Monitoring Modern Infrastructure

DATADOG

> “Measure what is measurable,
> and make measurable what is not so.”
>
> — Galileo

UPDATED MARCH 2023

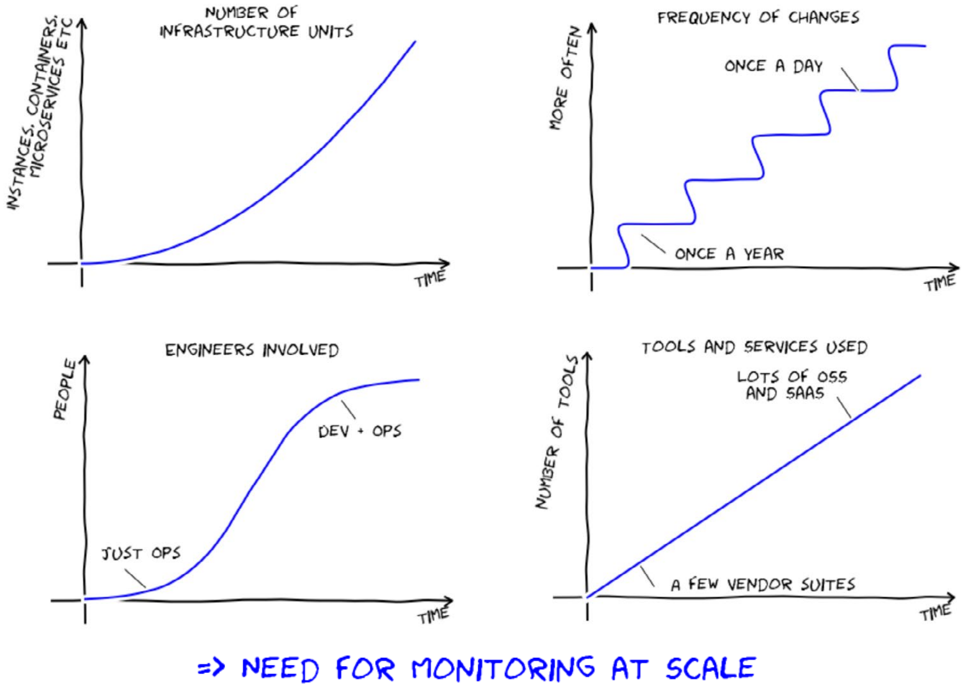
This version builds on previous editions of our Monitoring Modern Infrastructure eBook to cover Kubernetes and AWS Lambda.

Chapter 1: Constant Change

In recent years, the nature of IT infrastructure has changed dramatically. Countless organizations have migrated away from using on-premise data storage, servers, and services to take advantage of the agility and scalability afforded by public and private cloud infrastructure. And for new organizations, architecting applications for the cloud is now the default.

The cloud has effectively removed the logistical and economic barriers to accessing production-ready infrastructure. Now, any organization or individual can harness the same technologies that power some of the biggest companies in the world.

The shift toward the cloud has brought about a fundamental change on the operations side as well. We are now in an era of dynamic, constantly changing infrastructure—and this requires new monitoring tools and methods.



In this book, we will outline an effective framework for monitoring modern infrastructure and applications, however complex or dynamic they may be. With this high-level framework in place, we will then dive into a key component of monitoring: metric graphing and visualization. Finally, we will ground these monitoring principles by showing how they apply to two popular infrastructure technologies: Kubernetes and AWS Lambda.

Elastic, Dynamic, Ephemeral Infrastructure

By its nature, cloud-based infrastructure is elastic, dynamic, and constantly changing. This presents both benefits and challenges for organizations.

Migrating to a hybrid or fully cloud infrastructure gives organizations and their teams more flexibility in how they design their services. Developers and sysadmins can now spin up nearly limitless cloud resources on demand, taking advantage of a wide range of innovative technologies to transform their environments. From cost-effective serverless and container ecosystems to highly available managed services, modern infrastructure has benefited significantly from the cloud.

In many cases, no manual intervention is required to add or subtract new resources, as auto-scaling allows infrastructure to expand or contract to keep pace with changing demand. Auto-scaling is a key feature of cloud services such as Amazon's EC2 and container-orchestration tools such as Kubernetes.

The elastic nature of modern infrastructure means that individual components are often ephemeral and/or single-purpose. Cloud computing instances can run for just hours or days before being destroyed. The shift toward containerization has accelerated this trend, as containers often have short lifetimes measured in minutes or hours.

Serverless has also experienced a considerable amount of growth. With its ability to abstract away the complexity of provisioning and managing underlying resources, serverless enables developers to deploy more efficient, independent services.

But these technologies have also required organizations to overhaul the way they build applications in the cloud and monitor their performance.

Pets vs. Cattle

With dynamic infrastructure, focusing on the performance of individual servers rarely makes sense—each compute instance or container is essentially a replaceable cog that performs some function in support of a larger service.

A useful analogy in thinking about dynamic infrastructure is “pets versus cattle.” Pets are unique—they have names, and owners typically care greatly about the health and well-being of each one. Cattle, on the other hand, are part of a herd—they are numbered rather than named. Individual members of the herd will come and go, and owners tend to care more about the overall health of the herd than they do about any one animal.

In most cases, your servers, containers, and other cloud infrastructure components can be thought of as "cattle." Therefore, when it comes to monitoring, you should focus on the aggregate health and performance of services rather than on isolated datapoints from your hosts. Rarely should you page an engineer in the middle of the night for a host-level issue such as elevated CPU usage. On the other hand, if latency for your web application starts to surge, you’ll want to take action immediately.

The Evolution of DevOps

As cloud, container, and serverless technologies have reshaped the underlying infrastructure, software development and operations have become more dynamic as well. The DevOps movement has evolved to accommodate these new technologies and ensure that software is tested, deployed, and managed efficiently and safely. DevOps practices like "shifting left" focus on identifying bugs and performance issues earlier in the development process. Traditional deployment strategies such as canary and blue-green have evolved to support both serverless and container orchestration systems.

These trends have also prompted a change in how DevOps personnel manage continuous integration and delivery (CI/CD) pipelines and observability.

CONTINUOUS INTEGRATION AND DELIVERY

CI/CD is a cornerstone of many DevOps approaches. Rather than orchestrating large, infrequent releases, teams that use CI/CD push small, incremental code changes quickly and frequently. This simplifies the process of building, testing, and merging new commits and allows development teams to release bug fixes and new features much faster. It also enables engineers to quickly roll back any changes that cause unforeseen issues in production.

OBSERVABILITY

In control theory, observability is the property of being able to describe or reconstruct the internal state of a system using its external outputs. In practice, for an organization’s infrastructure, this means instrumenting all compute resources, apps, and services with “sensors” that dependably report metrics from those components. It also means making those metrics available on a central, easily accessible platform where observers can aggregate them to reconstruct a full picture of the system’s status and operation. Observability dovetails with DevOps practices, as it represents a cultural shift away from siloed, piecemeal views into critical systems toward a detailed, comprehensive view of an organization's environment.

Modern Approaches to Monitoring

Monitoring is the part of the DevOps toolchain that enables developers and operations teams to build observability into their systems. In fact, modern DevOps practices are made possible in part by monitoring tools.

Monitoring is a must now that development teams move faster than ever—some teams release new code dozens of times per day. It provides teams with the ability to understand and investigate complex, distributed applications where components can fail in unpredictable ways. In most cases, the motivation for monitoring is being able to catch and resolve performance issues before they cause problems for end users.

The core features of a modern monitoring system are **single source of truth**, **built-in aggregation**, **scalability**, **sophisticated alerting**, and **collaboration**.

SINGLE SOURCE OF TRUTH

Today's monitoring systems are comprehensive; they offer the ability to collect and analyze multiple types of telemetry data from various sources in an environment. This creates a centralized place for teams—including development, operations, security, product, and more—to review and collaborate on real-time performance data, regardless of its source. As organizations deploy more ephemeral and single-purpose resources, having a single source of truth like this is critical for tracking activity across all services.

BUILT-IN AGGREGATION

Powerful tagging and labeling schemes allow engineers to arbitrarily segment and aggregate metrics, so they can direct their focus at the service level rather than the host level. Remember: cattle, not pets.

SCALABILITY

Modern, dynamic monitoring systems accommodate the fact that individual hosts come and go, and scale gracefully with expanding or contracting infrastructure. When a new host is launched, the system should detect it and start monitoring it automatically. Strategies like "monitoring as code" accomplish these goals by creating a repeatable process for deploying observability solutions alongside infrastructure. This process ensures that any change to a system is instantly and immediately monitored.

SOPHISTICATED ALERTING

Virtually every monitoring tool can fire off an alert when a metric crosses a set threshold. But in rapidly scaling environments, such fixed alerts require constant updating and tuning. More advanced monitoring systems offer flexible alerts that adapt to changing baselines, including relative change alerts as well as automated outlier and anomaly detection.

COLLABORATION

When issues arise, a monitoring system should help engineers discover and correct the problem as quickly as possible. That means delivering alerts through a team's preferred communication channels and making it easy for incident responders to share graphs, dashboards, events, and comments.

How It's Done

In the next chapter, we dive into the how-to of monitoring, laying out the details of a practical monitoring framework for modern infrastructure. We'll start with data, which is at the core of any monitoring approach. You'll learn techniques for collecting, categorizing, and aggregating the various types of monitoring data produced by your systems. You'll also understand which data is most likely to help you identify and resolve performance issues.

This framework comes out of our experience monitoring large-scale infrastructure for thousands of customers, as well as for our own rapidly scaling application in the AWS cloud. It also draws on the work of Brendan Gregg of Netflix, Rob Ewaschuk of Google, and Baron Schwartz of VividCortex.

Chapter 2: Collecting the Right Data

Monitoring data comes in a variety of forms. Some systems pour out data continuously and others only produce data when specific events occur. Some data is most useful for *identifying* problems; some is primarily valuable for investigating problems. This chapter covers which data to collect, and how to classify that data so that you can:

1. Generate automated alerts for potential problems while minimizing false alarms
2. Quickly investigate and get to the bottom of performance issues

Whatever form your monitoring data takes, the unifying theme is this:

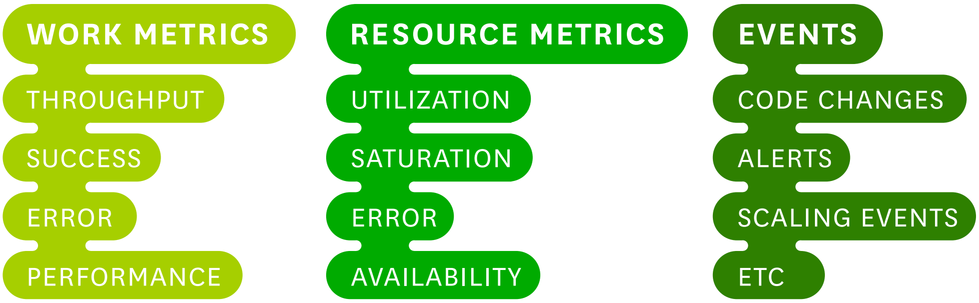
> Collecting data is cheap, but not having it when you need it can be expensive, so you should instrument everything, and collect all the useful data you reasonably can.

Most monitoring data falls into one of two categories: metrics and events. Below we'll explain each category, with examples, and describe their uses.

Metrics

Metrics capture a value pertaining to your systems at a *specific point in time*—for example, the number of users currently logged in to a web application. Therefore, metrics are usually collected at regular intervals (every 15 seconds, every minute, etc.) to monitor a system over time.

There are two important categories of metrics in our framework: work metrics and resource metrics. For each system in your infrastructure, consider which work metrics and resource metrics are reasonably available, and collect them all.



WORK METRICS

Work metrics indicate the top-level health of your system by measuring its useful output. These metrics are invaluable for surfacing real, often user-facing issues, as we'll discuss in the following chapter. When considering your work metrics, it’s often helpful to break them down into four subtypes:

- **throughput** is the amount of work the system is doing per unit time. Throughput is usually recorded as an absolute number.
- **success** metrics represent the percentage of work that was executed successfully.
- **error** metrics capture the number of erroneous results, usually expressed as a rate of errors per unit time, or normalized by the throughput to yield errors per unit of work. Error metrics are often captured separately from success metrics when there are several potential sources of error, some of which are more serious or actionable than others.
- **performance** metrics quantify how efficiently a component is doing its work. The most common performance metric is latency, which represents the time required to complete a unit of work. Latency can be expressed as an average or as a percentile, such as “99% of requests returned within 0.1 seconds.”

Below are example work metrics of all four subtypes for two common kinds of systems: a web server and a data store.

EXAMPLE WORK METRICS: WEB SERVER (AT TIME 2016-05-24 08:13:01 UTC)

SUBTYPE	DESCRIPTION	VALUE
THROUGHPUT	REQUESTS PER SECOND	312
SUCCESS	PERCENTAGE OF RESPONSES THAT ARE 2XX SINCE LAST MEASUREMENT	99.1
ERROR	PERCENTAGE OF RESPONSES THAT ARE 5XX SINCE LAST MEASUREMENT	0.1
PERFORMANCE	90TH PERCENTILE RESPONSE TIME IN SECONDS	0.4

EXAMPLE WORK METRICS: DATA STORE (AT TIME 2016-05-24 08:13:01 UTC)

SUBTYPE	DESCRIPTION	VALUE
THROUGHPUT	QUERIES PER SECOND	949
SUCCESS	PERCENTAGE OF QUERIES SUCCESSFULLY EXECUTED SINCE LAST MEASUREMENT	100
ERROR	PERCENTAGE OF QUERIES YIELDING EXCEPTIONS SINCE LAST MEASUREMENT	0
ERROR	PERCENTAGE OF QUERIES RETURNING STALE DATA SINCE LAST MEASUREMENT	4.2
PERFORMANCE	90TH PERCENTILE RESPONSE TIME IN SECONDS	0.02

RESOURCE METRICS

Most components of your software infrastructure serve as a resource to other systems. Some resources are low-level—for instance, a server's resources include such physical components as CPU, memory, disks, and network interfaces. But a higher-level component, such as a database or a geolocation microservice, can also be considered a resource if another system requires that component to produce work.

Resource metrics are especially valuable for the investigation and diagnosis of problems, which is the subject of chapter 4 of this book. For each resource in your system, try to collect metrics that cover four key areas:

— **utilization** is the percentage of time that the resource is busy, or the percentage of the resource's capacity that is in use.

— **saturation** is a measure of the amount of requested work that the resource cannot yet service. Saturation is often measured by queue length.

— **errors** represent internal errors that may not be observable in the work the resource produces.

- **availability** represents the percentage of time that the resource responded to requests. This metric is only well-defined for resources that can be actively and regularly checked for availability.

Here are example metrics for a handful of common resource types:

RESOURCE	UTILIZATION	SATURATION	ERRORS	AVAILABILITY
DISK IO	% TIME THAT DEVICE WAS BUSY	WAIT QUEUE LENGTH	# DEVICE ERRORS	% TIME WRITABLE
MEMORY	% OF TOTAL MEMORY CAPACITY IN USE	SWAP USAGE	N/A (NOT USUALLY OBSERVABLE)	N/A
MICROSERVICE	AVERAGE % TIME EACH REQUEST-SERVICING THREAD WAS BUSY	# ENQUEUED REQUESTS	# INTERNAL ERRORS SUCH AS CAUGHT EXCEPTIONS	% TIME SERVICE IS REACHABLE
DATABASE	AVERAGE % TIME EACH CONNECTION WAS BUSY	# ENQUEUED QUERIES	# INTERNAL ERRORS, E.G. REPLICATION ERRORS	% TIME DATABASE IS REACHABLE

OTHER METRICS

There are a few other types of metrics that are neither work nor resource metrics, but that nonetheless may come in handy in diagnosing causes of problems. Common examples include counts of cache hits or database locks. When in doubt, capture the data.

Events

In addition to metrics, which are collected more or less continuously, some monitoring systems can also capture events: discrete, infrequent occurrences that provide crucial context for understanding changes in your system's behavior. Some examples:

- Changes: Code releases, builds, and build failures

- Alerts: Notifications generated by your primary monitoring system or by integrated third-party tools

- Scaling events: Adding or subtracting hosts or containers

An event usually carries enough information that it can be interpreted on its own, unlike a single metric data point, which is generally only meaningful in context. Events capture *what happened*, at a point in *time*, with optional *additional information*. For example:

WHAT HAPPENED	TIME	ADDITIONAL INFORMATION
HOTFIX F464BFE RELEASED TO PRODUCTION	2016-04-15 04:13:25 UTC	TIME ELAPSED: 1.2 SECONDS
PULL REQUEST 1630 MERGED	2016-04-19 14:22:20 UTC	COMMITS: EA720D6
NIGHTLY DATA ROLLUP FAILED	2016-04-27 00:03:18 UTC	LINK TO LOGS OF FAILED JOB

Events are sometimes used used to generate alerts—someone should be notified of events such as the third example in the table above, which indicates that critical work has failed. But more often they are used to investigate issues and correlate across systems. Therefore, even though you may not inspect your events as often as you look at your metrics, they are valuable data to be collected wherever it is feasible.

Tagging

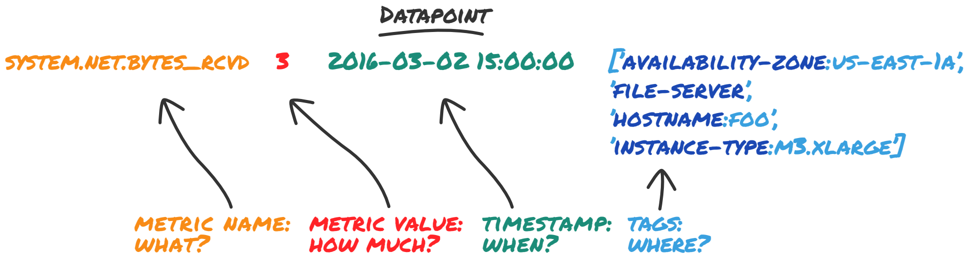
As discussed in chapter 1, modern infrastructure is constantly in flux. Auto-scaling servers die as quickly as they're spawned, and containers come and go with even greater frequency. With all of these transient changes, the signal-to-noise ratio in monitoring data can be quite low.

In most cases, you can boost the signal by shifting your monitoring away from the base level of hosts, VMs, or containers. After all, you don't care if a specific EC2 instance goes down, but you do care if latency for a given service, category of customers, or geographical region goes up.

Tagging your metrics enables you to reorient your monitoring along any lines you choose. By adding tags to your metrics you can observe and alert on metrics from different availability zones, instance types, software versions, services, roles—or any other level you may require.

WHAT'S A METRIC TAG?

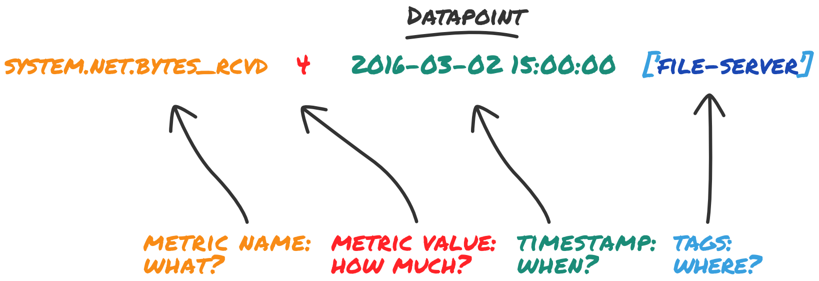
Tags are metadata that declare all the various scopes that a datapoint belongs to. Here's an example:



Tags allow you to filter and group your datapoints to generate exactly the view of your data that matters most. They also allow you to aggregate your metrics on the fly, without changing how the metrics are reported and collected.

FILTERING WITH SIMPLE METRIC TAGS

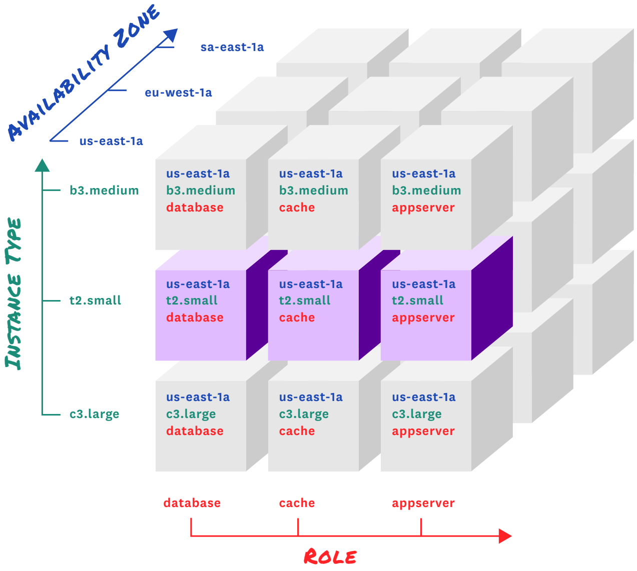
The following example shows a datapoint with the simple tag of `file-server`:



Simple tags can only be used to filter datapoints: either show the datapoint with a given tag, or do not.

CREATING NEW DIMENSIONS WITH KEY:VALUE TAGS

When you add a key:value tag to a metric, you're actually adding a new dimension (the key) and a new attribute in that dimension (the value). For example, a metric with the tag `instance-type:m3.xlarge` declares an `instance-type` dimension, and gives the metric the attribute `m3.xlarge` in that dimension. When using key:value tags, the "key" selects the level of abstraction you want to consider (e.g. instance type), and the "value" determines which datapoints belong together (e.g. metrics from instance type m3.xlarge).


AVAILABILITY ZONE
sa-east-1a
eu-west-1a
us-east-1a
INSTANCE TYPE
b3.medium
t2.small
c3.large
us-east-1a b3.medium database
us-east-1a b3.medium cache
us-east-1a b3.medium appserver
us-east-1a t2.small database
us-east-1a t2.small cache
us-east-1a t2.small appserver
us-east-1a c3.large database
us-east-1a c3.large cache
us-east-1a c3.large appserver
database
cache
appserver
ROLE

If you add other metrics with the same key, but different values, those metrics will automatically have new attributes in that dimension (e.g. `m3.medium`). Once your key:value tags are added, you can then slice and dice in any dimension.

What good data looks like

The data you collect should have four characteristics:

— **Well-understood.** You should be able to quickly determine how each metric or event was captured and what it represents. During an outage you don't want to spend time figuring out what your data means. Keep your metrics and events as simple as possible, use standard concepts described above, and name them clearly.

— **Granular.** If you collect metrics too infrequently or average values over long windows of time, you may lose important information about system behavior. For example, periods of 100% resource utilization will be obscured if they are averaged with periods of lower utilization. Collect metrics for each system at a frequency that will not conceal problems, without collecting so often that monitoring becomes perceptibly taxing on the system or samples time intervals that are too short to be meaningful.

— **Tagged by scope.** Each of your hosts operates simultaneously in multiple scopes, and you may want to check on the aggregate health of any of these scopes, or their combinations. For example: how is the production web application doing in aggregate? How about production in the AWS region "us-east-1?" How about a particular combination of software version and EC2 instance type? It is important to retain the multiple scopes associated with your data so that you can alert on problems from any scope, and quickly investigate outages without being limited by a fixed hierarchy of hosts. As described above, this is especially crucial for dynamic cloud infrastructure.

— **Long-lived.** If you discard data too soon, or if after a period of time your monitoring system aggregates your metrics to reduce storage costs, then you lose important information about what happened in the past. Retaining your raw data for a year or more makes it much easier to know what "normal" is, especially if your metrics have monthly, seasonal, or annual variations.

Collect ’em all

Now that we have explored the difference between events and metrics, and the further difference between work metrics and resource metrics, we will see in the next chapter how those data points can be effectively harnessed to monitor your dynamic infrastructure. But first, a brief recap of the key points in this chapter:

- Instrument everything and collect as many work metrics, resource metrics, and events as you reasonably can.

- Collect metrics with sufficient granularity to make important spikes and dips visible. The specific granularity depends on the system you are measuring, the cost of measuring and a typical duration between changes in metrics.

- To maximize the value of your data, tag metrics and events with the appropriate scopes, and retain them at full granularity for at least a year.

Chapter 3: Alerting on What Matters

Automated alerts are essential to monitoring. They allow you to spot problems anywhere in your infrastructure, so that you can rapidly identify their causes and minimize service degradation and disruption.

An alert should communicate something specific about your systems in plain language: "Two Cassandra nodes are down" or "90% of all web requests are taking more than 0.5s to process and respond." Automating alerts across as many of your systems as possible allows you to respond quickly to issues and provide better service, and it also saves time by freeing you from continual manual inspection of metrics.

But alerts aren't always as effective as they could be. In particular, real problems are often lost in a sea of noisy alarms. This chapter describes a simple approach to effective alerting, regardless of the scale and elasticity of the systems involved.
In short:

1. Page on symptoms, rather than causes
2. Alert liberally; page judiciously

Levels of Alerting Urgency

Not all alerts carry the same degree of urgency. Some require immediate human intervention, some require eventual human intervention, and some point to areas where attention may be needed in the future. All alerts should, at a minimum, be recorded in an easily accessible central location so they can be correlated with other metrics and events.

ALERTS AS RECORDS (LOW SEVERITY)

Many alerts will not be associated with a service problem, so a human may never even need to be aware of them. For instance, when a data store that supports a user-facing service starts serving queries much slower than usual, but not slow enough to make an appreciable difference in the overall service's response time, that should generate a low-urgency alert that is recorded in your monitoring system for future reference or investigation but does not interrupt anyone's work. After all, transient issues that could be to blame, such as network congestion, often go away on their own. But should a significant issue develop — say, if the service starts returning a large number of timeouts — that recorded alert will provide invaluable context for your investigation.

