Prometheus

Show nav

Overview

What is Prometheus?

Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud. Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community. It is now a standalone open source project and maintained independently of any company. To emphasize this, and to clarify the project's governance structure, Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes.

Prometheus collects and stores its metrics as time series data, i.e. metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels.

For more elaborate overviews of Prometheus, see the resources linked from the media section.

Features

Prometheus's main features are:

- a multi-dimensional data model with time series data identified by metric name and key/value pairs
- PromQL, a flexible query language to leverage this dimensionality
- no reliance on distributed storage; single server nodes are autonomous
- time series collection happens via a pull model over HTTP
- pushing time series is supported via an intermediary gateway
- targets are discovered via service discovery or static configuration
- multiple modes of graphing and dashboarding support

What are metrics?

number of active connections or active queries, and so on.

Metrics play an important role in understanding why your application is working in a certain way. Let's assume you are running a web application and discover that it is slow. To learn what is happening with your application, you will need some information. For example, when the number of requests is high, the application may become slow. If you have the request count metric, you can determine the cause and increase the number of servers to handle the load.

Components

The Prometheus ecosystem consists of multiple components, many of which are optional:

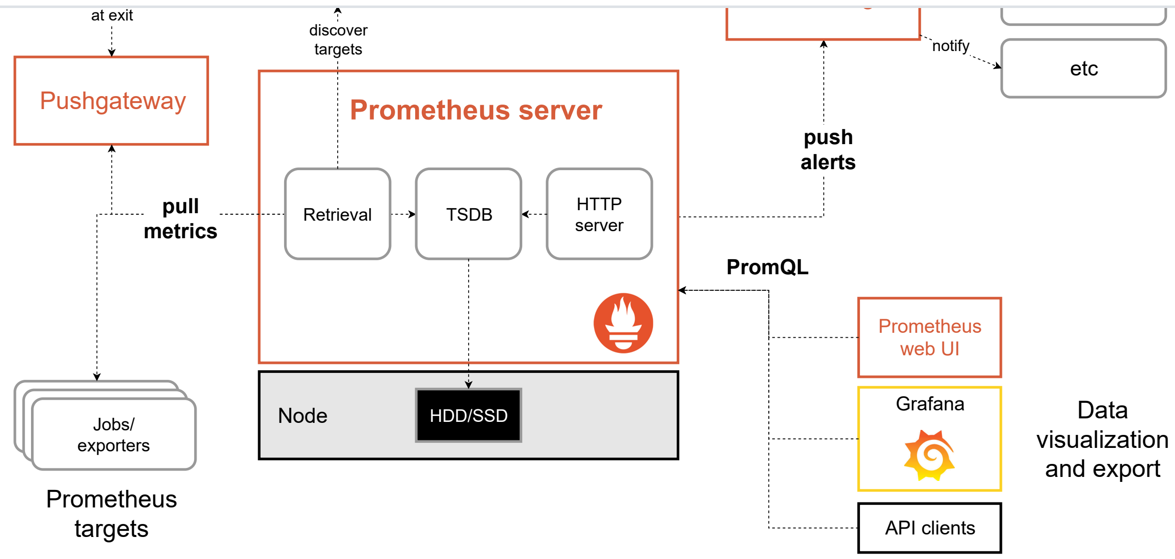
- the main Prometheus server which scrapes and stores time series data
- client libraries for instrumenting application code
- a push gateway for supporting short-lived jobs
- special-purpose exporters for services like HAProxy, StatsD, Graphite, etc.
- an alertmanager to handle alerts
- various support tools

Most Prometheus components are written in Go, making them easy to build and deploy as static binaries.

Architecture

This diagram illustrates the architecture of Prometheus and some of its ecosystem components:

Prometheus



Prometheus scrapes metrics from instrumented jobs, either directly or via an intermediary push gateway for short-lived jobs. It stores all scraped samples locally and runs rules over this data to either aggregate and record new time series from existing data or generate alerts. Grafana or other API consumers can be used to visualize the collected data.

When does it fit?

Prometheus works well for recording any purely numeric time series. It fits both machine-centric monitoring as well as monitoring of highly dynamic service-oriented architectures. In a world of microservices, its support for multi-dimensional data collection and querying is a particular strength.

Prometheus is designed for reliability, to be the system you go to during an outage to allow you to quickly diagnose problems. Each Prometheus server is standalone, not depending on network storage or other remote services. You can rely on it when other parts of your infrastructure are broken, and you do not need to setup extensive infrastructure to use it.

When does it not fit?

Prometheus values reliability. You can always view what statistics are available about your system, even under failure conditions. If you need 100% accuracy, such as for per-request billing, Prometheus is not a good choice as the collected data will likely not be detailed and complete enough. In such a case you would be best off using some other system to collect and analyze the data for billing, and Prometheus for the rest of your monitoring.

Prometheus

Prometheus

Show nav

First steps with Prometheus

Welcome to Prometheus! Prometheus is a monitoring platform that collects metrics from monitored targets by scraping metrics HTTP endpoints on these targets. This guide will show you how to install, configure and monitor our first resource with Prometheus. You'll download, install and run Prometheus. You'll also download and install an exporter, tools that expose time series data on hosts and services. Our first exporter will be Prometheus itself, which provides a wide variety of host-level metrics about memory usage, garbage collection, and more.

Downloading Prometheus

Download the latest release of Prometheus for your platform, then extract it:

```
tar xvfz prometheus-*.tar.gz
cd prometheus-*
```

The Prometheus server is a single binary called `prometheus` (or `prometheus.exe` on Microsoft Windows). We can run the binary and see help on its options by passing the `--help` flag.

```
./prometheus --help
usage: prometheus [<flags>]

The Prometheus monitoring server

. . .
```

Before starting Prometheus, let's configure it.

Configuring Prometheus

We've stripped out most of the comments in the example file to make it more succinct (comments are the lines prefixed with a `#`).

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s

rule_files:
  # - "first.rules"
  # - "second.rules"

scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: ['localhost:9090']
```

There are three blocks of configuration in the example configuration file: `global`, `rule_files`, and `scrape_configs`.

The `global` block controls the Prometheus server's global configuration. We have two options present. The first, `scrape_interval`, controls how often Prometheus will scrape targets. You can override this for individual targets. In this case the global setting is to scrape every 15 seconds. The `evaluation_interval` option controls how often Prometheus will evaluate rules. Prometheus uses rules to create new time series and to generate alerts.

The `rule_files` block specifies the location of any rules we want the Prometheus server to load. For now we've got no rules.

The last block, `scrape_configs`, controls what resources Prometheus monitors. Since Prometheus also exposes data about itself as an HTTP endpoint it can scrape and monitor its own health. In the default configuration there is a single job, called `prometheus`, which scrapes the time series data exposed by the Prometheus server. The job contains a single, statically configured, target, the `localhost` on port `9090`. Prometheus expects metrics to be available on targets on a path of `/metrics`. So this default job is scraping via the URL: http://localhost:9090/metrics.

The time series data returned will detail the state and performance of the Prometheus server.

For a complete specification of configuration options, see the configuration documentation.

containing the Prometheus binary and run:

```
./prometheus --config.file=prometheus.yml
```

Prometheus should start up. You should also be able to browse to a status page about itself at http://localhost:9090. Give it about 30 seconds to collect data about itself from its own HTTP metrics endpoint.

You can also verify that Prometheus is serving metrics about itself by navigating to its own metrics endpoint: http://localhost:9090/metrics.

Using the expression browser

Let us try looking at some data that Prometheus has collected about itself. To use Prometheus's built-in expression browser, navigate to http://localhost:9090/graph and choose the "Table" view within the "Graph" tab.

As you can gather from http://localhost:9090/metrics, one metric that Prometheus exports about itself is called `promhttp_metric_handler_requests_total` (the total number of `/metrics` requests the Prometheus server has served). Go ahead and enter this into the expression console:

```
promhttp_metric_handler_requests_total
```

This should return a number of different time series (along with the latest value recorded for each), all with the metric name `promhttp_metric_handler_requests_total`, but with different labels. These labels designate different requests statuses.

If we were only interested in requests that resulted in HTTP code `200`, we could use this query to retrieve that information:

```
promhttp_metric_handler_requests_total{code="200"}
```

To count the number of returned time series, you could write:

```
count(promhttp_metric_handler_requests_total)
```

Using the graphing interface

To graph expressions, navigate to http://localhost:9090/graph and use the "Graph" tab.

For example, enter the following expression to graph the per-second HTTP request rate returning status code 200 happening in the self-scraped Prometheus:

```
rate(promhttp_metric_handler_requests_total{code="200"}[1m])
```

You can experiment with the graph range parameters and other settings.

Monitoring other targets

Collecting metrics from Prometheus alone isn't a great representation of Prometheus' capabilities. To get a better sense of what Prometheus can do, we recommend exploring documentation about other exporters. The Monitoring Linux or macOS host metrics using a node exporter guide is a good place to start.

Summary

In this guide, you installed Prometheus, configured a Prometheus instance to monitor resources, and learned some basics of working with time series data in Prometheus' expression browser. To continue learning about Prometheus, check out the Overview for some ideas about what to explore next.

← **Previous** Overview

Edit

Next Comparison to alternatives →

Prometheus

Show nav

Comparison to alternatives

Prometheus vs. Graphite

Scope

Graphite focuses on being a passive time series database with a query language and graphing features. Any other concerns are addressed by external components.

Prometheus is a full monitoring and trending system that includes built-in and active scraping, storing, querying, graphing, and alerting based on time series data. It has knowledge about what the world should look like (which endpoints should exist, what time series patterns mean trouble, etc.), and actively tries to find faults.

Data model

Graphite stores numeric samples for named time series, much like Prometheus does. However, Prometheus's metadata model is richer: while Graphite metric names consist of dot-separated components which implicitly encode dimensions, Prometheus encodes dimensions explicitly as key-value pairs, called labels, attached to a metric name. This allows easy filtering, grouping, and matching by these labels via the query language.

Further, especially when Graphite is used in combination with StatsD, it is common to store only aggregated data over all monitored instances, rather than preserving the instance as a dimension and being able to drill down into individual problematic instances.

For example, storing the number of HTTP requests to API servers with the response code `500` and the method `POST` to the `/tracks` endpoint would commonly be encoded like this in Graphite/StatsD:

```
stats.api-server.tracks.post.500 -> 93
```

Prometheus

```
api_server_http_requests_total{method="POST",handler="/tracks",status="500",instance
api_server_http_requests_total{method="POST",handler="/tracks",status="500",instance
api_server_http_requests_total{method="POST",handler="/tracks",status="500",instance
```

Storage

Graphite stores time series data on local disk in the Whisper format, an RRD-style database that expects samples to arrive at regular intervals. Every time series is stored in a separate file, and new samples overwrite old ones after a certain amount of time.

Prometheus also creates one local file per time series, but allows storing samples at arbitrary intervals as scrapes or rule evaluations occur. Since new samples are simply appended, old data may be kept arbitrarily long. Prometheus also works well for many short-lived, frequently changing sets of time series.

Summary

Prometheus offers a richer data model and query language, in addition to being easier to run and integrate into your environment. If you want a clustered solution that can hold historical data long term, Graphite may be a better choice.

Prometheus vs. InfluxDB

InfluxDB is an open-source time series database, with a commercial option for scaling and clustering. The InfluxDB project was released almost a year after Prometheus development began, so we were unable to consider it as an alternative at the time. Still, there are significant differences between Prometheus and InfluxDB, and both systems are geared towards slightly different use cases.

Scope

For a fair comparison, we must also consider Kapacitor together with InfluxDB, as in combination they address the same problem space as Prometheus and the Alertmanager.

The same scope differences as in the case of Graphite apply here for InfluxDB itself. In addition InfluxDB offers continuous queries, which are equivalent to Prometheus recording

Alertmanager's notification functionality. Prometheus offers a more powerful query language for graphing and alerting. The Prometheus Alertmanager additionally offers grouping, deduplication and silencing functionality.

Data model / storage

Like Prometheus, the InfluxDB data model has key-value pairs as labels, which are called tags. In addition, InfluxDB has a second level of labels called fields, which are more limited in use. InfluxDB supports timestamps with up to nanosecond resolution, and float64, int64, bool, and string data types. Prometheus, by contrast, supports the float64 data type with limited support for strings, and millisecond resolution timestamps.

InfluxDB uses a variant of a log-structured merge tree for storage with a write ahead log, sharded by time. This is much more suitable to event logging than Prometheus's append-only file per time series approach.

Logs and Metrics and Graphs, Oh My! describes the differences between event logging and metrics recording.

Architecture

Prometheus servers run independently of each other and only rely on their local storage for their core functionality: scraping, rule processing, and alerting. The open source version of InfluxDB is similar.

The commercial InfluxDB offering is, by design, a distributed storage cluster with storage and queries being handled by many nodes at once.

This means that the commercial InfluxDB will be easier to scale horizontally, but it also means that you have to manage the complexity of a distributed storage system from the beginning. Prometheus will be simpler to run, but at some point you will need to shard servers explicitly along scalability boundaries like products, services, datacenters, or similar aspects. Independent servers (which can be run redundantly in parallel) may also give you better reliability and failure isolation.

Kapacitor's open-source release has no built-in distributed/redundant options for rules, alerting, or notifications. The open-source release of Kapacitor can be scaled via manual sharding by the user, similar to Prometheus itself. Influx offers Enterprise Kapacitor, which supports an HA/redundant alerting system.

Prometheus

Summary

There are many similarities between the systems. Both have labels (called tags in InfluxDB) to efficiently support multi-dimensional metrics. Both use basically the same data compression algorithms. Both have extensive integrations, including with each other. Both have hooks allowing you to extend them further, such as analyzing data in statistical tools or performing automated actions.

Where InfluxDB is better:

- If you're doing event logging.
- Commercial option offers clustering for InfluxDB, which is also better for long term data storage.
- Eventually consistent view of data between replicas.

Where Prometheus is better:

- If you're primarily doing metrics.
- More powerful query language, alerting, and notification functionality.
- Higher availability and uptime for graphing and alerting.

InfluxDB is maintained by a single commercial company following the open-core model, offering premium features like closed-source clustering, hosting and support. Prometheus is a fully open source and independent project, maintained by a number of companies and individuals, some of whom also offer commercial services and support.

Prometheus vs. OpenTSDB

OpenTSDB is a distributed time series database based on Hadoop and HBase.

Scope

The same scope differences as in the case of Graphite apply here.

Data model

Prometheus

Prometheus allows arbitrary characters in label values, while OpenTSDB is more restrictive. OpenTSDB also lacks a full query language, only allowing simple aggregation and math via its API.

Storage

OpenTSDB's storage is implemented on top of Hadoop and HBase. This means that it is easy to scale OpenTSDB horizontally, but you have to accept the overall complexity of running a Hadoop/HBase cluster from the beginning.

Prometheus will be simpler to run initially, but will require explicit sharding once the capacity of a single node is exceeded.

Summary

Prometheus offers a much richer query language, can handle higher cardinality metrics, and forms part of a complete monitoring system. If you're already running Hadoop and value long term storage over these benefits, OpenTSDB is a good choice.

Prometheus vs. Nagios

Nagios is a monitoring system that originated in the 1990s as NetSaint.

Scope

Nagios is primarily about alerting based on the exit codes of scripts. These are called “checks”. There is silencing of individual alerts, however no grouping, routing or deduplication.

There are a variety of plugins. For example, piping the few kilobytes of perfData plugins are allowed to return to a time series database such as Graphite or using NRPE to run checks on remote machines.

Data model

There is no notion of labels or a query language.

Storage

Nagios has no storage per-se, beyond the current check state. There are plugins which can store data such as for visualisation.

Architecture

Nagios servers are standalone. All configuration of checks is via file.

Summary

Nagios is suitable for basic monitoring of small and/or static systems where blackbox probing is sufficient.

If you want to do whitebox monitoring, or have a dynamic or cloud based environment, then Prometheus is a good choice.

Prometheus vs. Sensu

Sensu is an open source monitoring and observability pipeline with a commercial distribution which offers additional features for scalability. It can reuse existing Nagios plugins.

Scope

Sensu is an observability pipeline that focuses on processing and alerting of observability data as a stream of Events. It provides an extensible framework for event filtering, aggregation, transformation, and processing – including sending alerts to other systems and storing events in third-party systems. Sensu's event processing capabilities are similar in scope to Prometheus alerting rules and Alertmanager.

Data model

Event payload may include one or more metric `points`, represented as a JSON object containing a `name`, `tags` (key/value pairs), `timestamp`, and `value` (always a float).

Storage

Sensu stores current and recent event status information and real-time inventory data in an embedded database (etcd) or an external RDBMS (PostgreSQL).

Architecture

All components of a Sensu deployment can be clustered for high availability and improved event-processing throughput.

Summary

Sensu and Prometheus have a few capabilities in common, but they take very different approaches to monitoring. Both offer extensible discovery mechanisms for dynamic cloud-based environments and ephemeral compute platforms, though the underlying mechanisms are quite different. Both provide support for collecting multi-dimensional metrics via labels and annotations. Both have extensive integrations, and Sensu natively supports collecting metrics from all Prometheus exporters. Both are capable of forwarding observability data to third-party data platforms (e.g. event stores or TSDBs). Where Sensu and Prometheus differ the most is in their use cases.

Where Sensu is better:

- If you're collecting and processing hybrid observability data (including metrics *and/or* events)
- If you're consolidating multiple monitoring tools and need support for metrics *and* Nagios-style plugins or check scripts
- More powerful event-processing platform

Where Prometheus is better:

- If you're primarily collecting and evaluating metrics
- If you're monitoring homogeneous Kubernetes infrastructure (if 100% of the workloads you're monitoring are in K8s, Prometheus offers better K8s integration)
- More powerful query language, and built-in support for historical data analysis

maintained by a number of companies and individuals, some of whom also offer commercial services and support.

Previous
First steps

Edit

Next
FAQ

Prometheus

Show nav

Frequently asked questions

General

What is Prometheus?

Prometheus is an open-source systems monitoring and alerting toolkit with an active ecosystem. It is the only system directly supported by Kubernetes and the de facto standard across the cloud native ecosystem. See the overview.

How does Prometheus compare against other monitoring systems?

See the comparison page.

What dependencies does Prometheus have?

The main Prometheus server runs standalone as a single monolithic binary and has no external dependencies.

Is this cloud native?

Yes.

Cloud native is a flexible operating model, breaking up old service boundaries to allow for more flexible and scalable deployments.

Prometheus's service discovery integrates with most tools and clouds. Its dimensional data model and scale into the tens of millions of active series allows it to monitor large cloud-native deployments. There are always trade-offs to make when running services, and Prometheus values reliably getting alerts out to humans above all else.

be deduplicated by the Alertmanager.

Alertmanager supports high availability by interconnecting multiple Alertmanager instances to build an Alertmanager cluster. Instances of a cluster communicate using a gossip protocol managed via HashiCorp's Memberlist library.

I was told Prometheus “doesn't scale”.

This is often more of a marketing claim than anything else.

A single instance of Prometheus can be more performant than some systems positioning themselves as long term storage solution for Prometheus. You can run Prometheus reliably with tens of millions of active series.

If you need more than that, there are several options. Scaling and Federating Prometheus on the Robust Perception blog is a good starting point, as are the long storage systems listed on our integrations page.

What language is Prometheus written in?

Most Prometheus components are written in Go. Some are also written in Java, Python, and Ruby.

How stable are Prometheus features, storage formats, and APIs?

All repositories in the Prometheus GitHub organization that have reached version 1.0.0 broadly follow semantic versioning. Breaking changes are indicated by increments of the major version. Exceptions are possible for experimental components, which are clearly marked as such in announcements.

Even repositories that have not yet reached version 1.0.0 are, in general, quite stable. We aim for a proper release process and an eventual 1.0.0 release for each repository. In any case, breaking changes will be pointed out in release notes (marked by `[CHANGE]`) or communicated clearly for components that do not have formal releases yet.

Why do you pull rather than push?

developing changes.
- You can more easily and reliably tell if a target is down.
- You can manually go to a target and inspect its health with a web browser.

Overall, we believe that pulling is slightly better than pushing, but it should not be considered a major point when considering a monitoring system.

For cases where you must push, we offer the Pushgateway.

How to feed logs into Prometheus?

Short answer: Don't! Use something like Grafana Loki or OpenSearch instead.

Longer answer: Prometheus is a system to collect and process metrics, not an event logging system. The Grafana blog post Logs and Metrics and Graphs, Oh My! provides more details about the differences between logs and metrics.

If you want to extract Prometheus metrics from application logs, Grafana Loki is designed for just that. See Loki's metric queries documentation.

Who wrote Prometheus?

Prometheus was initially started privately by Matt T. Proud and Julius Volz. The majority of its initial development was sponsored by SoundCloud.

It's now maintained and extended by a wide range of companies and individuals.

What license is Prometheus released under?

Prometheus is released under the Apache 2.0 license.

What is the plural of Prometheus?

After extensive research, it has been determined that the correct plural of 'Prometheus' is 'Prometheis'.

If you can not remember this, "Prometheus instances" is a good workaround.

endpoint will reload and apply the configuration file. The various components attempt to handle failing changes gracefully.

Can I send alerts?

Yes, with the Alertmanager.

We support sending alerts through email, various native integrations, and a webhook system anyone can add integrations to.

Can I create dashboards?

Yes, we recommend Grafana for production usage. There are also Console templates.

Can I change the timezone? Why is everything in UTC?

To avoid any kind of timezone confusion, especially when the so-called daylight saving time is involved, we decided to exclusively use Unix time internally and UTC for display purposes in all components of Prometheus. A carefully done timezone selection could be introduced into the UI. Contributions are welcome. See issue #500 for the current state of this effort.

Instrumentation

Which languages have instrumentation libraries?

There are a number of client libraries for instrumenting your services with Prometheus metrics. See the client libraries documentation for details.

If you are interested in contributing a client library for a new language, see the exposition formats.

Can I monitor machines?

Yes, the Node Exporter exposes an extensive set of machine-level metrics on Linux and other Unix systems such as CPU usage, memory, disk utilization, filesystem fullness, and network bandwidth.

networks, there's also a Modbus exporter.

Can I monitor batch jobs?

Yes, using the Pushgateway. See also the best practices for monitoring batch jobs.

What applications can Prometheus monitor out of the box?

See the list of exporters and integrations.

Can I monitor JVM applications via JMX?

Yes, for applications that you cannot instrument directly with the Java client, you can use the JMX Exporter either standalone or as a Java Agent.

What is the performance impact of instrumentation?

Performance across client libraries and languages may vary. For Java, benchmarks indicate that incrementing a counter/gauge with the Java client will take 12-17ns, depending on contention. This is negligible for all but the most latency-critical code.

Implementation

Why are all sample values 64-bit floats?

We restrained ourselves to 64-bit floats to simplify the design. The IEEE 754 double-precision binary floating-point format supports integer precision for values up to 2^{53}. Supporting native 64 bit integers would (only) help if you need integer precision above 2^{53} but below 2^{63}. In principle, support for different sample value types (including some kind of big integer, supporting even more than 64 bit) could be implemented, but it is not a priority right now. A counter, even if incremented one million times per second, will only run into precision issues after over 285 years.

Prometheus

Prometheus

Show nav

Roadmap

The following is only a selection of some of the major features we plan to implement in the near future. To get a more complete overview of planned features and current work, see the issue trackers for the various repositories, for example, the Prometheus server.

Server-side metric metadata support

At this time, metric types and other metadata are only used in the client libraries and in the exposition format, but not persisted or utilized in the Prometheus server. We plan on making use of this metadata in the future. The first step is to aggregate this data in-memory in Prometheus and provide it via an experimental API endpoint.

Adopt OpenMetrics

The OpenMetrics working group is developing a new standard for metric exposition. We plan to support this format in our client libraries and Prometheus itself.

Retroactive rule evaluations

Add support for retroactive rule evaluations making use of backfill.

TLS and authentication in HTTP serving endpoints

TLS and authentication are currently being rolled out to the Prometheus, Alertmanager, and the official exporters. Adding this support will make it easier for people to deploy Prometheus components securely without requiring a reverse proxy to add those features externally.

Support the Ecosystem

Prometheus

Previous
FAQ

Edit

Next
Design documents

Prometheus

Show nav

Design documents

See the github.com/prometheus/proposals repository to see all the past and current proposals for the Prometheus Ecosystem.

If you are interested in creating a new proposal, read our proposal process.

Problem statements and exploratory documents

Sometimes we're looking even further into potential futures. The documents in this section are largely exploratory. They should be taken as informing our collective thoughts, not as anything concrete or specific.

Document	Initial date
Prometheus is not feature complete	2020-05
Thoughts about timestamps and durations in PromQL	2020-10
Prometheus, OpenMetrics & OTLP	2021-03
Prometheus Sparse Histograms and PromQL	2021-10
Quoting Prometheus names	2023-01

← Previous Roadmap

Edit

Next Media →

Prometheus

Prometheus

Show nav

Media

There is a subreddit collecting all Prometheus-related resources on the internet.

The following selection of resources are particularly useful to get started with Prometheus. Awesome Prometheus contains a more comprehensive community-maintained list of resources.

Blogs

- This site has its own blog.
- SoundCloud's blog post announcing Prometheus – a more elaborate overview than the one given on this site.
- Prometheus-related posts on the Robust Perception blog.

Tutorials

- Instructions and example code for a Prometheus workshop.
- How To Install Prometheus using Docker on Ubuntu 14.04.

Podcasts and interviews

- Prometheus on FLOSS Weekly 357 - Julius Volz on the FLOSS Weekly TWiT.tv show.
- Prometheus and Service Monitoring - Julius Volz on the Changelog podcast.

Recorded talks

- Prometheus: A Next-Generation Monitoring System – Julius Volz and Björn Rabenstein at SREcon15 Europe, Dublin.
- Prometheus: A Next-Generation Monitoring System - Brian Brazil at FOSDEM 2016 (slides).

- Monitoring Hadoop with Prometheus – Brian Brazil at the Hadoop User Group Ireland (slides).
- In German: Monitoring mit Prometheus – Michael Stapelberg at Easterhegg 2016.
- In German: Prometheus in der Praxis – Jonas Große Sundrup at MRMCD 2016

Presentation slides

General

- Prometheus Overview – by Brian Brazil.
- Systems Monitoring with Prometheus – Brian Brazil at Devops Ireland Meetup, Dublin.
- OMG! Prometheus – Benjamin Staffin, Fitbit Site Operations, explains the case for Prometheus to his team.

Docker

- Prometheus and Docker – Brian Brazil at Docker Galway.

Python

- Better Monitoring for Python – Brian Brazil at Pycon Ireland.
- Monitoring your Python with Prometheus – Brian Brazil at Python Ireland Meetup, Dublin.

← Previous Design documents

Edit

Next Glossary →

Prometheus

Prometheus

Show nav

Glossary

Alert

An alert is the outcome of an alerting rule in Prometheus that is actively firing. Alerts are sent from Prometheus to the Alertmanager.

Alertmanager

The Alertmanager takes in alerts, aggregates them into groups, de-duplicates, applies silences, throttles, and then sends out notifications to email, Pagerduty, Slack etc.

Bridge

A bridge is a component that takes samples from a client library and exposes them to a non-Prometheus monitoring system. For example, the Python, Go, and Java clients can export metrics to Graphite.

Client library

A client library is a library in some language (e.g. Go, Java, Python, Ruby) that makes it easy to directly instrument your code, write custom collectors to pull metrics from other systems and expose the metrics to Prometheus.

Collector

A collector is a part of an exporter that represents a set of metrics. It may be a single metric if it is part of direct instrumentation, or many metrics if it is pulling metrics from another system.

Direct instrumentation

Prometheus

Endpoint

A source of metrics that can be scraped, usually corresponding to a single process.

Exporter

An exporter is a binary running alongside the application you want to obtain metrics from. The exporter exposes Prometheus metrics, commonly by converting metrics that are exposed in a non-Prometheus format into a format that Prometheus supports.

Instance

An instance is a label that uniquely identifies a target in a job.

Job

A collection of targets with the same purpose, for example monitoring a group of like processes replicated for scalability or reliability, is called a job.

Notification

A notification represents a group of one or more alerts, and is sent by the Alertmanager to email, Pagerduty, Slack etc.

Promdash

Promdash was a native dashboard builder for Prometheus. It has been deprecated and replaced by Grafana.

Prometheus

Prometheus usually refers to the core binary of the Prometheus system. It may also refer to the Prometheus monitoring system as a whole.

including aggregation, slicing and dicing, prediction and joins.

Pushgateway

The Pushgateway persists the most recent push of metrics from batch jobs. This allows Prometheus to scrape their metrics after they have terminated.

Recording Rules

Recording rules precompute frequently needed or computationally expensive expressions and save their results as a new set of time series.

Remote Read

Remote read is a Prometheus feature that allows transparent reading of time series from other systems (such as long term storage) as part of queries.

Remote Read Adapter

Not all systems directly support remote read. A remote read adapter sits between Prometheus and another system, converting time series requests and responses between them.

Remote Read Endpoint

A remote read endpoint is what Prometheus talks to when doing a remote read.

Remote Write

Remote write is a Prometheus feature that allows sending ingested samples on the fly to other systems, such as long term storage.

Remote Write Adapter

Prometheus

Remote Write Endpoint

A remote write endpoint is what Prometheus talks to when doing a remote write.

Sample

A sample is a single value at a point in time in a time series.

In Prometheus, each sample consists of a float64 value and a millisecond-precision timestamp.

Silence

A silence in the Alertmanager prevents alerts, with labels matching the silence, from being included in notifications.

Target

A target is the definition of an object to scrape. For example, what labels to apply, any authentication required to connect, or other information that defines how the scrape will occur.

Time Series

The Prometheus time series are streams of timestamped values belonging to the same metric and the same set of labeled dimensions. Prometheus stores all data as time series.

← **Previous** Media

Edit

Next Long-term support →

Prometheus

Prometheus

Show nav

Long-term support

Prometheus LTS are selected releases of Prometheus that receive bugfixes for an extended period of time.

Every 6 weeks, a new Prometheus minor release cycle begins. After those 6 weeks, minor releases generally no longer receive bugfixes. If a user is impacted by a bug in a minor release, they often need to upgrade to the latest Prometheus release.

Upgrading Prometheus should be straightforward thanks to our API stability guarantees. However, there is a risk that new features and enhancements could also bring regressions, requiring another upgrade.

Prometheus LTS only receive bug, security, and documentation fixes, but over a time window of one year. The build toolchain will also be kept up-to-date. This allows companies that rely on Prometheus to limit the upgrade risks while still having a Prometheus server maintained by the community.

List of LTS releases

Release	Date	End of support
Prometheus 2.37	2022-07-14	2023-07-31
Prometheus 2.45	2023-06-23	2024-07-31
Prometheus 2.53	2024-06-16	2025-07-31
Prometheus 3.5	2025-07-14	2026-07-31

Limitations of LTS support

Some features are excluded from LTS support:

- Things listed as unstable in our API stability guarantees.
- Experimental features.

Prometheus

Previous
Glossary

Edit

Next
Data model

Prometheus

Show nav

Data model

Prometheus fundamentally stores all data as *time series*: streams of timestamped values belonging to the same metric and the same set of labeled dimensions. Besides stored time series, Prometheus may generate temporary derived time series as the result of queries.

Metric names and labels

Every time series is uniquely identified by its metric name and optional key-value pairs called labels.

Metric names:

- Metric names SHOULD specify the general feature of a system that is measured (e.g. `http_requests_total` - the total number of HTTP requests received).
- Metric names MAY use any UTF-8 characters.
- Metric names SHOULD match the regex `[a-zA-Z_:][a-zA-Z0-9_:]*` for the best experience and compatibility (see the warning below). Metric names outside of that set will require quoting e.g. when used in PromQL (see the UTF-8 guide).

> **NOTE**: Colons (':') are reserved for user-defined recording rules. They SHOULD NOT be used by exporters or direct instrumentation.

Metric labels:

Labels let you capture different instances of the same metric name. For example: all HTTP requests that used the method `POST` to the `/api/tracks` handler. We refer to this as Prometheus's "dimensional data model". The query language allows filtering and aggregation based on these dimensions. The change of any label's value, including adding or removing labels, will create a new time series.

- Label names MAY use any UTF-8 characters.

experience and compatibility (see the warning below). Label names outside of that regex will require quoting e.g. when used in PromQL (see the UTF-8 guide).

- Label values MAY contain any UTF-8 characters.
- Labels with an empty label value are considered equivalent to labels that do not exist.

WARNING: The UTF-8 support for metric and label names was added relatively recently in Prometheus v3.0.0. It might take time for the wider ecosystem (downstream PromQL compatible projects and vendors, tooling, third-party instrumentation, collectors, etc.) to adopt new quoting mechanisms, relaxed validation etc. For the best compatibility it's recommended to stick to the recommended ("SHOULD") character set.

INFO: See also the best practices for naming metrics and labels.

Samples

Samples form the actual time series data. Each sample consists of:

- a float64 or native histogram value
- a millisecond-precision timestamp

Notation

Given a metric name and a set of labels, time series are frequently identified using this notation:

```
<metric name>{<label name>="<label value>", ...}
```

For example, a time series with the metric name `api_http_requests_total` and the labels `method="POST"` and `handler="/messages"` could be written like this:

```
api_http_requests_total{method="POST", handler="/messages"}
```

This is the same notation that OpenTSDB uses.

Names with UTF-8 characters outside the recommended set must be quoted, using this notation:

Since metric name are internally represented as a label pair with a special label name (`__name__="<metric name>"`) one could also use the following notation:

```
{__name__="<metric name>", <label name>="<label value>", ...}
```

← **Previous** Long-term support

Edit

Next Metric types →

Prometheus

Show nav

Metric types

The Prometheus client libraries offer four core metric types. These are currently only differentiated in the client libraries (to enable APIs tailored to the usage of the specific types) and in the wire protocol. The Prometheus server does not yet make use of the type information and flattens all data into untyped time series. This may change in the future.

Counter

A *counter* is a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart. For example, you can use a counter to represent the number of requests served, tasks completed, or errors.

Do not use a counter to expose a value that can decrease. For example, do not use a counter for the number of currently running processes; instead use a gauge.

Client library usage documentation for counters:

- Go
- Java
- Python
- Ruby
- .Net
- Rust

Gauge

A *gauge* is a metric that represents a single numerical value that can arbitrarily go up and down.

Gauges are typically used for measured values like temperatures or current memory usage, but also "counts" that can go up and down, like the number of concurrent requests.

Client library usage documentation for gauges:

- Ruby
- .Net
- Rust

Histogram

A *histogram* samples observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values.

A histogram with a base metric name of `<basename>` exposes multiple time series during a scrape:

- cumulative counters for the observation buckets, exposed as `<basename>_bucket{le="<upper inclusive bound>"}`
- the **total sum** of all observed values, exposed as `<basename>_sum`
- the **count** of events that have been observed, exposed as `<basename>_count` (identical to `<basename>_bucket{le="+Inf"}` above)

Use the `histogram_quantile()` function to calculate quantiles from histograms or even aggregations of histograms. A histogram is also suitable to calculate an Apdex score. When operating on buckets, remember that the histogram is cumulative. See histograms and summaries for details of histogram usage and differences to summaries.

> **NOTE**: Beginning with Prometheus v2.40, there is experimental support for native histograms. A native histogram requires only one time series, which includes a dynamic number of buckets in addition to the sum and count of observations. Native histograms allow much higher resolution at a fraction of the cost. Detailed documentation will follow once native histograms are closer to becoming a stable feature.

> **NOTE**: Beginning with Prometheus v3.0, the values of the `le` label of classic histograms are normalized during ingestion to follow the format of OpenMetrics Canonical Numbers.

Client library usage documentation for histograms:

- Ruby
- .Net
- Rust

Summary

Similar to a *histogram*, a *summary* samples observations (usually things like request durations and response sizes). While it also provides a total count of observations and a sum of all observed values, it calculates configurable quantiles over a sliding time window.

A summary with a base metric name of `<basename>` exposes multiple time series during a scrape:

- streaming **φ-quantiles** (0 ≤ φ ≤ 1) of observed events, exposed as `<basename>{quantile="<φ>"}`
- the **total sum** of all observed values, exposed as `<basename>_sum`
- the **count** of events that have been observed, exposed as `<basename>_count`

See histograms and summaries for detailed explanations of φ-quantiles, summary usage, and differences to histograms.

> **NOTE**: Beginning with Prometheus v3.0, the values of the `quantile` label are normalized during ingestion to follow the format of OpenMetrics Canonical Numbers.

Client library usage documentation for summaries:

- Go
- Java
- Python
- Ruby
- .Net

Prometheus

Prometheus

Show nav

Jobs and instances

In Prometheus terms, an endpoint you can scrape is called an *instance*, usually corresponding to a single process. A collection of instances with the same purpose, a process replicated for scalability or reliability for example, is called a *job*.

For example, an API server job with four replicated instances:

- job: `api-server`
  - instance 1: `1.2.3.4:5670`
  - instance 2: `1.2.3.4:5671`
  - instance 3: `5.6.7.8:5670`
  - instance 4: `5.6.7.8:5671`

Automatically generated labels and time series

When Prometheus scrapes a target, it attaches some labels automatically to the scraped time series which serve to identify the scraped target:

- `job` : The configured job name that the target belongs to.
- `instance` : The `<host>:<port>` part of the target's URL that was scraped.

If either of these labels are already present in the scraped data, the behavior depends on the `honor_labels` configuration option. See the scrape configuration documentation for more information.

For each instance scrape, Prometheus stores a sample in the following time series:

- `up{job="<job-name>", instance="<instance-id>"}` : `1` if the instance is healthy, i.e. reachable, or `0` if the scrape failed.
- `scrape_duration_seconds{job="<job-name>", instance="<instance-id>"}` : duration of the scrape.
- `scrape_samples_post_metric_relabeling{job="<job-name>", instance="<instance-id>"}` : the number of samples remaining after metric relabeling was applied.
- `scrape_samples_scraped{job="<job-name>", instance="<instance-id>"}` : the number of samples the target exposed.

The `up` time series is useful for instance availability monitoring.

With the `extra-scrape-metrics` feature flag several additional metrics are available:

- `scrape_timeout_seconds{job="<job-name>", instance="<instance-id>"}` : The configured `scrape_timeout` for a target.
- `scrape_sample_limit{job="<job-name>", instance="<instance-id>"}` : The configured `sample_limit` for a target. Returns zero if there is no limit configured.
- `scrape_body_size_bytes{job="<job-name>", instance="<instance-id>"}` : The uncompressed size of the most recent scrape response, if successful. Scrapes failing because `body_size_limit` is exceeded report -1, other scrape failures report 0.

←
Previous
Metric types

Edit

Next
Getting started
→

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Getting started

This guide is a "Hello World"-style tutorial which shows how to install, configure, and use a simple Prometheus instance. You will download and run Prometheus locally, configure it to scrape itself and an example application, then work with queries, rules, and graphs to use collected time series data.

Downloading and running Prometheus

Download the latest release of Prometheus for your platform, then extract and run it:

```
tar xvfz prometheus-*.tar.gz
cd prometheus-*
```

Before starting Prometheus, let's configure it.

Configuring Prometheus to monitor itself

Prometheus collects metrics from *targets* by scraping metrics HTTP endpoints. Since Prometheus exposes data in the same manner about itself, it can also scrape and monitor its own health.

While a Prometheus server that collects only data about itself is not very useful, it is a good starting example. Save the following basic Prometheus configuration as a file named `prometheus.yml`:

```
global:
  scrape_interval:     15s # By default, scrape targets every 15 seconds.
  external_labels:
    monitor: 'codelab-monitor'

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped fror
  - job_name: 'prometheus'

    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:9090']
```

For a complete specification of configuration options, see the configuration documentation.

Starting Prometheus

To start Prometheus with your newly created configuration file, change to the directory containing the Prometheus binary and run:

```
# Start Prometheus.
# By default, Prometheus stores its database in ./data (flag --storage.tsdb.path).
./prometheus --config.file=prometheus.yml
```

Prometheus should start up. You should also be able to browse to a status page about itself at localhost:9090. Give it a couple of seconds to collect data about itself from its own HTTP metrics endpoint.

You can also verify that Prometheus is serving metrics about itself by navigating to its metrics endpoint: localhost:9090/metrics

Using the expression browser

Let us explore data that Prometheus has collected about itself. To use Prometheus's built-in expression browser, navigate to http://localhost:9090/graph and choose the "Table" view itself is named `prometheus_target_interval_length_seconds` (the actual amount of time between target scrapes). Enter the below into the expression console and then click "Execute":

```
prometheus_target_interval_length_seconds
```

This should return a number of different time series (along with the latest value recorded for each), each with the metric name `prometheus_target_interval_length_seconds`, but with different labels. These labels designate different latency percentiles and target group intervals.

If we are interested only in 99th percentile latencies, we could use this query:

```
prometheus_target_interval_length_seconds{quantile="0.99"}
```

To count the number of returned time series, you could write:

```
count(prometheus_target_interval_length_seconds)
```

For more about the expression language, see the expression language documentation.

Using the graphing interface

To graph expressions, navigate to http://localhost:9090/graph and use the "Graph" tab.

For example, enter the following expression to graph the per-second rate of chunks being created in the self-scraped Prometheus:

```
rate(prometheus_tsdb_head_chunks_created_total[1m])
```

Experiment with the graph range parameters and other settings.

Starting up some sample targets

Let's add additional targets for Prometheus to scrape.

```
tar -xzvf node_exporter-*.*.tar.gz
cd node_exporter-*.*

# Start 3 example targets in separate terminals:
./node_exporter --web.listen-address 127.0.0.1:8080
./node_exporter --web.listen-address 127.0.0.1:8081
./node_exporter --web.listen-address 127.0.0.1:8082
```

You should now have example targets listening on http://localhost:8080/metrics, http://localhost:8081/metrics, and http://localhost:8082/metrics.

Configure Prometheus to monitor the sample targets

Now we will configure Prometheus to scrape these new targets. Let's group all three endpoints into one job called `node`. We will imagine that the first two endpoints are production targets, while the third one represents a canary instance. To model this in Prometheus, we can add several groups of endpoints to a single job, adding extra labels to each group of targets. In this example, we will add the `group="production"` label to the first group of targets, while adding `group="canary"` to the second.

To achieve this, add the following job definition to the `scrape_configs` section in your `prometheus.yml` and restart your Prometheus instance:

```
scrape_configs:
  - job_name:       'node'

    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:8080', 'localhost:8081']
        labels:
          group: 'production'

      - targets: ['localhost:8082']
```

Go to the expression browser and verify that Prometheus now has information about time series that these example endpoints expose, such as `node_cpu_seconds_total`.

Configure rules for aggregating scraped data into new time series

Though not a problem in our example, queries that aggregate over thousands of time series can get slow when computed ad-hoc. To make this more efficient, Prometheus can prerecord expressions into new persisted time series via configured *recording rules*. Let's say we are interested in recording the per-second rate of cpu time (`node_cpu_seconds_total`) averaged over all cpus per instance (but preserving the `job`, `instance` and `mode` dimensions) as measured over a window of 5 minutes. We could write this as:

```
avg by (job, instance, mode) (rate(node_cpu_seconds_total[5m]))
```

Try graphing this expression.

To record the time series resulting from this expression into a new metric called `job_instance_mode:node_cpu_seconds:avg_rate5m`, create a file with the following recording rule and save it as `prometheus.rules.yml`:

```
groups:
- name: cpu-node
  rules:
  - record: job_instance_mode:node_cpu_seconds:avg_rate5m
    expr: avg by (job, instance, mode) (rate(node_cpu_seconds_total[5m]))
```

To make Prometheus pick up this new rule, add a `rule_files` statement in your `prometheus.yml`. The config should now look like this:

```
global:
  scrape_interval:     15s # By default, scrape targets every 15 seconds.
  evaluation_interval: 15s # Evaluate rules every 15 seconds.

  # Attach these extra labels to all timeseries collected by this Prometheus instan
  external_labels:
    monitor: 'codelab-monitor'
```

Prometheus

```
scrape_configs:
  - job_name: 'prometheus'

    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:9090']

  - job_name:       'node'

    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:8080', 'localhost:8081']
        labels:
          group: 'production'

      - targets: ['localhost:8082']
        labels:
          group: 'canary'
```

Restart Prometheus with the new configuration and verify that a new time series with the metric name `job_instance_mode:node_cpu_seconds:avg_rate5m` is now available by querying it through the expression browser or graphing it.

Reloading configuration

As mentioned in the configuration documentation a Prometheus instance can have its configuration reloaded without restarting the process by using the `SIGHUP` signal. If you're running on Linux this can be performed by using `kill -s SIGHUP <PID>`, replacing `<PID>` with your Prometheus process ID.

Shutting down your instance gracefully.

to the Prometheus process. For example, you can use `kill -s <SIGNAL> <PID>`, replacing `<SIGNAL>` with the signal name and `<PID>` with the Prometheus process ID. Alternatively, you can press the interrupt character at the controlling terminal, which by default is `^C` (Control-C).

← **Previous** Jobs and instances

Edit

Next Installation →

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Installation

Using pre-compiled binaries

We provide precompiled binaries for most official Prometheus components. Check out the download section for a list of all available versions.

From source

For building Prometheus components from source, see the `Makefile` targets in the respective repository.

Using Docker

All Prometheus services are available as Docker images on Quay.io or Docker Hub.

Running Prometheus on Docker is as simple as `docker run -p 9090:9090 prom/prometheus`. This starts Prometheus with a sample configuration and exposes it on port 9090.

The Prometheus image uses a volume to store the actual metrics. For production deployments it is highly recommended to use a named volume to ease managing the data on Prometheus upgrades.

Setting command line parameters

The Docker image is started with a number of default command line parameters, which can be found in the Dockerfile (adjust the link to correspond with the version in use).

Prometheus

Volumes & bind-mount

To provide your own configuration, there are several options. Here are two examples.

Bind-mount your `prometheus.yml` from the host by running:

```
docker run \
    -p 9090:9090 \
    -v /path/to/prometheus.yml:/etc/prometheus/prometheus.yml \
    prom/prometheus
```

Or bind-mount the directory containing `prometheus.yml` onto `/etc/prometheus` by running:

```
docker run \
    -p 9090:9090 \
    -v /path/to/config:/etc/prometheus \
    prom/prometheus
```

Save your Prometheus data

Prometheus data is stored in `/prometheus` dir inside the container, so the data is cleared every time the container gets restarted. To save your data, you need to set up persistent storage (or bind mounts) for your container.

Run Prometheus container with persistent storage:

```
# Create persistent volume for your data
docker volume create prometheus-data
# Start Prometheus container
docker run \
    -p 9090:9090 \
    -v /path/to/prometheus.yml:/etc/prometheus/prometheus.yml \
    -v prometheus-data:/prometheus \
    prom/prometheus
```

the image. This works well if the configuration itself is rather static and the same across all environments.

For this, create a new directory with a Prometheus configuration and a `Dockerfile` like this:

```
FROM prom/prometheus
ADD prometheus.yml /etc/prometheus/
```

Now build and run it:

```
docker build -t my-prometheus .
docker run -p 9090:9090 my-prometheus
```

A more advanced option is to render the configuration dynamically on start with some tooling or even have a daemon update it periodically.

Using configuration management systems

If you prefer using configuration management systems you might be interested in the following third-party contributions:

Ansible

- prometheus-community/ansible

Chef

- rayrod2030/chef-prometheus

Puppet

- puppet/prometheus

SaltStack

Prometheus

Previous
Getting started

Edit

Next
Configuration

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Configuration

Prometheus is configured via command-line flags and a configuration file. While the command-line flags configure immutable system parameters (such as storage locations, amount of data to keep on disk and in memory, etc.), the configuration file defines everything related to scraping jobs and their instances, as well as which rule files to load.

To view all available command-line flags, run `./prometheus -h`.

Prometheus can reload its configuration at runtime. If the new configuration is not well-formed, the changes will not be applied. A configuration reload is triggered by sending a `SIGHUP` to the Prometheus process or sending a HTTP POST request to the `/-/reload` endpoint (when the `--web.enable-lifecycle` flag is enabled). This will also reload any configured rule files.

Configuration file

To specify which configuration file to load, use the `--config.file` flag.

The file is written in YAML format, defined by the scheme described below. Brackets indicate that a parameter is optional. For non-list parameters the value is set to the specified default.

Generic placeholders are defined as follows:

- `<boolean>`: a boolean that can take the values `true` or `false`
- `<duration>`: a duration matching the regular expression `((([0-9]+)y)?(([0-9]+)w)?(([0-9]+)d)?(([0-9]+)h)?(([0-9]+)m)?(([0-9]+)s)?(([0-9]+)ms)?|0)`, e.g. `1d`, `1h30m`, `5m`, `10s`
- `<filename>`: a valid path in the current working directory
- `<float>`: a floating-point number

- `<labelname>` : a string matching the regular expression `[a-zA-Z_][a-zA-Z0-9_]*` . Any other unsupported character in the source label should be converted to an underscore. For example, the label `app.kubernetes.io/name` should be written as `app_kubernetes_io_name` .
- `<labelvalue>` : a string of unicode characters
- `<path>` : a valid URL path
- `<scheme>` : a string that can take the values `http` or `https`
- `<secret>` : a regular string that is a secret, such as a password
- `<string>` : a regular string
- `<size>` : a size in bytes, e.g. `512MB` . A unit is required. Supported units: B, KB, MB, GB, TB, PB, EB.
- `<tmpl_string>` : a string which is template-expanded before usage

The other placeholders are specified separately.

A valid example file can be found here.

The global configuration specifies parameters that are valid in all other configuration contexts. They also serve as defaults for other configuration sections.

```
global:
  # How frequently to scrape targets by default.
  [ scrape_interval: <duration> | default = 1m ]

  # How long until a scrape request times out.
  # It cannot be greater than the scrape interval.
  [ scrape_timeout: <duration> | default = 10s ]

  # The protocols to negotiate during a scrape with the client.
  # Supported values (case sensitive): PrometheusProto, OpenMetricsText0.0.1,
  # OpenMetricsText1.0.0, PrometheusText0.0.4.
  # The default value changes to [ PrometheusProto, OpenMetricsText1.0.0, OpenMetri
  # when native_histogram feature flag is set.
  [ scrape_protocols: [<string>, ...] | default = [ OpenMetricsText1.0.0, OpenMetri

  # How frequently to evaluate rules.
  [ evaluation_interval: <duration> | default = 1m ]

  # Offset the rule evaluation timestamp of this particular group by the
  # specified duration into the past to ensure the underlying metrics have
```

Prometheus

```
[ rule_query_offset: <duration> | default = 0s ]

# The labels to add to any time series or alerts when communicating with
# external systems (federation, remote storage, Alertmanager).
# Environment variable references `${var}` or `$var` are replaced according
# to the values of the current environment variables.
# References to undefined variables are replaced by the empty string.
# The `$` character can be escaped by using `$$`.
external_labels:
  [ <labelname>: <labelvalue> ... ]

# File to which PromQL queries are logged.
# Reloading the configuration will reopen the file.
[ query_log_file: <string> ]

# File to which scrape failures are logged.
# Reloading the configuration will reopen the file.
[ scrape_failure_log_file: <string> ]

# An uncompressed response body larger than this many bytes will cause the
# scrape to fail. 0 means no limit. Example: 100MB.
# This is an experimental feature, this behaviour could
# change or be removed in the future.
[ body_size_limit: <size> | default = 0 ]

# Per-scrape limit on the number of scraped samples that will be accepted.
# If more than this number of samples are present after metric relabeling
# the entire scrape will be treated as failed. 0 means no limit.
[ sample_limit: <int> | default = 0 ]

# Limit on the number of labels that will be accepted per sample. If more
# than this number of labels are present on any sample post metric-relabeling,
# the entire scrape will be treated as failed. 0 means no limit.
[ label_limit: <int> | default = 0 ]

# Limit on the length (in bytes) of each individual label name. If any label
# name in a scrape is longer than this number post metric-relabeling, the
# entire scrape will be treated as failed. Note that label names are UTF-8
# encoded, and characters can take up to 4 bytes. 0 means no limit.
[ label_name_length_limit: <int> | default = 0 ]
```

```
    # encoded, and characters can take up to 4 bytes. 0 means no limit.
    [ label_value_length_limit: <int> | default = 0 ]

    # Limit per scrape config on number of unique targets that will be
    # accepted. If more than this number of targets are present after target
    # relabeling, Prometheus will mark the targets as failed without scraping them.
    # 0 means no limit. This is an experimental feature, this behaviour could
    # change in the future.
    [ target_limit: <int> | default = 0 ]

    # Limit per scrape config on the number of targets dropped by relabeling
    # that will be kept in memory. 0 means no limit.
    [ keep_dropped_targets: <int> | default = 0 ]

    # Specifies the validation scheme for metric and label names. Either blank or
    # "utf8" for full UTF-8 support, or "legacy" for letters, numbers, colons,
    # and underscores.
    [ metric_name_validation_scheme <string> | default "utf8" ]

  runtime:
    # Configure the Go garbage collector GOGC parameter
    # See: https://tip.golang.org/doc/gc-guide#GOGC
    # Lowering this number increases CPU usage.
    [ gogc: <int> | default = 75 ]

  # Rule files specifies a list of globs. Rules and alerts are read from
  # all matching files.
  rule_files:
    [ - <filepath_glob> ... ]

  # Scrape config files specifies a list of globs. Scrape configs are read from
  # all matching files and appended to the list of scrape configs.
  scrape_config_files:
    [ - <filepath_glob> ... ]

  # A list of scrape configurations.
  scrape_configs:
    [ - <scrape_config> ... ]

  # Alerting specifies settings related to the Alertmanager.
  alerting:
      [ - <alertmanager_config> ... ]

  # Settings related to the remote write feature.
  remote_write:
    [ - <remote_write> ... ]

  # Settings related to the OTLP receiver feature.
  # See https://prometheus.io/docs/guides/opentelemetry/ for best practices.
  otlp:
    [ promote_resource_attributes: [<string>, ...] | default = [ ] ]
    # Configures translation of OTLP metrics when received through the OTLP metrics
    # endpoint. Available values:
    # - "UnderscoreEscapingWithSuffixes" refers to commonly agreed normalization used
    #   by OpenTelemetry in https://github.com/open-telemetry/opentelemetry-collector
    # - "NoUTF8EscapingWithSuffixes" is a mode that relies on UTF-8 support in Prometh
    #   It preserves all special characters like dots, but still adds required metric
    #   for units and _total, as UnderscoreEscapingWithSuffixes does.
    [ translation_strategy: <string> | default = "UnderscoreEscapingWithSuffixes" ]
    # Enables adding "service.name", "service.namespace" and "service.instance.id"
    # resource attributes to the "target_info" metric, on top of converting
    # them into the "instance" and "job" labels.
    [ keep_identifying_resource_attributes: <boolean> | default = false]

  # Settings related to the remote read feature.
  remote_read:
    [ - <remote_read> ... ]

  # Storage related settings that are runtime reloadable.
  storage:
    [ tsdb: <tsdb> ]
    [ exemplars: <exemplars> ]

  # Configures exporting traces.
  tracing:
    [ <tracing_config> ]
```

<scrape_config>

Prometheus

Targets may be statically configured via the `static_configs` parameter or dynamically discovered using one of the supported service-discovery mechanisms.

Additionally, `relabel_configs` allow advanced modifications to any target and its labels before scraping.

```
# The job name assigned to scraped metrics by default.
job_name: <job_name>

# How frequently to scrape targets from this job.
[ scrape_interval: <duration> | default = <global_config.scrape_interval> ]

# Per-scrape timeout when scraping this job.
# It cannot be greater than the scrape interval.
[ scrape_timeout: <duration> | default = <global_config.scrape_timeout> ]

# The protocols to negotiate during a scrape with the client.
# Supported values (case sensitive): PrometheusProto, OpenMetricsText0.0.1,
# OpenMetricsText1.0.0, PrometheusText0.0.4, PrometheusText1.0.0.
[ scrape_protocols: [<string>, ...] | default = <global_config.scrape_protocols> ]

# Fallback protocol to use if a scrape returns blank, unparseable, or otherwise
# invalid Content-Type.
# Supported values (case sensitive): PrometheusProto, OpenMetricsText0.0.1,
# OpenMetricsText1.0.0, PrometheusText0.0.4, PrometheusText1.0.0.
[ fallback_scrape_protocol: <string> ]

# Whether to scrape a classic histogram, even if it is also exposed as a native
# histogram (has no effect without --enable-feature=native-histograms).
[ always_scrape_classic_histograms: <boolean> | default = false ]

# The HTTP resource path on which to fetch metrics from targets.
[ metrics_path: <path> | default = /metrics ]

# honor_labels controls how Prometheus handles conflicts between labels that are
# already present in scraped data and labels that Prometheus would attach
# server-side ("job" and "instance" labels, manually configured target
# labels, and labels generated by service discovery implementations).
#
# If honor_labels is set to "true", label conflicts are resolved by keeping label
```

Prometheus

```
# conflicting labels in the scraped data to "exported_<original-label>" (for
# example "exported_instance", "exported_job") and then attaching server-side
# labels.
#
# Setting honor_labels to "true" is useful for use cases such as federation and
# scraping the Pushgateway, where all labels specified in the target should be
# preserved.
#
# Note that any globally configured "external_labels" are unaffected by this
# setting. In communication with external systems, they are always applied only
# when a time series does not have a given label yet and are ignored otherwise.
[ honor_labels: <boolean> | default = false ]

# honor_timestamps controls whether Prometheus respects the timestamps present
# in scraped data.
#
# If honor_timestamps is set to "true", the timestamps of the metrics exposed
# by the target will be used.
#
# If honor_timestamps is set to "false", the timestamps of the metrics exposed
# by the target will be ignored.
[ honor_timestamps: <boolean> | default = true ]

# track_timestamps_staleness controls whether Prometheus tracks staleness of
# the metrics that have an explicit timestamps present in scraped data.
#
# If track_timestamps_staleness is set to "true", a staleness marker will be
# inserted in the TSDB when a metric is no longer present or the target
# is down.
[ track_timestamps_staleness: <boolean> | default = false ]

# Configures the protocol scheme used for requests.
[ scheme: <scheme> | default = http ]

# Optional HTTP URL parameters.
params:
  [ <string>: [<string>, ...] ]

# If enable_compression is set to "false", Prometheus will request uncompressed
# response from the scraped target.
[ enable_compression: <boolean> | default = true ]
[ scrape_failure_log_file: <string> ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]

# List of Azure service discovery configurations.
azure_sd_configs:
  [ - <azure_sd_config> ... ]

# List of Consul service discovery configurations.
consul_sd_configs:
  [ - <consul_sd_config> ... ]

# List of DigitalOcean service discovery configurations.
digitalocean_sd_configs:
  [ - <digitalocean_sd_config> ... ]

# List of Docker service discovery configurations.
docker_sd_configs:
  [ - <docker_sd_config> ... ]

# List of Docker Swarm service discovery configurations.
dockerswarm_sd_configs:
  [ - <dockerswarm_sd_config> ... ]

# List of DNS service discovery configurations.
dns_sd_configs:
  [ - <dns_sd_config> ... ]

# List of EC2 service discovery configurations.
ec2_sd_configs:
  [ - <ec2_sd_config> ... ]

# List of Eureka service discovery configurations.
eureka_sd_configs:
  [ - <eureka_sd_config> ... ]

# List of file service discovery configurations.
file_sd_configs:
  [ - <file_sd_config> ... ]
  [ - <gce_sd_config> ... ]

# List of Hetzner service discovery configurations.
hetzner_sd_configs:
  [ - <hetzner_sd_config> ... ]

# List of HTTP service discovery configurations.
http_sd_configs:
  [ - <http_sd_config> ... ]


# List of IONOS service discovery configurations.
ionos_sd_configs:
  [ - <ionos_sd_config> ... ]

# List of Kubernetes service discovery configurations.
kubernetes_sd_configs:
  [ - <kubernetes_sd_config> ... ]

# List of Kuma service discovery configurations.
kuma_sd_configs:
  [ - <kuma_sd_config> ... ]

# List of Lightsail service discovery configurations.
lightsail_sd_configs:
  [ - <lightsail_sd_config> ... ]

# List of Linode service discovery configurations.
linode_sd_configs:
  [ - <linode_sd_config> ... ]

# List of Marathon service discovery configurations.
marathon_sd_configs:
  [ - <marathon_sd_config> ... ]

# List of AirBnB's Nerve service discovery configurations.
nerve_sd_configs:
  [ - <nerve_sd_config> ... ]

# List of Nomad service discovery configurations.
nomad_sd_configs:
```

```
openstack_sd_configs:
  [ - <openstack_sd_config> ... ]

# List of OVHcloud service discovery configurations.
ovhcloud_sd_configs:
  [ - <ovhcloud_sd_config> ... ]

# List of PuppetDB service discovery configurations.
puppetdb_sd_configs:
  [ - <puppetdb_sd_config> ... ]

# List of Scaleway service discovery configurations.
scaleway_sd_configs:
  [ - <scaleway_sd_config> ... ]

# List of Zookeeper Serverset service discovery configurations.
serverset_sd_configs:
  [ - <serverset_sd_config> ... ]

# List of Triton service discovery configurations.
triton_sd_configs:
  [ - <triton_sd_config> ... ]

# List of Uyuni service discovery configurations.
uyuni_sd_configs:
  [ - <uyuni_sd_config> ... ]

# List of labeled statically configured targets for this job.
static_configs:
  [ - <static_config> ... ]

# List of target relabel configurations.
relabel_configs:
  [ - <relabel_config> ... ]

# List of metric relabel configurations.
metric_relabel_configs:
  [ - <relabel_config> ... ]

# An uncompressed response body larger than this many bytes will cause the
# scrape to fail. 0 means no limit. Example: 100MB.
```

Prometheus

```
# Per-scrape limit on the number of scraped samples that will be accepted.
# If more than this number of samples are present after metric relabeling
# the entire scrape will be treated as failed. 0 means no limit.
[ sample_limit: <int> | default = 0 ]

# Limit on the number of labels that will be accepted per sample. If more
# than this number of labels are present on any sample post metric-relabeling,
# the entire scrape will be treated as failed. 0 means no limit.
[ label_limit: <int> | default = 0 ]

# Limit on the length (in bytes) of each individual label name. If any label
# name in a scrape is longer than this number post metric-relabeling, the
# entire scrape will be treated as failed. Note that label names are UTF-8
# encoded, and characters can take up to 4 bytes. 0 means no limit.
[ label_name_length_limit: <int> | default = 0 ]

# Limit on the length (in bytes) of each individual label value. If any label
# value in a scrape is longer than this number post metric-relabeling, the
# entire scrape will be treated as failed. Note that label values are UTF-8
# encoded, and characters can take up to 4 bytes. 0 means no limit.
[ label_value_length_limit: <int> | default = 0 ]

# Limit per scrape config on number of unique targets that will be
# accepted. If more than this number of targets are present after target
# relabeling, Prometheus will mark the targets as failed without scraping them.
# 0 means no limit. This is an experimental feature, this behaviour could
# change in the future.
[ target_limit: <int> | default = 0 ]

# Limit per scrape config on the number of targets dropped by relabeling
# that will be kept in memory. 0 means no limit.
[ keep_dropped_targets: <int> | default = 0 ]

# Specifies the validation scheme for metric and label names. Either blank or
# "utf8" for full UTF-8 support, or "legacy" for letters, numbers, colons, and
# underscores.
[ metric_name_validation_scheme <string> | default "utf8" ]

# Limit on total number of positive and negative buckets allowed in a single
# native histogram. The resolution of a histogram with more buckets will be
```

```
[ native_histogram_bucket_limit: <int> | default = 0 ]

# Lower limit for the growth factor of one bucket to the next in each native
# histogram. The resolution of a histogram with a lower growth factor will be
# reduced as much as possible until it is within the limit.
# To set an upper limit for the schema (equivalent to "scale" in OTel's
# exponential histograms), use the following factor limits:
#
# +----------------------------+----------------------------+
# |        growth factor       | resulting schema AKA scale |
# +----------------------------+----------------------------+
# |          65536             |             -4             |
# +----------------------------+----------------------------+
# |            256             |             -3             |
# +----------------------------+----------------------------+
# |             16             |             -2             |
# +----------------------------+----------------------------+
# |              4             |             -1             |
# +----------------------------+----------------------------+
# |              2             |              0             |
# +----------------------------+----------------------------+
# |              1.4           |              1             |
# +----------------------------+----------------------------+
# |              1.1           |              2             |
# +----------------------------+----------------------------+
# |              1.09          |              3             |
# +----------------------------+----------------------------+
# |              1.04          |              4             |
# +----------------------------+----------------------------+
# |              1.02          |              5             |
# +----------------------------+----------------------------+
# |              1.01          |              6             |
# +----------------------------+----------------------------+
# |              1.005         |              7             |
# +----------------------------+----------------------------+
# |              1.002         |              8             |
# +----------------------------+----------------------------+
#
# 0 results in the smallest supported factor (which is currently ~1.0027 or
```

Prometheus

Where `<job_name>` must be unique across all scrape configurations.

`<http_config>`

A `http_config` allows configuring HTTP requests.

