliability, latency and ingestion, and processing rate.

Low response time and quick processing of ingested data are key when iterating rapidly through hypothesis after hypothesis to explore the various potential sources of an issue with a production service, and you'd want every team to be able to query large volumes of (near-) realtime data with sub-second latencies.

Now that we understand the knobs we can tune, let's dive into how we can actually gather more dimensions for each metric, without degrading performance.

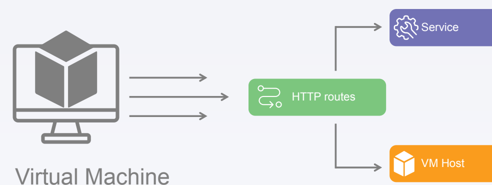


Image 3: a simple VM-based architecture

Imagine this single virtual machine-based example, where 300 virtual machines host 5 services. Each service has 5 HTTP status code types and 20 HTTP routes. This setup, while simple, already results in 150 thousand possible unique time series.

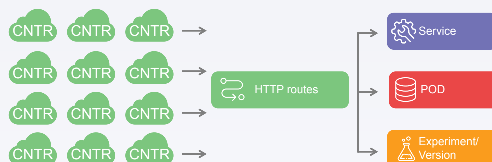


Image 4: a more complex container-based architecture

Now imagine a similar container-based example, with the same 5 services, 5 HTTP status code types and 20 HTTP routes, but now running on 12,000 pods results in six million possible unique time series.

Effectively balancing cardinality and cost/complexity

Each team has to continuously make accurate trade-offs between the cost of observing their service (or application), and the value of the insights the platform drives, looking for a continuously moving sweet spot. This sweet spot will be different for every service, as some have higher business value than others, so those services can capture more dimensions, with higher cardinality, better resolution, and longer retention than others.

So we now understand the tightrope that we must walk across, balancing highly detailed insights with the exponential cost and complexity associated with this level of detail. This constant balancing of cost and derived value also means there is no easy fix. No cheat code to an optimal observability system. There are, however, some things you can do. In this chapter, we'll dive into best practices that help you make the most impact.

Understanding the landscape

Every organization, according to Conway's law, is prone to shipping software systems that mirror their organizational chart. That means that any organization that benefits from a microservices architecture is inherently a complex organization, with a complex technical landscape. Teams need to be able to decipher and triage issues, cutting through both the organizational and technical complexities by having adequate observable systems that span the entire landscape of distributed services, not just their microservice.

That means teams must understand their place in the landscape, and know what part their services play in the organization's value streams to fully understand expected and normal traffic flows, requests and data. These help inform teams what parts of their service they need to instrument at what level.

There are various activities teams can engage in to create a better understanding of their place in the application services landscape. Value Stream Mapping is a key example that helps teams create an understanding of the value stream their services are a part of by visualizing the flow of value through the organization. It can be especially useful for figuring out technical and organizational dependencies. On the observability side of things, being able to see dashboards and metrics of services that they depend on, and of services that depend on them, can help create cohesion and understanding between teams with adjacent services.

Setting guardrails

Observability platforms are usually shared between teams. That means that teams can't take up more than their fair share of the platform's resources. What this fair share entails, depends on the business value of each of the team's services, and translates into a per-service or per-team budget.

Teams can then work out the cost vs. benefit tradeoff to stay within that budget, aligning the level of detail with the allotment of capacity. The observability team helps teams to stay within budget when they, inevitably, add dimensions and increase cardinality, resolution or retention as time passes.

With guardrails in place, teams can now make their services observable, adding the right dimensions, metrics, tags, and labels to be able to troubleshoot and triage services as issues occur, without overrunning their budget, overusing resources or grinding the platform to a halt.

Standardize common instrumentation

One key value an observability platform team brings to the table is the ability to standardize across the loosely coupled landscape of application or feature teams and their microservices.

They're uniquely positioned to create internal frameworks that standardize how to instrument common services, like container platforms, database services, load balancers, and more. This helps limit the cardinality of the overall system by removing the variation and duplication of many teams each doing the same boilerplate work a little differently.

And thanks to many open source tools like nginx and Envoy that standardized on the OpenMetrics format, it becomes quite easy for the observability team to enable standardized metrics out of the box.

