SPECIFICATIONS

OVERVIEW

What is Prometheus?

Prometheus (https://github.com/prometheus) is an open-source systems monitoring and alerting toolkit originally built at SoundCloud (https://soundcloud.com). Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community (/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 (https://cncf.io/) in 2016 as the second hosted project, after Kubernetes (http://kubernetes.io/).

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 (/docs/introduction/media/) section.

Features

Prometheus's main features are:

- a multi-dimensional data model (/docs/concepts/data_model/) with time series data identified by metric name and key/value pairs
- PromQL, a flexible query language (/docs/prometheus/latest/querying/basics/) 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 (/docs/instrumenting/pushing/) 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?

Metrics are numerical measurements in layperson terms. The term time series refers to the recording of changes over time. What users want to measure differs from application to application. For a web server, it could be request times; for a database, it could be the 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:

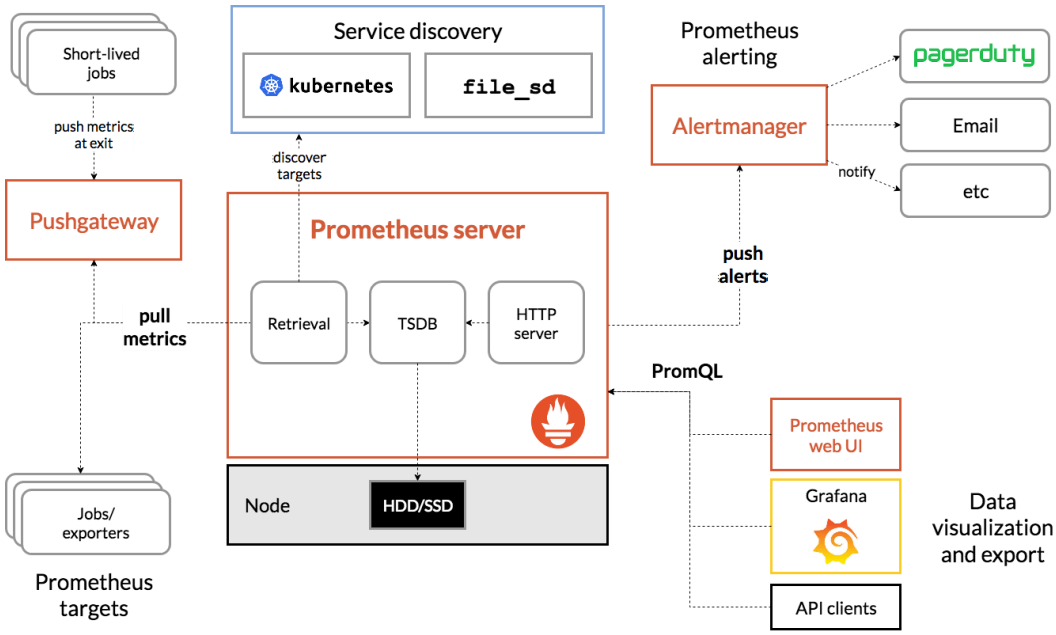
- the main Prometheus server (https://github.com/prometheus/prometheus) which scrapes and stores time series data
- client libraries (/docs/instrumenting/clientlibs/) for instrumenting application code
- a push gateway (https://github.com/prometheus/pushgateway) for supporting short-lived jobs
- special-purpose exporters (/docs/instrumenting/exporters/) for services like HAProxy, StatsD, Graphite, etc.
- an alertmanager (https://github.com/prometheus/alertmanager) to handle alerts

- various support tools

Most Prometheus components are written in Go (https://golang.org/), 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 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 (https://grafana.com/) 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.

SPECIFICATIONS

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 (/download) 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

Prometheus configuration is YAML (https://yaml.org/). The Prometheus download comes with a sample configuration in a file called `prometheus.yml` that is a good place to get started.

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 (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 (/docs/operating/configuration).

Starting Prometheus

To start Prometheus with our newly created configuration file, change to the directory 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 (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 (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 (http://localhost:9090/graph) and choose the "Table" view within the "Graph" tab.

As you can gather from http://localhost:9090/metrics (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)
```

For more about the expression language, see the expression language documentation (/docs/querying/basics/).

Using the graphing interface

To graph expressions, navigate to http://localhost:9090/graph (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 (/docs/guides/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 (/docs/introduction/overview) for some ideas about what to explore next.

SPECIFICATIONS

COMPARISON TO ALTERNATIVES

Prometheus vs. Graphite

Scope

Graphite (http://graphite.readthedocs.org/en/latest/) 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 (https://github.com/etsy/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
```

In Prometheus the same data could be encoded like this (assuming three api-server instances):

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

Storage

Graphite stores time series data on local disk in the Whisper (http://graphite.readthedocs.org/en/latest/whisper.html) 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 (https://influxdata.com/) 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 (https://github.com/influxdata/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 rules.

Kapacitor's scope is a combination of Prometheus recording rules, alerting rules, and the Alertmanager's notification functionality. Prometheus offers a more powerful query language for graphing and alerting (https://www.robustperception.io/translating-between-monitoring-languages/). 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 (https://docs.influxdata.com/influxdb/v1.7/concepts/storage_engine/), 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! (https://grafana.com/blog/2016/01/05/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 (https://docs.influxdata.com/enterprise_kapacitor), which supports an HA/redundant alerting system.

Prometheus and the Alertmanager by contrast offer a fully open-source redundant option via running redundant replicas of Prometheus and using the Alertmanager's High Availability (https://github.com/prometheus/alertmanager#high-availability) mode.

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

Prometheus vs. OpenTSDB

OpenTSDB (http://opentsdb.net/) is a distributed time series database based on Hadoop (http://hadoop.apache.org/) and HBase (http://hbase.apache.org/).

Scope

The same scope differences as in the case of Graphite (/docs/introduction/comparison/#prometheus-vs-graphite) apply here.

Data model

OpenTSDB's data model is almost identical to Prometheus's: time series are identified by a set of arbitrary key-value pairs (OpenTSDB tags are Prometheus labels). All data for a metric is stored together (http://opentsdb.net/docs/build/html/user_guide/writing/index.html#time-series-cardinality), limiting the cardinality of metrics. There are minor differences though: 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 (http://opentsdb.net/)'s storage is implemented on top of Hadoop (http://hadoop.apache.org/) and HBase (http://hbase.apache.org/). 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 (https://www.nagios.org/) 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 (https://github.com/shawn-sterling/graphios) or using NRPE to run checks on remote machines (https://exchange.nagios.org/directory/Addons/Monitoring-Agents/NRPE--2D-Nagios-Remote-Plugin-Executor/details).

Data model

Nagios is host-based. Each host can have one or more services and each service can perform one check.

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 (https://docs.pnp4nagios.org/).

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 (https://sensu.io) 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 (https://docs.sensu.io/sensu-go/latest/observability-pipeline/observe-events/events/). It provides an extensible framework for event filtering (https://docs.sensu.io/sensu-go/latest/observability-pipeline/observe-filter/), aggregation, transformation (https://docs.sensu.io/sensu-go/latest/observability-pipeline/observe-transform/), and processing (https://docs.sensu.io/sensu-go/latest/observability-pipeline/observe-process/) – 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

Sensu Events (https://docs.sensu.io/sensu-go/latest/observability-pipeline/observe-events/events/) represent service health and/or metrics (https://docs.sensu.io/sensu-go/latest/observability-pipeline/observe-events/events/#metric-attributes) in a structured data format identified by an entity (https://docs.sensu.io/sensu-go/latest/observability-pipeline/observe-entities/entities/) name (e.g. server, cloud compute instance, container, or service), an event name, and optional key-value metadata (https://docs.sensu.io/sensu-go/latest/observability-pipeline/observe-events/events/#metadata-attributes) called "labels" or "annotations". The Sensu Event payload may include one or more metric `points`

(https://docs.sensu.io/sensu-go/latest/observability-pipeline/observe-events/events/#points-attributes), 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

Sensu is maintained by a single commercial company following the open-core business model, offering premium features like closed-source event correlation and aggregation, federation, 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.

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 (https://kubernetes.io/) and the de facto standard across the cloud native ecosystem (https://landscape.cncf.io/). See the overview (/docs/introduction/overview/).

How does Prometheus compare against other monitoring systems?

See the comparison (/docs/introduction/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 (https://prometheus.io/docs/prometheus/latest/configuration/configuration/) 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.

Can Prometheus be made highly available?

Yes, run identical Prometheus servers on two or more separate machines. Identical alerts will be deduplicated by the Alertmanager (https://github.com/prometheus/alertmanager).

Alertmanager supports high availability (https://github.com/prometheus/alertmanager#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 (https://github.com/hashicorp/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 (https://www.robustperception.io/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 (https://prometheus.io/docs/operating/integrations/#remote-endpoints-and-storage).

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 (http://semver.org/). 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?

Pulling over HTTP offers a number of advantages:

- You can start extra monitoring instances as needed, e.g. on your laptop when 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 (/docs/instrumenting/pushing/).

How to feed logs into Prometheus?

Short answer: Don't! Use something like Grafana Loki (https://grafana.com/oss/loki/) or OpenSearch (https://opensearch.org/) 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! (https://grafana.com/blog/2016/01/05/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 (https://grafana.com/docs/loki/latest/logql/metric_queries/) documentation.

Who wrote Prometheus?

Prometheus was initially started privately by Matt T. Proud (http://www.matttproud.com) and Julius Volz (http://juliusv.com). The majority of its initial development was sponsored by SoundCloud (https://soundcloud.com).

It's now maintained and extended by a wide range of companies (https://prometheus.devstats.cncf.io/d/5/companies-table?orgId=1) and individuals (https://prometheus.io/governance).

What license is Prometheus released under?

Prometheus is released under the Apache 2.0 (https://github.com/prometheus/prometheus/blob/main/LICENSE) license.

What is the plural of Prometheus?

After extensive research (https://youtu.be/B_CDeYrqxjQ), it has been determined that the correct plural of 'Prometheus' is 'Prometheis'.

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

Can I reload Prometheus's configuration?

Yes, sending `SIGHUP` to the Prometheus process or an HTTP POST request to the `/-/reload` 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 (https://github.com/prometheus/alertmanager).

We support sending alerts through email, various native integrations (https://prometheus.io/docs/alerting/latest/configuration/), and a webhook system anyone can add integrations to (https://prometheus.io/docs/operating/integrations/#alertmanager-webhook-receiver).

Can I create dashboards?

Yes, we recommend Grafana (/docs/visualization/grafana/) for production usage. There are also Console templates (/docs/visualization/consoles/).

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 (https://github.com/prometheus/prometheus/issues/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 (/docs/instrumenting/clientlibs/) documentation for details.

If you are interested in contributing a client library for a new language, see the exposition formats (/docs/instrumenting/exposition_formats/).

Can I monitor machines?

Yes, the Node Exporter (https://github.com/prometheus/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.

Can I monitor network devices?

Yes, the SNMP Exporter (https://github.com/prometheus/snmp_exporter) allows monitoring of devices that support SNMP. For industrial networks, there's also a Modbus exporter (https://github.com/RichiH/modbus_exporter).

Can I monitor batch jobs?

Yes, using the Pushgateway (/docs/instrumenting/pushing/). See also the best practices (/docs/practices/instrumentation/#batch-jobs) for monitoring batch jobs.

What applications can Prometheus monitor out of the box?

See the list of exporters and integrations (/docs/instrumenting/exporters/).

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 (https://github.com/prometheus/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 (https://github.com/prometheus/client_java/blob/master/benchmarks/README.md) 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 (https://en.wikipedia.org/wiki/Double-precision_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.

SPECIFICATIONS

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 (https://github.com/prometheus/prometheus/issues).

- Server-side metric metadata support
- Adopt OpenMetrics
- Retroactive rule evaluations
- TLS and authentication in HTTP serving endpoints
- Support the Ecosystem

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 has a range of client libraries and exporters. There are always more languages that could be supported, or systems that would be useful to export metrics from. We will support the ecosystem in creating and expanding these.

DESIGN DOCUMENTS

See the github.com/prometheus/proposals (https://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 (https://github.com/prometheus/proposals#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 (https://docs.google.com/document/d/1lEP7pGYM2-5GT9fAIDqrOecG86VRU8-1qAV8b6xZ29Q)	2020-05
Thoughts about timestamps and durations in PromQL (https://docs.google.com/document/d/1jMeDsLvDfO92Qnry_JLAXalvMRzMSB1sBr9V7LolpYM)	2020-10
Prometheus, OpenMetrics & OTLP (https://docs.google.com/document/d/1hn-u6WKLHxIsqYT1_u6eh94lyQeXrFaAouMshJcQFXs)	2021-03
Prometheus Sparse Histograms and PromQL (https://docs.google.com/document/d/1ch6ru8GKg03N02jRjYriurt-CZqUVY09evPg6yKTA1s/edit)	2021-10
Quoting Prometheus names (https://docs.google.com/document/d/1yFj5QSd1AgCYecZ9EJ8f2t4OgF2KBZgJYVde-uzVEtI/edit)	2023-01

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

MEDIA

There is a subreddit (https://www.reddit.com/r/prometheusmonitoring) collecting all Prometheus-related resources on the internet.

- Blogs
- Tutorials
- Podcasts and interviews
- Recorded talks

The following selection of resources are particularly useful to get started with Prometheus. Awesome Prometheus (https://github.com/roaldnefs/awesome-prometheus) contains a more comprehensive community-maintained list of resources.

Blogs

- This site has its own blog (/blog/).
- SoundCloud's blog post announcing Prometheus (https://developers.soundcloud.com/blog/prometheus-monitoring-at-soundcloud) – a more elaborate overview than the one given on this site.
- Prometheus-related posts on the Robust Perception blog (https://www.robustperception.io/tag/prometheus/).

Tutorials

- Instructions and example code for a Prometheus workshop (https://github.com/juliusv/prometheus_workshop).
- How To Install Prometheus using Docker on Ubuntu 14.04 (https://www.digitalocean.com/community/tutorials/how-to-install-prometheus-using-docker-on-ubuntu-14-04).

Podcasts and interviews

- Prometheus on FLOSS Weekly 357 (https://twit.tv/shows/floss-weekly/episodes/357) - Julius Volz on the FLOSS Weekly TWiT.tv (https://twit.tv/shows/floss-weekly/) show.
- Prometheus and Service Monitoring (https://changelog.com/podcast/168) - Julius Volz on the Changelog (https://changelog.com/) podcast.

Recorded talks

- Prometheus: A Next-Generation Monitoring System (https://www.usenix.org/conference/srecon15europe/program/presentation/rabenstein) – Julius Volz and Björn Rabenstein at SREcon15 Europe, Dublin.
- Prometheus: A Next-Generation Monitoring System (https://www.youtube.com/watch?v=cwRmXqXKGtk) - Brian Brazil at FOSDEM 2016 (slides (http://www.slideshare.net/brianbrazil/prometheus-a-next-generation-monitoring-system-fosdem-2016)).
- What is your application doing right now? (https://youtu.be/Z0LlilNpX1U) – Matthias Gruter, Transmode, at DevOps Stockholm Meetup.
- Prometheus workshop (https://vimeo.com/131581353) – Jamie Wilkinson at Monitorama PDX 2015 (slides (https://docs.google.com/presentation/d/1X1rKozAUuF2MVc1YXElFWq9wkcWv3Axdldl8LOH9Vik/edit)).
- Monitoring Hadoop with Prometheus (https://www.youtube.com/watch?v=qs2sqOLNGtw) – Brian Brazil at the Hadoop User Group Ireland (slides (http://www.slideshare.net/brianbrazil/monitoring-hadoop-with-prometheus-hadoop-user-group-ireland-december-2015)).
- In German: Monitoring mit Prometheus (https://media.ccc.de/v/eh16-43-monitoring_mit_prometheus#video&t=2804) – Michael Stapelberg at Easterhegg 2016 (https://eh16.easterhegg.eu/).
- Presentation slides
  - General
  - Docker
  - Python

- In German: Prometheus in der Praxis (https://media.ccc.de/v/MRMCD16-7754-prometheus_in_der_praxis) – Jonas Große Sundrup at MRMCD 2016 (https://2016.mrmcd.net/)

Presentation slides

General

- Prometheus Overview (http://www.slideshare.net/brianbrazil/prometheus-overview) – by Brian Brazil.
- Systems Monitoring with Prometheus (http://www.slideshare.net/brianbrazil/devops-ireland-systems-monitoring-with-prometheus) – Brian Brazil at Devops Ireland Meetup, Dublin.
- OMG! Prometheus (https://www.dropbox.com/s/0l7kxhjqjbabtb0/prometheus%20site-ops%20preso.pdf?dl=0) – Benjamin Staffin, Fitbit Site Operations, explains the case for Prometheus to his team.

Docker

- Prometheus and Docker (http://www.slideshare.net/brianbrazil/prometheus-and-docker-docker-galway-november-2015) – Brian Brazil at Docker Galway.

Python

- Better Monitoring for Python (http://www.slideshare.net/brianbrazil/better-monitoring-for-python-inclusive-monitoring-with-prometheus-pycon-ireland-lightning-talk) – Brian Brazil at Pycon Ireland.
- Monitoring your Python with Prometheus (http://www.slideshare.net/brianbrazil/python-ireland-monitoring-your-python-with-prometheus) – Brian Brazil at Python Ireland Meetup, Dublin.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

SPECIFICATIONS

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 (../../alerting/overview/) 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

Direct instrumentation is instrumentation added inline as part of the source code of a program, using a client library.

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 (../../visualization/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.

PromQL

PromQL (/docs/prometheus/latest/querying/basics/) is the Prometheus Query Language. It allows for a wide range of operations including aggregation, slicing and dicing, prediction and joins.

Pushgateway

The Pushgateway (../../instrumenting/pushing/) 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

Not all systems directly support remote write. A remote write adapter sits between Prometheus and another system, converting the samples in the remote write into a format the other system can understand.

