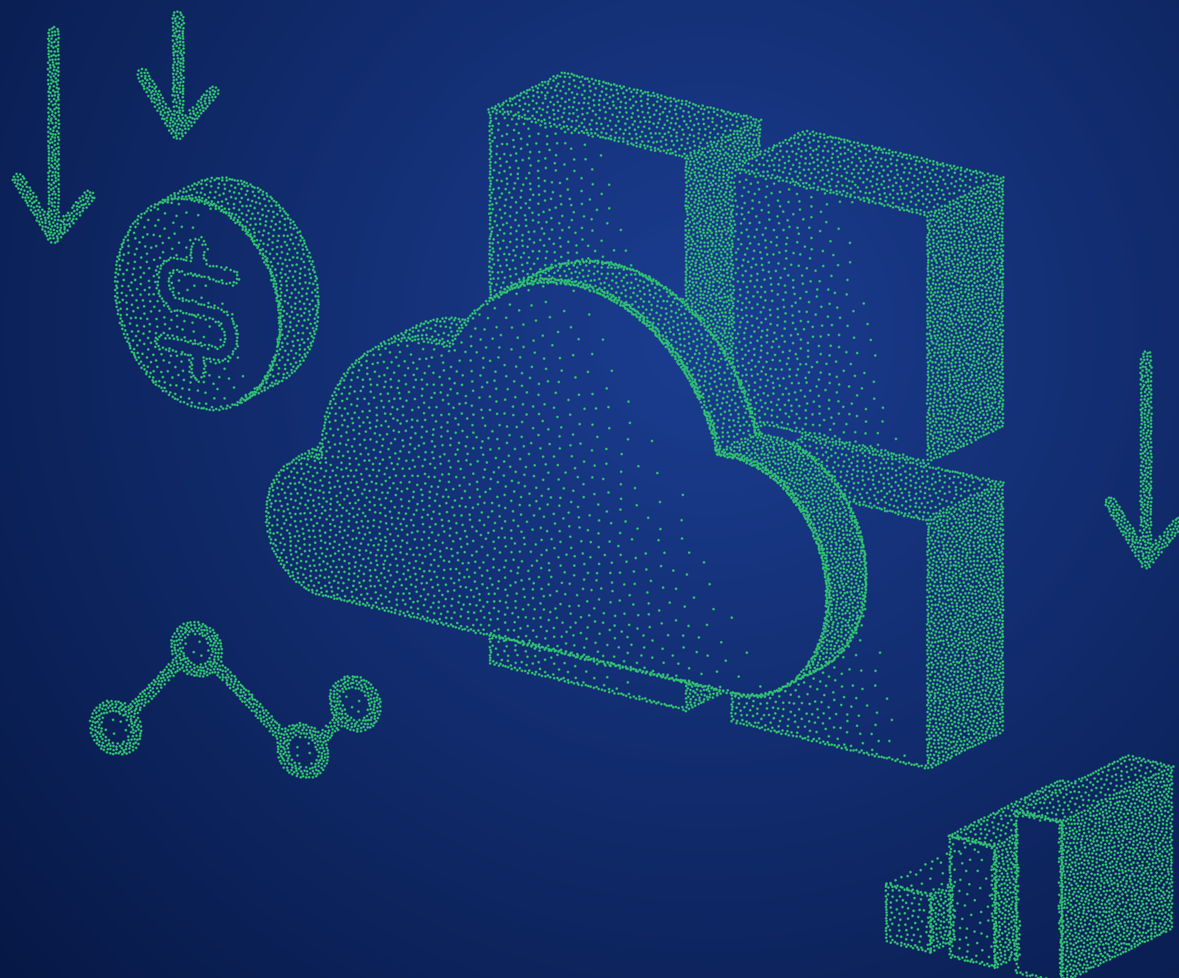
EBOOK

4 ways to reduce cloud native observability costs

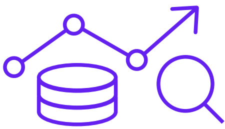
Affordably keep your systems and your business running at top performance



The cloud native observability challenge

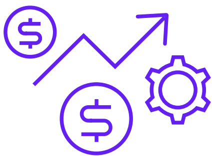
Companies of all sizes are rapidly moving to cloud native technologies and practices. This modern strategy offers speed, efficiency, availability, and the ability to innovate faster, which means organizations can seize business opportunities that simply aren't possible with a traditional monolithic architecture.

