Monitoring the SRE Golden Signals

Actually Getting the Metrics from Common Services

Steve Mushero - www.OpsStack.io

1st Edition

Table of Contents

Introduction

Site Reliability Engineering (SRE) and related concepts are very popular lately, in part due to the famous Google SRE book and others talking about the "Golden Signals" that you should be monitoring to keep your systems fast and reliable as they scale.

Everyone seems to agree these signals are important, but how do you actually monitor them?

No one seems to talk much about this.

These signals are much harder to get than traditional CPU or RAM monitoring, as each service and resource has different metrics, definitions, and especially tools required.

Microservices, Containers, and Serverless make getting signals even more challenging, but still worthwhile, as we need these signals — both to avoid traditional alert noise and to effectively troubleshoot our increasingly complex distributed systems.

This series of articles will walk through the signals and practical methods for a number of common services. First, we'll talk briefly about the signals themselves, then a bit about how you can use them in your monitoring system.

Finally, there is a list of service-specific guides on how to monitor the signals, for Load Balancers, Web & App Servers, DB & Cache Servers, General Linux, and more. This list and the details may evolve over time as we continually seek feedback and better methods to get better data.

Chapter 1

The Golden Signals

What are they?

There are three common lists or methodologies:

- From the Google SRE book: Latency, Traffic, Errors, and Saturation
- USE Method (from Brendan Gregg): Utilization, Saturation, and Errors
- RED Method (from Tom Wilkie): Rate, Errors, and Duration

You can see the overlap, and as Baron Schwartz notes in his Monitoring & Observability with USE and RED blog, each method varies in focus. He suggests USE is about resources with an internal view, while RED is about requests, real work, and thus an external view (from the service consumer's point of view). They are obviously related, and also complementary, as every service consumes resources to do work.

For our purposes, we'll focus on a simple superset of five signals:

- **Rate** — Request rate, in requests/sec
- **Errors** — Error rate, in errors/sec
- **Latency** — Response time, including queue/wait time, in milliseconds.
- **Saturation** — How overloaded something is, which is related to utilization, but more directly measured by things like queue depth (or sometimes concurrency). As a queue depth measurement, this becomes non-zero when you the system gets saturated. Usually a counter.
- **Utilization** — How busy the resource or system is. Usually expressed 0–100% and most useful for predictions (as Saturation is probably more useful). Note we are not using the Utilization Law to get this (~Rate x Service Time / Workers), but instead looking for more familiar direct measurements.

Of these Saturation & Utilization are often the hardest to get, and full of assumptions, caveats and calculation complexities — treat them as approximations, at best. However, they are often most valuable for hunting down current and future problems, so we put up with learn to love them.

All these measurements can be split and/or aggregated by various things. For example, HTTP could be spilt out to 4xx & 5xx errors, just as Latency or Rate could be broken out by URL.

In addition, there are more sophisticated ways to calculate things. For example, errors often have lower latency than successful requests so you could exclude errors from Latency, if you can (often you cannot).

As useful as these splits or aggregates are, they are outside the scope of this a article as they get much closer to metrics, events, high-cardinality analysis, etc. Let's focus on getting the basic data first, as that's hard enough.

Now we have our Signals, what do we do with them?

You can skip ahead to the actual data collection guides at the end of this post if you'd like, but we should talk about what to do with these signals once we have them.

One of the key reasons these are "Golden" Signals is they try to measure things that directly effect the end-user and work-producing parts of the system — they are direct measurements of things that matter.

This means, in theory, they are better and more useful than lots of less-direct measurements such as CPU, RAM, networks, replication lag, and endless other things (we should know, as we often monitor 200+ items per server; never a good time).

We collect the Golden Signals for a few reasons:

- **Alerting** — Tell us when something is wrong
- **Troubleshooting** — Help us find & fix the problem
- **Tuning & Capacity Planning** — Help us make things better over time

Our focus here is on Alerting, and how to alert on these Signals. What you do after that is between you and your signals.

Alerting has traditionally used static thresholds, in our beloved (ha!) Nagios, Zabbix, DataDog, etc. systems. That works, but is hard to set well and generates lots of alert noise, as you (and anyone you are living with) are mostly likely acutely aware.

But start with static if you must, based on your experience and best practices. These often work best when set to levels where we're pretty sure something is wrong ,or at least unusual is going on (e.g. 95% CPU, latency over 10 seconds, modest size queues, error rates above a few per second, etc.)

If you use static alerting, don't forget the lower bound alerts, such as near zero requests per second or latency, as these often mean something is wrong, even at 3 a.m. when traffic is light.

Are You Average or Percentile?

These alerts typically use average values, but do yourself a favor and try to use median values if you can, as these are less sensitive to big/small outlier values.

Averages have other problems, too, as Optimizely points out in their blog. Still, averages and medians are easily understood, accessible, and quite useful as a signal, as long as your measuring window is short (e.g. 1–5 minutes).

Even better is to start thinking about percentiles. For example, you can alert on your 95th percentile Latency, which is a much better measure of how bad things are for your users. If the 95th percentile is good, then virtually everyone is good.

However, percentiles are more complex than they appear, and of course Vivid Cortex has a blog on this: Why Percentiles Don't Work the Way you Think, where, for example, he warns that your system is really doing a percentile of an average over your measurement time (e.g. 1 or 5 minutes). But it's still useful for alerting and you should try it if you can (and you'll often be shocked how bad your percentiles are).

Are you an Anomaly, or just weird?

Ideally, you can also start using modern Anomaly Detection on your shiny new Golden Signals.

Anomaly Detection is especially useful to catch problems that occur off-peak or that cause unusually lower metric values. Plus they allow much tighter alerting bands so you find issues much earlier (but not so early that you drown in false alerts).

However, Anomaly Detection can be pretty challenging, as few on-premises monitoring solutions can even do it (Zabbix cannot). It's also fairly new, still-evolving, and hard to tune well (especially with the 'seasonality' and trending so common in our Golden Signals). You should be asking your vendors about this.