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.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

SPECIFICATIONS

LONG TERM SUPPORT

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

- List of LTS releases
- Limitations of LTS support

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 (https://prometheus.io/docs/prometheus/latest/stability/). 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

Limitations of LTS support

Some features are excluded from LTS support:

- Things listed as unstable in our API stability guarantees (https://prometheus.io/docs/prometheus/latest/stability/).
- Experimental features (https://prometheus.io/docs/prometheus/latest/feature_flags/).
- OpenBSD support.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

DATA MODEL

Prometheus fundamentally stores all data as *time series* (https://en.wikipedia.org/wiki/Time_series): streams of timestamped values belonging to

- Metric names and labels
- Samples

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.

- Notation

Metric names and labels

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

Metric names:

- 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 contain ASCII letters, digits, underscores, and colons. It must match the regex `[a-zA-Z_:][a-zA-Z0-9_:]*`.

Note: The colons are reserved for user defined recording rules. They should not be used by exporters or direct instrumentation.

Metric labels:

- Enable Prometheus's dimensional data model to identify any given combination of labels for the same metric name. It identifies a particular dimensional instantiation of that metric (for example: all HTTP requests that used the method `POST` to the `/api/tracks` handler). 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.
- Labels may contain ASCII letters, numbers, as well as underscores. They must match the regex `[a-zA-Z_][a-zA-Z0-9_]*`.
- Label names beginning with `__` (two "_") are reserved for internal use.
- Label values may contain any Unicode characters.
- Labels with an empty label value are considered equivalent to labels that do not exist.

See also the best practices for naming metrics and labels (/docs/practices/naming/).

Samples

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

- a float64 value
- a millisecond-precision timestamp

> **NOTE:** Beginning with Prometheus v2.40, there is experimental support for native histograms. Instead of a simple float64, the sample value may now take the form of a full histogram.

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 (http://opentsdb.net/) uses.

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)

- Counter
- Gauge
- Histogram
- Summary

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 (https://en.wikipedia.org/wiki/Monotonic_function) 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 (http://godoc.org/github.com/prometheus/client_golang/prometheus#Counter)
- Java (https://github.com/prometheus/client_java#counter)
- Python (https://prometheus.github.io/client_python/instrumenting/counter/)
- Ruby (https://github.com/prometheus/client_ruby#counter)
- .Net (https://github.com/prometheus-net/prometheus-net#counters)

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:

- Go (http://godoc.org/github.com/prometheus/client_golang/prometheus#Gauge)
- Java (https://github.com/prometheus/client_java#gauge)
- Python (https://prometheus.github.io/client_python/instrumenting/gauge/)
- Ruby (https://github.com/prometheus/client_ruby#gauge)
- .Net (https://github.com/prometheus-net/prometheus-net#gauges)

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 (/docs/prometheus/latest/querying/functions/#histogram_quantile) to calculate quantiles from histograms or even aggregations of histograms. A histogram is also suitable to calculate an Apdex score (https://en.wikipedia.org/wiki/Apdex). When operating on buckets, remember that the histogram is cumulative (https://en.wikipedia.org/wiki/Histogram#Cumulative_histogram). See histograms and summaries (/docs/practices/histograms) 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.

Client library usage documentation for histograms:

- Go (http://godoc.org/github.com/prometheus/client_golang/prometheus#Histogram)
- Java (https://github.com/prometheus/client_java#histogram)
- Python (https://prometheus.github.io/client_python/instrumenting/histogram/)
- Ruby (https://github.com/prometheus/client_ruby#histogram)

- .Net (https://github.com/prometheus-net/prometheus-net#histogram)

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 (/docs/practices/histograms) for detailed explanations of φ-quantiles, summary usage, and differences to histograms.

Client library usage documentation for summaries:

- Go (http://godoc.org/github.com/prometheus/client_golang/prometheus#Summary)
- Java (https://github.com/prometheus/client_java#summary)
- Python (https://prometheus.github.io/client_python/instrumenting/summary/)
- Ruby (https://github.com/prometheus/client_ruby#summary)
- .Net (https://github.com/prometheus-net/prometheus-net#summary)

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 (/docs/prometheus/latest/configuration/configuration/#scrape_config) for more information.

For each instance scrape, Prometheus stores a sample (/docs/introduction/glossary#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.
- `scrape_series_added{job="<job-name>", instance="<instance-id>"}` : the approximate number of new series in this scrape. *New in v2.10*

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

With the `extra-scrape-metrics` feature flag (/docs/prometheus/latest/feature_flags/#extra-scrape-metrics) several addditonal 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.

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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 (/download) of Prometheus for your platform, then extract and run it:

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

Before starting Prometheus, let's configure it.

- Downloading and running Prometheus
- Configuring Prometheus to monitor itself
- Starting Prometheus
- Using the expression browser
- Using the graphing interface
- Starting up some sample targets
- Configure Prometheus to monitor the sample targets
- Configure rules for aggregating scraped data into new time series
- Reloading configuration
- Shutting down your instance gracefully.

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.

  # Attach these labels to any time series or alerts when communicating with
  # external systems (federation, remote storage, Alertmanager).
  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 f
  - job_name: 'prometheus'

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

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

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

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 (http://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 (http://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 (http://localhost:9090/graph) and choose the "Table" view within the "Graph" tab.

As you can gather from localhost:9090/metrics (http://localhost:9090/metrics), one metric that Prometheus exports about 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 (../querying/basics/).

Using the graphing interface

To graph expressions, navigate to http://localhost:9090/graph (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.

The Node Exporter is used as an example target, for more information on using it see these instructions. (/docs/guides/node-exporter/)

```
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:8080/metrics), http://localhost:8081/metrics (http://localhost:8081/metrics), and

http://localhost:8082/metrics (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 second
    scrape_interval: 5s

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

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

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 inst
  external_labels:
    monitor: 'codelab-monitor'

rule_files:
  - 'prometheus.rules.yml'

scrape_configs:
  - job_name: 'prometheus'

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

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

  - job_name:       'node'

    # Override the global default and scrape targets from this job every 5 second
    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 (../configuration/configuration/) 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.

While Prometheus does have recovery mechanisms in the case that there is an abrupt process failure it is recommend to use the `SIGTERM` signal to cleanly shutdown a Prometheus instance. If you're running on Linux this can be performed by using `kill -s SIGTERM <PID>`, replacing `<PID>` with your Prometheus process ID.

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

INSTALLATION

Using pre-compiled binaries

We provide precompiled binaries for most official Prometheus components. Check out the download section (/download) 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

(https://quay.io/repository/prometheus/prometheus) or Docker Hub (https://hub.docker.com/r/prom/prometheus/).

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 (https://docs.docker.com/storage/volumes/) 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 (https://github.com/prometheus/prometheus/blob/main/Dockerfile) (adjust the link to correspond with the version in use).

If you want to add extra command line parameters to the `docker run` command, you will need to re-add these yourself as they will be overwritten.

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
```

Custom image

To avoid managing a file on the host and bind-mount it, the configuration can be baked into 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 (https://github.com/prometheus-community/ansible)

Chef

- rayrod2030/chef-prometheus (https://github.com/rayrod2030/chef-prometheus)

Puppet

- puppet/prometheus (https://forge.puppet.com/puppet/prometheus)

SaltStack

- saltstack-formulas/prometheus-formula (https://github.com/saltstack-formulas/prometheus-formula)

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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 (/docs/concepts/jobs_instances/), as well as which rule files to load (../recording_rules/#configuring-rules).

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 (https://en.wikipedia.org/wiki/YAML), 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`

- Configuration file
  - `<scrape_config>`
  - `<tls_config>`
  - `<oauth2>`
  - `<azure_sd_config>`
  - `<consul_sd_config>`
  - `<digitalocean_sd_config>`
  - `<docker_sd_config>`
  - `<dockerswarm_sd_config>`
  - `<dns_sd_config>`
  - `<ec2_sd_config>`
  - `<openstack_sd_config>`
  - `<ovhcloud_sd_config>`
  - `<puppetdb_sd_config>`
  - `<file_sd_config>`
  - `<gce_sd_config>`
  - `<hetzner_sd_config>`
  - `<http_sd_config>`
  - `<ionos_sd_config>`
  - `<kubernetes_sd_config>`
  - `<kuma_sd_config>`
  - `<lightsail_sd_config>`
  - `<linode_sd_config>`
  - `<marathon_sd_config>`
  - `<nerve_sd_config>`
  - `<nomad_sd_config>`
  - `<serverset_sd_config>`
  - `<triton_sd_config>`
  - `<eureka_sd_config>`
  - `<scaleway_sd_config>`
  - `<uyuni_sd_config>`
  - `<vultr_sd_config>`
  - `<static_config>`

- `<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
- `<host>` : a valid string consisting of a hostname or IP followed by an optional port number
- `<int>` : an integer value
- `<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
    - `<relabel_config>`
    - `<metric_relabel_configs>`
    - `<alert_relabel_configs>`
    - `<alertmanager_config>`
    - `<remote_write>`
    - `<remote_read>`
    - `<tsdb>`
    - `<exemplars>`
    - `<tracing_config>`

The other placeholders are specified separately.

A valid example file can be found here (https://github.com/prometheus/prometheus/blob/release-2.54/config/testdata/conf.good.yml).

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.
  [ 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, OpenMetricsText0.0.1, Promet
  # when native_histogram feature flag is set.
  [ scrape_protocols: [<string>, ...] | default = [ OpenMetricsText1.0.0, OpenMetricsText0.0.1, Promet

  # 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 p
  # Metric availability delays are more likely to occur when Prometheus is running as a remote write t
  [ 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).
  external_labels:
    [ <labelname>: <labelvalue> ... ]

  # File to which PromQL queries are logged.
  # Reloading the configuration will reopen the file.
  [ query_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 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 ]

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

  # Per-scrape limit on length of labels name that will be accepted for a sample.
  # If a label name is longer than this number post metric-relabeling, the entire
  # scrape will be treated as failed. 0 means no limit.
  [ label_name_length_limit: <int> | default = 0 ]

  # Per-scrape limit on length of labels value that will be accepted for a sample.
  # If a label value is longer than this number post metric-relabeling, the
  # entire scrape will be treated as failed. 0 means no limit.
```

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

  # Per-scrape config limit 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 ]

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:
  alert_relabel_configs:
    [ - <relabel_config> ... ]
  alertmanagers:
    [ - <alertmanager_config> ... ]

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

# 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>`

A `scrape_config` section specifies a set of targets and parameters describing how to scrape them. In the general case, one scrape configuration specifies a single job. In advanced configurations, this may change.

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.
[ 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.
[ scrape_protocols: [<string>, ...] | default = <global_config.scrape_protocols> ]

# Whether to scrape a classic histogram that is also exposed as a native
# histogram (has no effect without --enable-feature=native-histograms).
[ 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
# values from the scraped data and ignoring the conflicting server-side labels.
#
# If honor_labels is set to "false", label conflicts are resolved by renaming
# 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 ]

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

# Sets the `Authorization` header on every scrape 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 scrape requests follow HTTP 3xx redirects.
[ follow_redirects: <boolean> | default = true ]

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

# Configures the scrape 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]


# 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> ... ]

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

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

# Per-scrape limit on length of labels name that will be accepted for a sample.
# If a label name is longer than this number post metric-relabeling, the entire
# scrape will be treated as failed. 0 means no limit.
[ label_name_length_limit: <int> | default = 0 ]

# Per-scrape limit on length of labels value that will be accepted for a sample.
# If a label value is longer than this number post metric-relabeling, the
# entire scrape will be treated as failed. 0 means no limit.
[ label_value_length_limit: <int> | default = 0 ]

# Per-scrape config limit 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 ]
```

```
# Per-job limit on the number of targets dropped by relabeling
# that will be kept in memory. 0 means no limit.
[ keep_dropped_targets: <int> | default = 0 ]

# 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
# reduced until the number of buckets is within the limit. If the limit cannot
# be reached, the scrape will fail.
# 0 means no limit.
[ 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
# schema 8, but might change in the future).
[ native_histogram_min_bucket_factor: <float> | default = 0 ]
```

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

`<tls_config>`

A `tls_config` allows configuring TLS connections.

```
# CA certificate to validate API server certificate with. At most one of ca and ca_file is allowed.
[ 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> ]
[ key: <secret> ]
[ key_file: <filename> ]

# 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.
endpoint_params:
  [ <string>: <string> ... ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]
```

`<azure_sd_config>`

Azure SD configurations allow retrieving scrape targets from Azure VMs.

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 (https://docs.microsoft.com/en-us/azure/virtual-machine-scale-sets/))
- `__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-azure-resources/overv
# 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> ]

# Refresh interval to re-read the instance list.
[ refresh_interval: <duration> | default = 300s ]

# 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 ]

# Authentication information used to authenticate to the Azure API.
# Note that `basic_auth`, `authorization` and `oauth2` options are
# mutually exclusive.
# `password` and `password_file` are mutually exclusive.

# Optional HTTP basic authentication information, currently not support by Azure.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional `Authorization` header configuration, currently not supported by Azure.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to the credentials read from the configured file.
  # It is mutually exclusive with `credentials`.
  [ credentials_file: <filename> ]

# Optional OAuth 2.0 configuration, currently not supported by Azure.
oauth2:
  [ <oauth2> ]
```

```
# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

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

`<consul_sd_config>`

Consul SD configurations allow retrieving scrape targets from Consul's (https://www.consul.io) 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.
[ namespace: <string> ]
# Admin Partitions are only supported in Consul Enterprise.
[ partition: <string> ]
[ scheme: <string> | default = "http" ]
# The username and password fields are deprecated in favor of the basic_auth configuration.
[ username: <string> ]
[ password: <secret> ]

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

# See https://www.consul.io/api/catalog.html#list-nodes-for-service to know more
# about the possible filters that can be used.

# An optional list of tags used to filter nodes for a given service. Services must contain all tags in
tags:
  [ - <string> ]

# Node metadata key/value pairs to filter nodes for a given service.
[ 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.html). Will reduce lo
[ 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 will change all th
[ refresh_interval: <duration> | default = 30s ]

# Authentication information used to authenticate to the consul server.
# Note that `basic_auth`, `authorization` and `oauth2` options are
# mutually exclusive.
# `password` and `password_file` are mutually exclusive.

# Optional HTTP basic authentication information.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional `Authorization` header configuration.
```

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

# Optional OAuth 2.0 configuration.
oauth2:
  [ <oauth2> ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

# TLS configuration.
tls_config:
  [ <tls_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 setups, the relevant address is in `__meta_consul_service_address`. In those cases, you can use the relabel feature to replace the special `__address__` label.

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 (https://www.digitalocean.com/) 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 (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/prometheus-digitalocean.yml).

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

```
# Authentication information used to authenticate to the API server.
# Note that `basic_auth` and `authorization` options are
# mutually exclusive.
# password and password_file are mutually exclusive.

# Optional HTTP basic authentication information, not currently supported by DigitalOcean.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional `Authorization` header configuration.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to 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> ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

# TLS configuration.
tls_config:
  [ <tls_config> ]

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

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

`<docker_sd_config>`

Docker SD configurations allow retrieving scrape targets from Docker Engine (https://docs.docker.com/engine/) hosts.

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>

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

# TLS configuration.
tls_config:
  [ <tls_config> ]

# 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 ]

# Optional filters to limit the discovery process to a subset of available
# resources.
# The available filters are listed in the upstream documentation:
# 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 ]

# Authentication information used to authenticate to the Docker daemon.
# Note that `basic_auth` and `authorization` options are
# mutually exclusive.
# password and password_file are mutually exclusive.

# Optional HTTP basic authentication information.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional `Authorization` header configuration.
```

```
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to 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 HTTP requests follow HTTP 3xx redirects.
[ follow_redirects: <boolean> | default = true ]

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

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 (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/prometheus-docker.yml) for a detailed example of configuring Prometheus for Docker Engine.

`<dockerswarm_sd_config>`

Docker Swarm SD configurations allow retrieving scrape targets from Docker Swarm (https://docs.docker.com/engine/swarm/) engine.

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

`services`

The `services` role discovers all Swarm services (https://docs.docker.com/engine/swarm/key-concepts/#services-and-tasks) 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

- `__meta_dockerswarm_service_endpoint_port_name` : the name of the endpoint port, if available
- `__meta_dockerswarm_service_endpoint_port_publish_mode` : the publish mode of the 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 (https://docs.docker.com/engine/swarm/key-concepts/#services-and-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
- `__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_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 (https://docs.docker.com/engine/swarm/key-concepts/#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
- `__meta_dockerswarm_node_role` : the role of the node
- `__meta_dockerswarm_node_status` : the status of the node

See below for the configuration options for Docker Swarm discovery:

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

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

# TLS configuration.
tls_config:
  [ <tls_config> ]

# 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 ]

# Authentication information used to authenticate to the Docker daemon.
# Note that `basic_auth` and `authorization` options are
# mutually exclusive.
# password and password_file are mutually exclusive.

# Optional HTTP basic authentication information.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional `Authorization` header configuration.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
```

```
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to 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 HTTP requests follow HTTP 3xx redirects.
[ follow_redirects: <boolean> | default = true ]

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

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 (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/prometheus-dockerswarm.yml) for a detailed example of configuring Prometheus for Docker Swarm.

`<dns_sd_config>`

A DNS-based service discovery configuration allows specifying a set of DNS domain names which are periodically queried to discover a list of targets. The DNS servers to be contacted are read from `/etc/resolv.conf`.

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 (https://tools.ietf.org/html/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:DescribeAavailabilityZones` 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_availability_zone_id` : the availability zone ID (https://docs.aws.amazon.com/ram/latest/userguide/working-with-az-ids.html) in which the instance is running (requires `ec2:DescribeAvailabilityZones`)
- `__meta_ec2_instance_id` : the EC2 instance ID
- `__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.
[ profile: <string> ]

# AWS Role ARN, an alternative to using AWS API keys.
[ 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/API_Filter.html
filters:
  [ - name: <string>
      values: <string>, [...] ]

# Authentication information used to authenticate to the EC2 API.
# Note that `basic_auth`, `authorization` and `oauth2` options are
# mutually exclusive.
# `password` and `password_file` are mutually exclusive.

