Practices

- Alerting
- Consoles and dashboards
- Histograms and summaries
- Instrumentation
- Metric and label naming
- When to use the Pushgateway
- Remote write tuning
- Recording rules
- The Zen of Prometheus

Alerting

We recommend that you read My Philosophy on Alerting based on Rob Ewaschuk's observations at Google.

To summarize: keep alerting simple, alert on symptoms, have good consoles to allow pinpointing causes, and avoid having pages where there is nothing to do.

Naming

There are no strict restrictions regarding the naming of alerting rules, as alert names may contain any number of Unicode characters, just like any other label value. However, the community has rallied around using Camel Case for their alert names.

What to alert on

Aim to have as few alerts as possible, by alerting on symptoms that are associated with end-user pain rather than trying to catch every possible way that pain could be caused. Alerts should link to relevant consoles and make it easy to figure out which component is at fault.

Allow for slack in alerting to accommodate small blips.

Online serving systems

Typically alert on high latency and error rates as high up in the stack as possible.

Only page on latency at one point in a stack. If a lower-level component is slower than it should be, but the overall user latency is fine, then there is no need to page.

For error rates, page on user-visible errors. If there are errors further down the stack that will cause such a failure, there is no need to page on them separately. However, if some failures are not user-visible, but are otherwise severe enough to require human involvement (for example, you are losing a lot of money), add pages to be sent on those.

You may need alerts for different types of request if they have different characteristics, or problems in a low-traffic type of request would be drowned out by high-traffic requests.

Offline processing

For offline processing systems, the key metric is how long data takes to get through the system, so page if that gets high enough to cause user impact.

Batch jobs

For batch jobs it makes sense to page if the batch job has not succeeded recently enough, and this will cause user-visible problems.

This should generally be at least enough time for 2 full runs of the batch job. For a job that runs every 4 hours and takes an hour, 10 hours would be a reasonable threshold. If you cannot withstand a single run failing, run the job more frequently, as a single failure should not require human intervention.

Capacity

While not a problem causing immediate user impact, being close to capacity often requires human intervention to avoid an outage in the near future.

Metamonitoring

It is important to have confidence that monitoring is working. Accordingly, have alerts to ensure that Prometheus servers, Alertmanagers, PushGateways, and other monitoring infrastructure are available and running correctly.

As always, if it is possible to alert on symptoms rather than causes, this helps to reduce noise. For example, a blackbox test that alerts are getting from PushGateway to Prometheus to Alertmanager to email is better than individual alerts on each.

Supplementing the whitebox monitoring of Prometheus with external blackbox monitoring can catch problems that are otherwise invisible, and also serves as a fallback in case internal systems completely fail.

Consoles and dashboards

CAUTION: Starting with Prometheus 3.0, console templates and libraries are no longer bundled with Prometheus. If you wish to use console templates, you must provide your own templates and libraries by specifying the `--web.console.templates` and `--web.console.libraries` command-line flags. This documentation page is maintained for historical reference and to demonstrate the capabilities of console templates. Please be aware that any referenced console libraries from the Prometheus 2.x branch are no longer maintained and may contain known security vulnerabilities (CVEs).

It can be tempting to display as much data as possible on a dashboard, especially when a system like Prometheus offers the ability to have such rich instrumentation of your applications. This can lead to consoles that are impenetrable due to having too much information, that even an expert in the system would have difficulty drawing meaning from.

Instead of trying to represent every piece of data you have, for operational consoles think of what are the most likely failure modes and how you would use the consoles to differentiate them. Take advantage of the structure of your services. For example, if you have a big tree of services in an online-serving system, latency in some lower service is a typical problem. Rather than showing every service's information on a single large dashboard, build separate dashboards for each service that include the latency and errors for each service they talk to. You can then start at the top and work your way down to the problematic service.

We have found the following guidelines very effective:

- Have no more than 5 graphs on a console.
- Have no more than 5 plots (lines) on each graph. You can get away with more if it is a stacked/area graph.
- When using the provided console template examples, avoid more than 20-30 entries in the right-hand-side table.

If you find yourself exceeding these, it could make sense to demote the visibility of less important information, possibly splitting out some subsystems to a new console. For example, you could graph aggregated rather than broken-down data, move it to the right-hand-side table, or even remove data completely if it is rarely useful - you can always look at it in the expression browser!

Finally, it is difficult for a set of consoles to serve more than one master. What you want to know when oncall (what is broken?) tends to be very different from what you want when developing features (how many people hit corner case X?). In such cases, two separate sets of consoles can be useful.

Histograms and summaries

Histograms and summaries are more complex metric types. For historical reasons, histograms exist in two variants: classic histograms and native histograms, the latter even come in a number of sub-variants. This document helps to understand the difference between all those metric types, how to use them correctly, and how to pick the right metric type for your use case.

The most important lesson to learn from this document is simple: If you can, use native histograms and prefer them over both classic histograms and summaries.

Where things start to become tricky is if you find yourself in a situation where you cannot simply use native histograms. Most commonly, you might have to work with existing metrics that include classic histograms or summaries, or maybe the instrumentation library you are using does not support native histograms yet. Furthermore, there are a few specific use cases where you might prefer a summary or a classic histogram.

With this document, you should be able to navigate the related obstacles and subtleties.

Overview

Historically, a sample in the Prometheus world was just a timestamped floating point value. This value could be interpreted as a counter or as a gauge, i.e. most of the time Prometheus doesn't maintain a notion of “static typing”, and you just have to know what kind of metric you are dealing with (assisted by the convention that the name of a counter should end on `_total`).

But there are more metric types than counters and gauges. In particular, there is a need to represent distributions of observed values (usually simply called “observations” in Prometheus terminology). There are fundamentally two different approaches:

1. The instrumented program calculates a number of pre-configured quantiles (e.g. the median or the 90th percentile) over pre-configured time windows (e.g. the last ten minutes) and exposes them as additional metrics. Prometheus implements this approach in the form of a metric type called *summary*. Depending on the used algorithm, the pre-calculated quantiles are usually very accurate. But the calculation has a resource cost for the instrumented program. Also, you cannot “recalculate” the quantiles later if you desire another time window or another percentile, and most importantly, you cannot aggregate quantiles (e.g. to calculate the total 90th percentile latency for a service backed by multiple replicated workers).
2. The instrumented program represents the distribution in a more fundamental way that can later be used to calculate arbitrary quantiles over arbitrary time windows. Most

importantly, the distribution is represented in a way that can be aggregated with each other. This kind of representation is sometimes called a *digest*. Prometheus implements this approach in the form of a metric type called *histogram*, where observations are counted in buckets, as you might know it from the general concept of a histogram.

In both approaches, Prometheus also collects the count and the sum of observations (see details below).

Common to both approaches is the need to collect a whole lot of numerical values per sample, not just a single floating point value as before:

- In any case the count and sum of observations.
- In the case of summaries the pre-calculated quantiles.
- In the case of histograms a set of buckets with their population counts and boundaries.

The new types of metrics are also called *composite types*.

In a first approach, Prometheus preserved its data model of simple timestamped floating point values and mapped this multitude of values into one time series each, distinguished by specific labels. In this way, summaries and classic histograms were created. In both, the count and sum of observations are each tracked in a separate time series. Similarly, each pre-calculated quantile of a summary and each bucket of a histogram is tracked in its own time series. PromQL operators and functions act on these individual time series, as explained in detail further below.