Fortunately, newer SaaS / Cloud monitoring solutions such as DataDog, SignalFX, etc. can do this, as can new on-premises systems like Prometheus & InfluxDB.

Regardless of your anomaly tooling, Baron Schwartz has a good book on this that you should read to better understand the various options, algorithms, and challenges: Anomaly Detection for Monitoring.

Can I see you?

In addition to alerting, you should also visualize these signals. Weave Works has a nice format, with two graph columns, and Splunk has a nice view. On the left, a stacked graph of Request & Error Rates, and on the right, latency graphs. You could add in a 3rd mixed Saturation / Utilization graph, too.

You can also enrich your metrics with Tags/Events, such as deployments, auto-scale events, restarts, etc. And ideally show all these metrics on a System Architecture Map like Netsil does.

Fix me, fix you

As a final note on alerting, we've found SRE Golden Signal alerts more challenging to respond to, because they are actually symptoms of an underlying problem that is rarely directly exposed by the alert.

This often means engineers must have more system knowledge and be more skilled at digging into the problem, which can easily lie in any of a dozen services or resources

Engineers have always had to connect all the dots and dig below (or above) the alerts, even for basic high CPU or low RAM issues. But the Golden Signals are usually even more abstract, and it's easy to have a lot of them, i.e. a single high-latency problem on a low-level service can easily cause many latency and error alerts all over the system.

One problem Golden Signals help solve is that too often we only have useful data on a few services and a front-end problem creates a long hunt for the culprit. Collecting signals on each service helps nail down which service is the most likely cause (especially if you have dependency info), and thus where to focus.

That's it. Have fun with your signals, as they are both challenging and interesting to find, monitor, and alert on.

Below you will find the details on how to get the data from common services.

Getting the Data from each Service

Below are the appendix sections for various popular services, and we are working to add more over time. Again, we welcome feedback on these.

Note there are lots of nuances & challenges to getting this data in a usable way, so apologies in advance for all the notes and caveats sprinkled throughout as we balance being clear with being somewhat thorough.

Note also that you have to do your own processing in some cases, such as doing delta calculations when you sample counter-based metrics (most systems will do this automatically for you).

The service list, each of which has a chapter on the following pages (and links here to the on-line articles if you want to share):

- Load Balancers — AWS ALB/ELB, HAProxy
- Web Servers — Apache & Nginx
- App Servers — PHP, FPM, Java, Ruby, Node, Go, Python
- Database Servers — MySQL & AWS RDS
- Linux Servers — As underlying Resources

Since this is a long and complex set of articles, there are undoubtedly different views and experiences, thus we welcome feedback and other ideas. We will revise the text based on feedback and other's experience, so check back from time-to-time for updates on your favorite services.

Please see the per-service chapters below.

Chapter 2

Load Balancers' Golden Signals

What are they?

Load Balancers are key components of most modern systems, usually in front of an application, but increasingly inside systems, too, supporting Containers, socket services, databases, and more.

There are several popular LBs in use, so we'll cover the three most common:

- **HAProxy** — Everyone's Favorite non-cloud LB
- **AWS ELB** — Elastic Load Balance
- **AWS ALB** — Application Load Balancer

For Nginx, see the Web Server Section.

Some General Load Balancer Issues

First, some general things to think about for Load Balancers, which are much more complex than web servers to monitor.

Combined or Separate Frontends

There are two ways to look at Load Balancer signals — from the frontend or the backend. And for the frontend, we may have several of them, for different domains or parts of the site, for APIs, authentication, etc.

We are usually interested in the OVERALL signals for the LB, and thus all frontends, though we might also want to track each frontend separately if there are actually separate systems for Web, App, API, Search, etc. This means we'll have separate signals for all of these.

Note some systems have separate Listener/Backends for HTTP and HTTPS, but they serve mostly the same URLs, so it usually makes sense to combine them into a single unified view if you can.

Monitoring Backend Servers in the LB vs. Web Servers

LBs usually have several backend servers, and we can get signals for EACH backend server, too. In a perfect world, we wouldn't need this, as we could get better data directly from the backend web/app servers themselves.

However, as we'll see in the Web and App Server sections, this is not really true, for web server monitoring sucks. This means it's usually easier and better to get per-server backend server signals from the LB, rather than from the web servers themselves.

We've included notes on how to to this in the relevant sections below.

HAProxy

HAProxy is the most popular non-cloud LB, as a powerful, flexible, and very high-performance tool heavily used by everyone. HAProxy also has powerful logging methods and a nice UI, though there are some tricks to getting useful data from it—in some cases the data is so rich, we have to pick and choose what to get.

Caution—Single vs Multi-Process HAProxy

Most, if not all, versions of HAProxy report statistics for a single-process, which is okay for 99% of use cases. Some very-high performance systems use multi-process mode, but this is hard to monitor, as the stats are pulled randomly from one process. This can be a mess, so avoid if possible.

Caution—Complexity

All of the useful HAProxy stats are PER Listener, Backend, or Server, which is useful, but complicates getting a full picture. Simple websites or apps usually have a single (www.) listener and backend, but more complex systems usually have quite a few. It's easy to gather hundreds of metrics and get confused.

You can decide if you want to track the Signal per Listener/Frontend or sum them up to get a total view—this depends on how unified your system is. As noted, above, you usually want to combine HTTP & HTTPS if they serve the same URLs. However, if you have separate Web, App, API, Search, etc. then you probably want to separate the signals, too.

HAProxy, how shall I monitor thee?

There are three ways to monitor HAProxy, all of which use the same format.

- Pulling CSV data from the built-in web page
- Using the CLI tool
- Using the Unix Socket

See the HAProxy documentation for details on how to access each of these, as this will greatly depend on your monitoring system and tools.