```
# Sets the `Authorization` header on every request with the
# configured username and password.
# username and username_file are mutually exclusive.
# password and password_file are mutually exclusive.
basic_auth:
  [ username: <string> ]
  [ username_file: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Sets the `Authorization` header on every request with
# the configured credentials.
authorization:
  # Sets the authentication type of the request.
  [ type: <string> | default: Bearer ]
  # Sets the credentials of the request. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials of the request with the credentials read from the
  # configured file. It is mutually exclusive with `credentials`.
  [ credentials_file: <filename> ]

# Optional OAuth 2.0 configuration.
# Cannot be used at the same time as basic_auth or authorization.
oauth2:
  [ <oauth2> ]

# Configure whether requests follow HTTP 3xx redirects.
[ follow_redirects: <boolean> | default = true ]

# Whether to enable HTTP2.
[ enable_http2: <boolean> | default: true ]
```

```
# Optional proxy URL.
[ proxy_url: <string> ]
# Comma-separated string that can contain IPs, CIDR notation, domain names
# that should be excluded from proxying. IP and domain names can
# contain port numbers.
[ no_proxy: <string> ]
# Use proxy URL indicated by environment variables (HTTP_PROXY, https_proxy, HTTPs_F
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

# Custom HTTP headers to be sent along with each request.
# Headers that are set by Prometheus itself can't be overwritten.
http_headers:
  # Header name.
  [ <string>:
    # Header values.
    [ values: [<string>, ...] ]
    # Headers values. Hidden in configuration page.
    [ secrets: [<secret>, ...] ]
    # Files to read header values from.
    [ files: [<string>, ...] ] ]
```

<tls_config>

A `tls_config` allows configuring TLS connections.

```
# CA certificate to validate API server certificate with. At most one of ca and ca_f
[ ca: <string> ]
[ ca_file: <filename> ]

# Certificate and key for client cert authentication to the server.
# At most one of cert and cert_file is allowed.
# At most one of key and key_file is allowed.
[ cert: <string> ]
[ cert_file: <filename> ]
```

Prometheus

```
# ServerName extension to indicate the name of the server.
# https://tools.ietf.org/html/rfc4366#section-3.1
[ server_name: <string> ]

# Disable validation of the server certificate.
[ insecure_skip_verify: <boolean> ]

# Minimum acceptable TLS version. Accepted values: TLS10 (TLS 1.0), TLS11 (TLS
# 1.1), TLS12 (TLS 1.2), TLS13 (TLS 1.3).
# If unset, Prometheus will use Go default minimum version, which is TLS 1.2.
# See MinVersion in https://pkg.go.dev/crypto/tls#Config.
[ min_version: <string> ]
# Maximum acceptable TLS version. Accepted values: TLS10 (TLS 1.0), TLS11 (TLS
# 1.1), TLS12 (TLS 1.2), TLS13 (TLS 1.3).
# If unset, Prometheus will use Go default maximum version, which is TLS 1.3.
# See MaxVersion in https://pkg.go.dev/crypto/tls#Config.
[ max_version: <string> ]
```

`<oauth2>`

OAuth 2.0 authentication using the client credentials grant type. Prometheus fetches an access token from the specified endpoint with the given client access and secret keys.

```
client_id: <string>
[ client_secret: <secret> ]

# Read the client secret from a file.
# It is mutually exclusive with `client_secret`.
[ client_secret_file: <filename> ]

# Scopes for the token request.
scopes:
  [ - <string> ... ]

# The URL to fetch the token from.
token_url: <string>

# Optional parameters to append to the token URL.
```

```
  # Configures the token request's TLS settings.
  tls_config:
    [ <tls_config> ]

  # Optional proxy URL.
  [ proxy_url: <string> ]
  # Comma-separated string that can contain IPs, CIDR notation, domain names
  # that should be excluded from proxying. IP and domain names can
  # contain port numbers.
  [ no_proxy: <string> ]
  # Use proxy URL indicated by environment variables (HTTP_PROXY, https_proxy, HTTPs_F
  [ proxy_from_environment: <boolean> | default: false ]
  # Specifies headers to send to proxies during CONNECT requests.
  [ proxy_connect_header:
    [ <string>: [<secret>, ...] ] ]

  # Custom HTTP headers to be sent along with each request.
  # Headers that are set by Prometheus itself can't be overwritten.
  http_headers:
    # Header name.
    [ <string>:
      # Header values.
      [ values: [<string>, ...] ]
      # Headers values. Hidden in configuration page.
      [ secrets: [<secret>, ...] ]
      # Files to read header values from.
      [ files: [<string>, ...] ] ]
```

`<azure_sd_config>`

Azure SD configurations allow retrieving scrape targets from Azure VMs.

The discovery requires at least the following permissions:

- `Microsoft.Compute/virtualMachines/read` : Required for VM discovery
- `Microsoft.Network/networkInterfaces/read` : Required for VM discovery
- `Microsoft.Compute/virtualMachineScaleSets/virtualMachines/read` : Required for scale set (VMSS) discovery

The following meta labels are available on targets during relabeling:

- `__meta_azure_machine_id` : the machine ID
- `__meta_azure_machine_location` : the location the machine runs in
- `__meta_azure_machine_name` : the machine name
- `__meta_azure_machine_computer_name` : the machine computer name
- `__meta_azure_machine_os_type` : the machine operating system
- `__meta_azure_machine_private_ip` : the machine's private IP
- `__meta_azure_machine_public_ip` : the machine's public IP if it exists
- `__meta_azure_machine_resource_group` : the machine's resource group
- `__meta_azure_machine_tag_<tagname>` : each tag value of the machine
- `__meta_azure_machine_scale_set` : the name of the scale set which the vm is part of (this value is only set if you are using a scale set)
- `__meta_azure_machine_size` : the machine size
- `__meta_azure_subscription_id` : the subscription ID
- `__meta_azure_tenant_id` : the tenant ID

See below for the configuration options for Azure discovery:

```
# The information to access the Azure API.
# The Azure environment.
[ environment: <string> | default = AzurePublicCloud ]

# The authentication method, either OAuth, ManagedIdentity or SDK.
# See https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azu
# SDK authentication method uses environment variables by default.
# See https://learn.microsoft.com/en-us/azure/developer/go/azure-sdk-authentication
[ authentication_method: <string> | default = OAuth]
# The subscription ID. Always required.
subscription_id: <string>
# Optional tenant ID. Only required with authentication_method OAuth.
[ tenant_id: <string> ]
# Optional client ID. Only required with authentication_method OAuth.
[ client_id: <string> ]
# Optional client secret. Only required with authentication_method OAuth.
[ client_secret: <secret> ]

# Optional resource group name. Limits discovery to this resource group.
[ resource_group: <string> ]
```

```
# The port to scrape metrics from. If using the public IP address, this must
# instead be specified in the relabeling rule.
[ port: <int> | default = 80 ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

`<consul_sd_config>`

Consul SD configurations allow retrieving scrape targets from Consul's Catalog API.

The following meta labels are available on targets during relabeling:

- `__meta_consul_address` : the address of the target
- `__meta_consul_dc` : the datacenter name for the target
- `__meta_consul_health` : the health status of the service
- `__meta_consul_partition` : the admin partition name where the service is registered
- `__meta_consul_metadata_<key>` : each node metadata key value of the target
- `__meta_consul_node` : the node name defined for the target
- `__meta_consul_service_address` : the service address of the target
- `__meta_consul_service_id` : the service ID of the target
- `__meta_consul_service_metadata_<key>` : each service metadata key value of the target
- `__meta_consul_service_port` : the service port of the target
- `__meta_consul_service` : the name of the service the target belongs to
- `__meta_consul_tagged_address_<key>` : each node tagged address key value of the target
- `__meta_consul_tags` : the list of tags of the target joined by the tag separator

```
# The information to access the Consul API. It is to be defined
# as the Consul documentation requires.
[ server: <host> | default = "localhost:8500" ]
# Prefix for URIs for when consul is behind an API gateway (reverse proxy).
[ path_prefix: <string> ]
[ token: <secret> ]
[ datacenter: <string> ]
# Namespaces are only supported in Consul Enterprise.
```

Prometheus

```
[ scheme: <string> | default = "http" ]
# The username and password fields are deprecated in favor of the basic_auth config
[ username: <string> ]
[ password: <secret> ]

# A list of services for which targets are retrieved. If omitted, all services
# are scraped.
services:
  [ - <string> ]

# A Consul Filter expression used to filter the catalog results
# See https://www.consul.io/api-docs/catalog#list-services to know more
# about the filter expressions that can be used.
[ filter: <string> ]

# The `tags` and `node_meta` fields are deprecated in Consul in favor of `filter`.
# An optional list of tags used to filter nodes for a given service. Services must 
tags:
  [ - <string> ]

# Node metadata key/value pairs to filter nodes for a given service. As of Consul 1
[ node_meta:
  [ <string>: <string> ... ] ]

# The string by which Consul tags are joined into the tag label.
[ tag_separator: <string> | default = , ]

# Allow stale Consul results (see https://www.consul.io/api/features/consistency.htm
[ allow_stale: <boolean> | default = true ]

# The time after which the provided names are refreshed.
# On large setup it might be a good idea to increase this value because the catalog
[ refresh_interval: <duration> | default = 30s ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

Note that the IP number and port used to scrape the targets is assembled as `<__meta_consul_address>:<__meta_consul_service_port>`. However, in some Consul

The relabeling phase is the preferred and more powerful way to filter services or nodes for a service based on arbitrary labels. For users with thousands of services it can be more efficient to use the Consul API directly which has basic support for filtering nodes (currently by node metadata and a single tag).

`<digitalocean_sd_config>`

DigitalOcean SD configurations allow retrieving scrape targets from DigitalOcean's Droplets API. This service discovery uses the public IPv4 address by default, by that can be changed with relabeling, as demonstrated in the Prometheus digitalocean-sd configuration file.

The following meta labels are available on targets during relabeling:

- `__meta_digitalocean_droplet_id` : the id of the droplet
- `__meta_digitalocean_droplet_name` : the name of the droplet
- `__meta_digitalocean_image` : the slug of the droplet's image
- `__meta_digitalocean_image_name` : the display name of the droplet's image
- `__meta_digitalocean_private_ipv4` : the private IPv4 of the droplet
- `__meta_digitalocean_public_ipv4` : the public IPv4 of the droplet
- `__meta_digitalocean_public_ipv6` : the public IPv6 of the droplet
- `__meta_digitalocean_region` : the region of the droplet
- `__meta_digitalocean_size` : the size of the droplet
- `__meta_digitalocean_status` : the status of the droplet
- `__meta_digitalocean_features` : the comma-separated list of features of the droplet
- `__meta_digitalocean_tags` : the comma-separated list of tags of the droplet
- `__meta_digitalocean_vpc` : the id of the droplet's VPC

```
# The port to scrape metrics from.
[ port: <int> | default = 80 ]

# The time after which the droplets are refreshed.
[ refresh_interval: <duration> | default = 60s ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

This SD discovers "containers" and will create a target for each network IP and port the container is configured to expose.

Available meta labels:

- `__meta_docker_container_id` : the id of the container
- `__meta_docker_container_name` : the name of the container
- `__meta_docker_container_network_mode` : the network mode of the container
- `__meta_docker_container_label_<labelname>` : each label of the container, with any unsupported characters converted to an underscore
- `__meta_docker_network_id` : the ID of the network
- `__meta_docker_network_name` : the name of the network
- `__meta_docker_network_ingress` : whether the network is ingress
- `__meta_docker_network_internal` : whether the network is internal
- `__meta_docker_network_label_<labelname>` : each label of the network, with any unsupported characters converted to an underscore
- `__meta_docker_network_scope` : the scope of the network
- `__meta_docker_network_ip` : the IP of the container in this network
- `__meta_docker_port_private` : the port on the container
- `__meta_docker_port_public` : the external port if a port-mapping exists
- `__meta_docker_port_public_ip` : the public IP if a port-mapping exists

See below for the configuration options for Docker discovery:

```
# Address of the Docker daemon.
host: <string>

# The port to scrape metrics from, when `role` is nodes, and for discovered
# tasks and services that don't have published ports.
[ port: <int> | default = 80 ]

# The host to use if the container is in host networking mode.
[ host_networking_host: <string> | default = "localhost" ]

# Sort all non-nil networks in ascending order based on network name and
# get the first network if the container has multiple networks defined,
# thus avoiding collecting duplicate targets.
[ match_first_network: <boolean> | default = true ]
```

```
# https://docs.docker.com/engine/api/v1.40/#operation/ContainerList
[ filters:
  [ - name: <string>
      values: <string>, [...] ]

# The time after which the containers are refreshed.
[ refresh_interval: <duration> | default = 60s ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

The relabeling phase is the preferred and more powerful way to filter containers. For users with thousands of containers it can be more efficient to use the Docker API directly which has basic support for filtering containers (using `filters`).

See this example Prometheus configuration file for a detailed example of configuring Prometheus for Docker Engine.

`<dockerswarm_sd_config>`

Docker Swarm SD configurations allow retrieving scrape targets from Docker Swarm engine.

One of the following roles can be configured to discover targets:

`services`

The `services` role discovers all Swarm services and exposes their ports as targets. For each published port of a service, a single target is generated. If a service has no published ports, a target per service is created using the `port` parameter defined in the SD configuration.

Available meta labels:

- `__meta_dockerswarm_service_id`: the id of the service
- `__meta_dockerswarm_service_name`: the name of the service
- `__meta_dockerswarm_service_mode`: the mode of the service

endpoint port
- `__meta_dockerswarm_service_label_<labelname>` : each label of the service, with any unsupported characters converted to an underscore
- `__meta_dockerswarm_service_task_container_hostname` : the container hostname of the target, if available
- `__meta_dockerswarm_service_task_container_image` : the container image of the target
- `__meta_dockerswarm_service_updating_status` : the status of the service, if available
- `__meta_dockerswarm_network_id` : the ID of the network
- `__meta_dockerswarm_network_name` : the name of the network
- `__meta_dockerswarm_network_ingress` : whether the network is ingress
- `__meta_dockerswarm_network_internal` : whether the network is internal
- `__meta_dockerswarm_network_label_<labelname>` : each label of the network, with any unsupported characters converted to an underscore
- `__meta_dockerswarm_network_scope` : the scope of the network

`tasks`

The `tasks` role discovers all Swarm tasks and exposes their ports as targets. For each published port of a task, a single target is generated. If a task has no published ports, a target per task is created using the `port` parameter defined in the SD configuration.

Available meta labels:

- `__meta_dockerswarm_container_label_<labelname>` : each label of the container, with any unsupported characters converted to an underscore
- `__meta_dockerswarm_task_id` : the id of the task
- `__meta_dockerswarm_task_container_id` : the container id of the task
- `__meta_dockerswarm_task_desired_state` : the desired state of the task
- `__meta_dockerswarm_task_slot` : the slot of the task
- `__meta_dockerswarm_task_state` : the state of the task
- `__meta_dockerswarm_task_port_publish_mode` : the publish mode of the task port
- `__meta_dockerswarm_service_id` : the id of the service
- `__meta_dockerswarm_service_name` : the name of the service
- `__meta_dockerswarm_service_mode` : the mode of the service
- `__meta_dockerswarm_service_label_<labelname>` : each label of the service, with any unsupported characters converted to an underscore
- `__meta_dockerswarm_network_id` : the ID of the network
- `__meta_dockerswarm_network_name` : the name of the network

unsupported characters converted to an underscore
- `__meta_dockerswarm_network_label` : each label of the network, with any unsupported characters converted to an underscore
- `__meta_dockerswarm_network_scope` : the scope of the network
- `__meta_dockerswarm_node_id` : the ID of the node
- `__meta_dockerswarm_node_hostname` : the hostname of the node
- `__meta_dockerswarm_node_address` : the address of the node
- `__meta_dockerswarm_node_availability` : the availability of the node
- `__meta_dockerswarm_node_label_<labelname>` : each label of the node, with any unsupported characters converted to an underscore
- `__meta_dockerswarm_node_platform_architecture` : the architecture of the node
- `__meta_dockerswarm_node_platform_os` : the operating system of the node
- `__meta_dockerswarm_node_role` : the role of the node
- `__meta_dockerswarm_node_status` : the status of the node

The `__meta_dockerswarm_network_*` meta labels are not populated for ports which are published with `mode=host` .

nodes

The `nodes` role is used to discover Swarm nodes.

Available meta labels:

- `__meta_dockerswarm_node_address` : the address of the node
- `__meta_dockerswarm_node_availability` : the availability of the node
- `__meta_dockerswarm_node_engine_version` : the version of the node engine
- `__meta_dockerswarm_node_hostname` : the hostname of the node
- `__meta_dockerswarm_node_id` : the ID of the node
- `__meta_dockerswarm_node_label_<labelname>` : each label of the node, with any unsupported characters converted to an underscore
- `__meta_dockerswarm_node_manager_address` : the address of the manager component of the node
- `__meta_dockerswarm_node_manager_leader` : the leadership status of the manager component of the node (true or false)
- `__meta_dockerswarm_node_manager_reachability` : the reachability of the manager component of the node
- `__meta_dockerswarm_node_platform_architecture` : the architecture of the node
- `__meta_dockerswarm_node_platform_os` : the operating system of the node

See below for the configuration options for Docker Swarm discovery:

```
# Address of the Docker daemon.
host: <string>

# Role of the targets to retrieve. Must be `services`, `tasks`, or `nodes`.
role: <string>

# The port to scrape metrics from, when `role` is nodes, and for discovered
# tasks and services that don't have published ports.
[ port: <int> | default = 80 ]

# Optional filters to limit the discovery process to a subset of available
# resources.
# The available filters are listed in the upstream documentation:
# Services: https://docs.docker.com/engine/api/v1.40/#operation/ServiceList
# Tasks: https://docs.docker.com/engine/api/v1.40/#operation/TaskList
# Nodes: https://docs.docker.com/engine/api/v1.40/#operation/NodeList
[ filters:
  [ - name: <string>
      values: <string>, [...] ]

# The time after which the service discovery data is refreshed.
[ refresh_interval: <duration> | default = 60s ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

The relabeling phase is the preferred and more powerful way to filter tasks, services or nodes. For users with thousands of tasks it can be more efficient to use the Swarm API directly which has basic support for filtering nodes (using `filters`).

See this example Prometheus configuration file for a detailed example of configuring Prometheus for Docker Swarm.

`<dns_sd_config>`

This service discovery method only supports basic DNS A, AAAA, MX, NS and SRV record queries, but not the advanced DNS-SD approach specified in RFC6763.

The following meta labels are available on targets during relabeling:

- `__meta_dns_name` : the record name that produced the discovered target.
- `__meta_dns_srv_record_target` : the target field of the SRV record
- `__meta_dns_srv_record_port` : the port field of the SRV record
- `__meta_dns_mx_record_target` : the target field of the MX record
- `__meta_dns_ns_record_target` : the target field of the NS record

```
# A list of DNS domain names to be queried.
names:
  [ - <string> ]

# The type of DNS query to perform. One of SRV, A, AAAA, MX or NS.
[ type: <string> | default = 'SRV' ]

# The port number used if the query type is not SRV.
[ port: <int>]

# The time after which the provided names are refreshed.
[ refresh_interval: <duration> | default = 30s ]
```

`<ec2_sd_config>`

EC2 SD configurations allow retrieving scrape targets from AWS EC2 instances. The private IP address is used by default, but may be changed to the public IP address with relabeling.

The IAM credentials used must have the `ec2:DescribeInstances` permission to discover scrape targets, and may optionally have the `ec2:DescribeAvailabilityZones` permission if you want the availability zone ID available as a label (see below).

The following meta labels are available on targets during relabeling:

- `__meta_ec2_ami` : the EC2 Amazon Machine Image
- `__meta_ec2_architecture` : the architecture of the instance
- `__meta_ec2_availability_zone` : the availability zone in which the instance is running

- `__meta_ec2_instance_lifecycle` : the lifecycle of the EC2 instance, set only for 'spot' or 'scheduled' instances, absent otherwise
- `__meta_ec2_instance_state` : the state of the EC2 instance
- `__meta_ec2_instance_type` : the type of the EC2 instance
- `__meta_ec2_ipv6_addresses` : comma separated list of IPv6 addresses assigned to the instance's network interfaces, if present
- `__meta_ec2_owner_id` : the ID of the AWS account that owns the EC2 instance
- `__meta_ec2_platform` : the Operating System platform, set to 'windows' on Windows servers, absent otherwise
- `__meta_ec2_primary_ipv6_addresses` : comma separated list of the Primary IPv6 addresses of the instance, if present. The list is ordered based on the position of each corresponding network interface in the attachment order.
- `__meta_ec2_primary_subnet_id` : the subnet ID of the primary network interface, if available
- `__meta_ec2_private_dns_name` : the private DNS name of the instance, if available
- `__meta_ec2_private_ip` : the private IP address of the instance, if present
- `__meta_ec2_public_dns_name` : the public DNS name of the instance, if available
- `__meta_ec2_public_ip` : the public IP address of the instance, if available
- `__meta_ec2_region` : the region of the instance
- `__meta_ec2_subnet_id` : comma separated list of subnets IDs in which the instance is running, if available
- `__meta_ec2_tag_<tagkey>` : each tag value of the instance
- `__meta_ec2_vpc_id` : the ID of the VPC in which the instance is running, if available

See below for the configuration options for EC2 discovery:

```
# The information to access the EC2 API.

# The AWS region. If blank, the region from the instance metadata is used.
[ region: <string> ]

# Custom endpoint to be used.
[ endpoint: <string> ]

# The AWS API keys. If blank, the environment variables `AWS_ACCESS_KEY_ID`
# and `AWS_SECRET_ACCESS_KEY` are used.
[ access_key: <string> ]
[ secret_key: <secret> ]
# Named AWS profile used to connect to the API.
```

```
[ role_arn: <string> ]

# Refresh interval to re-read the instance list.
[ refresh_interval: <duration> | default = 60s ]

# The port to scrape metrics from. If using the public IP address, this must
# instead be specified in the relabeling rule.
[ port: <int> | default = 80 ]

# Filters can be used optionally to filter the instance list by other criteria.
# Available filter criteria can be found here:
# https://docs.aws.amazon.com/AWSEC2/latest/APIReference/API_DescribeInstances.html
# Filter API documentation: https://docs.aws.amazon.com/AWSEC2/latest/APIReference/A
filters:
  [ - name: <string>
      values: <string>, [...] ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

The relabeling phase is the preferred and more powerful way to filter targets based on arbitrary labels. For users with thousands of instances it can be more efficient to use the EC2 API directly which has support for filtering instances.

`<openstack_sd_config>`

OpenStack SD configurations allow retrieving scrape targets from OpenStack Nova instances.

One of the following `<openstack_role>` types can be configured to discover targets:

`hypervisor`

The `hypervisor` role discovers one target per Nova hypervisor node. The target address defaults to the `host_ip` attribute of the hypervisor.

The following meta labels are available on targets during relabeling:

- `__meta_openstack_hypervisor_host_ip`: the hypervisor node's IP address.

- `__meta_openstack_hypervisor_status` : the hypervisor node's status.
- `__meta_openstack_hypervisor_type` : the hypervisor node's type.

`instance`

The `instance` role discovers one target per network interface of Nova instance. The target address defaults to the private IP address of the network interface.

The following meta labels are available on targets during relabeling:

- `__meta_openstack_address_pool` : the pool of the private IP.
- `__meta_openstack_instance_flavor` : the flavor name of the OpenStack instance, or the flavor ID if the flavor name isn't available.
- `__meta_openstack_instance_id` : the OpenStack instance ID.
- `__meta_openstack_instance_image` : the ID of the image the OpenStack instance is using.
- `__meta_openstack_instance_name` : the OpenStack instance name.
- `__meta_openstack_instance_status` : the status of the OpenStack instance.
- `__meta_openstack_private_ip` : the private IP of the OpenStack instance.
- `__meta_openstack_project_id` : the project (tenant) owning this instance.
- `__meta_openstack_public_ip` : the public IP of the OpenStack instance.
- `__meta_openstack_tag_<key>` : each metadata item of the instance, with any unsupported characters converted to an underscore.
- `__meta_openstack_user_id` : the user account owning the tenant.

`loadbalancer`

The `loadbalancer` role discovers one target per Octavia loadbalancer with a `PROMETHEUS` listener. The target address defaults to the VIP address of the load balancer.

The following meta labels are available on targets during relabeling:

- `__meta_openstack_loadbalancer_availability_zone` : the availability zone of the OpenStack load balancer.
- `__meta_openstack_loadbalancer_floating_ip` : the floating IP of the OpenStack load balancer.
- `__meta_openstack_loadbalancer_id` : the OpenStack load balancer ID.
- `__meta_openstack_loadbalancer_name` : the OpenStack load balancer name.

OpenStack load balancer.

- `__meta_openstack_loadbalancer_provisioning_status` : the provisioning status of the OpenStack load balancer.
- `__meta_openstack_loadbalancer_tags` : comma separated list of the OpenStack load balancer.
- `__meta_openstack_loadbalancer_vip` : the VIP of the OpenStack load balancer.
- `__meta_openstack_project_id` : the project (tenant) owning this load balancer.

See below for the configuration options for OpenStack discovery:

```
# The information to access the OpenStack API.

# The OpenStack role of entities that should be discovered.
role: <openstack_role>

# The OpenStack Region.
region: <string>

# identity_endpoint specifies the HTTP endpoint that is required to work with
# the Identity API of the appropriate version. While it's ultimately needed by
# all of the identity services, it will often be populated by a provider-level
# function.
[ identity_endpoint: <string> ]

# username is required if using Identity V2 API. Consult with your provider's
# control panel to discover your account's username. In Identity V3, either
# userid or a combination of username and domain_id or domain_name are needed.
[ username: <string> ]
[ userid: <string> ]

# password for the Identity V2 and V3 APIs. Consult with your provider's
# control panel to discover your account's preferred method of authentication.
[ password: <secret> ]

# At most one of domain_id and domain_name must be provided if using username
# with Identity V3. Otherwise, either are optional.
[ domain_name: <string> ]
[ domain_id: <string> ]

# The project_id and project_name fields are optional for the Identity V2 API.
```

```
[ project_name: <string> ]
[ project_id: <string> ]

# The application_credential_id or application_credential_name fields are
# required if using an application credential to authenticate. Some providers
# allow you to create an application credential to authenticate rather than a
# password.
[ application_credential_name: <string> ]
[ application_credential_id: <string> ]

# The application_credential_secret field is required if using an application
# credential to authenticate.
[ application_credential_secret: <secret> ]

# Whether the service discovery should list all instances for all projects.
# It is only relevant for the 'instance' role and usually requires admin permission
[ all_tenants: <boolean> | default: false ]

# Refresh interval to re-read the instance list.
[ refresh_interval: <duration> | default = 60s ]

# The port to scrape metrics from. If using the public IP address, this must
# instead be specified in the relabeling rule.
[ port: <int> | default = 80 ]

# The availability of the endpoint to connect to. Must be one of public, admin or in
[ availability: <string> | default = "public" ]

# TLS configuration.
tls_config:
  [ <tls_config> ]
```

`<ovhcloud_sd_config>`

OVHcloud SD configurations allow retrieving scrape targets from OVHcloud's dedicated servers and VPS using their API. Prometheus will periodically check the REST endpoint and create a target for every discovered server. The role will try to use the public IPv4 address as default address, if there's none it will try to use the IPv6 one. This may be

VPS

- `__meta_ovhcloud_vps_cluster` : the cluster of the server
- `__meta_ovhcloud_vps_datacenter` : the datacenter of the server
- `__meta_ovhcloud_vps_disk` : the disk of the server
- `__meta_ovhcloud_vps_display_name` : the display name of the server
- `__meta_ovhcloud_vps_ipv4` : the IPv4 of the server
- `__meta_ovhcloud_vps_ipv6` : the IPv6 of the server
- `__meta_ovhcloud_vps_keymap` : the KVM keyboard layout of the server
- `__meta_ovhcloud_vps_maximum_additional_ip` : the maximum additional IPs of the server
- `__meta_ovhcloud_vps_memory_limit` : the memory limit of the server
- `__meta_ovhcloud_vps_memory` : the memory of the server
- `__meta_ovhcloud_vps_monitoring_ip_blocks` : the monitoring IP blocks of the server
- `__meta_ovhcloud_vps_name` : the name of the server
- `__meta_ovhcloud_vps_netboot_mode` : the netboot mode of the server
- `__meta_ovhcloud_vps_offer_type` : the offer type of the server
- `__meta_ovhcloud_vps_offer` : the offer of the server
- `__meta_ovhcloud_vps_state` : the state of the server
- `__meta_ovhcloud_vps_vcore` : the number of virtual cores of the server
- `__meta_ovhcloud_vps_version` : the version of the server
- `__meta_ovhcloud_vps_zone` : the zone of the server

Dedicated servers

- `__meta_ovhcloud_dedicated_server_commercial_range` : the commercial range of the server
- `__meta_ovhcloud_dedicated_server_datacenter` : the datacenter of the server
- `__meta_ovhcloud_dedicated_server_ipv4` : the IPv4 of the server
- `__meta_ovhcloud_dedicated_server_ipv6` : the IPv6 of the server
- `__meta_ovhcloud_dedicated_server_link_speed` : the link speed of the server
- `__meta_ovhcloud_dedicated_server_name` : the name of the server
- `__meta_ovhcloud_dedicated_server_no_intervention` : whether datacenter intervention is disabled for the server
- `__meta_ovhcloud_dedicated_server_os` : the operating system of the server
- `__meta_ovhcloud_dedicated_server_rack` : the rack of the server
- `__meta_ovhcloud_dedicated_server_reverse` : the reverse DNS name of the server

See below for the configuration options for OVHcloud discovery:

```
# Access key to use. https://api.ovh.com
application_key: <string>
application_secret: <secret>
consumer_key: <secret>
# Service of the targets to retrieve. Must be `vps` or `dedicated_server`.
service: <string>
# API endpoint. https://github.com/ovh/go-ovh#supported-apis
[ endpoint: <string> | default = "ovh-eu" ]
# Refresh interval to re-read the resources list.
[ refresh_interval: <duration> | default = 60s ]
```

<puppetdb_sd_config>

PuppetDB SD configurations allow retrieving scrape targets from PuppetDB resources.

This SD discovers resources and will create a target for each resource returned by the API.

The resource address is the `certname` of the resource and can be changed during relabeling.

The following meta labels are available on targets during relabeling:

- `__meta_puppetdb_query` : the Puppet Query Language (PQL) query
- `__meta_puppetdb_certname` : the name of the node associated with the resource
- `__meta_puppetdb_resource` : a SHA-1 hash of the resource's type, title, and parameters, for identification
- `__meta_puppetdb_type` : the resource type
- `__meta_puppetdb_title` : the resource title
- `__meta_puppetdb_exported` : whether the resource is exported (`"true"` or `"false"`)
- `__meta_puppetdb_tags` : comma separated list of resource tags
- `__meta_puppetdb_file` : the manifest file in which the resource was declared
- `__meta_puppetdb_environment` : the environment of the node associated with the resource
- `__meta_puppetdb_parameter_<parametername>` : the parameters of the resource

See below for the configuration options for PuppetDB discovery:

Prometheus

```
# Puppet Query Language (PQL) query. Only resources are supported.
# https://puppet.com/docs/puppetdb/latest/api/query/v4/pql.html
query: <string>

# Whether to include the parameters as meta labels.
# Due to the differences between parameter types and Prometheus labels,
# some parameters might not be rendered. The format of the parameters might
# also change in future releases.
#
# Note: Enabling this exposes parameters in the Prometheus UI and API. Make sure
# that you don't have secrets exposed as parameters if you enable this.
[ include_parameters: <boolean> | default = false ]

# Refresh interval to re-read the resources list.
[ refresh_interval: <duration> | default = 60s ]

# The port to scrape metrics from.
[ port: <int> | default = 80 ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

See this example Prometheus configuration file for a detailed example of configuring Prometheus with PuppetDB.

`<file_sd_config>`

File-based service discovery provides a more generic way to configure static targets and serves as an interface to plug in custom service discovery mechanisms.

It reads a set of files containing a list of zero or more `<static_config>` s. Changes to all defined files are detected via disk watches and applied immediately.

While those individual files are watched for changes, the parent directory is also watched implicitly. This is to handle atomic renaming efficiently and to detect new files that match the configured globs. This may cause issues if the parent directory contains a large number of other files, as each of these files will be watched too, even though the events related to them are not relevant.

Files must contain a list of static configs, using these formats:

JSON

```
[
  {
    "targets": [ "<host>", ... ],
    "labels": {
      "<labelname>": "<labelvalue>", ...
    }
  },
  ...
]
```

YAML

```
- targets:
  [ - '<host>' ]
  labels:
    [ <labelname>: <labelvalue> ... ]
```

As a fallback, the file contents are also re-read periodically at the specified refresh interval.

Each target has a meta label `__meta_filepath` during the relabeling phase. Its value is set to the filepath from which the target was extracted.

There is a list of integrations with this discovery mechanism.

```
# Patterns for files from which target groups are extracted.
files:
  [ - <filename_pattern> ... ]

# Refresh interval to re-read the files.
[ refresh_interval: <duration> | default = 5m ]
```

Where `<filename_pattern>` may be a path ending in `.json`, `.yml` or `.yaml`. The last path segment may contain a single `*` that matches any character sequence, e.g. `my/path/tg_*.json`.

IP address is used by default, but may be changed to the public IP address with relabeling.

The following meta labels are available on targets during relabeling:

- `__meta_gce_instance_id` : the numeric id of the instance
- `__meta_gce_instance_name` : the name of the instance
- `__meta_gce_label_<labelname>` : each GCE label of the instance, with any unsupported characters converted to an underscore
- `__meta_gce_machine_type` : full or partial URL of the machine type of the instance
- `__meta_gce_metadata_<name>` : each metadata item of the instance
- `__meta_gce_network` : the network URL of the instance
- `__meta_gce_private_ip` : the private IP address of the instance
- `__meta_gce_interface_ipv4_<name>` : IPv4 address of each named interface
- `__meta_gce_project` : the GCP project in which the instance is running
- `__meta_gce_public_ip` : the public IP address of the instance, if present
- `__meta_gce_subnetwork` : the subnetwork URL of the instance
- `__meta_gce_tags` : comma separated list of instance tags
- `__meta_gce_zone` : the GCE zone URL in which the instance is running

See below for the configuration options for GCE discovery:

```
# The information to access the GCE API.

# The GCP Project
project: <string>

# The zone of the scrape targets. If you need multiple zones use multiple
# gce_sd_configs.
zone: <string>

# Filter can be used optionally to filter the instance list by other criteria
# Syntax of this filter string is described here in the filter query parameter sect:
# https://cloud.google.com/compute/docs/reference/latest/instances/list
[ filter: <string> ]

# Refresh interval to re-read the instance list
[ refresh_interval: <duration> | default = 60s ]

# The port to scrape metrics from. If using the public IP address, this must
# instead be specified in the relabeling rule.
```

```
  [ tag_separator: <string> | default = , ]
```

Credentials are discovered by the Google Cloud SDK default client by looking in the following places, preferring the first location found:

1. a JSON file specified by the `GOOGLE_APPLICATION_CREDENTIALS` environment variable
2. a JSON file in the well-known path `$HOME/.config/gcloud/application_default_credentials.json`
3. fetched from the GCE metadata server

If Prometheus is running within GCE, the service account associated with the instance it is running on should have at least read-only permissions to the compute resources. If running outside of GCE make sure to create an appropriate service account and place the credential file in one of the expected locations.

`<hetzner_sd_config>`

Hetzner SD configurations allow retrieving scrape targets from Hetzner Cloud API and Robot API. This service discovery uses the public IPv4 address by default, but that can be changed with relabeling, as demonstrated in the Prometheus hetzner-sd configuration file .

The following meta labels are available on all targets during relabeling:

- `__meta_hetzner_server_id` : the ID of the server
- `__meta_hetzner_server_name` : the name of the server
- `__meta_hetzner_server_status` : the status of the server
- `__meta_hetzner_public_ipv4` : the public ipv4 address of the server
- `__meta_hetzner_public_ipv6_network` : the public ipv6 network (/64) of the server
- `__meta_hetzner_datacenter` : the datacenter of the server

The labels below are only available for targets with `role` set to `hcloud` :

- `__meta_hetzner_hcloud_image_name` : the image name of the server
- `__meta_hetzner_hcloud_image_description` : the description of the server image
- `__meta_hetzner_hcloud_image_os_flavor` : the OS flavor of the server image
- `__meta_hetzner_hcloud_image_os_version` : the OS version of the server image
- `__meta_hetzner_hcloud_datacenter_location` : the location of the server

- `__meta_hetzner_hcloud_cpu_cores` : the CPU cores count of the server
- `__meta_hetzner_hcloud_cpu_type` : the CPU type of the server (shared or dedicated)
- `__meta_hetzner_hcloud_memory_size_gb` : the amount of memory of the server (in GB)
- `__meta_hetzner_hcloud_disk_size_gb` : the disk size of the server (in GB)
- `__meta_hetzner_hcloud_private_ipv4_<networkname>` : the private ipv4 address of the server within a given network
- `__meta_hetzner_hcloud_label_<labelname>` : each label of the server, with any unsupported characters converted to an underscore
- `__meta_hetzner_hcloud_labelpresent_<labelname>` : `true` for each label of the server, with any unsupported characters converted to an underscore

The labels below are only available for targets with `role` set to `robot` :

- `__meta_hetzner_robot_product` : the product of the server
- `__meta_hetzner_robot_cancelled` : the server cancellation status

```
# The Hetzner role of entities that should be discovered.
# One of robot or hcloud.
role: <string>

# The port to scrape metrics from.
[ port: <int> | default = 80 ]

# The time after which the servers are refreshed.
[ refresh_interval: <duration> | default = 60s ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

`<http_sd_config>`

HTTP-based service discovery provides a more generic way to configure static targets and serves as an interface to plug in custom service discovery mechanisms.

It fetches targets from an HTTP endpoint containing a list of zero or more `<static_config>` s. The target must reply with an HTTP 200 response. The HTTP header `Content-Type` must be `application/json` , and the body must be valid JSON.

```
  {
    "targets": [ "<host>", ... ],
    "labels": {
      "<labelname>": "<labelvalue>", ...
    }
  },
  ...
]
```

The endpoint is queried periodically at the specified refresh interval. The `prometheus_sd_http_failures_total` counter metric tracks the number of refresh failures.

Each target has a meta label `__meta_url` during the relabeling phase. Its value is set to the URL from which the target was extracted.

```
# URL from which the targets are fetched.
url: <string>

# Refresh interval to re-query the endpoint.
[ refresh_interval: <duration> | default = 60s ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

`<ionos_sd_config>`

IONOS SD configurations allows retrieving scrape targets from IONOS Cloud API. This service discovery uses the first NICs IP address by default, but that can be changed with relabeling. The following meta labels are available on all targets during relabeling:

- `__meta_ionos_server_availability_zone` : the availability zone of the server
- `__meta_ionos_server_boot_cdrom_id` : the ID of the CD-ROM the server is booted from
- `__meta_ionos_server_boot_image_id` : the ID of the boot image or snapshot the server is booted from
- `__meta_ionos_server_boot_volume_id` : the ID of the boot volume
- `__meta_ionos_server_cpu_family` : the CPU family of the server to

- `__meta_ionos_server_name` : the name of the server
- `__meta_ionos_server_nic_ip_<nic_name>` : comma separated list of IPs, grouped by the name of each NIC attached to the server
- `__meta_ionos_server_servers_id` : the ID of the servers the server belongs to
- `__meta_ionos_server_state` : the execution state of the server
- `__meta_ionos_server_type` : the type of the server

```
# The unique ID of the data center.
datacenter_id: <string>

# The port to scrape metrics from.
[ port: <int> | default = 80 ]

# The time after which the servers are refreshed.
[ refresh_interval: <duration> | default = 60s ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

`<kubernetes_sd_config>`

Kubernetes SD configurations allow retrieving scrape targets from Kubernetes' REST API and always staying synchronized with the cluster state.

One of the following `role` types can be configured to discover targets:

node

The `node` role discovers one target per cluster node with the address defaulting to the Kubelet's HTTP port. The target address defaults to the first existing address of the Kubernetes node object in the address type order of `NodeInternalIP` , `NodeExternalIP` , `NodeLegacyHostIP` , and `NodeHostName` .

Available meta labels:

- `__meta_kubernetes_node_name` : The name of the node object.

any unsupported characters converted to an underscore.
- `__meta_kubernetes_node_labelpresent_<labelname>` : `true` for each label from the node object, with any unsupported characters converted to an underscore.
- `__meta_kubernetes_node_annotation_<annotationname>` : Each annotation from the node object.
- `__meta_kubernetes_node_annotationpresent_<annotationname>` : `true` for each annotation from the node object.
- `__meta_kubernetes_node_address_<address_type>` : The first address for each node address type, if it exists.

In addition, the `instance` label for the node will be set to the node name as retrieved from the API server.

`service`

The `service` role discovers a target for each service port for each service. This is generally useful for blackbox monitoring of a service. The address will be set to the Kubernetes DNS name of the service and respective service port.

Available meta labels:

- `__meta_kubernetes_namespace` : The namespace of the service object.
- `__meta_kubernetes_service_annotation_<annotationname>` : Each annotation from the service object.
- `__meta_kubernetes_service_annotationpresent_<annotationname>` : "true" for each annotation of the service object.
- `__meta_kubernetes_service_cluster_ip` : The cluster IP address of the service. (Does not apply to services of type ExternalName)
- `__meta_kubernetes_service_loadbalancer_ip` : The IP address of the loadbalancer. (Applies to services of type LoadBalancer)
- `__meta_kubernetes_service_external_name` : The DNS name of the service. (Applies to services of type ExternalName)
- `__meta_kubernetes_service_label_<labelname>` : Each label from the service object, with any unsupported characters converted to an underscore.
- `__meta_kubernetes_service_labelpresent_<labelname>` : `true` for each label of the service object, with any unsupported characters converted to an underscore.
- `__meta_kubernetes_service_name` : The name of the service object.
- `__meta_kubernetes_service_port_name` : Name of the service port for the target.
- `__meta_kubernetes_service_port_number` : Number of the service port for the target.

Prometheus

pod

The `pod` role discovers all pods and exposes their containers as targets. For each declared port of a container, a single target is generated. If a container has no specified ports, a port-free target per container is created for manually adding a port via relabeling.

Available meta labels:

- `__meta_kubernetes_namespace` : The namespace of the pod object.
- `__meta_kubernetes_pod_name` : The name of the pod object.
- `__meta_kubernetes_pod_ip` : The pod IP of the pod object.
- `__meta_kubernetes_pod_label_<labelname>` : Each label from the pod object, with any unsupported characters converted to an underscore.
- `__meta_kubernetes_pod_labelpresent_<labelname>` : `true` for each label from the pod object, with any unsupported characters converted to an underscore.
- `__meta_kubernetes_pod_annotation_<annotationname>` : Each annotation from the pod object.
- `__meta_kubernetes_pod_annotationpresent_<annotationname>` : `true` for each annotation from the pod object.
- `__meta_kubernetes_pod_container_init` : `true` if the container is an InitContainer
- `__meta_kubernetes_pod_container_name` : Name of the container the target address points to.
- `__meta_kubernetes_pod_container_id` : ID of the container the target address points to. The ID is in the form `<type>://<container_id>` .
- `__meta_kubernetes_pod_container_image` : The image the container is using.
- `__meta_kubernetes_pod_container_port_name` : Name of the container port.
- `__meta_kubernetes_pod_container_port_number` : Number of the container port.
- `__meta_kubernetes_pod_container_port_protocol` : Protocol of the container port.
- `__meta_kubernetes_pod_ready` : Set to `true` or `false` for the pod's ready state.
- `__meta_kubernetes_pod_phase` : Set to `Pending` , `Running` , `Succeeded` , `Failed` or `Unknown` in the lifecycle .
- `__meta_kubernetes_pod_node_name` : The name of the node the pod is scheduled onto.
- `__meta_kubernetes_pod_host_ip` : The current host IP of the pod object.
- `__meta_kubernetes_pod_uid` : The UID of the pod object.
- `__meta_kubernetes_pod_controller_kind` : Object kind of the pod controller.
- `__meta_kubernetes_pod_controller_name` : Name of the pod controller.

address one target is discovered per port. If the endpoint is backed by a pod, all additional container ports of the pod, not bound to an endpoint port, are discovered as targets as well.

Available meta labels:

- `__meta_kubernetes_namespace` : The namespace of the endpoints object.
- `__meta_kubernetes_endpoints_name` : The names of the endpoints object.
- `__meta_kubernetes_endpoints_label_<labelname>` : Each label from the endpoints object, with any unsupported characters converted to an underscore.
- `__meta_kubernetes_endpoints_labelpresent_<labelname>` : `true` for each label from the endpoints object, with any unsupported characters converted to an underscore.
- `__meta_kubernetes_endpoints_annotation_<annotationname>` : Each annotation from the endpoints object.
- `__meta_kubernetes_endpoints_annotationpresent_<annotationname>` : `true` for each annotation from the endpoints object.
- For all targets discovered directly from the endpoints list (those not additionally inferred from underlying pods), the following labels are attached:
    - `__meta_kubernetes_endpoint_hostname` : Hostname of the endpoint.
    - `__meta_kubernetes_endpoint_node_name` : Name of the node hosting the endpoint.
    - `__meta_kubernetes_endpoint_ready` : Set to `true` or `false` for the endpoint's ready state.
    - `__meta_kubernetes_endpoint_port_name` : Name of the endpoint port.
    - `__meta_kubernetes_endpoint_port_protocol` : Protocol of the endpoint port.
    - `__meta_kubernetes_endpoint_address_target_kind` : Kind of the endpoint address target.
    - `__meta_kubernetes_endpoint_address_target_name` : Name of the endpoint address target.
- If the endpoints belong to a service, all labels of the `role: service` discovery are attached.
- For all targets backed by a pod, all labels of the `role: pod` discovery are attached.

`endpointslice`

The `endpointslice` role discovers targets from existing endpointslices. For each endpoint address referenced in the endpointslice object one target is discovered. If the endpoint is backed by a pod, all additional container ports of the pod, not bound to an endpoint port, are discovered as targets as well.

- `__meta_kubernetes_namespace` : The namespace of the endpoints object.
- `__meta_kubernetes_endpointslice_name` : The name of endpointslice object.
- `__meta_kubernetes_endpointslice_label_<labelname>` : Each label from the endpointslice object, with any unsupported characters converted to an underscore.
- `__meta_kubernetes_endpointslice_labelpresent_<labelname>` : `true` for each label from the endpointslice object, with any unsupported characters converted to an underscore.
- `__meta_kubernetes_endpointslice_annotation_<annotationname>` : Each annotation from the endpointslice object.
- `__meta_kubernetes_endpointslice_annotationpresent_<annotationname>` : `true` for each annotation from the endpointslice object.
- For all targets discovered directly from the endpointslice list (those not additionally inferred from underlying pods), the following labels are attached:
  - `__meta_kubernetes_endpointslice_address_target_kind` : Kind of the referenced object.
  - `__meta_kubernetes_endpointslice_address_target_name` : Name of referenced object.
  - `__meta_kubernetes_endpointslice_address_type` : The ip protocol family of the address of the target.
  - `__meta_kubernetes_endpointslice_endpoint_conditions_ready` : Set to `true` or `false` for the referenced endpoint's ready state.
  - `__meta_kubernetes_endpointslice_endpoint_conditions_serving` : Set to `true` or `false` for the referenced endpoint's serving state.
  - `__meta_kubernetes_endpointslice_endpoint_conditions_terminating` : Set to `true` or `false` for the referenced endpoint's terminating state.
  - `__meta_kubernetes_endpointslice_endpoint_topology_kubernetes_io_hostname` : Name of the node hosting the referenced endpoint.
  - `__meta_kubernetes_endpointslice_endpoint_topology_present_kubernetes_io_hostname` : Flag that shows if the referenced object has a kubernetes.io/hostname annotation.
  - `__meta_kubernetes_endpointslice_endpoint_hostname` : Hostname of the referenced endpoint.
  - `__meta_kubernetes_endpointslice_endpoint_node_name` : Name of the Node hosting the referenced endpoint.
  - `__meta_kubernetes_endpointslice_endpoint_zone` : Zone the referenced endpoint exists in.
  - `__meta_kubernetes_endpointslice_port` : Port of the referenced endpoint.

endpoint.

- If the endpoints belong to a service, all labels of the `role: service` discovery are attached.
- For all targets backed by a pod, all labels of the `role: pod` discovery are attached.

`ingress`

The `ingress` role discovers a target for each path of each ingress. This is generally useful for blackbox monitoring of an ingress. The address will be set to the host specified in the ingress spec.

The role requires the `networking.k8s.io/v1` API version (available since Kubernetes v1.19).

Available meta labels:

- `__meta_kubernetes_namespace` : The namespace of the ingress object.
- `__meta_kubernetes_ingress_name` : The name of the ingress object.
- `__meta_kubernetes_ingress_label_<labelname>` : Each label from the ingress object, with any unsupported characters converted to an underscore.
- `__meta_kubernetes_ingress_labelpresent_<labelname>` : `true` for each label from the ingress object, with any unsupported characters converted to an underscore.
- `__meta_kubernetes_ingress_annotation_<annotationname>` : Each annotation from the ingress object.
- `__meta_kubernetes_ingress_annotationpresent_<annotationname>` : `true` for each annotation from the ingress object.
- `__meta_kubernetes_ingress_class_name` : Class name from ingress spec, if present.
- `__meta_kubernetes_ingress_scheme` : Protocol scheme of ingress, `https` if TLS config is set. Defaults to `http` .
- `__meta_kubernetes_ingress_path` : Path from ingress spec. Defaults to `/` .

See below for the configuration options for Kubernetes discovery:

```
# The information to access the Kubernetes API.

# The API server addresses. If left empty, Prometheus is assumed to run inside
# of the cluster and will discover API servers automatically and use the pod's
# CA certificate and bearer token file at /var/run/secrets/kubernetes.io/serviceacc
[ api_server: <host> ]

# The Kubernetes role of entities that should be discovered.
```

Prometheus

```
# Optional path to a kubeconfig file.
# Note that api_server and kube_config are mutually exclusive.
[ kubeconfig_file: <filename> ]

# Optional namespace discovery. If omitted, all namespaces are used.
namespaces:
  own_namespace: <boolean>
  names:
    [ - <string> ]

# Optional label and field selectors to limit the discovery process to a subset of a
# See https://kubernetes.io/docs/concepts/overview/working-with-objects/field-select
# and https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/ to l
# filters that can be used. The endpoints role supports pod, service and endpoints s
# The pod role supports node selectors when configured with `attach_metadata: {node
# Other roles only support selectors matching the role itself (e.g. node role can or

# Note: When making decision about using field/label selector make sure that this
# is the best approach - it will prevent Prometheus from reusing single list/watch
# for all scrape configs. This might result in a bigger load on the Kubernetes API,
# because per each selector combination there will be additional LIST/WATCH. On the
# if you just want to monitor small subset of pods in large cluster it's recommended
# Decision, if selectors should be used or not depends on the particular situation.
[ selectors:
  [ - role: <string>
    [ label: <string> ]
    [ field: <string> ] ]]

# Optional metadata to attach to discovered targets. If omitted, no additional metac
attach_metadata:
# Attaches node metadata to discovered targets. Valid for roles: pod, endpoints, enc
# When set to true, Prometheus must have permissions to get Nodes.
  [ node: <boolean> | default = false ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

See this example Prometheus configuration file for a detailed example of configuring Prometheus for Kubernetes.

Prometheus

`<kuma_sd_config>`

Kuma SD configurations allow retrieving scrape target from the Kuma control plane.

This SD discovers "monitoring assignments" based on Kuma Dataplane Proxies, via the MADS v1 (Monitoring Assignment Discovery Service) xDS API, and will create a target for each proxy inside a Prometheus-enabled mesh.

The following meta labels are available for each target:

- `__meta_kuma_mesh` : the name of the proxy's Mesh
- `__meta_kuma_dataplane` : the name of the proxy
- `__meta_kuma_service` : the name of the proxy's associated Service
- `__meta_kuma_label_<tagname>` : each tag of the proxy

See below for the configuration options for Kuma MonitoringAssignment discovery:

```
# Address of the Kuma Control Plane's MADS xDS server.
server: <string>

# Client id is used by Kuma Control Plane to compute Monitoring Assignment for spec
# This is useful when migrating between multiple Prometheus backends, or having sepa
# When not specified, system hostname/fqdn will be used if available, if not `promet
[ client_id: <string> ]

# The time to wait between polling update requests.
[ refresh_interval: <duration> | default = 30s ]

# The time after which the monitoring assignments are refreshed.
[ fetch_timeout: <duration> | default = 2m ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

The relabeling phase is the preferred and more powerful way to filter proxies and user-defined tags.

The private IP address is used by default, but may be changed to the public IP address with relabeling.

The following meta labels are available on targets during relabeling:

- `__meta_lightsail_availability_zone` : the availability zone in which the instance is running
- `__meta_lightsail_blueprint_id` : the Lightsail blueprint ID
- `__meta_lightsail_bundle_id` : the Lightsail bundle ID
- `__meta_lightsail_instance_name` : the name of the Lightsail instance
- `__meta_lightsail_instance_state` : the state of the Lightsail instance
- `__meta_lightsail_instance_support_code` : the support code of the Lightsail instance
- `__meta_lightsail_ipv6_addresses` : comma separated list of IPv6 addresses assigned to the instance's network interfaces, if present
- `__meta_lightsail_private_ip` : the private IP address of the instance
- `__meta_lightsail_public_ip` : the public IP address of the instance, if available
- `__meta_lightsail_region` : the region of the instance
- `__meta_lightsail_tag_<tagkey>` : each tag value of the instance

See below for the configuration options for Lightsail discovery:

```
# The information to access the Lightsail API.

# The AWS region. If blank, the region from the instance metadata is used.
[ region: <string> ]

# Custom endpoint to be used.
[ endpoint: <string> ]

# The AWS API keys. If blank, the environment variables `AWS_ACCESS_KEY_ID`
# and `AWS_SECRET_ACCESS_KEY` are used.
[ access_key: <string> ]
[ secret_key: <secret> ]
# Named AWS profile used to connect to the API.
[ profile: <string> ]

# AWS Role ARN, an alternative to using AWS API keys.
[ role_arn: <string> ]

# Refresh interval to re-read the instance list.
# instead be specified in the relabeling rule.
[ port: <int> | default = 80 ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

<linode_sd_config>

Linode SD configurations allow retrieving scrape targets from Linode's Linode APIv4. This service discovery uses the public IPv4 address by default, by that can be changed with relabeling, as demonstrated in the Prometheus linode-sd configuration file.

Linode APIv4 Token must be created with scopes: `linodes:read_only`, `ips:read_only`, and `events:read_only`.

The following meta labels are available on targets during relabeling:

- `__meta_linode_instance_id`: the id of the linode instance
- `__meta_linode_instance_label`: the label of the linode instance
- `__meta_linode_image`: the slug of the linode instance's image
- `__meta_linode_private_ipv4`: the private IPv4 of the linode instance
- `__meta_linode_public_ipv4`: the public IPv4 of the linode instance
- `__meta_linode_public_ipv6`: the public IPv6 of the linode instance
- `__meta_linode_private_ipv4_rdns`: the reverse DNS for the first private IPv4 of the linode instance
- `__meta_linode_public_ipv4_rdns`: the reverse DNS for the first public IPv4 of the linode instance
- `__meta_linode_public_ipv6_rdns`: the reverse DNS for the first public IPv6 of the linode instance
- `__meta_linode_region`: the region of the linode instance
- `__meta_linode_type`: the type of the linode instance
- `__meta_linode_status`: the status of the linode instance
- `__meta_linode_tags`: a list of tags of the linode instance joined by the tag separator
- `__meta_linode_group`: the display group a linode instance is a member of
- `__meta_linode_gpus`: the number of GPU's of the linode instance
- `__meta_linode_hypervisor`: the virtualization software powering the linode instance
- `__meta_linode_backups`: the backup service status of the linode instance

access to
- `__meta_linode_specs_vcpus` : the number of VCPUS this linode has access to
- `__meta_linode_specs_transfer_bytes` : the amount of network transfer the linode instance is allotted each month
- `__meta_linode_extra_ips` : a list of all extra IPv4 addresses assigned to the linode instance joined by the tag separator
- `__meta_linode_ipv6_ranges` : a list of IPv6 ranges with mask assigned to the linode instance joined by the tag separator

```
# Optional region to filter on.
[ region: <string> ]

# The port to scrape metrics from.
[ port: <int> | default = 80 ]

# The string by which Linode Instance tags are joined into the tag label.
[ tag_separator: <string> | default = , ]

# The time after which the linode instances are refreshed.
[ refresh_interval: <duration> | default = 60s ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

`<marathon_sd_config>`

Marathon SD configurations allow retrieving scrape targets using the Marathon REST API. Prometheus will periodically check the REST endpoint for currently running tasks and create a target group for every app that has at least one healthy task.

The following meta labels are available on targets during relabeling:

- `__meta_marathon_app` : the name of the app (with slashes replaced by dashes)
- `__meta_marathon_image` : the name of the Docker image used (if available)
- `__meta_marathon_task` : the ID of the Mesos task
- `__meta_marathon_app_label_<labelname>` : any Marathon labels attached to the app, with any unsupported characters converted to an underscore

any unsupported characters converted to an underscore
- `__meta_marathon_port_index` : the port index number (e.g. `1` for `PORT1`)

See below for the configuration options for Marathon discovery:

```
# List of URLs to be used to contact Marathon servers.
# You need to provide at least one server URL.
servers:
  - <string>

# Polling interval
[ refresh_interval: <duration> | default = 30s ]

# Optional authentication information for token-based authentication
# https://docs.mesosphere.com/1.11/security/ent/iam-api/#passing-an-authentication-t
# It is mutually exclusive with `auth_token_file` and other authentication mechanisr
[ auth_token: <secret> ]

# Optional authentication information for token-based authentication
# https://docs.mesosphere.com/1.11/security/ent/iam-api/#passing-an-authentication-t
# It is mutually exclusive with `auth_token` and other authentication mechanisms.
[ auth_token_file: <filename> ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

By default every app listed in Marathon will be scraped by Prometheus. If not all of your services provide Prometheus metrics, you can use a Marathon label and Prometheus relabeling to control which instances will actually be scraped. See the Prometheus marathon-sd configuration file for a practical example on how to set up your Marathon app and your Prometheus configuration.

By default, all apps will show up as a single job in Prometheus (the one specified in the configuration file), which can also be changed using relabeling.

`<nerve_sd_config>`

The following meta labels are available on targets during relabeling:

- `__meta_nerve_path` : the full path to the endpoint node in Zookeeper
- `__meta_nerve_endpoint_host` : the host of the endpoint
- `__meta_nerve_endpoint_port` : the port of the endpoint
- `__meta_nerve_endpoint_name` : the name of the endpoint

```
# The Zookeeper servers.
servers:
  - <host>
# Paths can point to a single service, or the root of a tree of services.
paths:
  - <string>
[ timeout: <duration> | default = 10s ]
```

`<nomad_sd_config>`

Nomad SD configurations allow retrieving scrape targets from Nomad's Service API.

The following meta labels are available on targets during relabeling:

- `__meta_nomad_address` : the service address of the target
- `__meta_nomad_dc` : the datacenter name for the target
- `__meta_nomad_namespace` : the namespace of the target
- `__meta_nomad_node_id` : the node name defined for the target
- `__meta_nomad_service` : the name of the service the target belongs to
- `__meta_nomad_service_address` : the service address of the target
- `__meta_nomad_service_id` : the service ID of the target
- `__meta_nomad_service_port` : the service port of the target
- `__meta_nomad_tags` : the list of tags of the target joined by the tag separator

```
# The information to access the Nomad API. It is to be defined
# as the Nomad documentation requires.
[ allow_stale: <boolean> | default = true ]
[ namespace: <string> | default = default ]
[ refresh_interval: <duration> | default = 60s ]
[ region: <string> | default = global ]
# The URL to connect to the API.
[ server: <string> ]
```

```
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

`<serverset_sd_config>`

Serverset SD configurations allow retrieving scrape targets from [Serversets] (https://github.com/twitter/finagle/tree/develop/finagle-serversets) which are stored in Zookeeper. Serversets are commonly used by Finagle and Aurora.

The following meta labels are available on targets during relabeling:

- `__meta_serverset_path` : the full path to the serverset member node in Zookeeper
- `__meta_serverset_endpoint_host` : the host of the default endpoint
- `__meta_serverset_endpoint_port` : the port of the default endpoint
- `__meta_serverset_endpoint_host_<endpoint>` : the host of the given endpoint
- `__meta_serverset_endpoint_port_<endpoint>` : the port of the given endpoint
- `__meta_serverset_shard` : the shard number of the member
- `__meta_serverset_status` : the status of the member

```
# The Zookeeper servers.
servers:
  - <host>
# Paths can point to a single serverset, or the root of a tree of serversets.
paths:
  - <string>
[ timeout: <duration> | default = 10s ]
```

Serverset data must be in the JSON format, the Thrift format is not currently supported.

`<triton_sd_config>`

Triton SD configurations allow retrieving scrape targets from Container Monitor discovery endpoints.

One of the following `<triton_role>` types can be configured to discover targets:

`container`

The following meta labels are available on targets during relabeling:

- `__meta_triton_groups` : the list of groups belonging to the target joined by a comma separator
- `__meta_triton_machine_alias` : the alias of the target container
- `__meta_triton_machine_brand` : the brand of the target container
- `__meta_triton_machine_id` : the UUID of the target container
- `__meta_triton_machine_image` : the target container's image type
- `__meta_triton_server_id` : the server UUID the target container is running on

`cn`

The `cn` role discovers one target for per compute node (also known as "server" or "global zone") making up the Triton infrastructure. The `account` must be a Triton operator and is currently required to own at least one `container` .

The following meta labels are available on targets during relabeling:

- `__meta_triton_machine_alias` : the hostname of the target (requires triton-cmon 1.7.0 or newer)
- `__meta_triton_machine_id` : the UUID of the target

See below for the configuration options for Triton discovery:

```
# The information to access the Triton discovery API.

