Chapter 6 - Monitoring Distributed Systems

1. Table of Contents
2. Foreword
3. Preface
4. Part I - Introduction
5. 1. Introduction
6. 2. The Production Environment at Google, from the Viewpoint of an SRE
7. Part II - Principles
8. 3. Embracing Risk
9. 4. Service Level Objectives
10. 5. Eliminating Toil
11. 6. Monitoring Distributed Systems
12. 7. The Evolution of Automation at Google
13. 8. Release Engineering
14. 9. Simplicity
15. Part III - Practices
16. 10. Practical Alerting
17. 11. Being On-Call
18. 12. Effective Troubleshooting
19. 13. Emergency Response
20. 14. Managing Incidents
21. 15. Postmortem Culture: Learning from Failure
22. 16. Tracking Outages
23. 17. Testing for Reliability
24. 18. Software Engineering in SRE
25. 19. Load Balancing at the Frontend
26. 20. Load Balancing in the Datacenter
27. 21. Handling Overload
28. 22. Addressing Cascading Failures
29. 23. Managing Critical State: Distributed Consensus for Reliability
30. 24. Distributed Periodic Scheduling with Cron
31. 25. Data Processing Pipelines
32. 26. Data Integrity: What You Read Is What You Wrote
33. 27. Reliable Product Launches at Scale
34. Part IV - Management
35. 28. Accelerating SREs to On-Call and Beyond
36. 29. Dealing with Interrupts
37. 30. Embedding an SRE to Recover from Operational Overload
38. 31. Communication and Collaboration in SRE
39. 32. The Evolving SRE Engagement Model
40. Part V - Conclusions
41. 33. Lessons Learned from Other Industries
42. 34. Conclusion
43. Appendix A. Availability Table
44. Appendix B. A Collection of Best Practices for Production Services
45. Appendix C. Example Incident State Document
46. Appendix D. Example Postmortem
47. Appendix E. Launch Coordination Checklist
48. Appendix F. Example Production Meeting Minutes
49. Bibliography

Monitoring Distributed Systems

Written by Rob Ewaschuk
Edited by Betsy Beyer

Google’s SRE teams have some basic principles and best practices for building successful monitoring and alerting systems. This chapter offers guidelines for what issues should interrupt a human via a page, and how to deal with issues that aren’t serious enough to trigger a page.

Definitions

There’s no uniformly shared vocabulary for discussing all topics related to monitoring. Even within Google, usage of the following terms varies, but the most common interpretations are listed here.

Monitoring

> Collecting, processing, aggregating, and displaying real-time quantitative data about a system, such as query counts and types, error counts and types, processing times, and server lifetimes.

White-box monitoring

> Monitoring based on metrics exposed by the internals of the system, including logs, interfaces like the Java Virtual Machine Profiling Interface, or an HTTP handler that emits internal statistics.

Black-box monitoring

> Testing externally visible behavior as a user would see it.

Dashboard

> An application (usually web-based) that provides a summary view of a service’s core metrics. A dashboard may have filters, selectors, and so on, but is prebuilt to expose the metrics most important to its users. The dashboard might also display team information such as ticket queue length, a list of high-priority bugs, the current on-call engineer for a given area of responsibility, or recent pushes.

Alert

> A notification intended to be read by a human and that is pushed to a system such as a bug or ticket queue, an email alias, or a pager. Respectively, these alerts are classified as *tickets*, *email alerts*,[22] and *pages*.

Root cause

> A defect in a software or human system that, if repaired, instills confidence that this event won’t happen again in the same way. A given incident might have multiple root causes: for example, perhaps it was caused by a combination of insufficient process automation, software that crashed on bogus input, *and* insufficient testing of the script used to generate the configuration. Each of these factors might stand alone as a root cause, and each should be repaired.

Node and machine

> Used interchangeably to indicate a single instance of a running kernel in either a physical server, virtual machine, or container. There might be multiple *services* worth monitoring on a single machine. The services may either be:

- Related to each other: for example, a caching server and a web server
- Unrelated services sharing hardware: for example, a code repository and a master for a configuration system like Puppet or Chef

Push

> Any change to a service's running software or its configuration.