Yet moving to an architecture based on containers and microservices creates a new set of challenges that, if not managed well, will undermine the promised benefits.



Exploding observability data growth

Cloud native environments emit a massive amount of monitoring data — somewhere between 10 and 100 times more than traditional VM-based environments. This is because every container/microservice is emitting as much data as a single VM. Additionally, service owners start adding metrics to measure and track more granularly to run the business. Scaling containers into the thousands and collecting more and more complex data (eg: higher data cardinality) results in data volume becoming unmanageable.



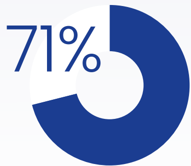
Rapid cost increases

The explosive growth in data volume and the need for engineers to collect an ever-increasing breadth of data has broken the economics and value of existing infrastructure and application monitoring and tools. Costs can unexpectedly spike from a single developer rolling out new code. Observability data costs can exceed the cost of the underlying infrastructure.

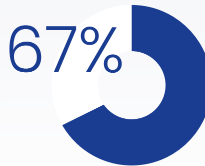
As the amount of metrics data being produced grows, the pressure on the observability platform grows, increasing cost and complexity to a point where the value of the platform diminishes. So how do observability teams take control over the growth of the platform's cost and complexity, without dialing down the usefulness of the platform? This eBook describes the trade-offs between cost and value that can come with investing in observability. It specifically dives into four ways you can significantly reduce costs and still get all the promised benefits from your observability platform:

1. Limit dimensionality
2. Use downsampling
3. Lower retention
4. Use aggregation

In a recent survey of 500 engineers,



said their business can't innovate effectively without good observability.



said having a strong observability function provides the foundation for all business value.

Cardinality: A primer

To understand the balance between cost and insight, it's important to understand cardinality. This is the number of possible ways you can group your data, depending on its properties, also called dimensions.

Metric cardinality is defined as the number of unique time series that are produced by a combination of metric names and associated dimensions. The total number of combinations that exist are cardinalities. The more combinations that are possible, the higher a metric's cardinality is. Here's a delicious practical example: purchasing fine cheese.

Understanding data sets

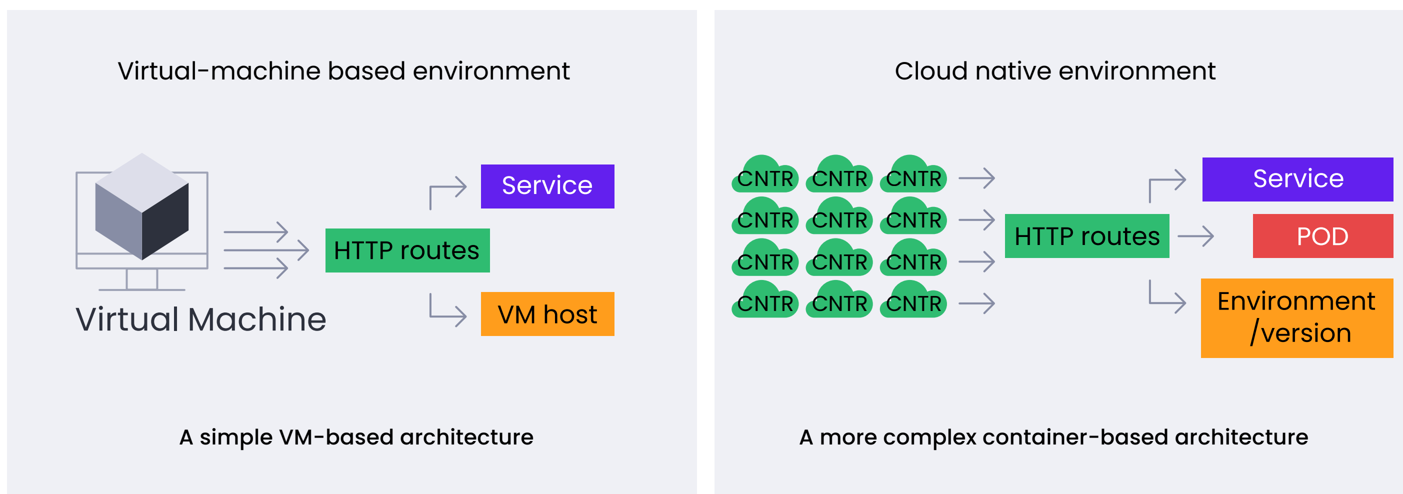
If your only preference is that the cheese you buy is made of sheep's milk, your data would have just one dimension. Analyze 100 different kinds of cheese based on that dimension, you'd have 100 data points, each labeling the cheese as either sheep's milk-based or not (made from another source). But then you decide you only want sheep's milk cheese made in France. That would add another dimension to track for each cheese made of sheep's milk — the country of origin. Think of all the cheese-producing countries in the world — about 200 — and you can understand how the cardinality, or the ways to group the data, can quickly increase.

