DevOps measurement: Monitoring and observability

star**Note:** *Monitoring and observability* is one of a set of capabilities that drive higher software delivery and organizational performance. These capabilities were discovered by the DORA State of DevOps research program (https://www.devops-research.com/research.html), an independent, academically rigorous investigation into the practices and capabilities that drive high performance. To learn more, read our DevOps resources (/devops).

Good monitoring is a staple of high-performing teams. DevOps Research and Assessment (DORA) (/devops) research (https://services.google.com/fh/files/misc/state-of-devops-2018.pdf) shows that a comprehensive monitoring and observability solution, along with a number of other technical practices, positively contributes to continuous delivery. (/architecture/devops/devops-tech-continuous-delivery)

DORA's research defined these terms as follows:

Monitoring is tooling or a technical solution that allows teams to watch and understand the state of their systems. Monitoring is based on gathering predefined sets of metrics or logs.

Observability is tooling or a technical solution that allows teams to actively debug their system. Observability is based on exploring properties and patterns not defined in advance.

To do a good job with monitoring and observability, your teams should have the following:

- Reporting on the overall health of systems (Are my systems functioning? Do my systems have sufficient resources available?).
- Reporting on system state as experienced by customers (Do my customers know if my system is down and have a bad experience?).
- Monitoring for key business and systems metrics.
- Tooling to help you understand and debug your systems in production.
- Tooling to find information about things you did not previously know (that is, you can identify *unknown unknowns*).
- Access to tools and data that help trace, understand, and diagnose infrastructure problems in your production environment, including interactions between services.

How to implement monitoring and observability

Monitoring and observability solutions are designed to do the following:

- Provide leading indicators of an outage or service degradation.
- Detect outages, service degradations, bugs, and unauthorized activity.
- Help debug outages, service degradations, bugs, and unauthorized activity.
- Identify long-term trends for capacity planning and business purposes.
- Expose unexpected side effects of changes or added functionality.

As with all DevOps capabilities, installing a tool is not enough to achieve the objectives, but tools can help or hinder the effort. Monitoring systems should not be confined to a single individual or team within an organization. Empowering all developers to be proficient with monitoring helps develop a culture of data-driven decision making and improves overall system debuggability, reducing outages.

There are a few keys to effective implementation of monitoring and observability. First, your monitoring should tell you what is broken and help you understand why, before too much damage is done. The key metric in the event of an outage or service degradation is time-to-restore (TTR). A key contributor to TTR is the ability to rapidly understand what broke and the quickest path to restoring service (which may not involve immediately remediating the underlying problems).

There are two high-level ways of looking at a system: *blackbox* monitoring, where the system's internal state and mechanisms are not made known, and *whitebox* monitoring, where they are.

For more information, see "Monitoring Distributed Systems" in the Site Reliability Engineering (https://landing.google.com/sre/sre-book/chapters/monitoring-distributed-systems/) book.

Blackbox monitoring

In a blackbox (or *synthetic*) monitoring system, input is sent to the system under examination in the same way a customer might. This might take the form of HTTP calls to a public API, or RPC calls to an exposed endpoint, or it might be calling for an entire web page to be rendered as a part of the monitoring process.

Blackbox monitoring is a sampling-based method. The same system that is responsible for user requests is monitored by the blackbox system. A blackbox system can also provide coverage of the target system's surface area. This could mean probing each external API method. You might also consider a representative mixture of requests to better mimic actual customer behavior. For example, you might perform 100 reads and only 1 write of a given API.

You can govern this process with a *scheduling system*, to ensure that these inputs are made at a sufficient rate in order to gain confidence in their sampling. Your system should also contain a *validation engine*, which can be as simple as checking response codes, or matching output with regular expressions, up to rendering a dynamic site in a headless browser and traversing its DOM tree, looking for specific elements. After a decision is made (pass, fail) on a given probe, you must store the result and metadata for reporting and alerting purposes. Examining a snapshot of a failure and its context can be invaluable for diagnosing an issue.