# Optional HTTP basic authentication information, currently not supported by AWS.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional `Authorization` header configuration, currently not supported by AWS.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to the credentials read from the configured file.
  # It is mutuall exclusive with `credentials`.
  [ credentials_file: <filename> ]
```

```
# Optional OAuth 2.0 configuration, currently not supported by AWS.
oauth2:
  [ <oauth2> ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

# TLS configuration.
tls_config:
  [ <tls_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_hostname` : the hypervisor node's name.
- `__meta_openstack_hypervisor_id` : the hypervisor node's ID.
- `__meta_openstack_hypervisor_state` : the hypervisor node's state.
- `__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.

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.
# Some providers allow you to specify a project_name instead of the project_id.
# Some require both. Your provider's authentication policies will determine
# how these fields influence authentication.
[ 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 permissions.
[ 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 internal.
[ 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 (https://www.ovhcloud.com/en/bare-metal/) and VPS (https://www.ovhcloud.com/en/vps/) using their API (https://api.ovh.com/). 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 changed with relabeling. For OVHcloud's public cloud instances (https://www.ovhcloud.com/en/public-cloud/) you can use the openstack*sd*config.

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
- `__meta_ovhcloud_dedicated_server_server_id` : the ID of the server
- `__meta_ovhcloud_dedicated_server_state` : the state of the server
- `__meta_ovhcloud_dedicated_server_support_level` : the support level 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 (https://puppet.com/docs/puppetdb/latest/index.html) 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:

```
# The URL of the PuppetDB root query endpoint.
url: <string>

# 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 ]

# TLS configuration to connect to the PuppetDB.
tls_config:
  [ <tls_config> ]

# basic_auth, authorization, and oauth2, are mutually exclusive.

# Optional HTTP basic authentication information.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# `Authorization` HTTP header configuration.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials 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> ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

See this example Prometheus configuration file (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/prometheus-puppetdb.yml) 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 (https://github.com/fsnotify/fsnotify/blob/c1467c02fba575afdb5f4201072ab8403bbf00f4/README.md?plain=1#L128) 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 may be provided in YAML or JSON format. Only changes resulting in well-formed target groups are applied.

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 (/docs/operating/integrations/#file-service-discovery) 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`.

`<gce_sd_config>`

GCE (https://cloud.google.com/compute/) SD configurations allow retrieving scrape targets from GCP GCE instances. 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_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 section:
# 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.
[ port: <int> | default = 80 ]

# The tag separator is used to separate the tags on concatenation
[ 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 (https://www.hetzner.com/) Cloud (https://www.hetzner.cloud/) API and Robot (https://docs.hetzner.com/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 (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/prometheus-hetzner.yml).

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_datacenter_location_network_zone` : the network zone of the server
- `__meta_hetzner_hcloud_server_type` : the type 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>

# Authentication information used to authenticate to the API server.
# Note that `basic_auth` and `authorization` options are
# mutually exclusive.
# password and password_file are mutually exclusive.

# Optional HTTP basic authentication information, required when role is robot
# Role hcloud does not support basic auth.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional `Authorization` header configuration, required when role is
# hcloud. Role robot does not support bearer token authentication.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to 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> ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

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

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

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

<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.

Example response body:

```
[
  {
    "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 ]

# Authentication information used to authenticate to the API server.
# Note that `basic_auth`, `authorization` and `oauth2` options are
# mutually exclusive.
# `password` and `password_file` are mutually exclusive.

# Optional HTTP basic authentication information.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

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

# Optional OAuth 2.0 configuration.
oauth2:
  [ <oauth2> ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

# TLS configuration.
```

```
tls_config:
  [ <tls_config> ]
```

`<ionos_sd_config>`

IONOS SD configurations allows retrieving scrape targets from IONOS Cloud (https://cloud.ionos.com/) 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_id` : the ID of the server
- `__meta_ionos_server_ip` : comma separated list of all IPs assigned to the server
- `__meta_ionos_server_lifecycle` : the lifecycle state of the server resource
- `__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>

# Authentication information used to authenticate to the API server.
# Note that `basic_auth` and `authorization` options are
# mutually exclusive.
# password and password_file are mutually exclusive.

# Optional HTTP basic authentication information, required when using IONOS
# Cloud username and password as authentication method.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional `Authorization` header configuration, required when using IONOS
# Cloud token as authentication method.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to 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> ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

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

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

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

<kubernetes_sd_config>

Kubernetes SD configurations allow retrieving scrape targets from Kubernetes' (https://kubernetes.io/) 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.
- `__meta_kubernetes_node_provider_id`: The cloud provider's name for the node object.
- `__meta_kubernetes_node_label_<labelname>`: Each label from the node object, with 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.
- `__meta_kubernetes_service_port_protocol` : Protocol of the service port for the target.
- `__meta_kubernetes_service_type` : The type of the service.

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 (https://kubernetes.io/docs/concepts/workloads/pods/init-containers/)
- `__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 (https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-phase).
- `__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.

`endpoints`

The `endpoints` role discovers targets from listed endpoints of a service. For each endpoint 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.

Available meta labels:

- `__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 (only available when using the `discovery.k8s.io/v1` API group).
  - `__meta_kubernetes_endpointslice_port`: Port of the referenced endpoint.
  - `__meta_kubernetes_endpointslice_port_name`: Named port of the referenced endpoint.
  - `__meta_kubernetes_endpointslice_port_protocol`: Protocol of the referenced 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.

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/serviceaccount/.
[ api_server: <host> ]

# The Kubernetes role of entities that should be discovered.
# One of endpoints, endpointslice, service, pod, node, or ingress.
role: <string>

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

# Optional authentication information used to authenticate to the API server.
# Note that `basic_auth` and `authorization` options are mutually exclusive.
# password and password_file are mutually exclusive.

# Optional HTTP basic authentication information.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional `Authorization` header configuration.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to 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> ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

# Configure whether HTTP requests follow HTTP 3xx redirects.
```

```
[ follow_redirects: <boolean> | default = true ]

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

# TLS configuration.
tls_config:
  [ <tls_config> ]

# 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 available resources
# See https://kubernetes.io/docs/concepts/overview/working-with-objects/field-selectors/
# and https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/ to learn more about th
# filters that can be used. The endpoints role supports pod, service and endpoints selectors.
# The pod role supports node selectors when configured with `attach_metadata: {node: true}`.
# Other roles only support selectors matching the role itself (e.g. node role can only contain node se

# 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 other hand,
# if you just want to monitor small subset of pods in large cluster it's recommended to use selectors.
# 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 metadata is attached.
attach_metadata:
# Attaches node metadata to discovered targets. Valid for roles: pod, endpoints, endpointslice.
# When set to true, Prometheus must have permissions to get Nodes.
  [ node: <boolean> | default = false ]
```

See this example Prometheus configuration file (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/prometheus-kubernetes.yml) for a detailed example of configuring Prometheus for Kubernetes.

You may wish to check out the 3rd party Prometheus Operator (https://github.com/prometheus-operator/prometheus-operator), which automates the Prometheus setup on top of Kubernetes.

`<kuma_sd_config>`

Kuma SD configurations allow retrieving scrape target from the Kuma (https://kuma.io) control plane.

This SD discovers "monitoring assignments" based on Kuma Dataplane Proxies (https://kuma.io/docs/latest/documentation/dps-and-data-model), 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 specific Prometheus bac
# This is useful when migrating between multiple Prometheus backends, or having separate backend for e
# When not specified, system hostname/fqdn will be used if available, if not `prometheus` will be used
[ 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 ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

# TLS configuration.
tls_config:
  [ <tls_config> ]

# Authentication information used to authenticate to the Docker daemon.
# Note that `basic_auth` and `authorization` options are
# mutually exclusive.
# password and password_file are mutually exclusive.

# Optional HTTP basic authentication information.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional the `Authorization` header configuration.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials 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 HTTP requests follow HTTP 3xx redirects.
[ follow_redirects: <boolean> | default = true ]

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

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

`<lightsail_sd_config>`

Lightsail SD configurations allow retrieving scrape targets from AWS Lightsail (https://aws.amazon.com/lightsail/) instances. 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.
[ 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 ]

# Authentication information used to authenticate to the Lightsail API.
# Note that `basic_auth`, `authorization` and `oauth2` options are
# mutually exclusive.
# `password` and `password_file` are mutually exclusive.

# Optional HTTP basic authentication information, currently not supported by AWS.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional `Authorization` header configuration, currently not supported by AWS.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to the credentials read from the configured file.
  # It is mutuall exclusive with `credentials`.
  [ credentials_file: <filename> ]

# Optional OAuth 2.0 configuration, currently not supported by AWS.
oauth2:
  [ <oauth2> ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

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

`<linode_sd_config>`

Linode SD configurations allow retrieving scrape targets from Linode's (https://www.linode.com/) 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 (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/prometheus-linode.yml).

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
- `__meta_linode_specs_disk_bytes` : the amount of storage space the linode instance has access to
- `__meta_linode_specs_memory_bytes` : the amount of RAM the linode instance has 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

```
# Authentication information used to authenticate to the API server.
# Note that `basic_auth` and `authorization` options are
# mutually exclusive.
# password and password_file are mutually exclusive.
# Note: Linode APIv4 Token must be created with scopes: 'linodes:read_only', 'ips:read_only', and 'eve

# Optional HTTP basic authentication information, not currently supported by Linode APIv4.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional the `Authorization` header configuration.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials 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> ]

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

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

# TLS configuration.
tls_config:
  [ <tls_config> ]

# 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 ]
```

`<marathon_sd_config>`

Marathon SD configurations allow retrieving scrape targets using the Marathon (https://mesosphere.github.io/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
- `__meta_marathon_port_definition_label_<labelname>` : the port definition labels, with any unsupported characters converted to an underscore
- `__meta_marathon_port_mapping_label_<labelname>` : the port mapping labels, with 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-token
# It is mutually exclusive with `auth_token_file` and other authentication mechanisms.
[ auth_token: <secret> ]

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

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

# Optional `Authorization` header configuration.
# NOTE: The current version of DC/OS marathon (v1.11.0) does not support
# standard `Authentication` header, use `auth_token` or `auth_token_file`
# instead.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to 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 HTTP requests follow HTTP 3xx redirects.
[ follow_redirects: <boolean> | default = true ]

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

# TLS configuration for connecting to marathon servers
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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]
```

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 (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/prometheus-marathon.yml) 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>`

Nerve SD configurations allow retrieving scrape targets from AirBnB's Nerve (https://github.com/airbnb/nerve) which are stored in Zookeeper (https://zookeeper.apache.org/).

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 (https://www.nomadproject.io/) 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 ]
[ server: <host> ]
[ tag_separator: <string> | default = ,]

# Authentication information used to authenticate to the nomad server.
# Note that `basic_auth`, `authorization` and `oauth2` options are
# mutually exclusive.
# `password` and `password_file` are mutually exclusive.

# Optional HTTP basic authentication information.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

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

# Optional OAuth 2.0 configuration.
oauth2:
  [ <oauth2> ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

# TLS configuration.
```

```
tls_config:
  [ <tls_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 (https://zookeeper.apache.org/). Serversets are commonly used by Finagle (https://twitter.github.io/finagle/) and Aurora (https://aurora.apache.org/).

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 (https://github.com/joyent/triton) SD configurations allow retrieving scrape targets from Container Monitor (https://github.com/joyent/rfd/blob/master/rfd/0027/README.md) discovery endpoints.

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

container

The `container` role discovers one target per "virtual machine" owned by the `account` . These are SmartOS zones or lx/KVM/bhyve branded zones.

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 zones) running on Tr
# * "cn" to discover compute nodes (servers/global zones) making up the Triton infrastructure
[ 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>

# A list of groups for which targets are retrieved, only supported when `role` == `container`.
# If omitted all containers owned by the requesting account are scraped.
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 (https://github.com/Netflix/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
- `__meta_eureka_app_instance_secure_port` : the secure port address of the app instance
- `__meta_eureka_app_instance_secure_port_enabled` : the secure port of the app 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>

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

# Optional `Authorization` header configuration.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to 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> ]

# Configures the scrape 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

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

See the Prometheus eureka-sd configuration file (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/prometheus-eureka.yml) 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 (https://www.scaleway.com/en/virtual-instances/) and baremetal services (https://www.scaleway.com/en/bare-metal-servers/).

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_cluster_id` : the cluster ID of the server location
- `__meta_scaleway_instance_location_hypervisor_id` : the hypervisor ID of the server location
- `__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 (https://developers.scaleway.com/en/products/instance/api/#introduction))

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 (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/prometheus-scaleway.yml). 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
- `__meta_scaleway_baremetal_zone` : the zone of the server (ex: `fr-par-1` , complete list here (https://developers.scaleway.com/en/products/instance/api/#introduction))

This role uses the public IPv4 address by default. This can be changed with relabeling, as demonstrated in the Prometheus scaleway-sd configuration file (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/prometheus-scaleway.yml).

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/credentials
# 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 request.
[ name_filter: <string> ]

# TagsFilter specify a tag filter (a server needs to have all defined tags to be listed) to apply on t
tags_filter:
[ - <string> ]

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

# Configure whether HTTP 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]
```

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

`<uyuni_sd_config>`

Uyuni SD configurations allow retrieving scrape targets from managed systems via Uyuni (https://www.uyuni-project.org/) 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 ]

# Optional HTTP basic authentication information, currently not supported by Uyuni.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional `Authorization` header configuration, currently not supported by Uyuni.
authorization:
  # Sets the authentication type.
    [ type: <string> | default: Bearer ]
    # Sets the credentials. It is mutually exclusive with
    # `credentials_file`.
    [ credentials: <secret> ]
    # Sets the credentials to the credentials read from the configured file.
    # It is mutually exclusive with `credentials`.
    [ credentials_file: <filename> ]

# Optional OAuth 2.0 configuration, currently not supported by Uyuni.
# Cannot be used at the same time as basic_auth or authorization.
oauth2:
  [ <oauth2> ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

# Whether to enable HTTP2.
```

```
[ enable_http2: <boolean> | default: true ]

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

See the Prometheus uyuni-sd configuration file (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/prometheus-uyuni.yml) for a practical example on how to set up Uyuni Prometheus configuration.

`<vultr_sd_config>`

Vultr SD configurations allow retrieving scrape targets from Vultr (https://www.vultr.com/).

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 (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/prometheus-vultr.yml).

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.

```
# Authentication information used to authenticate to the API server.
# Note that `basic_auth` and `authorization` options are
# mutually exclusive.
# password and password_file are mutually exclusive.

# Optional HTTP basic authentication information, not currently supported by Vultr.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional `Authorization` header configuration.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to 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> ]

# 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

# TLS configuration.
tls_config:
  [ <tls_config> ]

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

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

<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. The `__param_<name>` label is set to the value of the first passed URL parameter called `<name>`.

The `__scrape_interval__` and `__scrape_timeout__` labels are set to the target's interval and timeout.

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.

If a relabeling step needs to store a label value only temporarily (as the input to a subsequent relabeling step), use the `__tmp` label name prefix. This prefix is guaranteed to never be used by Prometheus itself.

```
# The source labels select values from existing labels. Their content is concatenated
# using the configured separator and matched against the configured regular expression
# 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 (https://github.com/google/re2/wiki/Syntax). 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.
- `keep`: Drop targets for which `regex` does not match the concatenated `source_labels`.
- `drop`: Drop targets for which `regex` matches the concatenated `source_labels`.
- `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>`

An `alertmanager_config` section specifies Alertmanager instances the Prometheus server sends alerts to. It also provides parameters to configure how to communicate with these Alertmanagers.

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 ]

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

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

# Optionally configures AWS's Signature Verification 4 signing process to
# sign requests. Cannot be set at the same time as basic_auth, authorization, or oauth2.
# 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 OAuth 2.0 configuration.
# Cannot be used at the same time as basic_auth or authorization.
oauth2:
  [ <oauth2> ]
```

```
# Configures the scrape 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

# 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 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:
  [ - <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 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:
  [ - <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 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 (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/remote_storage) of how to use this functionality.

```
# The URL of the endpoint to send samples to.
url: <string>

# protobuf message to use when writing to the remote write endpoint.
#
# * The `prometheus.WriteRequest` represents the message introduced in Remote Write 1.0, which
# will be deprecated eventually.
# * The `io.prometheus.write.v2.Request` was introduced in Remote Write 2.0 and replaces the former,
# by improving efficiency and sending metadata, created timestamp and native histograms by default.
#
# Before changing this value, consult with your remote storage provider (or test) what message it supp
# Read more on https://prometheus.io/docs/specs/remote_write_spec_2_0/#io-prometheus-write-v2-request
[ protobuf_message: <prometheus.WriteRequest | io.prometheus.write.v2.Request> | default = prometheus.

# 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 write configs.
# The name will be used in metrics and logging in place of a generated value to help users distinguish
# remote write configs.
[ name: <string> ]

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

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

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

# Optional `Authorization` header configuration.
authorization:
  # Sets the authentication type.
  [ type: <string> | default = Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to the credentials read from the configured file.
```

```
  # It is mutually exclusive with `credentials`.
  [ credentials_file: <filename> ]

# Optionally configures AWS's Signature Verification 4 signing process to
# sign requests. Cannot be set at the same time as basic_auth, authorization, oauth2, or azuread.
# 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 OAuth 2.0 configuration.
# Cannot be used at the same time as basic_auth, authorization, sigv4, or azuread.
oauth2:
  [ <oauth2> ]

# Optional AzureAD configuration.
# Cannot be used at the same time as basic_auth, authorization, oauth2, or sigv4.
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-authentication
  [ sdk:
      [ tenant_id: <string> ] ]

# Configures the remote write 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default = false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

# 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 ]
```

```
# Maximum number of samples per send.
[ max_samples_per_send: <int> | default = 500]
```

There is a list of integrations (/docs/operating/integrations/#remote-endpoints-and-storage) 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 read configs.
# The name will be used in metrics and logging in place of a generated value to help users distinguish
# 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 ]

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

# Optional `Authorization` header configuration.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to 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> ]

# Configures the remote read 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_PROXY, https_proxy,
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

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

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

There is a list of integrations (/docs/operating/integrations/#remote-endpoints-and-storage) with this feature.