Why Monitor?

There are many reasons to monitor a system, including:

Analyzing long-term trends

> How big is my database and how fast is it growing? How quickly is my daily-active user count growing?

Comparing over time or experiment groups

> Are queries faster with Acme Bucket of Bytes 2.72 versus Ajax DB 3.14? How much better is my memcache hit rate with an extra node? Is my site slower than it was last week?

Alerting

> Something is broken, and somebody needs to fix it right now! Or, something might break soon, so somebody should look soon.

Building dashboards

> Dashboards should answer basic questions about your service, and normally include some form of the four golden signals (discussed in The Four Golden Signals).

Conducting *ad hoc* retrospective analysis (i.e., debugging)

> Our latency just shot up; what else happened around the same time?

System monitoring is also helpful in supplying raw input into business analytics and in facilitating analysis of security breaches. Because this book focuses on the engineering domains in which SRE has particular expertise, we won't discuss these applications of monitoring here.

Monitoring and alerting enables a system to tell us when it's broken, or perhaps to tell us what's about to break. When the system isn't able to automatically fix itself, we want a human to investigate the alert, determine if there's a real problem at hand, mitigate the problem, and determine the root cause of the problem. Unless you're performing security auditing on very narrowly scoped components of a system, you should never trigger an alert simply because "something seems a bit weird."

Paging a human is a quite expensive use of an employee's time. If an employee is at work, a page interrupts their workflow. If the employee is at home, a page interrupts their personal time, and perhaps even their sleep. When pages occur too frequently, employees second-guess, skim, or even ignore incoming alerts, sometimes even ignoring a "real" page that's masked by the noise. Outages can be prolonged because other noise interferes with a rapid diagnosis and fix. Effective alerting systems have good signal and very low noise.

Setting Reasonable Expectations for Monitoring

Monitoring a complex application is a significant engineering endeavor in and of itself. Even with substantial existing infrastructure for instrumentation, collection, display, and alerting in place, a Google SRE team with 10–12 members typically has one or sometimes two members whose primary assignment is to build and maintain monitoring systems for their service. This number has decreased over time as we generalize and centralize common monitoring infrastructure, but every SRE team typically has at least one “monitoring person.” (That being said, while it can be fun to have access to traffic graph dashboards and the like, SRE teams carefully avoid any situation that requires someone to “stare at a screen to watch for problems.”)

In general, Google has trended toward simpler and faster monitoring systems, with better tools for *post hoc* analysis. We avoid "magic" systems that try to learn thresholds or automatically detect causality. Rules that detect unexpected changes in end-user request rates are one counterexample; while these rules are still kept as simple as possible, they give a very quick detection of a very simple, specific, severe anomaly. Other uses of monitoring data such as capacity planning and traffic prediction can tolerate more fragility, and thus, more complexity. Observational experiments conducted over a very long time horizon (months or years) with a low sampling rate (hours or days) can also often tolerate more fragility because occasional missed samples won’t hide a long-running trend.

Google SRE has experienced only limited success with complex dependency hierarchies. We seldom use rules such as, "If I know the database is slow, alert for a slow database; otherwise, alert for the website being generally slow." Dependency-reliant rules usually pertain to very stable parts of our system, such as our system for draining user traffic away from a datacenter. For example, "If a datacenter is drained, then don’t alert me on its latency" is one common datacenter alerting rule. Few teams at Google maintain complex dependency hierarchies because our infrastructure has a steady rate of continuous refactoring.

Some of the ideas described in this chapter are still aspirational: there is always room to move more rapidly from symptom to root cause(s), especially in ever-changing systems. So while this chapter sets out some goals for monitoring systems, and some ways to achieve these goals, it’s important that monitoring systems—especially the critical path from the onset of a production problem, through a page to a human, through basic triage and deep debugging—be kept simple and comprehensible by everyone on the team.

Similarly, to keep noise low and signal high, the elements of your monitoring system that direct to a pager need to be very simple and robust. Rules that generate alerts for humans should be simple to understand and represent a clear failure.