Mapping our signals to HAProxy, we see a bit of complexity:

- **Request Rate** — Requests per second, which we can get two ways:
    1. Request Count REQ_TOT is best, since as a counter it won't miss spikes, but you must do delta processing to get the rate. Not available per server, so instead use RATE for servers (though this is only over the last second).
    2. You can use REQ_RATE (req/sec), but this is only over the last second, so you can lose data spikes this way, especially if your monitoring only gets this every minute or less often.

- **Error Rate** — Response Errors, ERESP, which is for errors coming from the backend. This is a counter, so you must delta it. But be careful as the docs say this includes "write error on the client socket", thus it's not clear what kind of client error (such as on mobile phones) this might include. You can also get this per backend server.

  For more detail and just HTTP errors, you can just get 4xx and 5xx error counts, as these will be most sensitive to what users see. 4xx errors are usually not customer issues but if they suddenly rise, it's typically due to bad code or an attack of some type. Monitoring 5xx errors is critical for any system.

  You may also want to watch Request Errors, EREQ, though realize this includes Client Closes which can create a lot of noise, especially on slow or mobile networks. Front-end only.

- **Latency** — Use the Response Time RTIME (per backend) which does an average over the last 1024 requests (so it will miss spikes in busy systems, and be noisy at startup). There is no counter data for these items. This is also available per server.

- **Saturation** — Use the number of queued requests, QCUR. This is available for both the Backend (for requests not yet assigned to a server) and for each Server (not yet sent to the server).

  You probably want the sum of these for overall Saturation, and per server if you are tracking server saturation at this level (see web server section). If you use this per server, you can track each backend server's saturation (though realize the server itself is probably queuing also, so any queue at the LB indicates a serious problem).

- **Utilization** — HAProxy generally does not run out of capacity unless it truly runs out of CPU, but you can monitor actual Sessions SCUR / SLIM.

AWS ELB & ALB

The AWS ELB/ALB Load Balancers are extremely popular for any AWS-based systems. They started out with simple ELBs and have evolved into full-fledged and powerful balancers, especially with the introduction of the new ALBs.

Like most AWS services, metrics are extracted via a combination of Cloud Watch and logs pushed to S3. The former is pretty easy to deal with, but dealing with S3-located logs is always a bit challenging so we try to avoid those (in part as we can't really do real-time processing nor alerting on them).

Note the below are for HTTP, but the ELB & ALB have additional metrics for TCP-based connections that you can use in similar ways.

Details are available in the ELB CloudWatch Documentation.

Classic ELB

ELB metrics are available for the ELB as a whole, but not by backend group or server, unfortunately. Note if you only have one backend-server per AZ, then you could use the AZ Dimensional Filter.

Mapping our signals to the ELB, we get all of these from CloudWatch. Note the sum() part of the metrics which are the CloudWatch statistical functions. The metrics are:

- **Request Rate** — Requests per second, which we get from the sum(RequestCount) metric divided by the configured CloudWatch sampling time, either 1 or 5 minutes. This will include errors.
- **Error Rate** — You should add two metrics: sum(HTTPCode_Backend_5XX) and sum(HTTPCode_ELB_5XX), which captures server-generated errors and LB-generated (important to count backend unavailability and rejections due to full queue). You may also want to add sum(BackendConnectionErrors).
- **Latency** — The average(latency). Easy.
- **Saturation** — The max(SurgeQueueLength) which gets Requests in the backend queues. Note this is focused solely on backend saturation, not on the LB itself which can get saturated on CPU (before it auto-scales), but there appears to be no way to monitor this.

  You can also monitor and alert on sum(SpilloverCount) which will be > 0 when the LB is saturated and rejecting requests because the Surge Queue is full. As with 5xx errors, this is a very serious situation.
- **Utilization** — There is no good way to get utilization data on ELBs, as they auto-scale so their internal capacity is hard to get (though would be nice to get before they scale, such as when things surge).

Caution for ELB Percentiling Persons

If you will do percentiles and statistics on these signals, be sure to read the cautions and issues in the "Statistics for Classic Load Balancer Metrics" section of the CloudWatch docs.

New ALB

The ALB data is very similar to the ELB, with more available data and a few differences in metric names.

ALB metrics are available for the ALB as a whole, and by Target Group (via Dimension Filtering), which is how you can get the data for a given set of backend servers instead of monitoring the Web/App servers directly. Per-server data is not available from the ALB (though you can filter by AZ, which would be per-server if you have only one target backend server per AZ).

Mapping our signals to the ELB, we get all of these from CloudWatch. Note the sum() part of the metrics which are the CloudWatch statistical functions.

- **Request Rate** — Requests per second, which we get from the sum(RequestCount) metric divided by the configured CloudWatch sampling time, either 1 or 5 minutes. This will include errors.
- **Error Rate** — You should add two metrics: sum(HTTPCode_Backend_5XX) and sum(HTTPCode_ELB_5XX), which captures server-generated errors and LB-generated (important to count backend unavailability and rejections due to full queue). You may also want to add sum(TargetConnectionErrorCount).
- **Latency** — The average(TargetResponseTime). Easy.
- **Saturation** — There appears to be no way to get any queue data from the ALB, so we are left with sum(RejectedConnectionCount) which counts rejects when the ALB reached its max connection count.
- **Utilization** — There is no good way to get utilization data on ELBs, as they auto-scale so their internal capacity is hard to get (though would be nice to get before they scale, such as when things surge). Note you can monitor sum(ActiveConnectionCount) vs. the maximum connection count, which you must get manually or from AWS Config.

Caution for ALB Percentiling Persons

If you will do percentiles and statistics on these signals, be sure to read the cautions and issues in the "Statistics for Classic Load Balancer Metrics" section of the CloudWatch docs.

Chapter 3

Web Servers' Golden Signals

What are they?

Where would we be without web servers ? Probably with no web …

These days, the lowly web server has two basic functions: serving simple static content like HTML, JS, CSS, images, etc., and passing dynamic requests to an app backend such as PHP, Java, PERL, C, COBOL, etc.