# The account to use for discovering new targets.
account: <string>

# The type of targets to discover, can be set to:
# * "container" to discover virtual machines (SmartOS zones, lx/KVM/bhyve branded zo
# * "cn" to discover compute nodes (servers/global zones) making up the Triton infra
[ role : <string> | default = "container" ]

# The DNS suffix which should be applied to target.
dns_suffix: <string>

# The Triton discovery endpoint (e.g. 'cmon.us-east-3b.triton.zone'). This is
# often the same value as dns_suffix.
endpoint: <string>
```

```
groups:
  [ - <string> ... ]

# The port to use for discovery and metric scraping.
[ port: <int> | default = 9163 ]

# The interval which should be used for refreshing targets.
[ refresh_interval: <duration> | default = 60s ]

# The Triton discovery API version.
[ version: <int> | default = 1 ]

# TLS configuration.
tls_config:
  [ <tls_config> ]
```

`<eureka_sd_config>`

Eureka SD configurations allow retrieving scrape targets using the Eureka REST API. Prometheus will periodically check the REST endpoint and create a target for every app instance.

The following meta labels are available on targets during relabeling:

- `__meta_eureka_app_name` : the name of the app
- `__meta_eureka_app_instance_id` : the ID of the app instance
- `__meta_eureka_app_instance_hostname` : the hostname of the instance
- `__meta_eureka_app_instance_homepage_url` : the homepage url of the app instance
- `__meta_eureka_app_instance_statuspage_url` : the status page url of the app instance
- `__meta_eureka_app_instance_healthcheck_url` : the health check url of the app instance
- `__meta_eureka_app_instance_ip_addr` : the IP address of the app instance
- `__meta_eureka_app_instance_vip_address` : the VIP address of the app instance
- `__meta_eureka_app_instance_secure_vip_address` : the secure VIP address of the app instance
- `__meta_eureka_app_instance_status` : the status of the app instance
- `__meta_eureka_app_instance_port` : the port of the app instance
- `__meta_eureka_app_instance_port_enabled` : the port enabled of the app instance

instance

- `__meta_eureka_app_instance_country_id` : the country ID of the app instance
- `__meta_eureka_app_instance_metadata_<metadataname>` : app instance metadata
- `__meta_eureka_app_instance_datacenterinfo_name` : the datacenter name of the app instance
- `__meta_eureka_app_instance_datacenterinfo_<metadataname>` : the datacenter metadata

See below for the configuration options for Eureka discovery:

```
# The URL to connect to the Eureka server.
server: <string>

# Refresh interval to re-read the app instance list.
[ refresh_interval: <duration> | default = 30s ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
[ <http_config> ]
```

See the Prometheus eureka-sd configuration file for a practical example on how to set up your Eureka app and your Prometheus configuration.

`<scaleway_sd_config>`

Scaleway SD configurations allow retrieving scrape targets from Scaleway instances and baremetal services.

The following meta labels are available on targets during relabeling:

Instance role

- `__meta_scaleway_instance_boot_type` : the boot type of the server
- `__meta_scaleway_instance_hostname` : the hostname of the server
- `__meta_scaleway_instance_id` : the ID of the server
- `__meta_scaleway_instance_image_arch` : the arch of the server image
- `__meta_scaleway_instance_image_id` : the ID of the server image
- `__meta_scaleway_instance_image_name` : the name of the server image

- `__meta_scaleway_instance_location_node_id` : the node ID of the server location
- `__meta_scaleway_instance_name` : name of the server
- `__meta_scaleway_instance_organization_id` : the organization of the server
- `__meta_scaleway_instance_private_ipv4` : the private IPv4 address of the server
- `__meta_scaleway_instance_project_id` : project id of the server
- `__meta_scaleway_instance_public_ipv4` : the public IPv4 address of the server
- `__meta_scaleway_instance_public_ipv6` : the public IPv6 address of the server
- `__meta_scaleway_instance_region` : the region of the server
- `__meta_scaleway_instance_security_group_id` : the ID of the security group of the server
- `__meta_scaleway_instance_security_group_name` : the name of the security group of the server
- `__meta_scaleway_instance_status` : status of the server
- `__meta_scaleway_instance_tags` : the list of tags of the server joined by the tag separator
- `__meta_scaleway_instance_type` : commercial type of the server
- `__meta_scaleway_instance_zone` : the zone of the server (ex: `fr-par-1` , complete list here)

This role uses the first address it finds in the following order: private IPv4, public IPv4, public IPv6. This can be changed with relabeling, as demonstrated in the Prometheus scaleway-sd configuration file. Should an instance have no address before relabeling, it will not be added to the target list and you will not be able to relabel it.

Baremetal role

- `__meta_scaleway_baremetal_id` : the ID of the server
- `__meta_scaleway_baremetal_public_ipv4` : the public IPv4 address of the server
- `__meta_scaleway_baremetal_public_ipv6` : the public IPv6 address of the server
- `__meta_scaleway_baremetal_name` : the name of the server
- `__meta_scaleway_baremetal_os_name` : the name of the operating system of the server
- `__meta_scaleway_baremetal_os_version` : the version of the operating system of the server
- `__meta_scaleway_baremetal_project_id` : the project ID of the server
- `__meta_scaleway_baremetal_status` : the status of the server
- `__meta_scaleway_baremetal_tags` : the list of tags of the server joined by the tag separator
- `__meta_scaleway_baremetal_type` : the commercial type of the server

This role uses the public IPv4 address by default. This can be changed with relabeling, as demonstrated in the Prometheus scaleway-sd configuration file.

See below for the configuration options for Scaleway discovery:

```
# Access key to use. https://console.scaleway.com/project/credentials
access_key: <string>

# Secret key to use when listing targets. https://console.scaleway.com/project/crede
# It is mutually exclusive with `secret_key_file`.
[ secret_key: <secret> ]

# Sets the secret key with the credentials read from the configured file.
# It is mutually exclusive with `secret_key`.
[ secret_key_file: <filename> ]

# Project ID of the targets.
project_id: <string>

# Role of the targets to retrieve. Must be `instance` or `baremetal`.
role: <string>

# The port to scrape metrics from.
[ port: <int> | default = 80 ]

# API URL to use when doing the server listing requests.
[ api_url: <string> | default = "https://api.scaleway.com" ]

# Zone is the availability zone of your targets (e.g. fr-par-1).
[ zone: <string> | default = fr-par-1 ]

# NameFilter specify a name filter (works as a LIKE) to apply on the server listing
[ name_filter: <string> ]

# TagsFilter specify a tag filter (a server needs to have all defined tags to be lis
tags_filter:
[ - <string> ]

# Refresh interval to re-read the targets list.
[ refresh_interval: <duration> | default = 60s ]
```

Prometheus

`<uyuni_sd_config>`

Uyuni SD configurations allow retrieving scrape targets from managed systems via Uyuni API.

The following meta labels are available on targets during relabeling:

- `__meta_uyuni_endpoint_name` : the name of the application endpoint
- `__meta_uyuni_exporter` : the exporter exposing metrics for the target
- `__meta_uyuni_groups` : the system groups of the target
- `__meta_uyuni_metrics_path` : metrics path for the target
- `__meta_uyuni_minion_hostname` : hostname of the Uyuni client
- `__meta_uyuni_primary_fqdn` : primary FQDN of the Uyuni client
- `__meta_uyuni_proxy_module` : the module name if *Exporter Exporter* proxy is configured for the target
- `__meta_uyuni_scheme` : the protocol scheme used for requests
- `__meta_uyuni_system_id` : the system ID of the client

See below for the configuration options for Uyuni discovery:

```
# The URL to connect to the Uyuni server.
server: <string>

# Credentials are used to authenticate the requests to Uyuni API.
username: <string>
password: <secret>

# The entitlement string to filter eligible systems.
[ entitlement: <string> | default = monitoring_entitled ]

# The string by which Uyuni group names are joined into the groups label.
[ separator: <string> | default = , ]

# Refresh interval to re-read the managed targets list.
[ refresh_interval: <duration> | default = 60s ]

# HTTP client settings, including authentication methods (such as basic auth and
```

Prometheus

See the Prometheus uyuni-sd configuration file for a practical example on how to set up Uyuni Prometheus configuration.

`<vultr_sd_config>`

Vultr SD configurations allow retrieving scrape targets from Vultr.

This service discovery uses the main IPv4 address by default, which that be changed with relabeling, as demonstrated in the Prometheus vultr-sd configuration file.

The following meta labels are available on targets during relabeling:

- `__meta_vultr_instance_id` : A unique ID for the vultr Instance.
- `__meta_vultr_instance_label` : The user-supplied label for this instance.
- `__meta_vultr_instance_os` : The Operating System name.
- `__meta_vultr_instance_os_id` : The Operating System id used by this instance.
- `__meta_vultr_instance_region` : The Region id where the Instance is located.
- `__meta_vultr_instance_plan` : A unique ID for the Plan.
- `__meta_vultr_instance_main_ip` : The main IPv4 address.
- `__meta_vultr_instance_internal_ip` : The private IP address.
- `__meta_vultr_instance_main_ipv6` : The main IPv6 address.
- `__meta_vultr_instance_features` : List of features that are available to the instance.
- `__meta_vultr_instance_tags` : List of tags associated with the instance.
- `__meta_vultr_instance_hostname` : The hostname for this instance.
- `__meta_vultr_instance_server_status` : The server health status.
- `__meta_vultr_instance_vcpu_count` : Number of vCPUs.
- `__meta_vultr_instance_ram_mb` : The amount of RAM in MB.
- `__meta_vultr_instance_disk_gb` : The size of the disk in GB.
- `__meta_vultr_instance_allowed_bandwidth_gb` : Monthly bandwidth quota in GB.

```
# The port to scrape metrics from.
[ port: <int> | default = 80 ]

# The time after which the instances are refreshed.
[ refresh_interval: <duration> | default = 60s ]

# HTTP client settings, including authentication methods (such as basic auth and
```

`<static_config>`

A `static_config` allows specifying a list of targets and a common label set for them. It is the canonical way to specify static targets in a scrape configuration.

```
# The targets specified by the static config.
targets:
  [ - '<host>' ]

# Labels assigned to all metrics scraped from the targets.
labels:
  [ <labelname>: <labelvalue> ... ]
```

`<relabel_config>`

Relabeling is a powerful tool to dynamically rewrite the label set of a target before it gets scraped. Multiple relabeling steps can be configured per scrape configuration. They are applied to the label set of each target in order of their appearance in the configuration file.

Initially, aside from the configured per-target labels, a target's `job` label is set to the `job_name` value of the respective scrape configuration. The `__address__` label is set to the `<host>:<port>` address of the target. After relabeling, the `instance` label is set to the value of `__address__` by default if it was not set during relabeling.

The `__scheme__` and `__metrics_path__` labels are set to the scheme and metrics path of the target respectively, as specified in `scrape_config` .

The `__param_<name>` label is set to the value of the first passed URL parameter called `<name>` , as defined in `scrape_config` .

The `__scrape_interval__` and `__scrape_timeout__` labels are set to the target's interval and timeout, as specified in `scrape_config` .

Additional labels prefixed with `__meta_` may be available during the relabeling phase. They are set by the service discovery mechanism that provided the target and vary between mechanisms.

Labels starting with `__` will be removed from the label set after target relabeling is completed.

Prometheus

```
# The source labels select values from existing labels. Their content is concatenate
# using the configured separator and matched against the configured regular express:
# for the replace, keep, and drop actions.
[ source_labels: '[' <labelname> [, ...] ']' ]

# Separator placed between concatenated source label values.
[ separator: <string> | default = ; ]

# Label to which the resulting value is written in a replace action.
# It is mandatory for replace actions. Regex capture groups are available.
[ target_label: <labelname> ]

# Regular expression against which the extracted value is matched.
[ regex: <regex> | default = (.*) ]

# Modulus to take of the hash of the source label values.
[ modulus: <int> ]

# Replacement value against which a regex replace is performed if the
# regular expression matches. Regex capture groups are available.
[ replacement: <string> | default = $1 ]

# Action to perform based on regex matching.
[ action: <relabel_action> | default = replace ]
```

`<regex>` is any valid RE2 regular expression. It is required for the `replace`, `keep`, `drop`, `labelmap`, `labeldrop` and `labelkeep` actions. The regex is anchored on both ends. To un-anchor the regex, use `.*<regex>.*`.

`<relabel_action>` determines the relabeling action to take:

- `replace`: Match `regex` against the concatenated `source_labels`. Then, set `target_label` to `replacement`, with match group references (`${1}`, `${2}`, ...) in `replacement` substituted by their value. If `regex` does not match, no replacement takes place.
- `lowercase`: Maps the concatenated `source_labels` to their lower case.
- `uppercase`: Maps the concatenated `source_labels` to their upper case.

- `keepequal` : Drop targets for which the concatenated `source_labels` do not match `target_label` .
- `dropequal` : Drop targets for which the concatenated `source_labels` do match `target_label` .
- `hashmod` : Set `target_label` to the `modulus` of a hash of the concatenated `source_labels` .
- `labelmap` : Match `regex` against all source label names, not just those specified in `source_labels` . Then copy the values of the matching labels to label names given by `replacement` with match group references (`${1}` , `${2}` , ...) in `replacement` substituted by their value.
- `labeldrop` : Match `regex` against all label names. Any label that matches will be removed from the set of labels.
- `labelkeep` : Match `regex` against all label names. Any label that does not match will be removed from the set of labels.

Care must be taken with `labeldrop` and `labelkeep` to ensure that metrics are still uniquely labeled once the labels are removed.

`<metric_relabel_configs>`

Metric relabeling is applied to samples as the last step before ingestion. It has the same configuration format and actions as target relabeling. Metric relabeling does not apply to automatically generated timeseries such as `up` .

One use for this is to exclude time series that are too expensive to ingest.

`<alert_relabel_configs>`

Alert relabeling is applied to alerts before they are sent to the Alertmanager. It has the same configuration format and actions as target relabeling. Alert relabeling is applied after external labels.

One use for this is ensuring a HA pair of Prometheus servers with different external labels send identical alerts.

`<alertmanager_config>`

Alertmanagers may be statically configured via the `static_configs` parameter or dynamically discovered using one of the supported service-discovery mechanisms.

Additionally, `relabel_configs` allow selecting Alertmanagers from discovered entities and provide advanced modifications to the used API path, which is exposed through the `__alerts_path__` label.

```
# Per-target Alertmanager timeout when pushing alerts.
[ timeout: <duration> | default = 10s ]

# The api version of Alertmanager.
[ api_version: <string> | default = v2 ]

# Prefix for the HTTP path alerts are pushed to.
[ path_prefix: <path> | default = / ]

# Configures the protocol scheme used for requests.
[ scheme: <scheme> | default = http ]

# Optionally configures AWS's Signature Verification 4 signing process to sign reque
# Cannot be set at the same time as basic_auth, authorization, oauth2, azuread or go
# To use the default credentials from the AWS SDK, use `sigv4: {}`.
sigv4:
  # The AWS region. If blank, the region from the default credentials chain
  # is used.
  [ region: <string> ]

  # The AWS API keys. If blank, the environment variables `AWS_ACCESS_KEY_ID`
  # and `AWS_SECRET_ACCESS_KEY` are used.
  [ access_key: <string> ]
  [ secret_key: <secret> ]

  # Named AWS profile used to authenticate.
  [ profile: <string> ]

  # AWS Role ARN, an alternative to using AWS API keys.
  [ role_arn: <string> ]

# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
```

```
azure_sd_configs:
  [ - <azure_sd_config> ... ]

# List of Consul service discovery configurations.
consul_sd_configs:
  [ - <consul_sd_config> ... ]

# List of DNS service discovery configurations.
dns_sd_configs:
  [ - <dns_sd_config> ... ]

# List of EC2 service discovery configurations.
ec2_sd_configs:
  [ - <ec2_sd_config> ... ]

# List of Eureka service discovery configurations.
eureka_sd_configs:
  [ - <eureka_sd_config> ... ]

# List of file service discovery configurations.
file_sd_configs:
  [ - <file_sd_config> ... ]

# List of DigitalOcean service discovery configurations.
digitalocean_sd_configs:
  [ - <digitalocean_sd_config> ... ]

# List of Docker service discovery configurations.
docker_sd_configs:
  [ - <docker_sd_config> ... ]

# List of Docker Swarm service discovery configurations.
dockerswarm_sd_configs:
  [ - <dockerswarm_sd_config> ... ]

# List of GCE service discovery configurations.
gce_sd_configs:
  [ - <gce_sd_config> ... ]

# List of Hetzner service discovery configurations.
hetzner_sd_configs:
```

Prometheus

```
http_sd_configs:
  [ - <http_sd_config> ... ]

 # List of IONOS service discovery configurations.
ionos_sd_configs:
  [ - <ionos_sd_config> ... ]

# List of Kubernetes service discovery configurations.
kubernetes_sd_configs:
  [ - <kubernetes_sd_config> ... ]

# List of Lightsail service discovery configurations.
lightsail_sd_configs:
  [ - <lightsail_sd_config> ... ]

# List of Linode service discovery configurations.
linode_sd_configs:
  [ - <linode_sd_config> ... ]

# List of Marathon service discovery configurations.
marathon_sd_configs:
  [ - <marathon_sd_config> ... ]

# List of AirBnB's Nerve service discovery configurations.
nerve_sd_configs:
  [ - <nerve_sd_config> ... ]

# List of Nomad service discovery configurations.
nomad_sd_configs:
  [ - <nomad_sd_config> ... ]

# List of OpenStack service discovery configurations.
openstack_sd_configs:
  [ - <openstack_sd_config> ... ]

# List of OVHcloud service discovery configurations.
ovhcloud_sd_configs:
  [ - <ovhcloud_sd_config> ... ]

# List of PuppetDB service discovery configurations.
puppetdb_sd_configs:
scaleway_sd_configs:
  [ - <scaleway_sd_config> ... ]

# List of Zookeeper Serverset service discovery configurations.
serverset_sd_configs:
  [ - <serverset_sd_config> ... ]

# List of Triton service discovery configurations.
triton_sd_configs:
  [ - <triton_sd_config> ... ]

# List of Uyuni service discovery configurations.
uyuni_sd_configs:
  [ - <uyuni_sd_config> ... ]

# List of Vultr service discovery configurations.
vultr_sd_configs:
  [ - <vultr_sd_config> ... ]

# List of labeled statically configured Alertmanagers.
static_configs:
  [ - <static_config> ... ]

# List of Alertmanager relabel configurations.
relabel_configs:
  [ - <relabel_config> ... ]

# List of alert relabel configurations.
alert_relabel_configs:
  [ - <relabel_config> ... ]
```

`<remote_write>`

`write_relabel_configs` is relabeling applied to samples before sending them to the remote endpoint. Write relabeling is applied after external labels. This could be used to limit which samples are sent.

There is a small demo of how to use this functionality.

Prometheus

```
# protobuf message to use when writing to the remote write endpoint.
#
# * The `prometheus.WriteRequest` represents the message introduced in Remote Write
# will be deprecated eventually.
# * The `io.prometheus.write.v2.Request` was introduced in Remote Write 2.0 and rep
# by improving efficiency and sending metadata, created timestamp and native histog
#
# Before changing this value, consult with your remote storage provider (or test) w
# Read more on https://prometheus.io/docs/specs/remote_write_spec_2_0/#io-prometheu
[ protobuf_message: <prometheus.WriteRequest | io.prometheus.write.v2.Request> | de

# Timeout for requests to the remote write endpoint.
[ remote_timeout: <duration> | default = 30s ]

# Custom HTTP headers to be sent along with each remote write request.
# Be aware that headers that are set by Prometheus itself can't be overwritten.
headers:
  [ <string>: <string> ... ]

# List of remote write relabel configurations.
write_relabel_configs:
  [ - <relabel_config> ... ]

# Name of the remote write config, which if specified must be unique among remote w
# The name will be used in metrics and logging in place of a generated value to hel
# remote write configs.
[ name: <string> ]

# Enables sending of exemplars over remote write. Note that exemplar storage itself
[ send_exemplars: <boolean> | default = false ]

# Enables sending of native histograms, also known as sparse histograms, over remot
# For the `io.prometheus.write.v2.Request` message, this option is noop (always tru
[ send_native_histograms: <boolean> | default = false ]

# When enabled, remote-write will resolve the URL host name via DNS, choose one of
# When disabled, remote-write relies on Go's standard behavior, which is to try to
# The connection timeout applies to the whole operation, i.e. in the latter case it
# This is an experimental feature, and its behavior might still change, or even get
[ round_robin_dns: <boolean> | default = false ]
```

Prometheus

```
# To use the default credentials from the AWS SDK, use `sigv4: {}`.
sigv4:
  # The AWS region. If blank, the region from the default credentials chain
  # is used.
  [ region: <string> ]

  # The AWS API keys. If blank, the environment variables `AWS_ACCESS_KEY_ID`
  # and `AWS_SECRET_ACCESS_KEY` are used.
  [ access_key: <string> ]
  [ secret_key: <secret> ]

  # Named AWS profile used to authenticate.
  [ profile: <string> ]

  # AWS Role ARN, an alternative to using AWS API keys.
  [ role_arn: <string> ]

# Optional AzureAD configuration.
# Cannot be used at the same time as basic_auth, authorization, oauth2, sigv4 or goo
azuread:
  # The Azure Cloud. Options are 'AzurePublic', 'AzureChina', or 'AzureGovernment'.
  [ cloud: <string> | default = AzurePublic ]

  # Azure User-assigned Managed identity.
  [ managed_identity:
      [ client_id: <string> ] ]

  # Azure OAuth.
  [ oauth:
      [ client_id: <string> ]
      [ client_secret: <string> ]
      [ tenant_id: <string> ] ]

  # Azure SDK auth.
  # See https://learn.microsoft.com/en-us/azure/developer/go/azure-sdk-authenticatio
  [ sdk:
      [ tenant_id: <string> ] ]

# WARNING: Remote write is NOT SUPPORTED by Google Cloud. This configuration is rese
# Optional Google Cloud Monitoring configuration.
# Cannot be used at the same time as basic_auth, authorization, oauth2, sigv4 or azu
```

```
    credentials_file: <file_name>

  # Configures the queue used to write to remote storage.
  queue_config:
    # Number of samples to buffer per shard before we block reading of more
    # samples from the WAL. It is recommended to have enough capacity in each
    # shard to buffer several requests to keep throughput up while processing
    # occasional slow remote requests.
    [ capacity: <int> | default = 10000 ]
    # Maximum number of shards, i.e. amount of concurrency.
    [ max_shards: <int> | default = 50 ]
    # Minimum number of shards, i.e. amount of concurrency.
    [ min_shards: <int> | default = 1 ]
    # Maximum number of samples per send.
    [ max_samples_per_send: <int> | default = 2000]
    # Maximum time a sample will wait for a send. The sample might wait less
    # if the buffer is full. Further time might pass due to potential retries.
    [ batch_send_deadline: <duration> | default = 5s ]
    # Initial retry delay. Gets doubled for every retry.
    [ min_backoff: <duration> | default = 30ms ]
    # Maximum retry delay.
    [ max_backoff: <duration> | default = 5s ]
    # Retry upon receiving a 429 status code from the remote-write storage.
    # This is experimental and might change in the future.
    [ retry_on_http_429: <boolean> | default = false ]
    # If set, any sample that is older than sample_age_limit
    # will not be sent to the remote storage. The default value is 0s,
    # which means that all samples are sent.
    [ sample_age_limit: <duration> | default = 0s ]

  # Configures the sending of series metadata to remote storage
  # if the `prometheus.WriteRequest` message was chosen. When
  # `io.prometheus.write.v2.Request` is used, metadata is always sent.
  #
  # Metadata configuration is subject to change at any point
  # or be removed in future releases.
  metadata_config:
    # Whether metric metadata is sent to remote storage or not.
    [ send: <boolean> | default = true ]
    # How frequently metric metadata is sent to remote storage.
    [ send_interval: <duration> | default = 1m ]
```

```
# HTTP client settings, including authentication methods (such as basic auth and
# authorization), proxy configurations, TLS options, custom HTTP headers, etc.
# enable_http2 defaults to false for remote-write.
[ <http_config> ]
```

There is a list of integrations with this feature.

`<remote_read>`

```
# The URL of the endpoint to query from.
url: <string>

# Name of the remote read config, which if specified must be unique among remote rea
# The name will be used in metrics and logging in place of a generated value to hel
# remote read configs.
[ name: <string> ]

# An optional list of equality matchers which have to be
# present in a selector to query the remote read endpoint.
required_matchers:
  [ <labelname>: <labelvalue> ... ]

# Timeout for requests to the remote read endpoint.
[ remote_timeout: <duration> | default = 1m ]

# Custom HTTP headers to be sent along with each remote read request.
# Be aware that headers that are set by Prometheus itself can't be overwritten.
headers:
  [ <string>: <string> ... ]

# Whether reads should be made for queries for time ranges that
# the local storage should have complete data for.
[ read_recent: <boolean> | default = false ]

# Whether to use the external labels as selectors for the remote read endpoint.
[ filter_external_labels: <boolean> | default = true ]

# HTTP client settings, including authentication methods (such as basic auth and
```

Prometheus

There is a list of integrations with this feature.

<tsdb>

`tsdb` lets you configure the runtime-reloadable configuration settings of the TSDB.

```
# Configures how old an out-of-order/out-of-bounds sample can be w.r.t. the TSDB max
# An out-of-order/out-of-bounds sample is ingested into the TSDB as long as the time
# of the sample is >= TSDB.MaxTime-out_of_order_time_window.
#
# When out_of_order_time_window is >0, the errors out-of-order and out-of-bounds are
# combined into a single error called 'too-old'; a sample is either (a) ingestible
# into the TSDB, i.e. it is an in-order sample or an out-of-order/out-of-bounds samp
# that is within the out-of-order window, or (b) too-old, i.e. not in-order
# and before the out-of-order window.
#
# When out_of_order_time_window is greater than 0, it also affects experimental agen
# the agent's WAL to accept out-of-order samples that fall within the specified time
# to the timestamp of the last appended sample for the same series.
[ out_of_order_time_window: <duration> | default = 0s ]
```

<exemplars>

Note that exemplar storage is still considered experimental and must be enabled via `--enable-feature=exemplar-storage`.

```
# Configures the maximum size of the circular buffer used to store exemplars for all
[ max_exemplars: <int> | default = 100000 ]
```

<tracing_config>

`tracing_config` configures exporting traces from Prometheus to a tracing backend via the OTLP protocol. Tracing is currently an **experimental** feature and could change in the future.

Prometheus

```
# Endpoint to send the traces to. Should be provided in format <host>:<port>.
[ endpoint: <string> ]

# Sets the probability a given trace will be sampled. Must be a float from 0 througł
[ sampling_fraction: <float> | default = 0 ]

# If disabled, the client will use a secure connection.
[ insecure: <boolean> | default = false ]

# Key-value pairs to be used as headers associated with gRPC or HTTP requests.
headers:
  [ <string>: <string> ... ]

# Compression key for supported compression types. Supported compression: gzip.
[ compression: <string> ]

# Maximum time the exporter will wait for each batch export.
[ timeout: <duration> | default = 10s ]

# TLS configuration.
tls_config:
  [ <tls_config> ]
```

Previous
Installation

Edit

Next
Recording rules

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Defining recording rules

Configuring rules

Prometheus supports two types of rules which may be configured and then evaluated at regular intervals: recording rules and alerting rules. To include rules in Prometheus, create a file containing the necessary rule statements and have Prometheus load the file via the `rule_files` field in the Prometheus configuration. Rule files use YAML.

The rule files can be reloaded at runtime by sending `SIGHUP` to the Prometheus process. The changes are only applied if all rule files are well-formatted.

Note about native histograms (experimental feature): Native histogram are always recorded as gauge histograms (for now). Most cases will create gauge histograms naturally, e.g. after `rate()`.

Syntax-checking rules

To quickly check whether a rule file is syntactically correct without starting a Prometheus server, you can use Prometheus's `promtool` command-line utility tool:

```
promtool check rules /path/to/example.rules.yml
```

The `promtool` binary is part of the `prometheus` archive offered on the project's download page.

When the file is syntactically valid, the checker prints a textual representation of the parsed rules to standard output and then exits with a `0` return status.

Prometheus

Recording rules

Recording rules allow you to precompute frequently needed or computationally expensive expressions and save their result as a new set of time series. Querying the precomputed result will then often be much faster than executing the original expression every time it is needed. This is especially useful for dashboards, which need to query the same expression repeatedly every time they refresh.

Recording and alerting rules exist in a rule group. Rules within a group are run sequentially at a regular interval, with the same evaluation time. The names of recording rules must be valid metric names. The names of alerting rules must be valid label values.

The syntax of a rule file is:

```
groups:
  [ - <rule_group> ]
```

A simple example rules file would be:

```
groups:
  - name: example
    rules:
    - record: code:prometheus_http_requests_total:sum
      expr: sum by (code) (prometheus_http_requests_total)
```

`<rule_group>`

```
# The name of the group. Must be unique within a file.
name: <string>

# How often rules in the group are evaluated.
[ interval: <duration> | default = global.evaluation_interval ]

# Limit the number of alerts an alerting rule and series a recording
# rule can produce. 0 is no limit.
[ limit: <int> | default = 0 ]
```

```
# Labels to add or overwrite before storing the result for its rules.
# Labels defined in <rule> will override the key if it has a collision.
labels:
  [ <labelname>: <labelvalue> ]

rules:
  [ - <rule> ... ]
```

`<rule>`

The syntax for recording rules is:

```
# The name of the time series to output to. Must be a valid metric name.
record: <string>

# The PromQL expression to evaluate. Every evaluation cycle this is
# evaluated at the current time, and the result recorded as a new set of
# time series with the metric name as given by 'record'.
expr: <string>

# Labels to add or overwrite before storing the result.
labels:
  [ <labelname>: <labelvalue> ]
```

The syntax for alerting rules is:

```
# The name of the alert. Must be a valid label value.
alert: <string>

# The PromQL expression to evaluate. Every evaluation cycle this is
# evaluated at the current time, and all resultant time series become
# pending/firing alerts.
expr: <string>

# Alerts are considered firing once they have been returned for this long.
# Alerts which have not yet fired for long enough are considered pending.
[ for: <duration> | default = 0s ]
```

```
# Labels to add or overwrite for each alert.
labels:
  [ <labelname>: <tmpl_string> ]

# Annotations to add to each alert.
annotations:
  [ <labelname>: <tmpl_string> ]
```

See also the best practices for naming metrics created by recording rules.

Limiting alerts and series

A limit for alerts produced by alerting rules and series produced recording rules can be configured per-group. When the limit is exceeded, *all* series produced by the rule are discarded, and if it's an alerting rule, *all* alerts for the rule, active, pending, or inactive, are cleared as well. The event will be recorded as an error in the evaluation, and as such no stale markers are written.

Rule query offset

This is useful to ensure the underlying metrics have been received and stored in Prometheus. Metric availability delays are more likely to occur when Prometheus is running as a remote write target due to the nature of distributed systems, but can also occur when there's anomalies with scraping and/or short evaluation intervals.

Failed rule evaluations due to slow evaluation

If a rule group hasn't finished evaluating before its next evaluation is supposed to start (as defined by the `evaluation_interval`), the next evaluation will be skipped. Subsequent evaluations of the rule group will continue to be skipped until the initial evaluation either completes or times out. When this happens, there will be a gap in the metric produced by the recording rule. The `rule_group_iterations_missed_total` metric will be incremented for each missed iteration of the rule group.

Prometheus

← Configuration

Edit

Alerting rules →

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Alerting rules

Alerting rules allow you to define alert conditions based on Prometheus expression language expressions and to send notifications about firing alerts to an external service. Whenever the alert expression results in one or more vector elements at a given point in time, the alert counts as active for these elements' label sets.

Defining alerting rules

Alerting rules are configured in Prometheus in the same way as recording rules.

An example rules file with an alert would be:

```
groups:
- name: example
  labels:
    team: myteam
  rules:
  - alert: HighRequestLatency
    expr: job:request_latency_seconds:mean5m{job="myjob"} > 0.5
    for: 10m
    keep_firing_for: 5m
    labels:
      severity: page
    annotations:
      summary: High request latency
```

The optional `for` clause causes Prometheus to wait for a certain duration between first encountering a new expression output vector element and counting an alert as firing for this element. In this case, Prometheus will check that the alert continues to be active during each

There is also an optional `keep_firing_for` clause that tells Prometheus to keep this alert firing for the specified duration after the firing condition was last met. This can be used to prevent situations such as flapping alerts, false resolutions due to lack of data loss, etc. Alerting rules without the `keep_firing_for` clause will deactivate on the first evaluation where the condition is not met (assuming any optional `for` duration desribed above has been satisfied).

The `labels` clause allows specifying a set of additional labels to be attached to the alert. Any existing conflicting labels will be overwritten. The label values can be templated.

The `annotations` clause specifies a set of informational labels that can be used to store longer additional information such as alert descriptions or runbook links. The annotation values can be templated.

Templating

Label and annotation values can be templated using console templates. The `$labels` variable holds the label key/value pairs of an alert instance. The configured external labels can be accessed via the `$externalLabels` variable. The `$value` variable holds the evaluated value of an alert instance.

```
# To insert a firing element's label values:
{{ $labels.<labelname> }}
# To insert the numeric expression value of the firing element:
{{ $value }}
```

Examples:

```
groups:
- name: example
  rules:

  # Alert for any instance that is unreachable for >5 minutes.
  - alert: InstanceDown
    expr: up == 0
    for: 5m
    labels:
      severity: page
  # Alert for any instance that has a median request latency >1s.
  - alert: APIHighRequestLatency
    expr: api_http_request_latencies_second{quantile="0.5"} > 1
    for: 10m
    annotations:
      summary: "High request latency on {{ $labels.instance }}"
      description: "{{ $labels.instance }} has a median request latency above 1s (cu
```

Inspecting alerts during runtime

To manually inspect which alerts are active (pending or firing), navigate to the "Alerts" tab of your Prometheus instance. This will show you the exact label sets for which each defined alert is currently active.

For pending and firing alerts, Prometheus also stores synthetic time series of the form `ALERTS{alertname="<alert name>", alertstate="<pending or firing>", <additional alert labels>}`. The sample value is set to `1` as long as the alert is in the indicated active (pending or firing) state, and the series is marked stale when this is no longer the case.

Sending alert notifications

Prometheus's alerting rules are good at figuring what is broken *right now*, but they are not a fully-fledged notification solution. Another layer is needed to add summarization, notification rate limiting, silencing and alert dependencies on top of the simple alert definitions. In Prometheus's ecosystem, the Alertmanager takes on this role. Thus, Prometheus may be configured to periodically send information about alert states to an Alertmanager instance, which then takes care of dispatching the right notifications. Prometheus can be configured to automatically discover available Alertmanager instances through its service discovery integrations.

← Previous
Recording rules

Edit

Next
Template examples →

Prometheus

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Template examples

Prometheus supports templating in the annotations and labels of alerts, as well as in served console pages. Templates have the ability to run queries against the local database, iterate over data, use conditionals, format data, etc. The Prometheus templating language is based on the Go templating system.

Simple alert field templates

```
alert: InstanceDown
expr: up == 0
for: 5m
labels:
  severity: page
annotations:
  summary: "Instance {{$labels.instance}} down"
  description: "{{$labels.instance}} of job {{$labels.job}} has been down for more t
```

Alert field templates will be executed during every rule iteration for each alert that fires, so keep any queries and templates lightweight. If you have a need for more complicated templates for alerts, it is recommended to link to a console instead.

Simple iteration

This displays a list of instances, and whether they are up:

```
{{ end }}
```

The special `.` variable contains the value of the current sample for each loop iteration.

Display one value

```
{{ with query "some_metric{instance='someinstance'}" }}
  {{ . | first | value | humanize }}
{{ end }}
```

Go and Go's templating language are both strongly typed, so one must check that samples were returned to avoid an execution error. For example this could happen if a scrape or rule evaluation has not run yet, or a host was down.

The included `prom_query_drilldown` template handles this, allows for formatting of results, and linking to the expression browser.

Using console URL parameters

```
{{ with printf "node_memory_MemTotal{job='node',instance='%s'}" .Params.instance | 
  {{ . | first | value | humanize1024 }}B
{{ end }}
```

If accessed as `console.html?instance=hostname`, `.Params.instance` will evaluate to `hostname`.

Advanced iteration

```
<table>
{{ range printf "node_network_receive_bytes{job='node',instance='%s',device!='lo'}"
  <tr><th colspan=2>{{ .Labels.device }}</th></tr>
  <tr>
    <td>Received</td>
    <td>{{ with printf "rate(node_network_receive_bytes{job='node',instance='%s',de
  </tr>
  </tr>{{ end }}
</table>
```

Here we iterate over all network devices and display the network traffic for each.

As the `range` action does not specify a variable, `.Params.instance` is not available inside the loop as `.` is now the loop variable.

Defining reusable templates

Prometheus supports defining templates that can be reused. This is particularly powerful when combined with console library support, allowing sharing of templates across consoles.

```
{{/* Define the template */}}
{{define "myTemplate"}}
  do something
{{end}}

{{/* Use the template */}}
{{template "myTemplate"}}
```

Templates are limited to one argument. The `args` function can be used to wrap multiple arguments.

```
{{define "myMultiArgTemplate"}}
  First argument: {{.arg0}}
  Second argument: {{.arg1}}
{{end}}
{{template "myMultiArgTemplate" (args 1 2)}}
```

← Previous Alerting rules

Edit

Next Template reference →

Prometheus

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Template reference

Prometheus supports templating in the annotations and labels of alerts, as well as in served console pages. Templates have the ability to run queries against the local database, iterate over data, use conditionals, format data, etc. The Prometheus templating language is based on the Go templating system.

Data Structures

The primary data structure for dealing with time series data is the sample, defined as:

```
type sample struct {
        Labels map[string]string
        Value  interface{}
}
```

The metric name of the sample is encoded in a special `__name__` label in the `Labels` map.

`[]sample` means a list of samples.

`interface{}` in Go is similar to a void pointer in C.

Functions

In addition to the default functions provided by Go templating, Prometheus provides functions for easier processing of query results in templates.

If functions are used in a pipeline, the pipeline value is passed as the last argument.

query	query string	[]sample	Queries the database, does not support returning range vectors.
first	[]sample	sample	Equivalent to `index a 0`
label	label, sample	string	Equivalent to `index sample.Labels label`
value	sample	interface{}	Equivalent to `sample.Value`
sortByLabel	label, []samples	[]sample	Sorts the samples by the given label. Is stable.

`first`, `label` and `value` are intended to make query results easily usable in pipelines.

Numbers

Name	Arguments	Returns	Notes
humanize	number or string	string	Converts a number to a more readable format, using metric prefixes.
humanize1024	number or string	string	Like `humanize`, but uses 1024 as the base rather than 1000.
humanizeDuration	number or string	string	Converts a duration in seconds to a more readable format.
humanizePercentage	number or string	string	Converts a ratio value to a fraction of 100.
humanizeTimestamp	number or string	string	Converts a Unix timestamp in seconds to a more readable format.
toTime	number or string	*time.Time	Converts a Unix timestamp in seconds to a time.Time.

Humanizing functions are intended to produce reasonable output for consumption by humans, and are not guaranteed to return the same results between Prometheus versions.

Strings

Prometheus

			word.
toUpper	string	string	strings.ToUpper, converts all characters to upper case.
toLower	string	string	strings.ToLower, converts all characters to lower case.
stripPort	string	string	net.SplitHostPort, splits string into host and port, then returns only host.
match	pattern, text	boolean	regexp.MatchString Tests for a unanchored regexp match.
reReplaceAll	pattern, replacement, text	string	Regexp.ReplaceAllString Regexp substitution, unanchored.
graphLink	expr	string	Returns path to graph view in the expression browser for the expression.
tableLink	expr	string	Returns path to tabular ("Table") view in the expression browser for the expression.
parseDuration	string	float	Parses a duration string such as "1h" into the number of seconds it represents.
stripDomain	string	string	Removes the domain part of a FQDN. Leaves port untouched.

Others

Name	Arguments	Returns	Notes
args	[]interface{}	map[string]interface{}	This converts a list of objects to a map with keys arg0, arg1 etc. This is intended to allow multiple arguments to be passed to templates.
tmpl	string, []interface{}	nothing	Like the built-in `template`, but allows non-literals as the template name. Note that the result is assumed to be safe, and will not be auto-escaped. Only available in consoles.
safeHtml	string	string	Marks string as HTML not requiring auto-escaping.

Prometheus

			externally reachable.
pathPrefix	*none*	string	The external URL path for use in console templates.

Template type differences

Each of the types of templates provide different information that can be used to parameterize templates, and have a few other differences.

Alert field templates

`.Value` , `.Labels` , `.ExternalLabels` , and `.ExternalURL` contain the alert value, the alert labels, the globally configured external labels, and the external URL (configured with `--web.external-url`) respectively. They are also exposed as the `$value` , `$labels` , `$externalLabels` , and `$externalURL` variables for convenience.

Console templates

Consoles are exposed on `/consoles/` , and sourced from the directory pointed to by the `-web.console.templates` flag.

Console templates are rendered with html/template, which provides auto-escaping. To bypass the auto-escaping use the `safe*` functions.,

URL parameters are available as a map in `.Params` . To access multiple URL parameters by the same name, `.RawParams` is a map of the list values for each parameter. The URL path is available in `.Path` , excluding the `/consoles/` prefix. The globally configured external labels are available as `.ExternalLabels` . There are also convenience variables for all four: `$rawParams` , `$params` , `$path` , and `$externalLabels` .

Consoles also have access to all the templates defined with `{{define "templateName"}}...{{end}}` found in `*.lib` files in the directory pointed to by the `-web.console.libraries` flag. As this is a shared namespace, take care to avoid clashes with other users. Template names beginning with `prom` , `_prom` , and `__` are reserved for use by Prometheus, as are the functions listed above.

Prometheus

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Unit Testing for Rules

You can use `promtool` to test your rules.

```
# For a single test file.
./promtool test rules test.yml

# If you have multiple test files, say test1.yml,test2.yml,test2.yml
./promtool test rules test1.yml test2.yml test3.yml
```

Test file format

```
# This is a list of rule files to consider for testing. Globs are supported.
rule_files:
  [ - <file_name> ]

[ evaluation_interval: <duration> | default = 1m ]

# The order in which group names are listed below will be the order of evaluation o
# rule groups (at a given evaluation time). The order is guaranteed only for the gr
# All the groups need not be mentioned below.
group_eval_order:
  [ - <group_name> ]

# All the tests are listed here.
tests:
  [ - <test_group> ]
```

```
# Series data
[ interval: <duration> | default = evaluation_interval ]
input_series:
  [ - <series> ]

# Name of the test group
[ name: <string> ]

# Unit tests for the above data.

# Unit tests for alerting rules. We consider the alerting rules from the input file
alert_rule_test:
  [ - <alert_test_case> ]

# Unit tests for PromQL expressions.
promql_expr_test:
  [ - <promql_test_case> ]

# External labels accessible to the alert template.
external_labels:
  [ <labelname>: <string> ... ]

# External URL accessible to the alert template.
# Usually set using --web.external-url.
  [ external_url: <string> ]
```

`<series>`

```
# This follows the usual series notation '<metric name>{<label name>=<label value>,
# Examples:
#      series_name{label1="value1", label2="value2"}
#      go_goroutines{job="prometheus", instance="localhost:9090"}
series: <string>

# This uses expanding notation.
# Expanding notation:
#     'a+bxn' becomes 'a a+b a+(2*b) a+(3*b) … a+(n*b)'
#     Read this as series starts at a, then n further samples incrementing by b.
```

Prometheus

```
# There are special values to indicate missing and stale samples:
#     '_' represents a missing sample from scrape
#     'stale' indicates a stale sample
# Examples:
#     1. '-2+4x3' becomes '-2 2 6 10' - series starts at -2, then 3 further samples
#     2. ' 1-2x4' becomes '1 -1 -3 -5 -7' - series starts at 1, then 4 further sampl
#     3. ' 1x4' becomes '1 1 1 1 1' - shorthand for '1+0x4', series starts at 1, the
#     4. ' 1 _x3 stale' becomes '1 _ _ _ stale' - the missing sample cannot incremen
#
# Native histogram notation:
#     Native histograms can be used instead of floating point numbers using the fol
#     {{schema:1 sum:-0.3 count:3.1 z_bucket:7.1 z_bucket_w:0.05 buckets:[5.1 10 7]
#     Native histograms support the same expanding notation as floating point numbe
#     All properties are optional and default to 0. The order is not important. The
#     - schema (int):
#         Currently valid schema numbers are -4 <= n <= 8. They are all for
#         base-2 bucket schemas, where 1 is a bucket boundary in each case, and
#         then each power of two is divided into 2^n logarithmic buckets.  Or
#         in other words, each bucket boundary is the previous boundary times
#         2^(2^-n).
#     - sum (float):
#         The sum of all observations, including the zero bucket.
#     - count (non-negative float):
#         The number of observations, including those that are NaN and including the
#     - z_bucket (non-negative float):
#         The sum of all observations in the zero bucket.
#     - z_bucket_w (non-negative float):
#         The width of the zero bucket.
#         If z_bucket_w > 0, the zero bucket contains all observations -z_bucket_w <
#         Otherwise, the zero bucket only contains observations that are exactly 0.
#     - buckets (list of non-negative floats):
#         Observation counts in positive buckets. Each represents an absolute count
#     - offset (int):
#         The starting index of the first entry in the positive buckets.
#     - n_buckets (list of non-negative floats):
#         Observation counts in negative buckets. Each represents an absolute count
#     - n_offset (int):
#         The starting index of the first entry in the negative buckets.
#     - counter_reset_hint (one of 'unknown', 'reset', 'not_reset' or 'gauge')
```

Prometheus

<alert_test_case>

Prometheus allows you to have same alertname for different alerting rules. Hence in this unit testing, you have to list the union of all the firing alerts for the alertname under a single `<alert_test_case>`.

```
# The time elapsed from time=0s when the alerts have to be checked.
eval_time: <duration>

# Name of the alert to be tested.
alertname: <string>

# List of expected alerts which are firing under the given alertname at
# given evaluation time. If you want to test if an alerting rule should
# not be firing, then you can mention the above fields and leave 'exp_alerts' empty
exp_alerts:
  [ - <alert> ]
```

<alert>

```
# These are the expanded labels and annotations of the expected alert.
# Note: labels also include the labels of the sample associated with the
# alert (same as what you see in `/alerts`, without series `__name__` and `alertname
exp_labels:
  [ <labelname>: <string> ]
exp_annotations:
  [ <labelname>: <string> ]
```

<promql_test_case>

```
# Expression to evaluate
expr: <string>
```

```
# Expected samples at the given evaluation time.
exp_samples:
  [ - <sample> ]
```

`<sample>`

```
# Labels of the sample in usual series notation '<metric name>{<label name>=<label v
# Examples:
#      series_name{label1="value1", label2="value2"}
#      go_goroutines{job="prometheus", instance="localhost:9090"}
labels: <string>

# The expected value of the PromQL expression.
value: <number>
```

Example

This is an example input file for unit testing which passes the test. `test.yml` is the test file which follows the syntax above and `alerts.yml` contains the alerting rules.

With `alerts.yml` in the same directory, run `./promtool test rules test.yml`.

`test.yml`

```
# This is the main input for unit testing.
# Only this file is passed as command line argument.

rule_files:
    - alerts.yml

evaluation_interval: 1m

tests:
    # Test 1.
    - interval: 1m
```

```
          values: '0 0 0 0 0 0 0 0 0 0 0 0 0 0 0'
        - series: 'up{job="node_exporter", instance="localhost:9100"}'
          values: '1+0x6 0 0 0 0 0 0 0 0' # 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
        - series: 'go_goroutines{job="prometheus", instance="localhost:9090"}'
          values: '10+10x2 30+20x5' # 10 20 30 30 50 70 90 110 130
        - series: 'go_goroutines{job="node_exporter", instance="localhost:9100"}'
          values: '10+10x7 10+30x4' # 10 20 30 40 50 60 70 80 10 40 70 100 130

    # Unit test for alerting rules.
    alert_rule_test:
        # Unit test 1.
        - eval_time: 10m
          alertname: InstanceDown
          exp_alerts:
              # Alert 1.
              - exp_labels:
                    severity: page
                    instance: localhost:9090
                    job: prometheus
                exp_annotations:
                    summary: "Instance localhost:9090 down"
                    description: "localhost:9090 of job prometheus has been down 
    # Unit tests for promql expressions.
    promql_expr_test:
        # Unit test 1.
        - expr: go_goroutines > 5
          eval_time: 4m
          exp_samples:
              # Sample 1.
              - labels: 'go_goroutines{job="prometheus",instance="localhost:9090"
                value: 50
              # Sample 2.
              - labels: 'go_goroutines{job="node_exporter",instance="localhost:91
                value: 50
```

alerts.yml

```yaml
groups:
- name: example
  rules:

  - alert: InstanceDown
    expr: up == 0
    for: 5m
    labels:
        severity: page
    annotations:
        summary: "Instance {{ $labels.instance }} down"
        description: "{{ $labels.instance }} of job {{ $labels.job }} has been down

  - alert: AnotherInstanceDown
    expr: up == 0
    for: 10m
    labels:
        severity: page
    annotations:
        summary: "Instance {{ $labels.instance }} down"
        description: "{{ $labels.instance }} of job {{ $labels.job }} has been down
```

Previous
Template reference

Edit

Next
HTTPS and authentication

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

HTTPS and authentication

Prometheus supports basic authentication and TLS. This is **experimental** and might change in the future.

To specify which web configuration file to load, use the `--web.config.file` flag.

The file is written in YAML format, defined by the scheme described below. Brackets indicate that a parameter is optional. For non-list parameters the value is set to the specified default.

The file is read upon every http request, such as any change in the configuration and the certificates is picked up immediately.

Generic placeholders are defined as follows:

- `<boolean>` : a boolean that can take the values `true` or `false`
- `<filename>` : a valid path in the current working directory
- `<secret>` : a regular string that is a secret, such as a password
- `<string>` : a regular string

A valid example file can be found here.

```
tls_server_config:
  # Certificate and key files for server to use to authenticate to client.
  cert_file: <filename>
  key_file: <filename>

  # Server policy for client authentication. Maps to ClientAuth Policies.
  # For more detail on clientAuth options:
  # https://golang.org/pkg/crypto/tls/#ClientAuthType
  #
```

```
    # CA certificate for client certificate authentication to the server.
    [ client_ca_file: <filename> ]

    # Verify that the client certificate has a Subject Alternate Name (SAN)
    # which is an exact match to an entry in this list, else terminate the
    # connection. SAN match can be one or multiple of the following: DNS,
    # IP, e-mail, or URI address from https://pkg.go.dev/crypto/x509#Certificate.
    [ client_allowed_sans:
      [ - <string> ] ]

    # Minimum TLS version that is acceptable.
    [ min_version: <string> | default = "TLS12" ]

    # Maximum TLS version that is acceptable.
    [ max_version: <string> | default = "TLS13" ]

    # List of supported cipher suites for TLS versions up to TLS 1.2. If empty,
    # Go default cipher suites are used. Available cipher suites are documented
    # in the go documentation:
    # https://golang.org/pkg/crypto/tls/#pkg-constants
    #
    # Note that only the cipher returned by the following function are supported:
    # https://pkg.go.dev/crypto/tls#CipherSuites
    [ cipher_suites:
      [ - <string> ] ]

    # prefer_server_cipher_suites controls whether the server selects the
    # client's most preferred ciphersuite, or the server's most preferred
    # ciphersuite. If true then the server's preference, as expressed in
    # the order of elements in cipher_suites, is used.
    [ prefer_server_cipher_suites: <boolean> | default = true ]

    # Elliptic curves that will be used in an ECDHE handshake, in preference
    # order. Available curves are documented in the go documentation:
    # https://golang.org/pkg/crypto/tls/#CurveID
    [ curve_preferences:
      [ - <string> ] ]

  http_server_config:
    # Enable HTTP/2 support. Note that HTTP/2 is only supported with TLS.
    [ headers:
      # Set the Content-Security-Policy header to HTTP responses.
      # Unset if blank.
      [ Content-Security-Policy: <string> ]
      # Set the X-Frame-Options header to HTTP responses.
      # Unset if blank. Accepted values are deny and sameorigin.
      # https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options
      [ X-Frame-Options: <string> ]
      # Set the X-Content-Type-Options header to HTTP responses.
      # Unset if blank. Accepted value is nosniff.
      # https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Content-Type-Optic
      [ X-Content-Type-Options: <string> ]
      # Set the X-XSS-Protection header to all responses.
      # Unset if blank.
      # https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection
      [ X-XSS-Protection: <string> ]
      # Set the Strict-Transport-Security header to HTTP responses.
      # Unset if blank.
      # Please make sure that you use this with care as this header might force
      # browsers to load Prometheus and the other applications hosted on the same
      # domain and subdomains over HTTPS.
      # https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Se
      [ Strict-Transport-Security: <string> ] ]

  # Usernames and hashed passwords that have full access to the web
  # server via basic authentication. If empty, no basic authentication is
  # required. Passwords are hashed with bcrypt.
  basic_auth_users:
    [ <string>: <secret> ... ]
```

Prometheus

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Querying Prometheus

Prometheus provides a functional query language called PromQL (Prometheus Query Language) that lets the user select and aggregate time series data in real time.

When you send a query request to Prometheus, it can be an *instant query*, evaluated at one point in time, or a *range query* at equally-spaced steps between a start and an end time. PromQL works exactly the same in each cases; the range query is just like an instant query run multiple times at different timestamps.

In the Prometheus UI, the "Table" tab is for instant queries and the "Graph" tab is for range queries.

Other programs can fetch the result of a PromQL expression via the HTTP API.

Examples

This document is a Prometheus basic language reference. For learning, it may be easier to start with a couple of examples.

Expression language data types

In Prometheus's expression language, an expression or sub-expression can evaluate to one of four types:

- **Instant vector** - a set of time series containing a single sample for each time series, all sharing the same timestamp
- **Range vector** - a set of time series containing a range of data points over time for each time series
- **Scalar** - a simple numeric floating point value

only some of these types are legal as the result of a user-specified expression. For instant queries, any of the above data types are allowed as the root of the expression. Range queries only support scalar-typed and instant-vector-typed expressions.

Notes about the experimental native histograms:

- Ingesting native histograms has to be enabled via a feature flag.
- Once native histograms have been ingested into the TSDB (and even after disabling the feature flag again), both instant vectors and range vectors may now contain samples that aren't simple floating point numbers (float samples) but complete histograms (histogram samples). A vector may contain a mix of float samples and histogram samples.

Literals

String literals

String literals are designated by single quotes, double quotes or backticks.

PromQL follows the same escaping rules as Go. For string literals in single or double quotes, a backslash begins an escape sequence, which may be followed by `a`, `b`, `f`, `n`, `r`, `t`, `v` or `\`. Specific characters can be provided using octal (`\nnn`) or hexadecimal (`\xnn`, `\unnnn` and `\Unnnnnnnn`) notations.

Conversely, escape characters are not parsed in string literals designated by backticks. It is important to note that, unlike Go, Prometheus does not discard newlines inside backticks.

Example:

```
"this is a string"
'these are unescaped: \n \\ \t'
`these are not unescaped: \n ' " \t`
```

Float literals and time durations

Scalar float values can be written as literal integer or floating-point numbers in the format (whitespace only included for better readability):

```
    | 0[xX][0-9a-fA-F]+
    | [nN][aA][nN]
    | [iI][nN][fF]
)
```

Examples:

```
23
-2.43
3.4e-9
0x8f
-Inf
NaN
```

Additionally, underscores (_) can be used in between decimal or hexadecimal digits to improve readability.

Examples:

```
1_000_000
.123_456_789
0x_53_AB_F3_82
```

Float literals are also used to specify durations in seconds. For convenience, decimal integer numbers may be combined with the following time units:

- `ms` – milliseconds
- `s` – seconds – 1s equals 1000ms
- `m` – minutes – 1m equals 60s (ignoring leap seconds)
- `h` – hours – 1h equals 60m
- `d` – days – 1d equals 24h (ignoring so-called daylight saving time)
- `w` – weeks – 1w equals 7d
- `y` – years – 1y equals 365d (ignoring leap days)

Suffixing a decimal integer number with one of the units above is a different representation of the equivalent number of seconds as a bare float literal.

Examples:

```
1ms # Equivalent to 0.001.
-2h # Equivalent to -7200.
```

The following examples do *not* work:

```
0xABm # No suffixing of hexadecimal numbers.
1.5h # Time units cannot be combined with a floating point.
+Infd # No suffixing of ±Inf or NaN.
```

Multiple units can be combined by concatenation of suffixed integers. Units must be ordered from the longest to the shortest. A given unit must only appear once per float literal.

Examples:

```
1h30m # Equivalent to 5400s and thus 5400.
12h34m56s # Equivalent to 45296s and thus 45296.
54s321ms # Equivalent to 54.321.
```

Time series selectors

These are the basic building-blocks that instruct PromQL what data to fetch.

Instant vector selectors

Instant vector selectors allow the selection of a set of time series and a single sample value for each at a given timestamp (point in time). In the simplest form, only a metric name is specified, which results in an instant vector containing elements for all time series that have this metric name.

The value returned will be that of the most recent sample at or before the query's evaluation timestamp (in the case of an instant query) or the current step within the query (in the case of a range query). The `@` modifier allows overriding the timestamp relative to which the selection takes place. Time series are only returned if their most recent sample is less than the lookback period ago.

This example selects all time series that have the `http_requests_total` metric name, returning the most recent sample for each:

It is possible to filter these time series further by appending a comma-separated list of label matchers in curly braces (`{}`).

This example selects only those time series with the `http_requests_total` metric name that also have the `job` label set to `prometheus` and their `group` label set to `canary` :

```
http_requests_total{job="prometheus",group="canary"}
```

It is also possible to negatively match a label value, or to match label values against regular expressions. The following label matching operators exist:

- `=` : Select labels that are exactly equal to the provided string.
- `!=` : Select labels that are not equal to the provided string.
- `=~` : Select labels that regex-match the provided string.
- `!~` : Select labels that do not regex-match the provided string.

Regex matches are fully anchored. A match of `env=~"foo"` is treated as `env=~"^foo$"` .

For example, this selects all `http_requests_total` time series for `staging` , `testing` , and `development` environments and HTTP methods other than `GET` .

```
http_requests_total{environment=~"staging|testing|development",method!="GET"}
```

Label matchers that match empty label values also select all time series that do not have the specific label set at all. It is possible to have multiple matchers for the same label name.

For example, given the dataset:

```
http_requests_total
http_requests_total{replica="rep-a"}
http_requests_total{replica="rep-b"}
http_requests_total{environment="development"}
```

The query `http_requests_total{environment=""}` would match and return:

```
http_requests_total
http_requests_total{replica="rep-a"}
http_requests_total{replica="rep-b"}
```

Multiple matchers can be used for the same label name; they all must pass for a result to be returned.

The query:

```
http_requests_total{replica!="rep-a",replica=~"rep.*"}
```

Would then match:

```
http_requests_total{replica="rep-b"}
```

Vector selectors must either specify a name or at least one label matcher that does not match the empty string. The following expression is illegal:

```
{job=~".*"} # Bad!
```

In contrast, these expressions are valid as they both have a selector that does not match empty label values.

```
{job=~".+"}              # Good!
{job=~".*",method="get"} # Good!
```

Label matchers can also be applied to metric names by matching against the internal `__name__` label. For example, the expression `http_requests_total` is equivalent to `{__name__="http_requests_total"}`. Matchers other than `=` (`!=`, `=~`, `!~`) may also be used. The following expression selects all metrics that have a name starting with `job:`:

```
{__name__=~"job:.*"}
```

The metric name must not be one of the keywords `bool`, `on`, `ignoring`, `group_left` and `group_right`. The following expression is illegal:

```
on{} # Bad!
```

A workaround for this restriction is to use the `__name__` label:

Range Vector Selectors

Range vector literals work like instant vector literals, except that they select a range of samples back from the current instant. Syntactically, a float literal is appended in square brackets (`[]`) at the end of a vector selector to specify for how many seconds back in time values should be fetched for each resulting range vector element. Commonly, the float literal uses the syntax with one or more time units, e.g. `[5m]` . The range is a left-open and right-closed interval, i.e. samples with timestamps coinciding with the left boundary of the range are excluded from the selection, while samples coinciding with the right boundary of the range are included in the selection.

In this example, we select all the values recorded less than 5m ago for all time series that have the metric name `http_requests_total` and a `job` label set to `prometheus` :

```
http_requests_total{job="prometheus"}[5m]
```

Offset modifier

The `offset` modifier allows changing the time offset for individual instant and range vectors in a query.

For example, the following expression returns the value of `http_requests_total` 5 minutes in the past relative to the current query evaluation time:

```
http_requests_total offset 5m
```

Note that the `offset` modifier always needs to follow the selector immediately, i.e. the following would be correct:

```
sum(http_requests_total{method="GET"} offset 5m) // GOOD.
```

While the following would be *incorrect*:

```
sum(http_requests_total{method="GET"}) offset 5m // INVALID.
```

```
rate(http_requests_total[5m] offset 1w)
```

When querying for samples in the past, a negative offset will enable temporal comparisons forward in time:

```
rate(http_requests_total[5m] offset -1w)
```

Note that this allows a query to look ahead of its evaluation time.

@ modifier

The `@` modifier allows changing the evaluation time for individual instant and range vectors in a query. The time supplied to the `@` modifier is a unix timestamp and described with a float literal.

For example, the following expression returns the value of `http_requests_total` at `2021-01-04T07:40:00+00:00`:

```
http_requests_total @ 1609746000
```

Note that the `@` modifier always needs to follow the selector immediately, i.e. the following would be correct:

```
sum(http_requests_total{method="GET"} @ 1609746000) // GOOD.
```

While the following would be *incorrect*:

```
sum(http_requests_total{method="GET"}) @ 1609746000 // INVALID.
```

The same works for range vectors. This returns the 5-minute rate that `http_requests_total` had at `2021-01-04T07:40:00+00:00`:

```
rate(http_requests_total[5m] @ 1609746000)
```

The `@` modifier supports all representations of numeric literals described above. It works with the `offset` modifier where the offset is applied relative to the `@` modifier time. The

```
# offset after @
http_requests_total @ 1609746000 offset 5m
# offset before @
http_requests_total offset 5m @ 1609746000
```

Additionally, `start()` and `end()` can also be used as values for the `@` modifier as special values.

For a range query, they resolve to the start and end of the range query respectively and remain the same for all steps.

For an instant query, `start()` and `end()` both resolve to the evaluation time.

```
http_requests_total @ start()
rate(http_requests_total[5m] @ end())
```

Note that the `@` modifier allows a query to look ahead of its evaluation time.

Subquery

Subquery allows you to run an instant query for a given range and resolution. The result of a subquery is a range vector.

Syntax: `<instant_query> '[' <range> ':' [<resolution>] ']' [ @ <float_literal> ] [ offset <float_literal> ]`

- `<resolution>` is optional. Default is the global evaluation interval.

Operators

Prometheus supports many binary and aggregation operators. These are described in detail in the expression language operators page.

Functions

Prometheus supports several functions to operate on data. These are described in detail in the expression language functions page.