Whitebox monitoring

Monitoring and observability rely on signals sent from the workload under scrutiny into the monitoring system. This can generally take the form of the three most common components: *metrics*, *logs*, and *traces*. Some monitoring systems also track and report *events*, which can represent user interactions with an entire system, or state changes within the system itself.

Metrics are simply measurements taken inside a system, representing the state of that system in a measurable way. These are almost always numeric and tend to take the form of counters, distributions, and gauges. There are some cases where string metrics make sense, but generally numeric metrics are used due to the need to perform mathematical calculations on them to form statistics and draw visualizations.

Logs can be thought of as append-only files that represent the state of a single thread of work at a single point in time. These logs can be a single string like "User pushed button X" or a structured log entry which includes metadata such as the time the event happened, what server was processing it, and other environmental elements. Sometimes a system which cannot write structured logs will produce a semi-structured string like `[timestamp] [server] message [code]` which can be parsed after the fact, as needed. Log entries tend to be written using a client library like log4j, structlog, bunyan, log4net, or Nlog. Log processing can be a very reliable method of producing statistics that can be considered trustworthy, as they can be reprocessed based on immutable stored logs, even if the log processing system itself is buggy. Additionally, logs can be processed in real time to produce *log-based metrics*.

Traces are composed of *spans*, which are used to follow an event or user action through a distributed system. A span can show the path of a request through one server, while another span might run in parallel, both having the same parent span. These together form a trace, which is often visualized in a waterfall graph similar to those used in profiling tools. This lets developers understand time taken in a system, across many servers, queues, and network hops. A common framework for this is OpenTelemetry (https://opentelemetry.io/), which was formed from both OpenCensus (https://opencensus.io/) and OpenTracing (https://opentracing.io/).

Metrics, logs, and traces can be reported to the monitoring system by the server under measurement, or by an adjacent *agent* that can witness or infer things about the system.

Instrumentation

To make use of a monitoring system, your system must be *instrumented*. That is, code must be added to a system in order to expose its inner state. For example, if a simple program contains a pool of connections to another service, you might want to keep track of the size of that pool and the number of unused connections at any given time. In order to do so, a developer must write some code in the connection pool logic to keep track of when connections are formed or destroyed, when they are handed out, and when they are returned. This might take the form of log entries or events for each of these, or you might increment and decrement a gauge for the size of the queue, or you might increment a counter each time a connection is created, or each time a pool is expanded.

Correlation

Metrics can be collected from the application, as well as from its underlying systems, such as the JVM, guest OS, hypervisor, node OS, and the hardware itself. Note that as you go further down in a stack, you might start conflating metrics that are shared across workloads. For example, if a single machine serves several applications, watching the disk usage might not correspond directly to the system under observation. Correlating issues across applications on a shared system, however, can help you pin down a contributing factor (such as a slow disk). Drilling down from a single application to its underlying system metrics, then pulling up to show all similarly affected applications can be very powerful.

Measuring a distributed system means having observability in many places and being able to view them all together. This might mean both a frontend and its database, or it might mean a mobile application running on a customer's device, a cloud load balancer, and a set of

microservices. Being able to connect data from all of these sources in one place is a fundamental requirement in modern observability tools.

Computation

After you collect data from various sources for your system, you generate statistics and aggregate data across various realms. This might be cohorts of users, regions of your compute footprint, or geographic locations of your customers. Being able to develop these statistics on-the-fly based on raw events is very advantageous but can be costly both in terms of storage as well as real-time compute capacity required.

When you choose your tooling and plan your instrumentation, you must consider cardinality and dimensionality. These two aspects of metric collection can greatly affect your ability to scale your ability to observe a system.

