TEXT ONLY

Metrics That Matter

CRITICAL BUT OFT-NEGLECTED SERVICE METRICS THAT EVERY SRE AND PRODUCT OWNER SHOULD CARE ABOUT

BENJAMIN TREYNOR SLOSS, SHYLAJA NUKALA, AND VIVEK RAU

Site reliability engineering, or SRE, is a software-engineering specialization that focuses on the reliability and maintainability of large systems. In its experience in the field, Google has found some critical but oft-neglected metrics that are important for running reliable services.

This article, based on Ben Treynor's talk at the Google Cloud Next 2017 conference,[7] addresses those metrics, specifically for product development and SRE teams, managers of such teams, and anyone else who cares about the reliability of web products or infrastructure. To further explain its approach to product reliability, Google has published *Site Reliability Engineering: How Google Runs Production Systems*[1] (hereafter referred to as the SRE book) and *The Site Reliability Workbook: Practical Ways to Implement SRE*[2] (hereafter referred to as the SRE workbook).

WHY METRICS MATTER

One of the most important choices in offering a service is which service metrics to measure, and how to evaluate them. The difference between great, good, and poor metric and metric threshold choices is frequently the difference between a service that will surprise and delight its users with how well it works, one that will be acceptable for most users, and one that will actively drive away users—regardless of what the service actually offers.

For example, it is not uncommon to measure the QPS (queries per second) received at a web or API server, and to assess that this metric indicates good service health if (a) the graph of the metric over time has a smooth sinusoidal diurnal curve with no unexpected spikes or troughs, and (b) the peaks of the curve are rising over time, indicating user growth. Yet this is a poor metric choice—at best it will provide the operator with a lagging indicator of large-scale problems. It misses a host of real, common problems, including partial unreachability, error rates in the 0.1–3 percent range, high latency, and intervals of bad results.

These problems lead to unhappy users and service abandonment—yet throughout it all, the QPS Received graph continues to show its happy sinusoidal curves and to provide a soothing sense that all is well. The best that can be said about the QPS Received metric is that it's relatively simple to implement—and even that is a problem, because it is often implemented early and thus takes the place of more sophisticated and useful metrics that would provide an operator with more accurate and useful data about the service.

What follows are the types of metrics that the Google

SRE team has adopted for Google services. These metrics are not particularly easy to implement, and they may require changes to a service to instrument properly. It has been our consistent experience at Google, however, that every service team that implements these metrics is happy afterward that it made the effort to do so. The metrics investment is small compared with the overall effort to build and launch the service in the first place, and the prompt payback in user satisfaction and usage growth is outsized relative to the effort required. We believe you will find this is true for your service, too.

LESSON 1: MEASURE THE ACTUAL USER EXPERIENCE

The SRE book emphasizes that speed matters to users, as demonstrated by Google's research on shifts in behavior when users are exposed to delayed responses from a web service.[3] When services get too slow, users start to disengage, and when they get even slower, users leave. "Speed matters" is a good axiom for SREs to apply when thinking about what makes a service attractive to users.

A good follow-up question is, "Speed for whom?" Engineers often think about measuring speed on the server side, because it is relatively easy to instrument servers to export the required metrics, and standard monitoring tools are designed to capture such metrics from servers in dashboards and highlight anomalies with alerts. What this standard setup is measuring is the interval between the point in time when a user request enters a data center and the point in time when a response to that request leaves the data center. In other words, the metric being captured is server-side latency. Measuring server-side latency is not sufficient, though it is better than not measuring latency at all. Measuring and reporting on server-side latency can be a useful stopgap while solving the harder problem of measuring client-side latency.

The problem is that users have no interest in this server-side metric. Users care about how fast or slow the application is when responding to their actions, and, unfortunately, this can have very little correlation with server-side latency. Perhaps these users have a cheap phone, on a slow 2G network, in a country far away from your servers; if your product doesn't work for them, all your hard work building great features will be wasted, because users will be unhappy and will use a different product. The problem will be compounded if you are measuring only server-side latency, because you will be completely unaware that the product is slow for users. Even if you get anecdotal reports of slowness and try to follow up on them, you will have no way of determining which subset of users is experiencing slowness, and when.