```
    # This is a comment
```

Regular expressions

All regular expressions in Prometheus use RE2 syntax.

Regex matches are always fully anchored.

Gotchas

Staleness

The timestamps at which to sample data, during a query, are selected independently of the actual present time series data. This is mainly to support cases like aggregation (`sum` , `avg` , and so on), where multiple aggregated time series do not precisely align in time. Because of their independence, Prometheus needs to assign a value at those timestamps for each relevant time series. It does so by taking the newest sample that is less than the lookback period ago. The lookback period is 5 minutes by default, but can be set with the `--query.lookback-delta` flag

If a target scrape or rule evaluation no longer returns a sample for a time series that was previously present, this time series will be marked as stale. If a target is removed, the previously retrieved time series will be marked as stale soon after removal.

If a query is evaluated at a sampling timestamp after a time series is marked as stale, then no value is returned for that time series. If new samples are subsequently ingested for that time series, they will be returned as expected.

A time series will go stale when it is no longer exported, or the target no longer exists. Such time series will disappear from graphs at the times of their latest collected sample, and they will not be returned in queries after they are marked stale.

Some exporters, which put their own timestamps on samples, get a different behaviour: series that stop being exported take the last value for (by default) 5 minutes before disappearing. The `track_timestamps_staleness` setting can change this.

overload the server or browser. Thus, when constructing queries over unknown data, always start building the query in the tabular view of Prometheus's expression browser until the result set seems reasonable (hundreds, not thousands, of time series at most). Only when you have filtered or aggregated your data sufficiently, switch to graph mode. If the expression still takes too long to graph ad-hoc, pre-record it via a recording rule.

This is especially relevant for Prometheus's query language, where a bare metric name selector like `api_http_requests_total` could expand to thousands of time series with different labels. Also, keep in mind that expressions that aggregate over many time series will generate load on the server even if the output is only a small number of time series. This is similar to how it would be slow to sum all values of a column in a relational database, even if the output value is only a single number.

← **Previous** HTTPS and authentication

Edit

Next Operators →

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Operators

Binary operators

Prometheus's query language supports basic logical and arithmetic operators. For operations between two instant vectors, the matching behavior can be modified.

Arithmetic binary operators

The following binary arithmetic operators exist in Prometheus:

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `/` (division)
- `%` (modulo)
- `^` (power/exponentiation)

Binary arithmetic operators are defined between scalar/scalar, vector/scalar, and vector/vector value pairs.

Between two scalars, the behavior is obvious: they evaluate to another scalar that is the result of the operator applied to both scalar operands.

Between an instant vector and a scalar, the operator is applied to the value of every data sample in the vector. E.g. if a time series instant vector is multiplied by 2, the result is another vector in which every sample value of the original vector is multiplied by 2. The metric name is dropped.

The metric name is dropped. Entries for which no matching entry in the right-hand vector can be found are not part of the result.

Trigonometric binary operators

The following trigonometric binary operators, which work in radians, exist in Prometheus:

- `atan2` (based on https://pkg.go.dev/math#Atan2)

Trigonometric operators allow trigonometric functions to be executed on two vectors using vector matching, which isn't available with normal functions. They act in the same manner as arithmetic operators.

Comparison binary operators

The following binary comparison operators exist in Prometheus:

- `==` (equal)
- `!=` (not-equal)
- `>` (greater-than)
- `<` (less-than)
- `>=` (greater-or-equal)
- `<=` (less-or-equal)

Comparison operators are defined between scalar/scalar, vector/scalar, and vector/vector value pairs. By default they filter. Their behavior can be modified by providing `bool` after the operator, which will return `0` or `1` for the value rather than filtering.

Between two scalars, the `bool` modifier must be provided and these operators result in another scalar that is either `0` (`false`) or `1` (`true`), depending on the comparison result.

Between an instant vector and a scalar, these operators are applied to the value of every data sample in the vector, and vector elements between which the comparison result is `false` get dropped from the result vector. If the `bool` modifier is provided, vector elements that would be dropped instead have the value `0` and vector elements that would be kept have the value `1`. The metric name is dropped if the `bool` modifier is provided.

Between two instant vectors, these operators behave as a filter by default, applied to matching entries. Vector elements for which the expression is not true or which do not find a match on the other side of the expression get dropped from the result, while the others are labels again becoming the output label set. The metric name is dropped if the `bool` modifier is provided.

Logical/set binary operators

These logical/set binary operators are only defined between instant vectors:

- `and` (intersection)
- `or` (union)
- `unless` (complement)

`vector1 and vector2` results in a vector consisting of the elements of `vector1` for which there are elements in `vector2` with exactly matching label sets. Other elements are dropped. The metric name and values are carried over from the left-hand side vector.

`vector1 or vector2` results in a vector that contains all original elements (label sets + values) of `vector1` and additionally all elements of `vector2` which do not have matching label sets in `vector1`.

`vector1 unless vector2` results in a vector consisting of the elements of `vector1` for which there are no elements in `vector2` with exactly matching label sets. All matching elements in both vectors are dropped.

Vector matching

Operations between vectors attempt to find a matching element in the right-hand side vector for each entry in the left-hand side. There are two basic types of matching behavior: One-to-one and many-to-one/one-to-many.

Vector matching keywords

These vector matching keywords allow for matching between series with different label sets providing:

- `on`
- `ignoring`

Label lists provided to matching keywords will determine how vectors are combined. Examples can be found in One-to-one vector matches and in Many-to-one and one-to-

Group modifiers

These group modifiers enable many-to-one/one-to-many vector matching:

- `group_left`
- `group_right`

Label lists can be provided to the group modifier which contain labels from the "one"-side to be included in the result metrics.

Many-to-one and one-to-many matching are advanced use cases that should be carefully considered. Often a proper use of `ignoring(<labels>)` provides the desired outcome.

Grouping modifiers can only be used for comparison and arithmetic. Operations as `and`, `unless` and `or` operations match with all possible entries in the right vector by default.

One-to-one vector matches

One-to-one finds a unique pair of entries from each side of the operation. In the default case, that is an operation following the format `vector1 <operator> vector2`. Two entries match if they have the exact same set of labels and corresponding values. The `ignoring` keyword allows ignoring certain labels when matching, while the `on` keyword allows reducing the set of considered labels to a provided list:

```
<vector expr> <bin-op> ignoring(<label list>) <vector expr>
<vector expr> <bin-op> on(<label list>) <vector expr>
```

Example input:

```
method_code:http_errors:rate5m{method="get", code="500"}  24
method_code:http_errors:rate5m{method="get", code="404"}  30
method_code:http_errors:rate5m{method="put", code="501"}  3
method_code:http_errors:rate5m{method="post", code="500"} 6
method_code:http_errors:rate5m{method="post", code="404"} 21

method:http_requests:rate5m{method="get"}  600
method:http_requests:rate5m{method="del"}  34
method:http_requests:rate5m{method="post"} 120
```

This returns a result vector containing the fraction of HTTP requests with status code of 500 for each method, as measured over the last 5 minutes. Without `ignoring(code)` there would have been no match as the metrics do not share the same set of labels. The entries with methods `put` and `del` have no match and will not show up in the result:

```
{method="get"}  0.04            //  24 / 600
{method="post"} 0.05            //   6 / 120
```

Many-to-one and one-to-many vector matches

Many-to-one and **one-to-many** matchings refer to the case where each vector element on the "one"-side can match with multiple elements on the "many"-side. This has to be explicitly requested using the `group_left` or `group_right` modifiers, where left/right determines which vector has the higher cardinality.

```
<vector expr> <bin-op> ignoring(<label list>) group_left(<label list>) <vector expr
<vector expr> <bin-op> ignoring(<label list>) group_right(<label list>) <vector expr
<vector expr> <bin-op> on(<label list>) group_left(<label list>) <vector expr>
<vector expr> <bin-op> on(<label list>) group_right(<label list>) <vector expr>
```

The label list provided with the group modifier contains additional labels from the "one"-side to be included in the result metrics. For `on` a label can only appear in one of the lists. Every time series of the result vector must be uniquely identifiable.

Example query:

```
method_code:http_errors:rate5m / ignoring(code) group_left method:http_requests:rate
```

In this case the left vector contains more than one entry per `method` label value. Thus, we indicate this using `group_left`. The elements from the right side are now matched with multiple elements with the same `method` label on the left:

```
{method="get", code="500"}  0.04            //  24 / 600
{method="get", code="404"}  0.05            //  30 / 600
```

Prometheus

Aggregation operators

Prometheus supports the following built-in aggregation operators that can be used to aggregate the elements of a single instant vector, resulting in a new vector of fewer elements with aggregated values:

- `sum` (calculate sum over dimensions)
- `min` (select minimum over dimensions)
- `max` (select maximum over dimensions)
- `avg` (calculate the average over dimensions)
- `group` (all values in the resulting vector are 1)
- `stddev` (calculate population standard deviation over dimensions)
- `stdvar` (calculate population standard variance over dimensions)
- `count` (count number of elements in the vector)
- `count_values` (count number of elements with the same value)
- `bottomk` (smallest k elements by sample value)
- `topk` (largest k elements by sample value)
- `quantile` (calculate φ-quantile (0 ≤ φ ≤ 1) over dimensions)
- `limitk` (sample n elements)
- `limit_ratio` (sample elements with approximately *r* ratio if `r > 0`, and the complement of such samples if `r = -(1.0 - r)`)

These operators can either be used to aggregate over **all** label dimensions or preserve distinct dimensions by including a `without` or `by` clause. These clauses may be used before or after the expression.

```
<aggr-op> [without|by (<label list>)] ([parameter,] <vector expression>)
```

or

```
<aggr-op>([parameter,] <vector expression>) [without|by (<label list>)]
```

`label list` is a list of unquoted labels that may include a trailing comma, i.e. both `(label1, label2)` and `(label1, label2,)` are valid syntax.

`without` removes the listed labels from the result vector, while all other labels are preserved in the output. `by` does the opposite and drops labels that are not listed in the `by` clause, `limit_ratio`.

`count_values` outputs one time series per unique sample value. Each series has an additional label. The name of that label is given by the aggregation parameter, and the label value is the unique sample value. The value of each time series is the number of times that sample value was present.

`topk` and `bottomk` are different from other aggregators in that a subset of the input samples, including the original labels, are returned in the result vector. `by` and `without` are only used to bucket the input vector.

`limitk` and `limit_ratio` also return a subset of the input samples, including the original labels in the result vector, these are experimental operators that must be enabled with `--enable-feature=promql-experimental-functions`.

`quantile` calculates the φ-quantile, the value that ranks at number φ*N among the N metric values of the dimensions aggregated over. φ is provided as the aggregation parameter. For example, `quantile(0.5, ...)` calculates the median, `quantile(0.95, ...)` the 95th percentile. For φ = `NaN`, `NaN` is returned. For φ < 0, `-Inf` is returned. For φ > 1, `+Inf` is returned.

Example:

If the metric `http_requests_total` had time series that fan out by `application`, `instance`, and `group` labels, we could calculate the total number of seen HTTP requests per application and group over all instances via:

```
sum without (instance) (http_requests_total)
```

Which is equivalent to:

```
sum by (application, group) (http_requests_total)
```

If we are just interested in the total of HTTP requests we have seen in **all** applications, we could simply write:

```
sum(http_requests_total)
```

To count the number of binaries running each build version we could write:

To get the 5 largest HTTP requests counts across all instances we could write:

```
topk(5, http_requests_total)
```

To sample 10 timeseries, for example to inspect labels and their values, we could write:

```
limitk(10, http_requests_total)
```

To deterministically sample approximately 10% of timeseries we could write:

```
limit_ratio(0.1, http_requests_total)
```

Given that `limit_ratio()` implements a deterministic sampling algorithm (based on labels' hash), you can get the *complement* of the above samples, i.e. approximately 90%, but precisely those not returned by `limit_ratio(0.1, ...)` with:

```
limit_ratio(-0.9, http_requests_total)
```

You can also use this feature to e.g. verify that `avg()` is a representative aggregation for your samples' values, by checking that the difference between averaging two samples' subsets is "small" when compared to the standard deviation.

```
abs(
  avg(limit_ratio(0.5, http_requests_total))
  -
  avg(limit_ratio(-0.5, http_requests_total))
) <= bool stddev(http_requests_total)
```

Binary operator precedence

The following list shows the precedence of binary operators in Prometheus, from highest to lowest.

1. `^`
2. `*`, `/`, `%`, `atan2`
3. `+`, `-`

Operators on the same precedence level are left-associative. For example, `2 * 3 % 2` is equivalent to `(2 * 3) % 2`. However `^` is right associative, so `2 ^ 3 ^ 2` is equivalent to `2 ^ (3 ^ 2)`.

Operators for native histograms

Native histograms are an experimental feature. Ingesting native histograms has to be enabled via a feature flag. Once native histograms have been ingested, they can be queried (even after the feature flag has been disabled again). However, the operator support for native histograms is still very limited.

Logical/set binary operators work as expected even if histogram samples are involved. They only check for the existence of a vector element and don't change their behavior depending on the sample type of an element (float or histogram). The `count` aggregation operator works similarly.

The binary `+` and `-` operators between two native histograms and the `sum` and `avg` aggregation operators to aggregate native histograms are fully supported. Even if the histograms involved have different bucket layouts, the buckets are automatically converted appropriately so that the operation can be performed. (With the currently supported bucket schemas, that's always possible.) If either operator has to aggregate a mix of histogram samples and float samples, the corresponding vector element is removed from the output vector entirely.

The binary `*` operator works between a native histogram and a float in any order, while the binary `/` operator can be used between a native histogram and a float in that exact order.

All other operators (and unmentioned cases for the above operators) do not behave in a meaningful way. They either treat the histogram sample as if it were a float sample of value 0, or (in case of arithmetic operations between a scalar and a vector) they leave the histogram sample unchanged. This behavior will change to a meaningful one before native histograms are a stable feature.

← **Previous** Basics

Edit

Next Functions →

Prometheus

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Functions

Some functions have default arguments, e.g. `year(v=vector(time()) instant-vector)`. This means that there is one argument `v` which is an instant vector, which if not provided it will default to the value of the expression `vector(time())`.

Notes about the experimental native histograms:

- Ingesting native histograms has to be enabled via a feature flag. As long as no native histograms have been ingested into the TSDB, all functions will behave as usual.
- Functions that do not explicitly mention native histograms in their documentation (see below) will ignore histogram samples.
- Functions that do already act on native histograms might still change their behavior in the future.
- If a function requires the same bucket layout between multiple native histograms it acts on, it will automatically convert them appropriately. (With the currently supported bucket schemas, that's always possible.)

`abs()`

`abs(v instant-vector)` returns the input vector with all sample values converted to their absolute value.

`absent()`

`absent(v instant-vector)` returns an empty vector if the vector passed to it has any elements (floats or native histograms) and a 1-element vector with the value 1 if the vector passed to it has no elements.

```
absent(nonexistent{job="myjob"})
# => {job="myjob"}

absent(nonexistent{job="myjob",instance=~".*"})
# => {job="myjob"}

absent(sum(nonexistent{job="myjob"}))
# => {}
```

In the first two examples, `absent()` tries to be smart about deriving labels of the 1-element output vector from the input vector.

`absent_over_time()`

`absent_over_time(v range-vector)` returns an empty vector if the range vector passed to it has any elements (floats or native histograms) and a 1-element vector with the value 1 if the range vector passed to it has no elements.

This is useful for alerting on when no time series exist for a given metric name and label combination for a certain amount of time.

```
absent_over_time(nonexistent{job="myjob"}[1h])
# => {job="myjob"}

absent_over_time(nonexistent{job="myjob",instance=~".*"}[1h])
# => {job="myjob"}

absent_over_time(sum(nonexistent{job="myjob"})[1h:])
# => {}
```

In the first two examples, `absent_over_time()` tries to be smart about deriving labels of the 1-element output vector from the input vector.

`ceil()`

`ceil(v instant-vector)` rounds the sample values of all elements in `v` up to the nearest integer value greater than or equal to v.

- `ceil(1.78) = 2.0`

`changes()`

For each input time series, `changes(v range-vector)` returns the number of times its value has changed within the provided time range as an instant vector.

`clamp()`

`clamp(v instant-vector, min scalar, max scalar)` clamps the sample values of all elements in `v` to have a lower limit of `min` and an upper limit of `max`.

Special cases:

- Return an empty vector if `min > max`
- Return `NaN` if `min` or `max` is `NaN`

`clamp_max()`

`clamp_max(v instant-vector, max scalar)` clamps the sample values of all elements in `v` to have an upper limit of `max`.

`clamp_min()`

`clamp_min(v instant-vector, min scalar)` clamps the sample values of all elements in `v` to have a lower limit of `min`.

`day_of_month()`

`day_of_month(v=vector(time()) instant-vector)` returns the day of the month for each of the given times in UTC. Returned values are from 1 to 31.

`day_of_week()`

Prometheus

`day_of_year()`

`day_of_year(v=vector(time()) instant-vector)` returns the day of the year for each of the given times in UTC. Returned values are from 1 to 365 for non-leap years, and 1 to 366 in leap years.

`days_in_month()`

`days_in_month(v=vector(time()) instant-vector)` returns number of days in the month for each of the given times in UTC. Returned values are from 28 to 31.

`delta()`

`delta(v range-vector)` calculates the difference between the first and last value of each time series element in a range vector `v`, returning an instant vector with the given deltas and equivalent labels. The delta is extrapolated to cover the full time range as specified in the range vector selector, so that it is possible to get a non-integer result even if the sample values are all integers.

The following example expression returns the difference in CPU temperature between now and 2 hours ago:

```
delta(cpu_temp_celsius{host="zeus"}[2h])
```

`delta` acts on native histograms by calculating a new histogram where each component (sum and count of observations, buckets) is the difference between the respective component in the first and last native histogram in `v`. However, each element in `v` that contains a mix of float and native histogram samples within the range, will be missing from the result vector.

`delta` should only be used with gauges and native histograms where the components behave like gauges (so-called gauge histograms).

`deriv()`

slope and offset value calculated will be `NaN`.

`deriv` should only be used with gauges.

`exp()`

`exp(v instant-vector)` calculates the exponential function for all elements in `v`. Special cases are:

- `Exp(+Inf) = +Inf`
- `Exp(NaN) = NaN`

`floor()`

`floor(v instant-vector)` rounds the sample values of all elements in `v` down to the nearest integer value smaller than or equal to v.

- `floor(+Inf) = +Inf`
- `floor(±0) = ±0`
- `floor(1.49) = 1.0`
- `floor(1.78) = 1.0`

`histogram_avg()`

This function only acts on native histograms, which are an experimental feature. The behavior of this function may change in future versions of Prometheus, including its removal from PromQL.

`histogram_avg(v instant-vector)` returns the arithmetic average of observed values stored in a native histogram. Samples that are not native histograms are ignored and do not show up in the returned vector.

Use `histogram_avg` as demonstrated below to compute the average request duration over a 5-minute window from a native histogram:

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

Which is equivalent to the following query:

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

`histogram_count()` and `histogram_sum()`

Both functions only act on native histograms, which are an experimental feature. The behavior of these functions may change in future versions of Prometheus, including their removal from PromQL.

`histogram_count(v instant-vector)` returns the count of observations stored in a native histogram. Samples that are not native histograms are ignored and do not show up in the returned vector.

Similarly, `histogram_sum(v instant-vector)` returns the sum of observations stored in a native histogram.

Use `histogram_count` in the following way to calculate a rate of observations (in this case corresponding to "requests per second") from a native histogram:

```
histogram_count(rate(http_request_duration_seconds[10m]))
```

`histogram_fraction()`

This function only acts on native histograms, which are an experimental feature. The behavior of this function may change in future versions of Prometheus, including its removal from PromQL.

For a native histogram, `histogram_fraction(lower scalar, upper scalar, v instant-vector)` returns the estimated fraction of observations between the provided lower and upper values. Samples that are not native histograms are ignored and do not show up in the returned vector.

For example, the following expression calculates the fraction of HTTP requests over the last hour that took 200ms or less:

```
histogram_fraction(0, 0.2, rate(http_request_duration_seconds[1h]))
```

The error of the estimation depends on the resolution of the underlying native histogram and how closely the provided boundaries are aligned with the bucket boundaries in the above included negative observations (which shouldn't be the case for request durations), the appropriate lower boundary to include all observations less than or equal 0.2 would be `-Inf` rather than `0`.

Whether the provided boundaries are inclusive or exclusive is only relevant if the provided boundaries are precisely aligned with bucket boundaries in the underlying native histogram. In this case, the behavior depends on the schema definition of the histogram. The currently supported schemas all feature inclusive upper boundaries and exclusive lower boundaries for positive values (and vice versa for negative values). Without a precise alignment of boundaries, the function uses linear interpolation to estimate the fraction. With the resulting uncertainty, it becomes irrelevant if the boundaries are inclusive or exclusive.

`histogram_quantile()`

`histogram_quantile(φ scalar, b instant-vector)` calculates the φ-quantile (0 ≤ φ ≤ 1) from a classic histogram or from a native histogram. (See histograms and summaries for a detailed explanation of φ-quantiles and the usage of the (classic) histogram metric type in general.)

Note that native histograms are an experimental feature. The behavior of this function when dealing with native histograms may change in future versions of Prometheus.

The float samples in `b` are considered the counts of observations in each bucket of one or more classic histograms. Each float sample must have a label `le` where the label value denotes the inclusive upper bound of the bucket. (Float samples without such a label are silently ignored.) The other labels and the metric name are used to identify the buckets belonging to each classic histogram. The histogram metric type automatically provides time series with the `_bucket` suffix and the appropriate labels.

The native histogram samples in `b` are treated each individually as a separate histogram to calculate the quantile from.

As long as no naming collisions arise, `b` may contain a mix of classic and native histograms.

Use the `rate()` function to specify the time window for the quantile calculation.

Example: A histogram metric is called `http_request_duration_seconds` (and therefore the metric name for the buckets of a classic histogram is `http_request_duration_seconds_bucket`). To calculate the 90th percentile of request durations over the last 10m, use the following expression in case `http_request_duration_seconds` is a classic histogram:

For a native histogram, use the following expression instead:

```
histogram_quantile(0.9, rate(http_request_duration_seconds[10m]))
```

The quantile is calculated for each label combination in `http_request_duration_seconds`. To aggregate, use the `sum()` aggregator around the `rate()` function. Since the `le` label is required by `histogram_quantile()` to deal with classic histograms, it has to be included in the `by` clause. The following expression aggregates the 90th percentile by `job` for classic histograms:

```
histogram_quantile(0.9, sum by (job, le) (rate(http_request_duration_seconds_bucket
```

When aggregating native histograms, the expression simplifies to:

```
histogram_quantile(0.9, sum by (job) (rate(http_request_duration_seconds[10m])))
```

To aggregate all classic histograms, specify only the `le` label:

```
histogram_quantile(0.9, sum by (le) (rate(http_request_duration_seconds_bucket[10m]
```

With native histograms, aggregating everything works as usual without any `by` clause:

```
histogram_quantile(0.9, sum(rate(http_request_duration_seconds[10m])))
```

In the (common) case that a quantile value does not coincide with a bucket boundary, the `histogram_quantile()` function interpolates the quantile value within the bucket the quantile value falls into. For classic histograms, for native histograms with custom bucket boundaries, and for the zero bucket of other native histograms, it assumes a uniform distribution of observations within the bucket (also called *linear interpolation*). For the non-zero-buckets of native histograms with a standard exponential bucketing schema, the interpolation is done under the assumption that the samples within the bucket are distributed in a way that they would uniformly populate the buckets in a hypothetical histogram with higher resolution. (This is also called *exponential interpolation*.)

If `b` has 0 observations, `NaN` is returned. For φ < 0, `-Inf` is returned. For φ > 1, `+Inf` is returned. For φ = `NaN`, `NaN` is returned.

- The highest bucket must have an upper bound of `+Inf`. (Otherwise, `NaN` is returned.)
- If a quantile is located in the highest bucket, the upper bound of the second highest bucket is returned.
- The lower limit of the lowest bucket is assumed to be 0 if the upper bound of that bucket is greater than 0. In that case, the usual linear interpolation is applied within that bucket. Otherwise, the upper bound of the lowest bucket is returned for quantiles located in the lowest bucket.

Special cases for native histograms (relevant for the exact interpolation happening within the zero bucket):

- A zero bucket with finite width is assumed to contain no negative observations if the histogram has observations in positive buckets, but none in negative buckets.
- A zero bucket with finite width is assumed to contain no positive observations if the histogram has observations in negative buckets, but none in positive buckets.

You can use `histogram_quantile(0, v instant-vector)` to get the estimated minimum value stored in a histogram.

You can use `histogram_quantile(1, v instant-vector)` to get the estimated maximum value stored in a histogram.

Buckets of classic histograms are cumulative. Therefore, the following should always be the case:

- The counts in the buckets are monotonically increasing (strictly non-decreasing).
- A lack of observations between the upper limits of two consecutive buckets results in equal counts in those two buckets.

However, floating point precision issues (e.g. small discrepancies introduced by computing of buckets with `sum(rate(...))`) or invalid data might violate these assumptions. In that case, `histogram_quantile` would be unable to return meaningful results. To mitigate the issue, `histogram_quantile` assumes that tiny relative differences between consecutive buckets are happening because of floating point precision errors and ignores them. (The threshold to ignore a difference between two buckets is a trillionth (1e-12) of the sum of both buckets.) Furthermore, if there are non-monotonic bucket counts even after this adjustment, they are increased to the value of the previous buckets to enforce monotonicity. The latter is evidence for an actual issue with the input data and is therefore flagged with an informational annotation reading `input to histogram_quantile needed to be fixed for monotonicity`. If you encounter this annotation, you should find and remove the source of the invalid data.

Prometheus

of these functions may change in future versions of Prometheus, including their removal from PromQL.

`histogram_stddev(v instant-vector)` returns the estimated standard deviation of observations in a native histogram, based on the geometric mean of the buckets where the observations lie. Samples that are not native histograms are ignored and do not show up in the returned vector.

Similarly, `histogram_stdvar(v instant-vector)` returns the estimated standard variance of observations in a native histogram.

`double_exponential_smoothing()`

This function has to be enabled via the feature flag `--enable-feature=promql-experimental-functions`.

`double_exponential_smoothing(v range-vector, sf scalar, tf scalar)` produces a smoothed value for time series based on the range in `v`. The lower the smoothing factor `sf`, the more importance is given to old data. The higher the trend factor `tf`, the more trends in the data is considered. Both `sf` and `tf` must be between 0 and 1. For additional details, refer to NIST Engineering Statistics Handbook. In Prometheus V2 this function was called `holt_winters`. This caused confusion since the Holt-Winters method usually refers to triple exponential smoothing. Double exponential smoothing as implemented here is also referred to as "Holt Linear".

`double_exponential_smoothing` should only be used with gauges.

`hour()`

`hour(v=vector(time()) instant-vector)` returns the hour of the day for each of the given times in UTC. Returned values are from 0 to 23.

`idelta()`

`idelta(v range-vector)` calculates the difference between the last two samples in the range vector `v`, returning an instant vector with the given deltas and equivalent labels.

`idelta` should only be used with gauges.

Breaks in monotonicity (such as counter resets due to target restarts) are automatically adjusted for. The increase is extrapolated to cover the full time range as specified in the range vector selector, so that it is possible to get a non-integer result even if a counter increases only by integer increments.

The following example expression returns the number of HTTP requests as measured over the last 5 minutes, per time series in the range vector:

```
increase(http_requests_total{job="api-server"}[5m])
```

`increase` acts on native histograms by calculating a new histogram where each component (sum and count of observations, buckets) is the increase between the respective component in the first and last native histogram in `v`. However, each element in `v` that contains a mix of float and native histogram samples within the range, will be missing from the result vector.

`increase` should only be used with counters and native histograms where the components behave like counters. It is syntactic sugar for `rate(v)` multiplied by the number of seconds under the specified time range window, and should be used primarily for human readability. Use `rate` in recording rules so that increases are tracked consistently on a per-second basis.

`info()` (experimental)

The `info` function is an experiment to improve UX around including labels from info metrics. The behavior of this function may change in future versions of Prometheus, including its removal from PromQL. `info` has to be enabled via the feature flag `--enable-feature=promql-experimental-functions`.

`info(v instant-vector, [data-label-selector instant-vector])` finds, for each time series in `v`, all info series with matching *identifying* labels (more on this later), and adds the union of their *data* (i.e., non-identifying) labels to the time series. The second argument `data-label-selector` is optional. It is not a real instant vector, but uses a subset of its syntax. It must start and end with curly braces (`{ ... }`) and may only contain label matchers. The label matchers are used to constrain which info series to consider and which data labels to add to `v`.

Identifying labels of an info series are the subset of labels that uniquely identify the info series. The remaining labels are considered *data labels* (also called non-identifying). (Note that Prometheus's concept of time series identity always includes *all* the labels. For the sake of the info series. The data labels, which are the ones added to the regular series by the `info` function, effectively encode metadata key value pairs. (This implies that a change in the data labels in the conventional Prometheus view constitutes the end of one info series and the beginning of a new info series, while the “logical” view of the `info` function is that the same info series continues to exist, just with different “data”.)

The conventional approach of adding data labels is sometimes called a “join query”, as illustrated by the following example:

```
  rate(http_server_request_duration_seconds_count[2m])
* on (job, instance) group_left (k8s_cluster_name)
  target_info
```

The core of the query is the expression `rate(http_server_request_duration_seconds_count[2m])` . But to add data labels from an info metric, the user has to use elaborate (and not very obvious) syntax to specify which info metric to use (`target_info`), what the identifying labels are (`on (job, instance)`), and which data labels to add (`group_left (k8s_cluster_name)`).

This query is not only verbose and hard to write, it might also run into an “identity crisis”: If any of the data labels of `target_info` changes, Prometheus sees that as a change of series (as alluded to above, Prometheus just has no native concept of non-identifying labels). If the old `target_info` series is not properly marked as stale (which can happen with certain ingestion paths), the query above will fail for up to 5m (the lookback delta) because it will find a conflicting match with both the old and the new version of `target_info` .

The `info` function not only resolves this conflict in favor of the newer series, it also simplifies the syntax because it knows about the available info series and what their identifying labels are. The example query looks like this with the `info` function:

```
info(
  rate(http_server_request_duration_seconds_count[2m]),
  {k8s_cluster_name=~".+"}
)
```

The common case of adding *all* data labels can be achieved by omitting the 2nd argument of the `info` function entirely, simplifying the example even more:

While `info` normally automatically finds all matching info series, it's possible to restrict them by providing a `__name__` label matcher, e.g. `{__name__="target_info"}`.

Limitations

In its current iteration, `info` defaults to considering only info series with the name `target_info`. It also assumes that the identifying info series labels are `instance` and `job`. `info` does support other info series names however, through `__name__` label matchers. E.g., one can explicitly say to consider both `target_info` and `build_info` as follows: `{__name__=~"(target|build)_info"}`. However, the identifying labels always have to be `instance` and `job`.

These limitations are partially defeating the purpose of the `info` function. At the current stage, this is an experiment to find out how useful the approach turns out to be in practice. A final version of the `info` function will indeed consider all matching info series and with their appropriate identifying labels.

`irate()`

`irate(v range-vector)` calculates the per-second instant rate of increase of the time series in the range vector. This is based on the last two data points. Breaks in monotonicity (such as counter resets due to target restarts) are automatically adjusted for.

The following example expression returns the per-second rate of HTTP requests looking up to 5 minutes back for the two most recent data points, per time series in the range vector:

```
irate(http_requests_total{job="api-server"}[5m])
```

`irate` should only be used when graphing volatile, fast-moving counters. Use `rate` for alerts and slow-moving counters, as brief changes in the rate can reset the `FOR` clause and graphs consisting entirely of rare spikes are hard to read.

Note that when combining `irate()` with an aggregation operator (e.g. `sum()`) or a function aggregating over time (any function ending in `_over_time`), always take a `irate()` first, then aggregate. Otherwise `irate()` cannot detect counter resets when your target restarts.

For each timeseries in v , label_join(v instant-vector, dst_label string, separator string, src_label_1 string, src_label_2 string, ...) joins all the values of all the src_labels using separator and returns the timeseries with the label dst_label containing the joined value. There can be any number of src_labels in this function.

label_join acts on float and histogram samples in the same way.

This example will return a vector with each time series having a foo label with the value a,b,c added to it:

```
label_join(up{job="api-server",src1="a",src2="b",src3="c"}, "foo", ",", "src1", "src
```

label_replace()

For each timeseries in v , label_replace(v instant-vector, dst_label string, replacement string, src_label string, regex string) matches the regular expression regex against the value of the label src_label . If it matches, the value of the label dst_label in the returned timeseries will be the expansion of replacement , together with the original labels in the input. Capturing groups in the regular expression can be referenced with $1 , $2 , etc. Named capturing groups in the regular expression can be referenced with $name (where name is the capturing group name). If the regular expression doesn't match then the timeseries is returned unchanged.

label_replace acts on float and histogram samples in the same way.

This example will return timeseries with the values a:c at label service and a at label foo :

```
label_replace(up{job="api-server",service="a:c"}, "foo", "$1", "service", "(.*):.*"
```

This second example has the same effect than the first example, and illustrates use of named capturing groups:

```
label_replace(up{job="api-server",service="a:c"}, "foo", "$name", "service", "(?P<na
```

are:

- `ln(+Inf) = +Inf`
- `ln(0) = -Inf`
- `ln(x < 0) = NaN`
- `ln(NaN) = NaN`

`log2()`

`log2(v instant-vector)` calculates the binary logarithm for all elements in `v`. The special cases are equivalent to those in `ln`.

`log10()`

`log10(v instant-vector)` calculates the decimal logarithm for all elements in `v`. The special cases are equivalent to those in `ln`.

`minute()`

`minute(v=vector(time()) instant-vector)` returns the minute of the hour for each of the given times in UTC. Returned values are from 0 to 59.

`month()`

`month(v=vector(time()) instant-vector)` returns the month of the year for each of the given times in UTC. Returned values are from 1 to 12, where 1 means January etc.

`predict_linear()`

`predict_linear(v range-vector, t scalar)` predicts the value of time series `t` seconds from now, based on the range vector `v`, using simple linear regression. The range vector must have at least two samples in order to perform the calculation. When `+Inf` or `-Inf` are found in the range vector, the slope and offset value calculated will be `NaN`.

`predict_linear` should only be used with gauges.

in the range vector. Breaks in monotonicity (such as counter resets due to target restarts) are automatically adjusted for. Also, the calculation extrapolates to the ends of the time range, allowing for missed scrapes or imperfect alignment of scrape cycles with the range's time period.

The following example expression returns the per-second rate of HTTP requests as measured over the last 5 minutes, per time series in the range vector:

```
rate(http_requests_total{job="api-server"}[5m])
```

`rate` acts on native histograms by calculating a new histogram where each component (sum and count of observations, buckets) is the rate of increase between the respective component in the first and last native histogram in `v`. However, each element in `v` that contains a mix of float and native histogram samples within the range, will be missing from the result vector.

`rate` should only be used with counters and native histograms where the components behave like counters. It is best suited for alerting, and for graphing of slow-moving counters.

Note that when combining `rate()` with an aggregation operator (e.g. `sum()`) or a function aggregating over time (any function ending in `_over_time`), always take a `rate()` first, then aggregate. Otherwise `rate()` cannot detect counter resets when your target restarts.

`resets()`

For each input time series, `resets(v range-vector)` returns the number of counter resets within the provided time range as an instant vector. Any decrease in the value between two consecutive float samples is interpreted as a counter reset. A reset in a native histogram is detected in a more complex way: Any decrease in any bucket, including the zero bucket, or in the count of observation constitutes a counter reset, but also the disappearance of any previously populated bucket, an increase in bucket resolution, or a decrease of the zero-bucket width.

`resets` should only be used with counters and counter-like native histograms.

If the range vector contains a mix of float and histogram samples for the same series, counter resets are detected separately and their numbers added up. The change from a float to a histogram sample is *not* considered a counter reset. Each float sample is compared to the next float sample, and each histogram is comprared to the next histogram.

in `v` to the nearest integer. Ties are resolved by rounding up. The optional `to_nearest` argument allows specifying the nearest multiple to which the sample values should be rounded. This multiple may also be a fraction.

`scalar()`

Given a single-element input vector, `scalar(v instant-vector)` returns the sample value of that single element as a scalar. If the input vector does not have exactly one element, `scalar` will return `NaN`.

`sgn()`

`sgn(v instant-vector)` returns a vector with all sample values converted to their sign, defined as this: 1 if v is positive, -1 if v is negative and 0 if v is equal to zero.

`sort()`

`sort(v instant-vector)` returns vector elements sorted by their sample values, in ascending order. Native histograms are sorted by their sum of observations.

Please note that `sort` only affects the results of instant queries, as range query results always have a fixed output ordering.

`sort_desc()`

Same as `sort`, but sorts in descending order.

Like `sort`, `sort_desc` only affects the results of instant queries, as range query results always have a fixed output ordering.

`sort_by_label()`

This function has to be enabled via the feature flag `--enable-feature=promql-experimental-functions`.

Prometheus

Please note that the sort by label functions only affect the results of instant queries, as range query results always have a fixed output ordering.

This function uses natural sort order.

`sort_by_label_desc()`

This function has to be enabled via the feature flag `--enable-feature=promql-experimental-functions`.

Same as `sort_by_label`, but sorts in descending order.

Please note that the sort by label functions only affect the results of instant queries, as range query results always have a fixed output ordering.

This function uses natural sort order.

`sqrt()`

`sqrt(v instant-vector)` calculates the square root of all elements in `v`.

`time()`

`time()` returns the number of seconds since January 1, 1970 UTC. Note that this does not actually return the current time, but the time at which the expression is to be evaluated.

`timestamp()`

`timestamp(v instant-vector)` returns the timestamp of each of the samples of the given vector as the number of seconds since January 1, 1970 UTC. It also works with histogram samples.

`vector()`

`vector(s scalar)` returns the scalar `s` as a vector with no labels.

UTC.

`<aggregation>_over_time()`

The following functions allow aggregating each series of a given range vector over time and return an instant vector with per-series aggregation results:

- `avg_over_time(range-vector)` : the average value of all points in the specified interval.
- `min_over_time(range-vector)` : the minimum value of all points in the specified interval.
- `max_over_time(range-vector)` : the maximum value of all points in the specified interval.
- `sum_over_time(range-vector)` : the sum of all values in the specified interval.
- `count_over_time(range-vector)` : the count of all values in the specified interval.
- `quantile_over_time(scalar, range-vector)` : the φ-quantile (0 ≤ φ ≤ 1) of the values in the specified interval.
- `stddev_over_time(range-vector)` : the population standard deviation of the values in the specified interval.
- `stdvar_over_time(range-vector)` : the population standard variance of the values in the specified interval.
- `last_over_time(range-vector)` : the most recent point value in the specified interval.
- `present_over_time(range-vector)` : the value 1 for any series in the specified interval.

If the feature flag `--enable-feature=promql-experimental-functions` is set, the following additional functions are available:

- `mad_over_time(range-vector)` : the median absolute deviation of all points in the specified interval.

Note that all values in the specified interval have the same weight in the aggregation even if the values are not equally spaced throughout the interval.

`avg_over_time` , `sum_over_time` , `count_over_time` , `last_over_time` , and `present_over_time` handle native histograms as expected. All other functions ignore histogram samples.

Trigonometric Functions

The trigonometric functions work in radians:

`v` (special cases).

- `asin(v instant-vector)` : calculates the arcsine of all elements in `v` (special cases).
- `asinh(v instant-vector)` : calculates the inverse hyperbolic sine of all elements in `v` (special cases).
- `atan(v instant-vector)` : calculates the arctangent of all elements in `v` (special cases).
- `atanh(v instant-vector)` : calculates the inverse hyperbolic tangent of all elements in `v` (special cases).
- `cos(v instant-vector)` : calculates the cosine of all elements in `v` (special cases).
- `cosh(v instant-vector)` : calculates the hyperbolic cosine of all elements in `v` (special cases).
- `sin(v instant-vector)` : calculates the sine of all elements in `v` (special cases).
- `sinh(v instant-vector)` : calculates the hyperbolic sine of all elements in `v` (special cases).
- `tan(v instant-vector)` : calculates the tangent of all elements in `v` (special cases).
- `tanh(v instant-vector)` : calculates the hyperbolic tangent of all elements in `v` (special cases).

The following are useful for converting between degrees and radians:

- `deg(v instant-vector)` : converts radians to degrees for all elements in `v` .
- `pi()` : returns pi.
- `rad(v instant-vector)` : converts degrees to radians for all elements in `v` .

← **Previous** Operators

Edit

Next Examples →

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Query examples

Simple time series selection

Return all time series with the metric `http_requests_total` :

```
http_requests_total
```

Return all time series with the metric `http_requests_total` and the given `job` and `handler` labels:

```
http_requests_total{job="apiserver", handler="/api/comments"}
```

Return a whole range of time (in this case 5 minutes up to the query time) for the same vector, making it a range vector:

```
http_requests_total{job="apiserver", handler="/api/comments"}[5m]
```

Note that an expression resulting in a range vector cannot be graphed directly, but viewed in the tabular ("Console") view of the expression browser.

Using regular expressions, you could select time series only for jobs whose name match a certain pattern, in this case, all jobs that end with `server` :

```
http_requests_total{job=~".*server"}
```

To select all HTTP status codes except 4xx ones, you could run:

Prometheus

Subquery

Return the 5-minute rate of the `http_requests_total` metric for the past 30 minutes, with a resolution of 1 minute.

```
rate(http_requests_total[5m])[30m:1m]
```

This is an example of a nested subquery. The subquery for the `deriv` function uses the default resolution. Note that using subqueries unnecessarily is unwise.

```
max_over_time(deriv(rate(distance_covered_total[5s])[30s:5s])[10m:])
```

Using functions, operators, etc.

Return the per-second rate for all time series with the `http_requests_total` metric name, as measured over the last 5 minutes:

```
rate(http_requests_total[5m])
```

Assuming that the `http_requests_total` time series all have the labels `job` (fanout by job name) and `instance` (fanout by instance of the job), we might want to sum over the rate of all instances, so we get fewer output time series, but still preserve the `job` dimension:

```
sum by (job) (
  rate(http_requests_total[5m])
)
```

If we have two different metrics with the same dimensional labels, we can apply binary operators to them and elements on both sides with the same label set will get matched and propagated to the output. For example, this expression returns the unused memory in MiB for every instance (on a fictional cluster scheduler exposing these metrics about the instances it runs):

```
(instance_memory_limit_bytes - instance_memory_usage_bytes) / 1024 / 1024
```

```
    instance_memory_limit_bytes - instance_memory_usage_bytes
  ) / 1024 / 1024
```

If the same fictional cluster scheduler exposed CPU usage metrics like the following for every instance:

```
instance_cpu_time_ns{app="lion", proc="web", rev="34d0f99", env="prod", job="cluste
instance_cpu_time_ns{app="elephant", proc="worker", rev="34d0f99", env="prod", job='
instance_cpu_time_ns{app="turtle", proc="api", rev="4d3a513", env="prod", job="clus
instance_cpu_time_ns{app="fox", proc="widget", rev="4d3a513", env="prod", job="clus
...
```

...we could get the top 3 CPU users grouped by application (`app`) and process type (`proc`) like this:

```
topk(3, sum by (app, proc) (rate(instance_cpu_time_ns[5m])))
```

Assuming this metric contains one time series per running instance, you could count the number of running instances per application like this:

```
count by (app) (instance_cpu_time_ns)
```

If we are exploring some metrics for their labels, to e.g. be able to aggregate over some of them, we could use the following:

```
limitk(10, app_foo_metric_bar)
```

Alternatively, if we wanted the returned timeseries to be more evenly sampled, we could use the following to get approximately 10% of them:

```
limit_ratio(0.1, app_foo_metric_bar)
```

Prometheus

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

HTTP API

The current stable HTTP API is reachable under `/api/v1` on a Prometheus server. Any non-breaking additions will be added under that endpoint.

Format overview

The API response format is JSON. Every successful API request returns a `2xx` status code.

Invalid requests that reach the API handlers return a JSON error object and one of the following HTTP response codes:

- `400 Bad Request` when parameters are missing or incorrect.
- `422 Unprocessable Entity` when an expression can't be executed (RFC4918).
- `503 Service Unavailable` when queries time out or abort.

Other non- `2xx` codes may be returned for errors occurring before the API endpoint is reached.

An array of warnings may be returned if there are errors that do not inhibit the request execution. An additional array of info-level annotations may be returned for potential query issues that may or may not be false positives. All of the data that was successfully collected will be returned in the data field.

The JSON response envelope format is as follows:

```
{
  "status": "success" | "error",
  "data": <data>,

  // Only set if status is "error". The data field may still hold
```

```
    // Only set if there were warnings while executing the request.
    // There will still be data in the data field.
    "warnings": ["<string>"],
    // Only set if there were info-level annotations while executing the request.
    "infos": ["<string>"]
  }
```

Generic placeholders are defined as follows:

- `<rfc3339 | unix_timestamp>` : Input timestamps may be provided either in RFC3339 format or as a Unix timestamp in seconds, with optional decimal places for sub-second precision. Output timestamps are always represented as Unix timestamps in seconds.
- `<series_selector>` : Prometheus time series selectors like `http_requests_total` or `http_requests_total{method=~"(GET|POST)"}` and need to be URL-encoded.
- `<duration>` : the subset of Prometheus float literals using time units. For example, `5m` refers to a duration of 5 minutes.
- `<bool>` : boolean values (strings `true` and `false`).

Note: Names of query parameters that may be repeated end with `[]` .

Expression queries

Query language expressions may be evaluated at a single instant or over a range of time. The sections below describe the API endpoints for each type of expression query.

Instant queries

The following endpoint evaluates an instant query at a single point in time:

```
GET /api/v1/query
POST /api/v1/query
```

URL query parameters:

- `query=<string>` : Prometheus expression query string.
- `time=<rfc3339 | unix_timestamp>` : Evaluation timestamp. Optional.

but truncates the number of series for matrices and vectors. Optional. 0 means disabled.

The current server time is used if the `time` parameter is omitted.

You can URL-encode these parameters directly in the request body by using the `POST` method and `Content-Type: application/x-www-form-urlencoded` header. This is useful when specifying a large query that may breach server-side URL character limits.

The `data` section of the query result has the following format:

```
{
  "resultType": "matrix" | "vector" | "scalar" | "string",
  "result": <value>
}
```

`<value>` refers to the query result data, which has varying formats depending on the `resultType`. See the expression query result formats.

The following example evaluates the expression `up` at the time `2015-07-01T20:10:51.781Z`:

```
$ curl 'http://localhost:9090/api/v1/query?query=up&time=2015-07-01T20:10:51.781Z'
{
   "status" : "success",
   "data" : {
      "resultType" : "vector",
      "result" : [
         {
            "metric" : {
               "__name__" : "up",
               "job" : "prometheus",
               "instance" : "localhost:9090"
            },
            "value": [ 1435781451.781, "1" ]
         },
         {
            "metric" : {
               "__name__" : "up",
               "job" : "node",
               "instance" : "localhost:9100"
      ]
    }
  }
```

Range queries

The following endpoint evaluates an expression query over a range of time:

```
GET /api/v1/query_range
POST /api/v1/query_range
```

URL query parameters:

- `query=<string>` : Prometheus expression query string.
- `start=<rfc3339 | unix_timestamp>` : Start timestamp, inclusive.
- `end=<rfc3339 | unix_timestamp>` : End timestamp, inclusive.
- `step=<duration | float>` : Query resolution step width in `duration` format or float number of seconds.
- `timeout=<duration>` : Evaluation timeout. Optional. Defaults to and is capped by the value of the `-query.timeout` flag.
- `limit=<number>` : Maximum number of returned series. Optional. 0 means disabled.

You can URL-encode these parameters directly in the request body by using the `POST` method and `Content-Type: application/x-www-form-urlencoded` header. This is useful when specifying a large query that may breach server-side URL character limits.

The `data` section of the query result has the following format:

```
{
  "resultType": "matrix",
  "result": <value>
}
```

For the format of the `<value>` placeholder, see the range-vector result format.

The following example evaluates the expression `up` over a 30-second range with a query resolution of 15 seconds.

```
   "status" : "success",
   "data" : {
      "resultType" : "matrix",
      "result" : [
         {
            "metric" : {
               "__name__" : "up",
               "job" : "prometheus",
               "instance" : "localhost:9090"
            },
            "values" : [
               [ 1435781430.781, "1" ],
               [ 1435781445.781, "1" ],
               [ 1435781460.781, "1" ]
            ]
         },
         {
            "metric" : {
               "__name__" : "up",
               "job" : "node",
               "instance" : "localhost:9091"
            },
            "values" : [
               [ 1435781430.781, "0" ],
               [ 1435781445.781, "0" ],
               [ 1435781460.781, "1" ]
            ]
         }
      ]
   }
}
```

Formatting query expressions

The following endpoint formats a PromQL expression in a prettified way:

```
GET /api/v1/format_query
POST /api/v1/format_query
```

You can URL-encode these parameters directly in the request body by using the `POST` method and `Content-Type: application/x-www-form-urlencoded` header. This is useful when specifying a large query that may breach server-side URL character limits.

The `data` section of the query result is a string containing the formatted query expression. Note that any comments are removed in the formatted string.

The following example formats the expression `foo/bar`:

```
$ curl 'http://localhost:9090/api/v1/format_query?query=foo/bar'
{
   "status" : "success",
   "data" : "foo / bar"
}
```

Parsing a PromQL expressions into a abstract syntax tree (AST)

This endpoint is **experimental** and might change in the future. It is currently only meant to be used by Prometheus' own web UI, and the endpoint name and exact format returned may change from one Prometheus version to another. It may also be removed again in case it is no longer needed by the UI.

The following endpoint parses a PromQL expression and returns it as a JSON-formatted AST (abstract syntax tree) representation:

```
GET /api/v1/parse_query
POST /api/v1/parse_query
```

URL query parameters:

- `query=<string>`: Prometheus expression query string.

You can URL-encode these parameters directly in the request body by using the `POST` method and `Content-Type: application/x-www-form-urlencoded` header. This is useful when specifying a large query that may breach server-side URL character limits.

Prometheus

The following example parses the expression `foo/bar`:

```
$ curl 'http://localhost:9090/api/v1/parse_query?query=foo/bar'
{
   "data" : {
      "bool" : false,
      "lhs" : {
         "matchers" : [
            {
               "name" : "__name__",
               "type" : "=",
               "value" : "foo"
            }
         ],
         "name" : "foo",
         "offset" : 0,
         "startOrEnd" : null,
         "timestamp" : null,
         "type" : "vectorSelector"
      },
      "matching" : {
         "card" : "one-to-one",
         "include" : [],
         "labels" : [],
         "on" : false
      },
      "op" : "/",
      "rhs" : {
         "matchers" : [
            {
               "name" : "__name__",
               "type" : "=",
               "value" : "bar"
            }
         ],
         "name" : "bar",
         "offset" : 0,
         "startOrEnd" : null,
         "timestamp" : null,
         "type" : "vectorSelector"
```

```
    "status" : "success"
 }
```

Querying metadata

Prometheus offers a set of API endpoints to query metadata about series and their labels.

> **NOTE**: These API endpoints may return metadata for series for which there is no sample within the selected time range, and/or for series whose samples have been marked as deleted via the deletion API endpoint. The exact extent of additionally returned series metadata is an implementation detail that may change in the future.

Finding series by label matchers

The following endpoint returns the list of time series that match a certain label set.

```
GET /api/v1/series
POST /api/v1/series
```

URL query parameters:

- `match[]=<series_selector>` : Repeated series selector argument that selects the series to return. At least one `match[]` argument must be provided.
- `start=<rfc3339 | unix_timestamp>` : Start timestamp.
- `end=<rfc3339 | unix_timestamp>` : End timestamp.
- `limit=<number>` : Maximum number of returned series. Optional. 0 means disabled.

You can URL-encode these parameters directly in the request body by using the `POST` method and `Content-Type: application/x-www-form-urlencoded` header. This is useful when specifying a large or dynamic number of series selectors that may breach server-side URL character limits.

The `data` section of the query result consists of a list of objects that contain the label name/value pairs which identify each series.

Prometheus

```
$ curl -g 'http://localhost:9090/api/v1/series?' --data-urlencode 'match[]=up' --dat
{
   "status" : "success",
   "data" : [
      {
         "__name__" : "up",
         "job" : "prometheus",
         "instance" : "localhost:9090"
      },
      {
         "__name__" : "up",
         "job" : "node",
         "instance" : "localhost:9091"
      },
      {
         "__name__" : "process_start_time_seconds",
         "job" : "prometheus",
         "instance" : "localhost:9090"
      }
   ]
}
```

Getting label names

The following endpoint returns a list of label names:

```
GET /api/v1/labels
POST /api/v1/labels
```

URL query parameters:

- `start=<rfc3339 | unix_timestamp>` : Start timestamp. Optional.
- `end=<rfc3339 | unix_timestamp>` : End timestamp. Optional.
- `match[]=<series_selector>` : Repeated series selector argument that selects the series from which to read the label names. Optional.
- `limit=<number>` : Maximum number of returned series. Optional. 0 means disabled.

The `data` section of the JSON response is a list of string label names.

```json
{
    "status": "success",
    "data": [
        "__name__",
        "call",
        "code",
        "config",
        "dialer_name",
        "endpoint",
        "event",
        "goversion",
        "handler",
        "instance",
        "interval",
        "job",
        "le",
        "listener_name",
        "name",
        "quantile",
        "reason",
        "role",
        "scrape_job",
        "slice",
        "version"
    ]
}
```

Querying label values

The following endpoint returns a list of label values for a provided label name:

```
GET /api/v1/label/<label_name>/values
```

URL query parameters:

- `start=<rfc3339 | unix_timestamp>` : Start timestamp. Optional.
- `end=<rfc3339 | unix_timestamp>` : End timestamp. Optional.

The `data` section of the JSON response is a list of string label values.

This example queries for all label values for the `http_status_code` label:

```
$ curl http://localhost:9090/api/v1/label/http_status_code/values
{
   "status" : "success",
   "data" : [
      "200",
      "504"
   ]
}
```

Label names can optionally be encoded using the Values Escaping method, and is necessary if a name includes the `/` character. To encode a name in this way:

- Prepend the label with `U__`.
- Letters, numbers, and colons appear as-is.
- Convert single underscores to double underscores.
- For all other characters, use the UTF-8 codepoint as a hex integer, surrounded by underscores. So ` ` becomes `_20_` and a `.` becomes `_2e_`.

More information about text escaping can be found in the original UTF-8 Proposal document.

This example queries for all label values for the `http.status_code` label:

```
$ curl http://localhost:9090/api/v1/label/U__http_2e_status_code/values
{
   "status" : "success",
   "data" : [
      "200",
      "404"
   ]
}
```

Querying exemplars

```
GET /api/v1/query_exemplars
POST /api/v1/query_exemplars
```

URL query parameters:

- `query=<string>` : Prometheus expression query string.
- `start=<rfc3339 | unix_timestamp>` : Start timestamp.
- `end=<rfc3339 | unix_timestamp>` : End timestamp.

```
$ curl -g 'http://localhost:9090/api/v1/query_exemplars?query=test_exemplar_metric_t
{
    "status": "success",
    "data": [
        {
            "seriesLabels": {
                "__name__": "test_exemplar_metric_total",
                "instance": "localhost:8090",
                "job": "prometheus",
                "service": "bar"
            },
            "exemplars": [
                {
                    "labels": {
                        "trace_id": "EpTxMJ40fUus7aGY"
                    },
                    "value": "6",
                    "timestamp": 1600096945.479
                }
            ]
        },
        {
            "seriesLabels": {
                "__name__": "test_exemplar_metric_total",
                "instance": "localhost:8090",
                "job": "prometheus",
                "service": "foo"
            },
            "exemplars": [
                {
                    "labels": {
```

```
                    "timestamp": 1600096955.479
                },
                {
                    "labels": {
                        "trace_id": "hCtjygkIHwAN9vs4"
                    },
                    "value": "20",
                    "timestamp": 1600096965.489
                }
            ]
        }
    ]
}
```

Expression query result formats

Expression queries may return the following response values in the `result` property of the `data` section. `<sample_value>` placeholders are numeric sample values. JSON does not support special float values such as `NaN`, `Inf`, and `-Inf`, so sample values are transferred as quoted JSON strings rather than raw numbers.

The keys `"histogram"` and `"histograms"` only show up if the experimental native histograms are present in the response. Their placeholder `<histogram>` is explained in detail in its own section below.

Range vectors

Range vectors are returned as result type `matrix`. The corresponding `result` property has the following format:

```
[
  {
    "metric": { "<label_name>": "<label_value>", ... },
    "values": [ [ <unix_time>, "<sample_value>" ], ... ],
    "histograms": [ [ <unix_time>, <histogram> ], ... ]
  },
```

Each series could have the `"values"` key, or the `"histograms"` key, or both. For a given timestamp, there will only be one sample of either float or histogram type.

Series are returned sorted by `metric`. Functions such as `sort` and `sort_by_label` have no effect for range vectors.

Instant vectors

Instant vectors are returned as result type `vector`. The corresponding `result` property has the following format:

```
[
  {
    "metric": { "<label_name>": "<label_value>", ... },
    "value": [ <unix_time>, "<sample_value>" ],
    "histogram": [ <unix_time>, <histogram> ]
  },
  ...
]
```

Each series could have the `"value"` key, or the `"histogram"` key, but not both.

Series are not guaranteed to be returned in any particular order unless a function such as `sort` or `sort_by_label` is used.

Scalars

Scalar results are returned as result type `scalar`. The corresponding `result` property has the following format:

```
[ <unix_time>, "<scalar_value>" ]
```

Strings

String results are returned as result type `string`. The corresponding `result` property has the following format:

Prometheus

Native histograms

The `<histogram>` placeholder used above is formatted as follows.

Note that native histograms are an experimental feature, and the format below might still change.

```
{
  "count": "<count_of_observations>",
  "sum": "<sum_of_observations>",
  "buckets": [ [ <boundary_rule>, "<left_boundary>", "<right_boundary>", "<count_in_
}
```

The `<boundary_rule>` placeholder is an integer between 0 and 3 with the following meaning:

- 0: “open left” (left boundary is exclusive, right boundary in inclusive)
- 1: “open right” (left boundary is inclusive, right boundary in exclusive)
- 2: “open both” (both boundaries are exclusive)
- 3: “closed both” (both boundaries are inclusive)

Note that with the currently implemented bucket schemas, positive buckets are “open left”, negative buckets are “open right”, and the zero bucket (with a negative left boundary and a positive right boundary) is “closed both”.

Targets

The following endpoint returns an overview of the current state of the Prometheus target discovery:

```
GET /api/v1/targets
```

Both the active and dropped targets are part of the response by default. Dropped targets are subject to `keep_dropped_targets` limit, if set. `labels` represents the label set after relabeling has occurred. `discoveredLabels` represent the unmodified labels retrieved during service discovery before relabeling has occurred.

```
  "status": "success",
  "data": {
    "activeTargets": [
      {
        "discoveredLabels": {
          "__address__": "127.0.0.1:9090",
          "__metrics_path__": "/metrics",
          "__scheme__": "http",
          "job": "prometheus"
        },
        "labels": {
          "instance": "127.0.0.1:9090",
          "job": "prometheus"
        },
        "scrapePool": "prometheus",
        "scrapeUrl": "http://127.0.0.1:9090/metrics",
        "globalUrl": "http://example-prometheus:9090/metrics",
        "lastError": "",
        "lastScrape": "2017-01-17T15:07:44.723715405+01:00",
        "lastScrapeDuration": 0.050688943,
        "health": "up",
        "scrapeInterval": "1m",
        "scrapeTimeout": "10s"
      }
    ],
    "droppedTargets": [
      {
        "discoveredLabels": {
          "__address__": "127.0.0.1:9100",
          "__metrics_path__": "/metrics",
          "__scheme__": "http",
          "__scrape_interval__": "1m",
          "__scrape_timeout__": "10s",
          "job": "node"
        },
      }
    ]
  }
}
```

Prometheus

```
$ curl 'http://localhost:9090/api/v1/targets?state=active'
{
  "status": "success",
  "data": {
    "activeTargets": [
      {
        "discoveredLabels": {
          "__address__": "127.0.0.1:9090",
          "__metrics_path__": "/metrics",
          "__scheme__": "http",
          "job": "prometheus"
        },
        "labels": {
          "instance": "127.0.0.1:9090",
          "job": "prometheus"
        },
        "scrapePool": "prometheus",
        "scrapeUrl": "http://127.0.0.1:9090/metrics",
        "globalUrl": "http://example-prometheus:9090/metrics",
        "lastError": "",
        "lastScrape": "2017-01-17T15:07:44.723715405+01:00",
        "lastScrapeDuration": 50688943,
        "health": "up"
      }
    ],
    "droppedTargets": []
  }
}
```

The `scrapePool` query parameter allows the caller to filter by scrape pool name.

```
$ curl 'http://localhost:9090/api/v1/targets?scrapePool=node_exporter'
{
  "status": "success",
  "data": {
    "activeTargets": [
      {
        "discoveredLabels": {
          "__address__": "127.0.0.1:9091",
        },
        "labels": {
          "instance": "127.0.0.1:9091",
          "job": "node_exporter"
        },
        "scrapePool": "node_exporter",
        "scrapeUrl": "http://127.0.0.1:9091/metrics",
        "globalUrl": "http://example-prometheus:9091/metrics",
        "lastError": "",
        "lastScrape": "2017-01-17T15:07:44.723715405+01:00",
        "lastScrapeDuration": 50688943,
        "health": "up"
      }
    ],
    "droppedTargets": []
  }
}
```

Rules

The `/rules` API endpoint returns a list of alerting and recording rules that are currently loaded. In addition it returns the currently active alerts fired by the Prometheus instance of each alerting rule.

As the `/rules` endpoint is fairly new, it does not have the same stability guarantees as the overarching API v1.

```
GET /api/v1/rules
```

URL query parameters:

- `type=alert|record` : return only the alerting rules (e.g. `type=alert`) or the recording rules (e.g. `type=record`). When the parameter is absent or empty, no filtering is done.
- `rule_name[]=<string>` : only return rules with the given rule name. If the parameter is repeated, rules with any of the provided names are returned. If we've filtered out all the rules of a group, the group is not returned. When the parameter is absent or empty, no filtering is done.

- `file[]=<string>` : only return rules with the given filepath. If the parameter is repeated, rules with any of the provided filepaths are returned. When the parameter is absent or empty, no filtering is done.
- `exclude_alerts=<bool>` : only return rules, do not return active alerts.
- `match[]=<label_selector>` : only return rules that have configured labels that satisfy the label selectors. If the parameter is repeated, rules that match any of the sets of label selectors are returned. Note that matching is on the labels in the definition of each rule, not on the values after template expansion (for alerting rules). Optional.
- `group_limit=<number>` : The `group_limit` parameter allows you to specify a limit for the number of rule groups that is returned in a single response. If the total number of rule groups exceeds the specified `group_limit` value, the response will include a `groupNextToken` property. You can use the value of this `groupNextToken` property in subsequent requests in the `group_next_token` parameter to paginate over the remaining rule groups. The `groupNextToken` property will not be present in the final response, indicating that you have retrieved all the available rule groups. Please note that there are no guarantees regarding the consistency of the response if the rule groups are being modified during the pagination process.
- `group_next_token` : the pagination token that was returned in previous request when the `group_limit` property is set. The pagination token is used to iteratively paginate over a large number of rule groups. To use the `group_next_token` parameter, the `group_limit` parameter also need to be present. If a rule group that coincides with the next token is removed while you are paginating over the rule groups, a response with status code 400 will be returned.

