Instrumentation

- How to instrument
  - The three types of services
    - Online-serving systems
    - Offline processing
    - Batch jobs
  - Subsystems
    - Libraries
    - Logging
    - Failures
    - Threadpools
    - Caches
    - Collectors
- Things to watch out for
  - Use labels
  - Do not overuse labels
  - Counter vs. gauge, summary vs. histogram
  - Timestamps, not time since
  - Inner loops
  - Avoid missing metrics

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.

□□□ #2
□ □□□ □□□ 2026-10-31 13:20:58 UTC
□ □□□ □□□ 2026-10-31 13:21:00 UTC