To measure the actual user experience, you have to measure and record client-side latency. It can be hard work to instrument the client code to capture this latency metric and then to ship client-side metrics back to the data center for analysis. The work may be further complicated by the need to handle broken network connections by storing the data and uploading it later.

Though difficult, client-side metrics are essential and achievable.

For a browser application, you can write additional JavaScript that gathers these statistics for users on different platforms, in different countries, etc., and send these statistics back to the server. For a thick client, the path is more obvious, but it's still important to measure the time from the moment the user interacts with the client until the response is delivered. Either way, instrumenting the user experience takes a relatively small fraction of the effort previously expended to write the entire application, and the payback for this incremental effort is high.

To take an example from Google's own history, when Gmail was launched, most users accessed it through a web browser (not a mobile client), and Google's web client code had no instrumentation to capture client-side latency. So, we relied on server-side latency data, and the response time seemed quite acceptable. When Google finally launched an instrumented JavaScript client, at first we didn't believe the data it was sending back—it seemed impossible that the user experience was that bad. We went through the denial stage for a while, and then anger, and eventually got to bargaining.[4] We made some major changes to how the Gmail server and its client worked to improve our client-side latency, and the reward was a visible inflection point in Gmail's growth once the user experience improved. The long-term trends in our monitoring dashboards showed users responding to the improved product experience. For around three percent of the effort of writing and running Gmail, there was a major increase in its adoption and user happiness.

Many techniques are available to application developers for improving client-side response times, and not all of them require large engineering investments. Google's PageSpeed project was created to share with the world the company's insights into client-side response optimization, accompanied by tools that help engineers apply these insights to their own products and web pages.[5] One of the obvious rules is to reduce server response time as much as possible. PageSpeed analysis tools also recommend various well-known techniques for client-side optimization, including compression of static content, using a preprocessor to "minify" code (HTML, CSS, and JavaScript) by removing unnecessary and redundant text, setting cache-control headers correctly, compressing or inlining images, etc.

To recap, measure the actual user experience by measuring how long a user has to wait for a response after performing an action on your product. Do this, even though it is often not easy. Experience says that it will be well worth the effort.

LESSON 2: MEASURE SPEED AT THE 95TH AND 99TH PERCENTILES

While "Speed matters" is a good axiom when thinking about user (un)happiness, that still leaves an open question about how best to quantify the speed of a service. In other words, even if you understand and accept that the value of the latency metric (time to respond to user requests) should be low enough to keep users happy, do you know precisely what metric that is? Should you measure average latency, median latency, or n^{th}-percentile latency?

In the early days of Google's SRE organization, when we managed relatively few products other than Search and Ads, SLOs (service-level objectives) were set for speed based on median latency. (An SLO is a target value for a given metric, used to communicate the desired level of performance for a service. When the target is achieved, that aspect of the service is considered to be performing adequately. In the context of SLOs, the metric being evaluated is called an SLI, or service-level indicator.)

Over the years, particularly as the use of Search expanded to other continents, we learned that users could be unhappy even when we were meeting and beating our SLO targets. We then conducted research to determine the impact of slight degradations in response time on user behavior, and found that users would conduct significantly fewer searches when encountering incremental delays as small as 200 milliseconds.[3] Based on these and other findings, we have learned to measure "long-tail" latency—that is, latency must be measured at the 95th and 99th percentiles to capture the user experience accurately. After all, it doesn't matter if a product is serving the correct result 99.999 percent of the time if five percent of users are unhappy with how long it takes to get that correct result.