```
$ curl http://localhost:9090/api/v1/rules

{
    "data": {
        "groups": [
            {
                "rules": [
                    {
                        "alerts": [
                            {
                                "activeAt": "2018-07-04T20:27:12.60602144+02:00",
                                "annotations": {
                                    "summary": "High request latency"
                                },
                                "labels": {
```

Prometheus

```json
                                "state": "firing",
                                "value": "1e+00"
                            }
                        ],
                        "annotations": {
                            "summary": "High request latency"
                        },
                        "duration": 600,
                        "health": "ok",
                        "labels": {
                            "severity": "page"
                        },
                        "name": "HighRequestLatency",
                        "query": "job:request_latency_seconds:mean5m{job=\"myjob\"}
                        "type": "alerting"
                    },
                    {
                        "health": "ok",
                        "name": "job:http_inprogress_requests:sum",
                        "query": "sum by (job) (http_inprogress_requests)",
                        "type": "recording"
                    }
                ],
                "file": "/rules.yaml",
                "interval": 60,
                "limit": 0,
                "name": "example"
            }
        ]
    },
    "status": "success"
}
```

Alerts

The `/alerts` endpoint returns a list of all active alerts.

As the `/alerts` endpoint is fairly new, it does not have the same stability guarantees as the overarching API v1.

```
$ curl http://localhost:9090/api/v1/alerts

{
    "data": {
        "alerts": [
            {
                "activeAt": "2018-07-04T20:27:12.60602144+02:00",
                "annotations": {},
                "labels": {
                    "alertname": "my-alert"
                },
                "state": "firing",
                "value": "1e+00"
            }
        ]
    },
    "status": "success"
}
```

Querying target metadata

The following endpoint returns metadata about metrics currently scraped from targets. This is **experimental** and might change in the future.

```
GET /api/v1/targets/metadata
```

URL query parameters:

- `match_target=<label_selectors>` : Label selectors that match targets by their label sets. All targets are selected if left empty.
- `metric=<string>` : A metric name to retrieve metadata for. All metric metadata is retrieved if left empty.
- `limit=<number>` : Maximum number of targets to match.

The `data` section of the query result consists of a list of objects that contain metric metadata and the target label set.

The following example returns all metadata entries for the `go_goroutines` metric from the first two targets with label `job="prometheus"` .

```
    --data-urlencode 'match_target={job="prometheus"}' \
    --data-urlencode 'limit=2'
{
  "status": "success",
  "data": [
    {
      "target": {
        "instance": "127.0.0.1:9090",
        "job": "prometheus"
      },
      "type": "gauge",
      "help": "Number of goroutines that currently exist.",
      "unit": ""
    },
    {
      "target": {
        "instance": "127.0.0.1:9091",
        "job": "prometheus"
      },
      "type": "gauge",
      "help": "Number of goroutines that currently exist.",
      "unit": ""
    }
  ]
}
```

The following example returns metadata for all metrics for all targets with label `instance="127.0.0.1:9090"`.

```
curl -G http://localhost:9091/api/v1/targets/metadata \
    --data-urlencode 'match_target={instance="127.0.0.1:9090"}'
{
  "status": "success",
  "data": [
    // ...
    {
      "target": {
        "instance": "127.0.0.1:9090",
        "job": "prometheus"
      },
      "unit": ""
    },
    {
      "target": {
        "instance": "127.0.0.1:9090",
        "job": "prometheus"
      },
      "metric": "prometheus_tsdb_reloads_total",
      "type": "counter",
      "help": "Number of times the database reloaded block data from disk.",
      "unit": ""
    },
    // ...
  ]
}
```

Querying metric metadata

It returns metadata about metrics currently scraped from targets. However, it does not provide any target information. This is considered **experimental** and might change in the future.

```
GET /api/v1/metadata
```

URL query parameters:

- `limit=<number>` : Maximum number of metrics to return.
- `limit_per_metric=<number>` : Maximum number of metadata to return per metric.
- `metric=<string>` : A metric name to filter metadata for. All metric metadata is retrieved if left empty.

The `data` section of the query result consists of an object where each key is a metric name and each value is a list of unique metadata objects, as exposed for that metric name across all targets.

The following example returns two metrics. Note that the metric `http_requests_total` has more than one object in the list. At least one target has a value for `HELP` that do not match with the rest.

```
{
  "status": "success",
  "data": {
    "cortex_ring_tokens": [
      {
        "type": "gauge",
        "help": "Number of tokens in the ring",
        "unit": ""
      }
    ],
    "http_requests_total": [
      {
        "type": "counter",
        "help": "Number of HTTP requests",
        "unit": ""
      },
      {
        "type": "counter",
        "help": "Amount of HTTP requests",
        "unit": ""
      }
    ]
  }
}
```

The following example returns only one metadata entry for each metric.

```
curl -G http://localhost:9090/api/v1/metadata?limit_per_metric=1
```

```
{
  "status": "success",
  "data": {
    "cortex_ring_tokens": [
      {
        "type": "gauge",
        "help": "Number of tokens in the ring",
        "unit": ""
      }
    ],
    "http_requests_total": [
        "unit": ""
      }
    ]
  }
}
```

The following example returns metadata only for the metric `http_requests_total`.

```
curl -G http://localhost:9090/api/v1/metadata?metric=http_requests_total
```

```
{
  "status": "success",
  "data": {
    "http_requests_total": [
      {
        "type": "counter",
        "help": "Number of HTTP requests",
        "unit": ""
      },
      {
        "type": "counter",
        "help": "Amount of HTTP requests",
        "unit": ""
      }
    ]
  }
}
```

Alertmanagers

The following endpoint returns an overview of the current state of the Prometheus alertmanager discovery:

```
GET /api/v1/alertmanagers
```

Both the active and dropped Alertmanagers are part of the response.

```
  "status": "success",
  "data": {
    "activeAlertmanagers": [
      {
        "url": "http://127.0.0.1:9090/api/v1/alerts"
      }
    ],
    "droppedAlertmanagers": [
      {
        "url": "http://127.0.0.1:9093/api/v1/alerts"
      }
    ]
  }
}
```

Status

Following status endpoints expose current Prometheus configuration.

Config

The following endpoint returns currently loaded configuration file:

```
GET /api/v1/status/config
```

The config is returned as dumped YAML file. Due to limitation of the YAML library, YAML comments are not included.

```
$ curl http://localhost:9090/api/v1/status/config
{
  "status": "success",
  "data": {
    "yaml": "<content of the loaded config file in YAML>",
  }
}
```

Prometheus

```
GET /api/v1/status/flags
```

All values are of the result type `string`.

```
$ curl http://localhost:9090/api/v1/status/flags
{
  "status": "success",
  "data": {
    "alertmanager.notification-queue-capacity": "10000",
    "alertmanager.timeout": "10s",
    "log.level": "info",
    "query.lookback-delta": "5m",
    "query.max-concurrency": "20",
    ...
  }
}
```

New in v2.2

Runtime Information

The following endpoint returns various runtime information properties about the Prometheus server:

```
GET /api/v1/status/runtimeinfo
```

The returned values are of different types, depending on the nature of the runtime property.

```
$ curl http://localhost:9090/api/v1/status/runtimeinfo
{
  "status": "success",
  "data": {
    "startTime": "2019-11-02T17:23:59.301361365+01:00",
    "CWD": "/",
    "hostname" : "DESKTOP-717H17Q",
    "serverTime": "2025-01-05T18:27:33Z",
    "reloadConfigSuccess": true,
    "goroutineCount": 48,
    "GOMAXPROCS": 4,
    "GOGC": "",
    "GODEBUG": "",
    "storageRetention": "15d"
  }
}
```

> **NOTE**: The exact returned runtime properties may change without notice between Prometheus versions.

New in v2.14

Build Information

The following endpoint returns various build information properties about the Prometheus server:

```
GET /api/v1/status/buildinfo
```

All values are of the result type `string`.

```
$ curl http://localhost:9090/api/v1/status/buildinfo
{
  "status": "success",
  "data": {
    "version": "2.13.1",
    "revision": "cb7cbad5f9a2823a622aaa668833ca04f50a0ea7",
    "branch": "master",
    "buildUser": "julius@desktop",
    "buildDate": "20191102-16:19:59",
    "goVersion": "go1.13.1"
  }
}
```

Prometheus

New in v2.14

TSDB Stats

The following endpoint returns various cardinality statistics about the Prometheus TSDB:

```
GET /api/v1/status/tsdb
```

URL query parameters:

- `limit=<number>` : Limit the number of returned items to a given number for each set of statistics. By default, 10 items are returned.

The `data` section of the query result consists of:

- **headStats**: This provides the following data about the head block of the TSDB:
  - **numSeries**: The number of series.
  - **chunkCount**: The number of chunks.
  - **minTime**: The current minimum timestamp in milliseconds.
  - **maxTime**: The current maximum timestamp in milliseconds.
- **seriesCountByMetricName:** This will provide a list of metrics names and their series count.
- **labelValueCountByLabelName:** This will provide a list of the label names and their value count.
- **memoryInBytesByLabelName** This will provide a list of the label names and memory used in bytes. Memory usage is calculated by adding the length of all values for a given label name.
- **seriesCountByLabelPair** This will provide a list of label value pairs and their series count.

```
$ curl http://localhost:9090/api/v1/status/tsdb
{
  "status": "success",
  "data": {
    "headStats": {
      "numSeries": 508,
      "chunkCount": 937,
    "seriesCountByMetricName": [
      {
        "name": "net_conntrack_dialer_conn_failed_total",
        "value": 20
      },
      {
        "name": "prometheus_http_request_duration_seconds_bucket",
        "value": 20
      }
    ],
    "labelValueCountByLabelName": [
      {
        "name": "__name__",
        "value": 211
      },
      {
        "name": "event",
        "value": 3
      }
    ],
    "memoryInBytesByLabelName": [
      {
        "name": "__name__",
        "value": 8266
      },
      {
        "name": "instance",
        "value": 28
      }
    ],
    "seriesCountByLabelValuePair": [
      {
        "name": "job=prometheus",
        "value": 425
      },
      {
        "name": "instance=localhost:9090",
        "value": 425
      }
    ]
```

Prometheus

New in v2.15

WAL Replay Stats

The following endpoint returns information about the WAL replay:

```
GET /api/v1/status/walreplay
```

- **read**: The number of segments replayed so far.
- **total**: The total number segments needed to be replayed.
- **progress**: The progress of the replay (0 - 100%).
- **state**: The state of the replay. Possible states:
  - **waiting**: Waiting for the replay to start.
  - **in progress**: The replay is in progress.
  - **done**: The replay has finished.

```
$ curl http://localhost:9090/api/v1/status/walreplay
{
  "status": "success",
  "data": {
    "min": 2,
    "max": 5,
    "current": 40,
    "state": "in progress"
  }
}
```

NOTE: This endpoint is available before the server has been marked ready and is updated in real time to facilitate monitoring the progress of the WAL replay.

New in v2.28

TSDB Admin APIs

Prometheus

Snapshot

Snapshot creates a snapshot of all current data into `snapshots/<datetime>-<rand>` under the TSDB's data directory and returns the directory as response. It will optionally skip snapshotting data that is only present in the head block, and which has not yet been compacted to disk.

```
POST /api/v1/admin/tsdb/snapshot
PUT /api/v1/admin/tsdb/snapshot
```

URL query parameters:

- `skip_head=<bool>` : Skip data present in the head block. Optional.

```
$ curl -XPOST http://localhost:9090/api/v1/admin/tsdb/snapshot
{
  "status": "success",
  "data": {
    "name": "20171210T211224Z-2be650b6d019eb54"
  }
}
```

The snapshot now exists at `<data-dir>/snapshots/20171210T211224Z-2be650b6d019eb54`

New in v2.1 and supports PUT from v2.9

Delete Series

DeleteSeries deletes data for a selection of series in a time range. The actual data still exists on disk and is cleaned up in future compactions or can be explicitly cleaned up by hitting the Clean Tombstones endpoint.

If successful, a `204` is returned.

```
POST /api/v1/admin/tsdb/delete_series
PUT /api/v1/admin/tsdb/delete_series
```

URL query parameters:

minimum possible time.
- `end=<rfc3339 | unix_timestamp>` : End timestamp. Optional and defaults to maximum possible time.

Not mentioning both start and end times would clear all the data for the matched series in the database.

Example:

```
$ curl -X POST \
  -g 'http://localhost:9090/api/v1/admin/tsdb/delete_series?match[]=up&match[]=proce
```

NOTE: This endpoint marks samples from series as deleted, but will not necessarily prevent associated series metadata from still being returned in metadata queries for the affected time range (even after cleaning tombstones). The exact extent of metadata deletion is an implementation detail that may change in the future.

New in v2.1 and supports PUT from v2.9

Clean Tombstones

CleanTombstones removes the deleted data from disk and cleans up the existing tombstones. This can be used after deleting series to free up space.

If successful, a `204` is returned.

```
POST /api/v1/admin/tsdb/clean_tombstones
PUT /api/v1/admin/tsdb/clean_tombstones
```

This takes no parameters or body.

```
$ curl -XPOST http://localhost:9090/api/v1/admin/tsdb/clean_tombstones
```

New in v2.1 and supports PUT from v2.9 is not considered an efficient way of ingesting samples. Use it with caution for specific low-volume use cases. It is not suitable for replacing the ingestion via scraping and turning Prometheus into a push-based metrics collection system.

Enable the remote write receiver by setting `--web.enable-remote-write-receiver`. When enabled, the remote write receiver endpoint is `/api/v1/write`. Find more details here.

New in v2.33

OTLP Receiver

Prometheus can be configured as a receiver for the OTLP Metrics protocol. This is not considered an efficient way of ingesting samples. Use it with caution for specific low-volume use cases. It is not suitable for replacing the ingestion via scraping.

Enable the OTLP receiver by setting `--web.enable-otlp-receiver`. When enabled, the OTLP receiver endpoint is `/api/v1/otlp/v1/metrics`.

New in v2.47

OTLP Delta

Prometheus can convert incoming metrics from delta temporality to their cumulative equivalent. This is done using deltatocumulative from the OpenTelemetry Collector.

To enable, pass `--enable-feature=otlp-deltatocumulative`.

New in v3.2

Notifications

The following endpoints provide information about active status notifications concerning the Prometheus server itself. Notifications are used in the web UI.

These endpoints are **experimental**. They may change in the future.

Active Notifications

Example:

```
$ curl http://localhost:9090/api/v1/notifications
{
  "status": "success",
  "data": [
    {
      "text": "Prometheus is shutting down and gracefully stopping all operations.",
      "date": "2024-10-07T12:33:08.551376578+02:00",
      "active": true
    }
  ]
}
```

New in v3.0

Live Notifications

The `/api/v1/notifications/live` endpoint streams live notifications as they occur, using Server-Sent Events. Deleted notifications are sent with `active: false`. Active notifications will be sent when connecting to the endpoint.

```
GET /api/v1/notifications/live
```

Example:

```
$ curl http://localhost:9090/api/v1/notifications/live
data: {
  "status": "success",
  "data": [
    {
      "text": "Prometheus is shutting down and gracefully stopping all operations.",
      "date": "2024-10-07T12:33:08.551376578+02:00",
      "active": true
    }
```

Prometheus

Note: The `/notifications/live` endpoint will return a `204 No Content` response if the maximum number of subscribers has been reached. You can set the maximum number of listeners with the flag `--web.max-notifications-subscribers`, which defaults to 16.

```
GET /api/v1/notifications/live
204 No Content
```

New in v3.0

← **Previous** Examples

Edit

Next Remote Read API →

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Remote Read API

> This is not currently considered part of the stable API and is subject to change even between non-major version releases of Prometheus.

This API provides data read functionality from Prometheus. This interface expects snappy compression. The API definition is located here.

Request are made to the following endpoint.

```
/api/v1/read
```

Samples

This returns a message that includes a list of raw samples matching the requested query.

Streamed Chunks

These streamed chunks utilize an XOR algorithm inspired by the Gorilla compression to encode the chunks. However, it provides resolution to the millisecond instead of to the second.

Previous HTTP API

Edit

Next Storage

Prometheus

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Storage

Prometheus includes a local on-disk time series database, but also optionally integrates with remote storage systems.

Local storage

Prometheus's local time series database stores data in a custom, highly efficient format on local storage.

On-disk layout

Ingested samples are grouped into blocks of two hours. Each two-hour block consists of a directory containing a chunks subdirectory containing all the time series samples for that window of time, a metadata file, and an index file (which indexes metric names and labels to time series in the chunks directory). The samples in the chunks directory are grouped together into one or more segment files of up to 512MB each by default. When series are deleted via the API, deletion records are stored in separate tombstone files (instead of deleting the data immediately from the chunk segments).

The current block for incoming samples is kept in memory and is not fully persisted. It is secured against crashes by a write-ahead log (WAL) that can be replayed when the Prometheus server restarts. Write-ahead log files are stored in the `wal` directory in 128MB segments. These files contain raw data that has not yet been compacted; thus they are significantly larger than regular block files. Prometheus will retain a minimum of three write-ahead log files. High-traffic servers may retain more than three WAL files in order to keep at least two hours of raw data.

A Prometheus server's data directory looks something like this:

```
|   └── meta.json
├── 01BKGTZQ1SYQJTR4PB43C8PD98
|   ├── chunks
|   |   └── 000001
|   ├── tombstones
|   ├── index
|   └── meta.json
├── 01BKGTZQ1HHWHV8FBJXW1Y3W0K
|   └── meta.json
├── 01BKGV7JC0RY8A6MACW02A2PJD
|   ├── chunks
|   |   └── 000001
|   ├── tombstones
|   ├── index
|   └── meta.json
├── chunks_head
|   └── 000001
└── wal
    ├── 000000002
    └── checkpoint.00000001
        └── 00000000
```

Note that a limitation of local storage is that it is not clustered or replicated. Thus, it is not arbitrarily scalable or durable in the face of drive or node outages and should be managed like any other single node database.

Snapshots are recommended for backups. Backups made without snapshots run the risk of losing data that was recorded since the last WAL sync, which typically happens every two hours. With proper architecture, it is possible to retain years of data in local storage.

Alternatively, external storage may be used via the remote read/write APIs. Careful evaluation is required for these systems as they vary greatly in durability, performance, and efficiency.

For further details on file format, see TSDB format.

Compaction

The initial two-hour blocks are eventually compacted into longer blocks in the background.

Prometheus

Operational aspects

Prometheus has several flags that configure local storage. The most important are:

- `--storage.tsdb.path` : Where Prometheus writes its database. Defaults to `data/` .
- `--storage.tsdb.retention.time` : How long to retain samples in storage. If neither this flag nor `storage.tsdb.retention.size` is set, the retention time defaults to `15d` . Supported units: y, w, d, h, m, s, ms.
- `--storage.tsdb.retention.size` : The maximum number of bytes of storage blocks to retain. The oldest data will be removed first. Defaults to `0` or disabled. Units supported: B, KB, MB, GB, TB, PB, EB. Ex: "512MB". Based on powers-of-2, so 1KB is 1024B. Only the persistent blocks are deleted to honor this retention although WAL and m-mapped chunks are counted in the total size. So the minimum requirement for the disk is the peak space taken by the `wal` (the WAL and Checkpoint) and `chunks_head` (m-mapped Head chunks) directory combined (peaks every 2 hours).
- `--storage.tsdb.wal-compression` : Enables compression of the write-ahead log (WAL). Depending on your data, you can expect the WAL size to be halved with little extra cpu load. This flag was introduced in 2.11.0 and enabled by default in 2.20.0. Note that once enabled, downgrading Prometheus to a version below 2.11.0 will require deleting the WAL.

Prometheus stores an average of only 1-2 bytes per sample. Thus, to plan the capacity of a Prometheus server, you can use the rough formula:

```
needed_disk_space = retention_time_seconds * ingested_samples_per_second * bytes_pe
```

To lower the rate of ingested samples, you can either reduce the number of time series you scrape (fewer targets or fewer series per target), or you can increase the scrape interval. However, reducing the number of series is likely more effective, due to compression of samples within a series.

If your local storage becomes corrupted to the point where Prometheus will not start it is recommended to backup the storage directory and restore the corrupted block directories from your backups. If you do not have backups the last resort is to remove the corrupted files. For example you can try removing individual block directories or the write-ahead-log (wal) files. Note that this means losing the data for the time range those blocks or wal covers.

> (including AWS's EFS) are not supported. NFS could be POSIX-compliant, but most implementations are not. It is strongly recommended to use a local filesystem for reliability.

If both time and size retention policies are specified, whichever triggers first will be used.

Expired block cleanup happens in the background. It may take up to two hours to remove expired blocks. Blocks must be fully expired before they are removed.

Right-Sizing Retention Size

If you are utilizing `storage.tsdb.retention.size` to set a size limit, you will want to consider the right size for this value relative to the storage you have allocated for Prometheus. It is wise to reduce the retention size to provide a buffer, ensuring that older entries will be removed before the allocated storage for Prometheus becomes full.

At present, we recommend setting the retention size to, at most, 80-85% of your allocated Prometheus disk space. This increases the likelihood that older entries will be removed prior to hitting any disk limitations.

Remote storage integrations

Prometheus's local storage is limited to a single node's scalability and durability. Instead of trying to solve clustered storage in Prometheus itself, Prometheus offers a set of interfaces that allow integrating with remote storage systems.

Overview

Prometheus integrates with remote storage systems in four ways:

- Prometheus can write samples that it ingests to a remote URL in a Remote Write format.
- Prometheus can receive samples from other clients in a Remote Write format.
- Prometheus can read (back) sample data from a remote URL in a Remote Read format.
- Prometheus can return sample data requested by clients in a Remote Read format.

Prometheus

The remote read and write protocols both use a snappy-compressed protocol buffer encoding over HTTP. The read protocol is not yet considered as stable API.

The write protocol has a stable specification for 1.0 version and experimental specification for 2.0 version, both supported by Prometheus server.

For details on configuring remote storage integrations in Prometheus as a client, see the remote write and remote read sections of the Prometheus configuration documentation.

Note that on the read path, Prometheus only fetches raw series data for a set of label selectors and time ranges from the remote end. All PromQL evaluation on the raw data still happens in Prometheus itself. This means that remote read queries have some scalability limit, since all necessary data needs to be loaded into the querying Prometheus server first and then processed there. However, supporting fully distributed evaluation of PromQL was deemed infeasible for the time being.

Prometheus also serves both protocols. The built-in remote write receiver can be enabled by setting the `--web.enable-remote-write-receiver` command line flag. When enabled, the remote write receiver endpoint is `/api/v1/write`. The remote read endpoint is available on `/api/v1/read`.

Existing integrations

To learn more about existing integrations with remote storage systems, see the Integrations documentation.

Backfilling from OpenMetrics format

Overview

If a user wants to create blocks into the TSDB from data that is in OpenMetrics format, they can do so using backfilling. However, they should be careful and note that it is not safe to backfill data from the last 3 hours (the current head block) as this time range may overlap with the current head block Prometheus is still mutating. Backfilling will create new TSDB blocks, each containing two hours of metrics data. This limits the memory requirements of block creation. Compacting the two hour blocks into larger blocks is later done by the Prometheus server itself.

Prometheus

Note that native histograms and staleness markers are not supported by this procedure, as they cannot be represented in the OpenMetrics format.

Usage

Backfilling can be used via the Promtool command line. Promtool will write the blocks to a directory. By default this output directory is ./data/, you can change it by using the name of the desired output directory as an optional argument in the sub-command.

```
promtool tsdb create-blocks-from openmetrics <input file> [<output directory>]
```

After the creation of the blocks, move it to the data directory of Prometheus. If there is an overlap with the existing blocks in Prometheus, the flag `--storage.tsdb.allow-overlapping-blocks` needs to be set for Prometheus versions v2.38 and below. Note that any backfilled data is subject to the retention configured for your Prometheus server (by time or size).

Longer Block Durations

By default, the promtool will use the default block duration (2h) for the blocks; this behavior is the most generally applicable and correct. However, when backfilling data over a long range of times, it may be advantageous to use a larger value for the block duration to backfill faster and prevent additional compactions by TSDB later.

The `--max-block-duration` flag allows the user to configure a maximum duration of blocks. The backfilling tool will pick a suitable block duration no larger than this.

While larger blocks may improve the performance of backfilling large datasets, drawbacks exist as well. Time-based retention policies must keep the entire block around if even one sample of the (potentially large) block is still within the retention policy. Conversely, size-based retention policies will remove the entire block even if the TSDB only goes over the size limit in a minor way.

Therefore, backfilling with few blocks, thereby choosing a larger block duration, must be done with care and is not recommended for any production instances.

Backfilling for Recording Rules

only exists from the creation time on. `promtool` makes it possible to create historical recording rule data.

Usage

To see all options, use: `$ promtool tsdb create-blocks-from rules --help`.

Example usage:

```
$ promtool tsdb create-blocks-from rules \
    --start 1617079873 \
    --end 1617097873 \
    --url http://mypromserver.com:9090 \
    rules.yaml rules2.yaml
```

The recording rule files provided should be a normal Prometheus rules file.

The output of `promtool tsdb create-blocks-from rules` command is a directory that contains blocks with the historical rule data for all rules in the recording rule files. By default, the output directory is `data/`. In order to make use of this new block data, the blocks must be moved to a running Prometheus instance data dir `storage.tsdb.path` (for Prometheus versions v2.38 and below, the flag `--storage.tsdb.allow-overlapping-blocks` must be enabled). Once moved, the new blocks will merge with existing blocks when the next compaction runs.

Limitations

- If you run the rule backfiller multiple times with the overlapping start/end times, blocks containing the same data will be created each time the rule backfiller is run.
- All rules in the recording rule files will be evaluated.
- If the `interval` is set in the recording rule file that will take priority over the `eval-interval` flag in the rule backfill command.
- Alerts are currently ignored if they are in the recording rule file.
- Rules in the same group cannot see the results of previous rules. Meaning that rules that refer to other rules being backfilled is not supported. A workaround is to backfill multiple times and create the dependent data first (and move dependent data to the Prometheus server data dir so that it is accessible from the Prometheus API).

Prometheus

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Federation

Federation allows a Prometheus server to scrape selected time series from another Prometheus server.

Note about native histograms (experimental feature): To scrape native histograms via federation, the scraping Prometheus server needs to run with native histograms enabled (via the command line flag `--enable-feature=native-histograms` *), implying that the protobuf format is used for scraping. Should the federated metrics contain a mix of different sample types (float64, counter histogram, gauge histogram) for the same metric name, the federation payload will contain multiple metric families with the same name (but different types). Technically, this violates the rules of the protobuf exposition format, but Prometheus is nevertheless able to ingest all metrics correctly.*

Use cases

There are different use cases for federation. Commonly, it is used to either achieve scalable Prometheus monitoring setups or to pull related metrics from one service's Prometheus into another.

Hierarchical federation

Hierarchical federation allows Prometheus to scale to environments with tens of data centers and millions of nodes. In this use case, the federation topology resembles a tree, with higher-level Prometheus servers collecting aggregated time series data from a larger number of subordinated servers.

For example, a setup might consist of many per-datacenter Prometheus servers that collect data in high detail (instance-level drill-down), and a set of global Prometheus servers which

Prometheus

Cross-service federation

In cross-service federation, a Prometheus server of one service is configured to scrape selected data from another service's Prometheus server to enable alerting and queries against both datasets within a single server.

For example, a cluster scheduler running multiple services might expose resource usage information (like memory and CPU usage) about service instances running on the cluster. On the other hand, a service running on that cluster will only expose application-specific service metrics. Often, these two sets of metrics are scraped by separate Prometheus servers. Using federation, the Prometheus server containing service-level metrics may pull in the cluster resource usage metrics about its specific service from the cluster Prometheus, so that both sets of metrics can be used within that server.

Configuring federation

On any given Prometheus server, the `/federate` endpoint allows retrieving the current value for a selected set of time series in that server. At least one `match[]` URL parameter must be specified to select the series to expose. Each `match[]` argument needs to specify an instant vector selector like `up` or `{job="api-server"}`. If multiple `match[]` parameters are provided, the union of all matched series is selected.

To federate metrics from one server to another, configure your destination Prometheus server to scrape from the `/federate` endpoint of a source server, while also enabling the `honor_labels` scrape option (to not overwrite any labels exposed by the source server) and passing in the desired `match[]` parameters. For example, the following `scrape_configs` federates any series with the label `job="prometheus"` or a metric name starting with `job:` from the Prometheus servers at `source-prometheus-{1,2,3}:9090` into the scraping Prometheus:

```yaml
scrape_configs:
  - job_name: 'federate'
    scrape_interval: 15s

    honor_labels: true
    metrics_path: '/federate'

    params:
    static_configs:
      - targets:
        - 'source-prometheus-1:9090'
        - 'source-prometheus-2:9090'
        - 'source-prometheus-3:9090'
```

Previous
Storage

Edit

Next
HTTP SD

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Writing HTTP Service Discovery

Prometheus provides a generic HTTP Service Discovery, that enables it to discover targets over an HTTP endpoint.

The HTTP Service Discovery is complementary to the supported service discovery mechanisms, and is an alternative to File-based Service Discovery.

Comparison between File-Based SD and HTTP SD

Here is a table comparing our two generic Service Discovery implementations.

Item	File SD	HTTP SD
Event Based	Yes, via inotify	No
Update frequency	Instant, thanks to inotify	Following refresh_interval
Format	Yaml or JSON	JSON
Transport	Local file	HTTP/HTTPS
Security	File-Based security	TLS, Basic auth, Authorization header, OAuth2

Requirements of HTTP SD endpoints

If you implement an HTTP SD endpoint, here are a few requirements you should be aware of.

The response is consumed as is, unmodified. On each refresh interval (default: 1 minute), Prometheus will perform a GET request to the HTTP SD endpoint. The GET request contains a `X-Prometheus-Refresh-Interval-Seconds` HTTP header with the refresh interval.

unordered.

Prometheus caches target lists. If an error occurs while fetching an updated targets list, Prometheus keeps using the current targets list. The targets list is not saved across restart. The `prometheus_sd_http_failures_total` counter metric tracks the number of refresh failures.

The whole list of targets must be returned on every scrape. There is no support for incremental updates. A Prometheus instance does not send its hostname and it is not possible for a SD endpoint to know if the SD requests is the first one after a restart or not.

The URL to the HTTP SD is not considered secret. The authentication and any API keys should be passed with the appropriate authentication mechanisms. Prometheus supports TLS authentication, basic authentication, OAuth2, and authorization headers.

HTTP_SD format

```
[
  {
    "targets": [ "<host>", ... ],
    "labels": {
      "<labelname>": "<labelvalue>", ...
    }
  },
  ...
]
```

Examples:

```
[
    {
        "targets": ["10.0.10.2:9100", "10.0.10.3:9100", "10.0.10.4:9100", "10.0.10.
        "labels": {
            "__meta_datacenter": "london",
            "__meta_prometheus_job": "node"
        }
    },
    {
        "targets": ["10.0.40.2:9100", "10.0.40.3:9100"],
        }
    },
    {
        "targets": ["10.0.40.2:9093", "10.0.40.3:9093"],
        "labels": {
            "__meta_datacenter": "newyork",
            "__meta_prometheus_job": "alertmanager"
        }
    }
]
```

Previous
Federation

Edit

Next
Management API

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Management API

Prometheus provides a set of management APIs to facilitate automation and integration.

Health check

```
GET /-/healthy
HEAD /-/healthy
```

This endpoint always returns 200 and should be used to check Prometheus health.

Readiness check

```
GET /-/ready
HEAD /-/ready
```

This endpoint returns 200 when Prometheus is ready to serve traffic (i.e. respond to queries).

Reload

```
PUT  /-/reload
POST /-/reload
```

This endpoint triggers a reload of the Prometheus configuration and rule files. It's disabled by default and can be enabled via the `--web.enable-lifecycle` flag.

Prometheus

Quit

```
PUT  /-/quit
POST /-/quit
```

This endpoint triggers a graceful shutdown of Prometheus. It's disabled by default and can be enabled via the `--web.enable-lifecycle` flag.

Alternatively, a graceful shutdown can be triggered by sending a `SIGTERM` to the Prometheus process.

← **Previous** HTTP SD

Edit

Next prometheus →

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

prometheus

The Prometheus monitoring server

Flags

Flag	Description	Default
`-h`, `--help`	Show context-sensitive help (also try --help-long and --help-man).	
`--version`	Show application version.	
`--config.file`	Prometheus configuration file path.	`prometheus.yml`
`--config.auto-reload-interval`	Specifies the interval for checking and automatically reloading the Prometheus configuration file upon detecting changes.	`30s`
`--web.listen-address ...`	Address to listen on for UI, API, and telemetry. Can be repeated.	`0.0.0.0:9090`
`--auto-gomaxprocs`	Automatically set GOMAXPROCS to match Linux container CPU quota	`true`
`--auto-gomemlimit`	Automatically set GOMEMLIMIT to match Linux container or system memory limit	`true`
`--auto-gomemlimit.ratio`	The ratio of reserved GOMEMLIMIT memory to the detected maximum container or system memory	`0.9`
`--web.config.file`	[EXPERIMENTAL] Path to configuration file that can enable TLS or authentication.	
`--web.read-timeout`	Maximum duration before timing out read of the request, and closing idle connections.	`5m`
`--web.max-connections`	Maximum number of simultaneous connections across all listeners.	`512`
`--web.max-notifications-subscribers`	Limits the maximum number of subscribers that can concurrently receive live notifications. If the limit is reached, new subscription requests will be denied until existing connections close.	`16`

`--web.external-url`	example, if Prometheus is served via a reverse proxy). Used for generating relative and absolute links back to Prometheus itself. If the URL has a path portion, it will be used to prefix all HTTP endpoints served by Prometheus. If omitted, relevant URL components will be derived automatically.	
`--web.route-prefix`	Prefix for the internal routes of web endpoints. Defaults to path of --web.external-url.	
`--web.user-assets`	Path to static asset directory, available at /user.	
`--web.enable-lifecycle`	Enable shutdown and reload via HTTP request.	`false`
`--web.enable-admin-api`	Enable API endpoints for admin control actions.	`false`
`--web.enable-remote-write-receiver`	Enable API endpoint accepting remote write requests.	`false`
`--web.remote-write-receiver.accepted-protobuf-messages`	List of the remote write protobuf messages to accept when receiving the remote writes. Supported values: prometheus.WriteRequest, io.prometheus.write.v2.Request	`prometheus.WriteRequest`
`--web.enable-otlp-receiver`	Enable API endpoint accepting OTLP write requests.	`false`
`--web.console.templates`	Path to the console template directory, available at /consoles.	`consoles`
`--web.console.libraries`	Path to the console library directory.	`console_libraries`
`--web.page-title`	Document title of Prometheus instance.	`Prometheus Time Series Collection and Processing Server`
`--web.cors.origin`	Regex for CORS origin. It is fully anchored. Example: 'https?://(domain1|domain2).com'	`.*`
`--storage.tsdb.path`	Base path for metrics storage. Use with server mode only.	`data/`
`--storage.tsdb.retention.time`	How long to retain samples in storage. If neither this flag nor "storage.tsdb.retention.size" is set, the retention time defaults to 15d. Units Supported: y, w, d, h, m, s, ms. Use with server mode only.	
`--storage.tsdb.retention.size`	Maximum number of bytes that can be stored for blocks. A unit is required, supported units: B, KB, MB, GB, TB, PB, EB. Ex: "512MB". Based on powers-of-2, so 1KB is 1024B. Use with server mode only.	
`--storage.tsdb.no-lockfile`	Do not create lockfile in data directory. Use with server mode only.	`false`
`--storage.tsdb.head-chunks-write-queue-size`	Size of the queue through which head chunks are written to the disk to be m-mapped, 0 disables the queue completely. Experimental. Use with server mode only.	`0`
`--storage.agent.path`	Base path for metrics storage. Use with agent mode only.	`data-agent/`

compression		
`--storage.agent.retention.min-time`	Minimum age samples may be before being considered for deletion when the WAL is truncated Use with agent mode only.	
`--storage.agent.retention.max-time`	Maximum age samples may be before being forcibly deleted when the WAL is truncated Use with agent mode only.	
`--storage.agent.no-lockfile`	Do not create lockfile in data directory. Use with agent mode only.	`false`
`--storage.remote.flush-deadline`	How long to wait flushing sample on shutdown or config reload.	`1m`
`--storage.remote.read-sample-limit`	Maximum overall number of samples to return via the remote read interface, in a single query. 0 means no limit. This limit is ignored for streamed response types. Use with server mode only.	`5e7`
`--storage.remote.read-concurrent-limit`	Maximum number of concurrent remote read calls. 0 means no limit. Use with server mode only.	`10`
`--storage.remote.read-max-bytes-in-frame`	Maximum number of bytes in a single frame for streaming remote read response types before marshalling. Note that client might have limit on frame size as well. 1MB as recommended by protobuf by default. Use with server mode only.	`1048576`
`--rules.alert.for-outage-tolerance`	Max time to tolerate prometheus outage for restoring "for" state of alert. Use with server mode only.	`1h`
`--rules.alert.for-grace-period`	Minimum duration between alert and restored "for" state. This is maintained only for alerts with configured "for" time greater than grace period. Use with server mode only.	`10m`
`--rules.alert.resend-delay`	Minimum amount of time to wait before resending an alert to Alertmanager. Use with server mode only.	`1m`
`--rules.max-concurrent-evals`	Global concurrency limit for independent rules that can run concurrently. When set, "query.max-concurrency" may need to be adjusted accordingly. Use with server mode only.	`4`
`--alertmanager.notification-queue-capacity`	The capacity of the queue for pending Alertmanager notifications. Use with server mode only.	`10000`
`--alertmanager.drain-notification-queue-on-shutdown`	Send any outstanding Alertmanager notifications when shutting down. If false, any outstanding Alertmanager notifications will be dropped when shutting down. Use with server mode only.	`true`
`--query.lookback-delta`	The maximum lookback duration for retrieving metrics during expression evaluations and federation. Use with server mode only.	`5m`

	with server mode only.	
`--query.max-concurrency`	Maximum number of queries executed concurrently. Use with server mode only.	`20`
`--query.max-samples`	Maximum number of samples a single query can load into memory. Note that queries will fail if they try to load more samples than this into memory, so this also limits the number of samples a query can return. Use with server mode only.	`50000000`
`--enable-feature ...`	Comma separated feature names to enable. Valid options: exemplar-storage, expand-external-labels, memory-snapshot-on-shutdown, promql-per-step-stats, promql-experimental-functions, extra-scrape-metrics, auto-gomaxprocs, native-histograms, created-timestamp-zero-ingestion, concurrent-rule-eval, delayed-compaction, old-ui, otlp-deltatocumulative. See https://prometheus.io/docs/prometheus/latest/feature_flags/ for more details.	
`--agent`	Run Prometheus in 'Agent mode'.	
`--log.level`	Only log messages with the given severity or above. One of: [debug, info, warn, error]	`info`
`--log.format`	Output format of log messages. One of: [logfmt, json]	`logfmt`

← **Previous** Management API

Edit

Next promtool →

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

promtool

Tooling for the Prometheus monitoring system.

Flags

Flag	Description
`-h`, `--help`	Show context-sensitive help (also try --help-long and --help-man).
`--version`	Show application version.
`--experimental`	Enable experimental commands.
`--enable-feature ...`	Comma separated feature names to enable. Currently unused.

Commands

Command	Description
help	Show help.
check	Check the resources for validity.
query	Run query against a Prometheus server.
debug	Fetch debug information.
push	Push to a Prometheus server.
test	Unit testing.
tsdb	Run tsdb commands.

`promtool help`

Show help.

Arguments

Argument	Description
command	Show help on command.

`promtool check`

Check the resources for validity.

Flags

Flag	Description	Default
`--query.lookback-delta`	The server's maximum query lookback duration.	`5m`
`--extended`	Print extended information related to the cardinality of the metrics.	

`promtool check service-discovery`

Perform service discovery for the given job name and report the results, including relabeling.

Flags

Flag	Description	Default
`--timeout`	The time to wait for discovery results.	`30s`

Arguments

Prometheus

job	The job to run service discovery for.	Yes

`promtool check config`

Check if the config files are valid or not.

Flags

Flag	Description	Default
`--syntax-only`	Only check the config file syntax, ignoring file and content validation referenced in the config	
`--lint`	Linting checks to apply to the rules/scrape configs specified in the config. Available options are: all, duplicate-rules, none, too-long-scrape-interval. Use --lint=none to disable linting	`duplicate-rules`
`--lint-fatal`	Make lint errors exit with exit code 3.	`false`
`--ignore-unknown-fields`	Ignore unknown fields in the rule groups read by the config files. This is useful when you want to extend rule files with custom metadata. Ensure that those fields are removed before loading them into the Prometheus server as it performs strict checks by default.	`false`
`--agent`	Check config file for Prometheus in Agent mode.	

Arguments

Argument	Description	Required
config-files	The config files to check.	Yes

`promtool check web-config`

Check if the web config files are valid or not.

Arguments

`promtool check healthy`

Check if the Prometheus server is healthy.

Flags

Flag	Description	Default
`--http.config.file`	HTTP client configuration file for promtool to connect to Prometheus.	
`--url`	The URL for the Prometheus server.	`http://localhost:9090`

`promtool check ready`

Check if the Prometheus server is ready.

Flags

Flag	Description	Default
`--http.config.file`	HTTP client configuration file for promtool to connect to Prometheus.	
`--url`	The URL for the Prometheus server.	`http://localhost:9090`

`promtool check rules`

Check if the rule files are valid or not.

Flags

Flag	Description	Default
`--lint`	Linting checks to apply. Available options are: all, duplicate-rules, none. Use --lint=none to disable linting	`duplicate-rules`

`fatal`		
`--ignore-unknown-fields`	Ignore unknown fields in the rule files. This is useful when you want to extend rule files with custom metadata. Ensure that those fields are removed before loading them into the Prometheus server as it performs strict checks by default.	`false`

Arguments

Argument	Description
rule-files	The rule files to check, default is read from standard input.

`promtool check metrics`

Pass Prometheus metrics over stdin to lint them for consistency and correctness.

examples:

$ cat metrics.prom | promtool check metrics

$ curl -s http://localhost:9090/metrics | promtool check metrics

`promtool query`

Run query against a Prometheus server.

Flags

Flag	Description	Default
`-o`, `--format`	Output format of the query.	`promql`
`--http.config.file`	HTTP client configuration file for promtool to connect to Prometheus.	

`promtool query instant`

Run instant query.

`--time`	Query evaluation time (RFC3339 or Unix timestamp).

Arguments

Argument	Description	Required
server	Prometheus server to query.	Yes
expr	PromQL query expression.	Yes

`promtool query range`

Run range query.

Flags

Flag	Description
`--header`	Extra headers to send to server.
`--start`	Query range start time (RFC3339 or Unix timestamp).
`--end`	Query range end time (RFC3339 or Unix timestamp).
`--step`	Query step size (duration).

Arguments

Argument	Description	Required
server	Prometheus server to query.	Yes
expr	PromQL query expression.	Yes

`promtool query series`

Run series query.

Flags

`--start`	Start time (RFC3339 or Unix timestamp).
`--end`	End time (RFC3339 or Unix timestamp).

Arguments

Argument	Description	Required
server	Prometheus server to query.	Yes

`promtool query labels`

Run labels query.

Flags

Flag	Description
`--start`	Start time (RFC3339 or Unix timestamp).
`--end`	End time (RFC3339 or Unix timestamp).
`--match ...`	Series selector. Can be specified multiple times.

Arguments

Argument	Description	Required
server	Prometheus server to query.	Yes
name	Label name to provide label values for.	Yes

`promtool query analyze`

Run queries against your Prometheus to analyze the usage pattern of certain metrics.

Flags

`--type`	Type of metric: histogram.	
`--duration`	Time frame to analyze.	`1h`
`--time`	Query time (RFC3339 or Unix timestamp), defaults to now.	
`--match ...`	Series selector. Can be specified multiple times.	

`promtool debug`

Fetch debug information.

`promtool debug pprof`

Fetch profiling debug information.

Arguments

Argument	Description	Required
server	Prometheus server to get pprof files from.	Yes

`promtool debug metrics`

Fetch metrics debug information.

Arguments

Argument	Description	Required
server	Prometheus server to get metrics from.	Yes

`promtool debug all`

Fetch all debug information.

Arguments

`promtool push`

Push to a Prometheus server.

Flags

Flag	Description
`--http.config.file`	HTTP client configuration file for promtool to connect to Prometheus.

`promtool push metrics`

Push metrics to a prometheus remote write (for testing purpose only).

Flags

Flag	Description	Default
`--label`	Label to attach to metrics. Can be specified multiple times.	`job=promtool`
`--timeout`	The time to wait for pushing metrics.	`30s`
`--header`	Prometheus remote write header.	

Arguments

Argument	Description	Required
remote-write-url	Prometheus remote write url to push metrics.	Yes
metric-files	The metric files to push, default is read from standard input.	

`promtool test`

Unit testing.

Flags

`promtool test rules`

Unit tests for rules.

Flags

Flag	Description	Default
`--run ...`	If set, will only run test groups whose names match the regular expression. Can be specified multiple times.	
`--debug`	Enable unit test debugging.	`false`
`--diff`	[Experimental] Print colored differential output between expected & received output.	`false`
`--ignore-unknown-fields`	Ignore unknown fields in the test files. This is useful when you want to extend rule files with custom metadata. Ensure that those fields are removed before loading them into the Prometheus server as it performs strict checks by default.	`false`

Arguments

Argument	Description	Required
test-rule-file	The unit test file.	Yes

`promtool tsdb`

Run tsdb commands.

`promtool tsdb bench`

Run benchmarks.

`promtool tsdb bench write`

Run a write performance benchmark.

Prometheus

`--out`	Set the output path.	`benchout`
`--metrics`	Number of metrics to read.	`10000`
`--scrapes`	Number of scrapes to simulate.	`3000`

Arguments

Argument	Description	Default
file	Input file with samples data, default is (../../tsdb/testdata/20kseries.json).	`../../tsdb/testdata/20kseries.json`

`promtool tsdb analyze`

Analyze churn, label pair cardinality and compaction efficiency.

Flags

Flag	Description	Default
`--limit`	How many items to show in each list.	`20`
`--extended`	Run extended analysis.	
`--match`	Series selector to analyze. Only 1 set of matchers is supported now.	

Arguments

Argument	Description	Default
db path	Database path (default is data/).	`data/`
block id	Block to analyze (default is the last block).	

`promtool tsdb list`

List tsdb blocks.

Prometheus

`-r`, `--human-readable`	Print human readable values.

Arguments

Argument	Description	Default
db path	Database path (default is data/).	`data/`

`promtool tsdb dump`

Dump samples from a TSDB.

Flags

Flag	Description	Default
`--sandbox-dir-root`	Root directory where a sandbox directory will be created, this sandbox is used in case WAL replay generates chunks (default is the database path). The sandbox is cleaned up at the end.	
`--min-time`	Minimum timestamp to dump.	`-9223372036854775808`
`--max-time`	Maximum timestamp to dump.	`9223372036854775807`
`--match ...`	Series selector. Can be specified multiple times.	`{__name__=~'(?s:.*)'}`

Arguments

Argument	Description	Default
db path	Database path (default is data/).	`data/`

`promtool tsdb dump-openmetrics`

[Experimental] Dump samples from a TSDB into OpenMetrics text format, excluding native histograms and staleness markers, which are not representable in OpenMetrics.

`--sandbox-dir-root`	Root directory where a sandbox directory will be created, this sandbox is used in case WAL replay generates chunks (default is the database path). The sandbox is cleaned up at the end.	
`--min-time`	Minimum timestamp to dump.	`-9223372036854775808`
`--max-time`	Maximum timestamp to dump.	`9223372036854775807`
`--match ...`	Series selector. Can be specified multiple times.	`{__name__=~'(?s:.*)'}`

Arguments

Argument	Description	Default
db path	Database path (default is data/).	`data/`

`promtool tsdb create-blocks-from`

[Experimental] Import samples from input and produce TSDB blocks. Please refer to the storage docs for more details.

Flags

Flag	Description
`-r`, `--human-readable`	Print human readable values.
`-q`, `--quiet`	Do not print created blocks.

`promtool tsdb create-blocks-from openmetrics`

Import samples from OpenMetrics input and produce TSDB blocks. Please refer to the storage docs for more details.

Flags

`label`	label=label_name=label_value

Arguments

Argument	Description	Default	Required
input file	OpenMetrics file to read samples from.		Yes
output directory	Output directory for generated blocks.	`data/`	

`promtool tsdb create-blocks-from rules`

Create blocks of data for new recording rules.

Flags

Flag	Description	Default
`--http.config.file`	HTTP client configuration file for promtool to connect to Prometheus.	
`--url`	The URL for the Prometheus API with the data where the rule will be backfilled from.	`http://localhost:9090`
`--start`	The time to start backfilling the new rule from. Must be a RFC3339 formatted date or Unix timestamp. Required.	
`--end`	If an end time is provided, all recording rules in the rule files provided will be backfilled to the end time. Default will backfill up to 3 hours ago. Must be a RFC3339 formatted date or Unix timestamp.	
`--output-dir`	Output directory for generated blocks.	`data/`
`--eval-interval`	How frequently to evaluate rules when backfilling if a value is not set in the recording rule files.	`60s`

Arguments

rule-files	recording rules listed in the files will be backfilled. Alerting rules are not evaluated.	Yes

`promtool promql`

PromQL formatting and editing. Requires the `--experimental` flag.

`promtool promql format`

Format PromQL query to pretty printed form.

Arguments

Argument	Description	Required
query	PromQL query.	Yes

`promtool promql label-matchers`

Edit label matchers contained within an existing PromQL query.

`promtool promql label-matchers set`

Set a label matcher in the query.

Flags

Flag	Description	Default
`-t`, `--type`	Type of the label matcher to set.	`=`

Arguments

Argument	Description	Required
query	PromQL query.	Yes
name	Name of the label matcher to set.	Yes

`promtool promql label-matchers delete`

Delete a label from the query.

Arguments

Argument	Description	Required
query	PromQL query.	Yes
name	Name of the label to delete.	Yes

← Previous prometheus

Edit

Next Migration →

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Prometheus 3.0 migration guide

In line with our stability promise, the Prometheus 3.0 release contains a number of backwards incompatible changes. This document offers guidance on migrating from Prometheus 2.x to Prometheus 3.0 and newer versions.

Flags

- The following feature flags have been removed and they have been added to the default behavior of Prometheus v3:
  - `promql-at-modifier`
  - `promql-negative-offset`
  - `remote-write-receiver`
  - `new-service-discovery-manager`
  - `expand-external-labels`
    - Environment variable references `${var}` or `$var` in external label values are replaced according to the values of the current environment variables.
    - References to undefined variables are replaced by the empty string. The `$` character can be escaped by using `$$`.
  - `no-default-scrape-port`
    - Prometheus v3 will no longer add ports to scrape targets according to the specified scheme. Target will now appear in labels as configured.
    - If you rely on scrape targets like `https://example.com/metrics` or `http://exmaple.com/metrics` to be represented as `https://example.com/metrics:443` and `http://example.com/metrics:80` respectively, add them to your target URLs
  - `agent`
    - Instead use the dedicated `--agent` CLI flag.

running outside of containers, the system memory total is used. To disable this, `--no-auto-gomemlimit` is available.
    - `auto-gomaxprocs`
        - Prometheus v3 will automatically set `GOMAXPROCS` to match the Linux container CPU quota. To disable this, `--no-auto-gomaxprocs` is available.

Prometheus v3 will log a warning if you continue to pass these to `--enable-feature`.

Configuration

- The scrape job level configuration option `scrape_classic_histograms` has been renamed to `always_scrape_classic_histograms`. If you use the `--enable-feature=native-histograms` feature flag to ingest native histograms and you also want to ingest classic histograms that an endpoint might expose along with native histograms, be sure to add this configuration or change your configuration from the old name.
- The `http_config.enable_http2` in `remote_write` items default has been changed to `false`. In Prometheus v2 the remote write http client would default to use http2. In order to parallelize multiple remote write queues across multiple sockets its preferable to not default to http2. If you prefer to use http2 for remote write you must now set `http_config.enable_http2: true` in your `remote_write` configuration section.

PromQL

Regular expressions match newlines

The `.` pattern in regular expressions in PromQL matches newline characters. With this change a regular expressions like `.*` matches strings that include `\n`. This applies to matchers in queries and relabel configs.

For example, the following regular expressions now match the accompanying strings, whereas in Prometheus v2 these combinations didn't match. - `.*` additionally matches `foo\n` and `Foo\nBar` - `foo.?bar` additionally matches `foo\nbar` - `foo.+bar` additionally matches `foo\nbar`

If you want Prometheus v3 to behave like v2, you will have to change your regular expressions by replacing all `.` patterns with `[^\n]`, e.g. `foo[^\n]*`.

Lookback and range selectors are now left-open and right-closed (previously left-closed and right-closed), which makes their behavior more consistent. This change affects queries where the left boundary of a range or the lookback delta coincides with the timestamp of one or more samples.

For example, assume we are querying a timeseries with evenly spaced samples exactly 1 minute apart. Before Prometheus v3, a range query with `5m` would usually return 5 samples. But if the query evaluation aligns perfectly with a scrape, it would return 6 samples. In Prometheus v3 queries like this will always return 5 samples given even spacing.

This change will typically affect subqueries because their evaluation timing is naturally perfectly evenly spaced and aligned with timestamps that are multiples of the subquery resolution. Furthermore, query frontends often align subqueries to multiples of the step size. In combination, this easily creates a situation of perfect mutual alignment, often unintended and unknown by the user, so that the new behavior might come as a surprise. Before Prometheus V3, a subquery of `foo[1m:1m]` on such a system might have always returned two points, allowing for rate calculations. In Prometheus V3, however, such a subquery will only return one point, which is insufficient for a rate or increase calculation, resulting in No Data returned.

Such queries will need to be rewritten to extend the window to properly cover more than one point. In this example, `foo[2m:1m]` would always return two points no matter the query alignment. The exact form of the rewritten query may depend on the intended results and there is no universal drop-in replacement for queries whose behavior has changed.

Tests are similarly more likely to affected. To fix those either adjust the expected number of samples or extend the range.

holt_winters function renamed

The `holt_winters` function has been renamed to `double_exponential_smoothing` and is now guarded by the `promql-experimental-functions` feature flag. If you want to keep using `holt_winters`, you have to do both of these things:

- Rename `holt_winters` to `double_exponential_smoothing` in your queries.
- Pass `--enable-feature=promql-experimental-functions` in your Prometheus CLI invocation.

Scrape protocols

unrecognised. This could lead to incorrect data being parsed in the scrape. Prometheus v3 will now fail the scrape in such cases.

If a scrape target is not providing the correct Content-Type header the fallback protocol can be specified using the `fallback_scrape_protocol` parameter. See Prometheus scrape_config documentation.

This is a breaking change as scrapes that may have succeeded with Prometheus v2 may now fail if this fallback protocol is not specified.

Miscellaneous

TSDB format and downgrade

The TSDB format has been changed slightly in Prometheus v2.55 in preparation for changes to the index format. Consequently, a Prometheus v3 TSDB can only be read by a Prometheus v2.55 or newer. Keep that in mind when upgrading to v3 -- you will be only able to downgrade to v2.55, not lower, without losing your TSDB persitent data.

As an extra safety measure, you could optionally consider upgrading to v2.55 first and confirm Prometheus works as expected, before upgrading to v3.

TSDB storage contract

TSDB compatible storage is now expected to return results matching the specified selectors. This might impact some third party implementations, most likely implementing `remote_read`.

This contract is not explicitly enforced, but can cause undefined behavior.

UTF-8 names

Prometheus v3 supports UTF-8 in metric and label names. This means metric and label names can change after upgrading according to what is exposed by endpoints. Furthermore, metric and label names that would have previously been flagged as invalid no longer will be.

Users wishing to preserve the original validation behavior can update their Prometheus yaml configuration to specify the legacy validation scheme:

Or on a per-scrape basis:

```
scrape_configs:
  - job_name: job1
    metric_name_validation_scheme: utf8
  - job_name: job2
    metric_name_validation_scheme: legacy
```

Log message format

Prometheus v3 has adopted `log/slog` over the previous `go-kit/log`. This results in a change of log message format. An example of the old log format is:

```
ts=2024-10-23T22:01:06.074Z caller=main.go:627 level=info msg="No time or size reter
ts=2024-10-23T22:01:06.074Z caller=main.go:671 level=info msg="Starting Prometheus S
ts=2024-10-23T22:01:06.074Z caller=main.go:676 level=info build_context="(go=go1.23
ts=2024-10-23T22:01:06.074Z caller=main.go:677 level=info host_details="(Linux 5.15
```

a similar sequence in the new log format looks like this:

```
time=2024-10-24T00:03:07.542+02:00 level=INFO source=/home/user/go/src/github.com/pr
time=2024-10-24T00:03:07.542+02:00 level=INFO source=/home/user/go/src/github.com/pr
time=2024-10-24T00:03:07.542+02:00 level=INFO source=/home/user/go/src/github.com/pr
```

`le` and `quantile` label values

In Prometheus v3, the values of the `le` label of classic histograms and the `quantile` label of summaries are normalized upon ingestion. In Prometheus v2 the value of these labels depended on the scrape protocol (protobuf vs text format) in some situations. This led to label values changing based on the scrape protocol. E.g. a metric exposed as `my_classic_hist{le="1"}` would be ingested as `my_classic_hist{le="1"}` via the text format, but as `my_classic_hist{le="1.0"}` via protobuf. This changed the identity of the metric and caused problems when querying the metric. In Prometheus v3 these label values will always be normalized to a float like representation. I.e. the above example will always

Ways to deal with this change either globally or on a per metric basis:

- Fix references to integer `le`, `quantile` label values, but otherwise do nothing and accept that some queries that span the transition time will produce inaccurate or unexpected results. *This is the recommended solution.*
- Use `metric_relabel_config` to retain the old labels when scraping targets. This should **only** be applied to metrics that currently produce such labels.

```
metric_relabel_configs:
  - source_labels:
      - quantile
    target_label: quantile
    regex: (\d+)\.0+
  - source_labels:
      - le
      - __name__
    target_label: le
    regex: (\d+)\.0+;.*_bucket
```

Disallow configuring Alertmanager with the v1 API

Prometheus 3 no longer supports Alertmanager's v1 API. Effectively Prometheus 3 requires Alertmanager 0.16.0 or later. Users with older Alertmanager versions or configurations that use `alerting: alertmanagers: [api_version: v1]` need to upgrade Alertmanager and change their configuration to use `api_version: v2`.

Prometheus 2.0 migration guide

For the Prometheus 1.8 to 2.0 please refer to the Prometheus v2.55 documentation.

← Previous promtool

Edit

Next API Stability →

Prometheus

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

API Stability Guarantees

Prometheus promises API stability within a major version, and strives to avoid breaking changes for key features. Some features, which are cosmetic, still under development, or depend on 3rd party services, are not covered by this.

Things considered stable for 3.x:

- The query language and data model
- Alerting and recording rules
- The ingestion exposition format
- v1 HTTP API (used by dashboards and UIs)
- Configuration file format (minus the service discovery remote read/write, see below)
- Rule/alert file format
- Console template syntax and semantics
- Remote write sending, per the 1.0 specification and receiving
- Agent mode
- OTLP receiver endpoint

Things considered unstable for 3.x:

- Any feature listed as experimental or subject to change, including:
  - The `double_exponential_smoothing` PromQL function
  - Remote read and the remote read endpoint
- Server-side HTTPS and basic authentication
- Service discovery integrations, with the exception of `static_configs`, `file_sd_configs` and `http_sd_config`
- Go APIs of packages that are part of the server
- HTML generated by the web UI
- The metrics in the /metrics endpoint of Prometheus itself

Prometheus

Prometheus 2.x stability guarantees can be found in the 2.x documentation.

As long as you are not using any features marked as experimental/unstable, an upgrade within a major version can usually be performed without any operational adjustments and very little risk that anything will break. Any breaking changes will be marked as `CHANGE` in release notes.

Previous
Migration

Edit

Next
Feature flags

Prometheus

Show nav

Outdated version

This page documents version 3.2, which is outdated. Check out the latest stable version.

Feature flags

Here is a list of features that are disabled by default since they are breaking changes or are considered experimental. Their behaviour can change in future releases which will be communicated via the release changelog.

You can enable them using the `--enable-feature` flag with a comma separated list of features. They may be enabled by default in future versions.

Exemplars storage

```
--enable-feature=exemplar-storage
```

OpenMetrics introduces the ability for scrape targets to add exemplars to certain metrics. Exemplars are references to data outside of the MetricSet. A common use case are IDs of program traces.

Exemplar storage is implemented as a fixed size circular buffer that stores exemplars in memory for all series. Enabling this feature will enable the storage of exemplars scraped by Prometheus. The config file block storage/exemplars can be used to control the size of circular buffer by # of exemplars. An exemplar with just a `trace_id=<jaeger-trace-id>` uses roughly 100 bytes of memory via the in-memory exemplar storage. If the exemplar storage is enabled, we will also append the exemplars to WAL for local persistence (for WAL duration).

Memory snapshot on shutdown

```
--enable-feature=memory-snapshot-on-shutdown
```

This takes a snapshot of the chunks that are in memory along with the series information when shutting down and stores it on disk. This will reduce the startup time since the memory

Prometheus

Extra scrape metrics

`--enable-feature=extra-scrape-metrics`

When enabled, for each instance scrape, Prometheus stores a sample in the following additional time series:

- `scrape_timeout_seconds` . The configured `scrape_timeout` for a target. This allows you to measure each target to find out how close they are to timing out with `scrape_duration_seconds / scrape_timeout_seconds` .
- `scrape_sample_limit` . The configured `sample_limit` for a target. This allows you to measure each target to find out how close they are to reaching the limit with `scrape_samples_post_metric_relabeling / scrape_sample_limit` . Note that `scrape_sample_limit` can be zero if there is no limit configured, which means that the query above can return `+Inf` for targets with no limit (as we divide by zero). If you want to query only for targets that do have a sample limit use this query: `scrape_samples_post_metric_relabeling / (scrape_sample_limit > 0)` .
- `scrape_body_size_bytes` . The uncompressed size of the most recent scrape response, if successful. Scrapes failing because `body_size_limit` is exceeded report `-1` , other scrape failures report `0` .

Per-step stats

`--enable-feature=promql-per-step-stats`

When enabled, passing `stats=all` in a query request returns per-step statistics. Currently this is limited to totalQueryableSamples.

When disabled in either the engine or the query, per-step statistics are not computed at all.

Native Histograms

`--enable-feature=native-histograms`

When enabled, Prometheus will ingest native histograms (formerly also known as sparse histograms or high-res histograms). Native histograms are still highly experimental. Expect breaking changes to happen (including those rendering the TSDB unreadable).

Prometheus will try to negotiate the protobuf format first. The instrumented target needs to support the protobuf format, too, *and* it needs to expose native histograms. The protobuf format allows to expose classic and native histograms side by side. With this feature flag disabled, Prometheus will continue to parse the classic histogram (albeit via the text format). With this flag enabled, Prometheus will still ingest those classic histograms that do not come with a corresponding native histogram. However, if a native histogram is present, Prometheus will ignore the corresponding classic histogram, with the notable exception of exemplars, which are always ingested. To keep the classic histograms as well, enable `always_scrape_classic_histograms` in the scrape job.

Experimental PromQL functions

`--enable-feature=promql-experimental-functions`

Enables PromQL functions that are considered experimental. These functions might change their name, syntax, or semantics. They might also get removed entirely.

Created Timestamps Zero Injection

`--enable-feature=created-timestamp-zero-ingestion`

Enables ingestion of created timestamp. Created timestamps are injected as 0 valued samples when appropriate. See PromCon talk for details.

Currently Prometheus supports created timestamps only on the traditional Prometheus Protobuf protocol (WIP for other protocols). As a result, when enabling this feature, the Prometheus protobuf scrape protocol will be prioritized (See `scrape_config.scrape_protocols` settings for more details).

Besides enabling this feature in Prometheus, created timestamps need to be exposed by the application being scraped.

Concurrent evaluation of independent rules

`--enable-feature=concurrent-rule-eval`

By default, rule groups execute concurrently, but the rules within a group execute sequentially; this is because rules can use the output of a preceding rule as its input.

the potential to improve rule group evaluation latency and resource utilization at the expense of adding more concurrent query load.

The number of concurrent rule evaluations can be configured with `--rules.max-concurrent-rule-evals`, which is set to `4` by default.

Serve old Prometheus UI

Fall back to serving the old (Prometheus 2.x) web UI instead of the new UI. The new UI that was released as part of Prometheus 3.0 is a complete rewrite and aims to be cleaner, less cluttered, and more modern under the hood. However, it is not fully feature complete and battle-tested yet, so some users may still prefer using the old UI.