If you then decide to analyze the data based on the type of cheese, it adds many hundreds of other dimensions for grouping (think of all the different kinds of cheese in the world). Finally, you decide you want to only consider Camembert, and group Camembert cheese only by whether it was made with raw milk, warm milk, or completely pasteurized milk. That's three more dimensions. You'd be right in thinking that, with all these dimensions, the cardinality would be high — even in traditional on-premises, VM-based environments. A key point, it's difficult to calculate the overall cardinality of a data set. You can't just multiply together the cardinality of individual dimensions to know what the overall cardinality is — you will frequently have dimensions that only apply to a subset of your data.

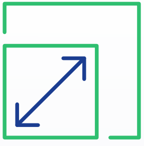
Controlling cardinality

With the transition from monolithic to cloud native environments, there's been an explosion of metrics data in terms of cardinality. This is because microservices and containerized applications generate metrics data an order of magnitude more than monolithic applications on VM-based cloud environments. To achieve good observability in a cloud native system, then, you need to deal with large-scale data and take steps to understand and control cardinality.

From 150,000 to 150 million metrics with cloud native architecture

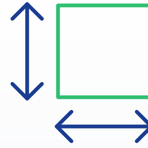


In addition to cardinality, it's important to understand two other terms when managing data quantity in an observability platform: resolution and retention.



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, making them not even show up in the data; time precision is an important aspect of catching transient and spiky behaviors.



Retention

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

Classifying cardinality

When it comes to cardinality in metrics, you can classify dimensions into three high-level buckets to consider the balance between value and cardinality:

High value

These are the dimensions you need to measure to understand your systems, and they are always or often preserved when consuming metrics in alerts or dashboards. An example is including service/endpoint as a dimension for a metric tracking request latency. There's no question that this is essential for visibility to make decisions about your system. But in a microservices environment, even a simple example like this can end up adding quite a lot of cardinality. When you have dozens of services each with a handful of endpoints, you quickly end up with many thousands of series even before you add other sensible dimensions such as region or status code.

Low value

These dimensions are of more questionable value. They may not even be intentionally included, but rather come because of how metrics are collected from your systems. An example dimension here is the instance label in Prometheus — it is automatically added to every metric you collect. Although in some cases you may be interested in per-instance metrics, looking at a metric such as request latency for a stateless service running in Kubernetes, you may not look at per-instance latency at all. Having it as a dimension does not necessarily add much value.

No value (useless or even harmful)

These are essentially anti-patterns to be avoided at all costs. Including them can result in serious consequences to your metric system's health by exploding the amount of data you collect and causing significant problems when you query metrics.

4 ways to keep your observability costs low

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. 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.