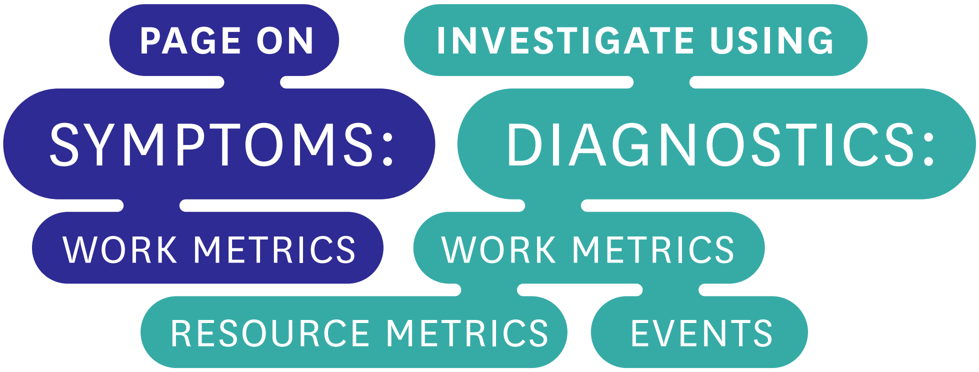
ALERTS AS NOTIFICATIONS (MODERATE SEVERITY)

The next tier of alerting urgency is for issues that do require intervention, but not right away. Perhaps the data store is running low on disk space and should be scaled out in the next several days. Sending an email or posting a notification in the service owner's chat room is a perfect way to deliver these alerts — both message types are highly visible, but they won't wake anyone in the middle of the night or disrupt an engineer's flow.

ALERTS AS PAGES (HIGH SEVERITY)

The most urgent alerts should receive special treatment and be escalated to a page (as in "pager") to urgently request human attention. Response times for your web application, for instance, should have an internal SLA that is at least as aggressive as your strictest customer-facing SLA. Any instance of response times exceeding your internal SLA would warrant immediate attention, whatever the hour.

Data for Alerts, Data for Diagnostics



The table below maps examples of the different data types described in the previous chapter to different levels of alerting urgency. Note that depending on severity, a notification may be more appropriate than a page, or vice versa:

DATA	ALERT	TRIGGER
WORK METRIC: THROUGHPUT	PAGE	VALUE IS MUCH HIGHER OR LOWER THAN USUAL, OR THERE IS AN ANOMALOUS RATE OF CHANGE
WORK METRIC: SUCCESS	PAGE	THE PERCENTAGE OF WORK THAT IS SUCCESSFULLY PROCESSED DROPS BELOW A THRESHOLD
WORK METRIC: ERRORS	PAGE	THE ERROR RATE EXCEEDS A THRESHOLD
WORK METRIC: PERFORMANCE	PAGE	WORK TAKES TOO LONG TO COMPLETE (E.G., PERFORMANCE VIOLATES INTERNAL SLA)
RESOURCE METRIC: UTILIZATION	NOTIFICATION	APPROACHING CRITICAL RESOURCE LIMIT (E.G., FREE DISK SPACE DROPS BELOW A THRESHOLD)
RESOURCE METRIC: SATURATION	RECORD	NUMBER OF WAITING PROCESSES EXCEEDS A THRESHOLD
RESOURCE METRIC: ERRORS	RECORD	NUMBER OF INTERNAL ERRORS DURING A FIXED PERIOD EXCEEDS A THRESHOLD
RESOURCE METRIC: AVAILABILITY	RECORD	THE RESOURCE IS UNAVAILABLE FOR A PERCENTAGE OF TIME THAT EXCEEDS A THRESHOLD
EVENT: WORK-RELATED	PAGE	CRITICAL WORK THAT SHOULD HAVE BEEN COMPLETED IS REPORTED AS INCOMPLETE OR FAILED

WHEN TO LET A SLEEPING ENGINEER LIE

Whenever you consider setting an alert, ask yourself three questions to determine the alert's level of urgency and how it should be handled:

1. **Is this issue real?**
   It may seem obvious, but if the issue is not real, it usually should not generate an alert. The examples below can trigger alerts but probably are not symptomatic of real problems. Sending visible alerts or pages on occurrences such as these contributes to alert fatigue and can cause more serious issues to be overlooked:

   — Metrics in a test environment are out of bounds

   — A single server is doing its work very slowly, but it is part of a cluster with fast-failover to other machines, and it reboots periodically anyway

   — Planned upgrades are causing large numbers of machines to report as offline

   If the issue is indeed **real**, it should generate an alert. Even if the alert is not linked to a notification, it should be recorded within your monitoring system for later analysis and correlation.

2. **Does this issue require attention?**
   If you can reasonably automate a response to an issue, you should consider doing so. There is a very real cost to calling someone away from work, sleep, or personal time. If the issue is **real and it requires attention**, it should generate an alert that notifies someone who can investigate and fix the problem. At minimum, the notification should be sent via email, chat or a ticketing system so that the recipients can prioritize their response.

3. **Is this issue urgent?**
   Not all issues are emergencies. For example, perhaps a moderately higher than normal percentage of system responses have been very slow, or perhaps a slightly elevated share of queries are returning stale data. Both issues may need to be addressed soon, but not at 4:00 A.M. If, on |the other hand, a key system stops doing its work at an acceptable rate, an engineer should take a look immediately. If the symptom is **real and it requires attention and it is urgent**, it should generate a page.

PAGE ON SYMPTOMS

Pages deserve special mention: they are extremely effective for delivering information, but they can be quite disruptive if overused, or if they are linked to alerts that are prone to flapping. In general, a page is the most appropriate kind of alert when the system you are responsible for stops doing useful work with acceptable throughput, latency, or error rates. Those are the sort of problems that you want to know about immediately.

The fact that your system stopped doing useful work is a *symptom*. It is a manifestation of an issue that may have any number of different *causes*. For example: if your website has been responding very slowly for the last three minutes, that is a symptom. Possible causes include high database latency, failed application servers, Memcached being down, high load, and so on. Whenever possible, build your pages around symptoms rather than causes. The distinction between work metrics and resource metrics introduced in chapter 2 is often useful for separating symptoms and causes: work metrics are usually associated with symptoms and resource metrics with causes.

Paging on symptoms surfaces real, oftentimes user-facing problems, rather than hypothetical or internal problems. Contrast paging on a symptom, such as slow website responses, with paging on potential causes of the symptom, such as high load on your web servers. Your users will not know or care about server load if the website is still responding quickly, and your engineers will resent being bothered for something that is only internally noticeable and that may revert to normal levels without intervention.

DURABLE ALERT DEFINITIONS

Another good reason to page on symptoms is that symptom-triggered alerts tend to be durable. This means that regardless of how underlying system architectures may change, if the system stops doing work as well as it should, you will get an appropriate page even without updating your alert definitions.

EXCEPTION TO THE RULE: EARLY WARNING SIGNS

It is sometimes necessary to call human attention to a small handful of metrics even when the system is performing adequately. Early warning metrics reflect an unacceptably high probability that serious symptoms will soon develop and require immediate intervention.

Disk space is a classic example. Unlike running out of free memory or CPU, when you run out of disk space, the system will not likely recover, and you probably will have only a few seconds before your system hard stops. Of course, if you can notify someone with plenty of lead time, then there is no need to wake anyone in the middle of the night. Better yet, you can anticipate some situations when disk space will run low and build automated remediation based on the data you can afford to erase, such as logs or data that exists somewhere else.

Get Serious About Symptoms

In the next chapter we'll cover what to do once you receive an alert. But first, a quick roundup of the key points in this chapter:

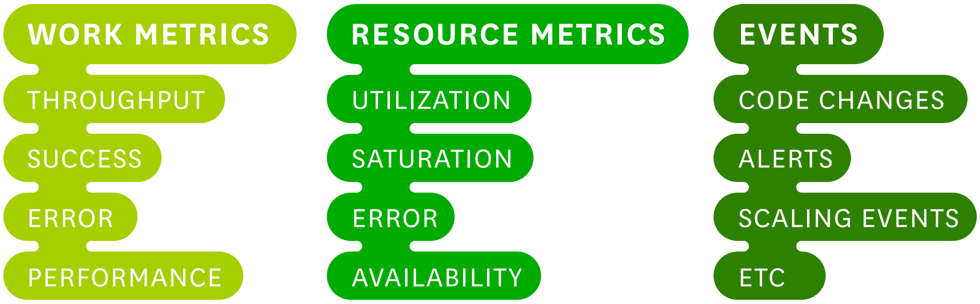
- Send a page only when symptoms of urgent problems in your system's work are detected, or if a critical and finite resource limit is about to be reached.

- Set up your monitoring system to record alerts whenever it detects real issues in your infrastructure, even if those issues have not yet affected overall performance.

Chapter 4: Investigating Performance Issues

The responsibilities of a monitoring system do not end with symptom detection. Once your monitoring system has notified you of a real symptom that requires attention, its job is to help you diagnose the root cause. Often this is the least structured aspect of monitoring, driven largely by hunches and guess-and-check. This chapter describes a more directed approach that can help you to find and correct root causes more efficiently.

A Brief Data Refresher



WORK METRICS
THROUGHPUT
SUCCESS
ERROR
PERFORMANCE

RESOURCE METRICS
UTILIZATION
SATURATION
ERROR
AVAILABILITY

EVENTS
CODE CHANGES
ALERTS
SCALING EVENTS
ETC

As you'll recall from chapter 2, there are three main types of monitoring data that can help you investigate the root causes of problems in your infrastructure:

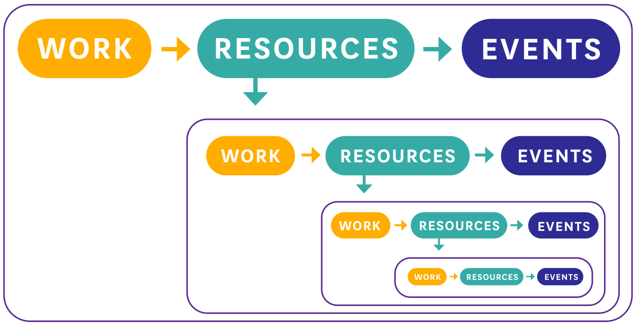
- **Work metrics** indicate the top-level health of your system by measuring its useful output
- **Resource metrics** quantify the utilization, saturation, errors, or availability of a resource that your system depends on
- **Events** describe discrete, infrequent occurrences in your system such as code changes, internal alerts, and scaling events

By and large, work metrics will surface the most serious symptoms and should therefore generate the most serious alerts, as discussed in the previous chapter. But the other metric types are invaluable for investigating the causes of those symptoms.

IT'S RESOURCES ALL THE WAY DOWN

Most of the components of your infrastructure can be thought of as resources. At the highest levels, each of your systems that produces useful work likely relies on other systems. For instance, the Apache server in a LAMP stack relies on a MySQL database as a resource to support its work of serving requests. One level down, MySQL has unique resources that the database uses to do *its work*, such as the finite pool of client connections. At a lower level still are the physical resources of the server running MySQL, such as CPU, memory, and disks.

Thinking about which systems *produce* useful work, and which resources *support* that work, can help you to efficiently get to the root of any issues that surface. When an alert notifies you of a possible problem, the following process will help you to approach your investigation systematically.



1. **Start at the top with work metrics**
   First ask yourself, “Is there a problem? How can I characterize it?” If you don’t describe the issue clearly at the outset, it’s easy to lose track as you dive deeper into your systems to diagnose the issue.

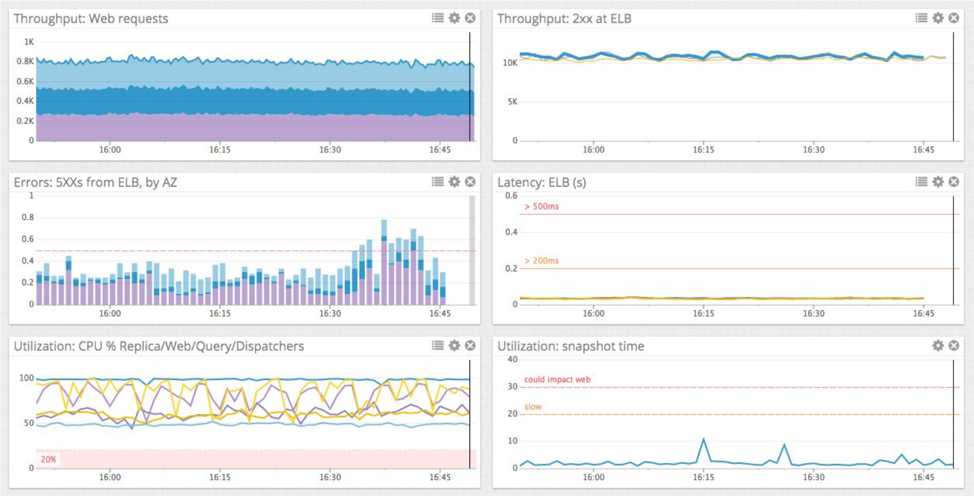
   Next examine the work metrics for the highest-level system that is exhibiting problems. These metrics will often point to the source of the problem, or at least set the direction for your investigation. For example, if the percentage of work that is successfully processed drops below a set threshold, diving into error metrics, and especially the types of errors being returned, will often help narrow the focus of your investigation. Alternatively, if latency is high, and the throughput of work being requested by outside systems is also very high, perhaps the system is simply overburdened.

2. **Dig into resources**
   If you haven’t found the cause of the problem by inspecting top-level work metrics, next examine the resources that the system uses—physical resources as well as software or external services that serve as resources to the system. Setting up dashboards for each system ahead of time, as outlined below, enables you to quickly find and peruse metrics for the relevant resources. Are those resources unavailable? Are they highly utilized or saturated? If so, recurse into those resources and begin investigating each of them at step 1.

3. **Did something change?**
   Next consider alerts and other events that may be correlated with your metrics. If a code release, internal alert, or other event was registered slightly before problems started occurring, investigate whether they may be connected to the problem.

4. **Fix it (and don’t forget it)**
   Once you have determined what caused the issue, correct it. Your investigation is complete when symptoms disappear—you can now think about how to change the system to avoid similar problems in the future.

BUILD DASHBOARDS BEFORE YOU NEED THEM



In an outage, every minute is crucial. To speed your investigation and keep your focus on the task at hand, set up dashboards in advance. You may want to set up one dashboard for your high-level application metrics, and one dashboard for each subsystem. Each system's dashboard should render the work metrics of that system, along with resource metrics of the system itself and key metrics of the subsystems it depends on. If event data is available, overlay relevant events on the graphs for correlation analysis.

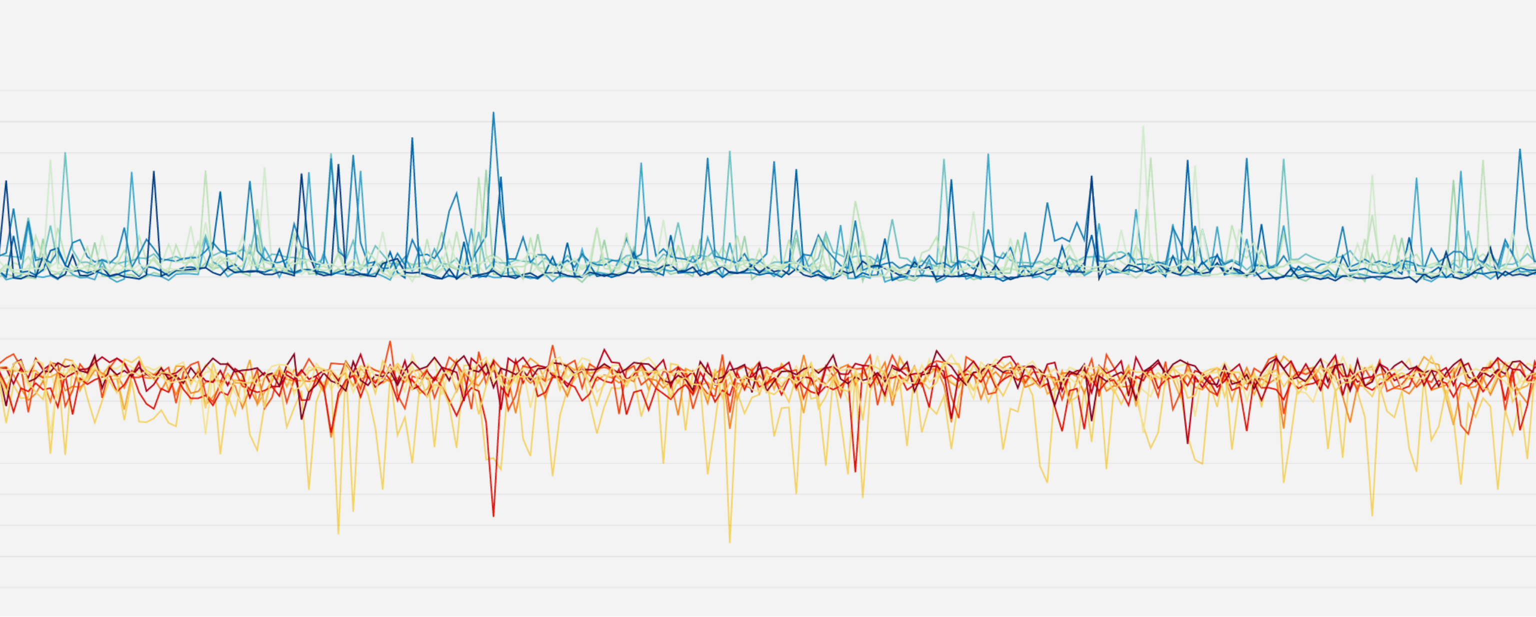
FOLLOW THE METRICS

Adhering to a standardized monitoring framework allows you to investigate problems more systematically:

- For each system in your infrastructure, set up a dashboard ahead of time that displays all its key metrics, with relevant events overlaid.
- Investigate causes of problems by starting with the highest-level system that is showing symptoms, reviewing its work and resource metrics and any associated events.
- If problematic resources are detected, apply the same investigation pattern to the resource (and its constituent resources) until your root problem is discovered and corrected.

We've now stepped through a high-level framework for data collection and tagging (chapter 2), automated alerting (chapter 3), and incident response and investigation (chapter 4). In the next chapter we'll go further into detail on how to monitor your metrics using a variety of graphs and other visualizations.

Chapter 5: Visualizing Metrics with Timeseries Graphs



In order to turn your metrics into actionable insights, it's important to choose the right visualization for your data. There is no one-size-fits-all solution: you can see different things in the same metric with different graph types.

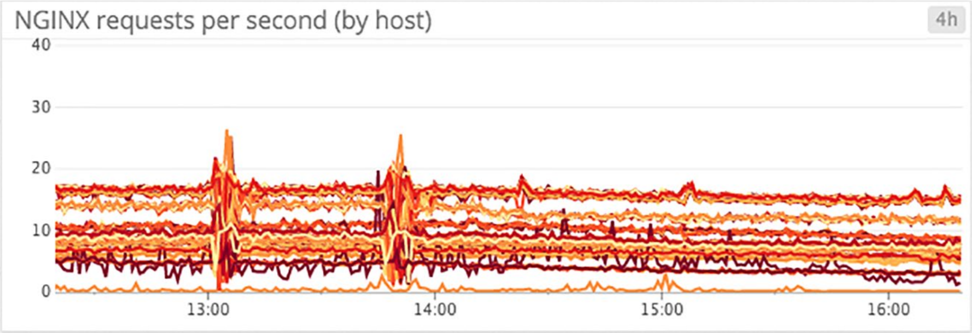
To help you effectively visualize your metrics, this chapter explores four different types of *timeseries graphs*: line graphs, stacked area graphs, bar graphs, and heat maps. These graphs all have time on the x-axis and metric values on the y-axis. For each graph type, we'll explain how it works, when to use it, and when to use something else.

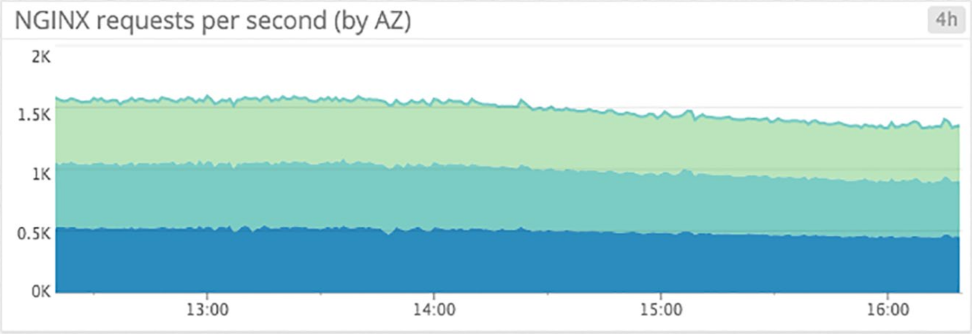
But first we'll quickly touch on aggregation in timeseries graphs, which is critical for visualizing metrics from dynamic, cloud-scale infrastructure.

Aggregation Across Space

Not all metric queries make sense broken out by host, container, or other unit of infrastructure. So you will often need some *aggregation across space* to sensibly visualize your metrics. This aggregation can take many forms: aggregating metrics by messaging queue, by database table, by application, or by some attribute of your hosts themselves (operating system, availability zone, hardware profile, etc.).

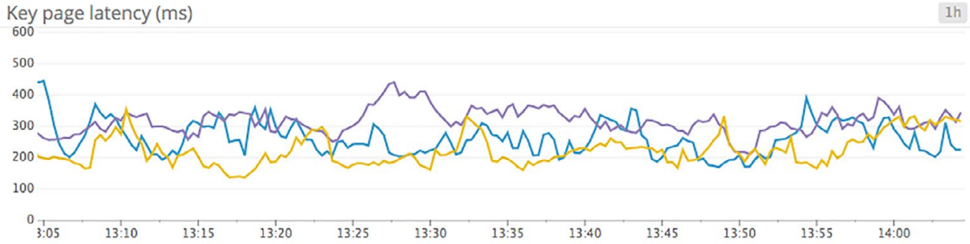
Aggregation across space allows you to slice and dice your infrastructure to isolate exactly the metrics that matter most to you. It also allows you to make otherwise noisy graphs much more readable. For instance, it is hard to make sense of a host-level graph of web requests, but the same data is easily interpreted when the metrics are aggregated by availability zone:





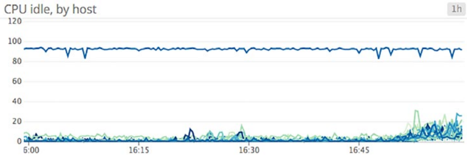
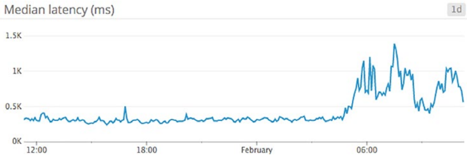
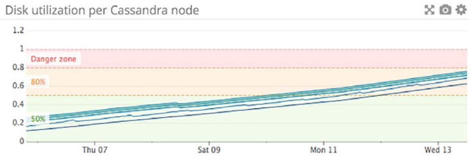
Tagging your metrics, as discussed in chapter 2, makes it easy to perform these aggregations on the fly when you are building your graphs and dashboards.

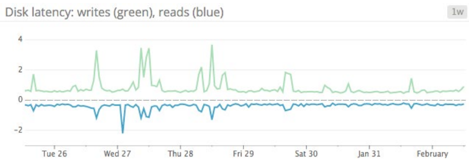
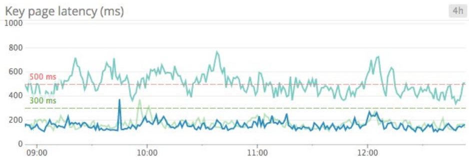
Line Graphs



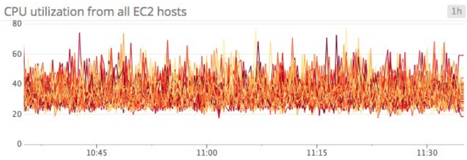
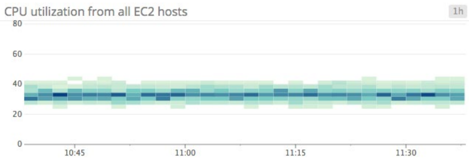
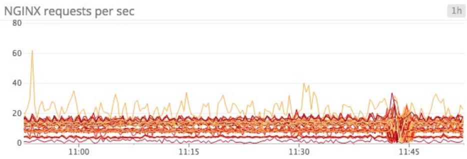
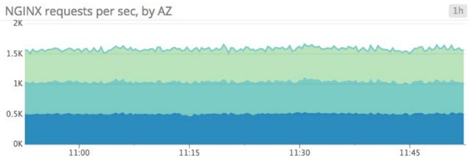
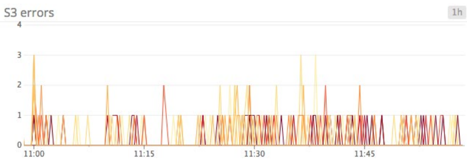
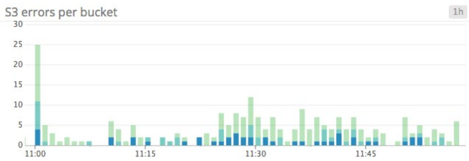
Line graphs are the simplest way to translate metric data into visuals, but often they're used by default when a different graph would be more appropriate. For instance, a graph of wildly fluctuating metrics from hundreds of hosts quickly becomes harder to disentangle than steel wool.