On the one hand, this approach has worked quite well. While keeping the data model simple, it satisfies many use cases. On the other hand, it suffers from many limitations, especially when it comes to histograms. Thus, much later in Prometheus's lifetime, native histograms were introduced. A native histogram sample is a “composition of values”, where a single sample contains the count and sum of observations and a dynamic number of buckets with their population count and boundaries. In the Prometheus TSDB, one histogram results in one time series of native histogram samples rather than a bunch of independent time series. PromQL operators and functions now have to act on these composite samples rather than on the individual time series of floats before.

You can read everything about native histograms in their specification, but be warned that this is a very technical and detailed document. If you read on here, you can expect a more digestible and usage-focused explanation.

If you are interested in Prometheus's journey towards native representation of composite types, you can read more in a blog post.

Instrumentation library support

First of all, check the library support for histograms and summaries.

Summaries are usually supported by all libraries, but some might only track the count and sum of observations and omit the quantile calculation. (Quantile-less summaries is still a legitimate use of summaries, see below.)

Classic histogram support is also widespread, but native histogram support is still rare. Currently, the latter requires exposition via the protobuf format, limiting the support to protobuf-enabled libraries, like the Java and the Go library. Support in a text-based format is underway as part of OpenMetrics v2. Things should be moving very soon, so definitely check what your library has to offer.

Even if your instrumented program only exposes classic histogram, you can configure Prometheus to ingest them as native histograms anyway. This will happen in the form of *Native Histograms with Custom Bucket boundaries* (NHCB). These NHCBs have some limitations compared to the usual native histograms (which feature so-called standard exponential buckets), but they are still much more efficient to store than pure classic histograms. NHCBs handling in PromQL is the same as for other native histograms, so a later migration to “real” native histograms will be easy.

Ingestion via Open Telemetry

Maybe you aren't even using a Prometheus instrumentation library, but your metrics come from a collector adhering to the Open Telemetry (OTel) standard. When ingesting OTel metrics into a Prometheus-compatible backends, the “normal” OTel histograms can be converted into classic histograms or NHCBs on the Prometheus side (hint: prefer the latter), while OTel's *exponential histograms* are always converted into the usual native histograms (with standard exponential buckets).

Count and sum of observations

Histograms and summaries both record observations, typically request durations or response sizes. In all variants (even quantile-less summaries), they track the number of observations *and* the sum of the observed values, allowing you to calculate the *average* of the observed values.

To do so, you generally first take a `rate` over the desired duration and then divide the “rate of the sum” by the “rate of the count”.

For a native histogram (including an NHCB), you extract the sum and count of observations with the functions `histogram_sum` and `histogram_count`, respectively. For example, to calculate the average request duration over the last 5m from a native histogram called `http_request_duration_seconds`, use the following PromQL expression:

```
  histogram_sum(rate(http_request_duration_seconds[5m]))
/
  histogram_count(rate(http_request_duration_seconds[5m]))
```

For the calculation of the average, there is also a shorter version using the function `histogram_avg`. The following is equivalent to the above:

```
histogram_avg(rate(http_request_duration_seconds[5m]))
```

In the case of a summary or a classic histogram, you have separate time series for the sum and count of observations, marked by the magic suffixes `_sum` and `_count`, respectively. Thus, a summary or classic histogram called `http_request_duration_seconds` will result in the series `http_request_duration_seconds_sum` and `http_request_duration_seconds_count`, and the expression to calculate the average request duration over the last 5m will look like this:

```
  rate(http_request_duration_seconds_sum[5m])
/
  rate(http_request_duration_seconds_count[5m])
```

The denominator in both expressions above is also useful on its own. It represents the requests per second served over the last 5m. Another way of putting it is that the `http_request_duration_seconds_count` series behaves exactly like a counter for the HTTP requests (which you would call `http_requests_total` if you did not already have the histogram or summary to replace it). The key property of a counter in Prometheus is that it always goes up, unless there is a counter reset.

If your observations are never negative, the `http_request_duration_seconds_sum` series also always goes up (unless there is a counter reset). However, if negative observations are in the mix, the sum of observations may also go down, breaking assumptions made by PromQL. Such a drop would erroneously be considered a counter reset in the `rate(http_request_duration_seconds_sum[5m])` calculation above, throwing off the result. Note that this problem only affects summaries and classic histograms. Native histograms (including NHCBs) are `rate`'d as a whole, thereby detecting counter resets correctly. In the rare cases where you cannot avoid negative observations and are stuck with summaries or classic histograms, you can use two separate summaries or histograms, one for positive and one for negative observations (the latter with inverted sign), and combine the results later with suitable PromQL expressions.

Both the sum and count of observations are additive, so you can easily aggregate – after the rate, but before the division, and no matter what the underlying metric type is. These are the expressions to calculate the average request duration for each `job`:

Native histograms:

```
  sum by (job) (histogram_sum(rate(http_request_duration_seconds[5m])))
/
  sum by (job) (histogram_count(rate(http_request_duration_seconds[5m])))
```

Summaries or classic histograms:

```
  sum by (job) (rate(http_request_duration_seconds_sum[5m]))
/
  sum by (job) (rate(http_request_duration_seconds_count[5m]))
```

Bucketing

Histograms are essentially bucketed counters, so the most obvious use case that separates histograms from summaries is to count observations falling into particular buckets of observation values.

If you instrument code with classic histograms, you will configure fixed bucket boundaries. If you let Prometheus ingest these classic histograms in the classic way, each bucket configured in that way will create a series suffixed with `_bucket`, no matter if the bucket is populated or not. More buckets give you more options and accuracy in the various queries (see below), but the “one series per bucket” cost is quite significant.

If you ingest the classic histograms as NHCBs, unpopulated buckets have a negligible cost, and even populated ones are handled in a more efficient way because each NHCB is represented by a single series of composite samples (rather than by a separate series of floats for each bucket and the sum and count of observations).

However, picking the right buckets in advance can be challenging. And changing buckets later creates a lot of disruption (as you will see below). If you instrument code directly with native histograms, you do not pick any bucket boundaries explicitly, but you configure a desired resolution. Buckets are created dynamically following an exponential bucketing schema, covering the whole range of floating point numbers from -Inf to +Inf. Higher resolution causes higher resource usage, but generally you can reach much higher resolution than with classic histograms for the same resource cost. Instrumentation libraries also offer various strategies to limit the count of populated buckets, like occasional resets of the histogram or adaptive resolution reduction. See the documentation of the instrumentation library you are using for details.

To query the fraction of observations falling into a certain range based on a native histogram, use an expression like the following:

```
histogram_fraction(0, 0.3, sum by (job) (rate(http_request_duration_seconds[5m])))
```

This calculates the fraction of HTTP requests for each `job` that lasted between 0ms and 300ms in the last 5m. (300ms are represented here as `0.3` seconds as you should always use base units in Prometheus.) Note how the `sum` correctly aggregates by summing up the corresponding buckets in the involved histograms. If the histograms have different bucket layouts, they are reconciled first. With the usual exponential bucketing schema, this works smoothly, essentially by falling back to the lowest common resolution among all involved histograms. The same is done to reconcile different bucket layouts over time (in the 5m range that is used in the `rate` calculation). With NHCBs, the effects of this depend heavily on the details of the different bucket layouts. It is well possible that the reconciled aggregated histogram has just one bucket left, containing all observations. Because of the potentially severe effects, the query result gets an info-level annotation if NHCBs needed to be reconciled. This is also one of the reasons why native histograms with the dynamic exponential buckets are much easier to handle.