Symptoms Versus Causes

Your monitoring system should address two questions: what’s broken, and why?

The "what’s broken" indicates the symptom; the "why" indicates a (possibly intermediate) cause. Table 6-1 lists some hypothetical symptoms and corresponding causes.

Table 6-1. Example symptoms and causes

Symptom	Cause
I’m serving HTTP 500s or 404s	Database servers are refusing connections
My responses are slow	CPUs are overloaded by a bogosort, or an Ethernet cable is crimped under a rack, visible as partial packet loss
Users in Antarctica aren’t receiving animated cat GIFs	Your Content Distribution Network hates scientists and felines, and thus blacklisted some client IPs

Symptom	Cause
Private content is world-readable	A new software push caused ACLs to be forgotten and allowed all requests

"What" versus "why" is one of the most important distinctions in writing good monitoring with maximum signal and minimum noise.

Black-Box Versus White-Box

We combine heavy use of white-box monitoring with modest but critical uses of black-box monitoring. The simplest way to think about black-box monitoring versus white-box monitoring is that black-box monitoring is symptom-oriented and represents active—not predicted—problems: "The system isn't working correctly, right now." White-box monitoring depends on the ability to inspect the innards of the system, such as logs or HTTP endpoints, with instrumentation. White-box monitoring therefore allows detection of imminent problems, failures masked by retries, and so forth.

Note that in a multilayered system, one person's symptom is another person's cause. For example, suppose that a database's performance is slow. Slow database reads are a symptom for the database SRE who detects them. However, for the frontend SRE observing a slow website, the same slow database reads are a cause. Therefore, white-box monitoring is sometimes symptom-oriented, and sometimes cause-oriented, depending on just how informative your white-box is.

When collecting telemetry for debugging, white-box monitoring is essential. If web servers seem slow on database-heavy requests, you need to know both how fast the web server perceives the database to be, and how fast the database believes itself to be. Otherwise, you can't distinguish an actually slow database server from a network problem between your web server and your database.

For paging, black-box monitoring has the key benefit of forcing discipline to only nag a human when a problem is both already ongoing and contributing to real symptoms. On the other hand, for not-yet-occurring but imminent problems, black-box monitoring is fairly useless.

The Four Golden Signals

The four golden signals of monitoring are latency, traffic, errors, and saturation. If you can only measure four metrics of your user-facing system, focus on these four.

Latency

> The time it takes to service a request. It's important to distinguish between the latency of successful requests and the latency of failed requests. For example, an HTTP 500 error triggered due to loss of connection to a database or other critical backend might be served very quickly; however, as an HTTP 500 error indicates a failed request, factoring 500s into your overall latency might result in misleading calculations. On the other hand, a slow error is even worse than a fast error! Therefore, it's important to track error latency, as opposed to just filtering out errors.

Traffic

> A measure of how much demand is being placed on your system, measured in a high-level system-specific metric. For a web service, this measurement is usually HTTP requests per second, perhaps broken out by the nature of the requests (e.g., static versus dynamic content). For an audio streaming system, this measurement might focus on network I/O rate or concurrent sessions. For a key-value storage system, this measurement might be transactions and retrievals per second.

Errors

The rate of requests that fail, either explicitly (e.g., HTTP 500s), implicitly (for example, an HTTP 200 success response, but coupled with the wrong content), or by policy (for example, "If you committed to one-second response times, any request over one second is an error"). Where protocol response codes are insufficient to express all failure conditions, secondary (internal) protocols may be necessary to track partial failure modes. Monitoring these cases can be drastically different: catching HTTP 500s at your load balancer can do a decent job of catching all completely failed requests, while only end-to-end system tests can detect that you're serving the wrong content.

Saturation

How "full" your service is. A measure of your system fraction, emphasizing the resources that are most constrained (e.g., in a memory-constrained system, show memory; in an I/O-constrained system, show I/O). Note that many systems degrade in performance before they achieve 100% utilization, so having a utilization target is essential.