WHEN TO USE LINE GRAPHS

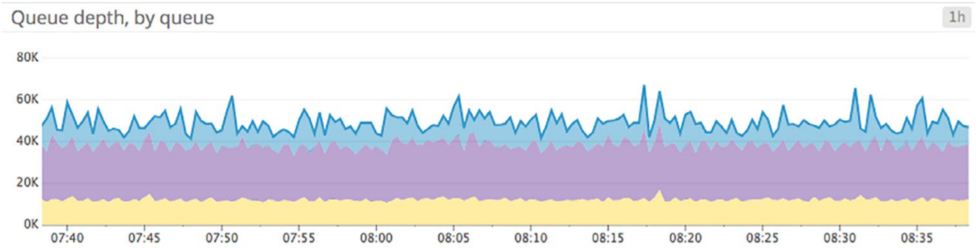
WHAT	WHY	EXAMPLE
THE SAME METRIC REPORTED BY **DIFFERENT SCOPES**	TO SPOT OUTLIERS AT A GLANCE	CPU IDLE FOR EACH HOST IN A CLUSTER 
TRACKING **SINGLE METRICS** FROM ONE SOURCE, OR AS AN AGGREGATE	TO CLEARLY COMMUNICATE A KEY METRIC'S EVOLUTION OVER TIME	MEDIAN LATENCY ACROSS ALL WEB SERVERS 
METRICS FOR WHICH **UNAGGREGATED VALUES** FROM A PARTICULAR SLICE OF YOUR INFRASTRUCTURE ARE ESPECIALLY VALUABLE	TO SPOT INDIVIDUAL DEVIATIONS INTO UNACCEPTABLE RANGES	DISK SPACE UTILIZATION PER DATABASE NODE 

RELATED METRICS SHARING THE SAME UNITS	TO SPOT CORRELATIONS AT A GLANCE	LATENCY FOR DISK READS AND DISK WRITES ON THE SAME MACHINE 
METRICS THAT HAVE A CLEAR **ACCEPTABLE DOMAIN**	TO EASILY SPOT SERVICE DEGRADATIONS	LATENCY FOR PROCESSING WEB REQUESTS 

WHEN TO USE SOMETHING ELSE

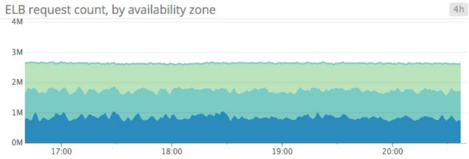
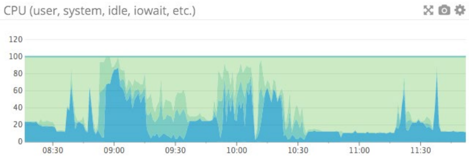
WHAT	EXAMPLE	INSTEAD USE...
HIGHLY VARIABLE METRICS REPORTED BY A **LARGE NUMBER OF SOURCES**	CPU FROM ALL HOSTS  	HEAT MAPS TO MAKE NOISY DATA MORE INTERPRETABLE 
METRICS THAT ARE **MORE ACTIONABLE AS AGGREGATES** THAN AS SEPARATE DATA POINTS	WEB REQUESTS PER SECOND OVER DOZENS OF WEB SERVERS  	AREA GRAPHS TO AGGREGATE ACROSS TAGGED GROUPS 
SPARSE METRICS	COUNT OF RELATIVELY RARE S3 ACCESS ERRORS  	BAR GRAPHS TO AVOID JUMPY INTERPOLATIONS 

Stacked Area Graphs

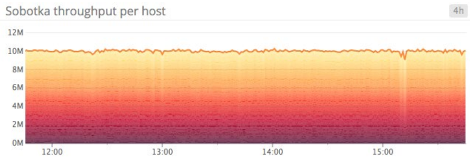
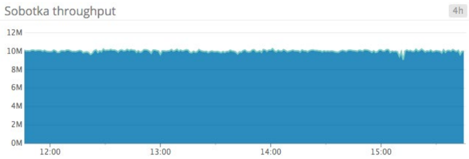
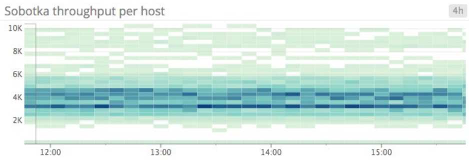
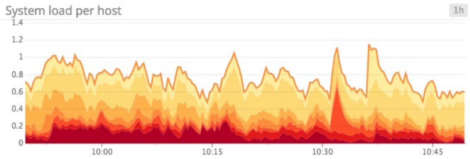
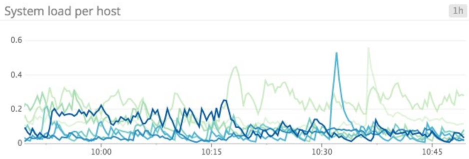


Area graphs are similar to line graphs, except the metric values are represented by two-dimensional bands rather than lines. Multiple timeseries can be summed together simply by stacking the bands.

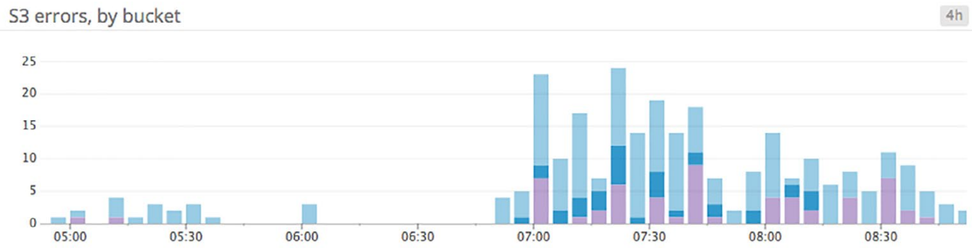
WHEN TO USE STACKED AREA GRAPHS

WHAT	WHY	EXAMPLE
THE SAME METRIC FROM **DIFFERENT SCOPES**, STACKED	TO CHECK BOTH THE SUM AND THE CONTRIBUTION OF EACH OF ITS PARTS AT A GLANCE	LOAD BALANCER REQUESTS PER AVAILABILITY ZONE 
SUMMING **COMPLEMENTARY METRICS** THAT SHARE THE SAME UNIT	TO SEE HOW A FINITE RESOURCE IS BEING UTILIZED	CPU UTILIZATION METRICS (USER, SYSTEM, IDLE, ETC.) 

WHEN TO USE SOMETHING ELSE

WHAT	EXAMPLE	INSTEAD USE...
UNAGGREGATED METRICS FROM LARGE NUMBERS OF HOSTS, MAKING THE SLICES TOO THIN TO BE MEANINGFUL	THROUGHPUT METRICS ACROSS HUNDREDS OF APP SERVERS  	LINE GRAPH OR SOLID-COLOR AREA GRAPH TO TRACK TOTAL, AGGREGATE VALUE  HEAT MAPS TO TRACK HOST-LEVEL DATA 
METRICS THAT **CAN'T BE ADDED** SENSIBLY	SYSTEM LOAD ACROSS MULTIPLE SERVERS  	LINE GRAPHS, OR HEAT MAPS FOR LARGE NUMBERS OF HOSTS 

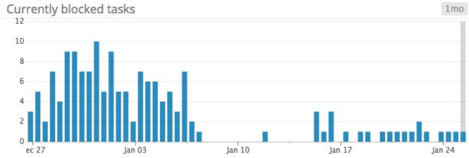
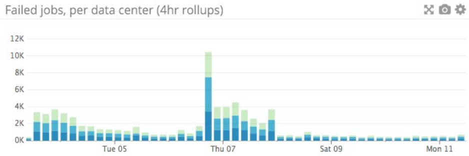
Bar Graphs



In a bar graph, each bar represents a metric rollup over a time interval. This feature makes bar graphs ideal for representing counts. Unlike *gauge metrics*, which represent an instantaneous value, *count* metrics only make sense when paired with a time interval (e.g., 13 query errors in the past five minutes).

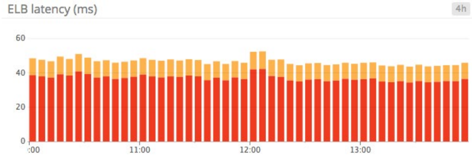
Bar graphs require no interpolation to connect one interval to the next, making them especially useful for representing sparse metrics. Like area graphs, they naturally accommodate stacking and summing of metrics.

WHEN TO USE BAR GRAPHS

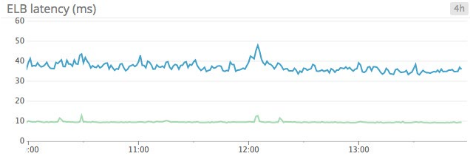
WHAT	WHY	EXAMPLE
SPARSE METRICS	TO CONVEY METRIC VALUES WITHOUT JUMPY OR MISLEADING INTERPOLATIONS	BLOCKED TASKS IN CASSANDRA'S INTERNAL QUEUES 
METRICS THAT REPRESENT A **COUNT** (RATHER THAN A GAUGE)	TO CONVEY BOTH THE TOTAL COUNT AND THE CORRESPONDING TIME INTERVAL	FAILED JOBS, BY DATA CENTER (4-HOUR INTERVALS) 

WHEN TO USE SOMETHING ELSE

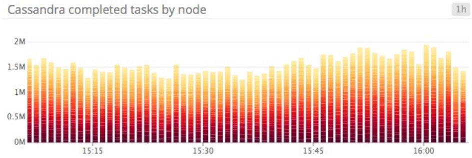
<table>
<tr><th>WHAT</th><th>EXAMPLE</th><th>INSTEAD USE...</th></tr>
<tr>
<td>METRICS THAT <b>CAN'T BE ADDED</b> SENSIBLY</td>
<td>AVERAGE LATENCY PER LOAD BALANCER



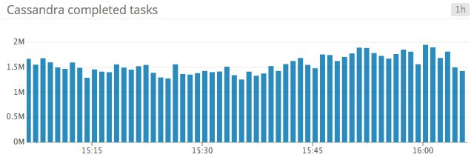
</td>
<td>LINE GRAPHS TO ISOLATE TIMESERIES FROM EACH HOST



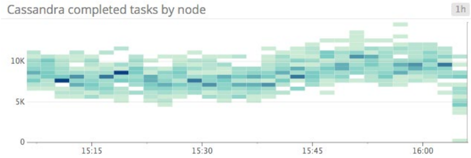
</td>
</tr>
<tr>
<td><b>UNAGGREGATED METRICS</b> FROM LARGE NUMBERS OF SOURCES, MAKING THE SLICES TOO THIN TO BE MEANINGFUL</td>
<td>COMPLETED TASKS ACROSS DOZENS OF CASSANDRA NODES



</td>
<td>SOLID-COLOR BARS TO TRACK TOTAL, AGGREGATE METRIC VALUE

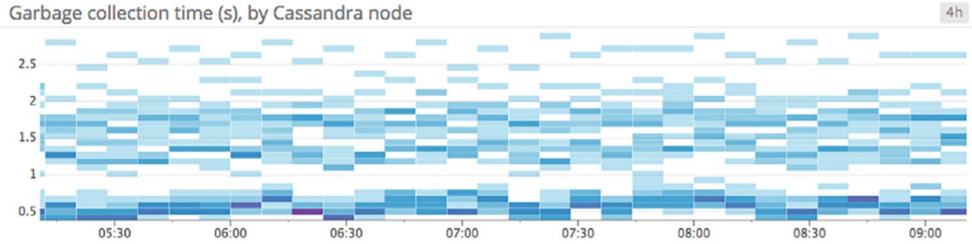


HEAT MAPS TO TRACK HOST-LEVEL VALUES



</td>
</tr>
</table>

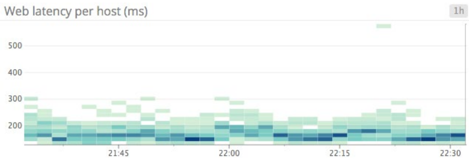
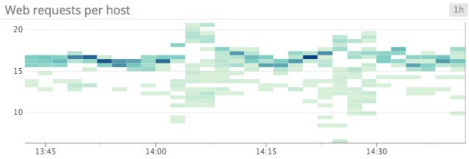
Heat Maps



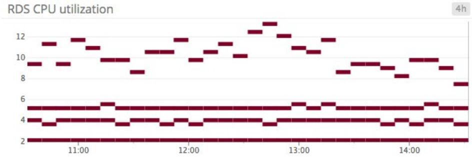
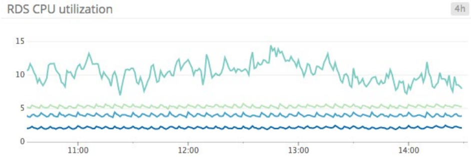
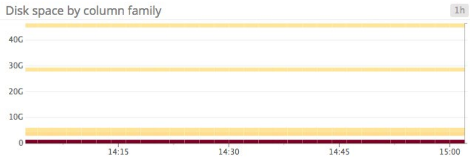
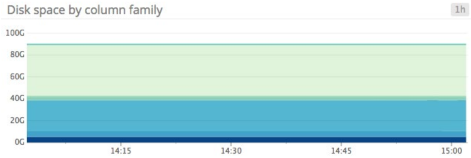
Heat maps show the distribution of values for a metric evolving over time. Specifically, each column represents a distribution of values during a particular time slice. Each cell's shading corresponds to the number of entities reporting that particular value during that particular time.

Heat maps are designed to visualize metrics from large numbers of entities, so they are often used to graph unaggregated metrics at the individual host or container level. Heat maps are closely related to distribution graphs, except that heat maps show change over time, and distribution graphs are a snapshot of a particular window of time. Distributions are covered in the following chapter.

WHEN TO USE HEAT MAPS

WHAT	WHY	EXAMPLE
SINGLE METRIC REPORTED BY A LARGE NUMBER OF GROUPS	TO CONVEY GENERAL TRENDS AT A GLANCE	WEB LATENCY PER HOST 
	TO SEE TRANSIENT VARIATIONS ACROSS MEMBERS OF A GROUP	REQUESTS RECEIVED PER HOST 

WHEN TO USE SOMETHING ELSE

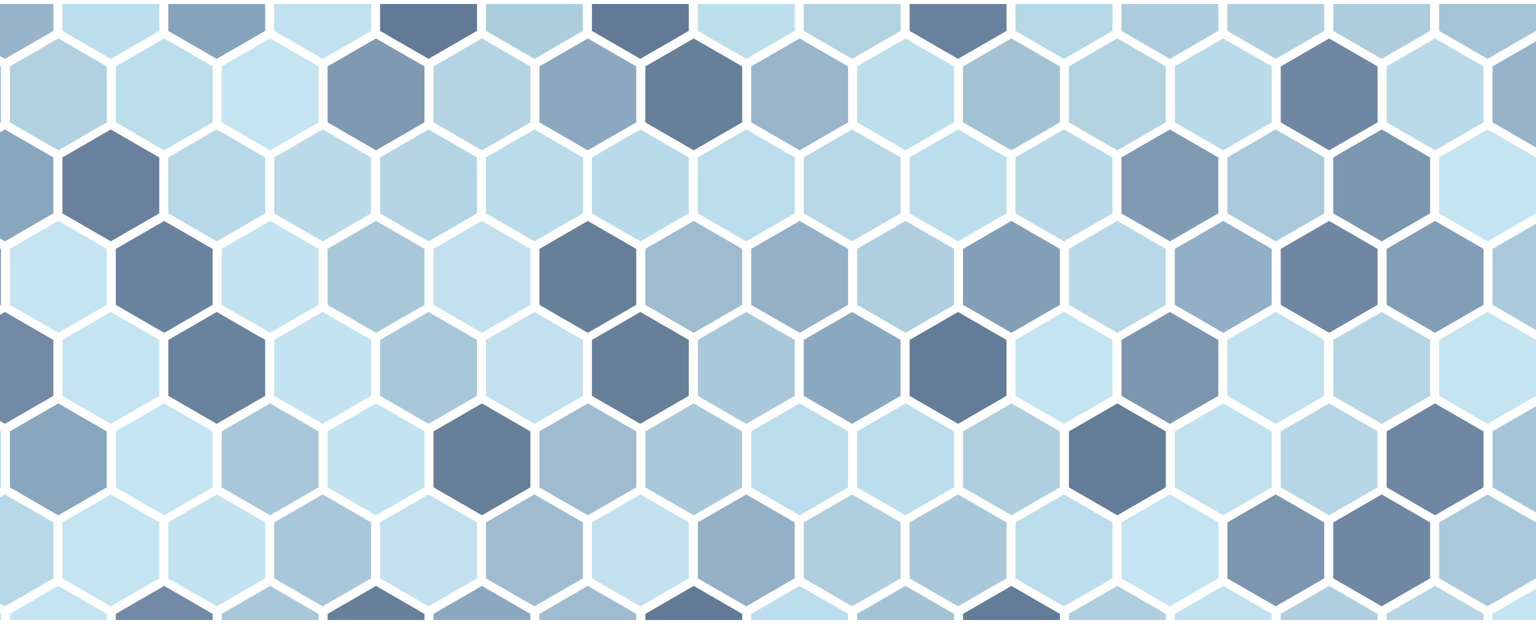
WHAT	WHY	EXAMPLE
METRICS COMING FROM ONLY A **FEW INDIVIDUAL SOURCES**	CPU UTILIZATION ACROSS A SMALL NUMBER OF RDS INSTANCES 	LINE GRAPHS TO ISOLATE TIMESERIES FROM EACH HOST 
METRICS WHERE **AGGREGATES MATTER MORE** THAN INDIVIDUAL VALUES	DISK UTILIZATION PER CASSANDRA COLUMN FAMILY 	AREA GRAPHS TO SUM VALUES ACROSS A SET OF TAGS 

Know Your Graphs

By understanding the ideal use cases and limitations of each kind of timeseries graph, you can extract actionable information from your metrics more quickly.

In the following chapter, we'll explore summary graphs, which are visualizations that compress time out of view to display a summary view of your metrics.

Chapter 6: Visualizing Metrics with Summary Graphs



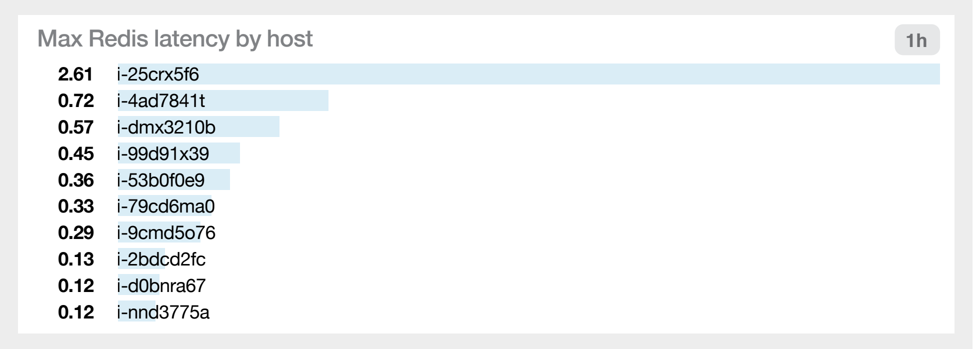
In chapter 5, we discussed timeseries graphs — visualizations that show infrastructure metrics evolving through time. In this post we cover summary graphs, which are visualizations that **flatten** a particular span of time to provide a summary window into your infrastructure. The summary graph types covered in this chapter are: single-value summaries, toplists, change graphs, host maps, and distributions.

For each graph type, we'll explain how it works and when to use it. But first, we'll quickly discuss two concepts that are necessary to understand infrastructure summary graphs: aggregation across time (which you can think of as "time flattening" or "snapshotting"), and aggregation across space.

Aggregation Across Time

To provide a summary view of your metrics, a visualization must flatten a timeseries into a single value by compressing the time dimension out of view. This *aggregation* *across time* can mean simply displaying the latest value returned by a metric query, or a more complex aggregation to return a computed value over a moving time window.

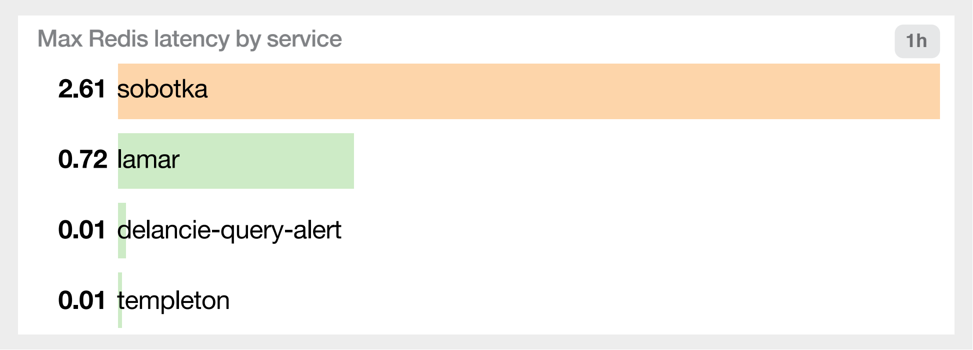
For example, instead of displaying only the latest reported value for a metric query, you may want to display the maximum value reported by each host over the past 60 minutes to surface problematic spikes:

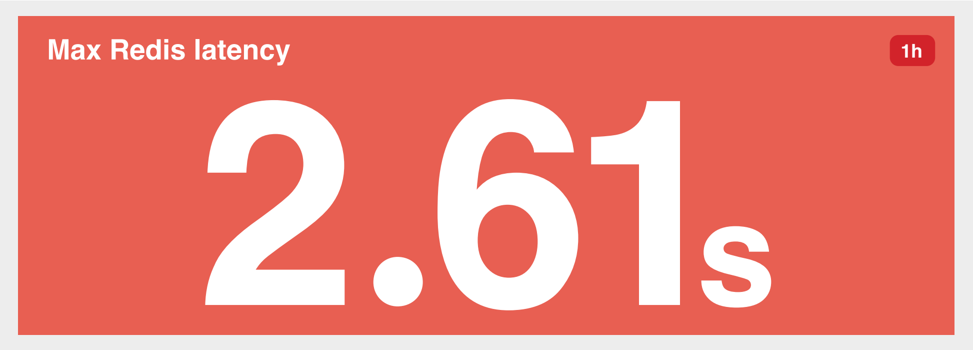


Aggregation Across Space

As disussed in chapter 5, you will often need some *aggregation across space* to sensibly visualize your metrics. This can mean aggregating by some property of your hosts (availability zone, instance type, operating system) or by tags applied to your metrics (service name, messaging queue, database table, etc.).

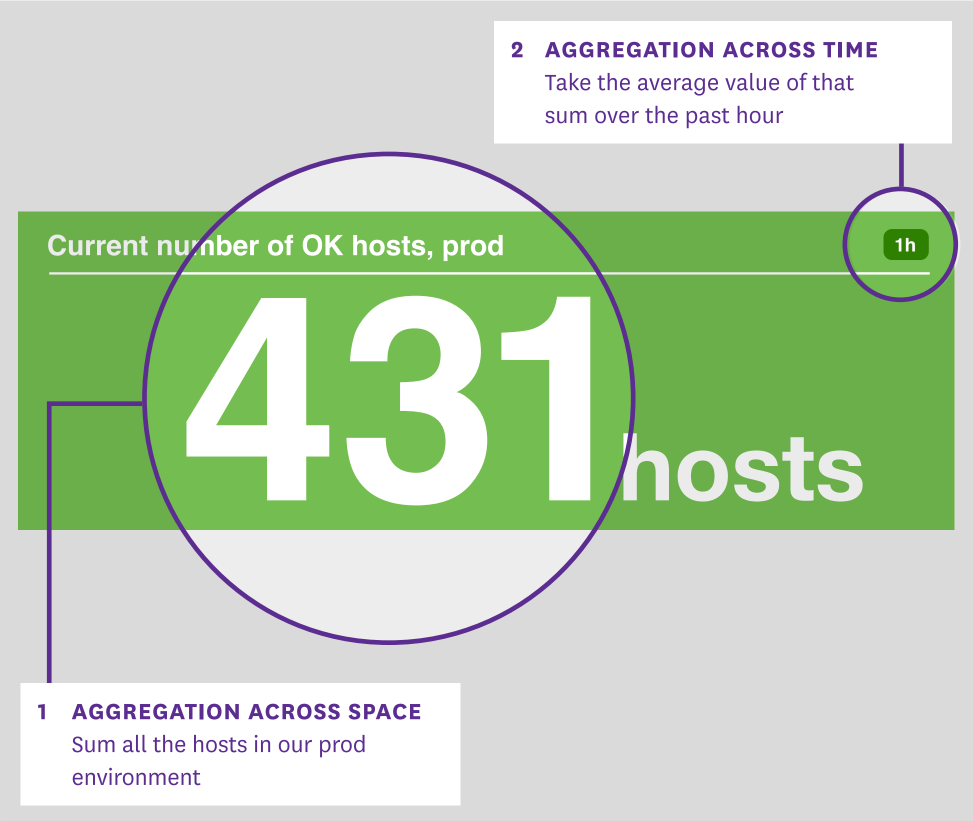
Instead of listing peak Redis latencies at the host level as in the example pictured above, it may be more useful to see peak latencies for each internal service that is built on Redis. Or you could surface only the maximum value reported by any one host in your infrastructure:



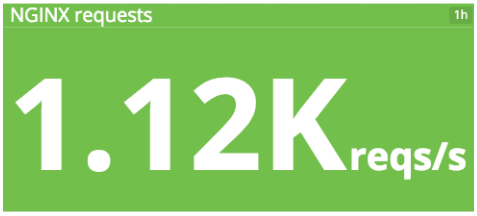
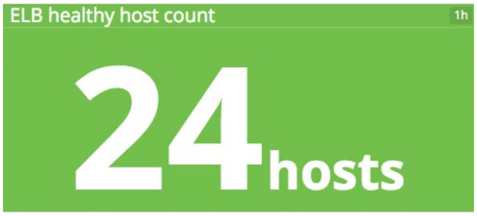
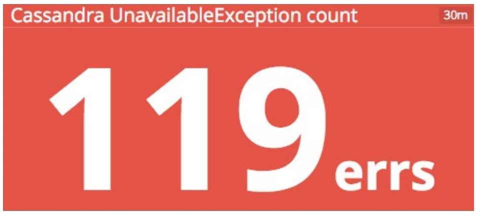
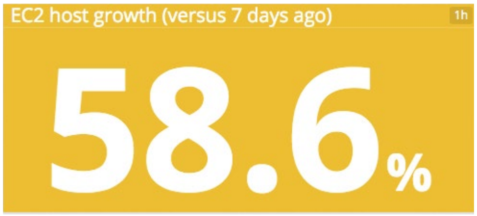


Single-Value Summaries

Single-value summaries display the current value of a given metric query, with conditional formatting (such as a green/yellow/red background) to convey whether or not the value is in the expected range. The value displayed by a single-value summary need not represent an instantaneous measurement. The widget can display the latest value reported, or an aggregate computed from all query values across the time window.