`<tsdb>`

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

NOTE: Out-of-order ingestion is an experimental feature, but you do not need any additional flag to enable it. Setting `out_of_order_time_window` to a positive duration enables it.

```
# Configures how old an out-of-order/out-of-bounds sample can be w.r.t. the TSDB max time.
# An out-of-order/out-of-bounds sample is ingested into the TSDB as long as the timestamp
# 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 sample
# 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 agent. It allows
# the agent's WAL to accept out-of-order samples that fall within the specified time window relative
# 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 series. Resizable
[ 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.

```
# Client used to export the traces. Options are 'http' or 'grpc'.
[ client_type: <string> | default = grpc ]

# 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 through 1.
[ 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> ]
```

> Feature flags (/docs/prometheus/latest/feature_flags/)

VISUALIZATION

INSTRUMENTING

OPERATING

ALERT MANAGER

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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 (../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 (../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 (/download/).

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.

If there are any syntax errors or invalid input arguments, it prints an error message to standard error and exits with a `1` return status.

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 (/docs/concepts/data_model/#metric-names-and-labels). The names of alerting rules must be valid label values (/docs/concepts/data_model/#metric-names-and-labels).

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 ]

# Offset the rule evaluation timestamp of this particular group by the specified
[ query_offset: <duration> | default = global.rule_query_offset ]

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 ]

# How long an alert will continue firing after the condition that triggered it
# has cleared.
[ keep_firing_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 (/docs/practices/rules/#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.

Feature flags (/docs/prometheus/latest/feature_flags/)

VISUALIZATION

INSTRUMENTING

OPERATING

ALERT MANAGER

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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
- Inspecting alerts during runtime
- Sending alert notifications

Defining alerting rules

Alerting rules are configured in Prometheus in the same way as recording rules (../recording_rules/).

An example rules file with an alert would be:

```
groups:
- name: example
  rules:
  - alert: HighRequestLatency
    expr: job:request_latency_seconds:mean5m{job="myjob"} > 0.5
    for: 10m
    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 evaluation for 10 minutes before firing the alert. Elements that are active, but not firing yet, are in the pending state. Alerting rules without the `for` clause will become active on the first evaluation.

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 (/docs/visualization/consoles). 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
    annotations:
      summary: "Instance {{ $labels.instance }} down"
      description: "{{ $labels.instance }} of job {{ $labels.job }} has been down

  # 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
```

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 (/docs/alerting/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 (../configuration/) to automatically discover available Alertmanager instances through its service discovery integrations.

> API Stability (/docs/prometheus/latest/stability/)
>
> Feature flags (/docs/prometheus/latest/feature_flags/)

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 (https://golang.org/pkg/text/template/) 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 mor
```

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:

```
{{ range query "up" }}
  {{ .Labels.instance }} {{ .Value }}
{{ 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 (/docs/visualization/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',
  </tr>
  <tr>
    <td>Transmitted</td>
    <td>{{ with printf "rate(node_network_transmit_bytes{job='node',instance='%s
  </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 (../template_reference/#console-templates) 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)}}
```

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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 (https://golang.org/pkg/text/template/) 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 (https://golang.org/pkg/text/template/#hdr-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.

Queries

Name	Arguments	Returns	Notes
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 (https://en.wikipedia.org/wiki/Metric_prefix).
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

Name	Arguments	Returns	Notes
title	string	string	strings.Title (https://golang.org/pkg/strings/#Title), capitalises first character of each word.

Name	Arguments	Returns	Notes
toUpper	string	string	strings.ToUpper (https://golang.org/pkg/strings/#ToUpper), converts all characters to upper case.
toLower	string	string	strings.ToLower (https://golang.org/pkg/strings/#ToLower), converts all characters to lower case.
stripPort	string	string	net.SplitHostPort (https://pkg.go.dev/net#SplitHostPort), splits string into host and port, then returns only host.
match	pattern, text	boolean	regexp.MatchString (https://golang.org/pkg/regexp/#MatchString) Tests for a unanchored regexp match.
reReplaceAll	pattern, replacement, text	string	Regexp.ReplaceAllString (https://golang.org/pkg/regexp/#Regexp.ReplaceAllString) Regexp substitution, unanchored.
graphLink	expr	string	Returns path to graph view in the expression browser (/docs/visualization/browser/) for the expression.
tableLink	expr	string	Returns path to tabular ("Table") view in the expression browser (/docs/visualization/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.

Name	Arguments	Returns	Notes
safeHtml	string	string	Marks string as HTML not requiring auto-escaping.
externalURL	*none*	string	The external URL under which Prometheus is externally reachable.
pathPrefix	*none*	string	The external URL path (https://pkg.go.dev/net/url#URL) 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 (https://golang.org/pkg/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.

API Stability (/docs/prometheus/latest/stability/)

Feature flags (/docs/prometheus/latest/feature_flags/)

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.y
./promtool test rules test1.yml test2.yml test
```

- Test file format
  - `<test_group>`
  - `<series>`
  - `<alert_test_case>`
  - `<alert>`
  - `<promql_test_case>`
  - `<sample>`
- Example
  - `test.yml`
  - `alerts.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
# rule groups (at a given evaluation time). The order is guaranteed only for the
# All the groups need not be mentioned below.
group_eval_order:
  [ - <group_name> ]

# All the tests are listed here.
tests:
  [ - <test_group> ]
```

<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 fi
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.
#     '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 decrementing by b (
#     'axn' becomes 'a a a … a' (a n+1 times) - it's a shorthand for 'a+0xn'
# 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 sampl
#     2. ' 1-2x4' becomes '1 -1 -3 -5 -7' - series starts at 1, then 4 further sa
#     3. ' 1x4' becomes '1 1 1 1 1' - shorthand for '1+0x4', series starts at 1,
#     4. ' 1 _x3 stale' becomes '1 _ _ _ stale' - the missing sample cannot incre
#
# Native histogram notation:
#     Native histograms can be used instead of floating point numbers using the f
#     {{schema:1 sum:-0.3 count:3.1 z_bucket:7.1 z_bucket_w:0.05 buckets:[5.1 10
#     Native histograms support the same expanding notation as floating point num
#     All properties are optional and default to 0. The order is not important. T
#     - 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
#     - 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_
#         Otherwise, the zero bucket only contains observations that are exactly
#     - buckets (list of non-negative floats):
#         Observation counts in positive buckets. Each represents an absolute cou
#     - 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 cou
#     - n_offset (int):
#         The starting index of the first entry in the negative buckets.
values: <string>
```

`<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' emp
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 `alertn
exp_labels:
  [ <labelname>: <string> ]
exp_annotations:
  [ <labelname>: <string> ]
```

<promql_test_case>

```
# Expression to evaluate
expr: <string>

# The time elapsed from time=0s when the expression has to be evaluated.
eval_time: <duration>

# Expected samples at the given evaluation time.
exp_samples:
  [ - <sample> ]
```

<sample>

```
# Labels of the sample in usual series notation '<metric name>{<label name>=<labe
# 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
      # Series data.
      input_series:
          - series: 'up{job="prometheus", instance="localhost:9090"}'
            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' # 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 dow
      # 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:909
                  value: 50
                # Sample 2.
                - labels: 'go_goroutines{job="node_exporter",instance="localhost
                  value: 50
```

alerts.yml

```
# This is the rules file.

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 do

  - 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 do
```

> Feature flags (/docs/prometheus/latest/feature_flags/)

VISUALIZATION

INSTRUMENTING

OPERATING

ALERT MANAGER

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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 (https://en.wikipedia.org/wiki/YAML), 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 (https://github.com/prometheus/prometheus/blob/release-2.54/documentation/examples/web-config.yml).

```
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
  #
  # NOTE: If you want to enable client authentication, you need to use
  # RequireAndVerifyClientCert. Other values are insecure.
  [ client_auth_type: <string> | default = "NoClientCert" ]

  # 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.
  # This can not be changed on the fly.
  [ http2: <boolean> | default = true ]
  # List of headers that can be added to HTTP responses.
  [ 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-Op
    [ 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-
    [ 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> ... ]
```

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

Feature flags (/docs/prometheus/latest/feature_flags/)

VISUALIZATION

INSTRUMENTING

OPERATING

ALERT MANAGER

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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. The result of an expression can either be shown as a graph, viewed as tabular data in Prometheus's expression browser, or consumed by external systems via the HTTP API (../api/).

Examples

This document is a Prometheus basic language reference. For learning, it may be easier to start with a couple of examples (../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
- **String** - a simple string value; currently unused

Depending on the use-case (e.g. when graphing vs. displaying the output of an expression), only some of these types are legal as the result of a user-specified expression. For example, an expression that returns an instant vector is the only type which can be graphed.

Notes about the experimental native histograms:

- Ingesting native histograms has to be enabled via a feature flag (../../../feature_flags/#native-histograms).
- 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 (https://golang.org/ref/spec#String_literals). 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

Scalar float values can be written as literal integer or floating-point numbers in the format (whitespace only included for better readability):

```
[-+]?(
      [0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?
    | 0[xX][0-9a-fA-F]+
    | [nN][aA][nN]
    | [iI][nN][fF]
)
```

Examples:

```
23
-2.43
3.4e-9
0x8f
-Inf
NaN
```

As of version 2.54, float literals can also be represented using the syntax of time durations, where the time duration is converted into a float value corresponding to the number of seconds the time duration represents. This is an experimental feature and might still change.

Examples:

```
1s # Equivalent to 1.0
2m # Equivalent to 120.0
1ms # Equivalent to 0.001
```

Time series selectors

Time series selectors are responsible for selecting the times series and raw or inferred sample timestamps and values.

Time series *selectors* are not to be confused with higher level concept of instant and range *queries* that can execute the time series *selectors*. A higher level instant query would evaluate the given selector at one point in time, however the range query would evaluate the selector at multiple different times in between a minimum and maximum timestamp at regular steps.

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.

This example selects all time series that have the `http_requests_total` metric name:

```
http_requests_total
```

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"}
```

and would exclude:

```
http_requests_total{environment="development"}
```

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:

```
{__name__="on"} # Good!
```

All regular expressions in Prometheus use RE2 syntax (https://github.com/google/re2/wiki/Syntax).

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 time duration is appended in square brackets (`[]`) at the end of a vector selector to specify how far back in time values should be fetched for each resulting range vector element. The range is a closed interval, i.e. samples with timestamps coinciding with either boundary of the range are still included in the selection.

In this example, we select all the values we have recorded within the last 5 minutes 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]
```

Time Durations

Time durations are specified as a number, followed immediately by one of the following units:

- `ms` - milliseconds
- `s` - seconds
- `m` - minutes
- `h` - hours
- `d` - days - assuming a day always has 24h
- `w` - weeks - assuming a week always has 7d
- `y` - years - assuming a year always has 365d[1]

[1] For days in a year, the leap day is ignored, and conversely, for a minute, a leap second is ignored.

Time durations can be combined by concatenation. Units must be ordered from the longest to the shortest. A given unit must only appear once in a time duration.

Here are some examples of valid time durations:

```
5h
1h30m
5m
10s
```

As of version 2.54, time durations can also be represented using the syntax of float literals, implying the number of seconds of the time duration. This is an experimental feature and might still change.

Examples:

```
1.0 # Equivalent to 1s
0.001 # Equivalent to 1ms
120 # Equivalent to 2m
```

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.
```

The same works for range vectors. This returns the 5-minute rate (../functions/#rate) that `http_requests_total` had a week ago:

```
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 results are the same irrespective of the order of the modifiers.

For example, these two queries will produce the same result:

```
# 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 <duration> ]`

- `<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 (../operators/) page.

Functions

Prometheus supports several functions to operate on data. These are described in detail in the expression language functions (../functions/) page.

Comments

PromQL supports line comments that start with `#`. Example:

```
# This is a comment
```

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 before this timestamp within the lookback period. The lookback period is 5 minutes by default.

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.

Avoiding slow queries and overloads

If a query needs to operate on a substantial amount of data, graphing it might time out or 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 (../../configuration/recording_rules/#recording-rules).

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.

Feature flags (/docs/prometheus/latest/feature_flags/)

VISUALIZATION

INSTRUMENTING

OPERATING

ALERT MANAGER

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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.

- One-to-one vector matches
- Many-to-one and one-to-many vector matches
- Aggregation operators
- Binary operator precedence
- Operators for native histograms

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.

Between two instant vectors, a binary arithmetic operator is applied to each entry in the left-hand side vector and its matching element in the right-hand vector. The result is propagated into the result vector with the grouping labels becoming the output label set. 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 (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 propagated into a result vector with the grouping labels becoming the output label set. If the `bool` modifier is provided, vector elements that would have been dropped instead have the value `0` and vector elements that would be kept have the value `1`, with the grouping 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-many vector matches

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
```

Example query:

```
method_code:http_errors:rate5m{code="500"} / ignoring(code) method:http_requests
```

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 ex
<vector expr> <bin-op> ignoring(<label list>) group_right(<label list>) <vector e
<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:r
```

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
{method="post", code="500"} 0.05            //   6 / 120
{method="post", code="404"} 0.175           //  21 / 120
```

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, even if their label values are identical between all elements of the vector.

`parameter` is only required for `count_values`, `quantile`, `topk`, `bottomk`, `limitk` and `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:

```
count_values("version", build_version)
```

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. `+`, `-`

4. `==`, `!=`, `<=`, `<`, `>=`, `>`
5. `and`, `unless`
6. `or`

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 (../../../feature_flags/#native-histograms). 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.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

Feature flags (/docs/prometheus/latest/feature_flags/)

VISUALIZATION

INSTRUMENTING

OPERATING

ALERT MANAGER

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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 (../../feature_flags/#native-histograms). As long as no native histograms have been ingested into the TSDB, all functions will behave as usual.

- `abs()`
- `absent()`
- `absent_over_time()`
- `ceil()`
- `changes()`
- `clamp()`
- `clamp_max()`
- `clamp_min()`
- `day_of_month()`
- `day_of_week()`
- `day_of_year()`
- `days_in_month()`
- `delta()`
- `deriv()`

- 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.

This is useful for alerting on when no time series exist for a given metric name and label combination.

- `exp()`
- `floor()`
- `histogram_avg()`
- `histogram_count()` and `histogram_sum()`
- `histogram_fraction()`
- `histogram_quantile()`
- `histogram_stddev()` and `histogram_stdvar()`
- `holt_winters()`
- `hour()`
- `idelta()`
- `increase()`
- `irate()`
- `label_join()`
- `label_replace()`
- `ln()`
- `log2()`
- `log10()`
- `minute()`
- `month()`
- `predict_linear()`
- `rate()`
- `resets()`
- `round()`
- `scalar()`
- `sgn()`
- `sort()`
- `sort_desc()`
- `sort_by_label()`
- `sort_by_label_desc()`
- `sqrt()`
- `time()`
- `timestamp()`
- `vector()`

```
absent(nonexistent{job="myjob"})
# => {job="myjob"}

absent(nonexistent{job="myjob",instance=~".*"})
# => {job="myjob"}

absent(sum(nonexistent{job="myjob"}))
# => {}
```

- `year()`
- `<aggregation>_over_time()`
- Trigonometric Functions

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(+Inf) = +Inf`
- `ceil(±0) = ±0`
- `ceil(1.49) = 2.0`
- `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()

`day_of_week(v=vector(time()) instant-vector)` returns the day of the week for each of the given times in UTC. Returned values are from 0 to 6, where 0 means Sunday etc.

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()

`deriv(v range-vector)` calculates the per-second derivative of the time series in a range vector `v`, using simple linear regression (https://en.wikipedia.org/wiki/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`.

`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_sum(rate(http_request_duration_seconds[5m]))
/
  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 histogram.

`+Inf` and `-Inf` are valid boundary values. For example, if the histogram in the expression 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 (/docs/concepts/metric_types/#histogram) or from a native histogram. (See histograms and summaries (/docs/practices/histograms) 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 (/docs/concepts/metric_types/#histogram) 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:

```
histogram_quantile(0.9, rate(http_request_duration_seconds_bucket[10m]))
```

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_buck
```

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[10
```

With native histograms, aggregating everything works as usual without any `by` clause:

```
histogram_quantile(0.9, sum(rate(http_request_duration_seconds[10m])))
```

The `histogram_quantile()` function interpolates quantile values by assuming a linear distribution within a bucket.

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 following is only relevant for classic histograms: If `b` contains fewer than two buckets, `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. A 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.

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.

`histogram_stddev()` and `histogram_stdvar()`

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_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.

`holt_winters()`

`holt_winters(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.

`holt_winters` 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.

`increase()`

`increase(v range-vector)` calculates the increase in the time series in the range vector. 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.