This constant balancing of cost and derived value also means there is no easy fix. There are, however, some things you can do to keep costs in check.

1. Limit 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 the observability team is uniquely positioned to help teams set appropriate 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, which reduces 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.

2. Use downsampling

Downsamping is a tactic to reduce the overall volume of data by lowering 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.

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.

3. Lower 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, 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 or 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.

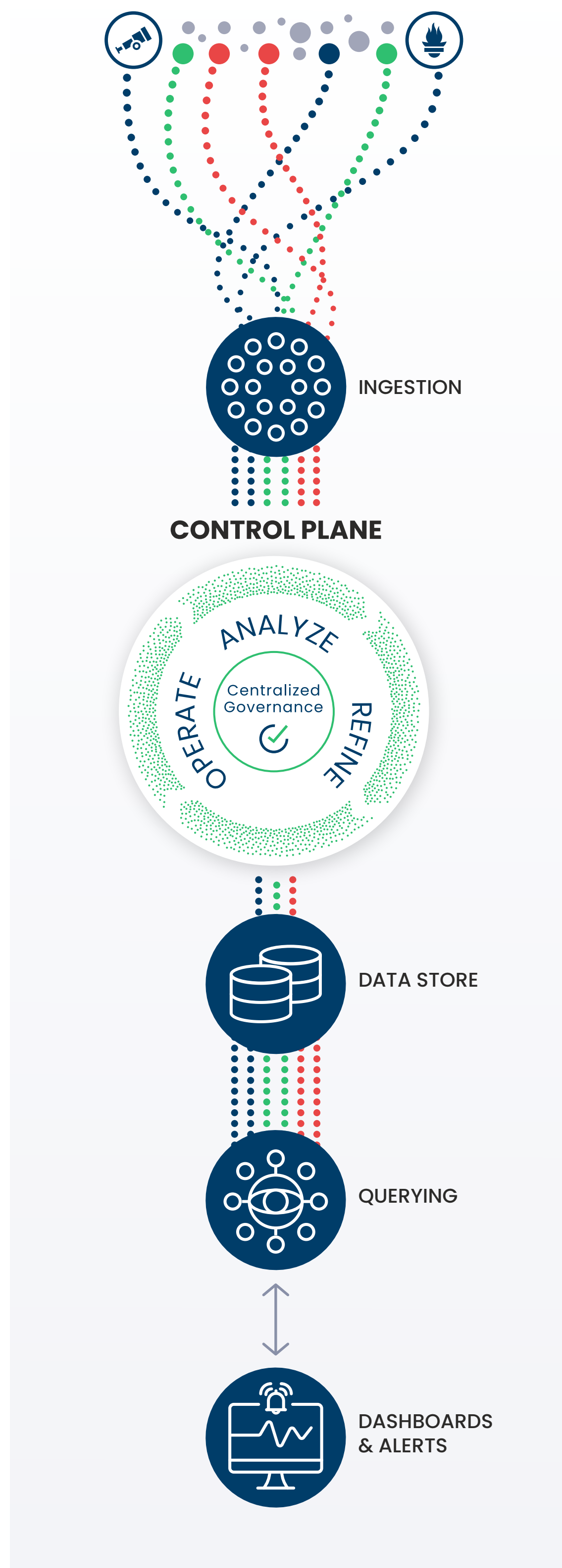
For production environments, keeping a long-term record, even at a lower resolution, is key to being able to look at longer trends and being able to compare year-over-year. However, we don't need all dimensions or even metrics for this long-term analysis. Helping teams choose what data to keep, at a 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. The same is true 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.