Once upon a time, Google used to measure only raw availability. In fact, most SLOs even today are framed around availability: how many requests return a good result versus how many return an error. Availability was computed the following way:

```
% Availability = 1 - % error responses
```

Suppose you have a user service that normally responds in half a second, which sounds good enough for a user on a smartphone, given typical wireless network delays. Now suppose one request in 30 has an internal problem causing a delay that leads to the mobile client app retrying the request after 10 seconds. Now further suppose that the retry almost always succeeds. The availability metrics (as computed above) will say "100% availability." Users will say "97% available"—because if they are accustomed to receiving a response in 500 milliseconds, after three to five seconds they will hit retry or switch apps. It doesn't matter if the user documentation says, "The application may take up to 10 seconds to respond"; once the user base is trained to get an answer in 500 milliseconds most of the time, that's what they'll expect, and they'll behave like a 10-second response delay is an outage. Meanwhile, the SREs will (incorrectly) be happy, at least for the time being, because their measurements say the service is 100 percent available. This disconnect can be avoided by correcting the availability computation as follows:

```
% Availability = 1 - % (error responses + slow responses)
```

Therefore, when an SLO is defined for long-tail latency, you must choose a target response time that does not render the service effectively unavailable. The 99th-percentile latency should be such that users experiencing that latency do not find it completely unacceptable relative to their expectations. Note that their expectations were probably set by the median latency. You really do need to know what your users consider minimally acceptable. A good practice is to conduct experiments that measure how many users are actually lost as latency is artificially increased. These experiments should be

How to define percentile-based SLOs

There is a technique to phrasing SLO definitions optimally—a linguistic point illustrated here with an amusing puzzle. Consider these two alternative SLO definitions for a given web service, using slightly different language in each definition:

1. The 99th-percentile latency for user requests, averaged over a trailing five-minute time window, will be less than 800 milliseconds.
2. Ninety-nine percent of user requests, averaged over a trailing five-minute time window, will complete in less than 800 milliseconds.

Assume that the SLO will be measured every 10 seconds in either case, and an alert will be fired if N consecutive measurements are out of range. Before reading further, think about which SLO definition is better, and why.

The answer is that from a user-happiness perspective, the two SLOs are practically equivalent; and yet, from a computational perspective, alternative number 2 is distinctly superior.

To appreciate this, consider a hypothetical web service receiving 10,000 user requests per second, on average, under peak load conditions. With SLO definition 1, the measurement algorithm actually has to compute a percentile value every 10 seconds. A naive approach to this computation is as follows:

conducted infrequently, using a tiny fraction of randomly sampled users to minimize the risk to your product's brand and reputation.

A good practical rule of thumb learned from these experiments at Google is that the 99th-percentile latency should be no more than three to five times the median latency. This means that if a hypothetical service with median latency of 400 milliseconds starts exhibiting more than two seconds response time for the slowest one percent of requests, this is undesirable. We tune our production systems such that if this undesired behavior continues for some predefined period, an alert will fire or some automated corrective action will be taken (such as shifting traffic around or provisioning more servers). We find that the 50th-, 95th-,

- Store the response times for 10,000 × 300 = 3 million queries in memory to capture five minutes' worth of data (this will use >11MB of memory to store 3 million 32-bit integers, each representing the response time for one query in milliseconds).
- Sort these 3 million integer values.
- Read the 99th-percentile value (i.e., the 30,000th latency value in the sorted list, counting from the maximum downward).

More efficient algorithms are definitely available, such as using 16-bit short integers for latency values and using two heaps instead of sorting a linear list every 10 seconds, but even these improved approaches involve significant overhead.

In contrast, SLO definition 2 requires storing only two integers in memory: the count of user requests with completion times greater than 800 milliseconds, and the total count of user requests. Determining SLO compliance is then a simple division operation, and you don't have to remember latency values at all.

Be sure to define your long-tail latency SLOs using format 2.

and 99th-percentile latency measures for a service are each individually valuable, and we will ideally set SLOs around each of them.

Our recommendations for latency metrics can be applied equally well to other kinds of SLIs, some of them applicable to systems that are not web services. As discussed in the SRE book, storage systems also care about *durability* (whether data is available when needed), and data-processing pipelines care about throughput and *freshness* (how long it takes for data to progress from ingestion to completion).