2 AGGREGATION ACROSS TIME
Take the average value of that sum over the past hour
Current number of OK hosts, prod
1h
431 hosts
1 AGGREGATION ACROSS SPACE
Sum all the hosts in our prod environment

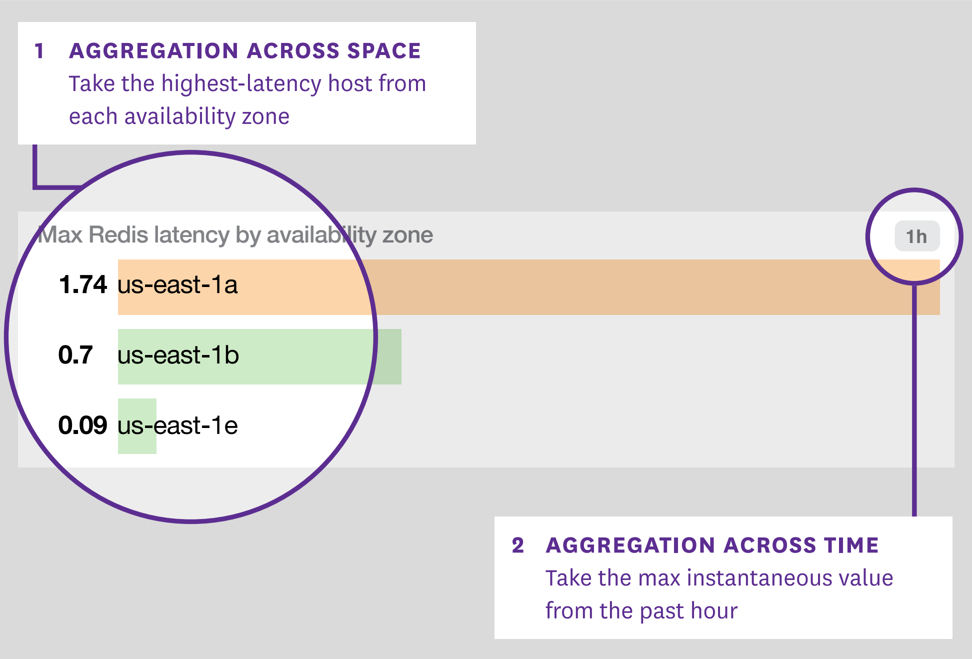
WHEN TO USE SINGLE-VALUE SUMMARIES

WHAT	EXAMPLE	EXAMPLE
WORK METRICS FROM A GIVEN SYSTEM	TO MAKE KEY METRICS IMMEDIATELY VISIBLE	WEB SERVER REQUESTS PER SECOND  NGINX requests 1h 1.12K reqs/s
CRITICAL **RESOURCE METRICS**	TO PROVIDE AN OVERVIEW OF RESOURCE STATUS AND HEALTH AT A GLANCE	HEALTHY HOSTS BEHIND LOAD BALANCER  ELB healthy host count 1h 24 hosts
ERROR METRICS	TO QUICKLY DRAW ATTENTION TO POTENTIAL PROBLEMS	FATAL DATABASE EXCEPTIONS  Cassandra UnavailableException count 30m 119 errs
COMPUTED **METRIC CHANGES** AS COMPARED TO PREVIOUS VALUES	TO COMMUNICATE KEY TRENDS CLEARLY	HOSTS IN USE VERSUS ONE WEEK AGO  EC2 host growth (versus 7 days ago) 1h 58.6%

Toplists

Toplists are ordered lists that allow you to rank hosts, clusters, or any other segment of your infrastructure by their metric values. Because they are so easy to interpret, toplists are especially useful in high-level status boards.

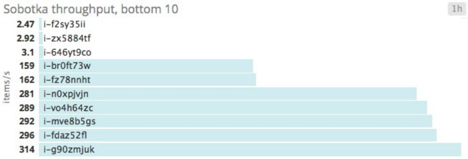
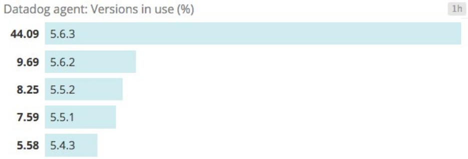
Compared to single-value summaries, toplists have an additional layer of aggregation across space, in that the value of the metric query is broken out by group. Each group can be a single host or an aggregation of related hosts.



1 AGGREGATION ACROSS SPACE
Take the highest-latency host from each availability zone

2 AGGREGATION ACROSS TIME
Take the max instantaneous value from the past hour

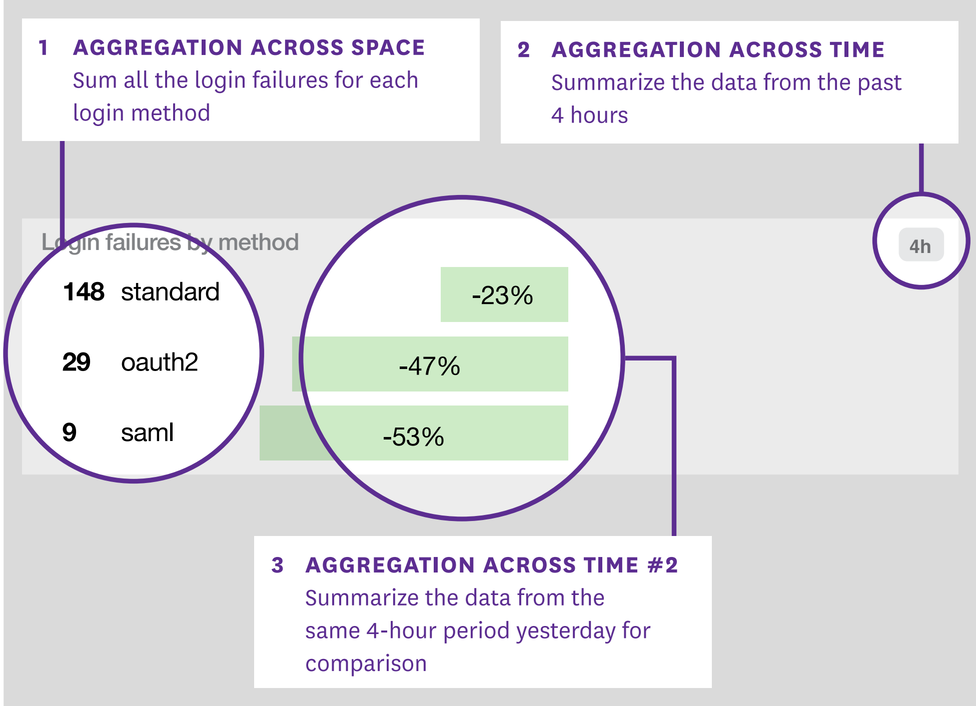
WHEN TO USE TOPLISTS

WHAT	WHY	EXAMPLE
WORK OR RESOURCE METRICS TAKEN FROM DIFFERENT HOSTS OR GROUPS	TO SPOT OUTLIERS, UNDERPERFORMERS, OR RESOURCE OVERCONSUMERS AT A GLANCE	POINTS PROCESSED PER APP SERVER 
CUSTOM METRICS RETURNED AS A LIST OF VALUES	TO CONVEY KPIS IN AN EASY-TO-READ FORMAT (E.G. FOR STATUS BOARDS ON WALL-MOUNTED DISPLAYS)	VERSIONS OF THE DATADOG AGENT IN USE 

Change Graphs

Whereas toplists give you a summary of recent metric values, change graphs compare a metric's current value against its value at a point in the past.

The key difference between change graphs and other visualizations is that change graphs take two different timeframes as parameters: one for the size of the evaluation window and one to set the lookback window.

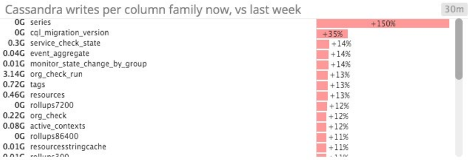


1 **AGGREGATION ACROSS SPACE**
Sum all the login failures for each login method

2 **AGGREGATION ACROSS TIME**
Summarize the data from the past 4 hours

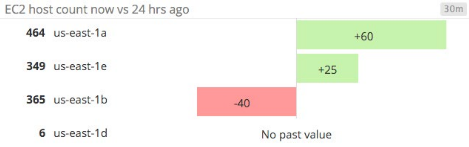
3 **AGGREGATION ACROSS TIME #2**
Summarize the data from the same 4-hour period yesterday for comparison

WHEN TO USE CHANGE GRAPHS

WHAT	WHY	EXAMPLE
CYCLIC METRICS THAT RISE AND FALL DAILY, WEEKLY, OR MONTHLY	TO SEPARATE GENUINE TRENDS FROM PERIODIC BASELINES	DATABASE WRITE THROUGHPUT, COMPARED TO SAME TIME LAST WEEK 

Cassandra writes per column family now, vs last week 30m

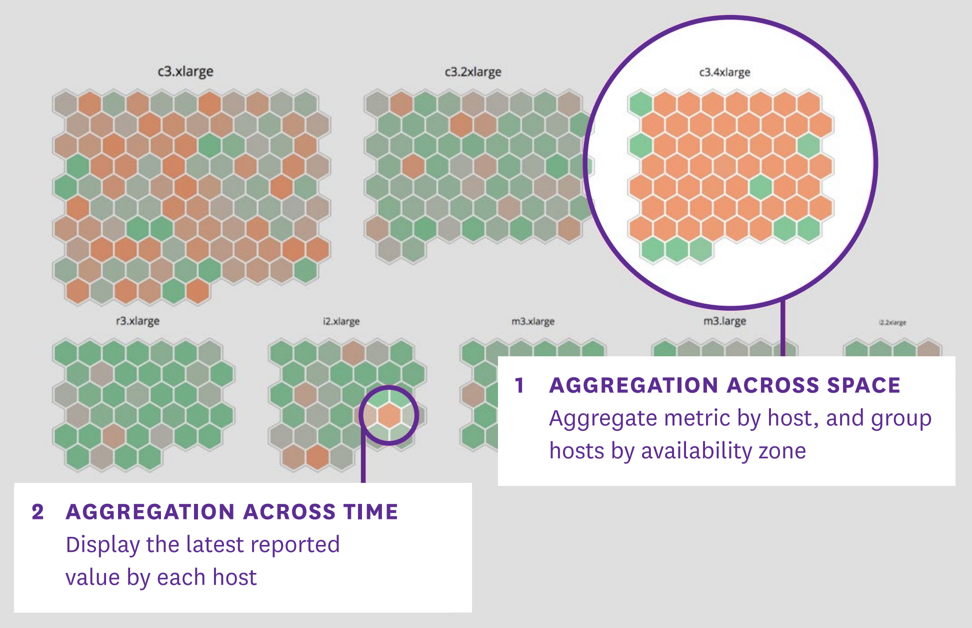
0G	series	+150%
0G	cql_migration_version	+35%
0.3G	service_check_state	+14%
0.04G	event_aggregate	+14%
0.01G	monitor_state_change_by_group	+14%
3.14G	org_check_run	+13%
0.72G	tags	+13%
0.46G	resources	+13%
0G	rollups7200	+12%
0.22G	org_check	+12%
0.08G	active_contexts	+12%
0G	rollups86400	+11%
0.01G	resourcesstringcache	+11%
0.01G	rollups300	+11%

HIGH-LEVEL INFRASTRUCTURE METRICS	TO QUICKLY IDENTIFY LARGE-SCALE TRENDS	TOTAL HOST COUNT, COMPARED TO SAME TIME YESTERDAY 

Host Maps

Host maps are a unique way to see your entire infrastructure, or any slice of it, at a glance. However you slice and dice your infrastructure (by availability zone, by service name, by instance type, etc.), you will see each host in the selected group as a hexagon, color-coded and sized by any metrics reported by those hosts.

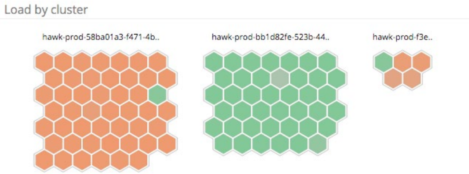
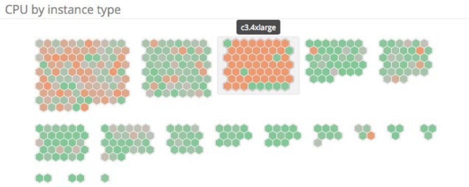
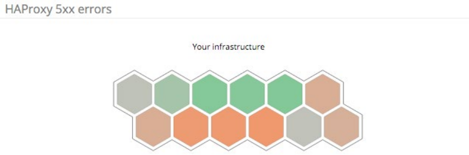
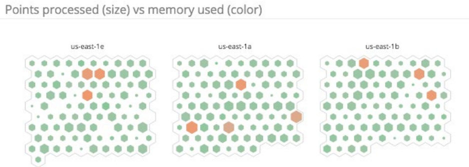
This particular visualization type is unique to Datadog. As such, it is specifically designed for infrastructure monitoring, in contrast to the general-purpose visualizations described elsewhere in this chapter.



1 AGGREGATION ACROSS SPACE
Aggregate metric by host, and group hosts by availability zone

2 AGGREGATION ACROSS TIME
Display the latest reported value by each host

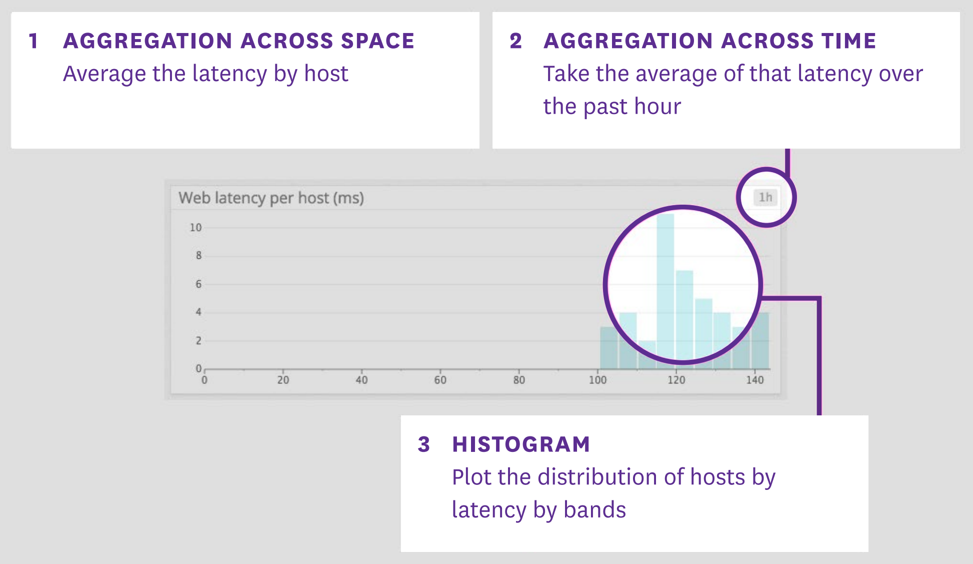
WHEN TO USE HOST MAPS

WHAT	EXAMPLE	EXAMPLE
RESOURCE UTILIZATION METRICS	TO SPOT OVERLOADED COMPONENTS AT A GLANCE	 Load by cluster hawk-prod-58ba01a3-f471-4b.. hawk-prod-bb1d82fe-523b-44.. hawk-prod-f3e..
	TO IDENTIFY RESOURCE MISALLOCATION (E.G. WHETHER ANY INSTANCES ARE OVER- OR UNDERSIZED)	
ERROR OR OTHER WORK METRICS	TO QUICKLY IDENTIFY DEGRADED HOSTS	
RELATED METRICS	TO SEE CORRELATIONS IN A SINGLE GRAPH	 Points processed (size) vs memory used (color) us-east-1e us-east-1a us-east-1b

Distributions

Distribution graphs show a histogram of a metric's value across a segment of infrastructure. Each bar in the graph represents a range of binned values, and its height corresponds to the number of entities reporting values in that range.

Distribution graphs are closely related to heat maps. The key difference between the two is that heat maps show change over time, whereas distributions are a summary of a time window. Like heat maps, distributions handily visualize large numbers of entities reporting a particular metric, so they are often used to graph metrics at the individual host or container level.

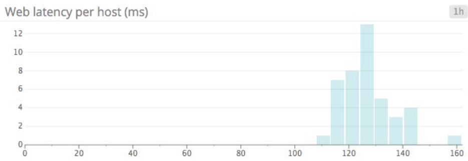
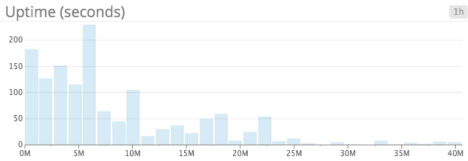


1 AGGREGATION ACROSS SPACE
Average the latency by host

2 AGGREGATION ACROSS TIME
Take the average of that latency over the past hour

3 HISTOGRAM
Plot the distribution of hosts by latency by bands

WHEN TO USE DISTRIBUTIONS

WHAT	WHY	EXAMPLE
SINGLE METRIC REPORTED BY A LARGE NUMBER OF ENTITIES	TO CONVEY GENERAL HEALTH OR STATUS AT A GLANCE	WEB LATENCY PER HOST 
	TO SEE VARIATIONS ACROSS MEMBERS OF A GROUP	UPTIME PER HOST 

In Summary

As you've seen here, each of these summary graphs has unique benefits and use cases. Understanding all the visualizations in your toolkit, and when to use each type, will help you convey actionable information clearly in your dashboards.

In the next chapters, we'll make these monitoring concepts more concrete by applying them to two extremely popular infrastructure technologies: Kubernetes and AWS Lambda.

Chapter 7: Putting It All Together – Monitoring Kubernetes

Container technologies have taken the infrastructure world by storm. Ideal for microservice architectures and environments that scale rapidly or have frequent releases, containers have seen a rapid increase in usage in recent years. But adopting Docker, containerd, or other container runtimes introduces significant complexity in terms of orchestration. That's where Kubernetes comes into play.

Kubernetes, often abbreviated as K8s, automates the scheduling, scaling, and maintenance of containers in any infrastructure environment. First open-sourced by Google in 2014, Kubernetes is now part of the Cloud Native Computing Foundation (CNCF). Since its introduction, Kubernetes has been steadily gaining in popularity across a range of industries and use cases. Datadog's research on container use shows that almost one-half of organizations running containers were using Kubernetes as of November 2022.

Kubernetes manages your containers—starting, stopping, creating, and destroying them automatically to reflect changes in demand or resource availability. Kubernetes automates your container infrastructure via:

- Container scheduling and auto-scaling
- Health checking and recovery
- Replication for parallelization and high availability
- Internal network management for service naming, discovery, and load balancing
- Resource allocation and management

Kubernetes can orchestrate your containers wherever they run, which facilitates multi-cloud deployments and migrations between infrastructure platforms. Hosted and self-managed flavors of Kubernetes abound, from enterprise-optimized platforms such as OpenShift and Pivotal Container Service to cloud services such as Google Kubernetes Engine, Amazon Elastic Kubernetes Service, Azure Kubernetes Service, and Oracle's Container Engine for Kubernetes.

This chapter walks through monitoring Kubernetes and is broken into three parts:

1. What does Kubernetes mean for your monitoring?
2. Key Kubernetes metrics
3. Monitoring Kubernetes with Datadog

What Does Kubernetes Mean for Your Monitoring?

Kubernetes requires you to rethink and reorient your monitoring strategies, especially if you are accustomed to monitoring traditional, long-lived hosts such as VMs or physical machines. Just as containers have completely transformed how we think about running services on virtual machines, Kubernetes has changed the way we interact with containerized applications.

The good news is that the abstraction inherent to a Kubernetes-based architecture already provides a framework for understanding and monitoring your applications in a dynamic container environment. With a proper monitoring approach that dovetails with Kubernetes's built-in abstractions, you can get a comprehensive view of application health and performance, even if the containers running those applications are constantly shifting across hosts or scaling up and down.

Monitoring Kubernetes differs from traditional monitoring of more static resources in several ways:

- Tags and labels are essential for continuous visibility
- Additional layers of abstraction mean more components to monitor
- Applications are highly distributed and constantly moving

Tags and Labels Were Important… Now They're Essential

Just as Kubernetes uses labels to identify which pods belong to a particular service, you can use these labels to aggregate data from individual pods and containers to get continuous visibility into services and other Kubernetes objects.

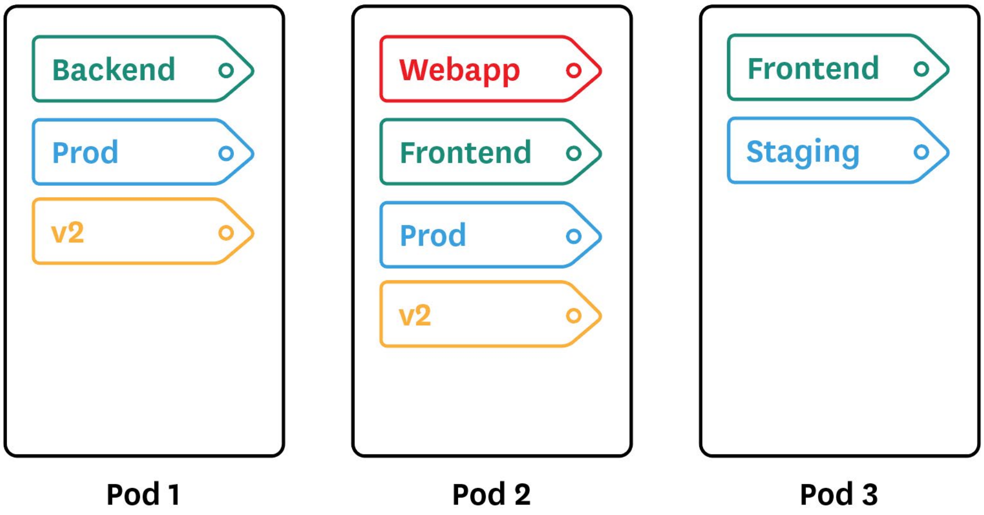
In the pre-container world, tags and labels were important for monitoring your infrastructure. They allowed you to group hosts and aggregate their metrics at any level of abstraction. In particular, tags have proved extremely useful for tracking the performance of dynamic cloud infrastructure and investigating issues that arise there.

A container environment brings even larger numbers of objects to track, with even shorter lifespans. The automation and scalability of Kubernetes only exaggerates this difference. With so many moving pieces in a typical Kubernetes cluster, labels provide the only reliable way to identify your pods and the applications within.

To make your observability data as useful as possible, you should label your pods in a way that allows you to look at any aspect of your applications and infrastructure, including:

- Environment (prod, staging, dev, etc.)
- App
- Team
- Version

These user-generated labels are essential for monitoring because they are the only way to slice and dice metrics and events across the different layers of your Kubernetes architecture.



By default, Kubernetes also exposes basic information about pods (**name**, **namespace**), containers (**container ID**, **image**), and nodes (**instance ID**, **hostname**). Some monitoring tools can ingest these attributes and turn them into tags so you can use them just like other custom Kubernetes labels.

Kubernetes also exposes some labels from Docker. But note that you cannot apply custom Docker labels to your images or containers when using Kubernetes. You can only apply Kubernetes labels to your pods.

Thanks to these Kubernetes labels at the pod level and Docker labels at the container level, you can easily slice and dice along any dimension to get a logical view of your infrastructure and applications. You can examine every layer in your stack (namespace, replica set, pod, or container) to aggregate your metrics and drill down for investigation.