`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 (../operators/#aggregation-operators) (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.

label_join()

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", "
```

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 (https://github.com/google/re2/wiki/Syntax) `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<name>.*):(?P<version>.*)")`

ln()

`ln(v instant-vector)` calculates the natural logarithm for all elements in `v`. Special cases 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 (https://en.wikipedia.org/wiki/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.

rate()

`rate(v range-vector)` calculates the per-second average rate of increase of the time series 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.

round()

`round(v instant-vector, to_nearest=1 scalar)` rounds the sample values of all elements 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 (../feature_flags/) `--enable-feature=promql-experimental-functions`.

`sort_by_label(v instant-vector, label string, ...)` returns vector elements sorted by their label values and sample value in case of label values being equal, in ascending 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 (https://en.wikipedia.org/wiki/Natural_sort_order).

`sort_by_label_desc()`

This function has to be enabled via the feature flag (../feature_flags/) `--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 (https://en.wikipedia.org/wiki/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.

year()

`year(v=vector(time()) instant-vector)` returns the year for each of the given times in 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 (../feature_flags/) `--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:

- `acos(v instant-vector)` : calculates the arccosine of all elements in `v` (special cases (https://pkg.go.dev/math#Acos)).
- `acosh(v instant-vector)` : calculates the inverse hyperbolic cosine of all elements in `v` (special cases (https://pkg.go.dev/math#Acosh)).
- `asin(v instant-vector)` : calculates the arcsine of all elements in `v` (special cases (https://pkg.go.dev/math#Asin)).
- `asinh(v instant-vector)` : calculates the inverse hyperbolic sine of all elements in `v` (special cases (https://pkg.go.dev/math#Asinh)).
- `atan(v instant-vector)` : calculates the arctangent of all elements in `v` (special cases (https://pkg.go.dev/math#Atan)).
- `atanh(v instant-vector)` : calculates the inverse hyperbolic tangent of all elements in `v` (special cases (https://pkg.go.dev/math#Atanh)).
- `cos(v instant-vector)` : calculates the cosine of all elements in `v` (special cases (https://pkg.go.dev/math#Cos)).
- `cosh(v instant-vector)` : calculates the hyperbolic cosine of all elements in `v` (special cases (https://pkg.go.dev/math#Cosh)).
- `sin(v instant-vector)` : calculates the sine of all elements in `v` (special cases (https://pkg.go.dev/math#Sin)).
- `sinh(v instant-vector)` : calculates the hyperbolic sine of all elements in `v` (special cases (https://pkg.go.dev/math#Sinh)).
- `tan(v instant-vector)` : calculates the tangent of all elements in `v` (special cases (https://pkg.go.dev/math#Tan)).
- `tanh(v instant-vector)` : calculates the hyperbolic tangent of all elements in `v` (special cases (https://pkg.go.dev/math#Tanh)).

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` .

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

Feature flags (/docs/prometheus/latest/feature_flags/)

VISUALIZATION

INSTRUMENTING

OPERATING

ALERT MANAGER

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

QUERY EXAMPLES

Simple time series selection

Return all time series with the metric `http_requests_total`:

```
http_requests_total
```

- Simple time series selection
- Subquery
- Using functions, operators, etc.

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 (../basics/#range-vector-selectors):

```
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"}
```

All regular expressions in Prometheus use RE2 syntax (https://github.com/google/re2/wiki/Syntax).

To select all HTTP status codes except 4xx ones, you could run:

```
http_requests_total{status!~"4.."}
```

Subquery

Return the 5-minute rate (../../functions/#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
```

The same expression, but summed by application, could be written like this:

```
sum by (app, proc) (
  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="clus
instance_cpu_time_ns{app="elephant", proc="worker", rev="34d0f99", env="prod", jo
instance_cpu_time_ns{app="turtle", proc="api", rev="4d3a513", env="prod", job="cl
instance_cpu_time_ns{app="fox", proc="widget", rev="4d3a513", env="prod", job="cl
...
```

...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)
```

Feature flags (/docs/prometheus/latest/feature_flags/)

VISUALIZATION

INSTRUMENTING

OPERATING

ALERT MANAGER

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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 (https://tools.ietf.org/html/rfc4918#page-78)).
- `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:

- Expression query result formats
  - Range vectors
  - Instant vectors
  - Scalars
  - Strings
  - Native histograms
- Targets
- Rules
- Alerts
- Querying target metadata
- Querying metric metadata
- Alertmanagers
- Status
  - Config
  - Flags
  - Runtime Information
  - Build Information
  - TSDB Stats
  - WAL Replay Stats
- TSDB Admin APIs
  - Snapshot
  - Delete Series
  - Clean Tombstones
- Remote Write Receiver
- OTLP Receiver

```
{
  "status": "success" | "error",
  "data": <data>,

  // Only set if status is "error". The data field may still hold
  // additional data.
  "errorType": "<string>",
  "error": "<string>",

  // 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 annnotations while executing the request.
  "infos": ["<string>"]
}
```

Generic placeholders are defined as follows:

- `<rfc3339 | unix_timestamp>` : Input timestamps may be provided either in RFC3339 (https://www.ietf.org/rfc/rfc3339.txt) 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 (../basics/#time-series-selectors) like `http_requests_total` or `http_requests_total{method=~"(GET|POST)"}` and need to be URL-encoded.
- `<duration>` : Prometheus duration strings (../basics/#time-durations). 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.
- `timeout=<duration>` : Evaluation timeout. Optional. Defaults to and is capped by the value of the `-query.timeout` flag.

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"
            },
            "value" : [ 1435781451.781, "0" ]
         }
      ]
   }
}
```

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.

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.

```
$ curl 'http://localhost:9090/api/v1/query_range?query=up&start=2015-07-01T20:10
{
   "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
```

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.

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"
}
```

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.

The following example returns all series that match either of the selectors `up` or `process_start_time_seconds{job="prometheus"}` :

```
$ curl -g 'http://localhost:9090/api/v1/series?' --data-urlencode 'match[]=up' --
{
   "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.

Here is an example.

```
$ curl 'localhost:9090/api/v1/labels'
{
    "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.
- `match[]=<series_selector>` : Repeated series selector argument that selects the series from which to read the label values. 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 values.

This example queries for all label values for the `job` label:

```
$ curl http://localhost:9090/api/v1/label/job/values
{
   "status" : "success",
   "data" : [
      "node",
      "prometheus"
   ]
}
```

Querying exemplars

This is **experimental** and might change in the future. The following endpoint returns a list of exemplars for a valid PromQL query for a specific time range:

```
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_metri
{
    "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": {
                        "trace_id": "Olp9XHlq763ccsfa"
                    },
                    "value": "19",
                    "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` (../functions/#sort) and `sort_by_label` (../functions/#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` (../functions/#sort) or `sort_by_label` (../functions/#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:

```
[ <unix_time>, "<string_value>" ]
```

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_
}
```

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.

```
$ curl http://localhost:9090/api/v1/targets
{
  "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"
        },
      }
    ]
  }
}
```

The `state` query parameter allows the caller to filter by active or dropped targets, (e.g., `state=active`, `state=dropped`, `state=any`). Note that an empty array is still returned for targets that are filtered out. Other values are ignored.

```
$ 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",
          "__metrics_path__": "/metrics",
          "__scheme__": "http",
          "job": "node_exporter"
        },
        "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.
- `rule_group[]=<string>` : only return rules with the given rule group name. If the parameter is repeated, rules with any of the provided rule group names are 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.

```
$ 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": {
                                    "alertname": "HighRequestLatency",
                                    "severity": "page"
                                },
                                "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.

```
GET /api/v1/alerts
```

```
$ 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"` .

```
curl -G http://localhost:9091/api/v1/targets/metadata \
    --data-urlencode 'metric=go_goroutines' \
    --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"
      },
      "metric": "prometheus_treecache_zookeeper_failures_total",
      "type": "counter",
      "help": "The total number of ZooKeeper failures.",
      "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.

```
curl -G http://localhost:9090/api/v1/metadata?limit=2

{
  "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": [
      {
        "type": "counter",
        "help": "Number of HTTP requests",
        "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.

```
$ curl http://localhost:9090/api/v1/alertmanagers
{
  "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>",
  }
}
```

Flags

The following endpoint returns flag values that Prometheus was configured with:

```
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": "/",
    "reloadConfigSuccess": true,
    "lastConfigTime": "2019-11-02T17:23:59+01:00",
    "timeSeriesCount": 873,
    "corruptionCount": 0,
    "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"
  }
}
```

NOTE: The exact returned build properties may change without notice between Prometheus versions.

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,
      "minTime": 1591516800000,
      "maxTime": 1598896800143,
    },
    "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
      }
    ]
  }
}
```

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

These are APIs that expose database functionalities for the advanced user. These APIs are not enabled unless the `--web.enable-admin-api` is set.

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:

- `match[]=<series_selector>` : Repeated label matcher argument that selects the series to delete. At least one `match[]` argument must be provided.
- `start=<rfc3339 | unix_timestamp>` : Start timestamp. Optional and defaults to 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[]=pr
```

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

Remote Write Receiver

Prometheus can be configured as a receiver for the Prometheus remote write 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 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 (../../storage/#overview).

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 the feature flag `--enable-feature=otlp-write-receiver`. When enabled, the OTLP receiver endpoint is `/api/v1/otlp/v1/metrics`.

New in v2.47

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.
>
> - Samples
> - Streamed Chunks

This API provides data read functionality from Prometheus. This interface expects snappy (https://github.com/google/snappy) compression. The API definition is located here (https://github.com/prometheus/prometheus/blob/master/prompb/remote.proto).

Request are made to the following endpoint. `/api/v1/read`

Samples

This returns a message that includes a list of raw samples.

Streamed Chunks

These streamed chunks utilize an XOR algorithm inspired by the Gorilla (http://www.vldb.org/pvldb/vol8/p1816-teller.pdf) compression to encode the chunks. However, it provides resolution to the millisecond instead of to the second.

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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:

```
./data
├── 01BKGV7JBM69T2G1BGBGM6KB12
│   └── 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 (../querying/api/#snapshot) 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 (/docs/operating/integrations/#remote-endpoints-and-storage). 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 (https://github.com/prometheus/prometheus/blob/release-2.54/tsdb/docs/format/README.md).

Compaction

The initial two-hour blocks are eventually compacted into longer blocks in the background.

Compaction will create larger blocks containing data spanning up to 10% of the retention time, or 31 days, whichever is smaller.

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. When this flag is set, it overrides `storage.tsdb.retention` . If neither this flag nor `storage.tsdb.retention` 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.retention` : Deprecated in favor of `storage.tsdb.retention.time` .
- `--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_
```

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 for whatever reason, the best strategy to address the problem is to shut down Prometheus then remove the entire storage directory. You can also try removing individual block directories, or the WAL directory to resolve the problem. Note that this means losing approximately two hours data per block directory. Again, Prometheus's local storage is not intended to be durable long-term storage; external solutions offer extended retention and data durability.

CAUTION: Non-POSIX compliant filesystems are not supported for Prometheus' local storage as unrecoverable corruptions may happen. NFS filesystems (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 entires will be removed prior to hitting any disk limitations.

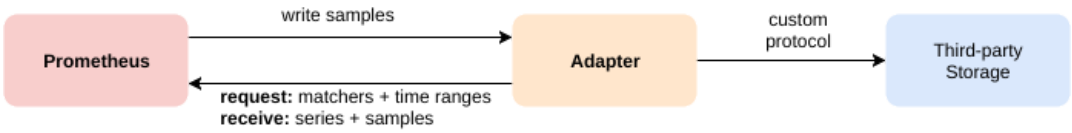
Remote storage integrations

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 three ways:

- Prometheus can write samples that it ingests to a remote URL in a standardized format.
- Prometheus can receive samples from other Prometheus servers in a standardized format.
- Prometheus can read (back) sample data from a remote URL in a standardized format.



The read and write protocols both use a snappy-compressed protocol buffer encoding over HTTP. The protocols are not considered as stable APIs yet and may change to use gRPC over HTTP/2 in the future, when all hops between Prometheus and the remote storage can safely be assumed to support HTTP/2.

For details on configuring remote storage integrations in Prometheus, see the remote write (../configuration/configuration/#remote_write) and remote read (../configuration/configuration/#remote_read) sections of the Prometheus configuration documentation.

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`.

For details on the request and response messages, see the remote storage protocol buffer definitions (https://github.com/prometheus/prometheus/blob/main/prompb/remote.proto).

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.

Existing integrations

To learn more about existing integrations with remote storage systems, see the Integrations documentation (/docs/operating/integrations/#remote-endpoints-and-storage).

Backfilling from OpenMetrics format

Overview

If a user wants to create blocks into the TSDB from data that is in OpenMetrics (https://openmetrics.io/) 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.

A typical use case is to migrate metrics data from a different monitoring system or time-series database to Prometheus. To do so, the user must first convert the source data into OpenMetrics (https://openmetrics.io/) format, which is the input format for the backfilling as described below.

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

Overview

When a new recording rule is created, there is no historical data for it. Recording rule data 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 (/docs/prometheus/latest/configuration/recording_rules/).

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).

improve it by filing issues or pull requests.

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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 collect and store only aggregated data (job-level drill-down) from those local servers. This provides an aggregate global view and detailed local views.

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 (../querying/basics/#instant-vector-selectors) 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:

```
scrape_configs:
  - job_name: 'federate'
    scrape_interval: 15s

    honor_labels: true
    metrics_path: '/federate'

    params:
      'match[]':
        - '{job="prometheus"}'
        - '{__name__=~"job:.*"}'

    static_configs:
      - targets:
        - 'source-prometheus-1:9090'
        - 'source-prometheus-2:9090'
        - 'source-prometheus-3:9090'
```

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

WRITING HTTP SERVICE DISCOVERY

Prometheus provides a generic HTTP Service Discovery

- Comparison between File-Based SD and HTTP SD
- Requirements of HTTP SD endpoints
- HTTP_SD format

(/docs/prometheus/latest/configuration/configuration/#http_sd_config), that enables it to discover targets over an HTTP endpoint.

The HTTP Service Discovery is complimentary to the supported service discovery mechanisms, and is an alternative to File-based Service Discovery (/docs/guides/file-sd/#use-file-based-service-discovery-to-discover-scrape-targets).

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

Item	File SD	HTTP SD
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.

The SD endpoint must answer with an HTTP 200 response, with the HTTP Header `Content-Type: application/json`. The answer must be UTF-8 formatted. If no targets should be transmitted, HTTP 200 must also be emitted, with an empty list `[]`. Target lists are 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", "10.0.1
        "labels": {
            "__meta_datacenter": "london",
            "__meta_prometheus_job": "node"
        }
    },
    {
        "targets": ["10.0.40.2:9100", "10.0.40.3:9100"],
        "labels": {
            "__meta_datacenter": "london",
            "__meta_prometheus_job": "alertmanager"
        }
    },
    {
        "targets": ["10.0.40.2:9093", "10.0.40.3:9093"],
        "labels": {
            "__meta_datacenter": "newyork",
            "__meta_prometheus_job": "alertmanager"
        }
    }
]
```

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

MANAGEMENT API

Prometheus provides a set of management APIs to facilitate automation and integration.

- Health check
- Readiness check
- Reload
- Quit

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.

Alternatively, a configuration reload can be triggered by sending a `SIGHUP` to the Prometheus process.

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.

PROMETHEUS

The Prometheus monitoring server

Flags

Flag	Description
`-h , --help`	Show context-sensitive help (also try --help-long and --help-man).
`--version`	Show application version.
`--config.file`	Prometheus configuration file path.
`--web.listen-address`	Address to listen on for UI, API, and telemetry.
`--auto-gomemlimit.ratio`	The ratio of reserved GOMEMLIMIT memory to the detected maximum container or system memory
`--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.
`--web.max-connections`	Maximum number of simultaneous connections.
`--web.external-url`	The URL under which Prometheus is externally reachable (for 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.
`--web.enable-admin-api`	Enable API endpoints for admin control actions.
`--web.enable-remote-write-receiver`	Enable API endpoint accepting remote write requests.
`--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
`--web.console.templates`	Path to the console template directory, available at /consoles.
`--web.console.libraries`	Path to the console library directory.
`--web.page-title`	Document title of Prometheus instance.
`--web.cors.origin`	Regex for CORS origin. It is fully anchored. Example: 'https?://(domain1\
`--storage.tsdb.path`	Base path for metrics storage. Use with server mode only.

Flag	Description
`--storage.tsdb.retention`	[DEPRECATED] How long to retain samples in storage. This flag has been deprecated, use "storage.tsdb.retention.time" instead. Use with server mode only.
`--storage.tsdb.retention.time`	How long to retain samples in storage. When this flag is set it overrides "storage.tsdb.retention". If neither this flag nor "storage.tsdb.retention" 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.
`--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.
`--storage.agent.path`	Base path for metrics storage. Use with agent mode only.
`--storage.agent.wal-compression`	Compress the agent WAL. Use with agent mode only.
`--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.
`--storage.remote.flush-deadline`	How long to wait flushing sample on shutdown or config reload.
`--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.
`--storage.remote.read-concurrent-limit`	Maximum number of concurrent remote read calls. 0 means no limit. Use with server mode only.
`--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.
`--rules.alert.for-outage-tolerance`	Max time to tolerate prometheus outage for restoring "for" state of alert. Use with server mode only.

Flag	Description
`--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.
`--rules.alert.resend-delay`	Minimum amount of time to wait before resending an alert to Alertmanager. Use with server mode only.
`--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.
`--alertmanager.notification-queue-capacity`	The capacity of the queue for pending Alertmanager notifications. Use with server mode only.
`--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.
`--query.lookback-delta`	The maximum lookback duration for retrieving metrics during expression evaluations and federation. Use with server mode only.
`--query.timeout`	Maximum time a query may take before being aborted. Use with server mode only.
`--query.max-concurrency`	Maximum number of queries executed concurrently. Use with server mode only.
`--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.
`--enable-feature`	Comma separated feature names to enable. Valid options: agent, auto-gomemlimit, exemplar-storage, expand-external-labels, memory-snapshot-on-shutdown, promql-per-step-stats, promql-experimental-functions, remote-write-receiver (DEPRECATED), extra-scrape-metrics, new-service-discovery-manager, auto-gomaxprocs, no-default-scrape-port, native-histograms, otlp-write-receiver, created-timestamp-zero-ingestion, concurrent-rule-eval. See https://prometheus.io/docs/prometheus/latest/feature_flags/ (/docs/prometheus/latest/feature_flags/) for more details.
`--log.level`	Only log messages with the given severity or above. One of: [debug, info, warn, error]
`--log.format`	Output format of log messages. One of: [logfmt, json]

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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 (only PromQL related and no-default-scrape-port). See https://prometheus.io/docs/prometheus/latest/feature_flags/ (/docs/prometheus/latest/feature_flags/) for the options and more details.

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.
promql	PromQL formatting and editing. Requires the --experimental flag.

`promtool help`

Show help.

Arguments

Argument	Description
command	Show help on command.

`promtool check`

Check the resources for validity.

Flags

Flag	Description
`--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

Argument	Description	Required
config-file	The prometheus config file.	Yes
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 specified in the config. Available options are: all, duplicate-rules, none. Use --lint=none to disable linting	`duplicate-rules`
`--lint-fatal`	Make lint errors exit with exit code 3.	`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

Argument	Description	Required
web-config-files	The config files to check.	Yes

`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`
`--lint-fatal`	Make lint errors exit with exit code 3.	`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 (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.

Flags

Flag	Description
`--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

Flag	Description
`--match`	Series selector. Can be specified multiple times.
`--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

Flag	Description	Default
`--server`	Prometheus server to query.	
`--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

Argument	Description	Required
server	Prometheus server to get all debug information from.	Yes

`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`

Flag	Description	Default
`--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.