For more advice on how to create SLOs for a service, read chapter 2, "Implementing SLOs," in the SRE workbook.

LESSON 3: MEASURE FUTURE LOAD

Demand forecasting, or quantifying the future load on a service, is different from typical SLO measurement because it's not a metric you monitor, nor a cause for generating alerts. Demand forecasting makes a service reliable by providing the information needed to provision the service such that it can handle its future load while continuing to meet its SLOs. The more effort you put into generating good demand forecasts, the less you will need to scramble at the last minute to add more compute resources to the service because it's melting down in the face of an unforeseen increase in traffic.

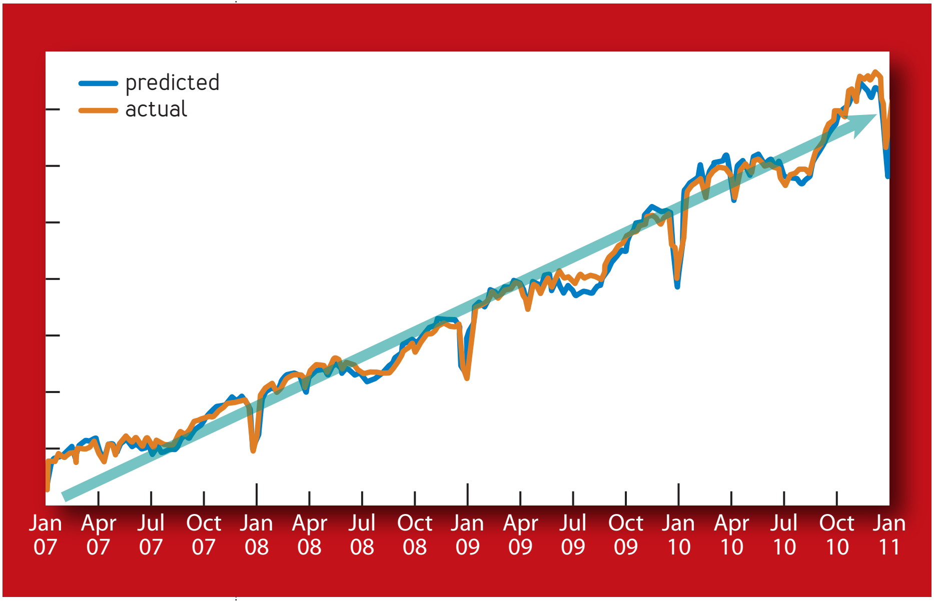
Load on a service is measured using different combinations of metrics depending on the type of service being discussed, but a common denominator unit for many services is QPS. Layered on top of QPS might be other service-dependent metrics such as storage size (gigabytes or terabytes), memory usage, network bandwidth, or I/O bandwidth (gigabits per second).

It's useful to break demand growth down into *organic* and *inorganic*. Organic growth is what you can forecast by extrapolating historical trends in traffic, and the forecasting problem can often be addressed using statistical tools. Inorganic growth is what you forecast for one-time events such as product launches, changes in service performance, or anticipated changes in user behavior, among other factors, and this growth cannot be extrapolated from historical data. Prediction of inorganic growth is less amenable to statistical tools and often relies on rules of thumb and estimates derived from similar events in the past. In the time leading up to a service launch, when there is not enough historical data available to make an organic growth forecast, teams estimate demand using techniques applicable to inorganic growth.