The calculated fraction is accurate if there happens to be a bucket boundary precisely at 0.3. In the common case that there is not, interpolation is used to return an estimated fraction. This estimation is more accurate with higher bucket resolutions. If you already know in advance that, for example, you have an SLO to serve 95% of requests within 300ms, you could use the fixed bucket boundaries of a classic histogram to allow an accurate calculation. However, if your SLO changes later, changing the fixed bucket layout accordingly will be quite tedious. (You have to change the instrumentation of your code. And you will run into the issues reconciling different bucket layouts as described above.) If you pick native histograms with the dynamic exponential buckets, you won't get a bucket boundary at exactly 0.3, but with a decent resolution, the interpolated estimate will still be quite accurate. In return, you gain the freedom of changing the range boundaries at will, which is not only helpful if your SLO changes, but also to explore questions like “Could we maintain a stricter SLO based on the data of the last quarter?”.

In the pure legacy case of classic histograms that were also ingested as classic histograms, the corresponding PromQL expression looks quite different:

```
  sum by (job) (rate(http_request_duration_seconds_bucket{le="0.3"}[5m]))
/
  sum by (job) (rate(http_request_duration_seconds_count[5m]))
```

The `le` label name stands for “less or equal”. This label's value is the upper inclusive boundary of a cumulative bucket (i.e. this bucket contains all observations less than or equal to 0.3 – including negative observations, which we assume wouldn't happen in the case of observing request durations).

Note that this expression strictly requires a bucket boundary configured at 0.3. If the histograms involved do not have a bucket with that boundary, no interpolation is applied. Instead of an estimation, no result is returned at all. If only some of the involved histograms have such a bucket, an incomplete result is returned, but without any warning, which is a pretty bad situation to be in. (Hint: Avoid this “purely classic” case. If you can, ingest classic histograms as NHCB. Or instrument with native histograms in the first place.)

Apdex score

When reading about fractions of requests served within a certain duration range, you might remember the Apdex score. For this score, you set a target request duration and a tolerated request duration (usually 4 times the target request duration). Let's say your target request duration is 300ms and the tolerable request duration is 1.2s. If you want to calculate the Apdex score by `job` over the last 5m, the PromQL expression for native histograms (including NHCB) is straightforward. Simply add the fraction of requests within your duration target to half of the fraction of requests with a duration between the target and the tolerated duration:

```
  histogram_fraction(0, 0.3, sum by (job) (rate(http_request_duration_seconds[5m])))
+
  histogram_fraction(0.3, 1.2, sum by (job) (rate(http_request_duration_seconds[5m]))) / 2
```

In the “pure classic” case, you *must* have buckets present at the exact boundaries (giving you an accurate calculation in return). The corresponding PromQL expression looks quite different because the classic buckets are cumulative:

```
    (
        sum by (job) (rate(http_request_duration_seconds_bucket{le="0.3"}[5m]))
      +
        sum by (job) (rate(http_request_duration_seconds_bucket{le="1.2"}[5m]))
    )
  /
    2
/
  sum by (job) (rate(http_request_duration_seconds_count[5m]))
```

(For the sake of simplicity, the above expressions do not explicitly exclude failed requests from the satisfied and tolerated parts of the calculation, as it would be required for a strictly correct Apdex calculation.)

Quantiles

You can use both summaries and histograms to calculate so-called φ-quantiles, where $0 \leq \varphi \leq 1$. The φ-quantile is the observation value that ranks at number φ*N among the N observations. Examples for φ-quantiles: The 0.5-quantile is known as the median. The 0.95-quantile is also called the 95th percentile.

The essential difference between summaries and histograms is that summaries calculate streaming φ-quantiles within the instrumented program and expose them directly, while histograms expose bucketed observation counts and the calculation of quantiles from the buckets of a histogram happens on the Prometheus server using the `histogram_quantile()` function. Histograms are further divided into native and classic histograms. The following table lists some implications of the different approaches.

	Native Histogram	Classic Histogram	Summary
Required configuration during instrumentation	Pick a desired resolution and maybe a strategy to limit the bucket count.	Pick buckets suitable for the expected range of observed values and the desired queries.	Pick desired φ-quantiles and sliding window. Other φ-quantiles and sliding windows cannot be calculated later.
Instrumentation cost	Observations are cheap as they only need to increment counters.	Observations are cheap as they only need to increment counters.	Observations are relatively expensive due to the streaming quantile calculation.
Query performance	The server has to calculate quantiles from complex histogram samples. You can use recording rules should the ad-hoc calculation take too long (e.g. in a large dashboard).	The server has to calculate quantiles from a large number of bucket series. You can use recording rules should the ad-hoc calculation take too long (e.g. in a large dashboard).	Fast (no quantile calculations on the server, and aggregations are impossible anyway, see below).
Number of time series per histogram/summary	One (with a composite sample type).	`_sum`, `_count`, and one for each configured bucket.	`_sum`, `_count`, and one for each configured quantile.
Quantile error (see below for details)	Limited by the configured resolution.	Error is limited by the width of the bucket the quantile is located in.	Configurable, generally very low.
Specification of φ-quantile and sliding time-window	Ad-hoc with PromQL expression.	Ad-hoc with PromQL expression.	Preconfigured during instrumentation.
Aggregation	Ad-hoc with PromQL expression, buckets are always compatible.	Ad-hoc with PromQL expression, provided there are no changes in bucket boundaries.	Not aggregatable.

As mentioned above, classic histograms can be ingested by the Prometheus server as a special form of native histograms, called NHCBs (Native Histograms with Custom Bucket boundaries). Therefore, they share some implications with classic histograms and some with the usual native histograms. On the instrumentation side, they behave exactly like classic histograms. (In fact, they are identical to classic histograms, as NHCBs are only created on the server side when a classic histogram is ingested as an NHCB.) The query performance and number of time series is the same as for the usual native histograms, but the quantile error is the same as with a corresponding classic histogram. NHCBs treat a change of the bucket layout a bit more gracefully than classic histograms, but it is still a problematic situation (which is at least flagged as such by an

annotation).

Note the importance of the last item in the table. Let us return to the SLO of serving 95% of requests within 300ms. This time, you do not want to display the percentage of requests served within 300ms, but instead the 95th percentile, i.e. the request duration within which you have served 95% of requests. To do that, you can either configure a summary with a 0.95-quantile and (for example) a 5-minute decay time, or you configure a native histogram with a decent resolution (for example, with the Go instrumentation library, you could use a value of 1.1 for the `NativeHistogramBucketFactor`), or you configure a classic histogram with a few buckets around the 300ms mark, e.g. `{le="0.1"}`, `{le="0.2"}`, `{le="0.3"}`, and `{le="0.45"}`. If your service runs replicated with a number of instances, you will collect request durations from every single one of them, and then you want to aggregate everything into an overall 95th percentile. However, aggregating the precomputed quantiles from a summary rarely makes sense. In this particular case, averaging the quantiles yields statistically nonsensical values.

```
avg(http_request_duration_seconds{quantile="0.95"}) // BAD!
```

Using histograms, the aggregation is perfectly possible with the `histogram_quantile()` function.