Thus the web server frontends most of our critical services, including dynamic web content, APIs, and all manner of specialized services, even databases, as everything seems to speak HTTP these days.

So it's critical to get good signals from the web servers.

Unfortunately, they don't measure nor report this data very well, and not at all in aggregate (at least for free). Thus we are left with three choices:

1. Use the very limited built-in status reports/pages
2. Collect & aggregate the web server's HTTP logs for this important data
3. Utilize the upstream Load Balancer per-server metrics, if we can

The last choice, to use the LB's per-backend server metrics, is usually the best way, and there are details in the above Load Balancer section on how to do it.

However, not all systems have the right LB type, and not all Monitoring Systems support getting this type of backend data — for example, this is quite hard in Zabbix, as it struggles to get data about one host (the web server) from another host (the LB). And the AWS ELB/ALB have limited data.

So if we must go to the Web Server Status Pages & HTTP logs, the following are a few painful, but worthwhile ways.

Preparing to Gather Web Server Metrics

There are a few things we need prepare to get the Logs & Status info:

Enable Status Monitoring

You need to enable status monitoring. For both Apache and Nginx, the best practice is to serve it on a different port (e.g. :81) or lock it down to your local network, monitoring systems, etc. so bad guys can't access it.

- **Apache** — Enable mod_status including ExtendedStatus. See DataDog's nice Apache guide.
- **Nginx** — Enable the stub_status_module. See DataDog's nice Nginx guide.

Enable Logging

We need the logs, which means you need to configure them, put them in a nice place, and ideally separate them by vhost.

We need to get response time in to the logs, which unfortunately is not part of any standard nor default format. Edit your web configs to get this:

- **Apache** — We need to add "%D" field into the log definition (usually at the end), which will get the response time in microseconds (use %T on Apache V1.x, but it only gets seconds).

This is the basic equivalent to the Nginx $request_time, below. Note there is an unsupported mod-log-firstbyte module that gets closer to the Nginx $upstream_time, which is really measuring the backend response time. See Apache Log docs.

- **Nginx** — We need to add the "$upstream_response_time" field for backend response time, usually at the end of the log line. Using this "backend time" avoids elongated times for systems that send large responses to slower clients.
- **Nginx** also supports a "$request_time" field in the log definition. This time is until the last byte is sent to the client, so it can capture large responses and/or slow clients.

This can be a more accurate picture of the user experience (not always), but may be too noisy if most of your issues are inside the system vs. the client. See Nginx Log docs.

Note that many people also add the X-Forwarded-For header to the log. If that or other fields are present, you may need to adjust your parsing tools to get the right field.

Log Processing for Metrics

As you'll see below, the most important web server signals, especially Latency, can only be obtained from the logs, which are hard to read, parse, and summarize. But do so we must.

There are quite a number of tools to read HTTP logs, though most of these focus on generating website data, user traffic, URL analyses, etc. We have a different focus, to get the Golden Signals — so first, we need a set of tools that can reliably read, parse, and summarize the logs in a way we can use in our monitoring systems.

You may very well have your favorite tools in this area, such as the increasingly-present ELK stack which can do this with a bit of work (as can Splunk, Sumologic, Logs, etc.) In addition, most SaaS monitoring tools such as DataDog can extract these metrics via their agents or supported 3rd-party tools.

For more traditional monitoring systems such as Zabbix, this is more challenging as they are not good native log readers nor processors. Plus, we are not very interested in the log lines themselves. Instead, we need aggregations of latency and error rate across time in the logs, which is counter to a lot of other 'logging' goals.

So if your monitoring system natively supports web server log metrics, you are all set, and see below. If not, there may be 3rd party or open source tools for this for your system, and you are all set, see below.

If you don't have an existing monitoring system nor 3rd party / open source tool for this, you are somewhat out of luck, as we can't find an out-of-the-box solution, especially to get this data in 1 or 5 minute blocks most useful to us.

The reality is log parsing and aggregation is harder than it appears, and there are very few tools that can do this on server to feed agent-based monitoring. It seems the GoAccess tool can do some of this, with CSV output you can parse. Otherwise, there are lots of good awk and PERL scripts around, but few that support a rolling time-window or even log rollover.

You need to find a system, tool, or service to extract metrics from web logs.

Mapping our signals to Web Servers, we have:

- **Request Rate** — Requests per second, which you can get the hard wayby reading the access logs and count lines to get the total Requests, and do the delta to get Requests per Second. Or the easy way by using the server's status info, and thus:

Apache — Use Total Accesses, which is a counter of total number of requests in the process lifetime. Do delta processing to get requests/sec. Do NOT use “Requests per Second” which is over the life of the server and thus useless.

Nginx — Use Requests, which is a counter of total number of requests. Do delta processing to get requests/sec.

- **Error Rate** — This has to come from the logs, where your tools should count the 5xx errors per second to get a rate. You can also count 4xx errors, but they can be noisy, and usually don’t cause user-facing errors. You really need to be sensitive to 5xx errors which directly affect the user and should be zero all the time.

- **Latency** — This has to come from the logs, where your tools should aggregate the Request or Response time you added to the logs, above. Generally you want to get the average (or better yet, the median) of the response times over your sampling period, such as 1 or 5 minutes.

As mentioned above, you need the right tools for this, as there is useful script nor tool to do this in a generic way. You usually have to send the logs to an ELK-like service (ELK, Sumo, Logs, etc.), monitoring system (DataDog, etc.), or APM system like New Relic, App Dynamics, or DynaTrace.

- **Saturation** — This is a challenging area that differs by web server:

Nginx — It’s nearly impossible to saturate the Nginx server itself, as long as your workers & max connections is set high enough (default is 1x1K, you should usually set much higher, we use 4x2048K).

Apache — Most people run in pre-fork mode, so there is one Apache process per connection, with a practical limit of 500 or so. Thus with a slow backend, it’s very easy to saturate Apache itself. Thus:

Monitor BusyWorkers vs. the smallest of your configured MaxRequestWorkers/ MaxClients/ServerLimit. When Busy = Max, this server is saturated and cannot accept new connections (new connections will be queued).