Forecasting organic growth

For mature products that have been in operation for a few years, you can forecast organic growth using statistical methods. Note that linear regression is not a useful tool in most cases, because it doesn't capture seasonal traffic fluctuations; it also doesn't work if growth is not linear. Many web services see significant drops in traffic (the "summer slump") because of the midyear vacation season, and, conversely, see big spikes in traffic during the year-end shopping season, followed by a major "holiday dip" in the last week of the year, followed in turn by a "back-to-work bounce" at the start of the new year (see figure 1). At Google, we even account for predictable changes with a cycle time of several

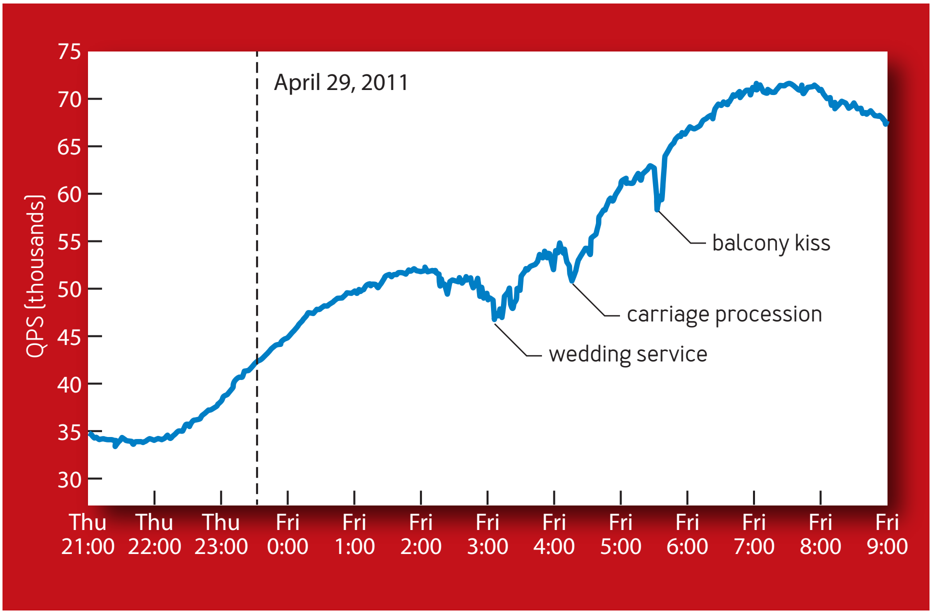
FIGURE 1: **AVERAGE WEEKLY GOOGLE SEARCH TRAFFIC**



The royal wedding as seen by Google search monitoring

Daily traffic fluctuations are far less important for capacity planning than monthly or yearly increases, but they provide an amusing illustration of the impact of external world events on the load presented to a web service. The chart in figure 2 was generated by the system that monitors load on Google's Search product and represents the number of search QPS on April 29, 2011, during the wedding of Prince William and Kate Middleton. The time values on the Y-axis are in the Pacific time zone (eight hours behind UK time), and the traffic pattern neatly captures key events during the ceremony. It is evident from charts like this one that when something really interesting happens in the world, people briefly stop searching the web, and when that event is over, they promptly resume searching.

FIGURE 2: **ROYAL WEDDING SEARCHES**



years, caused by events such as the FIFA World Cup.

Google uses a variety of forecasting models that attempt to capture seasonality on a monthly or annual time scale. There is uncertainty in forecasts, and they imply a confidence level, so rather than forecasting a line, we are forecasting a cone. Any given statistical model has its strengths and weaknesses, so many Google products use outputs generated from a large ensemble of models,[6] which include variants on many well-known approaches, such as the Bass Diffusion Model; Theta Model; logistic models; Bayesian Structural Time Series; STL (seasonal and trend decomposition using Loess); Holt-Winters and other exponential smoothing models; seasonal and other ARIMA (autoregressive integrated moving average)-based models; year-over-year growth models; custom models; and more.

Having generated independent estimates from each model in the ensemble, we then compute their mean after applying a configurable "trimming" parameter to eliminate outlier estimates, and this adjusted mean is used as the final prediction. Depending on the scale and global reach of a service and its different levels of adoption in different parts of the world, it might be more accurate to generate continent-level or country-level forecasts and aggregate them instead of attempting to forecast at the global level.

It is important to compare forecasts regularly with actual traffic in order to tune the model parameters over time and improve the accuracy of the models. Experience shows that the trimmed mean of the ensemble of models delivers superior accuracy compared with any individual model.