4. Use aggregation

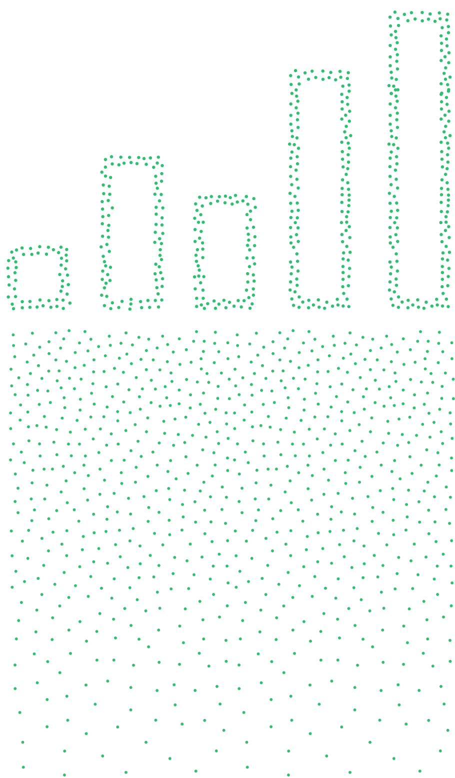
Instead of throwing away intermediate data points, aggregate individual data points into new summarized data points. This reduces the amount of data that needs to be processed and stored, lowering storage cost and improving query performance for larger, older data sets.

Aggregation can be a good 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 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.



Stream vs batch

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

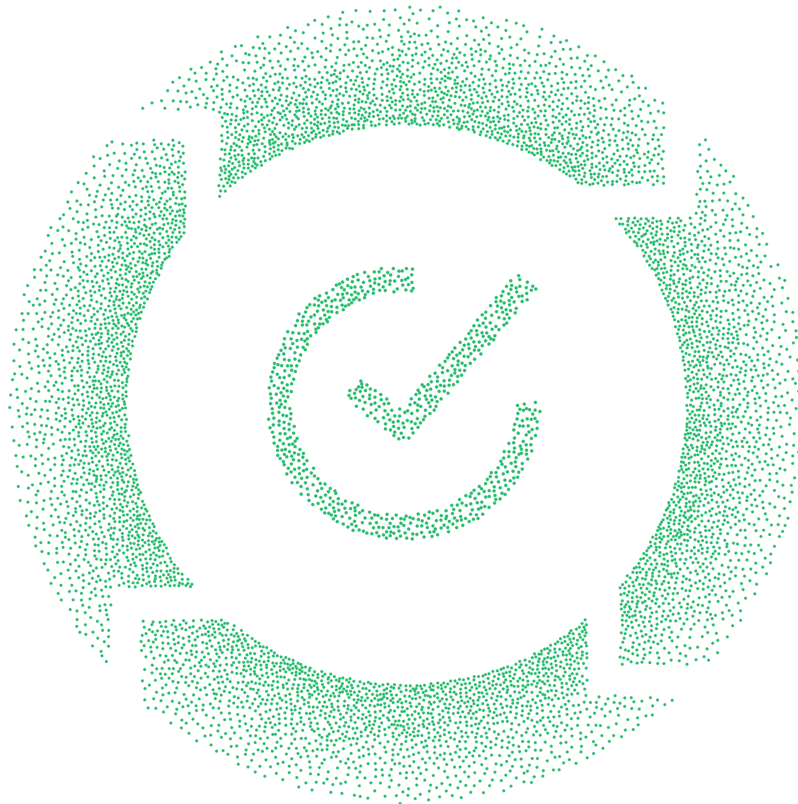
Batch aggregation first stores raw metrics in the time series database, and periodically fetches them and writes back the aggregated metrics. Because data is aggregated in batches over time, batch aggregation is typically done for larger swaths of data that isn't time-sensitive. Batch aggregation cannot 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.

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.

Rethink observability; control your costs

Before you adopt a cloud native observability platform, be sure it will help you keep costs low by enabling you to understand the value of your observability data as well as shaping and transforming data based on need, context, and utility. Get more from your investment, too, with capabilities that permit you to delegate responsibility for controlling cardinality and growth and continuously optimize platform performance.

The cloud native Chronosphere observability platform does all this, and more. It helps you keep costs low by identifying and reducing waste. It also improves engineers' experience by reducing noise. Best of all, teams remediate issues faster with Chronosphere's automated tools and optimized performance.



Learn more and watch a demo at **chronosphere.io**