```
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.	
`--diff`	[Experimental] Print colored differential output between expected & received output.	`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.

Flags

Flag	Description	Default
`--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.

Flags

Flag	Description
`-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 would be created in case WAL replay generates chunks. The sandbox directory is cleaned up at the end.	`data/`
`--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.

Flags

Flag	Description	Default
`--sandbox-dir-root`	Root directory where a sandbox directory would be created in case WAL replay generates chunks. The sandbox directory is cleaned up at the end.	`data/`
`--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.

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

Argument	Description	Required
rule-files	A list of one or more files containing recording rules to be backfilled. All 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
value	Value 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

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

PROMETHEUS 2.0 MIGRATION GUIDE

In line with our stability promise (/blog/2016/07/18/prometheus-1-0-released/#fine-print), the Prometheus 2.0 release contains a number of backwards incompatible changes. This document offers guidance on migrating from Prometheus 1.8 to Prometheus 2.0 and newer versions.

Flags

The format of Prometheus command line flags has changed. Instead of a single dash, all flags now use a double dash. Common flags (`--config.file`, `--web.listen-address` and `--web.external-url`) remain but almost all storage-related flags have been removed.

Some notable flags which have been removed:

- `-alertmanager.url` In Prometheus 2.0, the command line flags for configuring a static Alertmanager URL have been removed. Alertmanager must now be discovered via service discovery, see Alertmanager service discovery.
- `-log.format` In Prometheus 2.0 logs can only be streamed to standard error.

- `-query.staleness-delta` has been renamed to `--query.lookback-delta`; Prometheus 2.0 introduces a new mechanism for handling staleness, see staleness (../querying/basics/#staleness).
- `-storage.local.*` Prometheus 2.0 introduces a new storage engine; as such all flags relating to the old engine have been removed. For information on the new engine, see Storage.
- `-storage.remote.*` Prometheus 2.0 has removed the deprecated remote storage flags, and will fail to start if they are supplied. To write to InfluxDB, Graphite, or OpenTSDB use the relevant storage adapter.

Alertmanager service discovery

Alertmanager service discovery was introduced in Prometheus 1.4, allowing Prometheus to dynamically discover Alertmanager replicas using the same mechanism as scrape targets. In Prometheus 2.0, the command line flags for static Alertmanager config have been removed, so the following command line flag:

```
./prometheus -alertmanager.url=http://alertmanager:9093/
```

Would be replaced with the following in the `prometheus.yml` config file:

```
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      - alertmanager:9093
```

You can also use all the usual Prometheus service discovery integrations and relabeling in your Alertmanager configuration. This snippet instructs Prometheus to search for Kubernetes pods, in the `default` namespace, with the label `name: alertmanager` and with a non-empty port.

```
alerting:
  alertmanagers:
  - kubernetes_sd_configs:
      - role: pod
    tls_config:
      ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
    relabel_configs:
    - source_labels: [__meta_kubernetes_pod_label_name]
      regex: alertmanager
      action: keep
    - source_labels: [__meta_kubernetes_namespace]
      regex: default
      action: keep
    - source_labels: [__meta_kubernetes_pod_container_port_number]
      regex:
      action: drop
```

Recording rules and alerts

The format for configuring alerting and recording rules has been changed to YAML. An example of a recording rule and alert in the old format:

```
job:request_duration_seconds:histogram_quantile99 =
  histogram_quantile(0.99, sum by (le, job) (rate(request_duration_seconds_bucket

ALERT FrontendRequestLatency
  IF job:request_duration_seconds:histogram_quantile99{job="frontend"} > 0.1
  FOR 5m
  ANNOTATIONS {
    summary = "High frontend request latency",
  }
```

Would look like this:

```
groups:
- name: example.rules
  rules:
  - record: job:request_duration_seconds:histogram_quantile99
    expr: histogram_quantile(0.99, sum by (le, job) (rate(request_duration_secon
  - alert: FrontendRequestLatency
    expr: job:request_duration_seconds:histogram_quantile99{job="frontend"} > 0.1
    for: 5m
    annotations:
      summary: High frontend request latency
```

To help with the change, the `promtool` tool has a mode to automate the rules conversion. Given a `.rules` file, it will output a `.rules.yml` file in the new format. For example:

```
$ promtool update rules example.rules
```

You will need to use `promtool` from Prometheus 2.5 (https://github.com/prometheus/prometheus/releases/tag/v2.5.0) as later versions no longer contain the above subcommand.

Storage

The data format in Prometheus 2.0 has completely changed and is not backwards compatible with 1.8 and older versions. To retain access to your historic monitoring data we recommend you run a non-scraping Prometheus instance running at least version 1.8.1 in parallel with your Prometheus 2.0 instance, and have the new server read existing data from the old one via the remote read protocol.

Your Prometheus 1.8 instance should be started with the following flags and an config file containing only the `external_labels` setting (if any):

```
$ ./prometheus-1.8.1.linux-amd64/prometheus -web.listen-address ":9094" -config.f
```

Prometheus 2.0 can then be started (on the same machine) with the following flags:

```
$ ./prometheus-2.0.0.linux-amd64/prometheus --config.file prometheus.yml
```

Where `prometheus.yml` contains in addition to your full existing configuration, the stanza:

```
remote_read:
  - url: "http://localhost:9094/api/v1/read"
```

PromQL

The following features have been removed from PromQL:

- `drop_common_labels` function - the `without` aggregation modifier should be used instead.
- `keep_common` aggregation modifier - the `by` modifier should be used instead.
- `count_scalar` function - use cases are better handled by `absent()` or correct propagation of labels in operations.

See issue #3060 (https://github.com/prometheus/prometheus/issues/3060) for more details.

Miscellaneous

Prometheus non-root user

The Prometheus Docker image is now built to run Prometheus as a non-root user (https://github.com/prometheus/prometheus/pull/2859). If you want the Prometheus UI/API to listen on a low port number (say, port 80), you'll need to override it. For Kubernetes, you would use the following YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 0
...
```

See Configure a Security Context for a Pod or Container (https://kubernetes.io/docs/tasks/configure-pod-container/security-context/) for more details.

If you're using Docker, then the following snippet would be used:

```
docker run -p 9090:9090 prom/prometheus:latest
```

Prometheus lifecycle

If you use the Prometheus `/-/reload` HTTP endpoint to automatically reload your Prometheus config when it changes (../configuration/configuration/), these endpoints are disabled by default for security reasons in Prometheus 2.0. To enable them, set the `--web.enable-lifecycle` flag.

BEST PRACTICES

GUIDES

TUTORIALS

SPECIFICATIONS

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 2.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 (/docs/concepts/remote_write_spec/).

Things considered unstable for 2.x:

- Any feature listed as experimental or subject to change, including:
  - The `holt_winters` PromQL function (https://github.com/prometheus/prometheus/issues/2458)
  - Remote write receiving, remote read and the remote read endpoint
- Server-side HTTPS and basic authentication
- Service discovery integrations, with the exception of `static_configs` and `file_sd_configs`

- 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
- Exact on-disk format. Potential changes however, will be forward compatible and transparently handled by Prometheus
- The format of the logs

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.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

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

- Expand environment variables in external labels
- Remote Write Receiver
- Exemplars storage
- Memory snapshot on shutdown
- Extra scrape metrics
- New service discovery manager
- Prometheus agent
- Per-step stats
- Auto GOMAXPROCS
- Auto GOMEMLIMIT
- No default scrape port
- Native Histograms
- OTLP Receiver
- Experimental PromQL functions
- Created Timestamps Zero Injection
- Concurrent evaluation of independent rules
- Metadata WAL Records

(https://github.com/prometheus/prometheus/blob/main/CHANGELOG.md).

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.

Expand environment variables in external labels

```
--enable-feature=expand-external-labels
```

Replace `${var}` or `$var` in the `external_labels` (../configuration/configuration/#configuration-file) values 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 `$$`.

Remote Write Receiver

```
--enable-feature=remote-write-receiver
```

The remote write receiver allows Prometheus to accept remote write requests from other Prometheus servers. More details can be found here (../storage/#overview).

Activating the remote write receiver via a feature flag is deprecated. Use `--web.enable-remote-write-receiver` instead. This feature flag will be ignored in future versions of Prometheus.

Exemplars storage

```
--enable-feature=exemplar-storage
```

OpenMetrics (https://github.com/OpenObservability/OpenMetrics/blob/main/specification/OpenMetrics.md#exemplars) 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 (../configuration/configuration/#configuration-file)/exemplars (../configuration/configuration/#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 the 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 state can be restored with this snapshot and m-mapped chunks without the need of WAL replay.

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`.

New service discovery manager

```
--enable-feature=new-service-discovery-manager
```

When enabled, Prometheus uses a new service discovery manager that does not restart unchanged discoveries upon reloading. This makes reloads faster and reduces pressure on service discoveries' sources.

Users are encouraged to test the new service discovery manager and report any issues upstream.

In future releases, this new service discovery manager will become the default and this feature flag will be ignored.

Prometheus agent

```
--enable-feature=agent
```

When enabled, Prometheus runs in agent mode. The agent mode is limited to discovery, scrape and remote write.

This is useful when you do not need to query the Prometheus data locally, but only from a central remote endpoint (/docs/operating/integrations/#remote-endpoints-and-storage).

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.

Auto GOMAXPROCS

```
--enable-feature=auto-gomaxprocs
```

When enabled, GOMAXPROCS variable is automatically set to match Linux container CPU quota.

Auto GOMEMLIMIT

```
--enable-feature=auto-gomemlimit
```

When enabled, the GOMEMLIMIT variable is automatically set to match the Linux container memory limit. If there is no container limit, or the process is running outside of containers, the system memory total is used.

There is also an additional tuning flag, `--auto-gomemlimit.ratio`, which allows controlling how much of the memory is used for Prometheus. The remainder is reserved for memory outside the process. For example, kernel page cache. Page cache is important for Prometheus TSDB query performance. The default is `0.9`, which means 90% of the memory limit will be used for Prometheus.

No default scrape port

```
--enable-feature=no-default-scrape-port
```

When enabled, the default ports for HTTP (`:80`) or HTTPS (`:443`) will *not* be added to the address used to scrape a target (the value of the `__address_` label), contrary to the default behavior. In addition, if a default HTTP or HTTPS port has already been added either in a static configuration or by a service discovery mechanism and the respective scheme is specified (`http` or `https`), that port will be removed.

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).

Native histograms are currently only supported in the traditional Prometheus protobuf exposition format. This feature flag therefore also enables a new (and also experimental) protobuf parser, through which *all* metrics are ingested (i.e. not only native histograms). 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 `scrape_classic_histograms` in the scrape job.

Note about the format of `le` and `quantile` label values:

In certain situations, the protobuf parsing changes the number formatting of the `le` labels of classic histograms and the `quantile` labels of summaries. Typically, this happens if the scraped target is instrumented with client_golang (https://github.com/prometheus/client_golang) provided that promhttp.HandlerOpts.EnableOpenMetrics (https://pkg.go.dev/github.com/prometheus/client_golang/prometheus/promhttp#HandlerOpts) is set to `false`. In such a case, integer label values are represented in the text format as such, e.g. `quantile="1"` or `le="2"`. However, the protobuf parsing changes the representation to float-like (following the OpenMetrics specification), so the examples above become `quantile="1.0"` and `le="2.0"` after ingestion into Prometheus, which changes the identity of the metric compared to what was ingested before via the text format.

The effect of this change is that alerts, recording rules and dashboards that directly reference label values as whole numbers such as `le="1"` will stop working.

Aggregation by the `le` and `quantile` labels for vectors that contain the old and new formatting will lead to unexpected results, and range vectors that span the transition between the different formatting will contain additional series. The most common use case for both is the quantile calculation via `histogram_quantile`, e.g. `histogram_quantile(0.95, sum by (le) (rate(histogram_bucket[10m])))`. The `histogram_quantile` function already tries to mitigate the effects to some extent, but there will be inaccuracies, in particular for shorter ranges that cover only a few samples.

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, to get consistently normalized label values.* Also Prometheus 3.0 is expected to enforce normalization of these label values.
- 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
```

OTLP Receiver

`--enable-feature=otlp-write-receiver`

The OTLP receiver allows Prometheus to accept OpenTelemetry (https://opentelemetry.io/) metrics writes. Prometheus is best used as a Pull based system, and staleness, `up` metric, and other Pull enabled features won't work when you push OTLP metrics.

Experimental PromQL functions

`--enable-feature=promql-experimental-functions`

Enables PromQL functions that are considered experimental and whose name or semantics could change.

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 (https://youtu.be/nWf0BfQ5EEA) 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. However, if there is no detectable relationship between rules then there is no reason to run them sequentially. When the `concurrent-rule-eval` feature flag is enabled, rules without any dependency on other rules within a rule group will be evaluated concurrently. This has 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.

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.

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 (/docs/visualization/grafana/) or Console templates (/docs/visualization/consoles/).

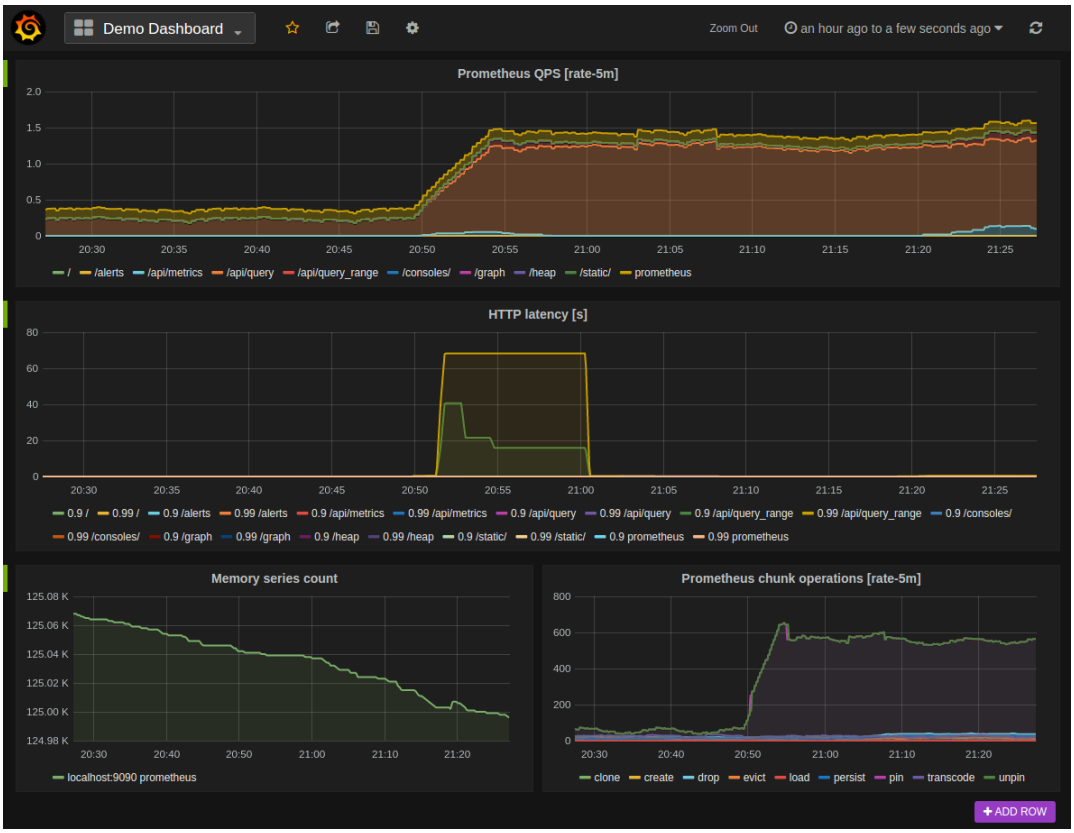
This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

GRAFANA SUPPORT FOR PROMETHEUS

Grafana (http://grafana.com/) supports querying Prometheus. 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:

- Creating a Prometheus data source
- Creating a Prometheus graph
- Importing pre-built dashboards from Grafana.com



(/assets/grafana_prometheus.png)

Installing

To install Grafana see the official Grafana documentation (https://grafana.com/grafana/download/).

Using

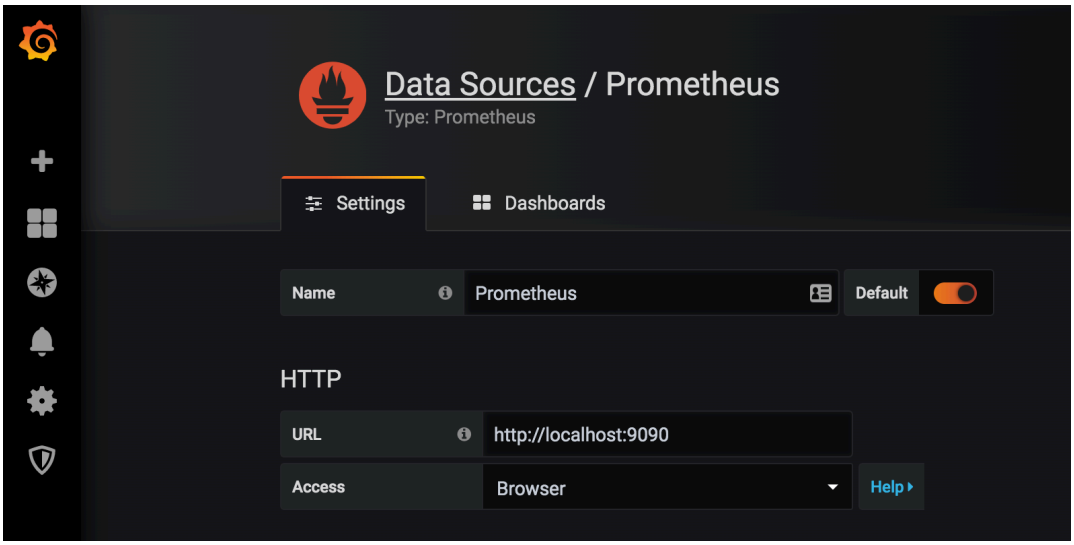
By default, Grafana will be listening on http://localhost:3000 (http://localhost:3000). The default login is "admin" / "admin".