That common and shared framework of instrumenting common services also allows the observability platform team to implement and show common instrumentation principles and best practices. Not only does this help with consistent implementation across the landscape, it also helps with onboarding teams, speeding up the instrumentation phase, and frees up teams to spend on more valuable work instead reducing resources and effort spent on the common layers of the infrastructure.

Now, a framework is just that: scaffolding that provides structural support. But the framework shouldn't be so prescriptive that teams cannot deviate; the goal is to create a framework that helps teams to get to 80% quickly and consistently by using languages that have pre-built support for automatic instrumentation, service scaffolding templates, or leveraging side car service mesh architectures. The goal of the 80% rule is to provide a basic, consistent implementation across common technologies and shared infrastructure, so that teams don't have to spend time and effort on non-differentiating work, and instead can focus on the 20%.

The 20% is likely highly specific to each individual team, which is where teams need the freedom and flexibility to deviate from a standard. The observability team acts as a trusted advisor to help other teams reduce the exponential growth that high-cardinality data sets pose and optimize the insights driven by this 20%.

Teaching all teams the same way of work, with the same best practices, has the added benefit of consistency, making it easy for teams to explore and observe adjacent services that they don't own, building a uniform and shared understanding across the services landscape, and helps keep the entire system maintainable and future-proof.

Limiting dimensionality

The simplest way of managing the explosion of observability data is by reducing what dimensions you collect for metrics. By setting standards on what types of labels are collected as part of a metric, some of the cardinality can be farmed out to a log or a trace, which are much less affected by the high cardinality problem. And as discussed in the previous chapter, the observability team is uniquely positioned to help teams set sane defaults for their services.

These standards may include how and what metrics will use what labels, moving higher cardinality dimensions like unique request IDs to the tracing system to unburden the metrics system.

This is a strategy that limits what is ingested, limiting the amount of data sent to the metrics platform. This can be a good strategy when teams and applications are emitting metrics data that is not relevant, reducing cardinality before it becomes a problem. And isn't preventing problems favorable to solving them?