In complex systems, saturation can be supplemented with higher-level load measurement: can your service properly handle double the traffic, handle only 10% more traffic, or handle even less traffic than it currently receives? For very simple services that have no parameters that alter the complexity of the request (e.g., "Give me a nonce" or "I need a globally unique monotonic integer") that rarely change configuration, a static value from a load test might be adequate. As discussed in the previous paragraph, however, most services need to use indirect signals like CPU utilization or network bandwidth that have a known upper bound. Latency increases are often a leading indicator of saturation. Measuring your 99th percentile response time over some small window (e.g., one minute) can give a very early signal of saturation.

Finally, saturation is also concerned with predictions of impending saturation, such as "It looks like your database will fill its hard drive in 4 hours."

If you measure all four golden signals and page a human when one signal is problematic (or, in the case of saturation, nearly problematic), your service will be at least decently covered by monitoring.

Worrying About Your Tail (or, Instrumentation and Performance)

When building a monitoring system from scratch, it's tempting to design a system based upon the mean of some quantity: the mean latency, the mean CPU usage of your nodes, or the mean fullness of your databases. The danger presented by the latter two cases is obvious: CPUs and databases can easily be utilized in a very imbalanced way. The same holds for latency. If you run a web service with an average latency of 100 ms at 1,000 requests per second, 1% of requests might easily take 5 seconds.[23] If your users depend on several such web services to render their page, the 99th percentile of one backend can easily become the median response of your frontend.

The simplest way to differentiate between a slow average and a very slow "tail" of requests is to collect request counts bucketed by latencies (suitable for rendering a histogram), rather than actual latencies: how many requests did I serve that took between 0 ms and 10 ms, between 10 ms and 30 ms, between 30 ms and 100 ms, between 100 ms and 300 ms, and so on? Distributing the histogram boundaries approximately exponentially (in this case by factors of roughly 3) is often an easy way to visualize the distribution of your requests.

Choosing an Appropriate Resolution for Measurements

Different aspects of a system should be measured with different levels of granularity. For example:

- Observing CPU load over the time span of a minute won't reveal even quite long-lived spikes that drive high tail latencies.

- On the other hand, for a web service targeting no more than 9 hours aggregate downtime per year (99.9% annual uptime), probing for a 200 (success) status more than once or twice a minute is probably unnecessarily frequent.
- Similarly, checking hard drive fullness for a service targeting 99.9% availability more than once every 1–2 minutes is probably unnecessary.

Take care in how you structure the granularity of your measurements. Collecting per-second measurements of CPU load might yield interesting data, but such frequent measurements may be very expensive to collect, store, and analyze. If your monitoring goal calls for high resolution but doesn't require extremely low latency, you can reduce these costs by performing internal sampling on the server, then configuring an external system to collect and aggregate that distribution over time or across servers. You might:

1. Record the current CPU utilization each second.
2. Using buckets of 5% granularity, increment the appropriate CPU utilization bucket each second.
3. Aggregate those values every minute.

This strategy allows you to observe brief CPU hotspots without incurring very high cost due to collection and retention.

As Simple as Possible, No Simpler

Piling all these requirements on top of each other can add up to a very complex monitoring system—your system might end up with the following levels of complexity:

- Alerts on different latency thresholds, at different percentiles, on all kinds of different metrics
- Extra code to detect and expose possible causes
- Associated dashboards for each of these possible causes

The sources of potential complexity are never-ending. Like all software systems, monitoring can become so complex that it's fragile, complicated to change, and a maintenance burden.

Therefore, design your monitoring system with an eye toward simplicity. In choosing what to monitor, keep the following guidelines in mind:

- The rules that catch real incidents most often should be as simple, predictable, and reliable as possible.
- Data collection, aggregation, and alerting configuration that is rarely exercised (e.g., less than once a quarter for some SRE teams) should be up for removal.
- Signals that are collected, but not exposed in any prebaked dashboard nor used by any alert, are candidates for removal.