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:



(/assets/grafana_configuring_datasource.png)

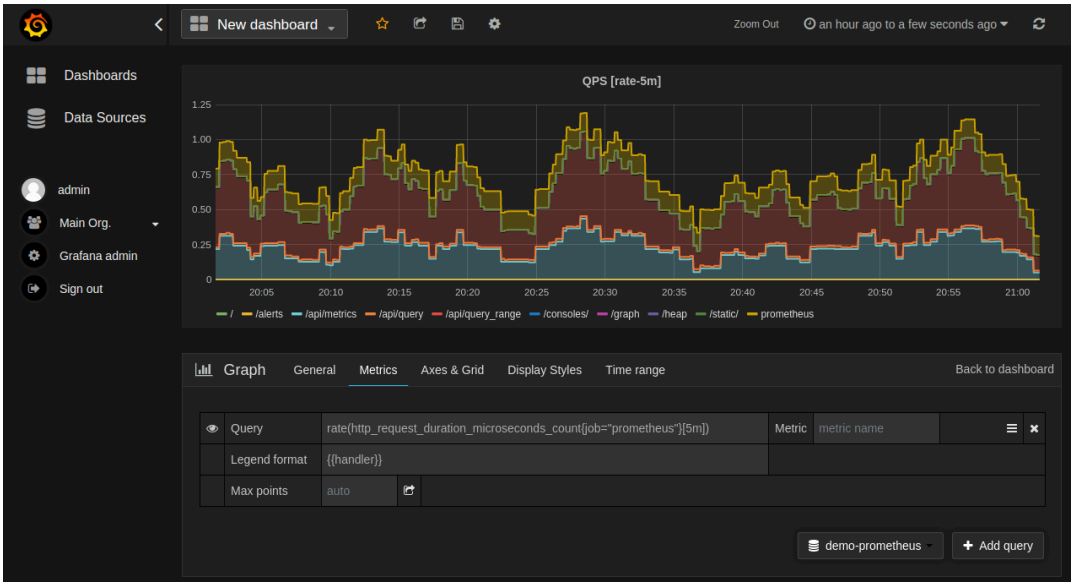
Creating a Prometheus graph

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.

4. To format the legend names of time series, use the "Legend format" input. For example, to show only the `method` and `status` labels of a returned query result, separated by a dash, you could use the legend format string `{{method}} - {{status}}`.
5. Tune other graph settings until you have a working graph.

The following shows an example Prometheus graph configuration:



(/assets/grafana_qps_graph.png)

In Grafana 7.2 and later, the `$__rate_interval` variable is recommended (https://grafana.com/docs/grafana/latest/datasources/prometheus/#using-__rate_interval) for use in the `rate` and `increase` functions.

Importing pre-built dashboards from Grafana.com

Grafana.com maintains a collection of shared dashboards (https://grafana.com/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.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

CONSOLE TEMPLATES

Console templates allow for creation of arbitrary consoles using the Go templating language (https://golang.org/pkg/text/template/). These are served from the Prometheus server.

- Getting started
- Example Console
- Graph Library

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 (/docs/visualization/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] (/docs/introduction/faq/#what-is-the-plural-of-prometheus), 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 (https://shutterstock.github.io/rickshaw/).

Finally, the table on the right can be used to display statistics in a more compact form than graphs.