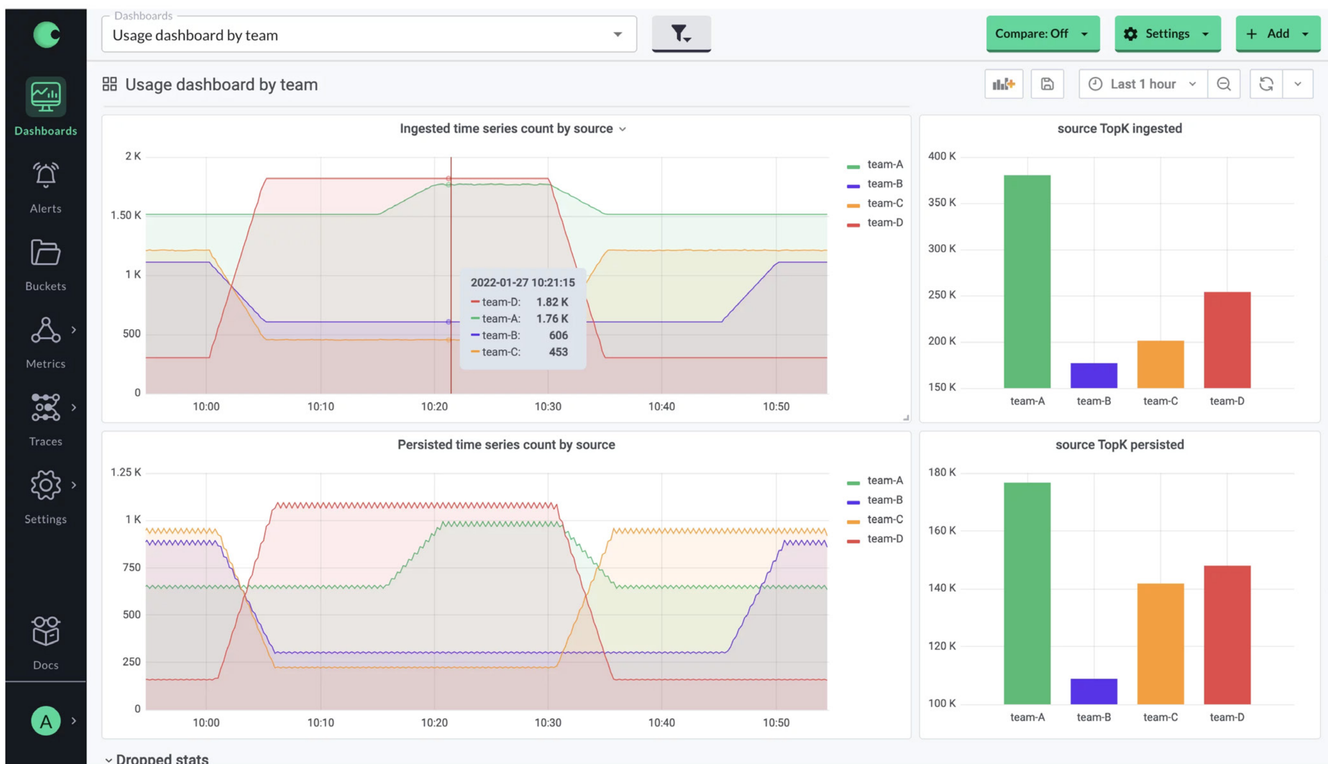


Image 5: monitoring the metrics

This can be done by setting up monitoring and alerting for what metrics data each service and/ or team emits, as seen by the example in image 5. By measuring the usage, teams can see what resources they are consuming, and weigh it against cost and derived value of the insights the data brings them. Observability teams can help developers understand what metrics data they are producing, and where high cardinality labels and dimensions may cause problems, and how to solve them.

Unfortunately, it's not always possible to reduce the cardinality of the metrics data as emitted by the monitored service. Sometimes, it's because we're troubleshooting a hard-to-pinpoint issue, and we need the additional context these labels give us to exclude contributing factors or dive deeper into a theory. There's also scenarios where teams have little control over what metrics data is emitted, for instance with commodity off-the-shelf services and applications. In these situations, reducing resolution, lowering retention, and aggregation give us alternatives to reduce both required storage capacity as well as computational load of queries.

Downsampling

Reducing resolution or downsampling is a tactic to reduce the overall volume of data by reducing the sampling rate of data. This is a great strategy to apply, as the value of the resolution of metrics data diminishes as it ages. Very high resolution is only really needed for the most recent data, and it's perfectly ok for older data to have a much lower resolution so it's cheaper to store and faster to query.

Downsampling can be done by reducing the rate at which metrics are emitted to the platform, or it can be done as it ages. This means that fresh data has the highest frequency, but more and more intermediate data points are removed from the data set as it ages. It is of course important to be able to apply resolution reduction policies at a granular level using filters, since different services and application components across different environments need different levels of granularity. Unfortunately, not every observability platform supports this level of granularity (like Prometheus), and only some do (like M3), making Prometheus ill-suited for long-term storage of metrics data.

By downsampling resolution as the metrics data ages, the amount of data that needs to be saved is reduced by orders of magnitude. Say we downsample data from 1 second to 1 minute, that is a 60x reduction of data we need to store. Additionally, it vastly improves query performance.

A solid downsampling strategy includes prioritizing what metrics data (per service, application or team) can be downsampled, and determining a staggering age strategy. Often, organizations adapt a week-month-year strategy to their exact needs, keeping high-resolution data for a week (or two), and stepping down resolution after a month (or two) and after a year, keeping a few years of data. With this strategy, teams retain the ability to do historical trend analysis with week-over-week, month-over-month and year-over-year.

Lowering retention

By lowering retention, we're tweaking the total amount of metrics data kept in the system by discarding older data (optionally after downsampling first).

By classifying and prioritizing data, as we discussed in the *Understanding the Landscape* chapter, we can get a handle on what data is ephemeral and only needed for a relatively short amount of time (such as dev or staging environments, low-business-value services), and what data is important to keep for a longer period of time to refer back to as teams are triaging issues. Again, being able to apply these retention policies granularly is key for any production-ready system, as a one-size-fits-all approach just doesn't work for every metric alike.

For production environments, keeping a long-term record, even at a **lower resolution**, is key to looking at longer trends and being able to compare year-over-year. However, we don't need all dimensions or even metrics for this long-term analysis. Helping teams choose what data to keep, at a low resolution, and what metrics to discard after a certain time will help limit the amount of metrics data that we store, but never look at again.

Similarly, we don't need to keep data for some kinds of environments, such as dev, test, or staging environments. Likewise for, services with low business value or non-customer facing, internal services. By choosing to limit retention for these, teams can balance their ability to query health and operational state, without overburdening the metrics platform.