Native histogram version (including NHCB):

```
histogram_quantile(0.95, sum(rate(http_request_duration_seconds[5m]))) // GOOD.
```

Classic histogram version:

```
histogram_quantile(0.95, sum by (le) (rate(http_request_duration_seconds_bucket[5m]))) //
GOOD.
```

Furthermore, should your SLO change and you now want to plot the 90th percentile, or you want to take into account the last 10 minutes instead of the last 5 minutes, you only have to adjust the expressions above and you do not need to reconfigure the instrumentation of the monitored programs.

Errors of quantile estimation

Quantiles, whether calculated by the instrumented binary or on the Prometheus server, are estimated. It is important to understand the errors of that estimation.

Continuing the histogram example from above, imagine your usual request durations are almost all very close to 220ms, or in other words, in a histogram with very high resolution, you would see a very sharp spike at 220ms, and the “true” 95th percentile is also close to 220ms.

With the `NativeHistogramBucketFactor` of 1.1 (following the Go instrumentation example), the bucket this spike would fall into has a lower boundary of approximately 0.210 and an upper

boundary of approximately 0.229. (This document deliberately avoids to explain the details how these boundaries are calculated. See the aforementioned spec for details.) To keep things simple, let's assume that indeed *all* request fall into this bucket. The interpolation logic of `histogram_quantile` will then estimate the 95th percentile to be 228ms (again glossing over the details of the calculation here). However, given the bucket boundaries above, the true value could be anywhere between 210ms and 229ms, depending on the actual distribution of requests within the bucket. So this is a fairly accurate estimation, even in the worst case (the true value could be 210ms rather than 220ms vs. the estimated value of 228ms).

Now let's apply the same to the classic histogram configured as described above. All observations, and therefore also the 95th percentile, will fall into the bucket labeled `{le="0.3"}`, i.e. the bucket from 200ms to 300ms. The interpolation would estimate 295ms in this case, with the guarantee that the true value is between 200ms and 300ms. Not only is the error margin much larger, also the estimated value of 295ms is much farther away from the true value of 220ms than in case of the native histogram, where the estimation was 228ms. Given that the SLO is at 300ms for the 95th percentile, the classic histogram gives you the impression that you are very close to breaching it, but in reality you are still doing quite well.

Next step in our thought experiment: A change in backend routing adds a fixed amount of 100ms to all request durations. Now the request duration has its sharp spike at 320ms.

The relevant bucket of the native histogram ranges from 297ms to 324ms (again just stating numbers here without telling you how they are calculated), with the interpolated estimation for the 95th percentile being 323ms. That's an almost perfect guess.

The classic histogram, however, will see almost all observations in the bucket from 300ms to 450ms. The 95th percentile is estimated to be 443ms, far away from the correct value close to 320ms. While you are only a tiny bit outside of your SLO, the estimated 95th quantile looks much worse.

A summary would have had no problem calculating the correct percentile value very accurately in both cases, at least if it uses an appropriate algorithm (like the one used by the Go instrumentation library – this algorithm will yield very accurate results for narrow distributions as in our example). Unfortunately, you cannot use a summary if you need to aggregate the observations from a number of instances.

Luckily, due to your appropriate choice of bucket boundaries for the classic histogram, in this contrived example of very sharp spikes in the distribution of observed values, the classic histogram was able to identify correctly if you were within or outside of your SLO (although it was bad in telling you how far away you were from breaching or keeping the SLO). However, the closer the actual value of the quantile is to the SLO (or in other words, the value you are actually most interested in), the more accurate the calculated value becomes.

Let us now modify the experiment once more. In the new setup, the distributions of request durations has a spike at 150ms, but it is not quite as sharp as before and only comprises 90% of

the observations. 10% of the observations are evenly spread out in a long tail between 150ms and 450ms. With that distribution, the 95th percentile happens to be exactly at our SLO of 300ms. With the classic histogram, the calculated value would be accurate in this (contrived) case, as the value of the 95th percentile happens to coincide with one of the configured bucket boundaries. Even slightly different values would still be accurate as the even distribution within the relevant buckets is exactly what the interpolation algorithm for classic histograms assumes.

The error of the quantile reported by a summary gets more interesting here. In the case of the Go instrumentation library, the error of the quantile in a summary is configured in the dimension of φ. In our case we might have configured 0.95±0.01, i.e. the calculated value will be between the 94th and 96th percentile. The 94th quantile with the distribution described above is 270ms, the 96th quantile is 330ms. The calculated value of the 95th percentile reported by the summary can be anywhere in the interval between 270ms and 330ms, which unfortunately is all the difference between clearly within the SLO vs. clearly outside the SLO.

The bottom line is: If you use a summary, you control the error in the dimension of φ. If you use a histogram, you control the error in the dimension of the observed value, via choosing the appropriate bucket layout in case of the classic histogram (tough) or via choosing a bucket resolution in case of a native histogram (easy). With a broad distribution, small changes in φ result in large deviations in the observed value. With a sharp distribution, a small interval of observed values covers a large interval of φ.

The rules of thumb are the following:

1. If you have access to native histograms, use them with a resolution that matches your accuracy requirements. This combines the required accuracy with the ability to aggregate and to change parameters (percentile, sliding window) ad hoc via the PromQL expression.
2. If you cannot use native histograms, but you need aggregations, you have to use classic histograms, which requires you to set appropriate bucket boundaries, covering the correct range of values and finding the right trade-off between the cost for the buckets and the required accuracy.
3. Only if aggregation isn't needed, you can start thinking about summaries. The main advantage is that it gives you very accurate quantile estimation (in the dimension of φ) at relatively low overall cost. However, the additional requirement to pick the desired quantiles and sliding window at instrumentation time is another severe drawback of summaries.

Visualization

While the pre-calculated quantiles of a summary can be visualized as any other time series of floats, visualizing a histogram is more complex. The Prometheus UI shows a graphical representation of a single histogram sample in the *Table* view. However, in the *Graph* view, it simply plots each component series of a classic histogram or – in case of a native histogram – only the sum of observations.

A very useful visualization of a histogram over time is a heatmap. The Prometheus UI does not support heatmaps yet (see tracking issue). However, popular dashboarding tools like Perses or Grafana are able to render heatmaps based on Prometheus histograms. The resolution of classic histograms is usually not high enough to create compelling heatmaps, but the higher resolution feasible with native histograms enables very detailed heatmaps.

Instrumentation

This page provides an opinionated set of guidelines for instrumenting your code.

How to instrument

The short answer is to instrument everything. Every library, subsystem and service should have at least a few metrics to give you a rough idea of how it is performing.

Instrumentation should be an integral part of your code. Instantiate the metric classes in the same file you use them. This makes going from alert to console to code easy when you are chasing an error.

The three types of services

For monitoring purposes, services can generally be broken down into three types: online-serving, offline-processing, and batch jobs. There is overlap between them, but every service tends to fit well into one of these categories.

Online-serving systems

An online-serving system is one where a human or another system is expecting an immediate response. For example, most database and HTTP requests fall into this category.

The key metrics in such a system are the number of performed queries, errors, and latency. The number of in-progress requests can also be useful.

For counting failed queries, see section Failures below.

Online-serving systems should be monitored on both the client and server side. If the two sides see different behaviors, that is very useful information for debugging. If a service has many clients, it is not practical for the service to track them individually, so they have to rely on their own stats.

Be consistent in whether you count queries when they start or when they end. When they end is suggested, as it will line up with the error and latency stats, and tends to be easier to code.

Offline processing

For offline processing, no one is actively waiting for a response, and batching of work is common. There may also be multiple stages of processing.