You can also count HTTP 503 errors from the logs which usually happen when the backend App Server is overloaded, though ideally, you can get that data directly from the App Server.

For many Apache systems, it's critical to measure Memory as a resource because it's by far the easiest way to kill an Apache-based system, to run out of RAM, especially if you are running modPHP or another mod-based app server.

- **Utilization** — For Nginx this is rarely relevant but for Apache it's the same as for Saturation (ratio of BusyWorkers vs. smallest of configured MaxRequestWorkers / Max Clients / Server Limit).

It'd be great to see someone write an Apache module or Nginx Lua/C module to report these signals in the same way that Load Balancers do. This doesn't seem that hard and the community would love them for it.

Overall, useful web server monitoring of these signals is not easy, but you should find a way to do it, ideally upstream via your LBs, or using the methods outlined above.

Chapter 3

App Servers' Golden Signals

What are they?

The Application Servers are usually where all the main work of an application is done, so getting good monitoring signals from them is darned important, though the methods diverge widely based on the language and server.

In a perfect world, you would also embed good observability metrics in your code on the App Servers, especially for Errors and Latency, so you can skip the hardest parts of that in the following processes. In fact, its seems this is the only way in Go, Python, and Node.js.

Let's tour our App Servers:

PHP

PHP may not be as fashionable as it once was, but it still powers the majority of Internet sites, especially in E-Commerce (Magento), Wordpress, Drupal, and a lot of other large systems based on Symfony, Laravel, etc. Thus ignore it we cannot.

PHP runs either as a mod_php in Apache or more commonly as PHP-FPM, which is a standalone process behind Apache or Nginx. Both are popular, though mod_php scales poorly.

For Apache-based mod_php, things are simple, and bad. There are no good external signals, so all you have is the Apache Status Page and Logs as covered in the Web Server section. mod_php's lack of metrics is yet another reason to use PHP-FPM.

For PHP-FPM we need to enable the status page, much as we did for Apache & Nginx. You can get this in JSON, XML, & HTML formats. See an Nginx guide.

You should also enable the PHP-FPM access, error, and slow logs, which can be useful for troubleshooting, and allow the Status Page to show a Slow Log event counter. The error log is set in the main PHP-FPM config file.

The access log is rarely used, but does exist and we need it, so turn it on AND set the format to include %d, the time taken to serve requests (this is done in the POOL configuration, usually www.conf, NOT in the main php-fpm.conf). The slow log is also set in this file. For more info, see this nice how-to page.

Finally, you should properly configure your php logs. By default, they usually go into the web server's error logs but then they are mixed in with 404 and other web errors, making them hard to analyze or aggregate. It's better to add an error_log override setting (php_admin_value[error_log]) in your PHP-FPM pool file (usually www.conf).

For PHP-FPM, our Signals are:

- **Rate** — Sadly, there is no direct way to get this other than read the access logs and aggregate them into requests per second.
- **Errors** — This depends a lot on where your PHP is logging, if at all. The PHP-FPM error logs are not that useful as they are mostly about slow request tracing, and are very noisy. So you should enable and monitor the actual php error logs, aggregating them into errors per second.
- **Latency** — From the PHP-FPM access log we can get the response time in milliseconds, just like for Apache/Nginx.

  As with Apache/Nginx Latency, it's usually easier to get from the Load Balancers if your monitoring system can do this (though this will include non-FPM requests, such as static JS/CSS files).

  If you can't get the logs, you can also monitor "Slow Requests" to get a count of slow responses, which you can delta into Slow Requests/sec.
- **Saturation** — Monitor the "Listen Queue" as this will be non-zero when there are no more FPM processes available and the FPM system is saturated. Note this can be due to CPU, RAM/Swapping, slow DB, slow external service, etc.

  You can also monitor "Max Children Reached" which will tell you how many times you've hit saturation on FPM, though this can't easily be converted to a rate (as you could be saturated all day). But any change in this counter during sampling indicates saturation during the sampling period.
- **Utilization** — We'd like to monitor the in-use FPM processes ("Active Processes") vs. the configured maximum, though the latter is hard to get (parse the config), and you may have to hard-code it.

Java

Java powers a lot of the heavier-duty websites, larger e-commerce, and complex-logic sites, often for enterprises. It runs in a variety of diverse servers and environments, though mostly in Tomcat, so we'll focus on that here. Other Java containers should have data similar to Tomcat, either via JMX or their Logs.

Like all app servers, the Golden Signals may be better monitored upstream a bit in the web server, especially as there is often a dedicated Nginx server in front of each Tomcat instance on the same host, though this is becoming less common. Load Balancers directly in front of Tomcat (or its fronting Nginx) can also provide Rate & Latency metrics.

If we'll be monitoring Tomcat directly we first need to set it up for monitoring, which means making sure you have good access/error logging, and turning on JMX support.

JMX provides or data, so we have to enable this at the JVM level and restart Tomcat. When you turn on JMX, be sure it's read-only and includes security limiting access to the local machine, with a read-only user. See a good guide, the Docs, and a good monitoring PDF.

In recent versions, you can also get JMX data via the Tomcat Manager and HTTP, though be sure to activate proper security on the Manager & interface.

Then our Tomcat Signals are:

- **Rate** — Via JMX, you can get GlobalRequestProcessor's requestCount and do delta processing to get Requests/Second. This counts all requests, including HTTP, but hopefully HTTP is the bulk of these requests and it seems you can specify a Processor name as a filter, which should be your configured HTTP Processor name.

- **Errors** — Via JMX, you can get GlobalRequestProcessor's errorCount and do delta processing to get Errors/Second. Includes non-HTTP stuff unles you can filter it by processor.

- **Latency** — Via JMX, you can get GlobalRequestProcessor's processingTime but this is total time since restart, which when divided by requestCount will give you the long-term average response time, but this is not very useful.