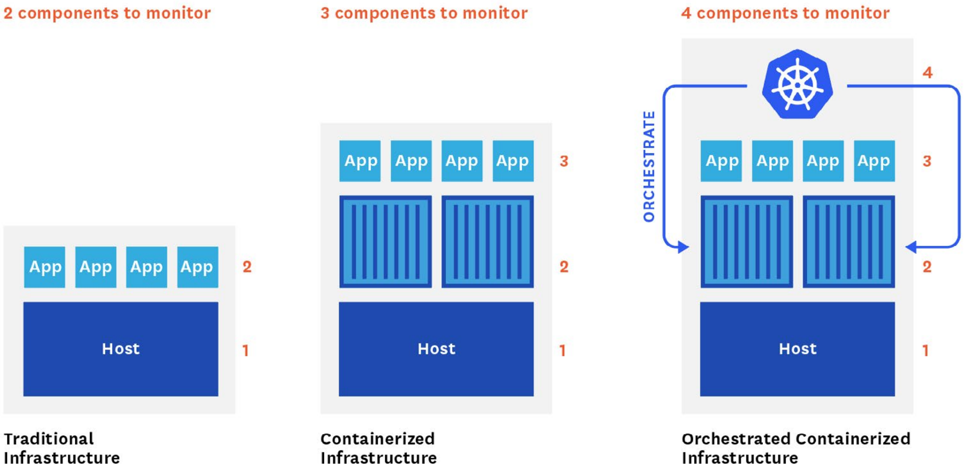
Because they are the only way to generate an accessible, up-to-date view of your pods and applications, labels and tags should form the basis of your monitoring and alerting strategies. The performance metrics you track won't be attached to hosts; instead, they are aggregated around labels that you will use to group or filter the pods you are interested in. Make sure that you define a logical and easy-to-follow schema for your namespaces and labels so that the meaning of a particular label is easily comprehensible to anyone in your organization.

More Components to Monitor

In traditional, host-centric infrastructure, you had only two main layers to monitor: your applications and the hosts running them. Then containers added a new layer of abstraction between the host and your applications.

Now Kubernetes, which orchestrates your containers, also needs to be monitored in order to comprehensively track your infrastructure. That makes four different components that now need to be monitored, each with their own specificities and challenges:

- Your hosts, even if you don't know which containers and applications they are actually running
- Your containers, even if you don't know where they're running
- Your containerized applications
- The Kubernetes cluster itself



Furthermore, the architecture of a Kubernetes cluster introduces a new wrinkle when it comes to monitoring the applications running on your containers...

Your Applications Are Moving!

To effectively monitor the health of your Kubernetes infrastructure, it's essential to collect metrics and events from all your containers and pods. But to understand what your customers or users are experiencing, you also need to monitor the applications running in these pods. With Kubernetes, which automatically schedules your workloads, you usually have very little control over where those applications are running. Kubernetes does allow you to assign node affinity or anti-affinity to particular pods, but most users will delegate such control to Kubernetes to benefit from its automatic scheduling and resource management.

Given the rate of change in a typical Kubernetes cluster, manually configuring checks to collect monitoring data from these applications every time a container is started or restarted is simply not possible. So what else can you do?

A Kubernetes-aware monitoring tool with service discovery lets you make full use of the scaling and automation built into Kubernetes, without sacrificing visibility. Service discovery enables your monitoring system to detect any change in your inventory of running pods and automatically reconfigure your data collection so you can continuously monitor your containerized workloads even as they expand, contract, and shift across hosts.

With orchestration tools like Kubernetes, **service discovery** mechanisms become a must-have for monitoring. In the next part of this chapter, we'll explore the key Kubernetes metrics that are available for monitoring.

Key Kubernetes Metrics

Kubernetes requires you to rethink your approach when it comes to monitoring. But if you know what to observe, where to find the relevant data, and how to aggregate and interpret that data, you can ensure that your applications are performant and that Kubernetes is doing its job effectively. Monitoring a Kubernetes environment requires a different approach than monitoring VM-based workloads or even unorchestrated containers. The good news is that Kubernetes is built around objects such as Deployments and DaemonSets, which provide long-lived abstractions on top of dynamic container workloads. So even though individual containers and pods may come and go, you can use these abstractions to aggregate your data and monitor the performance of your workloads.

Additionally, Kubernetes provides a wealth of APIs for automation and cluster management, including robust APIs for collecting performance data. Organizations can leverage two complementary add-ons for aggregating and reporting valuable monitoring data from your cluster: Metrics Server and kube-state-metrics.

Metrics Server collects resource usage statistics from the kubelet on each node and provides aggregated metrics through the Metrics API. Metrics Server stores only near-real-time metrics in memory, so it is primarily valuable for spot checks of CPU or memory usage, or for periodic querying by a full-featured monitoring service that retains data over longer timespans.

kube-state-metrics is a service that makes cluster state information easily consumable. Whereas Metrics Server exposes metrics on resource usage by pods and nodes, kube-state-metrics listens to the Control Plane API server for data on the overall status of Kubernetes objects (nodes, pods, Deployments, etc.), as well as the resource limits and allocations for those objects. It then generates metrics from that data which are available through the Metrics API.

We'll briefly discuss how Kubernetes generates metrics from these APIs. Then we'll dig into the data you can collect to monitor the Kubernetes platform itself, focusing on the following areas:

- Cluster state metrics
- Resource metrics from Kubernetes nodes and pods
- Work metrics from the Kubernetes Control Plane

We'll also touch on the value of collecting Kubernetes events.

Cluster State Metrics

The Kubernetes API server emits data about the count, health, and availability of various Kubernetes objects, such as pods. Internal Kubernetes processes and components use this information to track whether pods are being launched and maintained as expected and to properly schedule new pods. These cluster state metrics can also provide you with a high-level view of your cluster and its state. They can surface issues with nodes or pods, alerting you to the possibility that you need to investigate a bottleneck or scale out your cluster.

For Kubernetes objects that are deployed to your cluster, several similar but distinct metrics are available, depending on the type of **controller** that manages those objects. Two important types of controllers are:

- Deployments, which create a specified number of pods (often combined with a Service that creates a persistent point of access to the pods in the Deployment)
- DaemonSets, which ensure that a particular pod is running on every node (or on a specified set of nodes)

You can learn about these and other types of controllers in the Kubernetes documentation.

Node status

This cluster state metric provides a high-level overview of a node's health and whether the scheduler can place pods on that node. It runs checks on the following node conditions:

- `OutOfDisk`
- `Ready` (node is ready to accept pods)
- `MemoryPressure` (node memory is too low)
- `PIDPressure` (too many running processes)
- `DiskPressure` (remaining disk capacity is too low)
- `NetworkUnavailable`

Each check returns `true`, `false`, or—if the worker node hasn't communicated with the relevant Control Plane node for a grace period (which defaults to 40 seconds)—`unknown`. In particular, the `Ready` and `NetworkUnavailable` checks can alert you to nodes that are unavailable or otherwise not usable so that you can troubleshoot further. If a node returns `true` for the `MemoryPressure` or `DiskPressure` check, the kubelet attempts to reclaim resources. This includes garbage collection and possibly deleting pods from the node.

DESIRED VS. CURRENT PODS

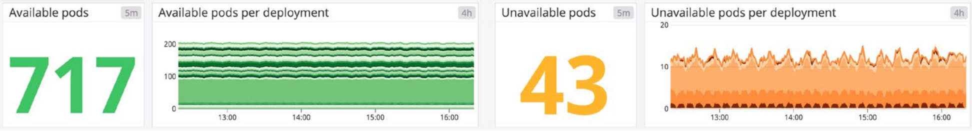
While you can manually launch individual pods for small-scale experiments, in production you will more likely use controllers to describe the desired state of your cluster and automate the creation of pods. For example, a Deployment manifest includes a statement of how many replicas of each pod should run. This ensures that the Control Plane will attempt to keep that many replicas running at all times, even if one or more nodes or pods crashes.

Alternatively, a DaemonSet launches one pod on every node in your cluster (unless you specify a subset of nodes). This is often useful for installing a monitoring agent or other node-level utility across your cluster.

Kubernetes provides metrics that reflect the number of desired pods (e.g., `kube_deployment_spec_replicas`) and the number of currently running pods (e.g., `kube_deployment_status_replicas`). Typically, these numbers should match unless you are in the midst of a deployment or other transitional phase, so comparing these metrics can alert you to issues with your cluster. In particular, a large disparity between desired and running pods can point to bottlenecks, such as your nodes lacking the resource capacity to schedule new pods. It could also indicate a problem with your configuration that is causing pods to fail.

AVAILABLE AND UNAVAILABLE PODS

A pod may be running but not available, meaning it is not ready and able to accept traffic. This is normal during certain circumstances, such as when a pod is newly launched or when a change is made and deployed to the specification of that pod. But if you see spikes in the number of unavailable pods, or pods that are consistently unavailable, it might indicate a problem with their configuration.



In particular, a large number of unavailable pods might point to poorly configured readiness probes. Developers can set readiness probes to give a pod time to perform initial startup tasks (such as loading large files) before accepting requests so that the application or service doesn't experience problems. Checking a pod's logs can provide clues as to why it is stuck in a state of unavailability.

Resource Metrics

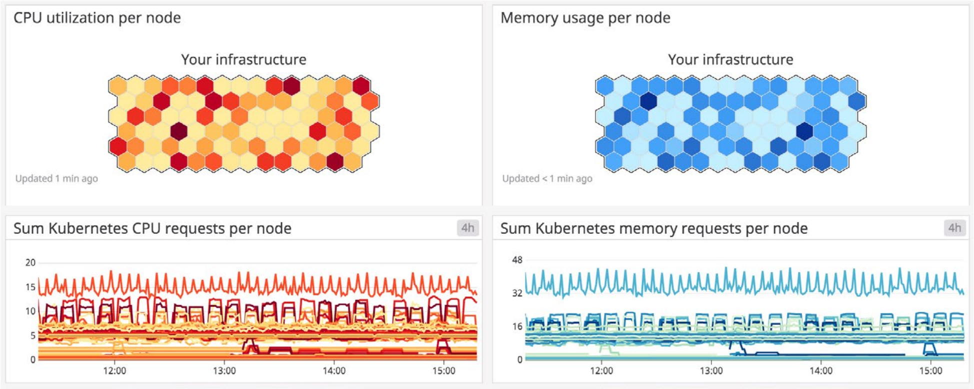
Monitoring memory, CPU, and disk usage within nodes and pods can help you detect and troubleshoot application-level problems. But monitoring the resource usage of Kubernetes objects is not as straightforward as monitoring a traditional application. In Kubernetes, you still need to track the actual resource usage of your workloads, but those statistics become more actionable when you monitor them in the context of resource requests and limits, which govern how Kubernetes manages finite resources and schedules workloads across the cluster.

In a manifest, you can declare a **request** and a **limit** for CPU (measured in cores) and memory (measured in bytes) for each container running on a pod. A request is the minimum amount of CPU or memory that a node will allocate to the container; a limit is the maximum amount that the container will be allowed to use. The requests and limits for an entire pod are calculated from the sum of the requests and limits of its constituent containers.

Requests and limits do not define a pod's actual resource utilization, but they significantly affect how Kubernetes schedules pods on nodes. Specifically, new pods will only be placed on a node that can meet their requests. Requests and limits are also integral to how a kubelet manages available resources by terminating pods (stopping the processes running on its containers) or evicting pods (deleting them from a node), which we'll cover in more detail below.

You can read more in the Kubernetes documentation about configuring requests and limits and how Kubernetes responds when resources run low.

Comparing resource utilization with resource requests and limits will provide a more complete picture of whether your cluster has the capacity to run its workloads and accommodate new ones. It's important to keep track of resource usage at different layers of your cluster, particularly for your nodes and for the pods running on them.

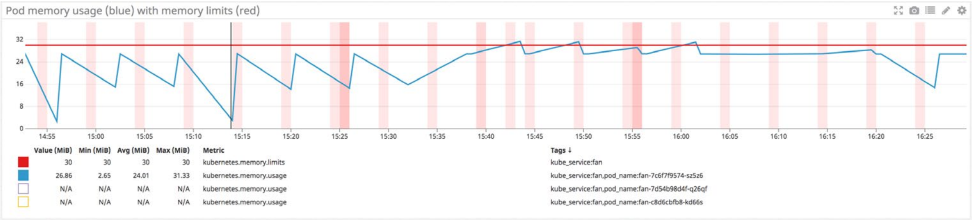


MEMORY LIMITS PER POD VS. MEMORY UTILIZATION PER POD

When specified, a memory limit represents the maximum amount of memory that a node will allocate to a container. If a limit is not provided in the manifest and there is not an overall configured default, a pod could use the entirety of a node's available memory. Note that a node can be oversubscribed, meaning that the sum of the limits for all pods running on a node might be greater than that node's total allocatable memory. This requires that the pods' defined requests are below the limit. The node's kubelet will reduce resource allocation to individual pods if they use more than they request as long as that allocation at least meets their requests.

Tracking pods' actual memory usage in relation to their specified limits is particularly important because memory is a non-compressible resource. In other words, if a pod uses more memory than its defined limit, the kubelet can't throttle its memory allocation, so it terminates the processes running on that pod instead. If this happens, the pod will show a status of `OOMKilled`.

Comparing your pods' memory usage to their configured limits will alert you to whether they are at risk of being OOM killed, as well as whether their limits make sense. If a pod's limit is too close to its standard memory usage, the pod may get terminated due to an unexpected spike. On the other hand, you may not want to set a pod's limit to be significantly higher than its typical usage because that can lead to poor scheduling decisions. For example, a pod with a memory request of 1 GiB and a limit of 4 GiB can be scheduled on a node with 2 GiB of allocatable memory (more than sufficient to meet its request). But if the pod suddenly needs 3 GiB of memory, it will be killed even though it's well below its memory limit.


Pod memory usage (blue) with memory limits (red)

	Value (MiB)	Min (MiB)	Avg (MiB)	Max (MiB)	Metric	Tags ↓
🟥	30	30	30	30	kubernetes.memory.limits	kube_service:fan
🟦	26.86	2.65	24.01	31.33	kubernetes.memory.usage	kube_service:fan,pod_name:fan-7c6f7f9574-sz5z6
□	N/A	N/A	N/A	N/A	kubernetes.memory.usage	kube_service:fan,pod_name:fan-7d54b98d4f-q26qf
□	N/A	N/A	N/A	N/A	kubernetes.memory.usage	kube_service:fan,pod_name:fan-c8d6cbfb8-kd66s

MEMORY UTILIZATION

Keeping an eye on memory usage at the pod and node levels can provide important insight into your cluster's performance and its ability to successfully run workloads. As we've seen, pods whose actual memory usage exceeds their limits will be terminated. Additionally, if a node runs low on available memory, the kubelet flags it as being under memory pressure and begins to reclaim resources.

In order to reclaim memory, the kubelet can evict pods, meaning it will delete these pods from the node. The Control Plane will attempt to reschedule evicted pods on another node with sufficient resources. If your pods' memory usage significantly exceeds their defined requests, it can cause the kubelet to prioritize those pods for eviction, so comparing requests with actual usage can help surface which pods might be vulnerable to eviction.

Habitually exceeding requests could also indicate that your pods are not configured appropriately. As mentioned above, scheduling is largely based on a pod's request, so a pod with a bare-minimum memory request could be placed on a node without enough resources to withstand any spikes or increases in memory needs. Correlating and comparing each pod's actual utilization against its requests can give insight into whether the requests and limits specified in your manifests make sense, or if there might be some issue that is causing your pods to use more resources than expected.

Monitoring overall memory utilization on your nodes can also help you determine when you need to scale your cluster. If node-level usage is high, you may need to add nodes to the cluster to share the workload.

MEMORY REQUESTS PER NODE VS. ALLOCATABLE MEMORY PER NODE

Memory requests, as discussed above, are the minimum amounts of memory a node's kubelet will assign to a container. If a request is not provided, it will default to whatever the value is for the container's limit (which, if also not set, could be all memory on the node). Allocatable memory reflects the amount of memory on a node that is available for pods. Specifically, it takes the overall capacity and subtracts memory requirements for OS and Kubernetes system processes to ensure they will not fight user pods for resources.

Although memory capacity is a static value, maintaining an awareness of the sum of pod memory requests on each node versus each node's allocatable memory is important for capacity planning. These metrics will inform you if your nodes have enough capacity to meet the memory requirements of all current pods and whether the Control Plane is able to schedule new ones. Kubernetes's scheduling process uses several levels of criteria to determine if it can place a pod on a specific node. One of the initial tests is whether a node has enough allocatable memory to satisfy the sum of the requests of all the pods running on that node plus the new pod.

Comparing memory requests to capacity metrics can also help you troubleshoot problems with launching and running the desired number of pods across your cluster. If you notice that your cluster's count of current pods is significantly less than the number of desired pods, these metrics might show you that your nodes don't have the resource capacity to host new pods, so the Control Plane is failing to find a node to assign desired pods to. One straightforward remedy for this issue is to provision more nodes for your cluster.

DISK UTILIZATION

Like memory, disk space is a non-compressible resource, so if a kubelet detects low disk space on its root volume, it can cause problems with scheduling pods. If a node's remaining disk capacity crosses a certain resource threshold, it will get flagged as under disk pressure. The following are the default resource thresholds for a node to come under disk pressure:

DISK PRESSURE SIGNAL	THRESHOLD	DESCRIPTION
IMAGEFS.AVAILABLE	15%	AVAILABLE DISK SPACE FOR THE `imagefs` FILESYSTEM, USED FOR IMAGES AND CONTAINER-WRITABLE LAYERS
IMAGEFS.INODESFREE	5%	AVAILABLE INDEX NODES FOR THE `imagefs` FILESYSTEM
NODEFS.AVAILABLE	10%	AVAILABLE DISK SPACE FOR THE ROOT FILESYSTEM
NODEFS.INODESFREE	5%	AVAILABLE INDEX NODES FOR THE ROOT FILESYSTEM

Crossing one of these thresholds leads the kubelet to initiate garbage collection to reclaim disk space by deleting unused images or dead containers. As a next step, if it still needs to reclaim resources, it will start evicting pods.

In addition to node-level disk utilization, you should also track the usage levels of the volumes used by your pods. This helps you stay ahead of problems at the application or service level. Once these volumes have been provisioned and attached to a node, the node's kubelet exposes several volume-level disk utilization metrics, such as the volume's capacity, utilization, and available space. These volume metrics are available from Kubernetes's Metrics API.

If a volume runs out of remaining space, any applications that depend on that volume will likely experience errors as they try to write new data to the volume. Setting an alert to trigger when a volume reaches 80 percent usage can give you time to create new volumes or scale up the storage request to avoid problems.

CPU REQUESTS PER NODE VS. ALLOCATABLE CPU PER NODE

As with memory, allocatable CPU reflects the CPU resources on the node that are available for pod scheduling, while requests are the minimum amount of CPU a node will attempt to allocate to a pod.

As mentioned previously, Kubernetes measures CPU in cores. Tracking overall CPU requests per node and comparing them to each node's allocatable CPU capacity is valuable for capacity planning of a cluster and will provide insight into whether your cluster can support more pods.

CPU LIMITS PER POD VS. CPU UTILIZATION PER POD

These metrics let you track the maximum amount of CPU that a node will allocate to a pod compared to how much CPU it's actually using. Unlike memory, CPU is a compressible resource. This means that if a pod's CPU usage exceeds its defined limit, the node will throttle the amount of CPU available to that pod but allow it to continue running. This throttling can lead to performance issues, so even if your pods won't be terminated, keeping an eye on these metrics will help you determine if your limits are configured properly based on the pods' actual CPU needs.

CPU UTILIZATION

Tracking the amount of CPU your pods are using compared to their configured requests and limits, as well as CPU utilization at the node level, will give you important insight into cluster performance. Much like a pod exceeding its CPU limits, a lack of available CPU at the node level can lead to the node throttling the amount of CPU allocated to each pod.

Measuring actual utilization compared to requests and limits per pod can help determine if these are configured appropriately and your pods are requesting enough CPU to run properly. Alternatively, consistently higher than expected CPU usage might point to problems with the pod that need to be identified and addressed.

Control Plane Metrics

As mentioned briefly above, the Kubernetes Control Plane provides several core services and resources for cluster management:

- **The API server** is the gateway through which developers and administrators query and interact with the cluster
- **Controller managers** implement and track the lifecycle of the different controllers that are deployed to the cluster (e.g., to regulate the replication of workloads)
- **Schedulers** query the state of the cluster and schedule and assign workloads to worker nodes
- **etcd** data stores maintain a record of the state of the cluster in a distributed key-value store so that other components can ensure that all worker nodes are healthy and running the desired workloads

Kubernetes exposes metrics for each of these components, which you can collect and track to ensure that your cluster's central nervous system is healthy. Note that in managed Kubernetes environments (such as Google Kubernetes Engine or Amazon Elastic Kubernetes Service clusters), the Control Plane is managed by the cloud provider, and you may not have access to all the components and metrics listed below. Also, the availability or names of certain metrics may be different depending on which version of Kubernetes you are using.

ETCD_SERVER_HAS_LEADER

Except during leader election events, the etcd cluster should always have a leader, which is necessary for the operation of the key-value store. If a particular member of an etcd cluster reports a value of 0 for `etcd_server_has_leader` (perhaps due to network issues), that member of the cluster does not recognize a leader and is unable to serve queries. Therefore, if every cluster member reports a value of 0, the entire etcd cluster is down. A failed etcd key-value store deprives Kubernetes of necessary information about the state of cluster objects, and prevents Kubernetes from making any changes to cluster state. Because of its critical role in cluster operations, etcd provides snapshot and recovery operations to mitigate the impact of failure scenarios.

ETCD_SERVER_LEADER_CHANGES_SEEN_TOTAL

This metric tracks the number of leader transitions within the cluster. Frequent leader changes, though not necessarily damaging on their own, can alert you to issues with connectivity or resource limitations in the etcd cluster.

APISERVER_REQUEST_LATENCIES_COUNT, APISERVER_REQUEST_LATENCIES_SUM

Kubernetes provides metrics on the number and duration of requests to the API server for each combination of resource (e.g., pods, Deployments) and verb (e.g., GET, LIST, POST, DELETE). By dividing the summed latency for a specific type of request by the number of requests of that type, you can compute a per-request average latency. You can also track these metrics over time and divide their deltas to compute a real-time average. By tracking the number and latency of specific kinds of requests, you can see if the cluster is falling behind in executing any user-initiated commands to create, delete, or query resources, likely due to the API server being overwhelmed with requests.

WORKQUEUE_QUEUE_DURATION_SECONDS, WORKQUEUE_WORK_DURATION_SECONDS

These latency metrics provide insight into the performance of the controller manager, which queues up each actionable item (such as the replication of a pod) before it's carried out. Each metric is tagged with the name of the relevant queue, such as `queue:daemonset` or `queue:node_lifecycle_controller`. The metric `workqueue_queue_duration_seconds` tracks how much time, in aggregate, items in a specific queue have spent awaiting processing, whereas `workqueue_work_duration_seconds` reports how much time it took to actually process those items. If you see a latency increase in the automated actions of your controllers, you can look at the logs for the controller manager to gather more details about the cause.

SCHEDULER_SCHEDULE_ATTEMPTS_TOTAL, END-TO-END SCHEDULING LATENCY

You can track the work of the Kubernetes scheduler by monitoring its overall number of attempts to schedule pods on nodes, as well as the end-to-end latency of carrying out those attempts. The metric `scheduler_schedule_attempts_total` breaks out the scheduler's attempts by result (`error`, `schedulable`, or `unschedulable`), so you can identify problems with matching pods to worker nodes. An increase in `unschedulable` pods indicates that your cluster may lack the resources needed to launch new pods, whereas an attempt that results in an `error` status reflects an internal issue with the scheduler itself.

The end-to-end latency metrics report both how long it takes to select a node for a particular pod, as well as how long it takes to notify the API server of the scheduling decision so it can be applied to the cluster. If you notice a discrepancy between the number of desired and current pods, you can dig into these latency metrics to see if a scheduling issue is behind the lag. Note that the end-to-end latency is reported differently depending on the version of Kubernetes you are running (pre-v1.14 or 1.14+), with different time units as well.

Kubernetes Events

Collecting events from Kubernetes and from the container engine (such as Docker) allows you to see how pod creation, destruction, starting, or stopping affects the performance of your infrastructure—and vice versa.

While Docker events trace container lifecycles, Kubernetes events report on pod lifecycles and deployments. Tracking Kubernetes Pending pods and pod failures, for example, can point you to misconfigured launch manifests or issues of resource saturation on your nodes. That's why you should correlate events with Kubernetes metrics for easier investigation.

Using Datadog to Monitor Kubernetes

So far in this chapter, you've learned how you can use different Kubernetes commands and add-ons to spot-check the health and resource usage of Kubernetes cluster objects. Next, we'll show you how you can get more comprehensive visibility into your cluster by collecting all of your telemetry data in one place and tracking it over time.