For each stage, track the items coming in, how many are in progress, the last time you processed something, and how many items were sent out. If batching, you should also track batches going in and out.

Knowing the last time that a system processed something is useful for detecting if it has stalled, but it is very localised information. A better approach is to send a heartbeat through the system: some dummy item that gets passed all the way through and includes the timestamp when it was inserted. Each stage can export the most recent heartbeat timestamp it has seen, letting you know how long items are taking to propagate through the system. For systems that do not have quiet periods where no processing occurs, an explicit heartbeat may not be needed.

Batch jobs

There is a fuzzy line between offline-processing and batch jobs, as offline processing may be done in batch jobs. Batch jobs are distinguished by the fact that they do not run continuously, which makes scraping them difficult.

The key metric of a batch job is the last time it succeeded. It is also useful to track how long each major stage of the job took, the overall runtime and the last time the job completed (successful or failed). These are all gauges, and should be pushed to a PushGateway. There are generally also some overall job-specific statistics that would be useful to track, such as the total number of records processed.

For batch jobs that take more than a few minutes to run, it is useful to also scrape them using pull-based monitoring. This lets you track the same metrics over time as for other types of jobs, such as resource usage and latency when talking to other systems. This can aid debugging if the job starts to get slow.

For batch jobs that run very often (say, more often than every 15 minutes), you should consider converting them into daemons and handling them as offline-processing jobs.

Subsystems

In addition to the three main types of services, systems have sub-parts that should also be monitored.

Libraries

Libraries should provide instrumentation with no additional configuration required by users.

If it is a library used to access some resource outside of the process (for example, network, disk, or IPC), track the overall query count, errors (if errors are possible) and latency at a minimum.

Depending on how heavy the library is, track internal errors and latency within the library itself, and any general statistics you think may be useful.

A library may be used by multiple independent parts of an application against different resources, so take care to distinguish uses with labels where appropriate. For example, a database connection pool should distinguish the databases it is talking to, whereas there is no need to differentiate between users of a DNS client library.

Logging

As a general rule, for every line of logging code you should also have a counter that is incremented. If you find an interesting log message, you want to be able to see how often it has been happening and for how long.

If there are multiple closely-related log messages in the same function (for example, different branches of an if or switch statement), it can sometimes make sense to increment a single counter for all of them.

It is also generally useful to export the total number of info/error/warning lines that were logged by the application as a whole, and check for significant differences as part of your release process.

Failures

Failures should be handled similarly to logging. Every time there is a failure, a counter should be incremented. Unlike logging, the error may also bubble up to a more general error counter depending on how your code is structured.

When reporting failures, you should generally have some other metric representing the total number of attempts. This makes the failure ratio easy to calculate.

Threadpools

For any sort of threadpool, the key metrics are the number of queued requests, the number of threads in use, the total number of threads, the number of tasks processed, and how long they took. It is also useful to track how long things were waiting in the queue.

Caches

The key metrics for a cache are total queries, hits, overall latency and then the query count, errors and latency of whatever online-serving system the cache is in front of.

Collectors

When implementing a non-trivial custom metrics collector, it is advised to export a gauge for how long the collection took in seconds and another for the number of errors encountered.

This is one of the two cases when it is okay to export a duration as a gauge rather than a summary or a histogram, the other being batch job durations. This is because both represent information about that particular push/scrape, rather than tracking multiple durations over time.

Things to watch out for

There are some general things to be aware of when doing monitoring, and also Prometheus-specific ones in particular.

Use labels

Few monitoring systems have the notion of labels and an expression language to take advantage of them, so it takes a bit of getting used to.

When you have multiple metrics that you want to add/average/sum, they should usually be one metric with labels rather than multiple metrics.

For example, rather than `http_responses_500_total` and `http_responses_403_total`, create a single metric called `http_responses_total` with a `code` label for the HTTP response code. You can then process the entire metric as one in rules and graphs.

As a rule of thumb, no part of a metric name should ever be procedurally generated (use labels instead). The one exception is when proxying metrics from another monitoring/instrumentation system.

See also the naming section.

Do not overuse labels

Each labelset is an additional time series that has RAM, CPU, disk, and network costs. Usually the overhead is negligible, but in scenarios with lots of metrics and hundreds of labelsets across hundreds of servers, this can add up quickly.

As a general guideline, try to keep the cardinality of your metrics below 10, and for metrics that exceed that, aim to limit them to a handful across your whole system. The vast majority of your metrics should have no labels.

If you have a metric that has a cardinality over 100 or the potential to grow that large, investigate alternate solutions such as reducing the number of dimensions or moving the analysis away from monitoring and to a general-purpose processing system.

To give you a better idea of the underlying numbers, let's look at node_exporter. node_exporter exposes metrics for every mounted filesystem. Every node will have in the tens of timeseries for, say, `node_filesystem_avail`. If you have 10,000 nodes, you will end up with roughly 100,000 timeseries for `node_filesystem_avail`, which is fine for Prometheus to handle.

If you were to now add quota per user, you would quickly reach a double digit number of millions with 10,000 users on 10,000 nodes. This is too much for the current implementation of Prometheus. Even with smaller numbers, there's an opportunity cost as you can't have other, potentially more useful metrics on this machine any more.

If you are unsure, start with no labels and add more labels over time as concrete use cases arise.

Counter vs. gauge, summary vs. histogram

It is important to know which of the four main metric types to use for a given metric.

To pick between counter and gauge, there is a simple rule of thumb: if the value can go down, it is a gauge.

Counters can only go up (and reset, such as when a process restarts). They are useful for accumulating the number of events, or the amount of something at each event. For example, the total number of HTTP requests, or the total number of bytes sent in HTTP requests. Raw counters are rarely useful. Use the `rate()` function to get the per-second rate at which they are increasing.

Gauges can be set, go up, and go down. They are useful for snapshots of state, such as in-progress requests, free/total memory, or temperature. You should never take a `rate()` of a gauge.

Summaries and histograms are more complex metric types discussed in their own section.

Timestamps, not time since

If you want to track the amount of time since something happened, export the Unix timestamp at which it happened - not the time since it happened.

With the timestamp exported, you can use the expression `time() - my_timestamp_metric` to calculate the time since the event, removing the need for update logic and protecting you against the update logic getting stuck.

Inner loops

In general, the additional resource cost of instrumentation is far outweighed by the benefits it brings to operations and development.

For code which is performance-critical or called more than 100k times a second inside a given process, you may wish to take some care as to how many metrics you update.

A Java counter takes 12-17ns to increment depending on contention. Other languages will have similar performance. If that amount of time is significant for your inner loop, limit the number of metrics you increment in the inner loop and avoid labels (or cache the result of the label lookup, for example, the return value of `With()` in Go or `labels()` in Java) where possible.

Beware also of metric updates involving time or durations, as getting the time may involve a syscall. As with all matters involving performance-critical code, benchmarks are the best way to determine the impact of any given change.

Avoid missing metrics

Time series that are not present until something happens are difficult to deal with, as the usual simple operations are no longer sufficient to correctly handle them. To avoid this, export a default value such as `0` for any time series you know may exist in advance.

Most Prometheus client libraries (including Go, Java, and Python) will automatically export a `0` for you for metrics with no labels.

Metric and label naming

The metric and label conventions presented in this document are not required for using Prometheus, but can serve as both a style-guide and a collection of best practices. Individual organizations may want to approach some of these practices, e.g. naming conventions, differently.

Metric names

A metric name...