  Ideally your monitoring system or scripts can store both these values each time you sample, then get the differences and divide them — it's not ideal, but it's all we have.

- **Saturation** — If ThreadPool's currentThreadsBusy = maxThreads then you will queue and thus be saturated, so monitor these two.

- **Utilization** — Use JMX to get (currentThreadsBusy / maxThreads) to see what percentage of your threads are in use.

Python

Python is increasingly popular for web and API applications, and runs under a variety of App Servers, starting with the popular Django framework.

Now we also see Flask, Gunicorn, and others becoming popular, but unfortunately all of these are very hard to monitor — it seems most people monitor by embedding metrics in the code and/or using a special package / APM tool for this.

For example a module like Django / Gunicorn's Statsd can be used to emit useful metrics that you can collect if you use statsd — several services such as DataDog can do this for you. But you still have to code the metrics yourself.

As with Node.js, there may be libraries or frameworks that can provide these metrics on a API, logging, or other basis. For example, Django has a Prometheus module.

If you don't use one of these, you should ways to embed the Golden Signals in your code, such as:

- **Rate** — This is hard to get in code, as most things run on a per-request basis. The easiest way is probably set a global request counter and emit that directly, though this means shared memory and global locks.
- **Errors** — Similar to getting the Rate, probably using a global counter.
- **Latency** — The easiest to code, as you can capture the start and end time of the request and emit a duration metric.
- **Saturation** — This is hard to get in code, unless there are data available from the dispatcher on available workers and queues.
- **Utilization** — Same as Saturation.

If you can't do any of these, it seems best to get our signals from the the upstream Web Server or Load Balancer.

Node.js

Like Python's App Servers, everything in Node.js seems to be custom or code-embedded, so there is no standard way to monitor it to get our Signals. Many monitoring services like DataDog have custom integrations or support, and there are lots of add-on modules to export metrics in various ways.

There are some open source / commercially open tools such as KeyMetrics and AppMetrics that provide an API to get a lot of this data.

To make code changes or add your own Golden Signal metrics, see the summary under Python, above.

Otherwise, the best we can do without changing the code is get data from the upstream Web Server or Load Balancer.

Golang

Go runs its own HTTP server and is generally instrumented by adding emitted metrics for things you want to see, such as Latency and Request Rate. Some people put Nginx (which you can monitor) in front of Go, while others use Caddy, which can provide Latency via the {latency_ms} field, similar to Apache/Nginx, but has no great status or rate data (though there is a Prometheus plug-in that has a few).

To make code changes or add your own Golden Signal metrics, see the summary under Python, above.

Otherwise, the best we can do without changing the code is get data from the upstream Web Server or Load Balancer.

Ruby

Ruby remains quite popular for a wide variety of web applications, often as a replacement for PHP in more complex systems over the last few years. It's similar to PHP in many ways, as it runs under an App Server such as Passenger that sits behind Nginx in reverse-proxy mode.

As with most of the app servers, it's often better to get data from the web server or Load Balancers.

For Ruby running under Passenger, you can query the passenger-status page to get key metrics.

For Passenger or Apache mod_rails, our Signals are:

- **Rate** — Get the "Processed" count per Application Group and do delta processing on this to get requests per second.
- **Errors** — There is no obvious way to get this, unless it is in the logs, but Passenger has no separate error log (only a very mixed log, often shared with the web server, though you can set the log level to errors only).
- **Latency** — You have to get this from the Apache/Nginx access logs. See the Web Server section.
- **Saturation** — The "Requests in Queue" per Application Group will tell you the server is saturated. Do not use "Requests in top-level queue" as this should always be zero, according to the docs.
- **Utilization** — There is no obvious way to get this. We can get the total number of processes, but it's not obvious how to tell how many are busy. As with mod_PHP or PHP-FPM, it's important to monitor memory use as it's easy to run out.

COBOL & FORTAN

Just kidding.

Chapter 4

MySQL's Golden Signals

What are they?

MySQL is a key element of many, if not most, modern online systems, so its performance is of the utmost interest, including and especially if it's used as-a-service such as AWS RDS.

The gold standard on MySQL monitoring is Vivid Cortex and their SaaS platform's ability to get and analyze all these signals and much more. But if you don't use Vivid Cortex and/or want to include these signals in your own core / integrated monitoring system, here is how.

All these methods require you to have a good MySQL connection, as there is no good way to get the signals without it (though you can get some stuff from the logs, and Vivid Cortex uses a protocol analyzer). Please make sure your monitoring user has very limited permissions.

Getting MySQL's Golden Signals varies in complexity based on the version you are running, and if you are running MySQL yourself or using RDS. I apologize in advance for all the pieces & parts below.

A tricky thing, this MySQL

Note that MySQL Latency is fundamentally hard to get accurately, and there are tricky issues with some of the data, including how the Performance Schema works. The methods below outline a couple ways; there are several others, all of which have pros and cons.

None are great, but they give general latency you can use to know how things are doing. Be aware the methods below will not capture prepared statements nor stored procedures (both of which are probably rare for most people, depending on your programming language, app server, and framework).

What about RDS?

Everything below should apply to AWS MySQL RDS as long as the Performance Schema is on — you must turn this on via your AWS RDS Instance Parameter Group and then sadly restart your DB.

Mapping our signals to MySQL, we see:

- **Request Rate** — Queries per second, most easily measured by the sum of MySQL's status variables com_select, com_insert, com_update, com_delete and then do delta processing on the final sum to get queries per second.

  Different monitoring tools get these differently. Usually via SQL like:

  ```
  SHOW GLOBAL STATUS LIKE "com_select";
  SHOW GLOBAL STATUS LIKE "com_insert";
  etc.
  ```

  Or you can sum all four at once to get a single number:

  ```
  SELECT sum(variable_value)
  FROM information_schema.global_status
  WHERE variable_name IN ("com_select", "com_update", "com_delete",
  "com_insert") ;
  ```

  You can also monitor the status variable Questions, but this includes lots of extra non-work things that add noise to your data. Less useful.