Datadog's integrations with Kubernetes, Docker, containerd, etcd, Istio, and other related technologies are designed to tackle the considerable challenges of monitoring orchestrated containers and services. Datadog integrates with each part of your Kubernetes cluster to provide a complete picture of health and performance:

- The Datadog Agent's Kubernetes integration collects metrics, events, and logs from your cluster components, workload pods, and other Kubernetes objects

- Integrations with container runtimes including Docker and containerd collect container-level metrics for detailed resource breakdowns

- Wherever your Kubernetes clusters are running—including Amazon Web Services, Google Cloud Platform, or Azure—the Datadog Agent automatically monitors the nodes of your Kubernetes clusters

- Datadog's Autodiscovery and 600+ built-in integrations automatically monitor the technologies you are deploying

- APM and distributed tracing provide transaction-level insight into applications running in your Kubernetes clusters

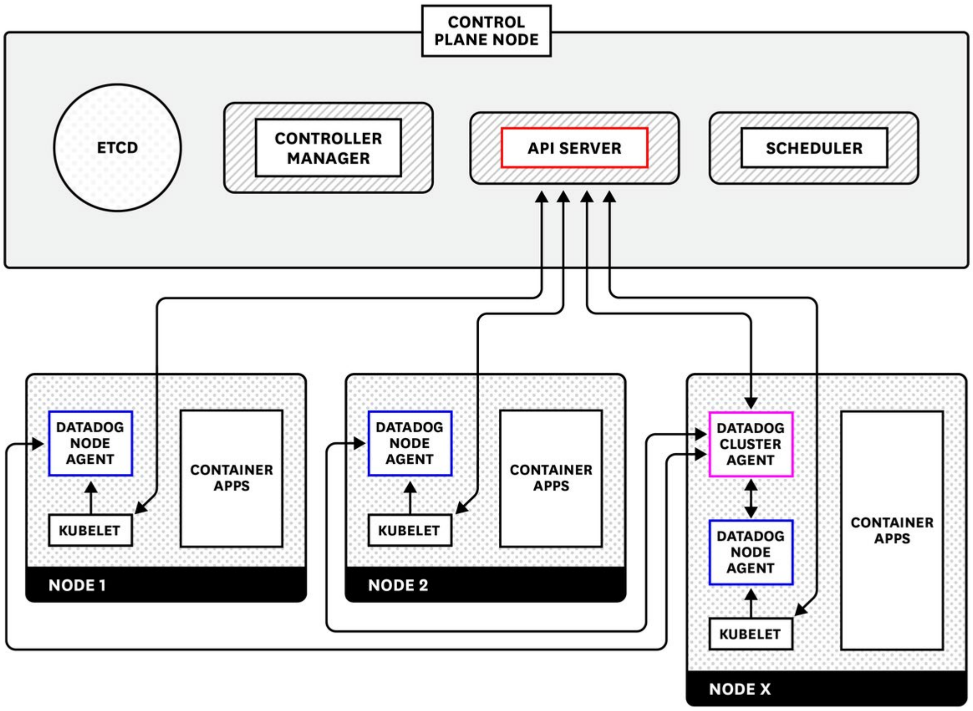
COLLECT, VISUALIZE, AND ALERT ON KUBERNETES METRICS IN MINUTES

The first step in setting up comprehensive Kubernetes monitoring is deploying the Datadog Agent to the nodes of your cluster.

The Datadog Agent is open source software that collects and reports metrics, distributed traces, and logs from each of your nodes, so you can view and monitor your entire infrastructure in one place. In addition to collecting telemetry data from Kubernetes, Docker, and other infrastructure technologies, the Agent automatically collects and reports resource metrics (such as CPU, memory, and network traffic) from your nodes, whatever the underlying infrastructure platform.

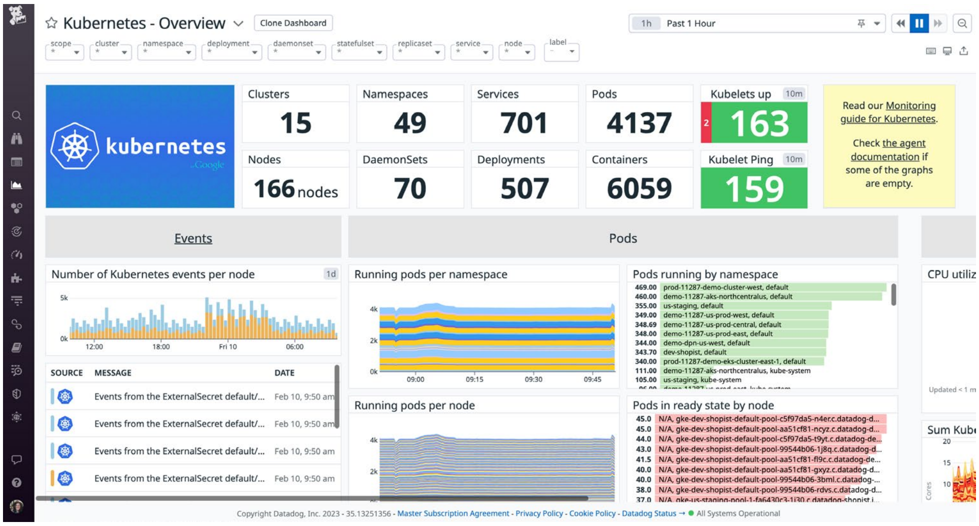
The Datadog documentation outlines multiple methods for installing the Agent, including using the Helm package manager and installing the Agent directly onto the nodes of your cluster. The recommended approach for the majority of use cases, however, is to deploy the containerized version of the Agent. By deploying the containerized Agent to your cluster as a DaemonSet, you can ensure that one copy of the Agent runs on each host in your cluster, even as the cluster scales up and down. You can also specify a subset of nodes that you wish to run the Agent by using nodeSelectors.

Datadog's Agent deployment instructions for Kubernetes provide a full manifest for deploying the containerized node-based Agent as a DaemonSet. If you wish to get started quickly for experimentation purposes, you can follow those directions to roll out the Agent across your cluster. To take it one step further, you can read our guide to learn how to install the Agent on all your nodes, and deploy the specialized Datadog Cluster Agent, which provides several additional benefits for large-scale, production use cases.



Dive into the Metrics

You can view the data you're already collecting in Datadog's built-in Kubernetes dashboard.



SOURCE	MESSAGE	DATE
	Events from the ExternalSecret default/...	Feb 10, 9:50 am
	Events from the ExternalSecret default/...	Feb 10, 9:50 am
	Events from the ExternalSecret default/...	Feb 10, 9:50 am
	Events from the ExternalSecret default/...	Feb 10, 9:50 am

Pods running by namespace

469.00	prod-11287-demo-cluster-west, default
460.00	demo-11287-aks-northcentralus, default
355.00	us-staging, default
349.00	demo-11287-us-prod-west, default
348.69	demo-11287-us-prod-central, default
348.00	demo-11287-us-prod-east, default
344.00	demo-dpn-us-west, default
343.70	dev-shopist, default
340.00	prod-11287-demo-eks-cluster-east-1, default
111.00	demo-11287-aks-northcentralus, kube-system
105.00	us-staging, kube-system

Pods in ready state by node

45.0	N/A, gke-dev-shopist-default-pool-c5f97da5-n4er.c.datadog-d...
45.0	N/A, gke-dev-shopist-default-pool-aa51cf81-ncyz.c.datadog-d...
44.0	N/A, gke-dev-shopist-default-pool-c5f97da5-t9yt.c.datadog-de...
43.0	N/A, gke-dev-shopist-default-pool-99544b06-1j8q.c.datadog-d...
41.5	N/A, gke-dev-shopist-default-pool-aa51cf81-fl9c.c.datadog-de...
40.0	N/A, gke-dev-shopist-default-pool-aa51cf81-gxyz.c.datadog-d...
40.0	N/A, gke-dev-shopist-default-pool-99544b06-3bml.c.datadog-...
38.0	N/A, gke-dev-shopist-default-pool-99544b06-rdvs.c.datadog-d...

You may recall from earlier in this chapter that certain cluster-level metrics—specifically, the counts of Kubernetes objects such as the count of pods desired, currently available, and currently unavailable—are provided by an optional cluster add-on called kube-state-metrics. If you see that this data is missing from the dashboard, it means that you have not deployed the kube-state-metrics service. To add these statistics to the lower-level resource metrics already being collected by the Agent, you simply need to deploy kube-state-metrics to your cluster.

Once kube-state-metrics is up and running, your cluster state metrics will start pouring into Datadog automatically, without any further configuration. That's because the Datadog Agent's Autodiscovery functionality detects when certain services are running and automatically enables metric collection from those services. Since kube-state-metrics is among the integrations automatically enabled by Autodiscovery, there's nothing more you need to do to start collecting your cluster state metrics in Datadog. Check out our in-depth guide for more information about the Agent's Autodiscovery feature.

Datadog includes integrations with the individual components of your cluster's Control Plane, including the API server, scheduler, and etcd cluster. Once you deploy Datadog to your cluster, metrics from these components automatically appear so that you can easily visualize and track the overall health and workload of your cluster. See our documentation for more information on our integrations with the API server, controller manager, scheduler, and etcd.

Monitoring Kubernetes using Tags

Datadog automatically imports metadata from Kubernetes, Docker, cloud services, and other technologies, and creates tags that you can use to sort, filter, and aggregate your data. Tags (and their Kubernetes equivalent, labels) are essential for monitoring dynamic infrastructure, where host names, IP addresses, and other identifiers are constantly in flux. With tags in Datadog, you can filter and view your resources by Kubernetes Deployment (kube_deployment) or Service (kube_service), or by Docker image (image_name). Datadog also automatically pulls in tags from your cloud provider, so you can view your nodes or containers by availability zone, instance type, and so on.

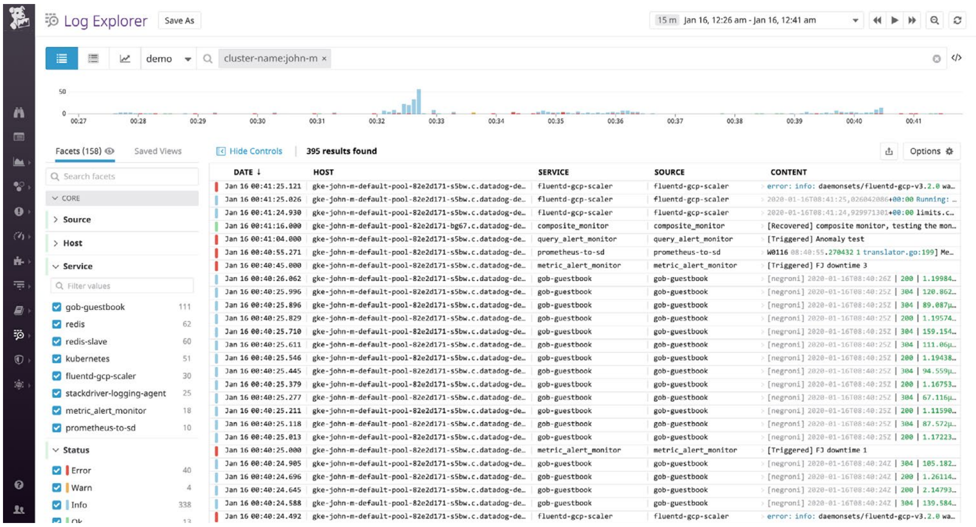
You can also import Kubernetes pod labels as tags, which captures pod-level metadata that you define in your manifests as tags, so you can use that metadata to filter and aggregate your telemetry data in Datadog.

The Datadog Agent automatically collects metrics from your nodes and containers. To get even more insight into your cluster, you can also configure Datadog to collect logs and distributed traces from your containerized applications.

Collect Logs from Your Kubernetes Clusters

Datadog can automatically collect logs from Kubernetes, Docker, and many other technologies you may be running on your cluster. Logs can be invaluable for troubleshooting problems, identifying errors, and giving you greater insight into the behavior of your infrastructure and applications. Check out our documentation to learn more about enabling log collection for your cluster.

With log collection enabled, you should start seeing logs flowing into the Log Explorer in Datadog. You can filter by your cluster name, node names, or any custom tag you applied earlier to drill down to the logs from only your deployment.



Service	
☑ gob-guestbook	111
☑ redis	62
☑ redis-slave	60
☑ kubernetes	51
☑ fluentd-gcp-scaler	30
☑ stackdriver-logging-agent	25
☑ metric_alert_monitor	18
☑ prometheus-to-sd	10

Status	
☑ Error	40
☑ Warn	4
☑ Info	338
☑ Ok	13

DATE ↓	HOST	SERVICE	SOURCE	CONTENT
Jan 16 00:41:25.121	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	fluentd-gcp-scaler	fluentd-gcp-scaler	error: info: daemonsets/fluentd-gcp-v3.2.0 wa…
Jan 16 00:41:25.026	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	fluentd-gcp-scaler	fluentd-gcp-scaler	2020-01-16T08:41:25,026042086+00:00 Running: …
Jan 16 00:41:24.930	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	fluentd-gcp-scaler	fluentd-gcp-scaler	2020-01-16T08:41:24,929971301+00:00 limits.c…
Jan 16 00:41:16.000	gke-john-m-default-pool-82e2d171-bg67.c.datadog-de…	composite_monitor	composite_monitor	[Recovered] composite monitor, testing the mon…
Jan 16 00:41:04.000	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	query_alert_monitor	query_alert_monitor	[Triggered] Anomaly test
Jan 16 00:40:55.271	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	prometheus-to-sd	prometheus-to-sd	W0116 08:40:55.270432 1 translator.go:199] Me…
Jan 16 00:40:45.000	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	metric_alert_monitor	metric_alert_monitor	[Triggered] FJ downtime 3
Jan 16 00:40:26.062	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:26Z | 200 | 1.19984…
Jan 16 00:40:25.996	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:25Z | 304 | 120.862…
Jan 16 00:40:25.896	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:25Z | 304 | 89.087µ…
Jan 16 00:40:25.829	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:25Z | 200 | 1.19574…
Jan 16 00:40:25.710	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:25Z | 304 | 159.154…
Jan 16 00:40:25.611	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:25Z | 304 | 111.06µ…
Jan 16 00:40:25.546	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:25Z | 200 | 1.19438…
Jan 16 00:40:25.445	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:25Z | 304 | 94.559µ…
Jan 16 00:40:25.379	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:25Z | 200 | 1.16753…
Jan 16 00:40:25.277	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:25Z | 304 | 67.116µ…
Jan 16 00:40:25.211	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:25Z | 200 | 1.11590…
Jan 16 00:40:25.118	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:25Z | 304 | 87.572µ…
Jan 16 00:40:25.013	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:25Z | 200 | 1.17223…
Jan 16 00:40:25.000	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	metric_alert_monitor	metric_alert_monitor	[Triggered] FJ downtime 1
Jan 16 00:40:24.905	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:24Z | 304 | 105.182…
Jan 16 00:40:24.696	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:24Z | 200 | 1.26114…
Jan 16 00:40:24.645	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:24Z | 200 | 2.14793…
Jan 16 00:40:24.588	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	gob-guestbook	gob-guestbook	[negroni] 2020-01-16T08:40:24Z | 304 | 139.584…
Jan 16 00:40:24.492	gke-john-m-default-pool-82e2d171-s5bw.c.datadog-de…	fluentd-gcp-scaler	fluentd-gcp-scaler	error: info: daemonsets/fluentd-gcp-v3.2.0 wa…

Note that in some container environments, including Google Kubernetes Engine, `/opt` is read-only, so the Agent will not be able to write to the standard path provided above. As a workaround, you can replace `/opt/datadog-agent/run` with `/var/lib/datadog-agent/logs` in your manifest if you run into read-only errors.

VIEW YOUR CLUSTER'S AUDIT LOGS

Kubernetes audit logs provide valuable information about requests made to your API servers. They record data such as which users or services are requesting access to cluster resources, and why the API server authorized or rejected those requests. Because audit logs are written in JSON, a monitoring service like Datadog can easily parse them for filtering and analysis. This enables you to set alerts on unusual behavior and troubleshoot potential API authentication issues that might affect whether users or services can access your cluster. See our documentation for steps on setting up audit log collection with Datadog.

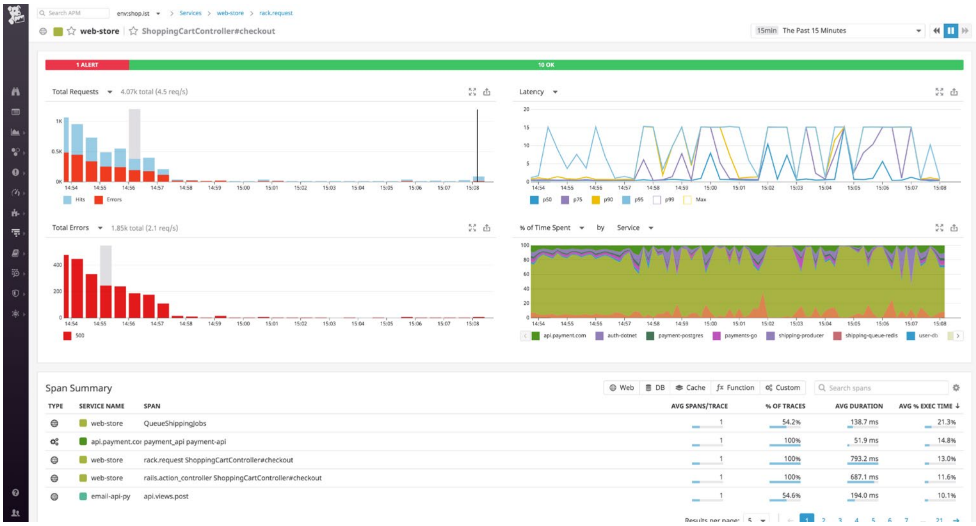
CATEGORIZE YOUR LOGS

Datadog automatically ingests, processes, and parses all of the logs from your Kubernetes cluster for analysis and visualization. To get the most value from your logs, ensure that they have a `source` tag and a `service` tag attached. For logs coming from one of Datadog's log integrations, the `source` sets the context for the log (e.g., `nginx`), enabling you to pivot between infrastructure metrics and related logs from the same system. The `source` tag also tells Datadog which log processing pipeline to use to properly parse those logs in order to extract structured facets and attributes. Likewise, the `service` tag (which is a core tag in Datadog APM) enables you to pivot seamlessly from logs to application-level metrics and request traces from the same service for rapid troubleshooting.

COLLECT TRACES FROM YOUR KUBERNETES CLUSTERS

Datadog APM traces requests to your application as they propagate across infrastructure and service boundaries. You can then visualize the full lifespan of these requests from end to end. APM gives you deep visibility into application performance, database queries, dependencies between services, and other insights that enable you to optimize and troubleshoot application performance. Datadog APM auto-instruments a number of languages and application frameworks; consult the documentation for supported languages and details on how to get started with instrumenting your language or framework.

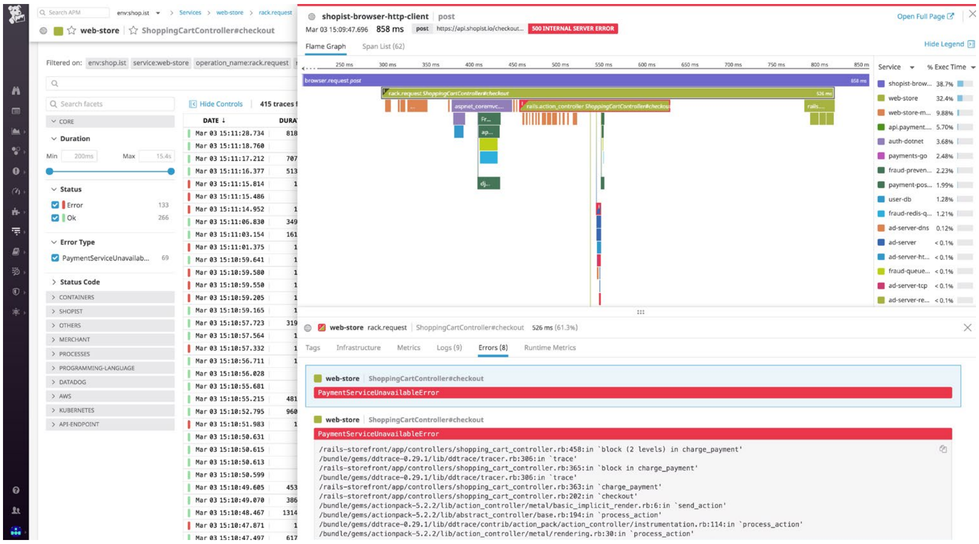
When you deploy your instrumented application, it will automatically begin sending traces to Datadog. From the APM tab of your Datadog account, you can see a breakdown of key performance indicators for each of your instrumented services with information about request throughput, latency, errors, and the performance of any service dependencies.



Span Summary

TYPE	SERVICE NAME	SPAN	AVG SPANS/TRACE	% OF TRACES	AVG DURATION	AVG % EXEC TIME ↓
	web-store	QueueShippingJobs	1	54.2%	138.7 ms	21.3%
	api.payment.coı	payment_api payment-api	1	100%	51.9 ms	14.8%
	web-store	rack.request ShoppingCartController#checkout	1	100%	793.2 ms	13.0%
	web-store	rails.action_controller ShoppingCartController#checkout	1	100%	687.1 ms	11.6%
	email-api-py	api.views.post	1	54.6%	194.0 ms	10.1%

You can then dive into individual traces to inspect a flame graph that breaks down the trace into spans, each one representing an individual database query, function call, or operation carried out as part of fulfilling the request. For each span, you can view system metrics, application runtime metrics, error messages, and relevant logs that pertain to that unit of work.



DATE ↓	DURA
Mar 03 15:11:28.734	818
Mar 03 15:11:18.760	
Mar 03 15:11:17.212	707
Mar 03 15:11:16.377	513
Mar 03 15:11:15.814	1
Mar 03 15:11:15.486	
Mar 03 15:11:14.952	1
Mar 03 15:11:06.830	349
Mar 03 15:11:03.154	161
Mar 03 15:11:01.375	1
Mar 03 15:10:59.641	1
Mar 03 15:10:59.580	1
Mar 03 15:10:59.550	1
Mar 03 15:10:59.205	1
Mar 03 15:10:59.165	1
Mar 03 15:10:57.723	319
Mar 03 15:10:57.564	1
Mar 03 15:10:57.332	1
Mar 03 15:10:56.711	1
Mar 03 15:10:56.028	
Mar 03 15:10:55.681	
Mar 03 15:10:55.215	481
Mar 03 15:10:52.795	960
Mar 03 15:10:51.983	1
Mar 03 15:10:50.631	
Mar 03 15:10:50.615	
Mar 03 15:10:50.613	
Mar 03 15:10:50.599	
Mar 03 15:10:49.605	453
Mar 03 15:10:49.070	386
Mar 03 15:10:48.467	1314
Mar 03 15:10:47.871	1
Mar 03 15:10:47.497	617

web-store | ShoppingCartController#checkout

PaymentServiceUnavailableError

web-store | ShoppingCartController#checkout

PaymentServiceUnavailableError

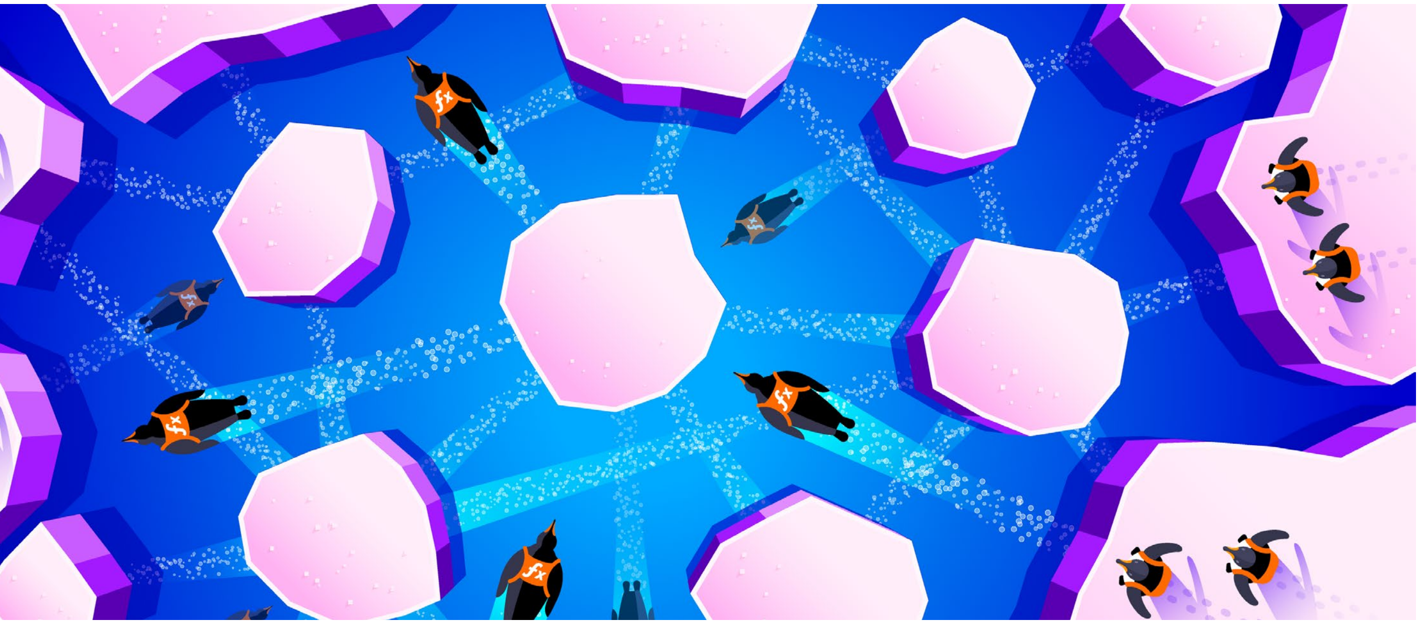
```
/rails-storefront/app/controllers/shopping_cart_controller.rb:458:in `block (2 levels) in charge_payment'
/bundle/gems/ddtrace-0.29.1/lib/ddtrace/tracer.rb:306:in `trace'
/rails-storefront/app/controllers/shopping_cart_controller.rb:365:in `block in charge_payment'
/bundle/gems/ddtrace-0.29.1/lib/ddtrace/tracer.rb:306:in `trace'
/rails-storefront/app/controllers/shopping_cart_controller.rb:363:in `charge_payment'
/rails-storefront/app/controllers/shopping_cart_controller.rb:202:in `checkout'
/bundle/gems/actionpack-5.2.2/lib/action_controller/metal/basic_implicit_render.rb:6:in `send_action'
/bundle/gems/actionpack-5.2.2/lib/abstract_controller/base.rb:194:in `process_action'
/bundle/gems/ddtrace-0.29.1/lib/ddtrace/contrib/action_pack/action_controller/instrumentation.rb:114:in `process_action'
/bundle/gems/actionpack-5.2.2/lib/action_controller/metal/rendering.rb:30:in `process_action'
```

Datadog provides even more Kubernetes monitoring functionality beyond the scope of this chapter. We encourage you to dive into Datadog's Kubernetes documentation to learn how to set up process monitoring, network performance monitoring, and the collection of custom metrics in Kubernetes.