Cardinality is the measure of distinct values in a system. For example, a field like `cpu-utilization` tends to need a range between 0 and 100. However, if you keep track of a user's unique identifier, they're all distinct, thus if you have 1M users, you have a cardinality of 1M. This makes a huge difference.

Dimensionality is the ability to record more than just a single value along with a timestamp, as you might have in a simple timeseries database that backs a monitoring system. Simply recording the value of a counter, for example requests-sent, might initially record just the value of the number of requests sent up to this point in time like `{time=x, value=y}`. However, as with structured logs, you might want to also record some environmental data, resulting in something like: `{time=x, value=y, server=foo, cluster=123, environment=prod, service=bar}`. Combining high cardinality and high dimensionality can result in dramatically increased compute and storage requirements, to the point where monitoring might not work as expected! This needs to be understood by developers who write dynamically generated data and metadata into monitoring systems.

Learning and improving

Part of operating a system is learning from outages and mistakes. The process of writing retrospectives or postmortems with corrective actions is well documented. One outcome of this process is the development of improved monitoring. It is critical to a fast-moving organization to allow their monitoring systems to be updated quickly and efficiently by anyone within the organization. Monitoring configuration is critical, so changes should be tracked by

means of review and approval (/architecture/devops/devops-process-streamlining-change-approval), just as with code development and delivery. Keeping your monitoring configuration in a version control system (/architecture/devops/devops-tech-version-control) is a good first step in allowing broad access to the system, while maintaining control on this critical part of your system. Developing automation around deploying monitoring configuration through an automation pipeline can also improve your ability to ensure these configurations are valid and applied consistently. After you treat your monitoring configuration as code, these improvements can all be accomplished by means of a deployment automation (/architecture/devops/devops-tech-deployment-automation) process, ideally the same system used by the rest of your team.

Common pitfalls of implementing monitoring and observability

When developing a system for monitoring and observability for your organization, you should be aware that there generally isn't a simple plug-and-play solution. Any decent monitoring system will require a deep understanding of each component that you want to measure, as well as direct manipulation of the code to instrument those systems. Avoid having a single monitoring person or dedicated team who is solely responsible for the system. This will not only help you prevent a single point of failure, but also increase your ability to understand and improve your system as an entire organization. Monitoring and observability needs to be built into the baseline knowledge of all your developers. A common pitfall here is for the operation team, NOC, or other similar team to be the only ones allowed to make changes to a monitoring system. This should be avoided and replaced with a system that follows CD patterns.

A common anti-pattern in writing alerts in monitoring systems is to attempt to enumerate all possible error conditions and write an alert for each of them. We call this *cause-based alerting,* and you should avoid it as much as possible. Instead, you should focus on *symptom-based alerting,* which only alerts you when a user-facing symptom is visible or is predicted to arise soon. You should still be able to observe non-user-facing systems, but they shouldn't be directly alerting on-call engineers if there are no user-facing symptoms. Note that the term *user-facing* can also include users internal to your organization.

When generating alerts, you should consider how they are delivered. Your alerts should have multiple pathways to oncall engineers, including but not limited to: SMS delivery, dedicated mobile apps, automated phone calls, or email. A common pitfall is to email alerts to an entire

team via an email distribution list. This can quickly result in ignored alerts due to diffusion of responsibility (https://wikipedia.org/wiki/Diffusion_of_responsibility). Another common failure is simply a poor signal-to-noise ratio (https://wikipedia.org/wiki/Signal-to-noise_ratio). If too many alerts are not actionable, or result in no improvement, the team will easily miss those alerts that are meaningful and possibly very important, a problem known as alarm fatigue (https://wikipedia.org/wiki/Alarm_fatigue). Any method to silence or suppress some set of alerts should be tracked very carefully to ensure it is not too broad or applied too often.

When building monitoring dashboards to visualize metrics, a common mistake is to spend a long time curating the "Perfect Dashboard". This is similar to the cause-based alerting mistake above. In a high-functioning team, the system under observation changes so fast that any time spent curating a dashboard will be out of date before it is done. Instead, focusing on the ability for team members to quickly create a dashboard or other set of visualizations that fits their needs is important.