Aggregation

Aggregation is a completely different form of downsampling. Instead of throwing away intermediate data points, you can aggregate individual data points into new summarized data points. This reduces the amount of data that needs to be processed and stored, lowering storage cost and improving query performance for larger, older data sets.

Aggregation can be a better strategy because it lets teams continue to emit highly dimensional, high-cardinality data from their services, and then adjust it based on the value it provides as it ages.

While tweaking resolution and retention are relatively simple ways to reduce the amount of data stored by deleting data, they don't do much to reduce the computational load on the observability system. Because teams often don't need to view metrics across all dimensions, a simplified, aggregate view (for instance, without a per-pod or per-label level) is good enough to understand how your system is performing at a high level. So instead of querying tens of thousands of time series across all pods and labels, we can make do with querying the aggregate view and with only a few hundred time series.

Aggregation is a way to roll up data into a more summarized but less-dimensional state, creating a specific view of metrics and dimensions that are important. The underlying raw metrics data can be kept for other use cases, or it can be discarded to save on storage space and to reduce cardinality of data if there is no use for the raw unaggregated data.

There are two schools of aggregation: streaming vs. batch. With stream aggregation, metrics data is streaming continuously, and the aggregation is done in memory on the streaming ingest path before writing results to the time series database. Because data is aggregated in real-time, streaming aggregation is typically meant for information that's needed immediately. This is especially useful for dashboards, which need to query the same expression repeatedly every time

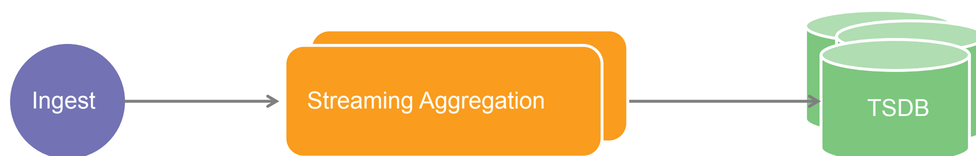


Image 6: streaming aggregation

they refresh. Streaming aggregation makes it easy to drop the raw unaggregated data to avoid unnecessary load on the database.

Batch aggregation first stores raw metrics in the time series database, and periodically fetches them and writes back the aggregated metrics. Because data is aggregated in batches over time, batch aggregation is typically done for larger swaths of data that isn't time-sensitive. Batch aggregation cannot skip ingesting the raw non-aggregated data, and even incurs additional load as written raw data has to be read, and re-written to the database, adding additional query overhead.

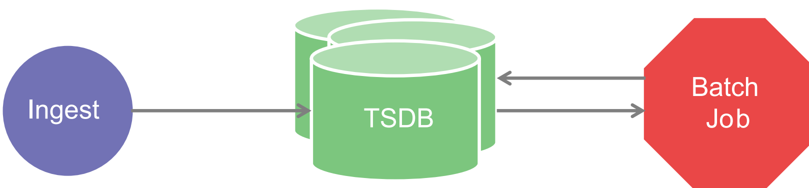


Image 7: batch aggregation

The additional overhead of batch aggregation makes streaming better suited to scaling the platform, but there are limits to the complexity real-time processing can handle due to the real time nature; batch processing can deal with more complex expressions and queries.

Optimizing for heavy queries is a prime example of where the observability team's expertise comes in. For instance, taking a highly-dimensional metric, and using aggregation to turn it into two metrics that retain a subset of the cardinality to match how the data is consumed, can optimize query performance significantly. The ability to share this knowledge across all teams using the platform ensures performance maintains adequate levels, regardless of who creates the queries. Aggregations are an important, perhaps even the most effective of these three strategies.

Suffice it to say that there are different approaches (like Prometheus' Recording Rules and M3's Rollup Rules). Choosing the right one is especially important when you'd want to drop the original data, which is not possible on all platforms. If you want to dive deeper into practical approaches, Chapter 4 of the O'Reilly *Report on Cloud Native Monitoring*[1] goes into a great level of detail.

[1] https://go.chronosphere.io/oreilly-report-cloud-native-monitoring.html

Observe the observability platform

Operating an observability platform that continuously drives more valuable insights than the cost and effort put into it is incredibly hard. It requires constant tweaking of the platform and the observability data each service emits onto the platform. It also requires making the right decisions on time-series database, ingestion infrastructure, and processing capabilities.