- ...MUST comply with the data model for valid characters.
- ...SHOULD have a (single-word) application prefix relevant to the domain the metric belongs to. The prefix is sometimes referred to as `namespace` by client libraries. For metrics specific to an application, the prefix is usually the application name itself. Sometimes, however, metrics are more generic, like standardized metrics exported by client libraries. Examples:
  - `prometheus_notifications_total` (specific to the Prometheus server)
  - `process_cpu_seconds_total` (exported by many client libraries)
  - `http_request_duration_seconds` (for all HTTP requests)
- ...MUST refer to a single unit (e.g. do not mix seconds with milliseconds) and to a single quantity (e.g. do not mix request size with request duration).
- ...SHOULD use base units (e.g. seconds, bytes, meters - not milliseconds, megabytes, kilometers). See below for a list of base units.
- ...SHOULD have a suffix describing the unit, in plural form. Note that an accumulating count has `total` as a suffix, in addition to the unit if applicable. Also note that this applies to units in the narrow sense (like the units in the table below), but not to countable things in general. For example, `connections` or `notifications` are not considered units for this rule and do not have to be at the end of the metric name. (See also examples in the next paragraph.)
  - `http_request_duration_seconds`
  - `node_memory_usage_bytes`
  - `http_requests_total` (for a unit-less accumulating count)
  - `process_cpu_seconds_total` (for an accumulating count with unit)
  - `foobar_build_info` (for a pseudo-metric that provides metadata about the running binary)
  - `data_pipeline_last_record_processed_timestamp_seconds` (for a timestamp that tracks the time of the latest record processed in a data processing pipeline)
- ...MAY order its name components in a way that leads to convenient grouping when a list of metric names is sorted lexicographically, as long as all the other rules are followed. The

following examples have their the common name components first so that all the related metrics are sorted together:
    - `prometheus_tsdb_head_truncations_closed_total`
    - `prometheus_tsdb_head_truncations_established_total`
    - `prometheus_tsdb_head_truncations_failed_total`
    - `prometheus_tsdb_head_truncations_total`
      The following examples are also valid, but are following a different trade-off. They are easier to read individually, but unrelated metrics like `prometheus_tsdb_head_series` might get sorted in between.
    - `prometheus_tsdb_head_closed_truncations_total`
    - `prometheus_tsdb_head_established_truncations_total`
    - `prometheus_tsdb_head_failed_truncations_total`
    - `prometheus_tsdb_head_truncations_total`
- ...SHOULD represent the same logical thing-being-measured across all label dimensions.
    - request duration
    - bytes of data transfer
    - instantaneous resource usage as a percentage

As a rule of thumb, either the `sum()` or the `avg()` over all dimensions of a given metric should be meaningful (though not necessarily useful). If it is not meaningful, split the data up into multiple metrics. For example, having the capacity of various queues in one metric is good, while mixing the capacity of a queue with the current number of elements in the queue is not.

Why include unit and type suffixes in metric names?

Some metric naming conventions (e.g. OpenTelemetry) do not recommend or even do not allow including information about a metric unit and type in the metric name. A common argument is that those pieces of information are already defined somewhere else (e.g. schema, metadata, other labels, etc.).

Prometheus strongly recommends including unit and type in a metric name, even if you store that information elsewhere, because of the following practical reasons:

- **Metric consumption reliability and UX**: When interacting with a modern UI to use such a metric in PromQL, it's possible to display rich information about the metric's type and unit (autocompletion, overlays, pop-ups). Unfortunately, interactive, adhoc querying in a powerful UI is not the only way that users interact with metrics. Metric consumption ecosystem is vast. Majority of the consumption comes in a form of the plain YAML configuration for variety of observability tools like alerting, recording, autoscaling, dashboards, analysis, processing, etc. It's **critical**, especially during monitoring/SRE incident practices to look on PromQL expressions in plain YAML and understand the underlying metric type and unit you work with.

- **Metric collisions**: With growing adoption and metric changes over time, there are cases where lack of unit and type information in the metric name will cause certain series to collide (e.g. `process_cpu` for seconds and milliseconds).

Labels

Use labels to differentiate the characteristics of the thing that is being measured:

- `api_http_requests_total` - differentiate request types: `operation="create|update|delete"`
- `api_request_duration_seconds` - differentiate request stages: `stage="extract|transform|load"`

Do not put the label names in the metric name, as this introduces redundancy and will cause confusion if the respective labels are aggregated away.

CAUTION: Remember that every unique combination of key-value label pairs represents a new time series, which can dramatically increase the amount of data stored. Do not use labels to store dimensions with high cardinality (many different label values), such as user IDs, email addresses, or other unbounded sets of values.

Base Units

Prometheus does not have any units hard coded. For better compatibility, base units should be used. The following lists some metrics families with their base unit. The list is not exhaustive.

Family	Base unit	Remark
Time	seconds	
Temperature	celsius	*celsius* is preferred over *kelvin* for practical reasons. *kelvin* is acceptable as a base unit in special cases like color temperature or where temperature has to be absolute.
Length	meters	
Bytes	bytes	
Bits	bytes	To avoid confusion combining different metrics, always use *bytes*, even where *bits* appear more common.

Family	Base unit	Remark
Percent	ratio	Values are 0–1 (rather than 0–100). `ratio` is only used as a suffix for names like `disk_usage_ratio`. The usual metric name follows the pattern `A_per_B`.
Voltage	volts	
Electric current	amperes	
Energy	joules	
Power		Prefer exporting a counter of joules, then `rate(joules[5m])` gives you power in Watts.
Mass	grams	*grams* is preferred over *kilograms* to avoid issues with the *kilo* prefix.

When to use the Pushgateway

The Pushgateway is an intermediary service which allows you to push metrics from jobs which cannot be scraped. For details, see Pushing metrics.

Should I be using the Pushgateway?

We only recommend using the Pushgateway in certain limited cases. There are several pitfalls when blindly using the Pushgateway instead of Prometheus's usual pull model for general metrics collection:

- When monitoring multiple instances through a single Pushgateway, the Pushgateway becomes both a single point of failure and a potential bottleneck.
- You lose Prometheus's automatic instance health monitoring via the `up` metric (generated on every scrape).
- The Pushgateway never forgets series pushed to it and will expose them to Prometheus forever unless those series are manually deleted via the Pushgateway's API.

The latter point is especially relevant when multiple instances of a job differentiate their metrics in the Pushgateway via an `instance` label or similar. Metrics for an instance will then remain in the Pushgateway even if the originating instance is renamed or removed. This is because the lifecycle of the Pushgateway as a metrics cache is fundamentally separate from the lifecycle of the processes that push metrics to it. Contrast this to Prometheus's usual pull-style monitoring: when an instance disappears (intentional or not), its metrics will automatically disappear along with it. When using the Pushgateway, this is not the case, and you would now have to delete any stale metrics manually or automate this lifecycle synchronization yourself.

Usually, the only valid use case for the Pushgateway is for capturing the outcome of a service-level batch job. A "service-level" batch job is one which is not semantically related to a specific machine or job instance (for example, a batch job that deletes a number of users for an entire service). Such a job's metrics should not include a machine or instance label to decouple the lifecycle of specific machines or instances from the pushed metrics. This decreases the burden for managing stale metrics in the Pushgateway. See also the best practices for monitoring batch jobs.