Forecasting inorganic growth

Inorganic growth is generated by one-time events that

Google Analytics lesson learned

An interesting case study of inorganic growth that was not generated by any engineering change and was not small involves the initial launch of Google Analytics, a service for gathering and analyzing traffic to any website. Google had acquired Urchin Software Corporation for its web-analytics product that provided traffic collection and analytics dashboards to paying customers. The inorganic traffic growth event occurred when the product was made available for free under the Google brand, permitting any website owner to sign up for it at no charge. Google correctly anticipated a flood of new users, based on prior experience launching the Keyhole (later called Google Earth) subscription-based product for free. Therefore, we carefully load tested and provisioned the product for the expected increase in traffic.

Our prediction for core product usage then performed reasonably well, but we had forgotten to account for traffic to the signup page! The page where new users signed up was backed by a single-threaded SQL database with limited transaction capacity, placing a strict and previously unknown limit on the number of signups per second, resulting in a stream of public complaints from users about site slowness and unavailability. We learned this lesson well, and our product launch checklist afterwards contained the question, "Do new users have to sign up for your service, and if so, have you estimated and tested the load on your signup page?"

have no periodicity, such as launches of new products, new features, or marketing promotions, or changes in user behavior that are triggered by some extraneous factor for which the timing is predictable but the resulting peak traffic volume has a high degree of uncertainty (like the FIFA World Cup or the Royal Wedding), among others. Inorganic growth involves an abrupt change in traffic, and is intrinsically unpredictable because it is triggered by an event that hasn't happened before. When the product owners and SREs have advance notice of such growth, such as when planning for a new feature launch, they need to apply intuition and rules of thumb to estimating post-launch traffic, and understand that their predictions will have a higher level of uncertainty.

General rules for forecasting inorganic growth for product/feature launches include the following:

- Examine historical traffic changes from past launches of similar or analogous features.
- For country- or market-specific launches, consider past user behavior in that market.
- Consider the level of publicity and promotion around the launch.
- Add a margin of uncertainty to the forecast where possible, by provisioning three to five times the resources implied by the forecast.
- While traffic from brand-new products is harder to predict, it is also usually small, so you can overprovision for this traffic without incurring too much cost.

LESSON 4: MEASURE SERVICE EFFICIENCY

SRE teams should regularly measure the efficiency of each service they run, using load tests and benchmarking programs to determine how many user requests per second can be handled with acceptable responses times, given a certain quantity of computing resource (CPU, memory, disk I/O, network bandwidth, etc.). While performance testing may seem an obvious best practice, in real life teams frequently forget about service efficiency. They may benchmark a service once a year, or just before a major release, and then assume unconsciously that the service's performance remains constant between benchmarks. In reality, even minor-seeming changes to the code, or to user behavior, can affect the amount of resources required to serve a given volume of traffic.

A common way of finding out that a service has become less efficient is through a product outage. The SRE team may think they have enough capacity to serve peak traffic even with two data centers' worth of resources turned down for maintenance or emergency repairs, but when the rare event occurs where both data centers are actually down during peak traffic hours, the performance of the service radically degrades and causes a partial outage or becomes so slow as to make the service unusable. In the worst case, this can turn into a "cascading failure" where all serving clusters collapse like a row of dominoes, inducing a global product outage.

Ironically, this type of massive failure is triggered by the system's attempt to recover from smaller failures. One cluster of servers happens to get a higher load for reasons of geography and/or user behavior, and this load is large enough to cause all the servers to crash. The traffic load-balancing system observes these servers going offline and performs a failover operation, diverting all the traffic formerly going to the crashed cluster and sending it to nearby clusters instead. As a result, each of these nearby servers now gets even more overloaded and crashes as well, resulting in more traffic being sent to even fewer live servers. The cycle repeats until every single server is dead and the service is globally unavailable.