```
--enable-feature=old-ui
```

Metadata WAL Records

```
--enable-feature=metadata-wal-records
```

When enabled, Prometheus will store metadata in-memory and keep track of metadata changes as WAL records on a per-series basis.

This must be used if you are also using remote write 2.0 as it will only gather metadata from the WAL.

Delay compaction start time

```
--enable-feature=delayed-compaction
```

A random offset, up to `10%` of the chunk range, is added to the Head compaction start time. This assists Prometheus instances in avoiding simultaneous compactions and reduces the load on shared resources.

Only auto Head compactions and the operations directly resulting from them are subject to this delay.

In the event of multiple consecutive Head compactions being possible, only the first compaction experiences this delay.

Despite the delay in compaction, the blocks produced are time-aligned in the same manner as they would be if the delay was not in place.

Delay `__name__` label removal for PromQL engine

```
--enable-feature=promql-delayed-name-removal
```

When enabled, Prometheus will change the way in which the `__name__` label is removed from PromQL query results (for functions and expressions for which this is necessary). Specifically, it will delay the removal to the last step of the query evaluation, instead of every time an expression or function creating derived metrics is evaluated.

This allows optionally preserving the `__name__` label via the `label_replace` and `label_join` functions, and helps prevent the "vector cannot contain metrics with the same labelset" error, which can happen when applying a regex-matcher to the `__name__` label.

Note that evaluating parts of the query separately will still trigger the labelset collision. This commonly happens when analyzing intermediate results of a query manually or with a tool like PromLens.

If a query refers to the already removed `__name__` label, its behavior may change while this feature flag is set. (Example: `sum by (__name__) (rate({foo="bar"}[5m]))`, see details on GitHub.) These queries are rare to occur and easy to fix. (In the above example, removing `by (__name__)` doesn't change anything without the feature flag and fixes the possible problem with the feature flag.)

Auto Reload Config

```
--enable-feature=auto-reload-config
```

When enabled, Prometheus will automatically reload its configuration file at a specified interval. The interval is defined by the `--config.auto-reload-interval` flag, which defaults to `30s`.

Configuration reloads are triggered by detecting changes in the checksum of the main configuration file or any referenced files, such as rule and scrape configurations. To ensure consistency and avoid issues during reloads, it's recommended to update these files atomically.

When enabled, Prometheus will convert OTLP metrics from delta temporality to their cumulative equivalent, instead of dropping them.

This uses deltatocumulative from the OTel collector, using its default settings.

Delta conversion keeps in-memory state to aggregate delta changes per-series over time. When Prometheus restarts, this state is lost, starting the aggregation from zero again. This results in a counter reset in the cumulative series.

This state is periodically (`max_stale`) cleared of inactive series.

Enabling this *can* have negative impact on performance, because the in-memory state is mutex guarded. Cumulative-only OTLP requests are not affected.

Previous
API Stability

Edit

Next
Expression browser

Prometheus

Show nav

Expression browser

The expression browser is available at `/graph` on the Prometheus server, allowing you to enter any expression and see its result either in a table or graphed over time.

This is primarily useful for ad-hoc queries and debugging. For graphs, use Grafana or Console templates.

← Previous Feature flags

Edit

Next Grafana →

Prometheus

Prometheus

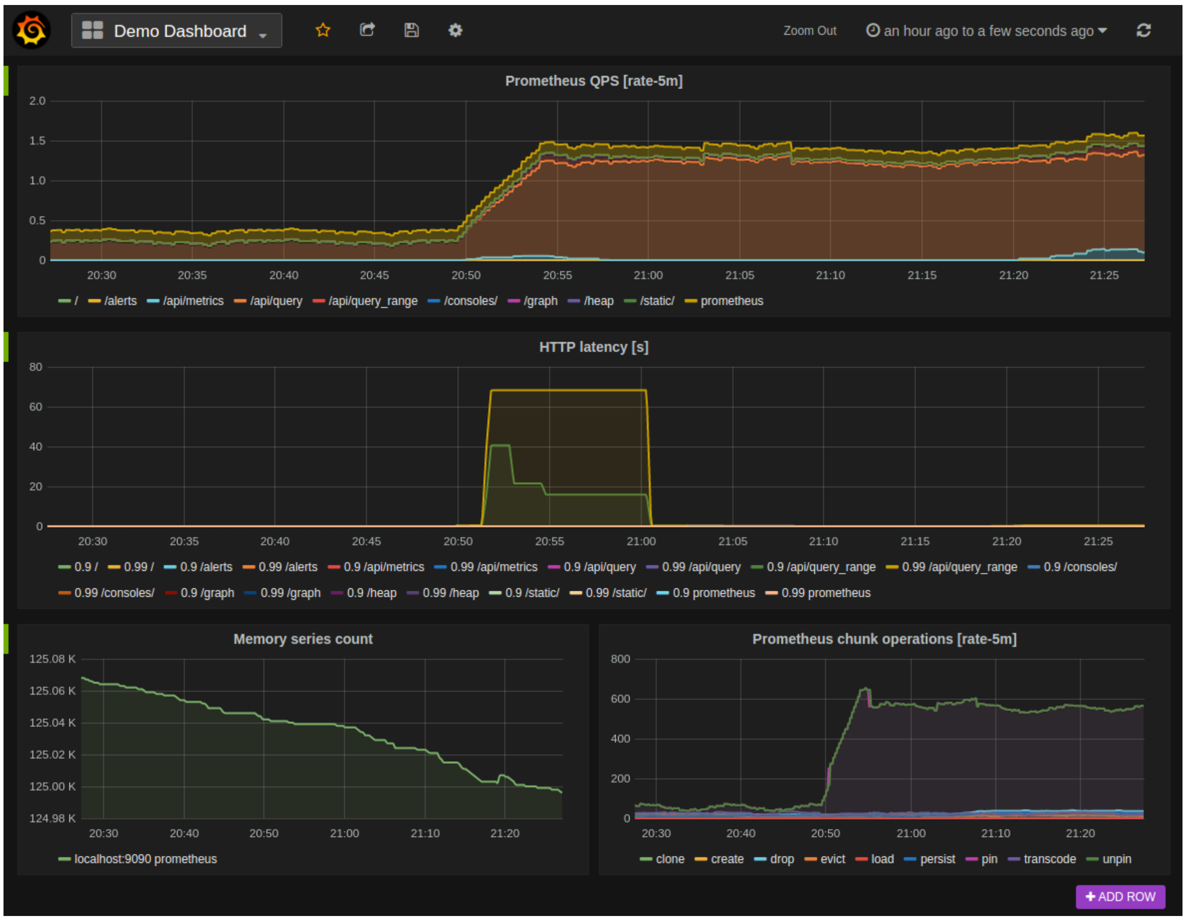
Show nav

Grafana support for Prometheus

Grafana is an open-source analytics and visualization platform used to monitor and analyze metrics from various data sources. It allows users to create, explore, and share interactive dashboards, supporting integrations with databases like Prometheus, InfluxDB, Elasticsearch, and more. Grafana is widely used for observability, providing alerting, plugin extensibility, and a flexible query editor for real-time data visualization.

Note: The Grafana data source for Prometheus is included since Grafana 2.5.0 (2015-10-28).

The following shows an example Grafana dashboard which queries Prometheus for data:


Demo Dashboard
Zoom Out
an hour ago to a few seconds ago
Prometheus QPS [rate-5m]
/ /alerts /api/metrics /api/query /api/query_range /consoles/ /graph /heap /static/ prometheus
HTTP latency [s]
0.9 / 0.99 / 0.9 /alerts 0.99 /alerts 0.9 /api/metrics 0.99 /api/metrics 0.9 /api/query 0.99 /api/query 0.9 /api/query_range 0.99 /api/query_range 0.9 /consoles/ 0.99 /consoles/ 0.9 /graph 0.99 /graph 0.9 /heap 0.99 /heap 0.9 /static/ 0.99 /static/ 0.9 prometheus 0.99 prometheus
Memory series count
localhost:9090 prometheus
Prometheus chunk operations [rate-5m]
clone create drop evict load persist pin transcode unpin
ADD ROW

Installing

To install Grafana see the official Grafana documentation.

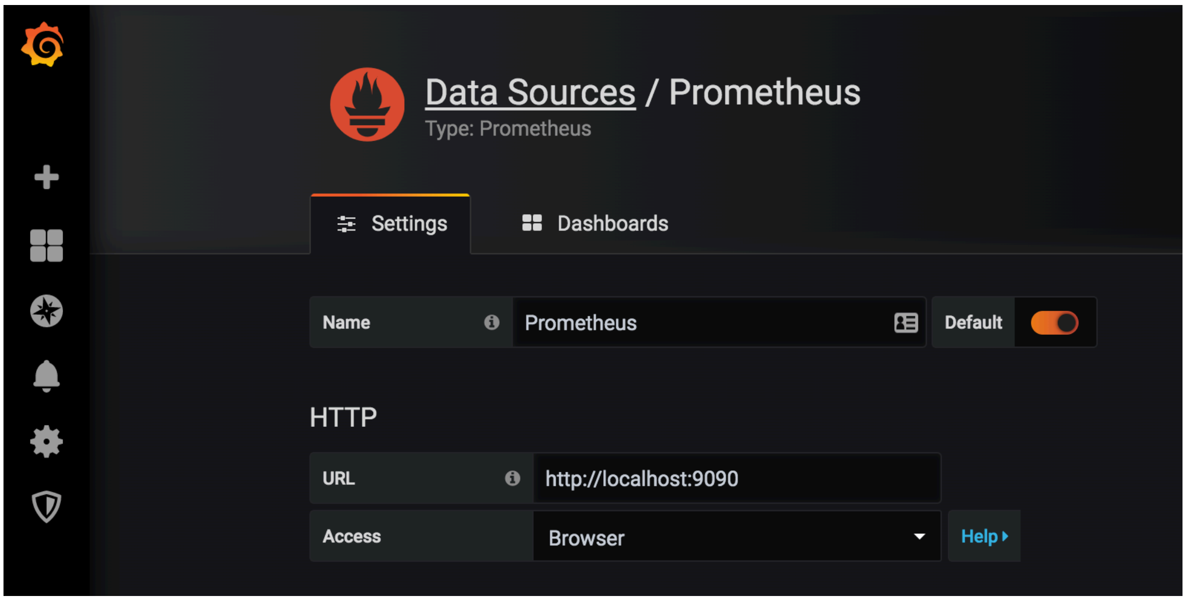
"admin".

Creating a Prometheus data source

To create a Prometheus data source in Grafana:

1. Click on the "cogwheel" in the sidebar to open the Configuration menu.
2. Click on "Data Sources".
3. Click on "Add data source".
4. Select "Prometheus" as the type.
5. Set the appropriate Prometheus server URL (for example, `http://localhost:9090/`)
6. Adjust other data source settings as desired (for example, choosing the right Access method).
7. Click "Save & Test" to save the new data source.

The following shows an example data source configuration:



Name: Prometheus
Default: on

HTTP
URL: http://localhost:9090
Access: Browser

Creating a Prometheus graph

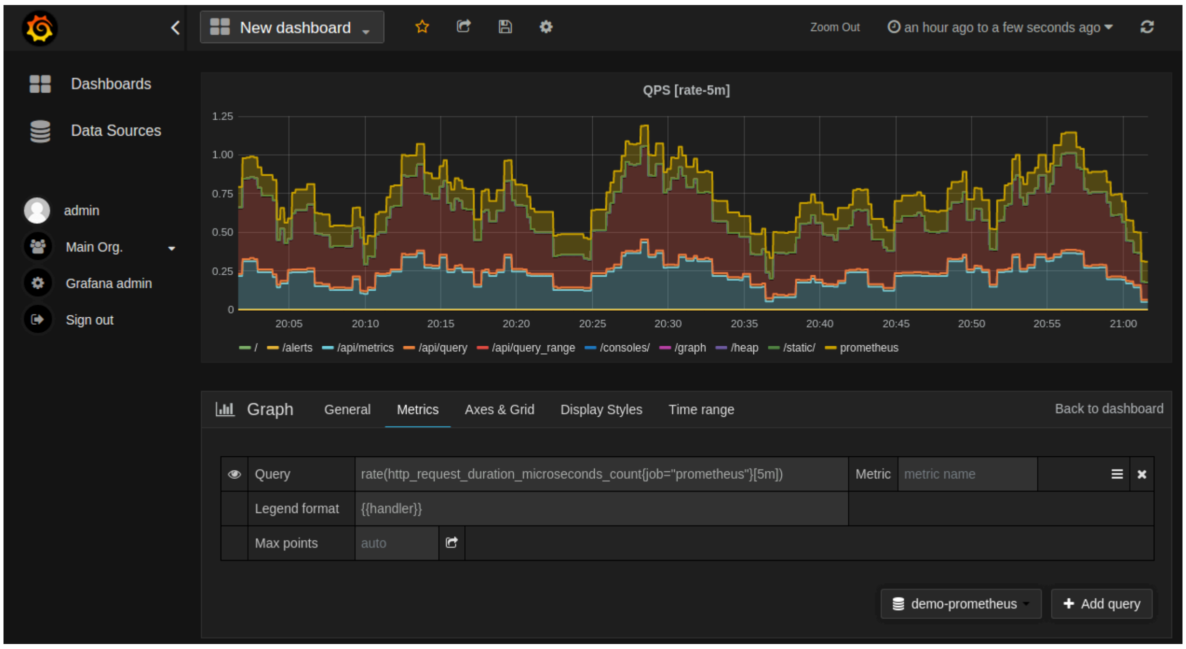
Follow the standard way of adding a new Grafana graph. Then:

1. Click the graph title, then click "Edit".
2. Under the "Metrics" tab, select your Prometheus data source (bottom right).
3. Enter any Prometheus expression into the "Query" field, while using the "Metric" field to lookup metrics via autocompletion.

`{{status}}`.

5. Tune other graph settings until you have a working graph.

The following shows an example Prometheus graph configuration:


New dashboard
Zoom Out
an hour ago to a few seconds ago
Dashboards
Data Sources
admin
Main Org.
Grafana admin
Sign out
QPS [rate-5m]
/ /alerts /api/metrics /api/query /api/query_range /consoles/ /graph /heap /static/ prometheus
Graph General Metrics Axes & Grid Display Styles Time range
Back to dashboard
Add query

Query: rate(http_request_duration_microseconds_count{job="prometheus"}[5m])
Metric: metric name
Legend format: {{handler}}
Max points: auto
demo-prometheus

In Grafana 7.2 and later, the `$__rate_interval` variable is recommended for use in the `rate` and `increase` functions.

Importing pre-built dashboards from Grafana.com

Grafana.com maintains a collection of shared dashboards which can be downloaded and used with standalone instances of Grafana. Use the Grafana.com "Filter" option to browse dashboards for the "Prometheus" data source only.

You must currently manually edit the downloaded JSON files and correct the `datasource:` entries to reflect the Grafana data source name which you chose for your Prometheus server. Use the "Dashboards" → "Home" → "Import" option to import the edited dashboard file into your Grafana install.

← **Previous** Expression browser

Edit

Next Perses →

Prometheus

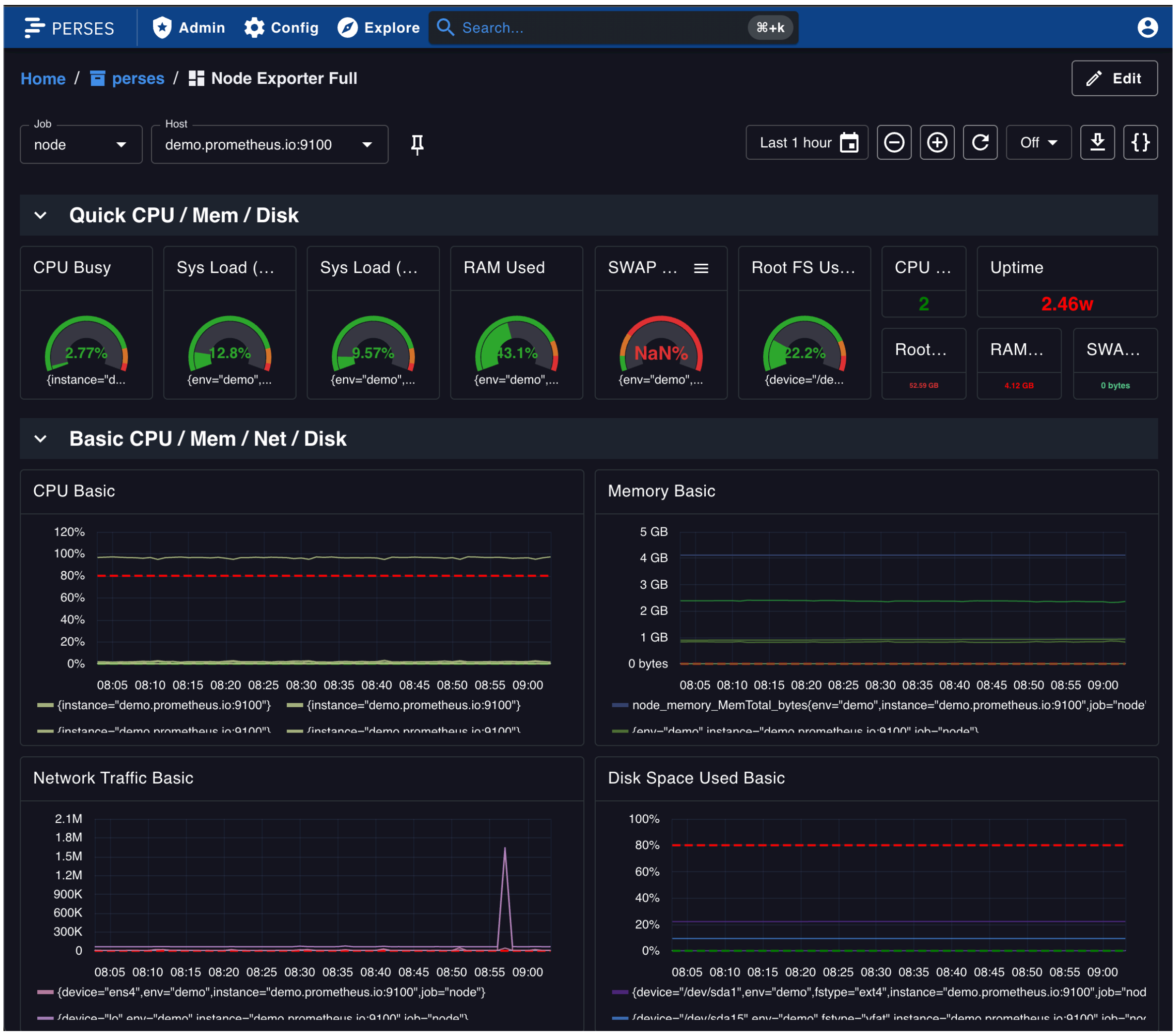
Prometheus

Show nav

Perses support for Prometheus

Perses is an open-source dashboard and visualization platform designed for observability, with native support for Prometheus as a data source. It enables users to create, manage, and share dashboards for monitoring metrics and visualizing data. Perses aims to provide a simple, flexible, and extensible alternative to other dashboarding tools, focusing on ease of use, community-driven development, GitOps capabilities and dashboard as code approach.

Here is an example of a Perses dashboard querying Prometheus for data:



Installing

To install Perses, see the official Perses documentation.

`http://localhost:8080` . There is no login by default.

Creating a Prometheus data source

To learn about how to set up a data source in Perses, please refer to Perses documentation. Once this connection to your Prometheus instance is configured, you are able to query it from the Dashboard and Explore views.

Importing pre-built dashboards

Perses is providing a set of pre-built dashboards that you can import into your instance. These dashboards are maintained by the community and can be found in the Perses dashboard repository

← **Previous** Grafana

Edit

Next Console templates →

Prometheus

Show nav

Console templates

Console templates allow for creation of arbitrary consoles using the Go templating language. These are served from the Prometheus server.

Console templates are the most powerful way to create templates that can be easily managed in source control. There is a learning curve though, so users new to this style of monitoring should try out Grafana first.

Getting started

Prometheus comes with an example set of consoles to get you going. These can be found at `/consoles/index.html.example` on a running Prometheus and will display Node Exporter consoles if Prometheus is scraping Node Exporters with a `job="node"` label.

The example consoles have 5 parts:

1. A navigation bar on top
2. A menu on the left
3. Time controls on the bottom
4. The main content in the center, usually graphs
5. A table on the right

The navigation bar is for links to other systems, such as other Prometheis [1], documentation, and whatever else makes sense to you. The menu is for navigation inside the same Prometheus server, which is very useful to be able to quickly open a console in another tab to correlate information. Both are configured in `console_libraries/menu.lib`.

The time controls allow changing of the duration and range of the graphs. Console URLs can be shared and will show the same graphs for others.

The main content is usually graphs. There is a configurable JavaScript graphing library provided that will handle requesting data from Prometheus, and rendering it via Rickshaw.

Finally, the table on the right can be used to display statistics in a more compact form than graphs.

CPU usage, and the average memory usage in the right-hand-side table. The main content has a queries-per-second graph.

```
{{template "head" .}}

{{template "prom_right_table_head"}}
<tr>
  <th>MyJob</th>
  <th>{{ template "prom_query_drilldown" (args "sum(up{job='myjob'})") }}
      / {{ template "prom_query_drilldown" (args "count(up{job='myjob'})") }}
  </th>
</tr>
<tr>
  <td>CPU</td>
  <td>{{ template "prom_query_drilldown" (args
      "avg by(job)(rate(process_cpu_seconds_total{job='myjob'}[5m]))"
      "s/s" "humanizeNoSmallPrefix") }}
  </td>
</tr>
<tr>
  <td>Memory</td>
  <td>{{ template "prom_query_drilldown" (args
       "avg by(job)(process_resident_memory_bytes{job='myjob'})"
       "B" "humanize1024") }}
  </td>
</tr>
{{template "prom_right_table_tail"}}


{{template "prom_content_head" .}}
<h1>MyJob</h1>

<h3>Queries</h3>
<div id="queryGraph"></div>
<script>
new PromConsole.Graph({
  node: document.querySelector("#queryGraph"),
  expr: "sum(rate(http_query_count{job='myjob'}[5m]))",
  name: "Queries",
  yAxisFormatter: PromConsole.NumberFormatter.humanizeNoSmallPrefix,
})
</script>

{{template "prom_content_tail" .}}

{{template "tail"}}
```

The `prom_right_table_head` and `prom_right_table_tail` templates contain the right-hand-side table. This is optional.

`prom_query_drilldown` is a template that will evaluate the expression passed to it, format it, and link to the expression in the expression browser. The first argument is the expression. The second argument is the unit to use. The third argument is how to format the output. Only the first argument is required.

Valid output formats for the third argument to `prom_query_drilldown` :

- Not specified: Default Go display output.
- `humanize` : Display the result using metric prefixes.
- `humanizeNoSmallPrefix` : For absolute values greater than 1, display the result using metric prefixes. For absolute values less than 1, display 3 significant digits. This is useful to avoid units such as milliqueries per second that can be produced by `humanize` .
- `humanize1024` : Display the humanized result using a base of 1024 rather than 1000. This is usually used with `B` as the second argument to produce units such as `KiB` and `MiB` .
- `printf.3g` : Display 3 significant digits.

Custom formats can be defined. See prom.lib for examples.

Graph Library

The graph library is invoked as:

```
<div id="queryGraph"></div>
<script>
new PromConsole.Graph({
  node: document.querySelector("#queryGraph"),
  expr: "sum(rate(http_query_count{job='myjob'}[5m]))"
```

Prometheus

The `head` template loads the required Javascript and CSS.

Parameters to the graph library:

Name	Description
expr	Required. Expression to graph. Can be a list.
node	Required. DOM node to render into.
duration	Optional. Duration of the graph. Defaults to 1 hour.
endTime	Optional. Unixtime the graph ends at. Defaults to now.
width	Optional. Width of the graph, excluding titles. Defaults to auto-detection.
height	Optional. Height of the graph, excluding titles and legends. Defaults to 200 pixels.
min	Optional. Minimum x-axis value. Defaults to lowest data value.
max	Optional. Maximum y-axis value. Defaults to highest data value.
renderer	Optional. Type of graph. Options are `line` and `area` (stacked graph). Defaults to `line`.
name	Optional. Title of plots in legend and hover detail. If passed a string, `[[ label ]]` will be substituted with the label value. If passed a function, it will be passed a map of labels and should return the name as a string. Can be a list.
xTitle	Optional. Title of the x-axis. Defaults to `Time`.
yUnits	Optional. Units of the y-axis. Defaults to empty.
yTitle	Optional. Title of the y-axis. Defaults to empty.
yAxisFormatter	Optional. Number formatter for the y-axis. Defaults to `PromConsole.NumberFormatter.humanize`.
yHoverFormatter	Optional. Number formatter for the hover detail. Defaults to `PromConsole.NumberFormatter.humanizeExact`.
colorScheme	Optional. Color scheme to be used by the plots. Can be either a list of hex color codes or one of the color scheme names supported by Rickshaw. Defaults to `'colorwheel'`.

If both `expr` and `name` are lists, they must be of the same length. The name will be applied to the plots for the corresponding expression.

Valid options for the `yAxisFormatter` and `yHoverFormatter`:

with 3 significant digits. This is useful to avoid units such as milliqueries per second that can be produced by `PromConsole.NumberFormatter.humanize`.

- `PromConsole.NumberFormatter.humanize1024`: Format the humanized result using a base of 1024 rather than 1000.

Previous
Perses

Edit

Next
Client libraries

Prometheus

Show nav

Client libraries

Before you can monitor your services, you need to add instrumentation to their code via one of the Prometheus client libraries. These implement the Prometheus metric types.

Choose a Prometheus client library that matches the language in which your application is written. This lets you define and expose internal metrics via an HTTP endpoint on your application's instance:

- Go
- Java or Scala
- Python
- Ruby
- Rust

Unofficial third-party client libraries:

- Bash
- C
- C++
- Common Lisp
- Dart
- Delphi
- Elixir
- Erlang
- Haskell
- Julia
- Lua for Nginx
- Lua for Tarantool
- .NET / C#
- Node.js
- OCaml
- Perl
- PHP
- R

If no client library is available for your language, or you want to avoid dependencies, you may also implement one of the supported exposition formats yourself to expose metrics.

When implementing a new Prometheus client library, please follow the guidelines on writing client libraries. Note that this document is still a work in progress. Please also consider consulting the development mailing list. We are happy to give advice on how to make your library as useful and consistent as possible.

← **Previous** Console templates

Edit

Next Writing client libraries →

Prometheus

Show nav

Writing client libraries

This document covers what functionality and API Prometheus client libraries should offer, with the aim of consistency across libraries, making the easy use cases easy and avoiding offering functionality that may lead users down the wrong path.

There are 10 languages already supported at the time of writing, so we've gotten a good sense by now of how to write a client. These guidelines aim to help authors of new client libraries produce good libraries.

Conventions

MUST/MUST NOT/SHOULD/SHOULD NOT/MAY have the meanings given in https://www.ietf.org/rfc/rfc2119.txt

In addition ENCOURAGED means that a feature is desirable for a library to have, but it's okay if it's not present. In other words, a nice to have.

Things to keep in mind:

- Take advantage of each language's features.
- The common use cases should be easy.
- The correct way to do something should be the easy way.
- More complex use cases should be possible.

The common use cases are (in order):

- Counters without labels spread liberally around libraries/applications.
- Timing functions/blocks of code in Summaries/Histograms.
- Gauges to track current states of things (and their limits).
- Monitoring of batch jobs.

structure described here.

The key class is the Collector. This has a method (typically called 'collect') that returns zero or more metrics and their samples. Collectors get registered with a CollectorRegistry. Data is exposed by passing a CollectorRegistry to a class/method/function "bridge", which returns the metrics in a format Prometheus supports. Every time the CollectorRegistry is scraped it must callback to each of the Collectors' collect method.

The interface most users interact with are the Counter, Gauge, Summary, and Histogram Collectors. These represent a single metric, and should cover the vast majority of use cases where a user is instrumenting their own code.

More advanced uses cases (such as proxying from another monitoring/instrumentation system) require writing a custom Collector. Someone may also want to write a "bridge" that takes a CollectorRegistry and produces data in a format a different monitoring/instrumentation system understands, allowing users to only have to think about one instrumentation system.

CollectorRegistry SHOULD offer `register()` / `unregister()` functions, and a Collector SHOULD be allowed to be registered to multiple CollectorRegistrys.

Client libraries MUST be thread safe.

For non-OO languages such as C, client libraries should follow the spirit of this structure as much as is practical.

Naming

Client libraries SHOULD follow function/method/class names mentioned in this document, keeping in mind the naming conventions of the language they're working in. For example, `set_to_current_time()` is good for a method name in Python, but `SetToCurrentTime()` is better in Go and `setToCurrentTime()` is the convention in Java. Where names differ for technical reasons (e.g. not allowing function overloading), documentation/help strings SHOULD point users towards the other names.

Libraries MUST NOT offer functions/methods/classes with the same or similar names to ones given here, but with different semantics.

Metrics

Counter and Gauge MUST be part of the client library. At least one of Summary and Histogram MUST be offered.

These should be primarily used as file-static variables, that is, global variables defined in the same file as the code they're instrumenting. The client library SHOULD enable this. The common use case is instrumenting a piece of code overall, not a piece of code in the context of one instance of an object. Users shouldn't have to worry about plumbing their metrics throughout their code, the client library should do that for them (and if it doesn't, users will write a wrapper around the library to make it "easier" - which rarely tends to go well).

There MUST be a default CollectorRegistry, the standard metrics MUST by default implicitly register into it with no special work required by the user. There MUST be a way to have metrics not register to the default CollectorRegistry, for use in batch jobs and unittests. Custom collectors SHOULD also follow this.

Exactly how the metrics should be created varies by language. For some (Java, Go) a builder approach is best, whereas for others (Python) function arguments are rich enough to do it in one call.

For example in the Java Simpleclient we have:

```
class YourClass {
  static final Counter requests = Counter.build()
      .name("requests_total")
      .help("Requests.").register();
}
```

This will register requests with the default CollectorRegistry. By calling `build()` rather than `register()` the metric won't be registered (handy for unittests), you can also pass in a CollectorRegistry to `register()` (handy for batch jobs).

Counter

Counter is a monotonically increasing counter. It MUST NOT allow the value to decrease, however it MAY be reset to 0 (such as by server restart).

A counter MUST have the following methods:

- `inc()` : Increment the counter by 1
- `inc(double v)` : Increment the counter by the given amount. MUST check that v >= 0.

types of exceptions. This is count_exceptions in Python.

Counters MUST start at 0.

Gauge

Gauge represents a value that can go up and down.

A gauge MUST have the following methods:

- `inc()` : Increment the gauge by 1
- `inc(double v)` : Increment the gauge by the given amount
- `dec()` : Decrement the gauge by 1
- `dec(double v)` : Decrement the gauge by the given amount
- `set(double v)` : Set the gauge to the given value

Gauges MUST start at 0, you MAY offer a way for a given gauge to start at a different number.

A gauge SHOULD have the following methods:

- `set_to_current_time()` : Set the gauge to the current unixtime in seconds.

A gauge is ENCOURAGED to have:

A way to track in-progress requests in some piece of code/function. This is `track_inprogress` in Python.

A way to time a piece of code and set the gauge to its duration in seconds. This is useful for batch jobs. This is startTimer/setDuration in Java and the `time()` decorator/context manager in Python. This SHOULD match the pattern in Summary/Histogram (though `set()` rather than `observe()`).

Summary

A summary samples observations (usually things like request durations) over sliding windows of time and provides instantaneous insight into their distributions, frequencies, and sums.

A summary MUST NOT allow the user to set "quantile" as a label name, as this is used internally to designate summary quantiles. A summary is ENCOURAGED to offer quantiles as

A summary MUST have the following methods:

- `observe(double v)` : Observe the given amount

A summary SHOULD have the following methods:

Some way to time code for users in seconds. In Python this is the `time()` decorator/context manager. In Java this is startTimer/observeDuration. Units other than seconds MUST NOT be offered (if a user wants something else, they can do it by hand). This should follow the same pattern as Gauge/Histogram.

Summary `_count` / `_sum` MUST start at 0.

Histogram

Histograms allow aggregatable distributions of events, such as request latencies. This is at its core a counter per bucket.

A histogram MUST NOT allow `le` as a user-set label, as `le` is used internally to designate buckets.

A histogram MUST offer a way to manually choose the buckets. Ways to set buckets in a `linear(start, width, count)` and `exponential(start, factor, count)` fashion SHOULD be offered. Count MUST include the `+Inf` bucket.

A histogram SHOULD have the same default buckets as other client libraries. Buckets MUST NOT be changeable once the metric is created.

A histogram MUST have the following methods:

- `observe(double v)` : Observe the given amount

A histogram SHOULD have the following methods:

Some way to time code for users in seconds. In Python this is the `time()` decorator/context manager. In Java this is `startTimer` / `observeDuration` . Units other than seconds MUST NOT be offered (if a user wants something else, they can do it by hand). This should follow the same pattern as Gauge/Summary.

Histogram `_count` / `_sum` and the buckets MUST start at 0.

Further metrics considerations

If there's a common use case you can make simpler then go for it, as long as it won't encourage undesirable behaviours (such as suboptimal metric/label layouts, or doing computation in the client).

Labels

Labels are one of the most powerful aspects of Prometheus, but easily abused. Accordingly client libraries must be very careful in how labels are offered to users.

Client libraries MUST NOT allow users to have different label names for the same metric for Gauge/Counter/Summary/Histogram or any other Collector offered by the library.

Metrics from custom collectors should almost always have consistent label names. As there are still rare but valid use cases where this is not the case, client libraries should not verify this.

While labels are powerful, the majority of metrics will not have labels. Accordingly the API should allow for labels but not dominate it.

A client library MUST allow for optionally specifying a list of label names at Gauge/Counter/Summary/Histogram creation time. A client library SHOULD support any number of label names. A client library MUST validate that label names meet the documented requirements.

The general way to provide access to labeled dimension of a metric is via a `labels()` method that takes either a list of the label values or a map from label name to label value and returns a "Child". The usual `.inc()` / `.dec()` / `.observe()` etc. methods can then be called on the Child.

The Child returned by `labels()` SHOULD be cacheable by the user, to avoid having to look it up again - this matters in latency-critical code.

Metrics with labels SHOULD support a `remove()` method with the same signature as `labels()` that will remove a Child from the metric no longer exporting it, and a `clear()` method that removes all Children from the metric. These invalidate caching of Children.

There SHOULD be a way to initialize a given Child with the default value, usually just calling `labels()`. Metrics without labels MUST always be initialized to avoid problems with missing metrics.

of Gauge/Counter/Summary/Histogram and in any other Collector offered with the library.

Many client libraries offer setting the name in three parts: `namespace_subsystem_name` of which only the `name` is mandatory.

Dynamic/generated metric names or subparts of metric names MUST be discouraged, except when a custom Collector is proxying from other instrumentation/monitoring systems. Generated/dynamic metric names are a sign that you should be using labels instead.

Metric description and help

Gauge/Counter/Summary/Histogram MUST require metric descriptions/help to be provided.

Any custom Collectors provided with the client libraries MUST have descriptions/help on their metrics.

It is suggested to make it a mandatory argument, but not to check that it’s of a certain length as if someone really doesn’t want to write docs we’re not going to convince them otherwise. Collectors offered with the library (and indeed everywhere we can within the ecosystem) SHOULD have good metric descriptions, to lead by example.

Exposition

Clients MUST implement the text-based exposition format outlined in the exposition formats documentation.

Reproducible order of the exposed metrics is ENCOURAGED (especially for human readable formats) if it can be implemented without a significant resource cost.

Standard and runtime collectors

Client libraries SHOULD offer what they can of the Standard exports, documented below.

These SHOULD be implemented as custom Collectors, and registered by default on the default CollectorRegistry. There SHOULD be a way to disable these, as there are some very niche use cases where they get in the way.

Process metrics

All memory values in bytes, all times in unixtime/seconds.

Metric name	Help string	Unit
`process_cpu_seconds_total`	Total user and system CPU time spent in seconds.	seconds
`process_open_fds`	Number of open file descriptors.	file descriptors
`process_max_fds`	Maximum number of open file descriptors.	file descriptors
`process_virtual_memory_bytes`	Virtual memory size in bytes.	bytes
`process_virtual_memory_max_bytes`	Maximum amount of virtual memory available in bytes.	bytes
`process_resident_memory_bytes`	Resident memory size in bytes.	bytes
`process_heap_bytes`	Process heap size in bytes.	bytes
`process_start_time_seconds`	Start time of the process since unix epoch in seconds.	seconds
`process_threads`	Number of OS threads in the process.	threads

Runtime metrics

In addition, client libraries are ENCOURAGED to also offer whatever makes sense in terms of metrics for their language's runtime (e.g. garbage collection stats), with an appropriate prefix such as `go_`, `hotspot_` etc.

Unit tests

Client libraries SHOULD have unit tests covering the core instrumentation library and exposition.

Client libraries are ENCOURAGED to offer ways that make it easy for users to unit-test their use of the instrumentation code. For example, the `CollectorRegistry.get_sample_value` in Python.

without breaking the application.

Accordingly, caution is advised when adding dependencies to the client library. For example, if you add a library that uses a Prometheus client that requires version x.y of a library but the application uses x.z elsewhere, will that have an adverse impact on the application?

It is suggested that where this may arise, that the core instrumentation is separated from the bridges/exposition of metrics in a given format. For example, the Java simpleclient `simpleclient` module has no dependencies, and the `simpleclient_servlet` has the HTTP bits.

Performance considerations

As client libraries must be thread-safe, some form of concurrency control is required and consideration must be given to performance on multi-core machines and applications.

In our experience the least performant is mutexes.

Processor atomic instructions tend to be in the middle, and generally acceptable.

Approaches that avoid different CPUs mutating the same bit of RAM work best, such as the DoubleAdder in Java's simpleclient. There is a memory cost though.

As noted above, the result of `labels()` should be cacheable. The concurrent maps that tend to back metric with labels tend to be relatively slow. Special-casing metrics without labels to avoid `labels()` -like lookups can help a lot.

Metrics SHOULD avoid blocking when they are being incremented/decremented/set etc. as it's undesirable for the whole application to be held up while a scrape is ongoing.

Having benchmarks of the main instrumentation operations, including labels, is ENCOURAGED.

Resource consumption, particularly RAM, should be kept in mind when performing exposition. Consider reducing the memory footprint by streaming results, and potentially having a limit on the number of concurrent scrapes.

Prometheus

Prometheus

Show nav

Pushing metrics

Occasionally you will need to monitor components which cannot be scraped. The Prometheus Pushgateway allows you to push time series from short-lived service-level batch jobs to an intermediary job which Prometheus can scrape. Combined with Prometheus's simple text-based exposition format, this makes it easy to instrument even shell scripts without a client library.

- For more information on using the Pushgateway and use from a Unix shell, see the project's README.md.
- For use from Java see the Pushgateway documentation.
- For use from Go see the Push and Add methods.
- For use from Python see Exporting to a Pushgateway.
- For use from Ruby see the Pushgateway documentation.
- To find out about Pushgateway support of client libraries maintained outside of the Prometheus project, refer to their respective documentation.

← Previous
Writing client libraries

Edit

Next
Exporters and integrations →

Prometheus

Prometheus

Show nav

Exporters and integrations

There are a number of libraries and servers which help in exporting existing metrics from third-party systems as Prometheus metrics. This is useful for cases where it is not feasible to instrument a given system with Prometheus metrics directly (for example, HAProxy or Linux system stats).

Third-party exporters

Some of these exporters are maintained as part of the official Prometheus GitHub organization, those are marked as *official*, others are externally contributed and maintained.

We encourage the creation of more exporters but cannot vet all of them for best practices. Commonly, those exporters are hosted outside of the Prometheus GitHub organization.

The exporter default port wiki page has become another catalog of exporters, and may include exporters not listed here due to overlapping functionality or still being in development.

The JMX exporter can export from a wide variety of JVM-based applications, for example Kafka and Cassandra.

Databases

- Aerospike exporter
- AWS RDS exporter
- ClickHouse exporter
- Consul exporter (**official**)
- Couchbase exporter
- CouchDB exporter
- Druid Exporter
- Elasticsearch exporter
- EventStore exporter
- IoTDB exporter

- MongoDB query exporter
- MongoDB Node.js Driver exporter
- MSSQL server exporter
- MySQL router exporter
- MySQL server exporter (**official**)
- OpenTSDB Exporter
- Oracle DB Exporter
- PgBouncer exporter
- PostgreSQL exporter
- Presto exporter
- ProxySQL exporter
- RavenDB exporter
- Redis exporter
- RethinkDB exporter
- SQL exporter
- Tarantool metric library
- Twemproxy

Hardware related

- apcupsd exporter
- BIG-IP exporter
- Bosch Sensortec BMP/BME exporter
- Collins exporter
- Dell Hardware OMSA exporter
- Disk usage exporter
- Fortigate exporter
- IBM Z HMC exporter
- IoT Edison exporter
- InfiniBand exporter
- IPMI exporter
- knxd exporter
- Modbus exporter
- Netgear Cable Modem Exporter
- Netgear Router exporter
- Network UPS Tools (NUT) exporter
- Node/system metrics exporter (**official**)
- NVIDIA GPU exporter

Prometheus

- Weathergoose Climate Monitor Exporter
- Windows exporter
- Intel® Optane™ Persistent Memory Controller Exporter

Issue trackers and continuous integration

- Bamboo exporter
- Bitbucket exporter
- Confluence exporter
- Jenkins exporter
- JIRA exporter

Messaging systems

- Beanstalkd exporter
- EMQ exporter
- Gearman exporter
- IBM MQ exporter
- Kafka exporter
- NATS exporter
- NSQ exporter
- Mirth Connect exporter
- MQTT blackbox exporter
- MQTT2Prometheus
- RabbitMQ exporter
- RabbitMQ Management Plugin exporter
- RocketMQ exporter
- Solace exporter

Storage

- Ceph exporter
- Ceph RADOSGW exporter
- Gluster exporter
- GPFS exporter
- Hadoop HDFS FSImage exporter
- HPE CSI info metrics provider

- Pure Storage exporter
- ScaleIO exporter
- Tivoli Storage Manager/IBM Spectrum Protect exporter

HTTP

- Apache exporter
- HAProxy exporter (**official**)
- Nginx metric library
- Nginx VTS exporter
- Passenger exporter
- Squid exporter
- Tinyproxy exporter
- Varnish exporter
- WebDriver exporter

APIs

- AWS ECS exporter
- AWS Health exporter
- AWS SQS exporter
- AWS SQS Prometheus exporter
- Azure Health exporter
- BigBlueButton
- Cloudflare exporter
- Cryptowat exporter
- DigitalOcean exporter
- Docker Cloud exporter
- Docker Hub exporter
- Fastly exporter
- GitHub exporter
- Gmail exporter
- GraphQL exporter
- InstaClustr exporter
- Mozilla Observatory exporter
- OpenWeatherMap exporter
- Pagespeed exporter
- Rancher exporter

Prometheus

Logging

- Fluentd exporter
- Google's mtail log data extractor
- Grok exporter

FinOps

- AWS Cost Exporter
- Azure Cost Exporter
- Kubernetes Cost Exporter

Other monitoring systems

- Akamai Cloudmonitor exporter
- Alibaba Cloudmonitor exporter
- AWS CloudWatch exporter (**official**)
- Azure Monitor exporter
- Cloud Foundry Firehose exporter
- Collectd exporter (**official**)
- Google Stackdriver exporter
- Graphite exporter (**official**)
- Heka dashboard exporter
- Heka exporter
- Huawei Cloudeye exporter
- InfluxDB exporter (**official**)
- ITM exporter
- Java GC exporter
- JavaMelody exporter
- JMX exporter (**official**)
- Munin exporter
- Nagios / Naemon exporter
- Neptune Apex exporter
- New Relic exporter
- NRPE exporter
- Osquery exporter
- OTC CloudEye exporter

- Sensu exporter
- site24x7_exporter
- SNMP exporter (**official**)
- StatsD exporter (**official**)
- TencentCloud monitor exporter
- ThousandEyes exporter
- StatusPage exporter

Miscellaneous

- ACT Fibernet Exporter
- BIND exporter
- BIND query exporter
- Bitcoind exporter
- Blackbox exporter (**official**)
- Bungeecord exporter
- BOSH exporter
- cAdvisor
- Cachet exporter
- ccache exporter
- c-lightning exporter
- DHCPD leases exporter
- Dovecot exporter
- Dnsmasq exporter
- eBPF exporter
- eBPF network traffic exporter
- Ethereum Client exporter
- FFmpeg exporter
- File statistics exporter
- JFrog Artifactory Exporter
- Hostapd Exporter
- IBM Security Verify Access / Security Access Manager Exporter
- IPsec exporter
- ipset exporter
- IRCd exporter
- Linux HA ClusterLabs exporter
- JMeter plugin
- JSON exporter

- kube-state-metrics
- Locust Exporter
- Meteor JS web framework exporter
- Minecraft exporter module
- Minecraft exporter
- NetBird exporter
- Nomad exporter
- nftables exporter
- OpenStack exporter
- OpenStack blackbox exporter
- OpenVPN exporter
- oVirt exporter
- Pact Broker exporter
- PHP-FPM exporter
- PowerDNS exporter
- Podman exporter
- Prefect2 exporter
- Process exporter
- rTorrent exporter
- Rundeck exporter
- SABnzbd exporter
- SAML exporter
- Script exporter
- Shield exporter
- Smokeping prober
- SMTP/Maildir MDA blackbox prober
- SoftEther exporter
- SSH exporter
- Teamspeak3 exporter
- Transmission exporter
- Unbound exporter
- WireGuard exporter
- Xen exporter

When implementing a new Prometheus exporter, please follow the guidelines on writing exporters Please also consider consulting the development mailing list. We are happy to give advice on how to make your exporter as useful and consistent as possible.

exporters are needed:

- Ansible Automation Platform Automation Controller (AWX)
- App Connect Enterprise
- Ballerina
- BFE
- Caddy (**direct**)
- Ceph
- CockroachDB
- Collectd
- Concourse
- CRG Roller Derby Scoreboard (**direct**)
- Diffusion
- Docker Daemon
- Doorman (**direct**)
- Dovecot
- Envoy
- Etcd (**direct**)
- Flink
- FreeBSD Kernel
- GitLab
- Grafana
- JavaMelody
- Kong
- Kubernetes (**direct**)
- LavinMQ
- Linkerd
- mgmt
- MidoNet
- midonet-kubernetes (**direct**)
- MinIO
- PATROL with Monitoring Studio X
- Netdata
- OpenZiti
- Pomerium
- Pretix
- Quobyte (**direct**)
- RabbitMQ
- RobustIRC

- Telegraf
- Traefik
- Vector
- VerneMQ
- Flux
- Xandikos (**direct**)
- Zipkin

The software marked *direct* is also directly instrumented with a Prometheus client library.

Other third-party utilities

This section lists libraries and other utilities that help you instrument code in a certain language. They are not Prometheus client libraries themselves but make use of one of the normal Prometheus client libraries under the hood. As for all independently maintained software, we cannot vet all of them for best practices.

- Clojure: iapetos
- Go: go-metrics instrumentation library
- Go: gokit
- Go: prombolt
- Java/JVM: EclipseLink metrics collector
- Java/JVM: Hystrix metrics publisher
- Java/JVM: Jersey metrics collector
- Java/JVM: Micrometer Prometheus Registry
- Python-Django: django-prometheus
- Node.js: swagger-stats

← **Previous** Pushing metrics

Edit

Next Writing exporters →

Prometheus

Prometheus

Show nav

Writing exporters

If you are instrumenting your own code, the general rules of how to instrument code with a Prometheus client library should be followed. When taking metrics from another monitoring or instrumentation system, things tend not to be so black and white.

This document contains things you should consider when writing an exporter or custom collector. The theory covered will also be of interest to those doing direct instrumentation.

If you are writing an exporter and are unclear on anything here, please contact us on IRC (#prometheus on libera) or the mailing list.

Maintainability and purity

The main decision you need to make when writing an exporter is how much work you're willing to put in to get perfect metrics out of it.

If the system in question has only a handful of metrics that rarely change, then getting everything perfect is an easy choice, a good example of this is the HAProxy exporter.

On the other hand, if you try to get things perfect when the system has hundreds of metrics that change frequently with new versions, then you've signed yourself up for a lot of ongoing work. The MySQL exporter is on this end of the spectrum.

The node exporter is a mix of these, with complexity varying by module. For example, the `mdadm` collector hand-parses a file and exposes metrics created specifically for that collector, so we may as well get the metrics right. For the `meminfo` collector the results vary across kernel versions so we end up doing just enough of a transform to create valid metrics.

Configuration

When working with applications, you should aim for an exporter that requires no custom configuration by the user beyond telling it where the application is. You may also need to offer the ability to filter out certain metrics if they may be too granular and expensive on

When working with other monitoring systems, frameworks and protocols you will often need to provide additional configuration or customization to generate metrics suitable for Prometheus. In the best case scenario, a monitoring system has a similar enough data model to Prometheus that you can automatically determine how to transform metrics. This is the case for Cloudwatch, SNMP and collectd. At most, we need the ability to let the user select which metrics they want to pull out.

In other cases, metrics from the system are completely non-standard, depending on the usage of the system and the underlying application. In that case the user has to tell us how to transform the metrics. The JMX exporter is the worst offender here, with the Graphite and StatsD exporters also requiring configuration to extract labels.

Ensuring the exporter works out of the box without configuration, and providing a selection of example configurations for transformation if required, is advised.

YAML is the standard Prometheus configuration format, all configuration should use YAML by default.

Metrics

Naming

Follow the best practices on metric naming.

Generally metric names should allow someone who is familiar with Prometheus but not a particular system to make a good guess as to what a metric means. A metric named `http_requests_total` is not extremely useful - are these being measured as they come in, in some filter or when they get to the user's code? And `requests_total` is even worse, what type of requests?

With direct instrumentation, a given metric should exist within exactly one file. Accordingly, within exporters and collectors, a metric should apply to exactly one subsystem and be named accordingly.

Metric names should never be procedurally generated, except when writing a custom collector or exporter.

Metric names for applications should generally be prefixed by the exporter name, e.g. `haproxy_up`.

better, specify a counter for each of the two components of the ratio.

Metric names should not include the labels that they're exported with, e.g. `by_type`, as that won't make sense if the label is aggregated away.

The one exception is when you're exporting the same data with different labels via multiple metrics, in which case that's usually the sanest way to distinguish them. For direct instrumentation, this should only come up when exporting a single metric with all the labels would have too high a cardinality.

Prometheus metrics and label names are written in `snake_case`. Converting `camelCase` to `snake_case` is desirable, though doing so automatically doesn't always produce nice results for things like `myTCPExample` or `isNaN` so sometimes it's best to leave them as-is.

Exposed metrics should not contain colons, these are reserved for user defined recording rules to use when aggregating.

Only `[a-zA-Z0-9:_]` are valid in metric names.

The `_sum`, `_count`, `_bucket` and `_total` suffixes are used by Summaries, Histograms and Counters. Unless you're producing one of those, avoid these suffixes.

`_total` is a convention for counters, you should use it if you're using the COUNTER type.

The `process_` and `scrape_` prefixes are reserved. It's okay to add your own prefix on to these if they follow matching semantics. For example, Prometheus has `scrape_duration_seconds` for how long a scrape took, it's good practice to also have an exporter-centric metric, e.g. `jmx_scrape_duration_seconds`, saying how long the specific exporter took to do its thing. For process stats where you have access to the PID, both Go and Python offer collectors that'll handle this for you. A good example of this is the HAProxy exporter.

When you have a successful request count and a failed request count, the best way to expose this is as one metric for total requests and another metric for failed requests. This makes it easy to calculate the failure ratio. Do not use one metric with a failed or success label. Similarly, with hit or miss for caches, it's better to have one metric for total and another for hits.

Consider the likelihood that someone using monitoring will do a code or web search for the metric name. If the names are very well-established and unlikely to be used outside of the realm of people used to those names, for example SNMP and network engineers, then leaving them as-is may be a good idea. This logic doesn't apply for all exporters, for example

Prometheus

Labels

Read the general advice on labels.

Avoid `type` as a label name, it's too generic and often meaningless. You should also try where possible to avoid names that are likely to clash with target labels, such as `region`, `zone`, `cluster`, `availability_zone`, `az`, `datacenter`, `dc`, `owner`, `customer`, `stage`, `service`, `environment` and `env`. If, however, that's what the application calls some resource, it's best not to cause confusion by renaming it.

Avoid the temptation to put things into one metric just because they share a prefix. Unless you're sure something makes sense as one metric, multiple metrics is safer.

The label `le` has special meaning for Histograms, and `quantile` for Summaries. Avoid these labels generally.

Read/write and send/receive are best as separate metrics, rather than as a label. This is usually because you care about only one of them at a time, and it is easier to use them that way.

The rule of thumb is that one metric should make sense when summed or averaged. There is one other case that comes up with exporters, and that's where the data is fundamentally tabular and doing otherwise would require users to do regexes on metric names to be usable. Consider the voltage sensors on your motherboard, while doing math across them is meaningless, it makes sense to have them in one metric rather than having one metric per sensor. All values within a metric should (almost) always have the same unit, for example consider if fan speeds were mixed in with the voltages, and you had no way to automatically separate them.

Don't do this:

```
my_metric{label="a"} 1
my_metric{label="b"} 6
my_metric{label="total"} 7
```

or this:

```
my_metric{label="a"} 1
my_metric{label="b"} 6
my_metric{} 7
```

doing the latter with direct instrumentation. Never do either of these, rely on Prometheus aggregation instead.

If your monitoring exposes a total like this, drop the total. If you have to keep it around for some reason, for example the total includes things not counted individually, use different metric names.

Instrumentation labels should be minimal, every extra label is one more that users need to consider when writing their PromQL. Accordingly, avoid having instrumentation labels which could be removed without affecting the uniqueness of the time series. Additional information around a metric can be added via an info metric, for an example see below how to handle version numbers.

However, there are cases where it is expected that virtually all users of a metric will want the additional information. If so, adding a non-unique label, rather than an info metric, is the right solution. For example the mysqld_exporter's `mysqld_perf_schema_events_statements_total`'s `digest` label is a hash of the full query pattern and is sufficient for uniqueness. However, it is of little use without the human readable `digest_text` label, which for long queries will contain only the start of the query pattern and is thus not unique. Thus we end up with both the `digest_text` label for humans and the `digest` label for uniqueness.

Target labels, not static scraped labels

If you ever find yourself wanting to apply the same label to all of your metrics, stop.

There’s generally two cases where this comes up.

The first is for some label it would be useful to have on the metrics such as the version number of the software. Instead, use the approach described at https://www.robustperception.io/how-to-have-labels-for-machine-roles/.

The second case is when a label is really a target label. These are things like region, cluster names, and so on, that come from your infrastructure setup rather than the application itself. It’s not for an application to say where it fits in your label taxonomy, that’s for the person running the Prometheus server to configure and different people monitoring the same application may give it different names.

Accordingly, these labels belong up in the scrape configs of Prometheus via whatever service discovery you’re using. It’s okay to apply the concept of machine roles here as well, as it’s likely useful information for at least some people scraping it.

means counters and gauges. The `_count` and `_sum` of summaries are also relatively common, and on occasion you'll see quantiles. Histograms are rare, if you come across one remember that the exposition format exposes cumulative values.

Often it won't be obvious what the type of metric is, especially if you're automatically processing a set of metrics. In general `UNTYPED` is a safe default.

Counters can't go down, so if you have a counter type coming from another instrumentation system that can be decremented, for example Dropwizard metrics then it's not a counter, it's a gauge. `UNTYPED` is probably the best type to use there, as `GAUGE` would be misleading if it were being used as a counter.

Help strings

When you're transforming metrics it's useful for users to be able to track back to what the original was, and what rules were in play that caused that transformation. Putting in the name of the collector or exporter, the ID of any rule that was applied and the name and details of the original metric into the help string will greatly aid users.

Prometheus doesn't like one metric having different help strings. If you're making one metric from many others, choose one of them to put in the help string.

For examples of this, the SNMP exporter uses the OID and the JMX exporter puts in a sample mBean name. The HAProxy exporter has hand-written strings. The node exporter also has a wide variety of examples.

Drop less useful statistics

Some instrumentation systems expose 1m, 5m, 15m rates, average rates since application start (these are called `mean` in Dropwizard metrics for example) in addition to minimums, maximums and standard deviations.

These should all be dropped, as they're not very useful and add clutter. Prometheus can calculate rates itself, and usually more accurately as the averages exposed are usually exponentially decaying. You don't know what time the min or max were calculated over, and the standard deviation is statistically useless and you can always expose sum of squares, `_sum` and `_count` if you ever need to calculate it.

Quantiles have related issues, you may choose to drop them or put them in a Summary.

`my.class.path.mymetric.labelvalue1.labelvalue2.labelvalue3`.

The Graphite and StatsD exporters share a way of transforming these with a small configuration language. Other exporters should implement the same. The transformation is currently implemented only in Go, and would benefit from being factored out into a separate library.

Collectors

When implementing the collector for your exporter, you should never use the usual direct instrumentation approach and then update the metrics on each scrape.

Rather create new metrics each time. In Go this is done with MustNewConstMetric in your `Collect()` method. For Python see https://github.com/prometheus/client_python#custom-collectors and for Java generate a `List<MetricFamilySamples>` in your collect method, see StandardExports.java for an example.

The reason for this is two-fold. Firstly, two scrapes could happen at the same time, and direct instrumentation uses what are effectively file-level global variables, so you'll get race conditions. Secondly, if a label value disappears, it'll still be exported.

Instrumenting your exporter itself via direct instrumentation is fine, e.g. total bytes transferred or calls performed by the exporter across all scrapes. For exporters such as the blackbox exporter and SNMP exporter, which aren't tied to a single target, these should only be exposed on a vanilla `/metrics` call, not on a scrape of a particular target.

Metrics about the scrape itself

Sometimes you'd like to export metrics that are about the scrape, like how long it took or how many records you processed.

These should be exposed as gauges as they're about an event, the scrape, and the metric name prefixed by the exporter name, for example `jmx_scrape_duration_seconds`. Usually the `_exporter` is excluded and if the exporter also makes sense to use as just a collector, then definitely exclude it.

Other scrape "meta" metrics should be avoided. For example, a counter for the number of scrapes, or a histogram of the scrape duration. Having the exporter track these metrics

Prometheus

Machine and process metrics

Many systems, for example Elasticsearch, expose machine metrics such as CPU, memory and filesystem information. As the node exporter provides these in the Prometheus ecosystem, such metrics should be dropped.

In the Java world, many instrumentation frameworks expose process-level and JVM-level stats such as CPU and GC. The Java client and JMX exporter already include these in the preferred form via DefaultExports.java, so these should also be dropped.

Similarly with other languages and frameworks.

Deployment

Each exporter should monitor exactly one instance application, preferably sitting right beside it on the same machine. That means for every HAProxy you run, you run a `haproxy_exporter` process. For every machine with a Mesos worker, you run the Mesos exporter on it, and another one for the master, if a machine has both.

The theory behind this is that for direct instrumentation this is what you'd be doing, and we're trying to get as close to that as we can in other layouts. This means that all service discovery is done in Prometheus, not in exporters. This also has the benefit that Prometheus has the target information it needs to allow users probe your service with the blackbox exporter.

There are two exceptions:

The first is where running beside the application you are monitoring is completely nonsensical. The SNMP, blackbox and IPMI exporters are the main examples of this. The IPMI and SNMP exporters as the devices are often black boxes that it's impossible to run code on (though if you could run a node exporter on them instead that'd be better), and the blackbox exporter where you're monitoring something like a DNS name, where there's also nothing to run on. In this case, Prometheus should still do service discovery, and pass on the target to be scraped. See the blackbox and SNMP exporters for examples.

Note that it is only currently possible to write this type of exporter with the Go, Python and Java client libraries.

The second exception is where you're pulling some stats out of a random instance of a system and don't care which one you're talking to. Consider a set of MySQL replicas you

This doesn’t apply when you’re monitoring a system with master-election, in that case you should monitor each instance individually and deal with the "masterness" in Prometheus. This is as there isn’t always exactly one master, and changing what a target is underneath Prometheus’s feet will cause oddities.

Scheduling

Metrics should only be pulled from the application when Prometheus scrapes them, exporters should not perform scrapes based on their own timers. That is, all scrapes should be synchronous.

Accordingly, you should not set timestamps on the metrics you expose, let Prometheus take care of that. If you think you need timestamps, then you probably need the Pushgateway instead.

If a metric is particularly expensive to retrieve, i.e. takes more than a minute, it is acceptable to cache it. This should be noted in the `HELP` string.

The default scrape timeout for Prometheus is 10 seconds. If your exporter can be expected to exceed this, you should explicitly call this out in your user documentation.

Pushes

Some applications and monitoring systems only push metrics, for example StatsD, Graphite and collectd.

There are two considerations here.

Firstly, when do you expire metrics? Collectd and things talking to Graphite both export regularly, and when they stop we want to stop exposing the metrics. Collectd includes an expiry time so we use that, Graphite doesn’t so it is a flag on the exporter.

StatsD is a bit different, as it is dealing with events rather than metrics. The best model is to run one exporter beside each application and restart them when the application restarts so that the state is cleared.

Secondly, these sort of systems tend to allow your users to send either deltas or raw counters. You should rely on the raw counters as far as possible, as that’s the general Prometheus model.

node exporter's textfile collector, rely on in-memory state (probably best if you don't need to persist over a reboot) or implement similar functionality to the textfile collector.

Failed scrapes

There are currently two patterns for failed scrapes where the application you're talking to doesn't respond or has other problems.

The first is to return a 5xx error.

The second is to have a `myexporter_up`, e.g. `haproxy_up`, variable that has a value of 0 or 1 depending on whether the scrape worked.

The latter is better where there's still some useful metrics you can get even with a failed scrape, such as the HAProxy exporter providing process stats. The former is a tad easier for users to deal with, as `up` works in the usual way, although you can't distinguish between the exporter being down and the application being down.

Landing page

It's nicer for users if visiting `http://yourexporter/` has a simple HTML page with the name of the exporter, and a link to the `/metrics` page.

Port numbers

A user may have many exporters and Prometheus components on the same machine, so to make that easier each has a unique port number.

https://github.com/prometheus/prometheus/wiki/Default-port-allocations is where we track them, this is publicly editable.

Feel free to grab the next free port number when developing your exporter, preferably before publicly announcing it. If you're not ready to release yet, putting your username and WIP is fine.

This is a registry to make our users' lives a little easier, not a commitment to develop particular exporters. For exporters for internal applications we recommend using ports outside of the range of default port allocations.

PR to add it to the list of available exporters by editing this GitHub repository file.

Prometheus

Show nav

Exposition formats

Metrics can be exposed to Prometheus using a simple text-based exposition format. There are various client libraries that implement this format for you. If your preferred language doesn't have a client library you can create your own.

Text-based format

As of Prometheus version 2.0, all processes that expose metrics to Prometheus need to use a text-based format. In this section you can find some basic information about this format as well as a more detailed breakdown of the format.

Basic info

Aspect	Description
Inception	April 2014
Supported in	Prometheus version `>=0.4.0`
Transmission	HTTP
Encoding	UTF-8, `\n` line endings
HTTP `Content-Type`	`text/plain; version=0.0.4` (A missing `version` value will lead to a fall-back to the most recent text format version.)
Optional HTTP `Content-Encoding`	`gzip`
Advantages	• Human-readable • Easy to assemble, especially for minimalistic cases (no nesting required) • Readable line by line (with the exception of type hints and docstrings)

Limitations	• Types and docstrings not integral part of the syntax, meaning little-to-nonexistent metric contract validation • Parsing cost
Supported metric primitives	• Counter • Gauge • Histogram • Summary • Untyped

Text format details

Prometheus' text-based format is line oriented. Lines are separated by a line feed character (`\n`). The last line must end with a line feed character. Empty lines are ignored.

Line format

Within a line, tokens can be separated by any number of blanks and/or tabs (and must be separated by at least one if they would otherwise merge with the previous token). Leading and trailing whitespace is ignored.

Comments, help text, and type information

Lines with a `#` as the first non-whitespace character are comments. They are ignored unless the first token after `#` is either `HELP` or `TYPE` . Those lines are treated as follows: If the token is `HELP` , at least one more token is expected, which is the metric name. All remaining tokens are considered the docstring for that metric name. `HELP` lines may contain any sequence of UTF-8 characters (after the metric name), but the backslash and the line feed characters have to be escaped as `\\` and `\n` , respectively. Only one `HELP` line may exist for any given metric name.

If the token is `TYPE` , exactly two more tokens are expected. The first is the metric name, and the second is either `counter` , `gauge` , `histogram` , `summary` , or `untyped` , defining the type for the metric of that name. Only one `TYPE` line may exist for a given metric name. The `TYPE` line for a metric name must appear before the first sample is reported for that metric name. If there is no `TYPE` line for a metric name, the type is set to `untyped` .