- **Error Rate** — We can get a global error rate which includes SQL, syntax, and most all other errors returned by MySQL. We get this from the Performance Schema, using the User Event table instead of Global Event table because we truncate the latter later for Latency measurements, and the User table is small enough to be fast and to not auto-purge due to size. This is a counter so you need to apply delta processing. The query is:

  ```
  SELECT sum(sum_errors) AS query_count
  FROM events_statements_summary_by_user_by_event_name
  WHERE event_name IN ('statement/sql/select', 'statement/sql/insert', 'statement/sql/
  update', 'statement/sql/delete');
  ```

- **Latency** — We can get the latency from the PERFORMANCE SCHEMA. As noted above, there are complications to getting decent data:

We need to use the events_statements_summary_global_by_event_name table, as data from the other potentially useful tables (e.g. events_statements_history_long) is deleted automatically when a thread/connection ends (which is very often on non-Java systems like PHP). And the oft-mentioned digest table is mathematically difficult to use due to overflows and complex clock/timer calculations.

But the events_statements_summary_global_by_event_name table is a summary table, so it includes data from the server start. Thus we must TRUNCATE this table whenever we read it — this is an issue if you have other tools using it, but it seems this cannot be avoided.

The query is below, and gets latency for SELECT only as getting data for all queries unfortunately involves much more complex math or queries — there are two statements here, one to get the data and one to then truncate the table:

```
SELECT (avg_timer_wait)/1e9 AS avg_latency_ms FROM
 performance_schema.events_statements_summary_global_by_event_name
WHERE event_name = ‘statement/sql/select’;

TRUNCATE TABLE
 performance_schema.events_statements_summary_global_by_event_name ;
```

- **Saturation** — The easiest way to see any saturation is by queue depth, which is very hard to get. So:

The best way is to parse the SHOW ENGINE INNODB STATUS output (or Innodb Status file). We do this in our systems, but it’s messy (see concurrency setting below). If you so this, the two variables you want is “Queries Running”# queries inside InnoDB” and “# queries in queue”. If you can only get one, get the queue, as that tells you InnoDB is saturated.

Note the above output depends on innodb_thread_concurrency being greater than zero which used to be very common, but recently and in later versions people are setting to 0, which disables these two stats in the Status output, in which case you have to use the global Threads Running below.

If innodb_thread_concurrency = 0 or cannot read the status output file, we have to look at the more easily obtainable THREADS_RUNNING global status variable, which tells us how many non-active threads there are, though we can't see if they are queued for CPU, Locks, IO, etc. This is an instantaneous measurement so it can be quite noisy. You may need to average it over several monitoring intervals. Two ways to get it:

```
SHOW GLOBAL STATUS LIKE "THREADS_RUNNING";

SELECT sum(variable_value)
FROM information_schema.global_status
WHERE Variable_name = "THREADS_RUNNING" ;
```

Note Threads Connected itself is not a good measure of saturation as it's often driven by saturation issues on the client, such as PHP processes (though that can alsobe caused by a slow DB).

You can also use Threads Connected vs. Threads Max, but this is really only useful when the system is out of threads, which should never happen as modern servers should set the max higher than any possible client number (e.g. 8192+) and is an emergency as the DB is refusing connections.

On AWS RDS, you can also get DiskQueueDepth from Cloud Watch which may be helpful if you have IO-limited workloads. For non-RDS, see the Linux Section for more IO info.

- **Utilization** — There are many ways MySQL can run out of capacity, but it's easiest to start with underlying CPU and I/O capacity, measured by CPU % and IO utilization %, both of which are a bit tricky to get (and to be taken with a grain of salt). See the Linux Section.

One useful Utilization measurement we use is "row"reads, which after you do delta processing will give you Row Reads Per Second — MySQL can do up to about one million per CPU Core, so if your DB has little IO (data in RAM), this can give you a good sense of utilization, though using Linux CPU is more accurate if you can get and combine it with MySQL data. For row reads, two ways:

```
SHOW GLOBAL STATUS LIKE "INNODB_ROWS_READ";

SELECT variable_value FROM information_schema.global_status
WHERE variable_name = "INNODB_ROWS_READ" ;
```

On AWS RDS you can also get CPUUtilization from Cloud Watch which may be helpful if you have CPU-limited workloads.

As you can see, MySQL is challenging and there are many more items you can read, both to get the above signals, and to get a broader sense of DB health for alerting, tuning, and so on.

Chapter 6

Linux's Golden Signals

What are they?

Most people are interested in the performance and stability of their application and its underlying services, such as those covered in these articles. But many of those services rely on Linux underneath them, as their core resource for CPU, RAM, Network, and IO.

Thus it's probably wise to get the Linux SRE Signals themselves, especially in cases where the upper-level service is very sensitive to underlying resource use (especially IO).

Even if you don't alert on these things, they provide valuable details for observed changes in higher-level metrics, such as when MySQL Latency suddenly rises — did we have changes in the SQL, the Data, or the underlying IO system?

For Linux, we are mostly interested in CPU and Disk, a bit of memory (for saturation), and for completeness, the network. Networks are rarely overloaded these days, so most people can ignore it, at least in terms of SRE Signals and alerting.

Though you could monitor network IN/OUT as a network-level Request Rate Signal, since network flow anomalies are interesting, and correlating this to other request metrics can be useful in troubleshooting).

CPU

For CPUs, the only real signals are Utilization & Saturation, though Latency is interesting while Errors, Latency, and Requests aren't very relevant. Thus we have:

- **Latency** — It's not clear what this means in Linux, i.e. could be the time taken to do something, or the time waiting in the CPU queue to do something (forgetting about swapping or IO delays).

  Unfortunately we have no way to measure the time it takes to do the may varied things Linux does, nor can we measure the time all tasks spend waiting in the CPU queue. We had hoped we could get this from /proc/schedstat but after lots of work we have concluded we cannot.