Failing to separate out product or executive-facing metrics such as user acquisition rate and revenue tracking away from operational or service health dashboards is also a common problem, as they are both very important, but distinct. Keeping these separate is highly recommended.

How to measure monitoring and observability

When implementing a monitoring and observability system in your organization, you can track some internal metrics to see how well you're doing. Here are some that you might wish to track with a monthly survey, or possibly by automatically analyzing your postmortems or alerting logs.

- **Changes made to monitoring configuration.** How many pull requests or changes per week are made to the repository containing the monitoring configuration? How often are these changes pushed to the monitoring system? (Daily? Batched? Immediately on PR?)
- **"Out of hours" alerts.** What percentage of alerts are handled at night? While some global businesses have a follow-the-sun support model which makes this a non-issue, it can be an indication that not enough attention has been paid to leading indicators of failures. Regular night-time alerts can lead to alert fatigue and burned-out teams.
- **Team alerting balance.** If you have teams in different locations responsible for a service, are alerts fairly distributed and addressed by all teams? If not, why?

- **False positives.** How many alerts resulted in no action, or were marked as "Working as Intended"? Alerts which aren't actionable and which haven't helped you predict failures should be deleted.
- **False negatives.** How many system failures happened with no alerting, or alerting later than expected? How often do your postmortems include adding new (symptom-based) alerts?
- **Alert creation.** How many alerts are created per week (total, or grouped by severity, team, etc.)?
- **Alert acknowledgement.** What percentage of alerts are acknowledged within the agreed deadline (such as 5 minutes, 30 minutes)? Sometimes this is coupled with or can be tracked by a metric like *alert fall-through* when a secondary oncall person is notified for an alert.
- **Alert silencing and silence duration.** How many alerts are in a silenced or suppressed state per week? How many are added to this pool, how many removed? If your alert silencing has an expiration system, how many silences are extended to last longer than initially expected? What is the mean and maximum silence period? (A fun one is "how many silences are effectively 'infinite'"?)
- **Unactionable alerts.** What percentage of alerts were considered "unactionable"? That is, the engineer alerted was not able to immediately take some action, either out of an inability to understand the alert implication, or due to a known issue. Unactionable alerts are a well known source of toil (https://landing.google.com/sre/sre-book/chapters/eliminating-toil/).
- **Usability: alerts, runbooks, dashboards.** How many graphs are on your dashboards? How many lines per graph? Can teams understand the graphs? Is there explanatory text to help out new engineers? Do people have to scroll and browse a lot to find the information they need? Can engineers navigate from alert to playbook to dashboards effectively? Are the alerts named in such a way to point engineers in the right direction? These might be measured by surveys of the team, over time.
- **MTTD, MTTR, impact**. The bottom line is time to detect, time to resolve, and impact. Consider measuring the "area under the curve" of the time that the outage was affecting customers times the number of customers affected. This can be estimated or done more precisely with tooling.

By tracking some or all of these metrics, you'll start to gain a better understanding of how well your monitoring and observability systems are working for your organization. Breaking these measurements down by product, by operational team, or other methods will give you insight not only into the health of your products but also your processes and your people.

What's next

- For links to other articles and resources, see the DevOps page (/devops).
- Visit Google Site Reliability Engineering Resources (https://landing.google.com/sre/resources/) to find the following SRE books:
  - Site Reliability Engineering
  - Site Reliability Engineering Workbook
  - Building Secure & Reliable Systems
- Google Cloud Operations (/products/operations)
- Explore our DevOps research program (https://www.devops-research.com/research.html).
- Take the DevOps quick check (https://www.devops-research.com/quickcheck.html) to understand where you stand in comparison with the rest of the industry.

Last updated 2022-05-26 UTC.