```
   "{" label_name "=" `"` label_value `"` { "," label_name "=" `"` label_value `"` }
 ] value [ timestamp ]
```

In the sample syntax:

- `metric_name` and `label_name` carry the usual Prometheus expression language restrictions.
- `label_value` can be any sequence of UTF-8 characters, but the backslash (\ ), double-quote ("), and line feed (\n) characters have to be escaped as `\\`, `\"`, and `\n`, respectively.
- `value` is a float represented as required by Go's `ParseFloat()` function. In addition to standard numerical values, `NaN`, `+Inf`, and `-Inf` are valid values representing not a number, positive infinity, and negative infinity, respectively.
- The `timestamp` is an `int64` (milliseconds since epoch, i.e. 1970-01-01 00:00:00 UTC, excluding leap seconds), represented as required by Go's `ParseInt()` function.

Grouping and sorting

All lines for a given metric must be provided as one single group, with the optional `HELP` and `TYPE` lines first (in no particular order). Beyond that, reproducible sorting in repeated expositions is preferred but not required, i.e. do not sort if the computational cost is prohibitive.

Each line must have a unique combination of a metric name and labels. Otherwise, the ingestion behavior is undefined.

Histograms and summaries

The `histogram` and `summary` types are difficult to represent in the text format. The following conventions apply:

- The sample sum for a summary or histogram named `x` is given as a separate sample named `x_sum`.
- The sample count for a summary or histogram named `x` is given as a separate sample named `x_count`.
- Each quantile of a summary named `x` is given as a separate sample line with the same name `x` and a label `{quantile="y"}`.

value of `x_count`.
- The buckets of a histogram and the quantiles of a summary must appear in increasing numerical order of their label values (for the `le` or the `quantile` label, respectively).

Text format example

Below is an example of a full-fledged Prometheus metric exposition, including comments, `HELP` and `TYPE` expressions, a histogram, a summary, character escaping examples, and more.

```
# HELP http_requests_total The total number of HTTP requests.
# TYPE http_requests_total counter
http_requests_total{method="post",code="200"} 1027 1395066363000
http_requests_total{method="post",code="400"}    3 1395066363000

# Escaping in label values:
msdos_file_access_time_seconds{path="C:\\DIR\\FILE.TXT",error="Cannot find file:\n\'

# Minimalistic line:
metric_without_timestamp_and_labels 12.47

# A weird metric from before the epoch:
something_weird{problem="division by zero"} +Inf -3982045

# A histogram, which has a pretty complex representation in the text format:
# HELP http_request_duration_seconds A histogram of the request duration.
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{le="0.05"} 24054
http_request_duration_seconds_bucket{le="0.1"} 33444
http_request_duration_seconds_bucket{le="0.2"} 100392
http_request_duration_seconds_bucket{le="0.5"} 129389
http_request_duration_seconds_bucket{le="1"} 133988
http_request_duration_seconds_bucket{le="+Inf"} 144320
http_request_duration_seconds_sum 53423
http_request_duration_seconds_count 144320

# Finally a summary, which has a complex representation, too:
# HELP rpc_duration_seconds A summary of the RPC duration in seconds.
# TYPE rpc_duration_seconds summary
```

```
rpc_duration_seconds{quantile="0.9"} 9001
rpc_duration_seconds{quantile="0.99"} 76656
rpc_duration_seconds_sum 1.7560473e+07
rpc_duration_seconds_count 2693
```

OpenMetrics Text Format

OpenMetrics is the an effort to standardize metric wire formatting built off of Prometheus text format. It is possible to scrape targets and it is also available to use for federating metrics since at least v2.23.0.

Exemplars (Experimental)

Utilizing the OpenMetrics format allows for the exposition and querying of Exemplars. Exemplars provide a point in time snapshot related to a metric set for an otherwise summarized MetricFamily. Additionally they may have a Trace ID attached to them which when used to together with a tracing system can provide more detailed information related to the specific service.

To enable this experimental feature you must have at least version v2.26.0 and add `--enable-feature=exemplar-storage` to your arguments.

Protobuf format

Earlier versions of Prometheus supported an exposition format based on Protocol Buffers (aka Protobuf) in addition to the current text-based format. With Prometheus 2.0, the Protobuf format was marked as deprecated and Prometheus stopped ingesting samples from said exposition format.

However, new (experimental) features were added to Prometheus where the Protobuf format was considered the most viable option. Making Prometheus accept Protocol Buffers once again.

When such features are enabled either by feature flag (`--enable-feature=created-timestamp-zero-ingestion`) or by setting the appropriate configuration option (`scrape_native_histograms: true`) then Protobuf will be favored over other exposition formats.

document.

The current version of the original Protobuf format (with the recent extensions for native histograms) is maintained in the prometheus/client_model repository.

Previous
Writing exporters

Edit

Next
UTF-8 escaping schemes

Prometheus

Show nav

UTF-8 metric and label name escaping schemes

Abstract

This document specifies the different escaping schemes used by Prometheus during generation of text exposition for metric and label names that contain characters outside the legacy character set. These schemes are negotiated during scraping via the `escaping` parameter in the Accept and Content-Type headers.

Introduction

Prometheus supports multiple escaping schemes to handle metric and label names in text exposition that contain characters outside the legacy character set (a-zA-Z0-9_:). The escaping scheme is negotiated during scraping and affects how metric producers should format their metric names.

Escaping Schemes

No Escaping (allow-utf-8)

Header Value: `escaping=allow-utf-8`

Behavior:

- Metric and label names MUST be valid UTF-8 strings.
- When names appear inside double quotes in the exposition format, `\`, `\n`, and `"` MUST be escaped with a backslash.
- When names appear unquoted in the exposition format, `\` and `\n` MUST be escaped with a backslash.

Prometheus

Underscore Escaping (underscores)

Header Value: `escaping=underscores`

Behavior:

- Any character that is not in the legacy character set (a-zA-Z0-9_:) MUST be replaced with an underscore.
- The first character MUST be either a letter, underscore, or colon.
- Subsequent characters MUST be either letters, numbers, underscores, or colons.
- Example: `metric.name/with/slashes` becomes `metric_name_with_slashes` .

Dots Escaping (dots)

Header Value: `escaping=dots`

Behavior:

- Dots (.) MUST be replaced with `_dot_` .
- Existing underscores MUST be replaced with double underscores (`__`).
- Other non-legacy characters MUST be replaced with single underscores.
- The first character MUST be either a letter, underscore, or colon.
- Subsequent characters MUST be either letters, numbers, underscores, or colons.
- Example: `metric.name.with.dots` becomes `metric_dot_name_dot_with_dot_dots` .

Value Encoding Escaping (values)

Header Value: `escaping=values`

Behavior:

- The name MUST be prefixed with `U__` .
- Each character that is not part of the legacy character set (a-zA-Z0-9_:) MUST be replaced with its Unicode code point in hexadecimal, surrounded by underscores.
- Single underscores MUST be replaced with double underscores.
- Example: `metric.name` becomes `U__metric_2E_name` (where 2E is the hex Unicode code point for '.').

used.

Security Considerations

1. Targets MUST validate input names before applying escaping.
2. The escaping scheme MUST be validated to prevent injection attacks.
3. The `allow-utf-8` scheme MUST only be used when both producer and consumer support UTF-8 names.

Previous
Exposition formats

Edit

Next
Content negotiation

Prometheus

Show nav

Scrape protocol content negotiation

Abstract

This document specifies the protocol negotiation mechanism used by Prometheus when scraping metrics from targets. It defines the Accept header format, supported Content Types, and the negotiation process for determining the best available format for metric exposition.

Introduction

Prometheus supports multiple formats for scraping metrics, including both text-based and binary protobuf formats. Based on the value of the Accept header, the target will pick the best available Content Type for its reply.

Protocol Types

Supported Protocols

The following protocols are supported by Prometheus:

1. `PrometheusProto` - Binary protobuf format
2. `PrometheusText0.0.4` - Prometheus text format version 0.0.4
3. `PrometheusText1.0.0` - Prometheus text format version 1.0.0
4. `OpenMetricsText0.0.1` - OpenMetrics text format version 0.0.1
5. `OpenMetricsText1.0.0` - OpenMetrics text format version 1.0.0

Protocol Headers

Each protocol MUST be associated with a specific MIME type and version:

Protocol	MIME Type	Parameters
PrometheusProto	application/vnd.google.protobuf	proto=io.prometheus.client.MetricFamily;encoding=delimited
PrometheusText0.0.4	text/plain	version=0.0.4
PrometheusText1.0.0	text/plain	version=1.0.0

OpenMetricsText1.0.0	application/openmetrics-text	version=1.0.0

Accept Header Construction

The Accept header is constructed by Prometheus to indicate what formats it supports.

Basic Format

The Accept header MUST be constructed as follows:

1. For each protocol supported by the target:
   - The protocol's MIME type and parameters MUST be specified.
   - For protobuf protocols, an encoding of "delimited" MUST be specified.
   - For PrometheusText1.0.0 and OpenMetricsText1.0.0, the escaping scheme parameter SHOULD be appended.
   - A quality value (q) parameter SHOULD be appended.
2. A catch-all `*/*` with the lowest quality value SHOULD be appended.

Quality Values

Quality values SHOULD be assigned in descending order based on the protocol's position in the Accept header:

- First protocol: q=0.{n+1}
- Second protocol: q=0.{n}
- And so on, where n is the number of supported protocols

Escaping Scheme

For PrometheusText1.0.0 and OpenMetricsText1.0.0 protocols, the Accept header SHOULD include an escaping scheme parameter: `escaping=<scheme>`

Where `<scheme>` MUST be one of:

- `allow-utf-8`
- `underscores`
- `dots`
- `values`

See Escaping Schemes spec for details on how the escaping schemes function.

Compression

- `identity` if compression is disabled

Selection of Format

The scrape target SHOULD use the following process to select an appropriate Content-Type based on the list of protocols in the Accept header generated by Prometheus:

1. It MUST use the protocol in the Accept header with the highest weighting that is supported by Prometheus.
2. If no protocols are supported, the target MAY use a user-configured fallback scrape protocol.
3. If no fallback is specified, the target MUST use PrometheusText0.0.4 as a last resort.

Content-Type Response

Targets SHOULD respond with a Content-Type header that matches one of the accepted formats. The Content-Type header MUST include:

1. The appropriate MIME type.
2. The version parameter.
3. For text formats version 1.0.0 and above, the escaping scheme parameter.

Security Considerations

1. Targets MUST validate the Accept header to prevent potential injection attacks
2. The escaping scheme parameter MUST be validated to prevent protocol confusion
3. Content-Type headers MUST be properly sanitized to prevent MIME type confusion

Examples

Default Accept Header

```
Accept: application/openmetrics-text;version=1.0.0;escaping=allow-utf-8;q=0.5,application/openm
```

Protobuf-First Accept Header

```
Accept: application/vnd.google.protobuf;proto=io.prometheus.client.MetricFamily;encoding=delimi
openmetrics-text;version=1.0.0;escaping=allow-utf-8;q=0.4,application/openmetrics-text;version=
```

Prometheus

UTF-8 escaping schemes

Edit

Security

Prometheus

Show nav

Security model

> **NOTE**: Before we dive into the technical details below, we would like to emphasize that Prometheus, as a monitoring system, collects and serves information about the systems it is monitoring. Therefore, the HTTP endpoints provided by Prometheus components should not be exposed to publicly accessible networks like the internet (unless you know what you are doing and have taken appropriate measures). This includes (but is not limited to) the `/metrics` endpoint of instrumented binaries, the various API endpoints of server components, and the `/pprof` endpoint of server components implemented in Go. Furthermore, it is easily possible to overload and ultimately DoS servers with requests to these endpoints.

Prometheus is a sophisticated system with many components and many integrations with other systems. It can be deployed in a variety of trusted and untrusted environments.

This page describes the general security assumptions of Prometheus and the attack vectors that some configurations may enable.

As with any complex system, it is near certain that bugs will be found, some of them security-relevant.

For *security bugs* that have already been publicly disclosed (e.g. via public CVEs) and require more than just a dependency version bump to fix, please open a Bug report including all relevant details, unless one has already been filed.

If you discover a *security bug* that has not yet been publicly disclosed, please report it privately to the maintainers listed in the MAINTAINERS of the relevant repository and CC prometheus-team@googlegroups.com. We will fix the issue as soon as possible and coordinate a release date with you. You will be able to choose if you want public acknowledgement of your effort and if you want to be mentioned by name.

Most dependency version updates are handled automatically. However, if a *security bug* fix only requires updating a dependency version and our automation missed it, feel free to

Automated security scanners

Special note for security scanner users: Please be mindful with the reports produced. Most scanners are generic and produce lots of false positives. More and more reports are being sent to us, and it takes a significant amount of work to go through all of them and reply with the care you expect. This problem is particularly bad with Go and NPM dependency scanners.

As a courtesy to us and our time, we would ask you not to submit raw reports. Instead, please submit them with an analysis outlining which specific results are applicable to us and why.

Additionally be aware that as an open source project, we generally do not have access to commercial scanning tools and find their output is often misleading or just plain wrong. For Go code if your report does not reproduce with the open source govulncheck tool run on the *source code* of the version you believe is affected (not a binary, as that cannot do a full analysis) -- then we ask you to triple check your findings (including whether that code is actually reachable from within the Prometheus codebase).

Prometheus is maintained by volunteers, not by a company. Therefore, fixing security issues is done on a best-effort basis. We strive to release security fixes within 7 days for: Prometheus, Alertmanager, Node Exporter, Blackbox Exporter, and Pushgateway.

Prometheus

It is presumed that untrusted users have access to the Prometheus HTTP endpoint and logs. They have access to all time series information contained in the database, plus a variety of operational/debugging information.

It is also presumed that only trusted users have the ability to change the command line, configuration file, rule files and other aspects of the runtime environment of Prometheus and other components.

Which targets Prometheus scrapes, how often and with what other settings is determined entirely via the configuration file. The administrator may decide to use information from service discovery systems, which combined with relabelling may grant some of this control to anyone who can modify data in that service discovery system.

Scraped targets may be run by untrusted users. It should not by default be possible for a target to expose data that impersonates a different target. The `honor_labels` option administrative HTTP API which includes functionality such as deleting time series. This is disabled by default. If enabled, administrative and mutating functionality will be accessible under the `/api/*/admin/` paths. The `--web.enable-lifecycle` flag controls HTTP reloads and shutdowns of Prometheus. This is also disabled by default. If enabled they will be accessible under the `/-/reload` and `/-/quit` paths.

In Prometheus 1.x, `/-/reload` and using `DELETE` on `/api/v1/series` are accessible to anyone with access to the HTTP API. The `/-/quit` endpoint is disabled by default, but can be enabled with the `-web.enable-remote-shutdown` flag.

The remote read feature allows anyone with HTTP access to send queries to the remote read endpoint. If for example the PromQL queries were ending up directly run against a relational database, then anyone with the ability to send queries to Prometheus (such as via Grafana) can run arbitrary SQL against that database.

Alertmanager

Any user with access to the Alertmanager HTTP endpoint has access to its data. They can create and resolve alerts. They can create, modify and delete silences.

Where notifications are sent to is determined by the configuration file. With certain templating setups it is possible for notifications to end up at an alert-defined destination. For example if notifications use an alert label as the destination email address, anyone who can send alerts to the Alertmanager can send notifications to any email address. If the alert-defined destination is a templatable secret field, anyone with access to either Prometheus or Alertmanager will be able to view the secrets.

Any secret fields which are templatable are intended for routing notifications in the above use case. They are not intended as a way for secrets to be separated out from the configuration files using the template file feature. Any secrets stored in template files could be exfiltrated by anyone able to configure receivers in the Alertmanager configuration file. For example in large setups, each team might have an alertmanager configuration file fragment which they fully control, that are then combined into the full final configuration file.

Pushgateway

Any user with access to the Pushgateway HTTP endpoint can create, modify and delete the metrics contained within. As the Pushgateway is usually scraped with `honor_labels`

The `--web.enable-admin-api` flag controls access to the administrative HTTP API, which includes functionality such as wiping all the existing metric groups. This is disabled by default. If enabled, administrative functionality will be accessible under the `/api/*/admin/` paths.

Exporters

Exporters generally only talk to one configured instance with a preset set of commands/requests, which cannot be expanded via their HTTP endpoint.

There are also exporters such as the SNMP and Blackbox exporters that take their targets from URL parameters. Thus anyone with HTTP access to these exporters can make them send requests to arbitrary endpoints. As they also support client-side authentication, this could lead to a leak of secrets such as HTTP Basic Auth passwords or SNMP community strings. Challenge-response authentication mechanisms such as TLS are not affected by this.

Client Libraries

Client libraries are intended to be included in users' applications.

If using a client-library-provided HTTP handler, it should not be possible for malicious requests that reach that handler to cause issues beyond those resulting from additional load and failed scrapes.

Authentication, Authorization, and Encryption

Prometheus, and most exporters, support TLS. Including authentication of clients via TLS client certificates. Details on configuring Prometheus are `here` .

The Go projects share the same TLS library, based on the Go crypto/tls library. We default to TLS 1.2 as minimum version. Our policy regarding this is based on Qualys SSL Labs recommendations, where we strive to achieve a grade 'A' with a default configuration and correctly provided certificates, while sticking as closely as possible to the upstream Go defaults. Achieving that grade provides a balance between perfect security and usability.

TLS will be added to Java exporters in the future.

If you have special TLS needs, like a different cipher suite or older TLS version, you can tune the minimum TLS version and the ciphers, as long as the cipher is not marked as insecure

HTTP Basic Authentication is also supported. Basic Authentication can be used without TLS, but it will then expose usernames and passwords in cleartext over the network.

On the server side, basic authentication passwords are stored as hashes with the bcrypt algorithm. It is your responsibility to pick the number of rounds that matches your security standards. More rounds make brute-force more complicated at the cost of more CPU power and more time to authenticate the requests.

Various Prometheus components support client-side authentication and encryption. If TLS client support is offered, there is often also an option called `insecure_skip_verify` which skips SSL verification.

API Security

As administrative and mutating endpoints are intended to be accessed via simple tools such as cURL, there is no built in CSRF protection as that would break such use cases. Accordingly when using a reverse proxy, you may wish to block such paths to prevent CSRF.

For non-mutating endpoints, you may wish to set CORS headers such as `Access-Control-Allow-Origin` in your reverse proxy to prevent XSS.

If you are composing PromQL queries that include input from untrusted users (e.g. URL parameters to console templates, or something you built yourself) who are not meant to be able to run arbitrary PromQL queries make sure any untrusted input is appropriately escaped to prevent injection attacks. For example `up{job="<user_input>"}` would become `up{job=""} or some_metric{zzz=""}` if the `<user_input>` was `"} or some_metric{zzz="`.

For those using Grafana note that dashboard permissions are not data source permissions, so do not limit a user's ability to run arbitrary queries in proxy mode.

Secrets

Non-secret information or fields may be available via the HTTP API and/or logs.

In Prometheus, metadata retrieved from service discovery is not considered secret. Throughout the Prometheus system, metrics are not considered secret.

Fields containing secrets in configuration files (marked explicitly as such in the documentation) will not be exposed in logs or via the HTTP API. Secrets should not be

Secrets from other sources used by dependencies (e.g. the `AWS_SECRET_KEY` environment variable as used by EC2 service discovery) may end up exposed due to code outside of our control or due to functionality that happens to expose wherever it is stored.

Denial of Service

There are some mitigations in place for excess load or expensive queries. However, if too many or too expensive queries/metrics are provided components will fall over. It is more likely that a component will be accidentally taken out by a trusted user than by malicious action.

It is the responsibility of the user to ensure they provide components with sufficient resources including CPU, RAM, disk space, IOPS, file descriptors, and bandwidth.

It is recommended to monitor all components for failure, and to have them automatically restart on failure.

Libraries

This document considers vanilla binaries built from the stock source code. Information presented here does not apply if you modify Prometheus source code, or use Prometheus internals (beyond the official client library APIs) in your own code.

Build Process

The build pipeline for Prometheus runs on third-party providers to which many members of the Prometheus development team and the staff of those providers have access. If you are concerned about the exact provenance of your binaries, it is recommended to build them yourself rather than relying on the pre-built binaries provided by the project.

Prometheus-Community

The repositories under the Prometheus-Community organization are supported by third-party maintainers.

Some repositories under that organization might have a different security model than the ones presented in this document. In such a case, please refer to the documentation of those repositories.

External audits

- In 2018, CNCF sponsored an external security audit by cure53 which ran from April 2018 to June 2018. For more details, please read the final report of the audit.
- In 2020, CNCF sponsored a second audit by cure53 of Node Exporter.
- In 2023, CNCF sponsored a software supply chain security assessment of Prometheus by Chainguard.

Previous
Content negotiation

Edit

Next
Integrations

Prometheus

Show nav

Integrations

In addition to client libraries and exporters and related libraries, there are numerous other generic integration points in Prometheus. This page lists some of the integrations with these.

Not all integrations are listed here, due to overlapping functionality or still being in development. The exporter default port wiki page also happens to include a few non-exporter integrations that fit in these categories.

File Service Discovery

For service discovery mechanisms not natively supported by Prometheus, file-based service discovery provides an interface for integrating.

- Kuma
- Lightsail
- Netbox
- Packet
- Scaleway

Remote Endpoints and Storage

The remote write and remote read features of Prometheus allow transparently sending and receiving samples. This is primarily intended for long term storage. It is recommended that you perform careful evaluation of any solution in this space to confirm it can handle your data volumes.

- AppOptics: write
- AWS Timestream: read and write
- Azure Data Explorer: read and write
- Azure Event Hubs: write
- Chronix: write
- Cortex: read and write
- CrateDB: read and write

- Google Cloud Spanner: read and write
- Grafana Mimir: read and write
- Graphite: write
- GreptimeDB: read and write
- InfluxDB: read and write
- Instana: write
- IRONdb: read and write
- Kafka: write
- M3DB: read and write
- Mezmo: write
- New Relic: write
- OpenTSDB: write
- QuasarDB: read and write
- SignalFx: write
- Splunk: read and write
- Sysdig Monitor: write
- TiKV: read and write
- Thanos: read and write
- VictoriaMetrics: write
- Wavefront: write

Prom-migrator is a tool for migrating data between remote storage systems.

Alertmanager Webhook Receiver

For notification mechanisms not natively supported by the Alertmanager, the webhook receiver allows for integration.

- alertmanager-webhook-logger: logs alerts
- Alertsnitch: saves alerts to a MySQL database
- All Quiet: on-call & incident management
- Asana
- AWS SNS
- Better Uptime
- Canopsis
- DingTalk
- Discord
- GitLab
- Gotify

- iLert
- IRC Bot
- JIRAlert
- Matrix
- Notion: creates/updates record in a Notion database
- Phabricator / Maniphest
- prom2teams: forwards notifications to Microsoft Teams
- Ansible Tower: call Ansible Tower (AWX) API on alerts (launch jobs etc.)
- Signal
- SIGNL4
- Simplepush
- SMS: supports multiple providers
- SNMP traps
- Squadcast
- STOMP
- Telegram bot
- xMatters
- XMPP Bot
- Zenduty
- Zoom

Management

Prometheus does not include configuration management functionality, allowing you to integrate it with your existing systems or build on top of it.

- Prometheus Operator: Manages Prometheus on top of Kubernetes
- Promgen: Web UI and configuration generator for Prometheus and Alertmanager

Other

- Alert analysis: Stores alerts into a ClickHouse database and provides alert analysis dashboards
- karma: alert dashboard
- PushProx: Proxy to transverse NAT and similar network setups
- Promdump: kubectl plugin to dump and restore data blocks
- Promregator: discovery and scraping for Cloud Foundry applications
- pint: Prometheus rule linter

Prometheus

Security

Edit

Alerting overview

Prometheus

Show nav

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 have a single unit (i.e. do not mix seconds with milliseconds, or seconds with bytes).
- ...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)

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:

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.

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.
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.

Previous
HTTPS and authentication

Edit

Next
Consoles and dashboards

Prometheus

Show nav

Consoles and dashboards

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.

Prometheus

Prometheus

Show nav

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.

Prometheus

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

Prometheus

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.

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.

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.

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.

Prometheus

Prometheus

Show nav

Histograms and summaries

NOTE: This document predates native histograms (added as an experimental feature in Prometheus v2.40 and becoming stable in v3.8). The intention is to thoroughly update this document in the foreseeable future.

Histograms and summaries are more complex metric types. Not only does a single histogram or summary create a multitude of time series, it is also more difficult to use these metric types correctly. This section helps you to pick and configure the appropriate metric type for your use case.

Library support

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

Some libraries support only one of the two types, or they support summaries only in a limited fashion (lacking quantile calculation).

Count and sum of observations

Histograms and summaries both sample observations, typically request durations or response sizes. They track the number of observations *and* the sum of the observed values, allowing you to calculate the *average* of the observed values. Note that the number of observations (showing up in Prometheus as a time series with a `_count` suffix) is inherently a counter (as described above, it only goes up). The sum of observations (showing up as a time series with a `_sum` suffix) behaves like a counter, too, as long as there are no negative observations. Obviously, request durations or response sizes are never negative. In principle, however, you can use summaries and histograms to observe negative values (e.g. temperatures in centigrade). In that case, the sum of observations can go down, so you cannot apply `rate()` to it anymore. In those rare cases where you need to apply `rate()` and cannot avoid negative observations, you can use two separate summaries, one for

To calculate the average request duration during the last 5 minutes from a histogram or summary called `http_request_duration_seconds`, use the following expression:

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

Apdex score

A straight-forward use of histograms (but not summaries) is to count observations falling into particular buckets of observation values.

You might have an SLO to serve 95% of requests within 300ms. In that case, configure a histogram to have a bucket with an upper limit of 0.3 seconds. You can then directly express the relative amount of requests served within 300ms and easily alert if the value drops below 0.95. The following expression calculates it by job for the requests served in the last 5 minutes. The request durations were collected with a histogram called `http_request_duration_seconds`.

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

You can approximate the well-known Apdex score in a similar way. Configure a bucket with the target request duration as the upper bound and another bucket with the tolerated request duration (usually 4 times the target request duration) as the upper bound. Example: The target request duration is 300ms. The tolerable request duration is 1.2s. The following expression yields the Apdex score for each job over the last 5 minutes:

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

Note that we divide the sum of both buckets. The reason is that the histogram buckets are cumulative. The `le="0.3"` bucket is also contained in the `le="1.2"` bucket; dividing it by 2 corrects for that.

Quantiles

You can use both summaries and histograms to calculate so-called φ-quantiles, where 0 ≤ φ ≤ 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 the 95th percentile.

The essential difference between summaries and histograms is that summaries calculate streaming φ-quantiles on the client side and expose them directly, while histograms expose bucketed observation counts and the calculation of quantiles from the buckets of a histogram happens on the server side using the `histogram_quantile()` function.

The two approaches have a number of different implications:

	Histogram	**Summary**
Required configuration	Pick buckets suitable for the expected range of observed values.	Pick desired φ-quantiles and sliding window. Other φ-quantiles and sliding windows cannot be calculated later.
Client performance	Observations are very cheap as they only need to increment counters.	Observations are expensive due to the streaming quantile calculation.
Server performance	The server has to calculate quantiles. You can use recording rules should the ad-hoc calculation take too long (e.g. in a large dashboard).	Low server-side cost.
Number of time series (in addition to the `_sum` and `_count` series)	One time series per configured bucket.	One time series per configured quantile.
Quantile error (see below for details)	Error is limited in the dimension of observed values by the width of the relevant bucket.	Error is limited in the dimension of φ by a configurable value.
Specification of φ-quantile and sliding time-window	Ad-hoc with Prometheus expressions.	Preconfigured by the client.
Aggregation	Ad-hoc with Prometheus expressions.	In general not aggregatable.

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 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.

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

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 expression above and you do not need to reconfigure the clients.

Errors of quantile estimation

Quantiles, whether calculated client-side or server-side, 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, if you could plot the "true" histogram, you would see a very sharp spike at 220ms. In the Prometheus histogram metric as configured above, almost 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 histogram implementation guarantees that the true 95th percentile is somewhere between 200ms and 300ms. To return a single value (rather than an interval), it applies linear interpolation, which yields 295ms in this case. The calculated quantile gives you the impression that you are close to breaching the SLO, but in reality, the 95th percentile is a tiny bit above 220ms, a quite comfortable distance to your SLO.

calculated to be 442.5ms, although the correct value is close to 320ms. While you are only a tiny bit outside of your SLO, the calculated 95th quantile looks much worse.

A summary would have had no problem calculating the correct percentile value in both cases, at least if it uses an appropriate algorithm on the client side (like the one used by the Go client). 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, even in this contrived example of very sharp spikes in the distribution of observed values, the histogram was able to identify correctly if you were within or outside of your SLO. Also, the closer the actual value of the quantile is to our SLO (or in other words, the value we 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 histogram, the calculated value is accurate, as the value of the 95th percentile happens to coincide with one of the bucket boundaries. Even slightly different values would still be accurate as the (contrived) even distribution within the relevant buckets is exactly what the linear interpolation within a bucket assumes.

The error of the quantile reported by a summary gets more interesting now. 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). 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.

Two rules of thumb:

1. If you need to aggregate, choose histograms.

What can I do if my client library does not support the metric type I need?

Implement it! Code contributions are welcome. In general, we expect histograms to be more urgently needed than summaries. Histograms are also easier to implement in a client library, so we recommend to implement histograms first, if in doubt.

← Previous Instrumentation

Edit

Next Alerting →

Prometheus

Show nav

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.

Prometheus

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.

Prometheus

Prometheus

Show nav

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.

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.

← **Previous** Recording rules

Edit

Next Remote write tuning →

Prometheus

Show nav

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.

`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`

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.

← **Previous** When to use the Pushgateway

Edit

Next Basic auth →

Prometheus

Prometheus

Show nav

Native Histograms

Native histograms were introduced as an experimental feature in November 2022. They are a concept that touches almost every part of the Prometheus stack. The first version of the Prometheus server supporting native histograms was v2.40.0. The support had to be enabled via a feature flag `--enable-feature=native-histograms` . Starting with v3.8.0, native histograms are supported as a stable feature. However, scraping native histograms still has to be activated explicitly via the `scrape_native_histograms` configuration setting. To ease transition from the feature flag to the configuration setting, setting the feature flag in v3.8 has the only remaining effect to set `scrape_native_histograms` to `true` by default. Starting with v3.9, the feature flag is a true no-op and explicitly setting `scrape_native_histograms` is required. Sending over Remote-Write needs to be enabled with by the `send_native_histograms` remote write config. (From v4 on, both `scrape_native_histograms` and `send_native_histograms` will default to `true` .)

Due to the pervasive nature of the changes related to native histograms, the documentation of those changes and explanation of the underlying concepts are widely distributed over various channels (like the documentation of affected Prometheus components, doc comments in source code, sometimes the source code itself, design docs, conference talks, …). This document intends to gather all these pieces of information and present them concisely in a unified context. This document prefers to link existing detailed documentation rather than restating it, but it contains enough information to be comprehensible without referring to other sources. With all that said, it should be noted that this document is neither suitable as an introduction for beginners nor does it focus on the needs of developers. For the former, the plan is to provide an updated version of the Best Practices article on histograms and summaries. (TODO: And a blog post or maybe even a series of them.) For the latter, there is Carrie Edward's Developer’s Guide to Prometheus Native Histograms.

While formal specifications are supposed to happen in their respective context (e.g. OpenMetrics changes will be specified in the general OpenMetrics specification), some parts of this document take the shape of a specification. In those parts, the key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” are used as described in RFC 2119.

This document still contains a lot of TODOs even though the feature is considered stable and we don't expect breaking changes before v4.0.0. These TODOs are reminders for completing

Introduction

The core idea of native histograms is to treat histograms as first class citizens in the Prometheus data model. Elevating histograms to a "native" sample type is the fundamental prerequisite for the key properties listed below, which explains the choice of the name *native histograms*.

Prior to the introduction of native histograms, all Prometheus sample values have been 64-bit floating point values (short *float64* or just *float*). These floats can directly represent *gauges* or *counters*. The Prometheus metric types *summary* and (the classic version of) *histogram*, as they exist in exposition formats, are broken down into float components upon ingestion: A *sum* and a *count* component for both types, a number of *quantile* samples for a summary and a number of *bucket* samples for a (classic) histogram.

With native histograms, a new structured sample type is introduced. A single sample represents the previously known *sum* and *count* plus a dynamic set of buckets. This is not limited to ingestion, but PromQL expressions may also return the new sample type where previously it was only possible to return float samples.

Native histograms have the following key properties:

1. A sparse bucket representation, allowing (near) zero cost for empty buckets.
2. Coverage of the full float64 range of values.
3. No configuration of bucket boundaries during instrumentation.
4. Dynamic resolution picked according to simple configuration parameters.
5. Sophisticated exponential bucketing schemas, ensuring mergeability between all histograms using those schemas.
6. An efficient data representation for both exposition and storage.

These key properties are fully realized with standard bucketing schemas. There are other schemas with different trade-offs that might only feature a subset of these properties. See the Schema section below for details

Compared to the previously existing "classic" histograms, native histograms (with standard bucketing schemas) allow a higher bucket resolution across arbitrary ranges of observed values at a lower storage and query cost with very little to no configuration required. Even partitioning histograms by labels is now much more affordable.

Because the sparse representation (property 1 in the list above) is so crucial for many of the other benefits of native histograms, *sparse histograms* was a common name for *native histograms* early during the design process. However, other key properties like the

Design docs

These are the design docs that guided the development of native histograms. Some details are obsolete now, but they describe rather well the underlying concepts and how they evolved.

- Sparse high-resolution histograms for Prometheus, the original design doc.
- Prometheus Sparse Histograms and PromQL, more an exploratory document than a proper design doc about the handling of native histograms in PromQL.

Conference talks

A more approachable way of learning about native histograms is to watch conference talks, of which a selection is presented below. As an introduction, it might make sense to watch these talks and then return to this document to learn about all the details and technicalities.

- Secret History of Prometheus Histograms about the classic histograms and why Prometheus kept them for so long.
- Prometheus Histograms – Past, Present, and Future is the inaugural talk about the new approach that led to native histograms.
- Better Histograms for Prometheus explains why the concepts work out in practice.
- Native Histograms in Prometheus presents and explains native histograms after the actual implementation.
- PromQL for Native Histograms explains the usage of native histograms in PromQL.
- Prometheus Native Histograms in Production provides an analysis of performance and resource consumption.
- Using OpenTelemetry's Exponential Histograms in Prometheus covers the interoperability with the OpenTelemetry.

Glossary

- A **native histogram** is an instance of the new complex sample type representing a full histogram that this document is about. Where the context is sufficiently clear, it is often just called a *histogram* below.
- A **classic histogram** is an instance of the older sample type representing a histogram with fixed buckets, formerly just called a *histogram*. It exists as such in the exposition formats, but is broken into a number of float samples upon ingestion into Prometheus.

Prometheus

Data model

This section describes the data model of native histograms in general. It avoids implementation specifics as far as possible. This includes terminology. For example, a *list* described in this section will become a *repeated message* in a protobuf implementation and (most likely) a *slice* in a Go implementation.

General structure

Similar to a classic histogram, a native histogram has a field for the *count* of observations and a field for the *sum* of observations. While the count of observation is generally non-negative (with the only exception being intermediate results in PromQL), the sum of observations might have any float64 value.

In addition, a native histogram contains the following components, which are described in detail in dedicated sections below:

- A *schema* to identify the method of determining the boundaries of any given bucket with an index *i*.
- A sparse representation of indexed buckets, mirrored for positive and negative observations.
- A *zero bucket* to count observations close to zero.
- A (possibly empty) list of *custom values*.
- *Exemplars*.

Flavors

Any native histogram has a specific flavor along each of two independent dimensions:

1. Counter vs. gauge: Usually, a histogram is "counter like", i.e. each of its buckets acts as a counter of observations. However, there are also "gauge like" histograms where each bucket is a gauge, representing arbitrary distributions at a point in time. The concept of a gauge histogram was previously introduced for classic histograms by OpenMetrics.
2. Integer vs. floating point (short: float): The obvious use case of histograms is to count observations, resulting in integer numbers of observations ≥ 0 within each bucket, including the *zero bucket*, and for the total *count* of observations, represented as

float64 in either case.

Float histograms are occasionally used in direct instrumentation for “weighted” observations, for example to count the number of seconds an observed value was falling into different buckets of a histogram. The far more common use case for float histograms is within PromQL, though. PromQL generally only acts on float values, so the PromQL engine converts every histogram retrieved from the TSDB to a float histogram first, and any histogram stored back into TSDB via recording rules is a float histogram. If such a histogram is effectively an integer histogram (because the value of all non-*sum* fields can be represented precisely as uint64), a TSDB implementation MAY convert them back to integer histograms to increase storage efficiency. (As of Prometheus v3.00, the TSDB implementation within Prometheus is not utilizing this option.) Note, however, that the most common PromQL function applied to a counter histogram is `rate`, which generally produces non-integer numbers, so that results of recording rules will commonly be float histograms with non-integer values anyway.

PromQL expression may even create “negative” histograms (e.g. by multiplying a histogram with -1). Those negative histograms are only allowed as intermediate results and are otherwise considered invalid. They cannot be represented in any of the exchange formats (exposition formats, remote-write, OTLP) and they cannot be stored in the TSDB. Also see the detailed section about negative histograms.

Treating native histograms explicitly as integer histograms vs. float histogram is a notable deviation from the treatment of conventional simple numeric samples, which are always treated as floats throughout the whole stack for the sake of simplicity.

The main reason for the more involved treatment of histograms is the easy efficiency gains in protobuf-based exposition formats. Protobuf uses varint encoding for integers, which reduces the data size for small integer values without requiring an additional compression layer. This benefit is amplified by the delta encoding of integer buckets, which generally results in smaller integer values. Floats, in contrast, always require 8 bytes in protobuf. In practice, many integers in an integer histogram will fit in 1 byte, and most will fit in 2 bytes, so that the explicit presence of integer histogram in a protobuf-exposition format results directly in a data size reduction approaching 8x for histograms with many buckets. This is particularly relevant as the overwhelming majority of histograms exposed by instrumented targets are integer histograms.

For similar reasons, the representation of integer histograms in RAM and on disk is generally more efficient than that of float histograms. This is less relevant than the benefits in the exposition format, though. For one, Prometheus uses Gorilla-style XOR encoding for floats, which reduces their size, albeit not as much as the double-delta encoding used for integers. More importantly, an implementation could always decide to internally use an integer

In a counter histogram, the total *count* of observation and the counts in the buckets individually behave like Prometheus counters, i.e. they only go down upon a counter reset. However, the *sum* of observation may decrease as a consequence of the observation of negative values. PromQL implementations MUST detect counter resets based on the whole histogram (see the counter reset considerations section below for details). (Note that this always has been a problem for the *sum* component of classic histograms and summaries, too. The approach so far was to accept that counter reset detection silently breaks for *sum* in those cases. Fortunately, negative observations are a very rare use case for Prometheus histograms and summaries.)

Schema

The *schema* is a signed integer value with a size of 8 bits (short: int8). It defines the way bucket boundaries are calculated. The currently valid values are -53 and the range between and including -4 and +8 (with a larger range between and including -9 and +52 being reserved, see below for details). More schemas may be added in the future. -53 is a schema for so-called *custom bucket boundaries* or short *custom buckets*, while the other schema numbers represent the different standard exponential schemas (short: *standard schemas*).

The standard schemas are mergeable with each other and are RECOMMENDED for general use cases. Larger schema numbers correspond to higher resolutions. Schema *n* has half the resolution of schema *n*+1, which implies that a histogram with schema *n*+1 can be converted into a histogram with schema *n* by merging neighboring buckets.

For any standard schema *n*, the boundaries of a bucket with index *i* calculated as follows (using Python syntax):

- The upper inclusive limit of a positive bucket: `(2**2**-n)**i`
- The lower exclusive limit of a positive bucket: `(2**2**-n)**(i-1)`
- The lower inclusive limit of a negative bucket: `-((2**2**-n)**i)`
- The upper exclusive limit of a negative bucket: `-((2**2**-n)**(i-1))`

i is an integer number that may be negative.

There are exceptions to the rules above concerning the largest and smallest finite values representable as a float64 (called `MaxFloat64` and `MinFloat64` in the following) and the positive and negative infinity values (`+Inf` and `-Inf`):

- The positive bucket that contains `MaxFloat64` (according to the boundary formulas above) has an upper inclusive limit of `MaxFloat64` (rather than the limit calculated by

called a *positive overflow bucket*.)
- The negative bucket that contains `MinFloat64` (according to the boundary formulas above) has a lower inclusive limit of `MinFloat64` (rather than the limit calculated by the formulas above, which would underflow float64).
- The next negative bucket (index i+1 relative to the bucket from the previous item) has an upper exclusive limit of `MinFloat64` and an lower inclusive limit of `-Inf`. (It could be called a *negative overflow bucket*.)
- Buckets beyond the `+Inf` and `-Inf` buckets described above MUST NOT be used.

There are more exceptions for values close to zero, see the zero bucket section below.

The current limits of -4 for the lowest resolution and 8 for the highest resolution have been chosen based on practical usefulness. Should a practical need arise for even lower or higher resolution, an extension of the range will be considered. However, a schema greater than 52 does not make sense as the growth factor from one bucket to the next would then be smaller than the difference between representable float64 numbers. Likewise, a schema smaller than -9 does not make sense either, as the growth factor would then exceed the largest float representable as float64. Therefore, the schema numbers between (and including) -9 and +52 are reserved for future standard schemas (following the formulas for bucket boundaries above) and MUST NOT be used for any other schemas.

Receivers of native histograms MAY, upon ingestion, reduce the schema and thereby the resolution of ingested histograms by merging buckets appropriately. Receivers MAY accept schemas between 9 and 52 if they reduce the schema upon ingestion to a valid number (i.e. between -4 and 8), following the formulas for bucket boundaries above.

If, after this optional schema conversion, the schema is still unknown to the receiver, there are the following options:

- If a scrape (including federation) contains one or more histograms with an unknown schema, the entire scrape MUST fail, following the Prometheus practice of avoiding incomplete scrapes.
- For any other ingestion paths (including replaying the WAL/WBL), the receiver MAY ignore histograms with unknown schemas and SHOULD notify the user about this omission in a suitable way.

When a TSDB implementation reads histograms from its permanent storage (excluding replaying the WAL/WBL), similar guidelines apply: Schemas between 9 and 52 MAY be converted to valid schemas. Otherwise, unknown schemas MUST return an error on retrieval, and the PromQL query that triggered the retrieval MUST fail.

can be used to represent a classic histogram as a native histogram. It can also be used if the exponential bucketing featured by the standard schemas is a bad match for the distribution to be represented by the histogram. Histograms with different custom bucket boundaries are generally not mergeable with each other. Therefore, schema -53 SHOULD only be used as an informed decision in specific use cases.

Buckets

For standard schemas, buckets are represented as two lists, one for positive buckets and one for negative buckets. For custom buckets (schema -53), only the positive bucket list is used, but repurposed for all buckets.

Any unpopulated buckets MAY be excluded from the lists. (Which is the reason why the buckets are often called *sparse buckets*.)

For float histograms, the elements of the lists are float64 and represent the bucket population directly. Bucket populations are generally non-negative, with the only exception being intermediate results in PromQL.

For integer histograms, the elements of the lists are signed 64-bit integers (short: int64), and each element represents the bucket population as a delta to the previous bucket in the list. The first bucket in each list contains an absolute population (which can also be seen as a delta relative to zero). The deltas MUST NOT evalute to a negative absolute bucket population.

To map buckets in the lists to the indices as defined in the previous section, there are two lists of so-called *spans*, one for the positive buckets and one for the negative buckets.

Each span consists of a pair of numbers, a signed 32-bit integer (short: int32) called *offset* and an unsigned 32-bit integer (short: uint32) called *length*. Only the first span in each list can have a negative offset. It defines the index of the first bucket in its corresponding bucket list. (Note that for NHCBs, the index is always positive, see the custom values section below for details.) The length defines the number of consecutive buckets the bucket list starts with. The offsets of the following spans define the number of excluded (and thus unpopulated buckets). The lengths define the number of consecutive buckets in the list following the excluded buckets.

The sum of all length values in each span list MUST be equal to the length of the corresponding bucket list.

although those offsets SHOULD be eliminated by adding their length to the previous span. Both cases are allowed so that producers of native histograms MAY pick whatever representation has the best resource trade-offs at that moment. For example, if a histogram is processed through various stages, it might be most efficient to only eliminate redundant spans after the last processing stage.

In a similar spirit, there are situation where excluding every unpopulated bucket from the bucket list is most efficient, but in other situations, it might be better to reduce the number of spans by representing small numbers of unpopulated buckets explicitly.

Note that future high resolution schemas might require offsets that are too large to be represented with an int32. An extension of the data model will be required in that case. (The current standard schema with the highest resolution is schema 8, for which the bucket that contains `MaxFloat64` has index 262144, and thus the `+Inf` overflow bucket has index 262145, while the largest number representable with int32 is 2147483647. The highest standard schema that would still work with int32 offsets would be schema 20, corresponding to a growth factor from bucket to bucket of only ~1.000000661.)

Examples

An integer histogram has the following positive buckets (index→population):

```
-2→3, -1→5, 0→0, 1→0, 2→1, 3→0, 4→3, 5→2
```

They could be represented in this way:

- Positive bucket list: `[3, 2, -4, 2, -1]`
- Positive span list: `[[-2, 2], [2,1], [1,2]]`

The second and third span could be merged into one if the single unpopulated bucket with index 3 is represented explicitly, leading to the following result:

- Positive bucket list: `[3, 2, -4, -1, 3, -1]`
- Positive span list: `[[-2, 2], [2,4]]`

Or merge all the spans into one by representing all unpopulated buckets above explicitly:

- Positive bucket list: `[3, 2, -5, 0, 1, -1, 3, -1]`
- Positive span list: `[[-2, 8]]`

Zero bucket

The number of observations in the zero bucket is tracked by a single uint64 (for integer histograms) or float64 (for float histograms). As for regular buckets, this number is generally non-negative.

The zero bucket has an additional parameter called the *zero threshold*, which is a float64 ≥ 0. If the threshold is set to zero, only observations of exactly zero go into the zero bucket, which is the case described above. If the threshold has a positive value, all observations within the closed interval [-threshold, +threshold] go to the zero bucket rather than a regular bucket. This has two use cases:

- Noisy observations close to zero tend to populate a high number of buckets. Those observations might happen due to numerical inaccuracies or if the source of the observations are actual physical measurements. A zero bucket with a relatively small threshold redirects those observations into a single bucket.
- If the user is more interested in the long tail of a distribution, far away from zero, a relatively large threshold of the zero bucket helps to avoid many high resolution buckets for a range that is not of interest.

The threshold of the zero bucket SHOULD coincide with a boundary of a regular bucket, which avoids the complication of the zero bucket overlapping with parts of a regular bucket. However, if such an overlap is happening, the observations that are counted in the regular bucket overlapping with the zero bucket MUST be outside of the [-threshold, +threshold] interval.

To merge histograms with the same zero threshold, the two zero buckets are simply added. If the zero thresholds in the source histograms are different, however, the largest threshold in any of the source histograms is chosen. If that threshold happens to be within any populated bucket in the other source histograms, the threshold is increased until one of the following is true for each source histogram:

- The new threshold coincides with the boundary of a populated bucket.
- The new threshold is not within any populated bucket.

Then the source zero buckets and any source buckets now inside the new threshold are added up to yield the population of the new zero bucket.

The zero bucket is not used if the schema is -53 (custom buckets).

Custom values

The only currently defined schema for which custom values are used is -53 (custom buckets). The remaining part of this section describes the usage of the custom values in more detail for this specific case.

The custom values represent the upper inclusive boundaries of the custom buckets. They are sorted in ascending fashion. The custom buckets themselves are stored using the positive bucket list and the positive span list, although their boundaries, as determined via the custom values, can be negative. The index of each of those "positive" buckets defines the zero-based position of their upper boundary within the custom values list.

The lower exclusive boundary is defined by the custom value preceding the upper boundary. For the first custom value (at position zero in the list), there is no preceding value, in which case the lower boundary is considered to be `-Inf` inclusively. Therefore, the custom bucket with index zero counts all observations between (and including) `-Inf` and the first custom value. In the common case that only positive observations are expected, the custom bucket with index zero SHOULD have an upper boundary of zero to clearly mark if there have been any observations at zero or below. (If there are indeed only positive observations, the custom bucket with index zero will stay unpopulated and therefore will never be represented explicitly. The only cost is the additional zero element at the beginning of the custom values list.)

Custom values MUST NOT be `+Inf`. Observations greater than the last custom value go into an overflow bucket with an upper boundary of `+Inf`. This overflow bucket is added with an index equal to the length of the custom values list. As a consequence, the upper boundary of the `+Inf` bucket often included in classic histograms is not represented explicitly in the custom values.

Custom values MUST NOT be `NaN`. This is explicitly excluded in OpenMetrics, but other exposition formats could, in principle, feature upper boundaries of `NaN` in classic histograms (presumably as a result of some error – such a boundary would not make any sense). Such a classic histogram MUST be rejected and cannot be converted into an NHCB.

Exemplars

A native histogram sample can have zero, one, or more exemplars. They work in the same way as conventional exemplars, but they are organized in a list (as there can be more than one), and they MUST have a timestamp.

Exemplars exposed as part of a classic histogram MAY be used by native histograms, if they have a timestamp.

limited sense in the context of a histogram. However, those values MUST still be handled properly, as described in the following.

The sum of observations is calculated as usual by adding the observation to the sum of observations, following normal floating point arithmetic. (For example, an observation of `NaN` will set the sum to `NaN`. An observation of `+Inf` will set the sum to `+Inf`, unless it is already `NaN` or `-Inf`, in which case the sum is set to `NaN`.)

An observation of `NaN` goes into no bucket, but increments the count of observations. This implies that the count of observations can be greater than the sum of all buckets (negative, positive, and zero buckets), and the difference is the number of `NaN` observations. (For an integer histogram without any `NaN` observations, the sum of all buckets is equal to the count of observations. Within the usual floating point precision limits, the same is true for a float histogram without any `NaN` observations.)

An observation of `+Inf` or `-Inf` increments the count of observations and increments a bucket chosen in the following way:

- With a standard schema, a `+Inf` observation increments the *positive overflow bucket* as described above.
- With a standard schema, a `-Inf` observation increments the *negative overflow bucket* as described above.
- With schema -53 (custom buckets), a `+Inf` observation increments the bucket with an index equal to the length of the custom values list.
- With schema -53 (custom buckets), a `-Inf` observation increments the bucket with index zero.

OpenTelemetry interoperability

Prometheus (Prom) native histograms with a standard schema can be easily mapped into an OpenTelemetry (OTel) exponential histogram and vice versa, as detailed in the following.

The Prom *schema* is equal to the *scale* in OTel, with the restriction that OTel allows lower values than -4 and higher values than +8. As described above, Prom has reserved more schema numbers to extend its range, should it ever by required in practice.

The index is offset by one, i.e. a Prom bucket with index *n* has index *n-1* for OTel.

OTel has a dense rather than a sparse representation of buckets. One might see OTel as "Prom with only one span".

(TODO: The OTel spec reads: “When zero_threshold is unset or 0, this bucket stores values that cannot be expressed using the standard exponential formula as well as values that have been rounded to zero.” Double-check if this really creates the same behavior. If there are problems close to zero, we could make Prom's spec more precise. If OTel counts NaN in the zero bucket, we have to add a note here.)

OTel exponential histograms only support standard exponential bucketing schemas (as the name suggests). Therefore, NHCBs (or native histograms with other future bucketing schemas) cannot be cleanly converted to OTel exponential histograms. However, conversion to a conventional OTel histogram with fixed buckets is still possible.

OTel histograms of any kind have optional fields for the minimum and maximum value observed in the histogram. These fields have no equivalent concept in Prometheus because counter histograms accumulate data over a long and unpredictable timespan and can be scraped at any time, so that tracking a minimum and maximum value is either infeasible or of limited use. Note, though, that native histograms enable a fairly accurate estimation of the maximum and minimum observation during arbitrary timespans, see the PromQL section.

Exposition formats

Metrics exposition in the classic Prometheus use case is dominated by strings because all the metric names, label names, and label values take much more space than the float64 sample values, even if the latter are represented in a potentially more verbose text form. This was one of the reasons why abandoning protobuf-based exposition seemed advantageous in the past.

In contrast, a native histogram, following the data model described above, consists of a lot more numerical data. This amplifies the advantages of a protobuf based format. Therefore, the previously abandoned protobuf-based exposition was revived to efficiently expose and scrape native histograms.

Classic Prometheus formats

At the time native histograms were conceived, OpenMetrics adoption was still lacking, and in particular, the protobuf version of OpenMetrics had no known applications at all. Therefore, the initial approach was to extend the classic Prometheus protobuf format to support native histograms. (An additional practical consideration was that the Go instrumentation library

The classic Prometheus text form was not extended for native histograms, and such an extension is not planned. (See also the OpenMetrics section below.)

There is a proto2 and a proto3 version of the protobuf specification, which both create the same wire format:

- proto2
- proto3

These files have comprehensive comments, which should enable an easy mapping from the proto spec to the data model described above.

Here are relevant parts from the proto3 file:

```
// [...]

message Histogram {
  uint64 sample_count       = 1;
  double sample_count_float = 4; // Overrides sample_count if > 0.
  double sample_sum         = 2;
  // Buckets for the classic histogram.
  repeated Bucket bucket = 3 [(gogoproto.nullable) = false]; // Ordered in increasi

  google.protobuf.Timestamp created_timestamp = 15;

  // Everything below here is for native histograms (also known as sparse histogram

  // schema defines the bucket schema. Currently, valid numbers are -4 <= n <= 8.
  // They are all for base-2 bucket schemas, where 1 is a bucket boundary in each ca
  // then each power of two is divided into 2^n logarithmic buckets.
  // Or in other words, each bucket boundary is the previous boundary times 2^(2^-n)
  // In the future, more bucket schemas may be added using numbers < -4 or > 8.
  sint32 schema           = 5;
  double zero_threshold   = 6; // Breadth of the zero bucket.
  uint64 zero_count       = 7; // Count in zero bucket.
  double zero_count_float = 8; // Overrides sb_zero_count if > 0.

  // Negative buckets for the native histogram.
  repeated BucketSpan negative_span = 9 [(gogoproto.nullable) = false];
  // Use either "negative_delta" or "negative_count", the former for
  // regular histograms with integer counts, the latter for float
```

Prometheus

```
  // Positive buckets for the native histogram.
  // Use a no-op span (offset 0, length 0) for a native histogram without any
  // observations yet and with a zero_threshold of 0. Otherwise, it would be
  // indistinguishable from a classic histogram.
  repeated BucketSpan positive_span = 12 [(gogoproto.nullable) = false];
  // Use either "positive_delta" or "positive_count", the former for
  // regular histograms with integer counts, the latter for float
  // histograms.
  repeated sint64 positive_delta = 13; // Count delta of each bucket compared to pre
  repeated double positive_count = 14; // Absolute count of each bucket.