Improved Visibility into Kubernetes Performance

In this chapter, we walked you through the key metrics you should monitor to keep tabs on your Kubernetes cluster. And we've shown you how integrating Kubernetes with Datadog enables you to build a more comprehensive view of your infrastructure. Monitoring with Datadog gives you critical visibility into what's happening with your containerized applications. You can easily create automated alerts on any metric, with triggers that are tailored precisely to your infrastructure and usage patterns.

Chapter 8: Putting It All Together – Monitoring AWS Lambda



In recent years, serverless technologies have experienced significant growth. More than half of all organizations operating in the cloud now leverage serverless, and AWS Lambda has become one of the most popular options available today. Lambda is a compute service at the center of the AWS serverless platform that deploys your serverless code as **functions**. Functions are event driven and can be triggered by events such as message queues, file uploads, HTTP requests, and cron jobs. You can trigger Lambda functions using events from other AWS services, such as API calls from Amazon API Gateway or changes to an Amazon DynamoDB table. When a service invokes a function for the first time, it initializes the function's runtime and **handler** method. AWS Lambda supports a range of runtimes, so you can write functions in the programming language of your choice (e.g., Go, Python, Node.js) and execute them within the same environment.

Key AWS Lambda Metrics

Since AWS Lambda manages infrastructure resources for you, you won't be able to capture typical system metrics such as CPU usage. Instead, Lambda reports the performance and efficiency of your functions as it runs them, so monitoring your functions involves tracking **function utilization**, **invocations**, and **concurrency** (including **provisioned concurrency**).

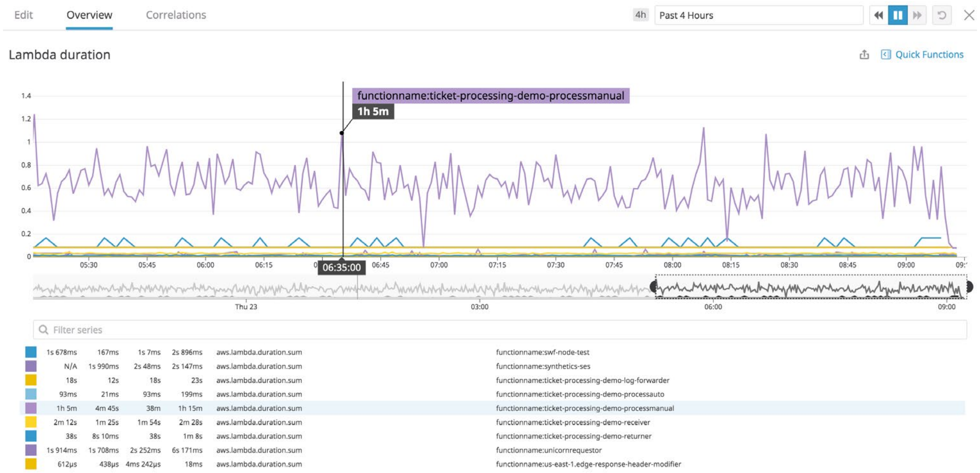
KEY FUNCTION PERFORMANCE AND UTILIZATION METRICS

Lambda automatically tracks the amount of time that a function is in use (performance) and how much memory a function uses during an invocation (utilization). Monitoring this data can help you optimize your functions and manage costs. Function utilization metrics are included in your Amazon CloudWatch logs, as seen in the following example log:

```
REPORT RequestId: f1d3fc9a-4875-4c34-b280-a5fae40abcf9 Duration:
72.51 ms      Billed Duration: 100 ms     Memory Size: 128 MB  Max
Memory Used: 58 MB   Init Duration: 2.04 ms
```

DURATION AND BILLED DURATION

Monitoring a function's execution time (i.e., its duration), can help you determine which functions can (or should) be optimized. Slow code execution could be the result of cold starts—an initial delay in response time for an inactive Lambda function—overly complex code, or network latency if your function relies on third-party or other AWS services. Lambda limits a function's total execution time to 15 minutes before it will terminate it and throw a timeout error, so monitoring duration helps you see when you are about to reach this threshold.



🟦	1s 678ms	167ms	1s 7ms	2s 896ms	aws.lambda.duration.sum	functionname:swf-node-test
🟪	N/A	1s 990ms	2s 48ms	2s 147ms	aws.lambda.duration.sum	functionname:synthetics-ses
🟨	18s	12s	18s	23s	aws.lambda.duration.sum	functionname:ticket-processing-demo-log-forwarder
🟦	93ms	21ms	93ms	199ms	aws.lambda.duration.sum	functionname:ticket-processing-demo-processauto
🟪	1h 5m	4m 45s	38m	1h 15m	aws.lambda.duration.sum	functionname:ticket-processing-demo-processmanual
🟨	2m 12s	1m 25s	1m 54s	2m 28s	aws.lambda.duration.sum	functionname:ticket-processing-demo-receiver
🟦	38s	8s 10ms	38s	1m 8s	aws.lambda.duration.sum	functionname:ticket-processing-demo-returner
🟪	1s 914ms	1s 708ms	2s 252ms	6s 171ms	aws.lambda.duration.sum	functionname:unicornrequestor
🟨	612μs	438μs	4ms 242μs	18ms	aws.lambda.duration.sum	functionname:us-east-1.edge-response-header-modifier

The **billed duration** measures the execution time rounded up to the nearest 100 ms. Billed duration is the basis for AWS's Lambda pricing, along with the function's memory size, which we will talk about next.

You can compare a function's duration with its billed duration to see if you can decrease execution time and lower costs. For instance, let's look at this function's log:

```
REPORT RequestId: f1d3fc9a-4875-4c34-b280-a5fae40abcf9 Duration:
102.25 ms     Billed Duration: 200 ms     Memory Size: 128 MB  Max
Memory Used: 120 MB  Init Duration: 2.04 ms
```

The function's duration was 102 ms, but what you will pay for is based on the 200 ms billed duration. If you notice the duration is consistent (e.g., around 102 ms), you may be able to add more memory in order to decrease the duration and the billed duration. For example, if you increase your function's memory from 128 MB to 192 MB and the duration drops to 98 ms, your billed duration would then be 100 ms. This means you would be charged less because you are in the 100 ms block instead of the 200 ms block for billed duration. Though we used a simple example, monitoring these two metrics is important for understanding the costs of your functions, especially if you are managing large volumes of requests across hundreds of functions.

MEMORY SIZE AND MAX MEMORY USED

A function's duration and billed duration are partially affected by how much memory it has—slower execution times may be a result of not having enough memory to process requests. Or, you may allocate more memory than your function needs. Both scenarios affect costs, so tracking memory usage can help you strike a balance between processing power and execution time. You can allot memory for your function, which Lambda logs refer to as its memory size, within AWS Lambda quotas.

You can compare a function's memory usage with its allocated memory in your CloudWatch logs, as seen below:

```
REPORT RequestId: f1d3fc9a-4875-4c34-b280-a5fae40abcf9 Duration:
102.25 ms     Billed Duration: 200 ms     Memory Size: 512 MB  Max
Memory Used: 58 MB   Init Duration: 2.04 ms
```

You can see that the function uses (`Max Memory Used`) only a fraction of its allocated memory. If this happens consistently, you may want to adjust the function's memory size to reduce costs. On the other hand, if a function's memory usage is consistently reaching its memory size then it doesn't have enough memory to process incoming requests, increasing execution times.

Key Function Invocation Metrics

You can invoke a Lambda function in one of three ways: synchronously, asynchronously, or via an event source mapping.

Synchronous services create the event, which Lambda passes directly to a function and waits for the function to return a response before passing the result back to the service. This is useful if you need the results of a function before moving on to the next step in the application workflow. If an error occurs, the AWS service that originally sent Lambda the event will retry the invocation.

Asynchronous invocation means that, when a service invokes a function, it immediately hands off the invocation event for Lambda to add to a queue. As soon as the service receives a response that the event was added to the queue successfully, it moves on to the next request. Asynchronous invocations can help decrease wait times for a service because it doesn't need to wait for Lambda to finish processing a request If a function returns an error—as a result of a timeout or an issue in the function code—Lambda will retry processing the event up to two times before discarding it. Lambda may also return events to the queue—and throw an error—if the function doesn't have enough concurrency to process them.

You can also use event source mapping to link an event source, such as Amazon Kinesis or DynamoDB streams, to a Lambda function. Mappings are resources that configure data streams or queues to serve as a trigger for a Lambda function. For example, when you map a Kinesis data stream to a function, the Lambda runtime will read batches of events, or records, from the shards (i.e., the sequences of records) in a stream and then send those batches to the function for processing. By default, if a function returns an error and cannot process a batch, it will retry the batch until it is successful or the records in the batch expire (for a data stream). To ensure a function doesn't stall when processing records, you can configure the number of retry attempts, the maximum age of a record in a batch, and the size of a batch when you create the event source mapping.

Each of these invocation methods results in some different metrics to monitor in addition to standard metrics that apply to all invocation types, such as invocation count and iterator age.

INVOCATIONS

Monitoring invocations can help you understand application activity and how your functions are performing overall. Anomalous changes in invocation counts could indicate either an issue with a function's code or a connected AWS service. For example, an outage for a function's downstream service could force multiple retries, increasing the function's invocation count.

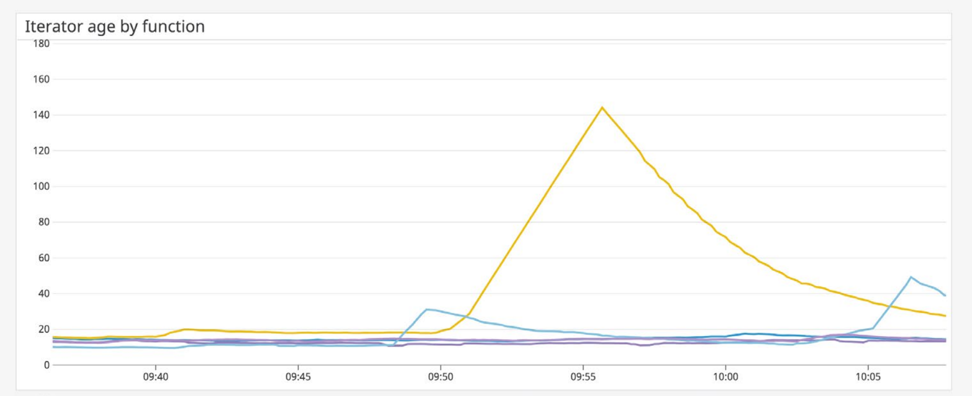
Additionally, if your functions are located in multiple regions, you can use the invocation count to determine if functions are running efficiently, with minimal latency. For example, you can quickly see which functions are invoked most frequently in which region and evaluate if you need to move them to another region or availability zone or modify load balancing in order to improve latency. Services like Lambda@Edge can improve latency by automatically running your code in regions that are closer to your customers.

ITERATOR AGE

Lambda emits the iterator age metric for stream-based invocations. The iterator age is the time between when the last record in a batch was written to a stream (e.g., Kinesis, DynamoDB) and when Lambda received the batch, letting you know if the amount of data that is being written to a stream is too much for a function to accept for processing.
There are a few scenarios that could increase the iterator age:

- a high execution duration for a function
- not enough shards in a stream
- invocation errors
- insufficient batch size

If you see the iterator age increase, it could mean the function is taking too long to process a batch of data and your application is building a large backlog of unprocessed events.



To decrease the iterator age, you need to decrease the time it takes for a Lambda function to process records in a batch. Long durations could result from not having enough memory for the function to operate efficiently. You can allocate more memory to the function or find ways to optimize your function code.

Adjusting a stream's batch size, which determines the maximum number of records that can be batched for processing, can also help decrease the iterator age in some cases. If a batch consists of mostly calls to simply trigger another downstream service, increasing the batch size allows functions to process more records in a single invocation, increasing throughput. However, if a batch contains records that require additional processing, then you may need to reduce the batch size to avoid stalled shards.

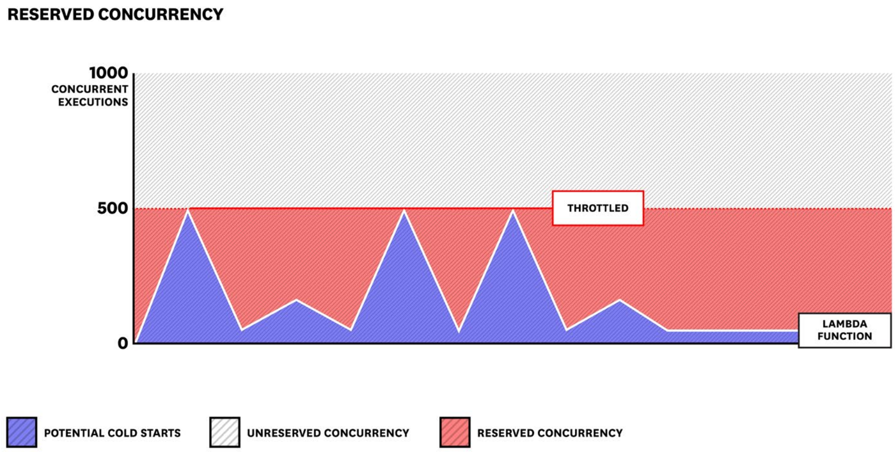
Another key component for monitoring a function's iterator age is tracking invocation errors—which you can view in Lambda's logs. Invocation errors can affect the time it takes for a function to process an event. If a batch of records consistently generates an error, the function cannot continue to the next batch, which increases the iterator age. Invocation errors may indicate issues with a stream accessing the function (e.g., incorrect permissions) or exceeding Lambda's concurrent execution limit.

Monitoring Function Concurrency

A function's concurrency is a measure of how many invocations that function can handle at one time. When a service invokes a function for the first time, the Lambda runtime creates a new instance of the function to process an event. If the service invokes a function while it is still processing an event, Lambda creates another instance. This cycle continues until there are enough function instances to serve incoming requests, or a function reaches its concurrency limit and is throttled.

By default, Lambda provides an initial pool of 1,000 concurrent executions per region, which are shared by all of your functions in that region. You can increase the per-region limit by submitting a request to AWS support. Lambda also requires the per-region concurrency pool to always have at least 100 available concurrent executions for all of your functions at all times. Monitoring concurrency along with a function's invocation count can help you manage overprovisioned functions and scale your functions to support the flow of application traffic. A burst of new invocations, for instance, could throttle your function if it does not have enough concurrency to process incoming traffic.

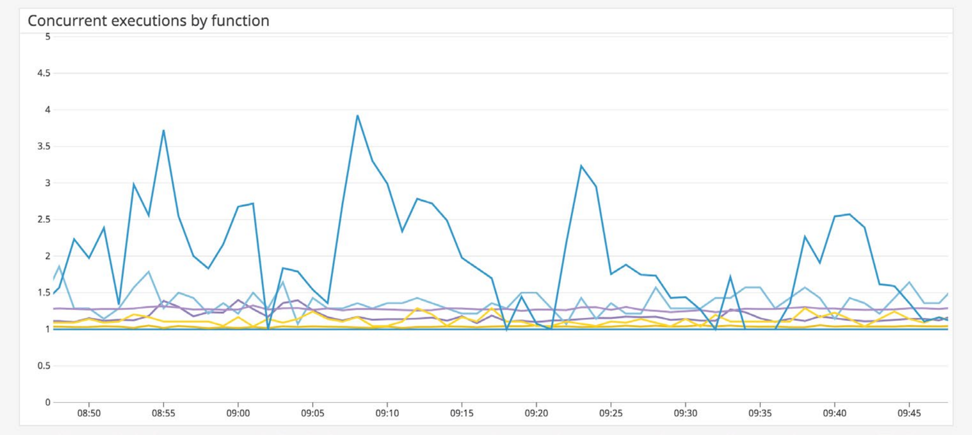
Lambda automatically scales function instances based on the number of incoming requests, though there is a limit on how many instances can be created during an initial burst. Once that limit is reached (between 500 and 3,000 instances, depending on your region), functions will scale at a rate of 500 instances per minute until they exhaust all available concurrency. This can come from the per-region concurrent executions limit or a function's reserved concurrency, which is the portion of the available pool of concurrent executions that you allocate to one or more functions. You can configure reserved concurrency to ensure that functions have enough concurrency to scale—or that they don't scale out of control and hog the concurrency pool.



Reserving concurrency for a function is useful if you know that function regularly requires more concurrency than others. You can also reserve concurrency to ensure that a function doesn't process too many requests and overwhelm a downstream service. Note that if a function uses all of its reserved concurrency, it will not access additional concurrency from the unreserved pool. Make sure you only reserve concurrency for your function(s) if it does not affect the performance of your other functions, as doing so will reduce the size of the available concurrency pool.

CONCURRENT EXECUTIONS

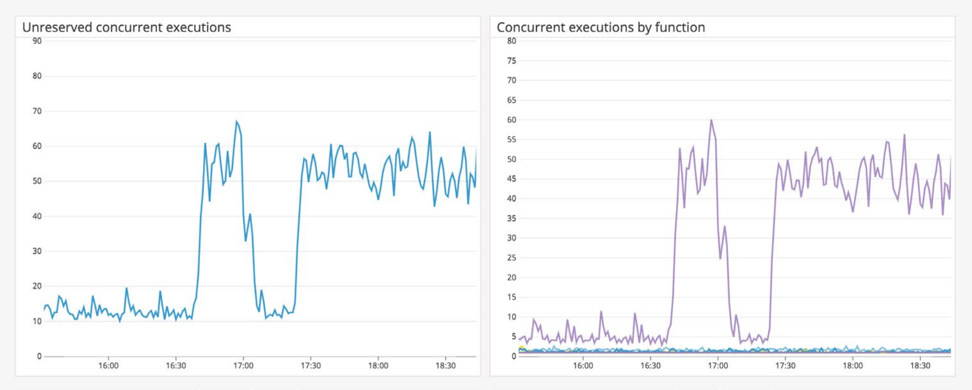
In order to monitor concurrency, Lambda emits the concurrent executions metric. This metric allows you to track when functions are using up all of the concurrency in the pool.



In the example above, you can see a spike in executions for a specific function. As mentioned previously, you can limit concurrent executions for a function by reserving concurrency from the common execution pool. This can be useful if you need to ensure that a function doesn't process too many requests simultaneously. However, keep in mind that Lambda will throttle the function if it uses all of its reserved concurrency.

UNRESERVED CONCURRENT EXECUTIONS

Unreserved executions are equivalent to the total number of available concurrent executions for your account, minus any reserved concurrency. You can compare the unreserved concurrent executions metric with the concurrent executions metric to monitor which functions are exhausting the remaining concurrency pool during heavier workloads.

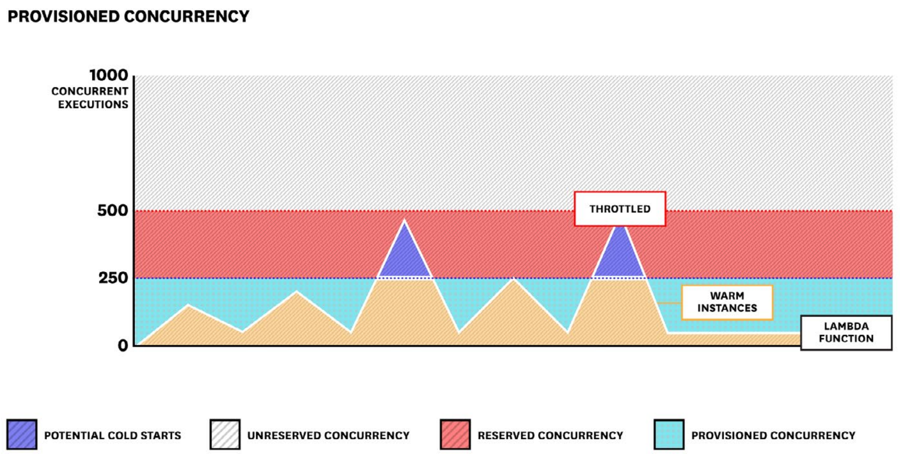


The graphs above show a spike in unreserved concurrency and one function using most of the available concurrency. This could be due to an upstream service sending too many requests to the function.

Monitoring Provisioned Concurrency

Since Lambda only runs your function code when needed, you may notice additional latency (cold starts) if your functions haven't been used in a while. This is because Lambda needs to initialize a new container and provision packaged dependencies for any inactive functions. Each initialization can add several seconds of latency to function execution. Lambda will keep containers alive for approximately 45 minutes, though that time may vary depending on your region or if you are using VPCs.

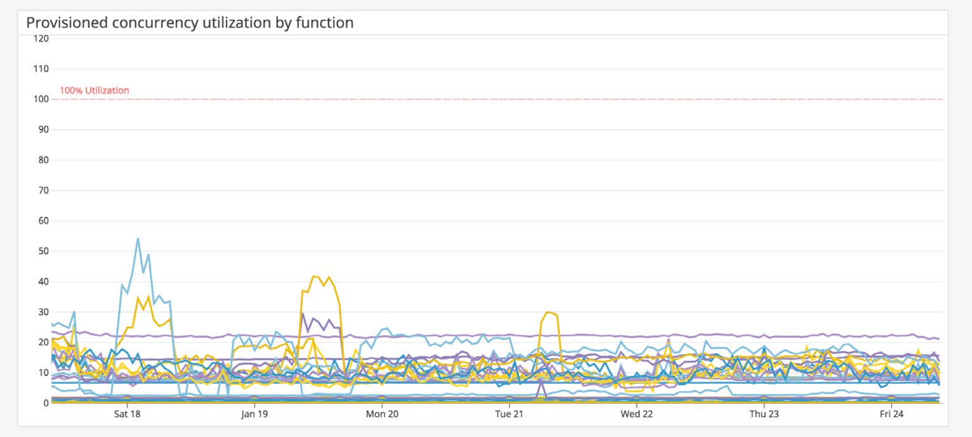
If a function has a long startup time (e.g., it has a large number of dependencies), requests may experience higher latency—especially if Lambda needs to initialize new instances to support a burst of requests. You can mitigate this by using provisioned concurrency, which automatically keeps function instances pre-initialized so that they'll be ready to quickly process requests.



Allocating a sufficient level of provisioned concurrency (e.g., the number of warm instances) for a function helps reduce the likelihood that it will encounter cold starts, which can be critical for applications that experience bursts in traffic during specific times of the day (e.g., a food delivery application). You can manage provisioned concurrency with Application Auto Scaling, which enables you to automatically adjust concurrency based on a scaling schedule or utilization to prepare for incoming traffic. Keep in mind that provisioned concurrency comes out of your account's regional concurrency pool and uses a different pricing model.

PROVISIONED CONCURRENCY UTILIZATION

One key metric for monitoring the efficiency of a function's provisioned concurrency is provisioned concurrency utilization. A function that is using up all of its available provisioned concurrency—its utilization threshold—may need additional concurrency. Or, if utilization is consistently low, you may have overprovisioned a function. You can disable or reduce provisioned concurrency for that function to manage costs.

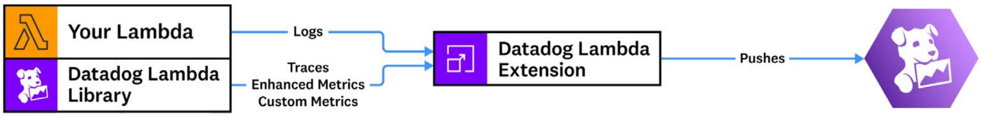


Using Datadog to Monitor AWS Lambda

Serverless applications introduce a new set of challenges for monitoring. You should be mindful of how incoming requests and other services interact with your functions, as well as how resilient your functions are to high demand or errors. For example, an influx of new requests could cause AWS to throttle your function if it does not have enough concurrency to process that traffic. Or, errors from an upstream service could stop your function code from executing.

To effectively monitor serverless applications, you need visibility into your entire serverless architecture so that you can understand how your functions and other AWS services are interoperating. Datadog provides full visibility into the state of your serverless applications in one place. You can identify service bottlenecks with end-to-end tracing, track custom metrics, correlate data, and get insight into function performance.

At a high level, Datadog provides a built-in AWS integration that collects CloudWatch data from all of your AWS services, including those used for your serverless applications (e.g., Lambda, Fargate, API Gateway, Step Functions). For deeper insights into your serverless functions, Datadog uses a dedicated AWS Lambda Extension to submit telemetry data directly from your Lambda functions. The Lambda Extension generates enhanced metrics, such as billed duration, timeouts, and estimated cost, from your AWS function logs at a higher granularity than standard CloudWatch metrics. This capability gives you real-time visibility into function performance.