Alternative strategies

If an inbound firewall or NAT is preventing you from pulling metrics from targets, consider moving the Prometheus server behind the network barrier as well. We generally recommend running Prometheus servers on the same network as the monitored instances. Otherwise, consider PushProx, which allows Prometheus to traverse a firewall or NAT.

For batch jobs that are related to a machine (such as automatic security update cronjobs or configuration management client runs), expose the resulting metrics using the Node Exporter's textfile collector instead of the Pushgateway.

Remote write tuning

Prometheus implements sane defaults for remote write, but many users have different requirements and would like to optimize their remote settings.

This page describes the tuning parameters available via the remote write configuration.

Remote write characteristics

Each remote write destination starts a queue which reads from the write-ahead log (WAL), writes the samples into an in memory queue owned by a shard, which then sends a request to the configured endpoint. The flow of data looks like:

```
       |-->  queue (shard_1)   --> remote endpoint
WAL --|-->  queue (shard_...) --> remote endpoint
       |-->  queue (shard_n)   --> remote endpoint
```

When one shard backs up and fills its queue, Prometheus will block reading from the WAL into any shards. Failures will be retried without loss of data unless the remote endpoint remains down for more than 2 hours. After 2 hours, the WAL will be compacted and data that has not been sent will be lost.

During operation, Prometheus will continuously calculate the optimal number of shards to use based on the incoming sample rate, number of outstanding samples not sent, and time taken to send each sample.

Resource usage

Using remote write increases the memory footprint of Prometheus. Most users report ~25% increased memory usage, but that number is dependent on the shape of the data. For each series in the WAL, the remote write code caches a mapping of series ID to label values, causing large amounts of series churn to significantly increase memory usage.

In addition to the series cache, each shard and its queue increases memory usage. Shard memory is proportional to the `number of shards * (capacity + max_samples_per_send)`. When tuning, consider reducing `max_shards` alongside increases to `capacity` and `max_samples_per_send` to avoid inadvertently running out of memory. The default values for `capacity: 10000` and `max_samples_per_send: 2000` will constrain shard memory usage to less than 2 MB per shard.

Remote write will also increase CPU and network usage. However, for the same reasons as above, it is difficult to predict by how much. It is generally a good practice to check for CPU and network saturation if your Prometheus server falls behind sending samples via remote write (`prometheus_remote_storage_samples_pending`).

Parameters

All the relevant parameters are found under the `queue_config` section of the remote write configuration.

`capacity`

Capacity controls how many samples are queued in memory per shard before blocking reading from the WAL. Once the WAL is blocked, samples cannot be appended to any shards and all throughput will cease.

Capacity should be high enough to avoid blocking other shards in most cases, but too much capacity can cause excess memory consumption and longer times to clear queues during resharding. It is recommended to set capacity to 3-10 times `max_samples_per_send`.

`max_shards`

Max shards configures the maximum number of shards, or parallelism, Prometheus will use for each remote write queue. Prometheus will try not to use too many shards, but if the queue falls behind the remote write component will increase the number of shards up to max shards to increase throughput. Unless remote writing to a very slow endpoint, it is unlikely that `max_shards` should be increased beyond the default. However, it may be necessary to reduce max shards if there is potential to overwhelm the remote endpoint, or to reduce memory usage when data is backed up.

`min_shards`

Min shards configures the minimum number of shards used by Prometheus, and is the number of shards used when remote write starts. If remote write falls behind, Prometheus will automatically scale up the number of shards so most users do not have to adjust this parameter. However, increasing min shards will allow Prometheus to avoid falling behind at the beginning while calculating the required number of shards.

`max_samples_per_send`

Max samples per send can be adjusted depending on the backend in use. Many systems work very well by sending more samples per batch without a significant increase in latency. Other backends will have issues if trying to send a large number of samples in each request. The default value is small enough to work for most systems.

`batch_send_deadline`

Batch send deadline sets the maximum amount of time between sends for a single shard. Even if the queued shards has not reached `max_samples_per_send`, a request will be sent. Batch send deadline can be increased for low volume systems that are not latency sensitive in order to increase request efficiency.

`min_backoff`

Min backoff controls the minimum amount of time to wait before retrying a failed request. Increasing the backoff spreads out requests when a remote endpoint comes back online. The backoff interval is doubled for each failed requests up to `max_backoff`.

`max_backoff`

Max backoff controls the maximum amount of time to wait before retrying a failed request.

Recording rules

A consistent naming scheme for recording rules makes it easier to interpret the meaning of a rule at a glance. It also avoids mistakes by making incorrect or meaningless calculations stand out.

This page documents proper naming conventions and aggregation for recording rules.

Naming

- Recording rules should be of the general form `level:metric:operations`.
- `level` represents the aggregation level and labels of the rule output.
- `metric` is the metric name and should be unchanged other than stripping `_total` off counters when using `rate()` or `irate()`.
- `operations` is a list of operations that were applied to the metric, newest operation first.

Keeping the metric name unchanged makes it easy to know what a metric is and easy to find in the codebase.

To keep the operations clean, `_sum` is omitted if there are other operations, as `sum()`. Associative operations can be merged (for example `min_min` is the same as `min`).

If there is no obvious operation to use, use `sum`. When taking a ratio by doing division, separate the metrics using `_per_` and call the operation `ratio`.

Aggregation

- When aggregating up ratios, aggregate up the numerator and denominator separately and then divide.
- Do not take the average of a ratio or average of an average, as that is not statistically valid.
- When aggregating up the `_count` and `_sum` of a Summary and dividing to calculate average observation size, treating it as a ratio would be unwieldy. Instead keep the metric name without the `_count` or `_sum` suffix and replace the `rate` in the operation with `mean`. This represents the average observation size over that time period.
- Always specify a `without` clause with the labels you are aggregating away. This is to preserve all the other labels such as `job`, which will avoid conflicts and give you more useful metrics and alerts.

Examples

Note the indentation style with outdented operators on their own line between two vectors. To make this style possible in Yaml, block quotes with an indentation indicator (e.g. `|2`) are used.

Aggregating up requests per second that has a `path` label:

```
- record: instance_path:requests:rate5m
  expr: rate(requests_total{job="myjob"}[5m])

- record: path:requests:rate5m
  expr: sum without (instance)(instance_path:requests:rate5m{job="myjob"})
```

Calculating a request failure ratio and aggregating up to the job-level failure ratio:

```
- record: instance_path:request_failures:rate5m
  expr: rate(request_failures_total{job="myjob"}[5m])

- record: instance_path:request_failures_per_requests:ratio_rate5m
  expr: |2
      instance_path:request_failures:rate5m{job="myjob"}
    /
      instance_path:requests:rate5m{job="myjob"}

# Aggregate up numerator and denominator, then divide to get path-level ratio.
- record: path:request_failures_per_requests:ratio_rate5m
  expr: |2
      sum without (instance)(instance_path:request_failures:rate5m{job="myjob"})
    /
      sum without (instance)(instance_path:requests:rate5m{job="myjob"})

# No labels left from instrumentation or distinguishing instances,
# so we use 'job' as the level.
- record: job:request_failures_per_requests:ratio_rate5m
  expr: |2
      sum without (instance, path)(instance_path:request_failures:rate5m{job="myjob"})
    /
      sum without (instance, path)(instance_path:requests:rate5m{job="myjob"})
```

Calculating average latency over a time period from a Summary:

```
- record: instance_path:request_latency_seconds_count:rate5m
  expr: rate(request_latency_seconds_count{job="myjob"}[5m])

- record: instance_path:request_latency_seconds_sum:rate5m
  expr: rate(request_latency_seconds_sum{job="myjob"}[5m])

- record: instance_path:request_latency_seconds:mean5m
  expr: |2
      instance_path:request_latency_seconds_sum:rate5m{job="myjob"}
    /
      instance_path:request_latency_seconds_count:rate5m{job="myjob"}

# Aggregate up numerator and denominator, then divide.
- record: path:request_latency_seconds:mean5m
  expr: |2
      sum without (instance)(instance_path:request_latency_seconds_sum:rate5m{job="myjob"})
    /
      sum without (instance)(instance_path:request_latency_seconds_count:rate5m{job="myjob"})
```

Calculating the average query rate across instances and paths is done using the `avg()` function:

```
- record: job:request_latency_seconds_count:avg_rate5m
  expr: avg without (instance,
path)(instance_path:request_latency_seconds_count:rate5m{job="myjob"})
```

Notice that when aggregating that the labels in the `without` clause are removed from the level of the output metric name compared to the input metric names. When there is no aggregation, the levels always match. If this is not the case a mistake has likely been made in the rules.