  // Only used for native histograms. These exemplars MUST have a timestamp.
  repeated Exemplar exemplars = 16;
}

message Bucket {
  uint64   cumulative_count       = 1; // Cumulative in increasing order.
  double   cumulative_count_float = 4; // Overrides cumulative_count if > 0.
  double   upper_bound            = 2; // Inclusive.
  Exemplar exemplar               = 3;
}

// A BucketSpan defines a number of consecutive buckets in a native
// histogram with their offset. Logically, it would be more
// straightforward to include the bucket counts in the Span. However,
// the protobuf representation is more compact in the way the data is
// structured here (with all the buckets in a single array separate
// from the Spans).
message BucketSpan {
  sint32 offset = 1; // Gap to previous span, or starting point for 1st span (which
  uint32 length = 2; // Length of consecutive buckets.
}


// A BucketSpan defines a number of consecutive buckets in a native
// histogram with their offset. Logically, it would be more
// straightforward to include the bucket counts in the Span. However,
// the protobuf representation is more compact in the way the data is
// structured here (with all the buckets in a single array separate
// from the Spans).
```

```
  }

  // [...]
```

Note the following:

- Both native histograms and classic histograms are encoded by the same `Histogram` proto message, i.e. the existing `Histogram` message got extended with fields for native histograms.
- The fields for the sum and the count of observations and the `created_timestamp` are shared between classic and native histograms and keep working in the same way for both.
- The format originally did not support classic float histograms. While extending the format for native histograms, support for classic float histograms was added as a byproduct (see fields `sample_count_float`, `cumulative_count_float`).
- The `Bucket` field and the `Bucket` message are used for the buckets of a classic histogram. It is perfectly possible to create a `Histogram` message that represents both a classic and a native version of the same histogram. Parsers have the freedom to pick either or both versions (see also the scrape configuration section).
- The bucket population is encoded as absolute numbers in case of float histograms, and as deltas to the previous bucket (or to zero for the first bucket) in case of integer histograms. The latter leads to smaller numbers, which encode to a smaller message size because protobuf uses varint encoding for the `sint64` type.
- A native histogram that has received no observations yet and a classic histogram that has no buckets configured would look exactly the same as a protobuf message. Therefore, a `Histogram` message that is meant to be parsed as a native histogram MUST contain a “no-op span”, i.e. a `BucketSpan` with `offset` and `length` set to 0, in the repeated `positive_span` field.
- Any number of exemplars for native histograms MAY be added in the repeated `Exemplar` field of the `Histogram` message, but each one MUST have a timestamp. If there are no exemplars provided in this way, a parser MAY use timestamped exemplars provided for classic buckets (as at most one exemplar per bucket in the `Exemplar` field of the `Bucket` message).
- The number and distribution of native histogram exemplars SHOULD fit the use case at hand. Generally, the exemplar payload SHOULD NOT be much larger than the remaining part of the `Histogram` message, and the exemplars SHOULD fall into different buckets and cover the whole spread of buckets approximately evenly. (This is generally preferred over an exemplar distribution that proportionally represents the

directly exposed, but presented as classic histograms, to be converted (back) to NHCB upon ingestion. This is also true for federation. We might still add fields for the custom values in the future, should the need arise, e.g. for future schemas that also utilize custom values.

OpenMetrics

Currently (2024-11-03), OpenMetrics does not support native histograms.

Adding support to the protobuf version of OpenMetrics is relatively straightforward due to its similarity to the classic Prometheus protobuf format. A proposal in the form of a PR is under review.

Adding support to the text version of OpenMetrics is harder, but also highly desirable because there are many situations where the generation of protobuf is infeasible. A text format has to make a trade-off between readability for humans and efficient handling by machines (encoding, transport, decoding). Work on it is in progress. See the design doc for more details.

(TODO: Update section as progress is made.)

Instrumentation libraries

The protobuf specification enables low-level creation of metrics exposition including native histograms using the language specific bindings created by the protobuf compiler. However, for direct code instrumentation, an instrumentation library is needed.

Currently (2024-11-03), there are two official Prometheus instrumentation libraries supporting native histograms:

- Go: source – documentation
- Java: source – documentation

Adding native histogram support to other instrumentation libraries is relatively easy if the library already supports protobuf exposition. For purely text based libraries, the completion of a text based exposition format is a prerequisite. (TODO: Update this as needed.)

This section does not cover details of how to use individual instrumentation libraries (see the documentation linked above for that) but focuses on the common usage patterns and also provides general guidelines how to implement native histogram support as part of an

The actual instrumentation API for histograms does not change for native histograms. Both classic histograms and native histograms receive observations in the same way (with subtle differences concerning exemplars, see next paragraph). Instrumentation libraries can even maintain a classic and a native version of the same histogram and expose them in parallel so that the scraper can choose which version to ingest (see the section about exposition formats for details). The user chooses whether to expose classic and/or native histograms via configuration settings.

Exemplars for classic histograms are usually tracked by storing and exposing the most recent exemplar for each bucket. As long as classic buckets are defined, an instrumentation library MAY expose the same exemplars for the native version of the same histogram, as long as each exemplar has a timestamp. (In fact, a scraper MAY use the exemplars provided with the classic version of the histogram even if it is otherwise only ingesting the native version, see details in the exposition formats section.) However, a native histogram MAY be assigned any number of exemplars, and an instrumentation library SHOULD use this liberty to meet the best practices for exemplars as described in the exposition formats section.

An instrumentation library SHOULD offer the following configuration parameters for native histograms following standard schemas. Names are examples from the Go library – they have to be adjusted to the idiomatic style in other languages. The value in parentheses is the default value that the library SHOULD offer.

- `NativeHistogramBucketFactor` (1.1): A float greater than one to determine the initial resolution. The library picks a starting schema that results in a growth of the bucket width from one bucket to the next by a factor not larger than the provided value. See table below for example values.
- `NativeHistogramZeroThreshold` (2^{-128}): A float of value zero or greater to set the initial threshold for the zero bucket.

The resolution is set via a growth factor rather than providing the schema directly because most users will not know the mathematics behind the schema numbers. The notion of an upper limit for the growth factor from bucket to bucket is understandable without knowing about the internal workings of native histograms. The following table lists an example factor for each valid schema.

`NativeHistogramBucketFactor`	**resulting schema**
65536	-4
256	-3

4	-1
2	0
1.5	1
1.2	2
1.1	3
1.05	4
1.03	5
1.02	6
1.01	7
1.005	8

Limiting the bucket count

Buckets of native histograms are created dynamically when they are populated for the first time. An unexpectedly broad distribution of observed values can lead to an unexpectedly high number of buckets, requiring more memory than anticipated. If the distribution of observed values can be manipulated from the outside, this could even be used as a DoS attack vector via exhausting all the memory available to the program. Therefore, an instrumentation library SHOULD offer a bucket limitation strategy. It MAY set one by default, depending on the typical use cases the library is used for. (TODO: Maybe we should say that a strategy SHOULD be set by default. The Go library is currently not limiting the buckets by default, and no issues have been reported with that so far.)

The following describes the bucket limitation strategy implemented by the Go instrumentation library. Other libraries MAY follow this example, but other strategies might be feasible as well, depending on the typical usage pattern of the library.

The strategy is defined by three parameters: an unsigned integer `NativeHistogramMaxBucketNumber`, a duration `NativeHistogramMinResetDuration`, and a float `NativeHistogramMaxZeroThreshold`. If `NativeHistogramMaxBucketNumber` is zero (which is the default), buckets are not limited at all, and the other two parameters are ignored. If `NativeHistogramMaxBucketNumber` is set to a positive value, the library attempts to keep the bucket count of each histogram to the provided value. A typical value for the limit is 160, which is also the default value used by OTel exponential histograms in a similar the limit again:

1. If at least `NativeHistogramMinResetDuration` has passed since the last reset of the histogram (which includes the creation of the histogram), the whole histogram is reset, i.e. all buckets are deleted and the sum and count of observations as well as the zero bucket are set to zero. Prometheus handles this as a normal counter reset, which means that some observations will be lost between scrapes, so resetting should happen rarely compared to the scraping interval. Additionally, frequent counter resets might lead to less efficient storage in the TSDB (see the TSDB section for details). A `NativeHistogramMinResetDuration` of one hour is a value that should work well in most situations.
2. If not enough time has passed since the last reset (or if `NativeHistogramMinResetDuration` is set to zero, which is the default value), no reset is performed. Instead, the zero threshold is increased to merge buckets close to zero into the zero bucket, reducing the number of buckets in that way. The increase of the threshold is limited by `NativeHistogramMaxZeroThreshold`. If this value is already reached (or it is set to zero, which is the default), nothing happens in this step.
3. If the number of buckets still exceeds the limit, the resolution of the histogram is reduced by converting it to the next lower schema, i.e. by merging neighboring buckets, thereby doubling the width of the buckets. This is repeated until the bucket count is within the configured limit or schema -4 is reached.

If step 2 or 3 have changed the histogram, a reset will be performed once `NativeHistogramMinResetDuration` has passed since the last reset, not only to remove the buckets but also to return to the initial values for the zero threshold and the bucket resolution. Note that this is treated like a reset for other reasons in all aspects, including updating the so-called created timestamp.

It is tempting to set a very low `NativeHistogramBucketFactor` (e.g. 1.005) together with a reasonable `NativeHistogramMaxBucketNumber` (e.g. 160). In this way, each histogram always has the highest possible resolution that is affordable within the given bucket count “budget”. (This is the default strategy used by the OTel exponential histogram. It starts with an even higher schema (20), which is currently not even available in Prometheus native histograms.) However, this strategy is generally *not* recommended for the Prometheus use case. The resolution will be reduced quite often after creation and after each reset as observations come in. This creates churn both in the instrumented program as well as in the TSDB, which is particularly problematic for the latter. All of this effort is mostly in vain because the typical queries involving histograms require many histograms to get merged, during which the lowest common resolution is used so that the user ends up with a lower resolution anyway. The TSDB can be protected against the churn by limiting the resolution upon ingestion (see where a reasonable resolution cannot be assumed at instrumentation time, and the scraper should have the flexibility to pick the desired resolution at scrape time.

Partitioning by labels

While partitioning of a classic histogram with many buckets by labels has to be done judiciously, the situation is more relaxed with native histograms. Partitioning a native histograms still creates a multiplicity of individual histograms. However, the resulting partitioned histograms will often populate fewer buckets each than the original unpartitioned histogram. (For example, if a histogram tracking the duration of HTTP requests is partitioned by HTTP status code, the individual histogram tracking requests responded by status code 404 might have a very sharp bucket distribution around the typical duration it takes to identify an unknown path, populating only a few buckets.) The total number of populated buckets for all partitioned histograms will still go up, but by a smaller factor than the number of partitioned histograms. (For example, if adding labels to an already quite heavy classic histogram results in 100 labeled histograms, the total cost will go up by a factor of 100. In case of a native histogram, the cost for the single histogram might already be lower if the classic histogram featured a high resolution. After partitioning, the total number of populated buckets in the labeled native histograms will be signifcantly smaller than 100 times the number of buckets in the original native histogram.)

NHCB

Currently (2024-11-03), instrumentation libraries offer no way to directly configure native histograms with custom bucket boundaries (NHCBs). The use case for NHCBs is to allow native-histogram enabled scrapers to convert classic histograms to NHCBs upon ingestion (see next section). However, there are valid use cases where custom buckets are desirable directly during instrumentation. In those cases, the current approach is to instrument with a classic histogram and configure the scraper to convert it to an NHCB upon ingestion. However, a more direct treatment of NHCBs in instrumentation libraries might happen in the future.

Scrape configuration

To enable the Prometheus server to scrape native histograms, set `scrape_native_histograms: true` in individual scrape configs, or in the global settings.

Fine-tuning content negotiation

It is possible to fine-tune the scrape protocol negotiation globally or per scrape config via the `scrape_protocols` config setting. It is a list defining the content negotiation priorities. Its value depends on what feature flags are enabled (for example `--enable-feature=created-timestamp-zero-ingestion`), what value the user sets in it directly and lastly whether `scrape_native_histograms` is enabled.

If `scrape_native_histograms` is enabled and `scrape_protocols` is not set by a feature flag or the user globally or per scrape config, then its effective value for a scrape config is changed to `[ PrometheusProto, OpenMetricsText1.0.0,OpenMetricsText0.0.1, PrometheusText0.0.4 ]` to enable scraping native histograms.

The `scrape_protocols` setting can be used to configure protobuf scrapes without ingesting native histograms or enforce a non-protobuf format for certain targets even with `scrape_native_histograms` enabled. As long as the classic Prometheus protobuf format (`PrometheusProto` in the configured list) is the only format supporting native histograms, both `scrape_native_histograms` and negotiation of protobuf is required to actually ingest native histograms.

> **NOTE**: Switching the used exposition format between text-based and protobuf-based has some non-obvious implications. Most importantly, certain implementation details result in the counter-intuitive effect that scraping with a text-based format is generally much less resource demanding than scraping with a protobuf-based format (see tracking issue for details). Even more subtle is the effect on the formatting of label values for `quantile` labels (used in summaries) and `le` labels (used in classic histograms). This problem only affects v2 of the Prometheus server (v3 has consistent formatting under all circumstances) and is not directly related to native histograms, but might show up in the same context because enabling native histograms requires the protobuf exposition format. See details in the documentation for the `native-histograms` feature flag for v2.55.

Limiting bucket count and resolution

program might be deliberately instrumented with high-resolution histograms to give different scrapers the option to reduce the resolution as they see fit.

The Prometheus scrape config offers two settings to address this need:

1. The `native_histogram_bucket_limit` sets an upper inclusive limit for the number of buckets in an individual histogram. If the limit is exceeded, the resolution of a histogram with a standard schema is repeatedly reduced (by doubling the width of the buckets, i.e. decreasing the schema) until the limit is reached. In case an NHCB exceeds the limit, or in the rare case that the limit cannot be satisfied even with schema -4, the scrape fails.
2. The `native_histogram_min_bucket_factor` sets a lower inclusive limit for the growth factor from bucket to bucket. This setting is only relevant for standard schemas and has no effect on NHCBs. Again, if the limit is exceeded, the resolution of the histogram is repeatedly reduced (by doubling the width of the buckets, i.e. decreasing the schema) until the limit is reached. However, once schema -4 is reached, the scrape will still succeed, even if a higher growth factor has been specified.

Both settings accept zero as a valid value, which implies “no limit”. In case of the bucket limit, this means that the number of buckets are indeed not checked at all. In the case of the bucket factor, Prometheus will still ensure that a standard schema will not exceed the capabilities of the used storage backend. Prometheus currently stores histograms with standard exponential schemas of at most 8. However, it accepts exponential schemas greater than 8 up to the reserved limit of 52 but reduces their resolution upon ingestion so that schema 8 is reached (or a lower one if required by the `native_histogram_bucket_limit` or `native_histogram_min_bucket_factor` settings).

If both settings have a non-zero values, the schema is decreased sufficiently to satisfy both limits.

Note that the bucket factor set during instrumentation is an upper limit (exposed bucket growth factor ≤ configured value), while the bucket factor set in the scrape config is a lower limit (ingested bucket growth factor ≥ configured value). The schemas resulting from certain limits are therefore slightly different. Some examples:

`native_histogram_min_bucket_factor`	**resulting max schema**
65536	-4
256	-3
16	-2

2	0
1.4	1
1.1	2
1.09	3
1.04	4
1.02	5
1.01	6
1.005	7
1.002	8

General considerations about setting the limits: `native_histogram_bucket_limit` is suitable to set a hard limit for the cost of an individual histogram. The same cannot be accomplished by `native_histogram_min_bucket_factor` because histograms can have many buckets even with a low resolution if the distribution of observations is sufficiently broad. `native_histogram_min_bucket_factor` is well suited to avoid needless overall resource costs. For example, if the use case at hand only requires a certain resolution, setting a corresponding `native_histogram_min_bucket_factor` for all histograms might free up enough resources to accept a very high bucket count on a few histograms with broad distributions of observed values. Another example is the case where some histograms have low resolution for some reason (maybe already on the instrumentation side). If aggregations regularly include those low resolution histograms, the outcome will have that same low resolution (see the PromQL details below). Storing other histograms regularly aggregated with the low resolution histograms at higher resolution might not be of much use.

Scraping both classic and native histograms

As described above, a histogram exposed by an instrumented program might contain both a classic and a native histograms, and some parts are even shared (like the count and sum of observations). This section explains which parts will be scraped by Prometheus, and how to control the behavior.

If `scrape_native_histograms` is `false` (default in v3) in the scrape config, Prometheus will completely ignore the native histogram parts during scraping. If `scrape_native_histograms` is `true` (default in v4+), Prometheus will prefer the native histogram parts over the classic

In situations like migration scenarios, it might be desired to scrape both versions, classic and native, for the same histogram, provided both versions are exposed by the instrumented program. To enable this behavior, there is a boolean setting `always_scrape_classic_histograms` in the scrape config. It defaults to false, but if set to true, both versions of each histogram will be scraped and ingested, provided there is at least one classic bucket and at least one native bucket span (which might be a no-op span). This will not cause any conflicts in the TSDB because classic histograms are ingested as a number of suffixed series, while native histograms are ingested as just one series with their unmodified name. (Example: A histogram called `rpc_latency_seconds` results in a native histogram series named `rpc_latency_seconds` and in a number of series for the classic part, namely `rpc_latency_seconds_sum`, `rpc_latency_seconds_count`, and a number of `rpc_latency_seconds_bucket` series with different `le` labels.)

Scraping classic histograms as NHCBs

The aforementioned NHCB is capable of modeling a classic histogram as a native histogram. Via the boolean scrape config option `convert_classic_histograms_to_nhcb`, Prometheus can be configured to ingest classic histograms as NHCBs.

While NHCBs support automatic reconciliation between different bucket layouts, their mergeability is still fundamentally limited. The reconciliation only retains exact matches of bucket boundaries between the involved NHCBs. This yields useful results, if most bucket boundaries match. However, abitrary changes in the bucket layout can easily create a situation where none of the boundaries match, resulting in a histogram with only one bucket (the overflow bucket).

A key advantage of NHCBs is that they are generally much less expensive to store. In particular, the incremental cost of adding additional buckets is relatively low, which allows affordable ingestion of classic histograms with many buckets.

TSDB

> **NOTE**: This section provides a high level overview of storing native histograms in the TSDB and also explains some important individual aspects that might be easy to miss. It is not meant to explain implementation details, define on-disk formats, or guide through the code base. There is a detailed documentation of the various storage formats and of course the usual generated GoDoc, with the tsdb

Prometheus

Integer histograms vs. float histograms

The TSDB stores integer histograms and float histograms differently. Generally, integer histograms are expected to compress better, so a TSDB implementation MAY store a float histogram as an integer histogram if all bucket counts and the count of observations have an integer value within the int64 range so that the conversion to an integer histogram creates a numerically precise representation of the original float histogram. (Note that the Prometheus TSDB is not utilizing this option yet.)

Encoding

Native histograms require two new chunk encodings (Go type `chunkenc.Encoding`) in the TSDB: `chunkenc.EncHistogram` (string representation `histogram`, numerical value 2) for integer histograms, and `chunkenc.EncFloatHistogram` (string representation `floathistogram`, numerical value 3) for float histograms.

Similarly, there are two new record types for the WAL and the in-memory snapshot (Go type `record.Type`): `record.HistogramSamples` (string representation `histogram_samples`, numerical value 9) for integer histograms, and `record.FloatHistogramSamples` (string representation `float_histogram_samples`, numerical value 10) for float histograms. For backwards compatibility reasons, there are two more histogram record types: `record.HistogramSamplesLegacy` (`histogram_samples_legacy`, 7) and `record.FloatHistogramSamplesLegacy` (`float_histogram_samples_legacy`, 8). They were used prior to the introduction of custom values needed for NHCB. They are supported so that reading old WALs is still possible.

Prometheus identifies time series just by their labels. Whether a sample in a series is a float (and as such a counter or a gauge) or a histogram (no matter what flavor) does not contribute to the series's identity. Therefore, a series MAY contain a mix of samples of different types and flavors. Changes of the sample type within a time series are expected to be very rare in practice. They usually happen after changes in the instrumentation of a target (in the rare case that the same metric name is used for e.g. a gauge float prior to the change and a counter histogram after the change) or after a change of a recording rule (e.g. where the old version of a rule created a gauge float and the new version of the rule now creates a gauge histogram while retaining its name). Frequent changes of the sample type are usually the consequence of a misconfiguration (e.g. two different recording rules creating different chunk, it closes that chunk and starts a new one with the appropriate encoding. (A time series that switches sample types back and forth for each sample will lead to a new chunk for each sample, which is indeed very inefficient.)

Histogram chunks use a number of custom encodings for numerical values, in order to reduce the data size by encoding common values in fewer bits than less common values. The details of each custom encoding are described in the low level chunk format documentation (and ultimately in the code linked from there). The following three encodings are used for a number of different fields and are therefore named here for later reference:

- *varbit-int* is a variable bitwidth encoding for signed integers. It uses between 1 bit and 9 bytes. Numbers closer to zero need fewer bits. This is similar to the timestamp encoding in chunks for float samples, but with a different bucketing of the various bit lengths, optimized for the value distribution commonly encountered in native histograms.
- *varbit-uint* is a similar encoding, but for unsigned integers.
- *varbit-xor* is a variable bitwidth encoding for a sequence of floats. It is based on XOR'ing the current and the previous float value in the sequence. It uses between 1 bit and 77 bits per float. This is exactly the same encoding the TSDB already uses for float samples.

Histogram chunks start as usual with the number of samples in the chunk (as a uint16), followed by one byte describing if the histogram is a gauge histogram or a counter histogram and providing counter reset information for the latter. See the corresponding section below for details. This is followed by the so called chunk layout, which contains the following information, *shared by all histograms in the chunk*:

- The threshold of the zero bucket, using a custom encoding that encodes common values (zero or certain powers of two) in just one byte, but requires 9 bytes for arbitrary values.
- The schema, encoded as varbit-int.
- The positive spans, encoded as the number of spans (varbit-uint), followed by the length (varbit-uint) and the offset (varbit-int) of each span in a repeated sequence.
- The negative spans in the same way.
- Only for schema -53 (NHCB) the custom values, encoded as the number of custom values (varbit-uint), followed by the custom values in a repeated sequence, using a custom encoding.

The chunk layout is followed by a repeated sequence of sample data. The sample data is different for integer histograms and float histograms. For an integer histogram, the data of each sample contains the following:

conventional float chunks, just with a different bit bucketing for the varbit-int encoding).

- The count of observations, encoded as varbit-uint for the 1st sample and as varbit-int for any further samples, using the same “delta of deltas” approach as for timestamps.
- The zero bucket population, encoded as varbit-uint for the 1st sample and as varbit-int for any further samples, using the same “delta of deltas” approach as for timestamps.
- The sum of observations, encoded as a float64 for the 1st sample and as varbit-xor for any further samples (XOR'ing between the current and previous sample).
- The bucket populations of the positive buckets, each as a delta to the previous bucket (or as the absolute population in the 1st bucket), encoded as varbit-int, using the same “delta of deltas” approach as for timestamps. (In other words, the “double delta” encoding is applied to values that are already deltas on their own, which is the reason why this is sometimes called “triple delta“ encoding.)
- The bucket populations of the negative buckets in the same way.

The sample data of a float histogram has the following differences:

- The count of observations and the zero bucket populations are floats now and therefore encoded in the same way as the sum of observations (float64 in the 1st sample, varbit-xor for any further samples).
- The bucket population are not only floats now, but also absolute population counts rather than deltas between buckets. In the 1st sample, all bucket populations are represented as plain float64's, while they are encoded as varbit-xor for all further samples, XOR'ing corresponding buckets from the current and the previous sample.

The following events trigger cutting a new chunk (for the reasons described in parentheses):

- A change of sample type between integer histogram and float histogram (because both require different chunk encodings altogether).
- A change of sample type between gauge histogram and counter histogram (because the leading byte has to denote the different type).
- A counter reset for a counter histogram (to be stored in the leading byte as counter reset information, see details below).
- A schema change (which means we need a new chunk layout, and a chunk can only have one chunk layout).
- A change of the zero threshold (which changes the chunk layout, see above).
- A change of the custom values (which changes the chunk layout, see above).
- A staleness marker is followed by a regular sample (which does not strictly require a new chunk, but it can be assumed that most histograms will change so much when they go away and come back that cutting a new chunk is the best option).

adding (explicitly represented) unpopulated buckets as needed so that all histograms in a chunk share the same span structure. This is straightforward if a bucket disappears, because the missing bucket is simply added to the new histogram as an unpopulated bucket while the histogram is appended to the chunk. However, disappearance of a formerly populated bucket constitutes a counter reset (see below), so this case can only happen for gauge histograms (which do not feature counter resets). The far more common case is that buckets exist in a newly appended histogram that did not exist in the previously appended histograms. In this case, these buckets have to be added as explicitly unpopulated buckets to all previously appended histograms. This requires a complete re-encoding of the entire chunk. (There is some optimization potential in only re-encoding the affected parts. Implementing this would be quite complicated. So far, the performance impact of the full re-encoding did not stick out as problematic.)

Staleness markers

> **NOTE**: To understand the following section, it is important to recall how staleness markers work in the TSDB. Staleness markers in float series are represented by one specific bit pattern among the many that can be used to represent the `NaN` value. This very specific float value is called "special stale `NaN` value" in the following section. It is (almost certainly) never returned by the usual arithmetic float operations and as such different from a "naturally occurring" `NaN` value, including those discussed in Special cases of observed values. In fact, the special stale `NaN` value is never returned directly when querying the TSDB, but it is handled internally before it reaches the caller.

To mark staleness in histogram series, the usual special stale `NaN` value could be used. However, this would require cutting a new chunk, just for the purpose of marking the series as stale, because a float value following a histogram value has to be stored in a different chunk (see above). Therefore, there is also a histogram version of a stale marker where the field for the sum of observations is set to the special stale `NaN` value. In this case, all other fields are ignored, which enables setting them to values suitable for efficient storage (as the histogram version of a stale marker is essentially just a storage optimization). This works for both float and integer histograms (as the sum field is a float value even in an integer histogram), and the appropriate version can be used to avoid cutting a new chunk. All version of a stale marker (float, integer histogram, float histogram) MUST be treated as equivalent by the TSDB.

for histogram chunks, too. However, individual histograms can become very large if they have many buckets, so blindly enforcing the size limit could lead to chunks with very few histograms. (In the most extreme case, a single histogram could even take more than 1024 bytes so that the size limit could not be enforced at all.) With very few histograms per chunk, the compression ratio becomes worse. Therefore, a minimum number of 10 histograms per chunks has to be reached before the size limit of 1024 bytes kicks in. This implies that histogram chunks can be much larger than 1024 bytes.

Requiring a minimum of 10 histograms per chunk is an initial, very simplistic approach, which might be improved in the future to find a better trade-off between chunk size and compression ratio.

Counter reset considerations

Generally, Prometheus considers a counter to have reset whenever its value drops from one sample to the next (but see also the next section about the created timestamp). The situation is more complex when detecting a counter reset between two histogram samples.

First of all, gauge histograms and counter histograms are explicitly different (whereas Prometheus generally treats all float samples equally after ingestion, no matter if they were ingested as a gauge or a counter metric). Counter resets do not apply to gauge histograms.

If a gauge histogram is followed by a counter histogram in a time series, a counter reset is assumed to have happened, because a change from gauge to counter is considered equivalent to the gauge being deleted and the counter being newly created from zero.

The most common case is a counter histogram being followed by another counter histogram. In this case, a possible counter reset is detected by the following procedure:

If the two histograms are both using a standard schema, but differ in schema or in the zero bucket width, these changes could be part of a compatible resolution reduction (which happens regularly to reduce the bucket count of a histogram). Both of the following is true for a compatible resolution reduction:

- If the schema has changed, its number has decreased from one standard exponential schema to another standard schema.
- If the zero bucket width has changed, any populated regular bucket in the first histogram is either completely included in the zero bucket of the second histogram or not at all (i.e. no partial overlap of old regular buckets with the new zero bucket).

If both conditions are met, the first histogram has to be converted so that its schema and zero bucket width matches those of the second histogram. This happens in the same way as previously described: Neighboring buckets are merged to reduce the schema, and regular buckets are merged with the zero bucket to increase the width of the zero bucket.

If both histograms are NHCBs (schema -53), any difference in their custom values is reconciled as described below.

At this point in the procedure, both histograms have the same schema and zero bucket width, either because this was the case from the beginning, or because one of the histograms was converted accordingly. (Note that NHCBs do not use the zero bucket. Their zero bucket widths and population counts are considered equal for the sake of this procedure.) In this situation, any of the following constitutes a counter reset:

- A drop in the count of observations (but notably *not* a drop in the sum of observations).
- A drop in the population count of any bucket, including the zero bucket. This includes the case where a populated bucket disappears, because a non-represented bucket is equivalent to a bucket with a population of zero.

If none of the above is the case, there is no counter reset.

As this whole procedure is relatively involved, the counter reset detection preferably happens once during ingestion, with the result being persisted for later use. Counter reset detection during ingestion has to happen anyway because a counter reset is one of the triggers to cut a new chunk.

Cutting a new chunk after a counter reset aims to improve the compression ratio. A counter reset sets all bucket populations to zero, so there are fewer buckets to represent. A chunk, however, has to represent the superset of all buckets of all histograms in the chunk, so cutting a new chunk enables a simpler set of buckets for the new chunk.

This in turn implies that there will never be a counter reset after the first sample in a chunk. Therefore, the only counter reset information that has to be persisted is that of the 1st histogram in a chunk. This happens in the so-called *histogram flags*, a single byte stored directly after the the number of samples in the chunk. This byte is currently only used for the counter reset information, but it may be used for other flags in the future. The counter reset information uses the first two bits. The four possible bit patterns are represented as Go constants of type `CounterResetHeader` in the `chunkenc` package. Their names and meanings are the following:

  of the previous chunk and the 1st histogram of this chunk. (It is likely that the counter reset was actually the reason why the new chunk was cut.)
- `NotCounterReset` (bit pattern `01`): No counter reset happened between the last histogram of the previous chunk and the 1st histogram of this chunk. (This commonly happens if a new chunk is cut because the previous chunk hit the size limit.)
- `UnknownCounterReset` (bit pattern `00`): It is unknown if there was a counter reset between the last histogram of the previous chunk and the 1st histogram of this chunk.

`UnknownCounterReset` is always a safe choice. It does not prevent counter reset detection, but merely requires that the counter reset detection procedure has to be performed (again) whenever counter reset information is needed.

The counter reset information is propagated to the caller when querying the TSDB (in the Go code as a field of type `CounterResetHint` in the Go types `Histogram` and `FloatHistogram`, using enumerated constants with the same names as the bit pattern constants above).

For gauge histogram, the `CounterResetHint` is always `GaugeType`. Any other `CounterResetHint` value implies that the histogram in question is a counter histogram. In this way, queriers (including the PromQL engine, see below) obtain the information if a histogram is a gauge or a counter (which is notably different from float samples).

As long as counter histograms are returned in order from a single chunk, the `CounterResetHint` for the 2nd and following histograms in a chunk is set to `NotCounterReset`. (Overlapping blocks and out-of-order ingestion may lead to histogram sequences coming from multiple chunks, which requires special treatment, see below.)

When returning the 1st histogram from a counter histogram chunk, the `CounterResetHint` MUST be set to `UnknownCounterReset` *unless* the TSDB implementation can ensure that the previously returned histogram was indeed the same histogram that was used as the preceding histogram to detect the counter reset at ingestion time. Only in the latter case, the counter reset information from the chunk MAY be used directly as the `CounterResetHint` of the returned histogram.

This precaution is needed because there are various ways how chunks might get removed or inserted (e.g. deletion via tombstones or adding blocks for backfilling). A counter reset, while attributed to one sample, is in fact happening *between* the marked sample and the preceding sample. Removing the preceding sample or inserting another sample in between the two samples invalidates the previously performed counter reset detection.

> Prometheus currently returns `UnknownCounterReset` for *all* 1st histograms from a counter histogram chunk. See tracking issue for efforts to change that.

As already implied above, the querier MUST perform the counter reset detection procedure (again), if the `CounterResetHint` is set to `UnknownCounterReset`.

Special caution has to be applied when processing overlapping blocks or out-of-order samples (for querying or during compaction). Both overdetection and underdetection of counter resets may happen in these cases, as illustrated by the following examples:

- *Example for underdetection:* One chunk contains samples ABC, without counter resets. Another chunk contains samples DEF, again without counter resets. The chunks are overlapping and refer to the same series. When querying them together, the temporal order of samples turns out to be ADBECF. There might now very well be a counter reset between some or even all of those samples. This is in fact likely if the two samples are actually from unrelated series and got merged into the same series by accident. However, even accidental merges like this have to be handled correctly by the TSDB. If the overlapping chunks are compacted into a new chunk, a new counter reset detection has to happen, catching the new counter resets. If querying the overlapping chunks directly (without prior compaction), a `CounterResetHint` of `UnknownCounterReset` has to be set for each sample that comes from a different chunk than the previously returned sample, which mandates a counter reset detection by the querier (utilizing the safe fallback described above).
- *Example for overdetection:* There is a sequence of samples ABCD with a counter reset happening between B and C. However, the initial ingestion missed B and C so that only A and D were ingested, with a counter reset detected between A and D. Later, B and C are ingested (via out-of-order ingestion or as separate chunks later added to the TSDB as a separate block), with a counter reset detected between B and C. In this case, each sample goes into its own chunk, so when assembling all the chunks, they do not even overlap. However, when returning the counter reset hints according to the rules above, both C and D will be returned to the querier with a `CounterResetHint` of `CounterReset`, although there is now no counter reset between C and D. Similar to the situation in the previous example, a new counter reset detection has to be performed between A and B, and another one between C and D. Or both B and D have to be returned with a `CounterResetHint` of `UnknownCounterReset`.

In summary, whenever the TSDB cannot safely establish that a counter reset detection between two samples has happened upon ingestion, it either has to perform another

Note that there is the possibility of counter resets that are not detected by the procedure described above, namely if the counts in the reset histogram have increased quickly enough so that the 1st sample after the counter reset has no counts that have decreased compared to the last sample prior to the counter reset. (This is also a problem for float counters, where it is actually more likely to happen.) With the mechanisms explained above, it is possible to store a counter reset even in this case, provided that the counter reset was detected by other means. However, due to the complications caused by insertion and removal of chunks, out-of-order samples, and overlapping blocks (as explained above), this information MAY get lost if a second round of counter reset detection is required. (TODO: Currently, this information is reliably lost, see TODO above.) A better way to safely mark a counter reset is via created timestamps (see next section).

Created timestamp handling

OpenMetrics introduced so-called created timestamps for counters, summaries, and classic counter histograms. (The term is probably short for “created-at timstamp”. The more appropriate term might have been “creation timestamp” or “reset timestamp”, but the term “created timestamp” is firmly established by now.)

The created timestamp provides the most recent time the metric was created or reset. A design doc describes how Prometheus handles created timestamps.

Created timestamps are also useful for native histograms. In the same way a synthetic zero sample is inserted for float counters, a zero value of a histogram sample is inserted for counter histograms. A zero value of a histogram has no populated buckets, and the sum of observations, the count of observations, and the zero bucket population are all zero. Schema, zero bucket width, custom values, and the float vs. integer flavor of the histogram SHOULD match the sample that directly follows the synthetic zero sample (to not trigger the detection of a spurious counter reset).

The counter reset information of the synthetic zero sample is always set to `CounterReset`.

Exemplars

Exemplars for native histograms are attached to the histogram sample as a whole, not to individual buckets. (See also the exposition formats section.) Therefore, it is allowed (and in fact the common case) that a single native histogram sample comes with multiple exemplars attached.

which any subset could be repeated exemplars from the last scrape. The TSDB MAY rely on the assumption that any new exemplar has a more recent timestamp than any of the previously exposed exemplars. (Remember that exemplars of native histograms MUST have a timestamp.) Duplicate detection is then possible in an efficient way:

1. The exemplars of a newly ingested native histogram are sorted by the following fields: first timestamp, then value, then labels.
2. The exemplars are appended to the exemplar storage in the sorted order.
3. The append fails for exemplars that would be sorted before or are equal to the last successfully appended exemplar (which might be from the previous scrape for the same metric).
4. The append succeeds for exemplars that would be sorted after the last successfully appended exemplar.

Exemplars are only counted as out of order if all exemplars of an ingested histogram would be sorted before the last successfully appended exemplar. This does not detect out-of-order exemplars that are mixed with newer exemplars or with a duplicate of the last successfully appended exemplar, which is considered acceptable.

PromQL

This section describes how PromQL handles native histograms. It focuses on general concepts rather than every single detail of individual operations. For the latter, refer to the PromQL documentation about operators and functions.

Annotations

The introduction of native histograms creates certain situations where a PromQL expression returns unexpected results, most commonly the case where some or all elements in the output vector are unexpectedly missing. To help users detect and understand those situations, operations acting on native histograms often use annotations. Annotations can have warn and info level and describe possible issues encountered during the evaluation. Warn level is used to mark situations that are most likely an actual problem the user has to act on. Info level is used for situations that might also be deliberate, but are still unusual enough to flag them.

Integer histograms vs. float histograms

Compatibility between histograms

When an operator or function acts on two or more native histograms, the histograms involved need to have the same schema, the same zero bucket width, and (if applicable) the same custom values. Within certain limits, histograms can be converted on the fly to meet these compatibility criteria:

- NHCBs (schema -53) are only compatible with each other. Different custom values need to be reconciled by conversion in the following way:
    - Identify the custom values that are shared by all of the original NHCBs. These are the new reconciled custom values.
    - Convert each original NHCB to the new custom values by merging its buckets into the unified bucket set described by the new custom values.
    - Note that it is easily possible that the original NHCBs do not share any custom values. In this case, the new bucket set will only consist of the overflow bucket, taking all observations from all of the original buckets.
    - Any query requiring reconciliation of custom values is flagged with an info-level annotation.
- Histograms with standard schemas can always be converted to the smallest (i.e. lowest resolution) common schema by decreasing the resolution of the histograms with greater schemas (i.e. higher resolution). This happens in the usual way by merging neighboring buckets into the larger buckets of the smaller schema.
- Different zero bucket widths are handled by expanding the smaller zero buckets, merging any populated regular bucket into the expanded zero bucket as appropriate. If the greatest common width happens to end up in the middle of any populated bucket, it is further expanded to coincide with the bucket boundary of that bucket. (See more details in the zero bucket section above.)

If incompatibility prevents an operation, a warn-level annotation is added to the result.

Counter resets

Counter resets are defined as described above. Counter reset hints returned from the TSDB MAY be taken into account to avoid explicit counter reset detection and to correctly process counter resets that are not detectable by the usual procedure. (This implies that these counter resets are only taken into account on a best effort basis. However, the same is true for the TSDB itself, see above.) A notable difference to the counter reset handling for classic histograms and summaries is that a decrease of the sum of observations does *not* constitute

Note that the counter reset hints of counter histograms returned by sub-queries MUST NOT be taken into account to avoid explicit counter reset detection, unless the PromQL engine can safely detect that consecutive counter histograms returned from the sub-query are also consecutive in the TSDB.

Gauge histograms vs. counter histograms

Via the counter reset hint returned from the TSDB, PromQL is aware if a native histogram is a gauge or a counter histogram. To mirror PromQL's treatment of float samples (where it cannot reliably distinguish between float counters and gauges), functions that act on counters will still process gauge histograms, and vice versa, but a warn-level annotation is returned with the result. Note that explicit counter reset detection has to be performed on a gauge histogram in that case, treating it as if it were a counter histogram.

Interpolation within a bucket

When estimating quantiles or fractions, PromQL has to apply interpolation within a bucket. In classic histograms, this interpolation happens in a linear fashion. It is based on the assumption that observations are equally distributed within the bucket. In reality, this assumption might be far off. (For example, an API endpoint might respond to almost all request with a latency of 110ms. The median latency and maybe even the 90th percentile latency would then be close to 110ms. If a classic histogram has bucket boundaries at 100ms and 200ms, it would see most observations in that range and estimate the median at 150ms and the 90th percentile at 190ms.) The worst case is an estimation at one end of a bucket where the actual value is at the other end of the bucket. Therefore, the maximum possible error is the whole width of a bucket. Not doing any interpolation and using some fixed midpoint within a bucket (for example the arithmetic mean or even the harmonic mean) would minimize the maximum possible error (which would then be half of the bucket width in case of the arithmetic mean), but in practice, the linear interpolation yields an error that is lower on average. Since the interpolation has worked well over many years of classic histogram usage, interpolation is also applied for native histograms.

For NHCBs, PromQL applies the same interpolation method as for classic histograms to keep results consistent. (The main use case for NHCBs is a drop-in replacement for classic histograms.) However, for standard exponential schemas, linear interpolation can be seen as a misfit. While exponential schemas primarily intend to minimize the relative error of quantile estimations, they also benefit from a balanced usage of buckets, at least over certain ranges of observed values. The basic assumption is that for most practically occurring

(i.e. doubling the resolution) will on average see similar populations in both new buckets. A more detailed explanation can be found in the PR implementing the interpolation method
.

A special case is interpolation within the zero bucket. The zero bucket breaks the exponential bucketing schema. Therefore, linear interpolation is applied within the zero bucket. Furthermore, if all populated regular buckets of a histogram are positive, it is assumed that all observations in the zero bucket are also positive, i.e. the interpolation is done between zero and the upper bound of the zero bucket. In the case of a histogram where all populated regular buckets are negative, the situation is mirrored, i.e. the interpolation within the zero bucket is done between the lower bond of the zero bucket and zero.

Mixed series

As already discussed above, neither the sample type nor the flavor of a native histogram is part of the identity of a series. Therefore, one and the same series might contain a mix of different sample types and flavors.

A mix of counter histograms and gauge histograms doesn't prevent any PromQL operation, but a warn-level annotation is returned with the result if some of the input samples have an inappropriate flavor (see above).

A mix of float samples and histogram samples is more problematic. Many functions that operate on range vectors will remove elements from the result where the input elements contain a mix of floats and histograms. If this happens, a warn-level annotation is added to the result. Concrete examples can be found below.

Unary minus and negative histograms

The unary minus can be used on native histograms. It returns a histogram where all bucket populations and the count and the sum of observations have their sign inverted. The counter reset hint is set to `GaugeType` in any case. Everything else stays the same. Enforcing `GaugeType` is needed because explicit counter reset detection will be thrown off by the inverted sign.

Generally, histograms with negative bucket populations or a negative count of observations do not really make sense on their own and are only supposed to act as intermediate results inside other expressions. They are always considered gauge histograms within PromQL. They cannot be persisted as a result of a recording rule. (A rule evaluating to a negative histogram

Prometheus

Binary operators

Most binary operators do not work between two histograms or between a histogram and a float or between a histogram and a scalar. If an operator processes such an impossible combination, the corresponding element is removed from the output vector and an info-level annotation is added to the result. (This situation is somewhat similar to label matching, where the sample type plays a role similar to a label. Therefore, such a mismatch might be known and deliberate, which is the reason why the level of the annotation is only info.)

The following describes all the operations that actually *do* work.

Addition (`+`) and subtraction (`-`) work between two compatible histograms. These operators add or subtract all matching bucket populations and the count and the sum of observations. Missing buckets are assumed to be empty and treated accordingly. Generally, both operands should be gauges. Adding and subtracting counter histograms requires caution, but PromQL allows it. Adding a gauge histogram and a counter histogram results in a gauge histogram. Adding two counter histograms results in a counter histogram. If the two operands share the same counter reset hint, the resulting counter histogram retains that counter reset hint. Otherwise, the resulting counter reset hint is set to `UnknownCounterReset` . The result of a subtraction is always marked as a gauge histogram because it might result in negative histograms, see notes above. Adding or subtracting two counter histograms with directly contradicting counter reset hints (i.e. `CounterReset` and `NotCounterReset`) triggers a warn-level annotation. (TODO: As described above, the TSDB currently does not return `NotCounterReset` , so this annotation will only happen under specific circumstances involving the `HistogramStatsIterator` , which includes additional counter reset tracking. See tracking issue.)

Multiplication (`*`) works between a float sample or a scalar on the one side and a histogram on the other side, in any order. It multiplies all bucket populations and the count and the sum of observations by the float (sample or scalar). This will lead to "scaled" and sometimes even negative histograms, which is usually only useful as intermediate results inside other expressions (see also notes above). Multiplication works for both counter histograms and gauge histograms, and their flavor is left unchanged by the operation.

Division (`/`) works between a histogram on the left hand side and a float sample or a scalar on the right hand side. It is equivalent to multiplication with the inverse of the float (sample or scalar). Division by zero results in a histogram with no regular buckets and the zero bucket population and the count and sum of observations all set to `+Inf` , `-Inf` , or `NaN` , Equality (`==`) and inequality (`!=`) work between two histograms, both in their filtering version as well as with the `bool` modifier. They compare the schema, the custom values, the zero threshold, all bucket populations, and the sum and count of observations. Whether the histograms have counter or gauge flavor is irrelevant for the comparison. (A counter histogram could be equal to a gauge histogram.)

The logical/set binary operators (`and` , `or` , `unless`) work as expected even if histogram samples are involved. They only check for the existence of a vector element and don't change their behavior depending on the sample type or flavor of an element (float or histogram, counter or gauge).

The “trim” operators `>/` and `</` were introduced specifically for native histograms. They only work for a histogram on the left hand side and a float sample or a scalar on the right hand side. (They do not work for float samples or scalars on *both* sides. An info-level annotation is returned in this case.) These operators remove observations from the histogram that are greater or smaller than the float value on the right side, respectively, and return the resulting histogram. The removal is only precise if the threshold coincides with a bucket boundary. Otherwise, interpolation within the affected buckets has to be used, as described above. The counter vs. gauge flavor of the histogram is preserved. (TODO: These operators are not yet implemented and might also change in detail, see tracking issue.)

Aggregation operators

The following aggregation operators work in the same way with float and histogram samples (for the reason stated in parentheses):

- `group` (The result of this aggregation does not depend on the sample values.)
- `count` (The result of this aggregation does not depend on the sample values.)
- `count_values` (The text representation as produced by the Go `FloatHistogram.String` method is used as the value of histograms.)
- `limitk` (The sampled elements are returned unchanged.)
- `limit_ratio` (The sampled elements are returned unchanged.)

The `sum` aggregation operator works with native histograms by summing up the histogram to be aggregated in the same way as described for the `+` operator above. The `avg` aggregation operator works in the same way, but divides the sum by the number of aggregated histograms (in the same way as described for the `/` operator above).

Both `sum` and `avg` remove elements from the output vector that would require the aggregation of float samples with histogram samples. Such a removal is flagged by a warn-aggregate counter histograms (and even a mix of both), but it requires caution to do so in a meaningful way. The implications for the gauge vs. counter flavor and the resulting counter reset hint are derived from those for the `+` operator above:

- If all aggregated histograms share the same counter reset hint, the result retains that same counter reset hint.
- If there is at least one gauge histogram among the aggregated histograms, the result is a gauge histogram.
- In all other cases, the counter reset hint of the result is set to `UnknownCounterReset`.
- In any case, any directly contradicting counter reset hints (i.e. `CounterReset` and `NotCounterReset`) among the aggregated histograms trigger a warn-level annotation.

All other aggregation operators do *not* work with native histograms. Histograms in the input vector are simply ignored, and an info-level annotation is added for each ignored histogram.

Functions

The following functions operate on range vectors of native histograms by applying the usual float operation individually to matching buckets (including the zero bucket) and the sum and count of observations, resulting in a new native histogram:

- `delta()` (For gauge histograms.)
- `increase()` (For counter histograms.)
- `rate()` (For counter histograms.)
- `idelta()` (For gauge histograms.)
- `irate()` (For counter histograms.)

These functions SHOULD be applied to either gauge histograms or counter histograms as noted above. However, they all work with both flavors, but if at least one histogram of an unsuitable flavor is contained in the range vector, a warn-level annotation is added to the result.

`delta()`, `increase()`, and `rate()` return no result for series that contain a mix of float samples and histogram samples within the range. `idelta()` and `irate()` return no result for series where the last two samples within the range are a mix of a float sample and a histogram sample. In either case, a warn-level annotation is added for each output element missing for these reasons.

All these functions return gauge histograms as results.

is a counter reset between the 1st and 2nd sample. In this case, the 1st sample is not included in the calculation, so an incompatible bucket layout between the 1st sample and the other samples is simply ignored silently.

To prevent extrapolation below zero, the same heuristics is applied as for float counters, but solely based on the count of observations. Therefore, individual buckets might still be extrapolated below zero in some cases. An alternative could have been to find the smallest extrapolation where neither the count nor any bucket would be extrapolated below zero. However, this does not necessarily lead to a better heuristics while inflicting a significant cost in complexity. In the common and important case where the first sample in the range is a synthetic zero sample derived from the created-at timestamp, the limited extrapolation will actually work perfectly precise, because the count and all buckets are zero at precisely the timestamp of the synthetic sample, which is also the point in time to which the extrapolation is limited. Note that classic histogram apply the heuristics independently to each bucket and the count and the sum (as they are all separate series). This is known to lead to inconsistencies. NHCBs do not reproduce this problem and work in the same way as other native histograms, which means that the result of `rate()` and `increase()` may be slightly different when comparing classic histograms and equivalent NHCBs.

`avg_over_time()` and `sum_over_time()` work with native histograms in a way that corresponds to the respective aggregation operators. In particular, if a series contains a mix of float samples and histogram samples within the range, the corresponding result is removed entirely from the output vector. Such a removal is flagged by a warn-level annotation.

The `changes()` and the `resets()` function work with native histogram samples in the same way as with float samples. They even work with a mix of float samples and histogram samples within the same series. In this case, a change from a float sample to a histogram sample and vice versa counts as a change for `changes()` and as a reset for `resets()`. A change in flavor from counter histogram to gauge histogram and vice versa does not count as a change for `changes()`. `resets()` SHOULD only be applied to counter floats and counter histograms, but the function still works with gauge histograms, applying explicit counter reset detection in this case. Furthermore, a change from counter histogram to gauge histogram and vice versa is counted as a reset.

The `histogram_quantile()` function has a very special role as it is the only function that treats a specific “magic” label specially, namely the `le` label used by classic histograms. `histogram_quantile()` also works for native histograms in a similar way, but without the special role of the `le` label. The function keeps treating float samples in the known way, while it uses the new “native” way for native histogram samples.

This is the corresponding query for a native histograms:

```
histogram_quantile(0.9, sum by (job) (rate(http_request_duration_seconds[10m])))
```

As with classic histograms, an estimation of the maximum and minimum observation in a histogram can be performed using 1 and 0, respectively, as the first parameter of `histogram_quantile`. However, native histograms with standard schemas enable much more useful results, not only because of the usually higher resolution of native histograms, but even more so because native histograms with standard schemas sustain the same resolution across the whole range of float64 numbers. With a classic histogram, the odds are that the maximum observation is in the +Inf bucket, so that the estimation simply returns the upper limit of the last bucket before the +Inf bucket. Similarly, the minimum observation will often be in the lowest bucket.

`histogram_quantile` treats observations of value `NaN` (which SHOULD NOT happen, see above) effectively as observations of `+Inf`. This follows the rationale that `NaN` is never less than any value that `histogram_quantile` returns. As long as the result falls into an existing bucket we return the result calculated as if `NaN` observations were observed as `+Inf` and also issue an info level annotation to let the user know that results are skewed due to `NaN`. This is consistent with how classic histograms usually treat `NaN` observations (which end up in the `+Inf` bucket in most implementations). If the result falls above all existing buckets, we return `NaN`. For a detailed explanation why, see `histogram_fraction` below; intuitively this case means that we don't have a number that is greater than all observations in the quantile as `NaN` is not comparable to any number. We also return an info level annotation specific to this case.

The following functions have been introduced specifically for native histograms:

- `histogram_avg()`
- `histogram_count()`
- `histogram_fraction()`
- `histogram_sum()`
- `histogram_stddev()`
- `histogram_stdvar()`

All these functions silently ignore float samples as input. Each function returns a vector of float samples.

recommended way to calculate a rate of the count or the sum of observations is to first rate the histogram and then apply `histogram_count()` or `histogram_sum()` to the result. For example, the following query calculates the rate of observations (in this case corresponding to "requests per second") from a native histogram:

```
histogram_count(rate(http_request_duration_seconds[10m]))
```

Note that the special counter reset detection for native histograms doesn't apply when using a sub-query on the result of `histogram_sum()` , i.e. negative observations may result in spurious counter resets.

`histogram_avg()` returns the arithmetic average of the observed values in a native histogram. (This is notably different from applying the `avg` aggregation operator to a number of native histograms. The latter returns an averaged histogram.)

Similarly, `histogram_stddev()` and `histogram_stdvar()` return the estimated standard deviation or standard variance, respectively, of the observations in a native histogram. For this estimation, all observations in a bucket are assumed to have the value of the mean of the bucket boundaries. For the zero bucket and for buckets with custom boundaries, the arithmetic mean is used. For standard exponential buckets, the geometric mean is used.

`histogram_fraction(lower, upper, histogram)` returns the estimated fraction of observations in `histogram` between the provided boundaries, the scalar values `lower` and `upper` . The error of the estimation depends on the resolution of the underlying native histogram and how closely the provided boundaries are aligned with the bucket boundaries in the histogram. `+Inf` and `-Inf` are valid boundary values and useful to estimate the fraction of all observations above or below a certain value. However, observations of value `NaN` are always considered to be outside of the specified boundaries (even `+Inf` and `-Inf`). Whether the provided boundaries are inclusive or exclusive is only relevant if the provided boundaries are precisely aligned with bucket boundaries in the underlying native histogram. In this case, the behavior depends on the precise definition of the schema of the histogram.

The value of `q = histogram_fraction(-Inf, x, histogram)` means that the fraction of observations less or equal to `x` is `q` . On the other hand `y = histogram_quantile(q, histogram)` means that `q` fraction of observations are less or equal to `y` . Since `histogram_quantile` calculates the approximate smallest value for `y` , it follows that `y<=x` in general. Consider the case when 90% of the observations are `NaN` . Then the maximum value of `histogram_fraction` is `0.1` since `histogram_fraction` considers `NaN` however this doesn't happen for any `y`, which is the reason we return `NaN` if the result of `histogram_quantile` would be outside all buckets.

The following functions do not interact directly with sample values and therefore work with native histogram samples in the same way as they work with float samples:

- `absent()`
- `absent_over_time()`
- `count_over_time()`
- `info()`
- `label_join()`
- `label_replace()`
- `last_over_time()`
- `present_over_time()`
- `sort_by_label()`
- `sort_by_label_desc()`
- `timestamp()`

All remaining functions not mentioned in this section do *not* work with native histograms. Histogram elements in the input vector are silently ignored. For `deriv()`, `double_exponential_smoothing()`, `predict_linear()`, and all the `<aggregation>_over_time()` functions not mentioned before, native histogram samples are removed from the input range vector. In case any series contains a mix of float samples and histogram samples within the range, the removal of histograms is flagged by an info-level annotation.

Recording rules

Recording rules MAY result in native histogram values. They are stored back into the TSDB as during normal ingestion, including whether the histogram is a gauge histogram or a counter histogram. In the latter case, a counter reset explicitly marked by the counter reset hint is also stored, while a new counter reset detection is initiated during ingestion otherwise.

TSDB implementations MAY convert the float histograms created by recording rules to integer histograms if this conversion precisely represents all the float values in the original histogram.

Alerting rules

method), which is hard to read for humans.

Testing framework

The PromQL testing framework has been extended so that both PromQL unit tests as well as rules unit tests via `promtool` can include native histograms. The histogram sample notation is complex and explained in the documentation for rules unit testing.

In the unit test framework there is an alternative `load` command called `load_with_nhcb`, which converts classic histograms to NHCBs and loads both the float series of the classic histogram as well as the NHCB series resulting from the conversion.

Not specific to native histograms, but very useful in their context, is the `expect` keyword in the unit test framework that can define expectations about the info- and warn-level annotations.

Optimizations

As usual, PromQL implementations MAY apply any optimizations they see fit as long as the behavior stays the same. Decoding native histograms can be quite expensive with the potentially many buckets. Similarly, deep-copying a histogram sample within the PromQL engine is much more expensive than copying a simple float sample. This creates a huge potential for optimization compared to a naive approach of always decoding everything and always copying everything.

Prometheus currently tries to avoid needless copies (TODO: but a proper CoW like approach still has to be implemented, as it would be much cleaner and less bug prone) and skips decoding of the buckets for special cases where only the sum and count of observations is required.

Prometheus query API

The query API documentation includes native histogram support. This section focuses on the parts relevant for native histograms and provides a bit of context not part of the API documentation.

Instant and range queries

Prometheus

The `vector` result type gets a new key `histogram` at the same level as the existing `value` key. Both these keys are mutually exclusive, i.e. each element in a `vector` has either a `value` key (for a float result) or a `histogram` key (for a histogram result). The value of the `histogram` key is structured similarly to the value of the `value` key (a two-element array), with the difference that the string representing the float sample value is replaced by a specific histogram object described below.

The `matrix` result type gets a new key `histograms` at the same level as the existing `values` key. These keys are *not* mutually exclusive. A series may contain both float values and histogram values, but for a given timestamp, there must be only one sample, either a float or a histogram. The value of the `histograms` key is structured similarly to the value of the `values` key (an array of *n* two-element arrays), with the difference that the strings representing float sample values are replaced by specific histogram objects described below.

Note that a better naming of the keys would be `float` / `histogram` and `floats` / `histograms` because both float values and histogram values are values. The current naming has historical reasons. (In the past, there was only one value type, namely floats, so calling the keys simply `value` and `values` was the obvious choice.) The intention here is to not break existing consumers that do not know about native histograms.

The histogram object mentioned above has the following structure:

```
{
  "count": "<count_of_observations>",
  "sum": "<sum_of_observations>",
  "buckets": [ [ <boundary_rule>, "<left_boundary>", "<right_boundary>", "<count_in_
}
```

`count` and `sum` directly correspond to the histogram fields of the same name. Each bucket is represented explicitly with its boundaries and count, including the zero bucket. Spans and the schema are therefore not part of the response, and the structure of the histogram object does not depend on the used schema.

The `<boundary_rule>` placeholder is an integer between 0 and 3 with the following meaning:

- 0: “open left” (left boundary is exclusive, right boundary in inclusive)
- 1: “open right” (left boundary is inclusive, right boundary in exclusive)
- 2: “open both” (both boundaries are exclusive)

and the zero bucket (with a negative left boundary and a positive right boundary) is “closed both”. For NHCBs, all buckets are “open left” (mirroring the behavior of classic histograms). Future schemas might utilize different boundary rules.

Metadata

For the `series` endpoint, series containing native histograms are included in the same way as conventional series containing only floats. The endpoint does not provide any information what sample types are included (and in fact, *any* series may contain either or both sample types). Note in particular that a histogram exposed by a target under the name `request_duration_seconds` will lead to a series called `request_duration_seconds` if it is exposed and ingested as a native histogram, but if it is exposed and ingested as a classic histogram, it will lead to a set of series called `request_duration_seconds_sum`, `request_duration_seconds_count`, and `request_duration_seconds_bucket`. If the histogram is ingested as *both* a native histogram and a classic histogram, all of the series names above will be returned by the `series` endpoint.

The target and metric metadata (endpoints `targets/metadata` and `metadata`) work a bit differently, as they are acting on the original name as exposed by the target. This means that a classic histogram called `request_duration_seconds` will be represented by these metadata endpoints only as `request_duration_seconds` (and not `request_duration_seconds_sum`, `request_duration_seconds_count`, or `request_duration_seconds_bucket`). A native histogram `request_duration_seconds` will also be represented under this name. Even in the case where `request_duration_seconds` is ingested as both a classic and a native histogram, there will be no collision as the metadata returned is actually the same (most notably the returned `type` will be `histogram`). In other words, there is currently no way of distinguishing native from classic histograms via the metadata endpoints alone. An additional look-up via the `series` endpoint is required. There are no plans to change this, as the existing metadata endpoints are anyway severely limited (no historical information, no metadata for metrics created by rules, limited ability to handle conflicting metadata between different targets). There are plans, though, to improve metadata handling in Prometheus in general. Those efforts will also take into account how to support native histograms properly. (TODO: Update as progress is made.)

Prometheus UI

This section describes the rendering of histograms by Prometheus's own UI. This MAY be used as a guideline for third party graphing frontends.

of each bar on the *x* axis is determined by the lower and upper limit of the corresponding bucket. The area of each bar is proportional to the population of the corresponding bucket (which is a core principle of rendering histograms in general).

The graphical histogram allows a choice between an exponential and a linear *x* axis. The former is the default. It is a good fit for the standard schemas. (TODO: Consider linear as a default for non-exponential schemas.) Conveniently, all regular buckets of an exponential schema have the same width on an exponential *x* axis. This means that the *y* axis can display actual bucket populations without violating the above principle that the *area* (not the height) of a bar is representative for the bucket population. The zero bucket is an exception to that. Technically, it has an infinite width. Prometheus simply renders it with the same width as the regular exponential buckets (which in turn means that the *x* axis is not strictly exponential around the zero point). (TODO: How to do the rendering for non-exponential schemas.)

With a linear *x* axis, the buckets generally have varying width. Therefore, the *y* axis displays the bucket population divided by its width. The Prometheus UI does not render values on the *y* axis as they would be hard to interpret for humans anyway. The population can still be inspected in the text representation.

In the *Graph* view, Prometheus displays a heatmap (TODO: not yet, see below), which could be seen as a series of histograms over time, rotated by 90 degrees and encoding the bucket population as a color rather than the height of a bar. The typical query to render a counter-like histogram as a heatmap would be a `rate` query. A heatmap is an extremely powerful representation that allows humans to easily spot characteristics of distributions as they change over time.

> **TODO**: Heatmaps are not implemented yet. Instead, the UI plots just the sum of observations as a conventional graph. See tracking issue. The same issue also discusses how to deal with the rendering of range vectors in the *Table* view.

Template expansion

Native histograms work in template expansion. They are rendered in a text representation inspired by the mathematical notation of open and closed intervals. (This is generated by the `FloatHistogram.String` method in Go.) As native histograms can have a lot of buckets and bucket boundaries tend to have boundaries with a lot of decimal places, the representation isn't necessarily very readable. Use native histograms in template expansion judiciously.

Remote-write & remote-read

The protobuf specs for remote-write & remote-read were extended for native histograms. Receivers not capable of processing native histograms will simply ignore the newly added fields. Nevertheless, Prometheus has to be configured to send native histograms via remote-write (by setting the `send_native_histograms` remote-write config setting to true).

In remote-write v2, native histograms are a stable feature.

It might appear tempting to convert classic histograms to NHCBs while sending or receiving them. However, this does not overcome the known consistency problems classic histograms suffer from when transmitted via remote-write. Instead, classic histograms SHOULD be converted to NHCBs during scraping. Similarly, explicit OTel histograms SHOULD be converted to NHCBs during OTLP ingestion already.

> **TODO**: A remaining possible problem with remote-write is what to do if multiple exemplars originally ingested for the same native histogram are sent in different remote-write requests.

Federation

Federation of native histograms works as expected, provided the federation scrape uses the protobuf format. A federation via OpenMetrics text format will be possible, at least in principle, once native histograms are supported in that format, but federation via protobuf is preferred for efficiency reasons anyway.

> **TODO**: Update once OM supports NH.

NHCBs are rendered as classic float histograms when exposed via the federation endpoint. Scrapers have the option of converting them back to NHCBs or ingest them as classic histograms. The latter could lead to naming collisions, though. Note that OpenMetrics v1

OTLP

The OTLP receiver built into Prometheus converts incoming OTel exponential histograms to Prometheus native histograms utilizing the compatibility described above. The resolution of a histogram using a schema ("scale" in OTel lingo) greater than 8 will be reduced to match schema 8. (In the unlikely case that a schema smaller than -4 is used, the ingestion will fail.)

Explicit OTel histograms are the equivalent of Prometheus's classic histograms. Prometheus therefore converts them to classic histograms by default, but optionally offers direct conversion to NHCBs.

Pushgateway

Native histogram support has been gradually added to the Pushgateway. Full support was reached in v1.9. The Pushgateway always has been based on the classic protobuf format as its internal data model, which made the necessary changes easy (mostly UI concerns). Combined histograms (with classic and native buckets) can be pushed and will be exposed as such via the `/metrics` endpoint. (However, the query API, which can be used to query the pushed metrics as JSON, will only be able to return one kind of buckets and will prefer native buckets if present.)

`promtool`

This section describes `promtool` commands added or changed to support native histograms. Commands not mentioned explicitly do not directly interact with native histograms and require no changes.

The `promtool query ...` commands work with native histograms. See the query API documentation to learn about the output format. A new command `promtool query analyze` was specifically added to analyze classic and native histogram usage patterns returned by the query API.

The rules unit testing via `promtool test rules` works with native histograms, using the format described above.

`promtool tsdb analyze` and `promtool tsdb list` work normally with native histograms. The `--extended` output of the former has specific sections for histogram chunks.

`promtool tsdb create-blocks-from rules` works with rules that emit native histograms.

The `promtool promql ...` commands support all the PromQL features added for native histograms.

While `promtool tsdb bench write` could in principle include native histograms, such a support is not planned at the moment.

The following commands depend on the OpenMetrics text format and therefore cannot support native histograms as long as there is no native histogram support in OpenMetrics:

- `promtool check metrics`
- `promtool push metrics`
- `promtool tsdb dump-openmetrics`
- `promtool tsdb create-blocks-from openmetrics`

TODO: Update as progress is made. See tracking issue.

`prom2json`

`prom2json` is a small tool that scrapes a Prometheus `/metrics` endpoint, converts the metrics to a bespoke JSON format, which it dumps to stdout. This is convenient for further processing with tools handling JSON, for example `jq`.

`prom2json` v1.4 added support for native histograms. If a histogram in the exposition contains at least one bucket span, `prom2json` will replace the usual classic bucket in the JSON output with the buckets of the native histogram, following a format inspired by the Prometheus query API.

Migration considerations

When migrating from classic to native histograms, there are three important sources of issues to consider:

1. Querying native histograms works differently from querying classic histograms. In most cases, the changes are minimal and straightforward, but there are tricky edge cases, which make it hard to perform a reliable auto-conversion.

transition point will inevitably be incomplete (i.e. a range vector selecting classic histograms will only contain data points in the earlier part of the range, and a range vector selecting native histograms will only contain data points in the later part of the range).

3. A classic histogram might be tailored to have bucket boundaries precisely at the points of interest. Native histograms with a standard schema can have a high resolution, but do not allow to set bucket boundaries at arbitrary values. In those cases, the user experience with native histograms might actually be worse.

To address (3), it is of course possible to not migrate the classic histogram in question and leave things as they are. Another option is to leave the instrumentation the same but convert classic histograms to NHCBs upon ingestion. This leverages the increased storage performance of native histograms, but still requires to address (1) and (2) in the same way as for a full migration to native histograms (see next paragraphs).

The conservative way of addressing (1) and (2) is to allow a long transition period, which comes at the cost of collecting and storing classic and native histograms in parallel for a while.

The first step is to update the instrumentation to expose classic and native histograms in parallel. (This step can be skipped if the plan is to stick with classic histogram in the instrumentation and simply convert them to NHCBs during scraping.)

Then configure Prometheus to scrape both classic and native histograms, see section about scraping both classic and native histograms above. (If needed, also activate the conversion of classic histograms to NHCB.)

The existing queries involving classic histograms will continue to work, but from now on, users can start working with native histograms and start to change queries in dashboards, alerts, recording rules,… As already mentioned above, it is important to pay attention to queries with longer range vectors like `histogram_quantile(0.9, rate(rpc_duration_seconds[1d]))` . This query calculates the 90th percentile latency over the last day. Hoewever, if native histograms haven't been collected for at least one day, the query will only cover that shorter period. Thus, the query should only be used once native histograms have been collected for at least 1d. For a dashboard that displays the daily 90th percentile latency over the last month, it is tempting to craft a query that correctly switches from classic to native histograms at the right moment. While that is in principle possible, it is tricky. If feasible, the transition period during which classic and native histograms are collected in parallel, can be quite long to minimize the necessity to implement tricky switch-overs. For example, once classic and native histograms have been collected in parallel for a

Once there is confidence that all queries have been migrated correctly, configure Prometheus to only scrape native histograms (which is the “normal” setting). (It is also possible to incrementally remove classic histograms with relabel rules in the scrape config.) If everything still works, it is time to remove classic histograms from the instrumentation.

The Grafana Mimir documentation contains a detailed migration guide following the same philosophy as described in this section.

Previous
Alerting based on metrics

Edit

Next
1.0