In Google's experience, basic collection and aggregation of metrics, paired with alerting and dashboards, has worked well as a relatively standalone system. (In fact Google's monitoring system is broken up into several binaries, but typically people learn about all aspects of these binaries.) It can be tempting to combine monitoring with other aspects of inspecting complex systems, such as detailed system profiling, single-process debugging, tracking details about exceptions or crashes, load testing, log collection and analysis, or traffic inspection. While most of these subjects share commonalities with basic monitoring, blending together too many results in overly complex and fragile systems. As in many other aspects of software engineering, maintaining distinct systems with clear, simple, loosely coupled points of integration is a better strategy (for example, using web APIs for pulling summary data in a format that can remain constant over an extended period of time).

Tying These Principles Together

The principles discussed in this chapter can be tied together into a philosophy on monitoring and alerting that's widely endorsed and followed within Google SRE teams. While this monitoring philosophy is a bit aspirational, it's a good starting point for writing or reviewing a new alert, and it can help your organization ask the right questions, regardless of the size of your organization or the complexity of your service or system.

When creating rules for monitoring and alerting, asking the following questions can help you avoid false positives and pager burnout:[24]

- Does this rule detect *an otherwise undetected condition* that is urgent, actionable, and actively or imminently user-visible?[25]
- Will I ever be able to ignore this alert, knowing it's benign? When and why will I be able to ignore this alert, and how can I avoid this scenario?
- Does this alert definitely indicate that users are being negatively affected? Are there detectable cases in which users aren't being negatively impacted, such as drained traffic or test deployments, that should be filtered out?
- Can I take action in response to this alert? Is that action urgent, or could it wait until morning? Could the action be safely automated? Will that action be a long-term fix, or just a short-term workaround?
- Are other people getting paged for this issue, therefore rendering at least one of the pages unnecessary?

These questions reflect a fundamental philosophy on pages and pagers:

- Every time the pager goes off, I should be able to react with a sense of urgency. I can only react with a sense of urgency a few times a day before I become fatigued.
- Every page should be actionable.
- Every page response should require intelligence. If a page merely merits a robotic response, it shouldn't be a page.
- Pages should be about a novel problem or an event that hasn't been seen before.

Such a perspective dissipates certain distinctions: if a page satisfies the preceding four bullets, it's irrelevant whether the page is triggered by white-box or black-box monitoring. This perspective also amplifies certain distinctions: it's better to spend much more effort on catching symptoms than causes; when it comes to causes, only worry about very definite, very imminent causes.

Monitoring for the Long Term

In modern production systems, monitoring systems track an ever-evolving system with changing software architecture, load characteristics, and performance targets. An alert that's currently exceptionally rare and hard to automate might become frequent, perhaps even meriting a hacked-together script to resolve it. At this point, someone should find and eliminate the root causes of the problem; if such resolution isn't possible, the alert response deserves to be fully automated.

It's important that decisions about monitoring be made with long-term goals in mind. Every page that happens today distracts a human from improving the system for tomorrow, so there is often a case for taking a short-term hit to availability or performance in order to improve the long-term outlook for the system. Let's take a look at two case studies that illustrate this trade-off.

Bigtable SRE: A Tale of Over-Alerting

Google's internal infrastructure is typically offered and measured against a service level objective (SLO; see Service Level Objectives). Many years ago, the Bigtable service's SLO was based on a synthetic wellbehaved client's mean performance. Because of problems in Bigtable and lower layers of the storage stack, the mean performance was driven by a "large" tail: the worst 5% of requests were often significantly slower than the rest.

Email alerts were triggered as the SLO approached, and paging alerts were triggered when the SLO was exceeded. Both types of alerts were firing voluminously, consuming unacceptable amounts of engineering time: the team spent significant amounts of time triaging the alerts to find the few that were really actionable, and we often missed the problems that actually affected users, because so few of them did. Many of the pages were non-urgent, due to well-understood problems in the infrastructure, and had either rote responses or received no response.

To remedy the situation, the team used a three-pronged approach: while making great efforts to improve the performance of Bigtable, we also temporarily dialed back our SLO target, using the 75th percentile request latency. We also disabled email alerts, as there were so many that spending time diagnosing them was infeasible.