Services can avoid cascading failures using the *drop overload* technique. Here the server code is designed to detect when it is overloaded and randomly drop some incoming requests under those circumstances, rather than attempting to handle all requests and eventually melting down. This results in a degraded customer experience for users whose requests are dropped, but that can be mitigated to a large extent by having the client retry the request; in any case, slower responses or outright error responses to a fraction of users are a lot better than a global service failure.

It would be better, of course, to avoid this situation altogether, and the only way to do that is to regularly measure service efficiency to confirm the SRE team's assumptions about how much serving capacity is available. For a service that ships out releases daily or more frequently, daily benchmarking is not an extreme practice—benchmarking can be built into the automated release testing procedure. When newly introduced performance regressions are detected early, the team can provision more resources in the short term and then get the performance bugs fixed in the long term to bring resource costs back in line.

If you run your service on a cloud platform, some cloud providers have an *autoscaling* service that will automatically provision more resources when your service load increases. This setup may be better than running products on premises or in a data center with fixed hardware resources, but it still does not get you off the hook for regular benchmarking. Even though the risk of a complete outage is lower, you may find out too late that your monthly cloud bill has increased dramatically just because someone modified the encoding scheme used for compressing data, or made some other seemingly innocuous code change. For these reasons, it is a best practice to measure service efficiency regularly.

Related articles

➡ A Purpose-built Global Network: Google's Move to SDN
A discussion with Amin Vahdat, David Clark, and Jennifer Rexford
https://queue.acm.org/detail.cfm?id=2856460

➡ From Here to There, the SOA Way
Terry Coatta
SOA is no more a silver bullet than the approaches which preceded it.
https://queue.acm.org/detail.cfm?id=1388788

➡ Voyage in the Agile Memeplex
Philippe Kruchten
In the world of agile development, context is key.
https://queue.acm.org/detail.cfm?id=1281893

For additional details, see chapter 11, "Managing Load," in the SRE workbook. This chapter contains two case studies of managing overload.

CONCLUSION

The metrics discussed in this article should be useful to those who run a service and care about reliability. If you measure these metrics, set the right targets, and go through the work to measure the metrics accurately, not as an approximation, you should find that (1) your service runs better; (2) you experience fewer outages; and (3) you see a lot more user adoption. Most of us like those three properties.

References

1. Beyer, B., Jones, C., Petoff, J., Murphy, N. R. 2016. *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly Media.
2. Beyer, B., Murphy, N. R., Rensin, D. K., Kawahara, K., Thorne, S. 2018. *The Site Reliability Workbook: Practical Ways to Implement SRE.* O'Reilly Media.
3. Brutlag, J. 2009. Speed matters. Google AI Blog; https://research.googleblog.com/2009/06/speed-matters.html.
4. Kübler-Ross "five stages of grief" model; https://

en.wikipedia.org/wiki/K%C3%BCbler-Ross_model.

5. Analyze and optimize your website with PageSpeed tools. 2018. Google Developers; https://developers.google.com/speed/
6. Tassone, E, Rohani, F. 2017. Our quest for robust time series forecasting at scale. The Unofficial Google Data Science Blog; http://www.unofficialgoogledatascience.com/2017/04/our-quest-for-robust-time-series.html.
7. Treynor, B. 2017. Metrics that matter (Google Cloud Next); https://youtu.be/iF9NoqYBb4U.

Benjamin Treynor Sloss *started programming at age 6 and joined Oracle as a software engineer at age 17. He was a software engineer at Oracle and Versant, then worked in engineering management at E.piphany, SEVEN, and finally Google (2003-present). His current team of approximately 4,700 at Google is responsible for site reliability engineering, networking, and data centers worldwide.*

Shylaja Nukala *is a technical writing lead for Google Site Reliability Engineering. She leads the documentation, information management, and select training efforts for SRE, Cloud, and Google engineers. She has a Ph.D. in communication studies from Rutgers University.*

Vivek Rau *is a site reliability engineer at Google, working on CRE (customer reliability engineering). The CRE team teaches customers core SRE principles, enabling them to build and operate highly reliable products on the Google Cloud Platform. Rau has a B.S. degree in computer science from IIT-Madras.*