The Zen of Prometheus

The Zen of Prometheus is beginner friendly set of core values and guidelines for instrumenting your applications and writing idiomatic alerts using Prometheus. This is a document that's intended to maintained by the Prometheus community. Feel free to contribute.

Instrument first, ask questions later

During development you will never know what questions you need to ask later. Software needs good instrumentation, it's not optional. A metric without labels is cheap. Use them generously, but be mindful of cardinality when adding labels.

The first and most important rule—if you have to remember only one thing, remember this one. Instrument all the things.

Measure what users care about

Do your users care if your database servers are down? Do they care about your CPU saturation? Yes, but not directly—they care about what they experience. They care about whether they can access the page they requested and whether their results are fresh. Think in terms of latencies and availability. Let your SLOs guide your instrumentation and alerting.

RED, USE and The Four Golden Signals are known frameworks to get you started.

You should still measure internal metrics like database availability and CPU saturation—use them for troubleshooting and capacity planning.

```
Use causal metrics to answer why something is broken.
```

Labels are the new hierarchies

Labels are the new hierarchies, yet more powerful and flexible. Labels are what make Prometheus strong. Using labels, one can group and aggregate measurements afterwards. Slice and dice using labels. Remember Instrument first, ask questions later, provide as much context as possible. However, you have to *use labels with care*. See explanations below.

Avoid missing metrics

Time series that are not present until something happens are difficult to deal with. To avoid this, export 0 for any time series you know may exist in advance. Initialize your metrics with zero values to prevent broken dashboards and misfiring alerts. For a more detailed explanation, check out Existential issues with metrics.

Remember, labels create time series, so initialize your metrics with the labels you expect to use—your client libraries cannot know what labels you will have.

Cardinality matters

Every unique set of labels creates a new time series. Use labels with care and watch out what you put into them. Avoid cardinality explosions; unbounded labels will blow up Prometheus. Keep in mind that labels are multiplicative across dimensions.

```
Prometheus performance almost always comes down to one thing: label cardinality.
```

Always remember that Cardinality is key.

Naming is hard

A metric name must have a single meaning within a job, and ideally the same meaning across jobs too (like `process_cpu_seconds_total`).

Respect conventions over preferences. Conventions are no one's favorite, yet conventions are everyone's favorite. See the documentation on Naming for the nitty-gritty details.

Counters rule and gauges suck

Expose raw counters and let Prometheus derive rates for you with `rate()` and `increase()`. Don't precalculate rates on the target—that throws away information. Remember Instrument first, ask questions later.

Of course, gauges have their place. Use them for periodic measurements where there is simply nothing to count—temperature, disk fullness, queue depth. But if something only ever increases, make it a counter.

Don't add metrics for aggregations; PromQL can do it for you.

First the rate, then aggregate

Be aware of counter resets. As stated in Rate then sum, never sum then rate. As a rule of thumb, the only mathematical operations you can safely directly apply to a counter's values are `rate`, `irate`, `increase`, and `resets`. Anything else will cause you problems.

If you can log it, you can have a metric for it

Logs are often critical for troubleshooting, but metrics are equally valuable—a troubleshooting journey typically jumps back and forth between different signals. Don't underestimate the value of metrics for understanding what is going on.

Whenever you log an event, consider counting it by broad category. This is cheap and gives you alerting on elevated error rates plus an overview of what is happening. Remember, a metric without labels is cheap—spread counters liberally.

Native histograms are almost always better than classic histograms

Native histograms solve the bucket layout problem of classic histograms. They require no predefined boundaries, adjust resolution dynamically, and use sparse representation where empty

buckets cost nothing. If your instrumentation library or OTel supports them, prefer native histograms for new instrumentation.

With classic histograms, creating a correct bucket layout is an art. To ensure usefulness of your observations and correctness of your alerts, you have to come up with a meaningful bucket layout. This conflicts with Instrument first, ask questions later because you need to have an idea about your latencies before you even measure. Let your SLOs guide your bucket layout; create boundaries to match your SLO.

Classic histograms underneath are just counters with labels, where bucket boundaries are used as labels. Be cautious when adding additional labels to your histograms. Remember, *labels are multiplicative* and Cardinality matters.

If you can graph it, you can alert on it

You can't look at dashboards 24/7. Prometheus unifies metrics, dashboarding, and alerting. PromQL is the core of every Prometheus alert, and a PromQL query is the source of any graph on a dashboard. Use it.

If you run it, then you should put an alert on it

Always have an alert, *at least*, on presence and healthiness of your targets. You can't rely on things you can't observe or take action on.

Avoid missing and unhealthy targets. Prometheus generates the `up` metric automatically for each target. Use it.

Alerts should be urgent, important, actionable, and real

Alerts should be urgent, important, actionable, and real. As plain as it goes.

And don't over alert, alert-fatigue is real.

Symptom-based alerts for paging, cause-based for troubleshooting

Similar to Measure what users care about, alert on what *really* matters. It doesn't matter if your CPU is saturated, as long as your users don't notice. Let your SLOs guide your alerting.

Not every alert needs to page someone. Cause-based alerts shouldn't wake anyone up—they should feed into dashboards or ticket queues to be checked when needed or handled in batches at leisure. Reserve paging for symptom-based alerts that indicate user-facing impact.

For more context My Philosophy on Alerting.

Please five more minutes

Prometheus alerting rules let you specify a `for` duration that determines how long a condition must be true before the alert fires. If you don't specify it, a single failed scrape could cause an alert to fire. You need tolerance. As a rule of thumb, use at least 5 minutes unless you have a specific reason to do otherwise. For more information, check out Setting Thresholds on Alerts.

Context is king

Preserve common and useful labels for your alerts. They are most useful for routing and silencing.

The inspiration behind the idea comes from The Zen of Go, Go Proverbs and The Zen of Python.

Thank you very much all others that contributed with their invaluable ideas.

The initial rules are gathered from several sources from the community, such as Prometheus Proverbs by Björn Rabenstein, Best Practices and Beastly Pitfalls by Julius Volz, Patterns for Instrumenting Your Go Services by Bartek Plotka and Kemal Akkoyun, Instrumenting Applications and Alerting with Prometheus by Simon Pasquier and Robust Perception Blog by Brian Brazil.

Further resources

Following these conventions enables you to benefit from community tooling:

- Monitoring Mixins - Reusable dashboards and alerts for common services