This strategy gave us enough breathing room to actually fix the longer-term problems in Bigtable and the lower layers of the storage stack, rather than constantly fixing tactical problems. On-call engineers could actually accomplish work when they weren't being kept up by pages at all hours. Ultimately, temporarily backing off on our alerts allowed us to make faster progress toward a better service.

Gmail: Predictable, Scriptable Responses from Humans

In the very early days of Gmail, the service was built on a retrofitted distributed process management system called Workqueue, which was originally created for batch processing of pieces of the search index. Workqueue was "adapted" to long-lived processes and subsequently applied to Gmail, but certain bugs in the relatively opaque codebase in the scheduler proved hard to beat.

At that time, the Gmail monitoring was structured such that alerts fired when individual tasks were "de-scheduled" by Workqueue. This setup was less than ideal because even at that time, Gmail had many, many thousands of tasks, each task representing a fraction of a percent of our users. We cared deeply about providing a good user experience for Gmail users, but such an alerting setup was unmaintainable.

To address this problem, Gmail SRE built a tool that helped "poke" the scheduler in just the right way to minimize impact to users. The team had several discussions about whether or not we should simply automate the entire loop from detecting the problem to nudging the rescheduler, until a better long-term solution was achieved, but some worried this kind of workaround would delay a real fix.

This kind of tension is common within a team, and often reflects an underlying mistrust of the team's self-discipline: while some team members want to implement a "hack" to allow time for a proper fix, others worry that a hack will be forgotten or that the proper fix will be deprioritized indefinitely. This concern is credible, as it's easy to build layers of unmaintainable technical debt by patching over problems instead of making real fixes. Managers and technical leaders play a key role in implementing true, long-term fixes by supporting and prioritizing potentially time-consuming long-term fixes even when the initial "pain" of paging subsides.

Pages with rote, algorithmic responses should be a red flag. Unwillingness on the part of your team to automate such pages implies that the team lacks confidence that they can clean up their technical debt. This is a major problem worth escalating.

The Long Run

A common theme connects the previous examples of Bigtable and Gmail: a tension between short-term and long-term availability. Often, sheer force of effort can help a rickety system achieve high availability, but this path is usually short-lived and fraught with burnout and dependence on a small number of heroic team members. Taking a controlled, short-term decrease in availability is often a painful, but strategic trade for the long-run stability of the system. It’s important not to think of every page as an event in isolation, but to consider whether the overall *level* of paging leads toward a healthy, appropriately available system with a healthy, viable team and long-term outlook. We review statistics about page frequency (usually expressed as incidents per shift, where an incident might be composed of a few related pages) in quarterly reports with management, ensuring that decision makers are kept up to date on the pager load and overall health of their teams.

Conclusion

A healthy monitoring and alerting pipeline is simple and easy to reason about. It focuses primarily on symptoms for paging, reserving cause-oriented heuristics to serve as aids to debugging problems. Monitoring symptoms is easier the further "up" your stack you monitor, though monitoring saturation and performance of subsystems such as databases often must be performed directly on the subsystem itself. Email alerts are of very limited value and tend to easily become overrun with noise; instead, you should favor a dashboard that monitors all ongoing subcritical problems for the sort of information that typically ends up in email alerts. A dashboard might also be paired with a log, in order to analyze historical correlations.

Over the long haul, achieving a successful on-call rotation and product includes choosing to alert on symptoms or imminent real problems, adapting your targets to goals that are actually achievable, and making sure that your monitoring supports rapid diagnosis.

[22]Sometimes known as "alert spam," as they are rarely read or acted on.

[23]If 1% of your requests are 50x the average, it means that the rest of your requests are about twice as fast as the average. But if you’re not measuring your distribution, the idea that most of your requests are near the mean is just hopeful thinking.

[24]See *Applying Cardiac Alarm Management Techniques to Your On-Call* [Hol14] for an example of alert fatigue in another context.

[25]Zero-redundancy ($N + 0$) situations count as imminent, as do "nearly full" parts of your service! For more details about the concept of redundancy, see *https://en.wikipedia.org/wiki/N%2B1_redundancy*.

previous

Chapter 5 - Eliminating Toil

next

Chapter 7 - The Evolution of Automation at Google