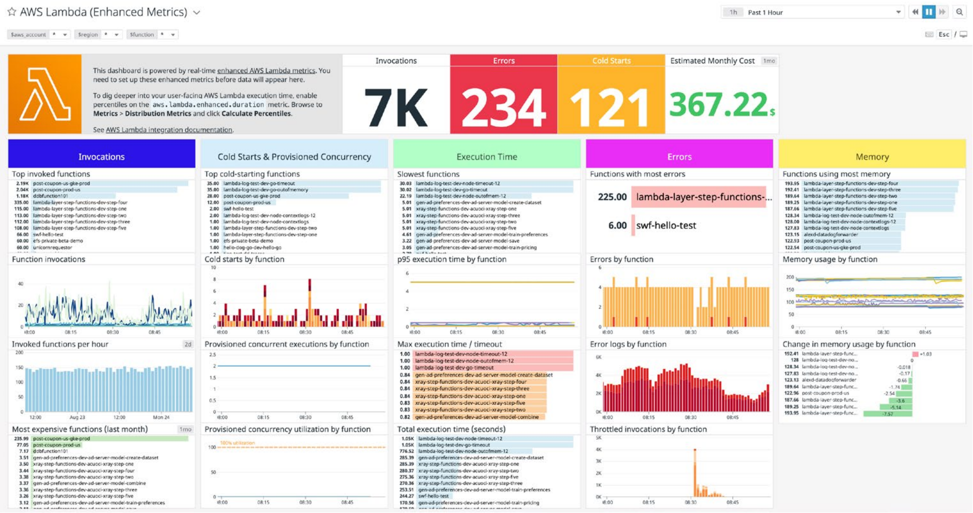
The Extension can also send custom metrics (synchronously or asynchronously), giving you additional insights into use cases that are unique to your application workflows, such as a user logging into your application, purchasing an item, or updating a user profile. Sending metrics asynchronously is recommended because it does not add any overhead to your code, making it an ideal solution for functions that power performance-critical tasks for your applications.

If you only need to collect Lambda logs, you can use Datadog's dedicated Lambda Layer and Forwarder. Datadog's Lambda Layer runs as a part of each function's runtime to automatically forward CloudWatch logs to the Datadog Forwarder, which then pushes them to Datadog. Deploying the Forwarder via CloudFormation is recommended as AWS will then automatically create the Lambda function with the appropriate role, add Datadog's Lambda Layer, and create relevant tags like `functionname`, `region`, and `account_id`, which you can then use in Datadog to search your logs.

Because the Forwarder is a Lambda function, it relies on triggers to execute. You can let Datadog automatically set these triggers up for you, or you can manually set them up to forward data as soon as they are added to S3 buckets or CloudWatch log groups. Once configured, Datadog's Lambda Forwarder will begin sending logs from Lambda (and any other AWS services you've configured) to your Datadog account.

VISUALIZE LAMBDA PERFORMANCE

Datadog provides out-of-the-box integration dashboards for your AWS infrastructure, including dashboards for both standard and enhanced Lambda metrics, giving you a high-level overview of how your serverless applications are performing.



This dashboard is powered by real-time enhanced AWS Lambda metrics. You need to set up these enhanced metrics before data will appear here.

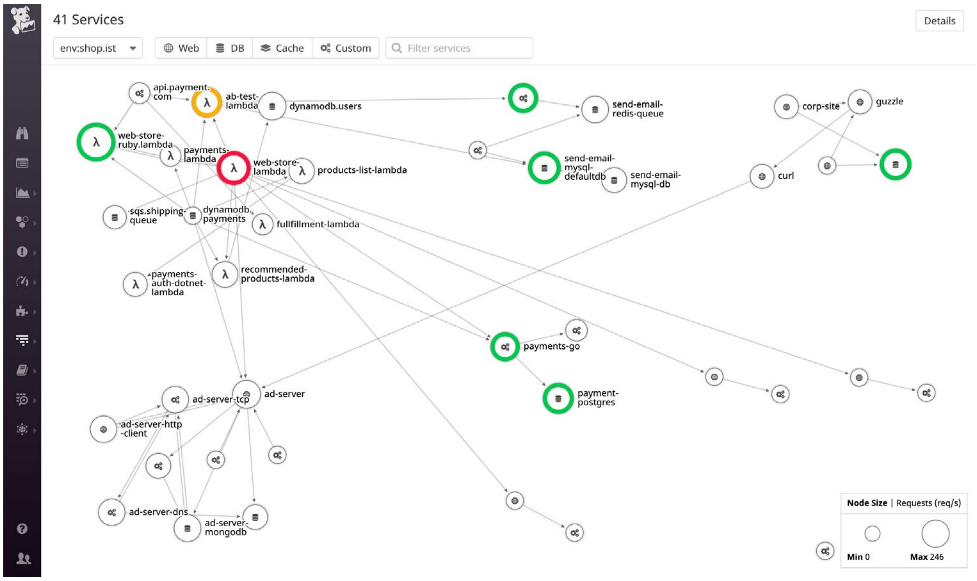
To dig deeper into your user-facing AWS Lambda execution time, enable percentiles on the `aws.lambda.enhanced.duration` metric. Browse to **Metrics** > **Distribution Metrics** and click **Calculate Percentiles**.

See AWS Lambda integration documentation.

AWS Lambda (Enhanced Metrics)
1h Past 1 Hour
$aws_account
$region
$function
Invocations 7K
Errors 234
Cold Starts 121
Estimated Monthly Cost 1mo 367.22$
Invocations
Cold Starts & Provisioned Concurrency
Execution Time
Errors
Memory
Top invoked functions
Top cold-starting functions
Slowest functions
Functions with most errors
225.00 lambda-layer-step-functions-...
6.00 swf-hello-test
Functions using most memory
Function invocations
Cold starts by function
p95 execution time by function
Errors by function
Memory usage by function
Invoked functions per hour
Provisioned concurrent executions by function
Max execution time / timeout
Error logs by function
Change in memory usage by function
Most expensive functions (last month)
Provisioned concurrency utilization by function
Total execution time (seconds)
Throttled invocations by function

For example, with the dashboard above, you can easily track cold starts, errors, and memory usage for all of your Lambda functions. You can customize your dashboards to include function logs and trace data, as well as metrics from any of your other services for easy correlation.

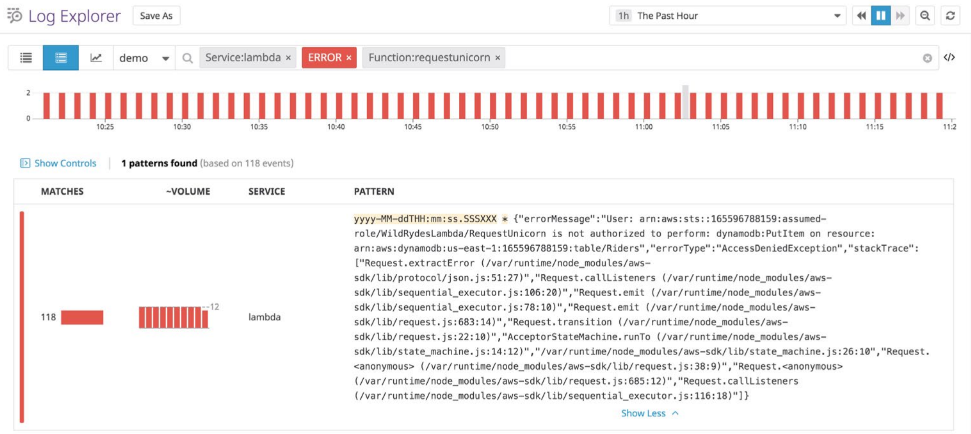
You can also use Datadog's Service Map to visualize all your serverless components in one place. This information helps you quickly understand the flow of traffic across upstream and downstream dependencies in your environment.



SEARCH AND ANALYZE SERVERLESS LOGS IN ONE PLACE

Lambda functions generate a large volume of logs, making it difficult to pinpoint issues during an incident or simply to monitor the current state of your functions. You can use Datadog's Log Patterns to help you surface interesting trends in your logs.

For example, if you notice a spike in Lambda errors on your dashboard, you can use Log Patterns to quickly search for the most common types of errors. In the example below, you can see a cluster of function logs recording an `AccessDeniedException` permissions error. The logs provide a stack trace so you can troubleshoot further.



```
yyyy-MM-ddTHH:mm:ss.SSSXXX * {"errorMessage":"User: arn:aws:sts::165596788159:assumed-
role/WildRydesLambda/RequestUnicorn is not authorized to perform: dynamodb:PutItem on resource:
arn:aws:dynamodb:us-east-1:165596788159:table/Riders","errorType":"AccessDeniedException","stackTrace":
["Request.extractError (/var/runtime/node_modules/aws-
sdk/lib/protocol/json.js:51:27)","Request.callListeners (/var/runtime/node_modules/aws-
sdk/lib/sequential_executor.js:106:20)","Request.emit (/var/runtime/node_modules/aws-
sdk/lib/sequential_executor.js:78:10)","Request.emit (/var/runtime/node_modules/aws-
sdk/lib/request.js:683:14)","Request.transition (/var/runtime/node_modules/aws-
sdk/lib/request.js:22:10)","AcceptorStateMachine.runTo (/var/runtime/node_modules/aws-
sdk/lib/state_machine.js:14:12)","/var/runtime/node_modules/aws-sdk/lib/state_machine.js:26:10","Request.
<anonymous> (/var/runtime/node_modules/aws-sdk/lib/request.js:38:9)","Request.<anonymous>
(/var/runtime/node_modules/aws-sdk/lib/request.js:685:12)","Request.callListeners
(/var/runtime/node_modules/aws-sdk/lib/sequential_executor.js:116:18)"]}
```

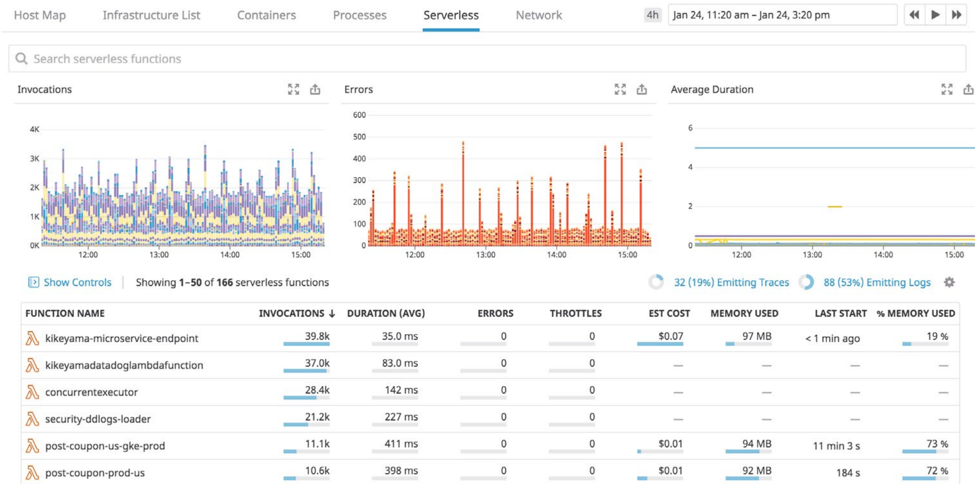
EXPLORE TRACE DATA USING DATADOG APM

Datadog's AWS Lambda Extension automatically propagates trace headers across services, providing end-to-end distributed tracing for your serverless applications. Datadog APM provides tracing libraries that you can use with the extension in order to natively trace request traffic across your serverless architecture. Traces are sent asynchronously, so they don't add any latency overhead to your serverless applications.

Datadog also provides integrations for other services you may use with your serverless applications, such as AWS Fargate, Amazon API Gateway, Amazon SNS, and Amazon SQS. This ensures that you get visibility into every layer of your serverless architecture. With these integrations enabled, you can drill down to specific functions that are generating errors or cold starts to optimize their performance. AWS charges based on the time it takes for a function to execute, how much memory is allocated to each function, and the number of requests for your function. This means that your costs could quickly increase if, for instance, a high-volume function makes a call to an API Gateway service experiencing network outages and has to wait for a response.

With tracing, you can map upstream and downstream dependencies such as API Gateway and trace requests across your entire stack to pinpoint any latency bottlenecks. The extension also allows you to analyze serverless logs to quickly identify the types of errors your functions generate.

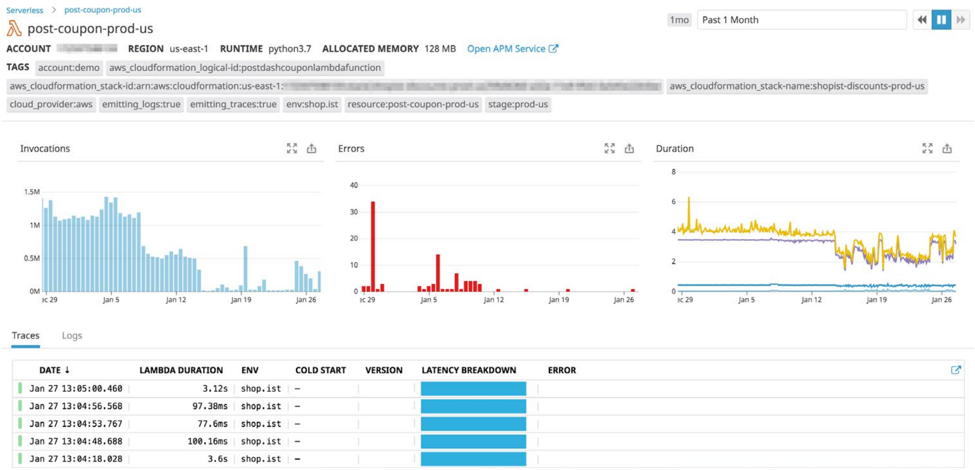
To start analyzing trace data from your serverless functions, you can use Datadog's Serverless view. This view gives a comprehensive look at all of your functions and includes key metrics such as invocation count and memory usage. You can search for a specific function or view performance metrics across all your functions. You can also sort your functions in the Serverless view by specific metrics to help surface functions that use the most memory, or that are invoked the most, as seen in the example below.



Showing **1–50** of **166** serverless functions

FUNCTION NAME	INVOCATIONS ↓	DURATION (AVG)	ERRORS	THROTTLES	EST COST	MEMORY USED	LAST START	% MEMORY USED
kikeyama-microservice-endpoint	39.8k	35.0 ms	0	0	$0.07	97 MB	< 1 min ago	19 %
kikeyamadatadoglambdafunction	37.0k	83.0 ms	0	0	—	—	—	—
concurrentexecutor	28.4k	142 ms	0	0	—	—	—	—
security-ddlogs-loader	21.2k	227 ms	0	0	—	—	—	—
post-coupon-us-gke-prod	11.1k	411 ms	0	0	$0.01	94 MB	11 min 3 s	73 %
post-coupon-prod-us	10.6k	398 ms	0	0	$0.01	92 MB	184 s	72 %

When you click on a function, you will see all of its associated traces and logs, as well as a detailed breakdown of information for each invocation such as duration, related error messages, and whether the function experienced a cold start during the invocation.



post-coupon-prod-us

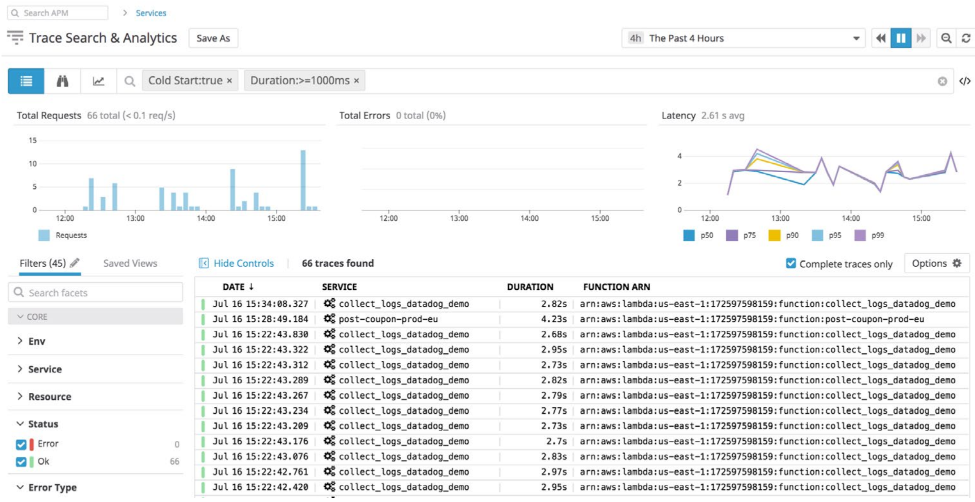
ACCOUNT [redacted] REGION us-east-1 RUNTIME python3.7 ALLOCATED MEMORY 128 MB

TAGS account:demo aws_cloudformation_logical-id:postdashcouponlambdafunction aws_cloudformation_stack-id:arn:aws:cloudformation:us-east-1:[redacted] aws_cloudformation_stack-name:shopist-discounts-prod-us cloud_provider:aws emitting_logs:true emitting_traces:true env:shop.ist resource:post-coupon-prod-us stage:prod-us

DATE ↓	LAMBDA DURATION	ENV	COLD START	VERSION	LATENCY BREAKDOWN	ERROR
Jan 27 13:05:00.460	3.12s	shop.ist	–			
Jan 27 13:04:56.568	97.38ms	shop.ist	–			
Jan 27 13:04:53.767	77.6ms	shop.ist	–			
Jan 27 13:04:48.688	100.16ms	shop.ist	–			
Jan 27 13:04:18.028	3.6s	shop.ist	–			

API latency and cold starts are two common issues with serverless functions, both of which can significantly increase a function's execution time. Cold starts typically occur when functions scale behind the scenes to handle more requests. API latency could be a result of network or other service outages. Datadog enables you to proactively monitor latency and cold starts for all your functions.

For example, you can create an anomaly alert to notify you when a function experiences unusual latency. From the triggered alert, you can pivot to traces and logs to determine if the latency was caused by cold starts or degradation in an API service dependency. Datadog also automatically detects when cold starts occur and applies a `cold_start` tag to your traces, so you can easily surface functions that are experiencing cold starts to troubleshoot further.



Trace Search & Analytics

Cold Start:true × Duration:>=1000ms ×

Total Requests 66 total (< 0.1 req/s)

Total Errors 0 total (0%)

Latency 2.61 s avg

☑ Error 0
☑ Ok 66

66 traces found

☑ Complete traces only

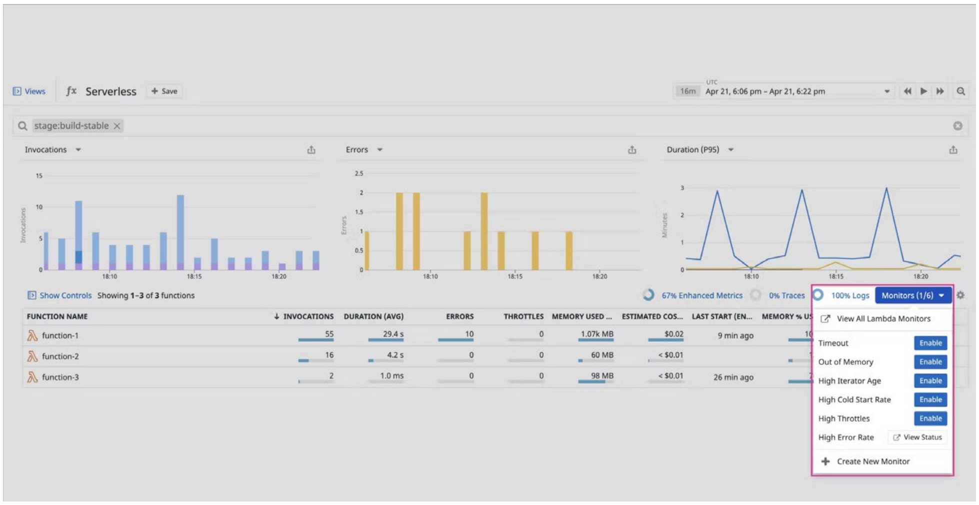
DATE ↓	SERVICE	DURATION	FUNCTION ARN
Jul 16 15:34:08.327	collect_logs_datadog_demo	2.82s	arn:aws:lambda:us-east-1:172597598159:function:collect_logs_datadog_demo
Jul 16 15:28:49.184	post-coupon-prod-eu	4.23s	arn:aws:lambda:us-east-1:172597598159:function:post-coupon-prod-eu
Jul 16 15:22:43.830	collect_logs_datadog_demo	2.68s	arn:aws:lambda:us-east-1:172597598159:function:collect_logs_datadog_demo
Jul 16 15:22:43.322	collect_logs_datadog_demo	2.95s	arn:aws:lambda:us-east-1:172597598159:function:collect_logs_datadog_demo
Jul 16 15:22:43.312	collect_logs_datadog_demo	2.73s	arn:aws:lambda:us-east-1:172597598159:function:collect_logs_datadog_demo
Jul 16 15:22:43.289	collect_logs_datadog_demo	2.82s	arn:aws:lambda:us-east-1:172597598159:function:collect_logs_datadog_demo
Jul 16 15:22:43.267	collect_logs_datadog_demo	2.79s	arn:aws:lambda:us-east-1:172597598159:function:collect_logs_datadog_demo
Jul 16 15:22:43.234	collect_logs_datadog_demo	2.77s	arn:aws:lambda:us-east-1:172597598159:function:collect_logs_datadog_demo
Jul 16 15:22:43.209	collect_logs_datadog_demo	2.73s	arn:aws:lambda:us-east-1:172597598159:function:collect_logs_datadog_demo
Jul 16 15:22:43.176	collect_logs_datadog_demo	2.7s	arn:aws:lambda:us-east-1:172597598159:function:collect_logs_datadog_demo
Jul 16 15:22:43.076	collect_logs_datadog_demo	2.83s	arn:aws:lambda:us-east-1:172597598159:function:collect_logs_datadog_demo
Jul 16 15:22:42.761	collect_logs_datadog_demo	2.97s	arn:aws:lambda:us-east-1:172597598159:function:collect_logs_datadog_demo
Jul 16 15:22:42.420	collect_logs_datadog_demo	2.95s	arn:aws:lambda:us-east-1:172597598159:function:collect_logs_datadog_demo

If a function's increased execution time is a result of too many cold starts, you can configure Lambda to reduce initialization latency by using provisioned concurrency. Latency from an API service, on the other hand, may be a result of cross-region calls. In that case, you may need to colocate your application resources into the same AWS region.

ALERT ON CRITICAL AWS LAMBDA METRICS

Monitoring Lambda enables you to visualize trends and identify issues during critical outages, but it’s easy to overlook an issue when you are monitoring a large volume of data in complex infrastructures. In order to ensure that you are aware of critical issues affecting your applications, you can create alerts to get notified about key issues detected in your Lambda metrics, logs, or traces.

Datadog provides a list of built-in alerts you can enable from the Serverless view to automatically notify you of critical performance issues with minimal configuration, such as a sudden increase in cold starts or out-of-memory errors.



FUNCTION NAME	↓ INVOCATIONS	DURATION (AVG)	ERRORS	THROTTLES	MEMORY USED ...	ESTIMATED COS...	LAST START (EN...	MEMORY % US
function-1	55	29.4 s	10	0	1.07k MB	$0.02	9 min ago	10
function-2	16	4.2 s	0	0	60 MB	< $0.01		1
function-3	2	1.0 ms	0	0	98 MB	< $0.01	26 min ago	7

There are also several alert types that you can use for creating custom alerts for your specific use case, so you can be notified about only the issues you care about. For example, you can create an alert to notify you if a function has been throttled frequently over a specific period of time. If you configure the alert to automatically trigger separate notifications per affected function, this saves you from creating duplicate alerts and enables you to get continuous, scalable coverage of your environment, no matter how many functions you’re running. Throttles occur when there is not enough capacity for a function, either because available concurrency is used up or because requests are coming in faster than the function can scale.

Full Visibility into Your Serverless Ecosystem

In this chapter, we looked at some key metrics for monitoring your serverless applications as well as how to troubleshoot common serverless problems. AWS provides a comprehensive suite of tools that enable you to focus more on building scalable services and less on provisioning or managing infrastructure resources. Datadog offers deep visibility into these serverless applications, enabling you to easily collect observability data through built-in service integrations and a dedicated Lambda extension. This enables you to visualize your serverless metrics with integration dashboards, sift through logs with Datadog's Log Explorer, and analyze distributed request traces with Datadog APM.

Chapter 9: Datadog Is Dynamic, Cloud-Scale Monitoring

In the preceding chapters we demonstrated how Datadog can help you track the health and performance of Amazon Elastic Load Balancing, Docker, and all their associated infrastructure. Whatever technologies you use, Datadog enables you to view and analyze metrics and events from across your infrastructure.

Datadog was built to meet the unique needs of modern, cloud-scale infrastructure:

- **Comprehensive monitoring.** Out of the box, Datadog collects monitoring data from more than 150 popular technologies. The Datadog Agent also includes a lightweight metrics aggregation server that can collect custom metrics from virtually any application.

- **Flexible aggregation.** Datadog's native support for tagging allows you to aggregate metrics and events on the fly to generate the views that matter most. Tagging allows you to monitor services rather than hosts so you can focus on the performance metrics that directly impact your users and your business.

- **Effortless scaling.** Datadog scales automatically with your infrastructure, whether you have tens, hundreds, or thousands of hosts. Datadog auto-enrolls new hosts and containers as they come online, and with service discovery you can continuously monitor your containerized services wherever they run.

- **Sophisticated alerting.** Virtually any type of monitoring data can be used to trigger a Datadog alert. Datadog can alert on fixed or dynamic metric thresholds, outliers, events, status checks, and more.

- **Collaboration baked in.** Datadog helps teams stay on the same page with easily sharable dashboards, graphs, and annotated snapshots. Seamless integrations with industry-leading collaboration tools such as PagerDuty, Slack, and HipChat make conversations around monitoring data as frictionless as possible.

If you are ready to apply the monitoring and visualization principles you've learned in this book, you can sign up for a full-featured Datadog trial at www.datadog.com

Above all, we hope that you find the information in this book to be instructive as you set out to implement monitoring for your infrastructure, or to improve on existing practices. We have found the frameworks outlined in these chapters to be extremely valuable in monitoring and scaling our own dynamic infrastructure, and we hope that you find them equally useful. Please get in touch with us by email (info@datadoghq.com) or on Twitter (@datadoghq) if you have questions or comments about this book or about Datadog.

Happy monitoring!

DATADOG