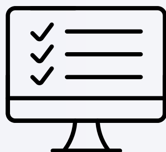
In addition to offering best practices to product teams to keep the system up and running, the observability team can use the platform to gain insights into how teams are using the platform and help them optimize their usage. For instance, by keeping track of the volume per labels or metrics, the observability team can put rate limiting in place, or limit the number of metrics/labels each team is allowed. They can also put query limitations in place to prevent a single user from overloading the entire system with a slow or computationally heavy query.

This way, the platform teams can create an overview of how teams are using the platform, giving them the ability to sit down with specific teams on specific issues, helping them optimize retention, resolution and aggregation where needed. The team then refines best practices, shares gained insights and learnings, and organizes periodic knowledge transfer sessions.

And this much needed, as observability is a moving goalpost. Observability moves where the business and technical priorities are. As the business evolves, and the technological landscape evolves, what we measure and monitor changes as well. To be successful, platform and development teams need to recognize, and stay on top of, growing pains, bottlenecks caused by (the removal of) technical debt, market conditions, seasonality, changing business priorities, or other factors.

This creates a sense of shared, but distinctly divided ownership where individual teams own the observability of their own services, but the platform team is responsible for the overall health and optimal usage of the observability platform.

The platform team helps individual teams adopt and implement observability, so that everyone can make the most out of the investments into the platform, but puts development teams squarely in charge of their own observability journey and destiny, and makes observability a first-class citizen and a requirement for shipping to production.



Summary

Observability is a key capability – and skill – to cope with the growing complexity of cloud native applications. It is the flashlight that helps shine light on complex, emergent, and transient application behaviors. It democratizes the expertise needed to create insights that help improve customer experience continuously – and sometimes drastically.

In this paper, we've seen that observability platform teams are the key providers, and educators, of these capabilities. To empower developers to own the resilience, performance, and security of their services in production.

To make this happen, observability teams must continuously balance cardinality, resolution retention, and aggregation to manage the unbridled data growth and cost of storing and processing these vast amounts of telemetry. It's their job to make observability cost-effective, and keep it that way as the platform scales and your business grows.

The key challenge is to allow teams to emit the right amount of data with the right precision and granularity so they can derive the right decisions from the data to continuously improve their customer service and solve problems faster.

Finding the right balance between too much information and not enough requires observability and development teams alike to understand the business that they're operating in, and to understand the business value of the services they're working on. That will help them understand where high cardinality is most valuable, to optimize resolution and retention for critical services, and to reduce cost by tweaking retention, resolution, and cardinality based on the business priorities.

We've seen that the right controls must be in place before the data volume becomes a problem to be successful in the future, instead of having to dial down precision because of platform limitations.

And really, what's better than being able to say 'sure, add another label' when a critical service is having a hard-to-troubleshoot performance issue?

Taming rampant data growth with Chronosphere

Chronopshere understands the high cardinality problem first hand. The company's co-founders Martin Mao (CEO) and Rob Skillington (CTO) previously ran the observability team at Uber where they solved the challenge of running large-scale observability for cloud native environments.

A key differentiator of Chronosphere's observability platform value is the control plane, which allows customers to aggregate, adjust resolution, and set retention on their observability data. This helps them manage the amount of metric data they are paying to keep, and eliminates surprise overages. Chronosphere customers have realized a reduction in metric cardinality by up to 98% per rule and 50% overall on spikes.

Chronosphere is the only observability platform that puts you back in control by taming rampant data growth and cloud native complexity, delivering increased business confidence. Engineering organizations at startups to well-known global brands in the Fortune 500 around the world trust Chronosphere to help them operate scalable, highly available, and resilient applications.

ABOUT TLA TECH

TLA Tech is dedicated to helping organizations create great developer experiences for their engineering teams by researching solutions, finding new ways to work and making technology choices simpler. TLA Tech's insights are based on Joep Piscaer's hands-on experience of making technology work in the real world as a CTO, industry analyst, engineer and technical pathfinder. Joep has seen enterprises struggle with ingesting and querying metrics, logs and traces while trying to keep up with the pace, scale and ephemeral nature of cloud native applications.

This paper was sponsored by Chronosphere. However, TLA Tech retains final editorial control over this publication. Copyright © 2022 TLA Tech B.V.. All rights reserved worldwide.