- **Saturation** — We want the CPU Run Queue length, which becomes > 0 when things have to wait on the CPU. It's a little hard to get accurately.

We'd love to directly use the Load Average, but unfortunately in Linux it also includes both CPU and IO, which really pollutes the data. However, it's easy to get and can be useful for Saturation if your service uses CPU only, with little/no IO, such as an App Server (but watch out for large logs being written to disks). A Load Average > 1–2 x CPU count is usually considered saturated.

If we want the real queue, i.e. exclude any IO, it's harder. The first problem is that all the tools that show this data do not average it over time, so it's an instantaneous measurement and very noisy. This applies to vmstat, sar -q, and /proc/stat's procs_running, etc.

The next challenge is what Linux calls the run queue actually includes what's currently running, so you have to subtract the number of CPUs to get the real queue. Any result > 0 means the CPU is saturated.

The final issue is how to get this data from the kernel. The best way is in the /proc/stat file. Get the "procs_running" count and subtract the number of cpus (grep -c processor /proc/cpuinfo) to get the queue.

That's still an instantaneous count, so it's very noisy. There seems no tool to same and aggregate this over time, so we wrote one in Go, called runqstat — see it on Github (where it's still under development).

Steal — You can also track CPU Steal % as this also means the CPU is saturated, from the perspective of the Hypervisor, though you should also see an immediate rise in the CPU Run Queue, too, unless you have a single threaded load like Nginx or HAProxy or Node.js where this is hard to see.

Note for single-threaded loads, this whole Saturation metric is less useful (as you can't have a queue if there is nothing else to run). In that case, you should look at Utilization, below.

- **Utilization** — We want the CPU %, but this also needs a bit of math, so get and sum CPU %: User + Nice + System + (probably) Steal. Do not add iowait % and obviously don't add idle %.

It’s critical to track CPU Steal % on any system running under virtualization. Do not miss this, and usually you want to alert if this is much above 10–25%.

For Docker and other containers, the best saturation metric is harder to determine and it depends if there are any caps in place. You have to read the /proc file system for each container and from the file cpu.stat for each, get the nr_throttled metric which will increment each time the Container was CPU-throttled. You probably should delta this to get throttles/second.

Disks

For Disks, we map the SRE signals to the following metrics, all from the iostat utility, though your monitoring system may extract these items directly from /proc/diskstats (most agents do this).

You can get this data for all disks, just the busiest disk, or the disk where your service’s data sits, which is often easiest since you should know where it is and focusing your alerting & dashboards on that avoids having to look at the other less-informative disks in the system.

- **Request Rate** — The IOPS to the disk system (after merging), which in iostat is r/s and w/s.
- **Error Rate** — There are no meaningful error rates you can measure with disks, unless you want to count some very high latencies. Most real errors such as ext4 or NFS are nearly fatal and you should alert on them ASAP, but finding them is hard, as usually you have to parse the kernel logs.
- **Latency** — The read and write times, which in iostat is Average Wait Time (await), which includes queue time, as this will rise sharply when the disk is saturated.

  It’s best to monitor read and write response time separately if you can (r_await & w_await) as these can have very different dynamics, and you can be more sensitive to actual changes without the noise of the ‘other’ IO direction, especially on databases.

  You can also measure iostat Service Time as a way to look at the health of the disk subsystem, such as AWS EBS, RAID, etc. This should be consistent under load & IO-sensitive services will have real problems when this rises.
- **Saturation** — Best measured by IO Queue depth per device, which if you get it from iostat is the aqu-sz variable. Note this includes IO requests being serviced, so it’s not a

true queue measurement (the math is a bit complex.

Note the iostat util % is not a very good measure of true saturation, especially on SSD & RAID arrays which can execute multiple IO request simultaneously, as it's actually the % of time at least one request is in-progress. However, if util % is all you can get, it can be used for simple monitoring to see if there are real changes compared to baselines.

- **Utilization** — Most people use the iostat util%, but as noted above, this is not a good measure of real utilization for other than single-spindle magnetic disks.

Memory

For memory, we only care about Saturation and Utilization.

- **Saturation** — Any swapping at all means RAM saturation, though today most cloud VMs don't use any swap file, so this is slowly becoming useless. The next best metric is really just Utilization, below.

  You should track OOM (Out-of-Memory) errors if at all possible, which are issued by the kernel itself when it is totally out of RAM or swap for a process. Unfortunately this can only be seen in the kernel logs, so ideally you can send these to a centralized ELK or SaaS-based log analyses system that can alert on this.

  Any OOM error implies serious RAM saturation, and is usually an emergency, as it usually kills the largest RAM user, such as MySQL.

  Note that if there is swap available and kernel swappiness is not set to 1, you will often see swapping long before RAM is fully utilized. This is because the kernel will not minimize the file cache before it starts swapping, and thus you can saturate RAM long before RAM is fully used by work-producing processes.

- **Utilization** — Calculate RAM use as (Total RAM- free-buffers-cache), which is the classic measure of RAM in use. Dividing by the total RAM gives the % in use.

Network

For networks, we are interested in throughput, which is our ‘requests per second’ in the sense of bandwidth.

- **Rate** — Read and Write bandwidth in bits-per-second. Most monitoring systems can get this easily.
- **Utilization** — Use Read and Write bandwidth divided by the maximum bandwidth of the network. Note maximum bandwidth may be hard to determine in mixed VM/ Container environments when traffic may use both the hosts’s localhost network & the main wired network. In that case, just tracking Read and Write bandwidth may be best.

Chapter 7

Conclusion

More to come …

Thank for your interest in monitoring the Golden Signals. This is a large and interesting area and there are so many services yet to monitor, or even expose the right metrics for all to see.

As you can see, even here in the midst of a cloud and DevOps revolution, many of our most mainstream tools still don't really do or provide all we need to effectively manage and maintain our infrastructure, but we all keep working to make it better, every day.