Example Console

This is a basic console. It shows the number of tasks, how many of them are up, the average 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,
  yHoverFormatter: PromConsole.NumberFormatter.humanizeNoSmallPrefix,
  yUnits: "/s",
  yTitle: "Queries"
})
</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 (/docs/visualization/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 (https://en.wikipedia.org/wiki/Metric_prefix).
- `humanizeNoSmallPrefix` : For absolute values greater than 1, display the result using metric prefixes (https://en.wikipedia.org/wiki/Metric_prefix). 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 (https://github.com/prometheus/prometheus/blob/main/console_libraries/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]))"
})
</script>
```

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` .

Name	Description
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 (https://github.com/shutterstock/rickshaw/blob/master/src/js/Rickshaw.Fixtures.Color.js) 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`:

- `PromConsole.NumberFormatter.humanize`: Format using metric prefixes (https://en.wikipedia.org/wiki/Metric_prefix).
- `PromConsole.NumberFormatter.humanizeNoSmallPrefix`: For absolute values greater than 1, format using using metric prefixes (https://en.wikipedia.org/wiki/Metric_prefix). For absolute values less than 1, format 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.

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 (/docs/concepts/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 (https://github.com/prometheus/client_golang)
- Java or Scala (https://github.com/prometheus/client_java)
- Python (https://github.com/prometheus/client_python)
- Ruby (https://github.com/prometheus/client_ruby)
- Rust (https://github.com/prometheus/client_rust)

Unofficial third-party client libraries:

- Bash (https://github.com/aecolley/client_bash)
- C (https://github.com/digitalocean/prometheus-client-c)
- C++ (https://github.com/jupp0r/prometheus-cpp)
- Common Lisp (https://github.com/deadtrickster/prometheus.cl)
- Dart (https://github.com/tentaclelabs/prometheus_client)
- Delphi (https://github.com/marcobreveglieri/prometheus-client-delphi)
- Elixir (https://github.com/deadtrickster/prometheus.ex)
- Erlang (https://github.com/deadtrickster/prometheus.erl)
- Haskell (https://github.com/fimad/prometheus-haskell)
- Julia (https://github.com/fredrikekre/Prometheus.jl)
- Lua (https://github.com/knyar/nginx-lua-prometheus) for Nginx
- Lua (https://github.com/tarantool/metrics) for Tarantool
- .NET / C# (https://github.com/prometheus-net/prometheus-net)
- Node.js (https://github.com/siimon/prom-client)
- OCaml (https://github.com/mirage/prometheus)
- Perl (https://metacpan.org/pod/Net::Prometheus)
- PHP (https://github.com/promphp/prometheus_client_php)
- R (https://github.com/cfmack/pRometheus)

When Prometheus scrapes your instance's HTTP endpoint, the client library sends the current state of all tracked metrics to the server.

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 (/docs/instrumenting/exposition_formats/) yourself to expose metrics.

When implementing a new Prometheus client library, please follow the guidelines on writing client libraries (/docs/instrumenting/writing_clientlibs). Note that this document is still a work in progress. Please also consider consulting the development mailing list (https://groups.google.com/forum/#!forum/prometheus-developers). We are happy to give advice on how to make your library as useful and consistent as possible.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

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 (/docs/instrumenting/clientlibs) 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 (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.

Overall structure

Clients MUST be written to be callback based internally. Clients SHOULD generally follow the 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

The Counter, Gauge, Summary and Histogram metric types (/docs/concepts/metric_types/) are the primary interface by users.

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 (/docs/concepts/metric_types/#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.

A counter is ENCOURAGED to have:

A way to count exceptions throw/raised in a given piece of code, and optionally only certain types of exceptions. This is count_exceptions in Python.

Counters MUST start at 0.

Gauge

Gauge (/docs/concepts/metric_types/#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 (/docs/concepts/metric_types/#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 exports, though these can't be aggregated and tend to be slow. A summary MUST allow not having quantiles, as just `_count` / `_sum` is quite useful and this MUST be the default.

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 (/docs/concepts/metric_types/#histogram) 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

Providing additional functionality in metrics beyond what's documented above as makes sense for a given language is ENCOURAGED.

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 (/docs/practices/instrumentation/#use-labels) of Prometheus, but easily abused (/docs/practices/instrumentation/#do-not-overuse-labels). 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 (/docs/concepts/data_model/#metric-names-and-labels).

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 (/docs/practices/instrumentation/#avoid-missing-metrics).

Metric names

Metric names must follow the specification (/docs/concepts/data_model/#metric-names-and-labels). As with label names, this MUST be met for uses 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 (/docs/instrumenting/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

These metrics have the prefix `process_`. If obtaining a necessary value is problematic or even impossible with the used language or runtime, client libraries SHOULD prefer leaving out the corresponding metric over exporting bogus, inaccurate, or special values (like `NaN`). 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

Metric name	Help string	Unit
`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.

Packaging and dependencies

Ideally, a client library can be included in any application to add some instrumentation 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.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

PUSHING METRICS

Occasionally you will need to monitor components which cannot be scraped. The Prometheus Pushgateway (https://github.com/prometheus/pushgateway) allows you to push time series from short-lived service-level batch jobs (/docs/practices/pushing/) 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 (https://github.com/prometheus/pushgateway/blob/master/README.md).
- For use from Java see the Pushgateway documentation (https://prometheus.github.io/client_java/exporters/pushgateway/).
- For use from Go see the Push (https://godoc.org/github.com/prometheus/client_golang/prometheus/push#Pusher.Push) and Add (https://godoc.org/github.com/prometheus/client_golang/prometheus/push#Pusher.Add) methods.
- For use from Python see Exporting to a Pushgateway (https://prometheus.github.io/client_python/exporting/pushgateway/).
- For use from Ruby see the Pushgateway documentation (https://github.com/prometheus/client_ruby#pushgateway).
    - To find out about Pushgateway support of client libraries maintained outside of the Prometheus project (/docs/instrumenting/clientlibs/), refer to their respective documentation.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

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 (https://github.com/prometheus), 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 (/docs/instrumenting/writing_exporters/). Commonly, those exporters are hosted outside of the Prometheus GitHub organization.

- Third-party exporters
  - Databases
  - Hardware related
  - Issue trackers and continuous integration
  - Messaging systems
  - Storage
  - HTTP
  - APIs
  - Logging
  - FinOps
  - Other monitoring systems
  - Miscellaneous
- Software exposing Prometheus metrics
- Other third-party utilities

The exporter default port (https://github.com/prometheus/prometheus/wiki/Default-port-allocations) 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 (https://github.com/prometheus/jmx_exporter) can export from a wide variety of JVM-based applications, for example Kafka (http://kafka.apache.org/) and Cassandra (http://cassandra.apache.org/).

Databases

- Aerospike exporter (https://github.com/aerospike/aerospike-prometheus-exporter)
- AWS RDS exporter (https://github.com/qonto/prometheus-rds-exporter)
- ClickHouse exporter (https://github.com/f1yegor/clickhouse_exporter)
- Consul exporter (https://github.com/prometheus/consul_exporter) (**official**)
- Couchbase exporter (https://github.com/blakelead/couchbase_exporter)
- CouchDB exporter (https://github.com/gesellix/couchdb-exporter)
- Druid Exporter (https://github.com/opstree/druid-exporter)
- Elasticsearch exporter (https://github.com/prometheus-community/elasticsearch_exporter)
- EventStore exporter (https://github.com/marcinbudny/eventstore_exporter)
- IoTDB exporter (https://github.com/fagnercarvalho/prometheus-iotdb-exporter)
- KDB+ exporter (https://github.com/KxSystems/prometheus-kdb-exporter)
- Memcached exporter (https://github.com/prometheus/memcached_exporter) (**official**)
- MongoDB exporter (https://github.com/percona/mongodb_exporter)
- MongoDB query exporter (https://github.com/raffis/mongodb-query-exporter)
- MongoDB Node.js Driver exporter (https://github.com/christiangalsterer/mongodb-driver-prometheus-exporter)
- MSSQL server exporter (https://github.com/awaragi/prometheus-mssql-exporter)
- MySQL router exporter (https://github.com/rluisr/mysqlrouter_exporter)
- MySQL server exporter (https://github.com/prometheus/mysqld_exporter) (**official**)
- OpenTSDB Exporter (https://github.com/cloudflare/opentsdb_exporter)
- Oracle DB Exporter (https://github.com/iamseth/oracledb_exporter)
- PgBouncer exporter (https://github.com/prometheus-community/pgbouncer_exporter)
- PostgreSQL exporter (https://github.com/prometheus-community/postgres_exporter)
- Presto exporter (https://github.com/yahoojapan/presto_exporter)
- ProxySQL exporter (https://github.com/percona/proxysql_exporter)
- RavenDB exporter (https://github.com/marcinbudny/ravendb_exporter)
- Redis exporter (https://github.com/oliver006/redis_exporter)
- RethinkDB exporter (https://github.com/oliver006/rethinkdb_exporter)
- SQL exporter (https://github.com/burningalchemist/sql_exporter)
- Tarantool metric library (https://github.com/tarantool/metrics)
- Twemproxy (https://github.com/stuartnelson3/twemproxy_exporter)

Hardware related

- apcupsd exporter (https://github.com/mdlayher/apcupsd_exporter)
- BIG-IP exporter (https://github.com/ExpressenAB/bigip_exporter)
- Bosch Sensortec BMP/BME exporter (https://github.com/David-Igou/bsbmp-exporter)
- Collins exporter (https://github.com/soundcloud/collins_exporter)
- Dell Hardware OMSA exporter (https://github.com/galexrt/dellhw_exporter)
- Disk usage exporter (https://github.com/dundee/disk_usage_exporter)
- Fortigate exporter (https://github.com/bluecmd/fortigate_exporter)
- IBM Z HMC exporter (https://github.com/zhmcclient/zhmc-prometheus-exporter)
- IoT Edison exporter (https://github.com/roman-vynar/edison_exporter)
- InfiniBand exporter (https://github.com/treydock/infiniband_exporter)
- IPMI exporter (https://github.com/soundcloud/ipmi_exporter)
- knxd exporter (https://github.com/RichiH/knxd_exporter)
- Modbus exporter (https://github.com/RichiH/modbus_exporter)
- Netgear Cable Modem Exporter (https://github.com/ickymettle/netgear_cm_exporter)
- Netgear Router exporter (https://github.com/DRuggeri/netgear_exporter)
- Network UPS Tools (NUT) exporter (https://github.com/DRuggeri/nut_exporter)

- Node/system metrics exporter (https://github.com/prometheus/node_exporter) (**official**)
- NVIDIA GPU exporter (https://github.com/mindprince/nvidia_gpu_prometheus_exporter)
- ProSAFE exporter (https://github.com/dalance/prosafe_exporter)
- SmartRAID exporter (https://gitlab.com/calestyo/prometheus-smartraid-exporter)
- Waveplus Radon Sensor Exporter (https://github.com/jeremybz/waveplus_exporter)
- Weathergoose Climate Monitor Exporter (https://github.com/branttaylor/watchdog-prometheus-exporter)
- Windows exporter (https://github.com/prometheus-community/windows_exporter)
- Intel® Optane™ Persistent Memory Controller Exporter (https://github.com/intel/ipmctl-exporter)

Issue trackers and continuous integration

- Bamboo exporter (https://github.com/AndreyVMarkelov/bamboo-prometheus-exporter)
- Bitbucket exporter (https://github.com/AndreyVMarkelov/prom-bitbucket-exporter)
- Confluence exporter (https://github.com/AndreyVMarkelov/prom-confluence-exporter)
- Jenkins exporter (https://github.com/lovoo/jenkins_exporter)
- JIRA exporter (https://github.com/AndreyVMarkelov/jira-prometheus-exporter)

Messaging systems

- Beanstalkd exporter (https://github.com/messagebird/beanstalkd_exporter)
- EMQ exporter (https://github.com/nuvo/emq_exporter)
- Gearman exporter (https://github.com/bakins/gearman-exporter)
- IBM MQ exporter (https://github.com/ibm-messaging/mq-metric-samples/tree/master/cmd/mq_prometheus)
- Kafka exporter (https://github.com/danielqsj/kafka_exporter)
- NATS exporter (https://github.com/nats-io/prometheus-nats-exporter)
- NSQ exporter (https://github.com/lovoo/nsq_exporter)
- Mirth Connect exporter (https://github.com/vynca/mirth_exporter)
- MQTT blackbox exporter (https://github.com/inovex/mqtt_blackbox_exporter)
- MQTT2Prometheus (https://github.com/hikhvar/mqtt2prometheus)
- RabbitMQ exporter (https://github.com/kbudde/rabbitmq_exporter)
- RabbitMQ Management Plugin exporter (https://github.com/deadtrickster/prometheus_rabbitmq_exporter)
- RocketMQ exporter (https://github.com/apache/rocketmq-exporter)
- Solace exporter (https://github.com/solacecommunity/solace-prometheus-exporter)

Storage

- Ceph exporter (https://github.com/digitalocean/ceph_exporter)
- Ceph RADOSGW exporter (https://github.com/blemmenes/radosgw_usage_exporter)
- Gluster exporter (https://github.com/ofesseler/gluster_exporter)
- GPFS exporter (https://github.com/treydock/gpfs_exporter)
- Hadoop HDFS FSImage exporter (https://github.com/marcelmay/hadoop-hdfs-fsimage-exporter)
- HPE CSI info metrics provider (https://scod.hpedev.io/csi_driver/metrics.html)
- HPE storage array exporter (https://hpe-storage.github.io/array-exporter/)
- Lustre exporter (https://github.com/HewlettPackard/lustre_exporter)
- NetApp E-Series exporter (https://github.com/treydock/eseries_exporter)
- Pure Storage exporter (https://github.com/PureStorage-OpenConnect/pure-exporter)
- ScaleIO exporter (https://github.com/syepes/sio2prom)
- Tivoli Storage Manager/IBM Spectrum Protect exporter (https://github.com/treydock/tsm_exporter)

HTTP

- Apache exporter (https://github.com/Lusitaniae/apache_exporter)
- HAProxy exporter (https://github.com/prometheus/haproxy_exporter) (**official**)
- Nginx metric library (https://github.com/knyar/nginx-lua-prometheus)
- Nginx VTS exporter (https://github.com/sysulq/nginx-vts-exporter)
- Passenger exporter (https://github.com/stuartnelson3/passenger_exporter)
- Squid exporter (https://github.com/boynux/squid-exporter)
- Tinyproxy exporter (https://github.com/gmm42/tinyproxy_exporter)
- Varnish exporter (https://github.com/jonnenauha/prometheus_varnish_exporter)

- WebDriver exporter (https://github.com/mattbostock/webdriver_exporter)

APIs

- AWS ECS exporter (https://github.com/slok/ecs-exporter)
- AWS Health exporter (https://github.com/Jimdo/aws-health-exporter)
- AWS SQS exporter (https://github.com/jmal98/sqs_exporter)
- Azure Health exporter (https://github.com/FXinnovation/azure-health-exporter)
- BigBlueButton (https://github.com/greenstatic/bigbluebutton-exporter)
- Cloudflare exporter (https://gitlab.com/gitlab-org/cloudflare_exporter)
- Cryptowat exporter (https://github.com/nbarrientos/cryptowat_exporter)
- DigitalOcean exporter (https://github.com/metalmatze/digitalocean_exporter)
- Docker Cloud exporter (https://github.com/infinityworks/docker-cloud-exporter)
- Docker Hub exporter (https://github.com/infinityworks/docker-hub-exporter)
- Fastly exporter (https://github.com/peterbourgon/fastly-exporter)
- GitHub exporter (https://github.com/githubexporter/github-exporter)
- Gmail exporter (https://github.com/jamesread/prometheus-gmail-exporter/)
- GraphQL exporter (https://github.com/ricardbejarano/graphql_exporter)
- InstaClustr exporter (https://github.com/fcgravalos/instaclustr_exporter)
- Mozilla Observatory exporter (https://github.com/Jimdo/observatory-exporter)
- OpenWeatherMap exporter (https://github.com/RichiH/openweathermap_exporter)
- Pagespeed exporter (https://github.com/foomo/pagespeed_exporter)
- Rancher exporter (https://github.com/infinityworks/prometheus-rancher-exporter)
- Speedtest exporter (https://github.com/nlamirault/speedtest_exporter)
- Tankerkönig API Exporter (https://github.com/lukasmalkmus/tankerkoenig_exporter)

Logging

- Fluentd exporter (https://github.com/V3ckt0r/fluentd_exporter)
- Google's mtail log data extractor (https://github.com/google/mtail)
- Grok exporter (https://github.com/fstab/grok_exporter)

FinOps

- AWS Cost Exporter (https://github.com/electrolux-oss/aws-cost-exporter)
- Azure Cost Exporter (https://github.com/electrolux-oss/azure-cost-exporter)
- Kubernetes Cost Exporter (https://github.com/electrolux-oss/kubernetes-cost-exporter)

Other monitoring systems

- Akamai Cloudmonitor exporter (https://github.com/ExpressenAB/cloudmonitor_exporter)
- Alibaba Cloudmonitor exporter (https://github.com/aylei/aliyun-exporter)
- AWS CloudWatch exporter (https://github.com/prometheus/cloudwatch_exporter) (**official**)
- Azure Monitor exporter (https://github.com/RobustPerception/azure_metrics_exporter)
- Cloud Foundry Firehose exporter (https://github.com/cloudfoundry-community/firehose_exporter)
- Collectd exporter (https://github.com/prometheus/collectd_exporter) (**official**)
- Google Stackdriver exporter (https://github.com/frodenas/stackdriver_exporter)
- Graphite exporter (https://github.com/prometheus/graphite_exporter) (**official**)
- Heka dashboard exporter (https://github.com/docker-infra/heka_exporter)
- Heka exporter (https://github.com/imgix/heka_exporter)
- Huawei Cloudeye exporter (https://github.com/huaweicloud/cloudeye-exporter)
- InfluxDB exporter (https://github.com/prometheus/influxdb_exporter) (**official**)
- ITM exporter (https://github.com/rafal-szypulka/itm_exporter)
- Java GC exporter (https://github.com/loyispa/jgc_exporter)
- JavaMelody exporter (https://github.com/fschlag/javamelody-prometheus-exporter)
- JMX exporter (https://github.com/prometheus/jmx_exporter) (**official**)
- Munin exporter (https://github.com/pvdh/munin_exporter)
- Nagios / Naemon exporter (https://github.com/Griesbacher/Iapetos)
- Neptune Apex exporter (https://github.com/dl-romero/neptune_exporter)

- New Relic exporter (https://github.com/mrf/newrelic_exporter)
- NRPE exporter (https://github.com/robustperception/nrpe_exporter)
- Osquery exporter (https://github.com/zwopir/osquery_exporter)
- OTC CloudEye exporter (https://github.com/tiagoReichert/otc-cloudeye-prometheus-exporter)
- Pingdom exporter (https://github.com/giantswarm/prometheus-pingdom-exporter)
- Promitor (Azure Monitor) (https://promitor.io)
- scollector exporter (https://github.com/tgulacsi/prometheus_scollector)
- Sensu exporter (https://github.com/reachlin/sensu_exporter)
- site24x7_exporter (https://github.com/svenstaro/site24x7_exporter)
- SNMP exporter (https://github.com/prometheus/snmp_exporter) (**official**)
- StatsD exporter (https://github.com/prometheus/statsd_exporter) (**official**)
- TencentCloud monitor exporter (https://github.com/tencentyun/tencentcloud-exporter)
- ThousandEyes exporter (https://github.com/sapcc/1000eyes_exporter)
- StatusPage exporter (https://github.com/sergeyshevch/statuspage-exporter)

Miscellaneous

- ACT Fibernet Exporter (https://git.captnemo.in/nemo/prometheus-act-exporter)
- BIND exporter (https://github.com/prometheus-community/bind_exporter)
- BIND query exporter (https://github.com/DRuggeri/bind_query_exporter)
- Bitcoind exporter (https://github.com/LePetitBloc/bitcoind-exporter)
- Blackbox exporter (https://github.com/prometheus/blackbox_exporter) (**official**)
- Bungeecord exporter (https://github.com/weihao/bungeecord-prometheus-exporter)
- BOSH exporter (https://github.com/cloudfoundry-community/bosh_exporter)
- cAdvisor (https://github.com/google/cadvisor)
- Cachet exporter (https://github.com/ContaAzul/cachet_exporter)
- ccache exporter (https://github.com/virtualtam/ccache_exporter)
- c-lightning exporter (https://github.com/lightningd/plugins/tree/master/prometheus)
- DHCPD leases exporter (https://github.com/DRuggeri/dhcpd_leases_exporter)
- Dovecot exporter (https://github.com/kumina/dovecot_exporter)
- Dnsmasq exporter (https://github.com/google/dnsmasq_exporter)
- eBPF exporter (https://github.com/cloudflare/ebpf_exporter)
- Ethereum Client exporter (https://github.com/31z4/ethereum-prometheus-exporter)
- File statistics exporter (https://github.com/michael-doubez/filestat_exporter)
- JFrog Artifactory Exporter (https://github.com/peimanja/artifactory_exporter)
- Hostapd Exporter (https://github.com/Fundacio-i2CAT/hostapd_prometheus_exporter)
- IBM Security Verify Access / Security Access Manager Exporter (https://gitlab.com/zeblawson/isva-prometheus-exporter)
- IPsec exporter (https://github.com/torilabs/ipsec-prometheus-exporter)
- IRCd exporter (https://github.com/dgl/ircd_exporter)
- Linux HA ClusterLabs exporter (https://github.com/ClusterLabs/ha_cluster_exporter)
- JMeter plugin (https://github.com/johrstrom/jmeter-prometheus-plugin)
- JSON exporter (https://github.com/prometheus-community/json_exporter)
- Kannel exporter (https://github.com/apostvav/kannel_exporter)
- Kemp LoadBalancer exporter (https://github.com/giantswarm/prometheus-kemp-exporter)
- Kibana Exporter (https://github.com/pjhampton/kibana-prometheus-exporter)
- kube-state-metrics (https://github.com/kubernetes/kube-state-metrics)
- Locust Exporter (https://github.com/ContainerSolutions/locust_exporter)
- Meteor JS web framework exporter (https://atmospherejs.com/sevki/prometheus-exporter)
- Minecraft exporter module (https://github.com/Baughn/PrometheusIntegration)
- Minecraft exporter (https://github.com/dirien/minecraft-prometheus-exporter)
- Nomad exporter (https://gitlab.com/yakshaving.art/nomad-exporter)
- nftables exporter (https://github.com/Intrinsec/nftables_exporter)
- OpenStack exporter (https://github.com/openstack-exporter/openstack-exporter)
- OpenStack blackbox exporter (https://github.com/infraly/openstack_client_exporter)
- oVirt exporter (https://github.com/czerwonk/ovirt_exporter)
- Pact Broker exporter (https://github.com/ContainerSolutions/pactbroker_exporter)

- PHP-FPM exporter (https://github.com/bakins/php-fpm-exporter)
- PowerDNS exporter (https://github.com/ledgr/powerdns_exporter)
- Podman exporter (https://github.com/containers/prometheus-podman-exporter)
- Prefect2 exporter (https://github.com/pathfinder177/prefect2-prometheus-exporter)
- Process exporter (https://github.com/ncabatoff/process-exporter)
- rTorrent exporter (https://github.com/mdlayher/rtorrent_exporter)
- Rundeck exporter (https://github.com/phsmith/rundeck_exporter)
- SABnzbd exporter (https://github.com/msroest/sabnzbd_exporter)
- SAML exporter (https://github.com/DoodleScheduling/saml-exporter)
- Script exporter (https://github.com/adhocteam/script_exporter)
- Shield exporter (https://github.com/cloudfoundry-community/shield_exporter)
- Smokeping prober (https://github.com/SuperQ/smokeping_prober)
- SMTP/Maildir MDA blackbox prober (https://github.com/cherti/mailexporter)
- SoftEther exporter (https://github.com/dalance/softether_exporter)
- SSH exporter (https://github.com/treydock/ssh_exporter)
- Teamspeak3 exporter (https://github.com/hikhvar/ts3exporter)
- Transmission exporter (https://github.com/metalmatze/transmission-exporter)
- Unbound exporter (https://github.com/kumina/unbound_exporter)
- WireGuard exporter (https://github.com/MindFlavor/prometheus_wireguard_exporter)
- Xen exporter (https://github.com/lovoo/xenstats_exporter)

When implementing a new Prometheus exporter, please follow the guidelines on writing exporters (/docs/instrumenting/writing_exporters) Please also consider consulting the development mailing list (https://groups.google.com/forum/#!forum/prometheus-developers). We are happy to give advice on how to make your exporter as useful and consistent as possible.

Software exposing Prometheus metrics

Some third-party software exposes metrics in the Prometheus format, so no separate exporters are needed:

- Ansible Automation Platform Automation Controller (AWX) (https://docs.ansible.com/automation-controller/latest/html/administration/metrics.html)
- App Connect Enterprise (https://github.com/ot4i/ace-docker)
- Ballerina (https://ballerina.io/)
- BFE (https://github.com/baidu/bfe)
- Caddy (https://caddyserver.com/docs/metrics) (**direct**)
- Ceph (https://docs.ceph.com/en/latest/mgr/prometheus/)
- CockroachDB (https://www.cockroachlabs.com/docs/stable/monitoring-and-alerting.html#prometheus-endpoint)
- Collectd (https://collectd.org/wiki/index.php/Plugin:Write_Prometheus)
- Concourse (https://concourse-ci.org/)
- CRG Roller Derby Scoreboard (https://github.com/rollerderby/scoreboard) (**direct**)
- Diffusion (https://docs.pushtechnology.com/docs/latest/manual/html/administratorguide/systemmanagement/r_statistics.html)
- Docker Daemon (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-metrics)
- Doorman (https://github.com/youtube/doorman) (**direct**)
- Dovecot (https://doc.dovecot.org/configuration_manual/stats/openmetrics/)
- Envoy (https://www.envoyproxy.io/docs/envoy/latest/operations/admin.html#get--stats?format=prometheus)
- Etcd (https://github.com/coreos/etcd) (**direct**)
- Flink (https://github.com/apache/flink)
- FreeBSD Kernel (https://www.freebsd.org/cgi/man.cgi?query=prometheus_sysctl_exporter&apropos=0&sektion=8&manpath=FreeBSD+12-current&arch=default&format=html)
- GitLab (https://docs.gitlab.com/ee/administration/monitoring/prometheus/gitlab_metrics.html)
- Grafana (https://grafana.com/docs/grafana/latest/administration/view-server/internal-metrics/)
- JavaMelody (https://github.com/javamelody/javamelody/wiki/UserGuideAdvanced#exposing-metrics-to-prometheus)
- Kong (https://github.com/Kong/kong-plugin-prometheus)

- Kubernetes (https://github.com/kubernetes/kubernetes) (**direct**)
- LavinMQ (https://lavinmq.com/)
- Linkerd (https://github.com/BuoyantIO/linkerd)
- mgmt (https://github.com/purpleidea/mgmt/blob/master/docs/prometheus.md)
- MidoNet (https://github.com/midonet/midonet)
- midonet-kubernetes (https://github.com/midonet/midonet-kubernetes) (**direct**)
- MinIO (https://docs.minio.io/docs/how-to-monitor-minio-using-prometheus.html)
- PATROL with Monitoring Studio X (https://www.sentrysoftware.com/library/swsyx/prometheus/exposing-patrol-parameters-in-prometheus.html)
- Netdata (https://github.com/firehol/netdata)
- OpenZiti (https://openziti.github.io)
- Pomerium (https://pomerium.com/reference/#metrics-address)
- Pretix (https://pretix.eu/)
- Quobyte (https://www.quobyte.com/) (**direct**)
- RabbitMQ (https://rabbitmq.com/prometheus.html)
- RobustIRC (http://robustirc.net/)
- ScyllaDB (https://github.com/scylladb/scylla)
- Skipper (https://github.com/zalando/skipper)
- SkyDNS (https://github.com/skynetservices/skydns) (**direct**)
- Telegraf (https://github.com/influxdata/telegraf/tree/master/plugins/outputs/prometheus_client)
- Traefik (https://github.com/containous/traefik)
- Vector (https://vector.dev)
- VerneMQ (https://github.com/vernemq/vernemq)
- Flux (https://github.com/fluxcd/flux2)
- Xandikos (https://www.xandikos.org/) (**direct**)
- Zipkin (https://github.com/openzipkin/zipkin/tree/master/zipkin-server#metrics)

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 (https://github.com/clj-commons/iapetos)
- Go: go-metrics instrumentation library (https://github.com/armon/go-metrics)
- Go: gokit (https://github.com/peterbourgon/gokit)
- Go: prombolt (https://github.com/mdlayher/prombolt)
- Java/JVM: EclipseLink metrics collector (https://github.com/VitaNuova/eclipselinkexporter)
- Java/JVM: Hystrix metrics publisher (https://github.com/ahus1/prometheus-hystrix)
- Java/JVM: Jersey metrics collector (https://github.com/VitaNuova/jerseyexporter)
- Java/JVM: Micrometer Prometheus Registry (https://micrometer.io/docs/registry/prometheus)
- Python-Django: django-prometheus (https://github.com/korfuri/django-prometheus)
- Node.js: swagger-stats (https://github.com/slanatech/swagger-stats)

WRITING EXPORTERS

If you are instrumenting your own code, the general rules of how to instrument code with a Prometheus client library (/docs/practices/instrumentation/) 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 (/community).

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 (https://github.com/prometheus/haproxy_exporter).

- Failed scrapes
- Landing page
- Port numbers
- Announcing

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 (https://github.com/prometheus/mysqld_exporter) is on this end of the spectrum.

The node exporter (https://github.com/prometheus/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 large setups, for example the HAProxy exporter (https://github.com/prometheus/haproxy_exporter) allows filtering of per-server stats. Similarly, there may be expensive metrics that are disabled by default.

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 (https://github.com/prometheus/cloudwatch_exporter), SNMP (https://github.com/prometheus/snmp_exporter) and collectd (https://github.com/prometheus/collectd_exporter). 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 (https://github.com/prometheus/jmx_exporter) is the worst offender here, with the Graphite (https://github.com/prometheus/graphite_exporter) and StatsD (https://github.com/prometheus/statsd_exporter) 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 (/docs/practices/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` .

Metrics must use base units (e.g. seconds, bytes) and leave converting them to something more readable to graphing tools. No matter what units you end up using, the units in the metric name must match the units in use. Similarly, expose ratios, not percentages. Even 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 (https://github.com/prometheus/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 the MySQL exporter metrics may be used by a variety of people, not just DBAs. A `HELP` string with the original name can provide most of the same benefits as using the original names.

Labels

Read the general advice (/docs/practices/instrumentation/#things-to-watch-out-for) 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
```

The former breaks for people who do a `sum()` over your metric, and the latter breaks sum and is quite difficult to work with. Some client libraries, for example Go, will actively try to stop you doing the latter in a custom collector, and all client libraries should stop you from 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 (https://github.com/prometheus/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/ (http://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.

Types

You should try to match up the types of your metrics to Prometheus types. This usually 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 (https://github.com/prometheus/haproxy_exporter) has hand-written strings. The node exporter (https://github.com/prometheus/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.

Dotted strings

Many monitoring systems don't have labels, instead doing things like `my.class.path.mymetric.labelvalue1.labelvalue2.labelvalue3`.

The Graphite (https://github.com/prometheus/graphite_exporter) and StatsD (https://github.com/prometheus/statsd_exporter) 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 (https://godoc.org/github.com/prometheus/client_golang/prometheus#MustNewConstMetric) in your `Collect()` method. For Python see https://github.com/prometheus/client_python#custom-collectors (https://prometheus.github.io/client_python/collector/custom/) and for Java generate a `List<MetricFamilySamples>` in

your collect method, see StandardExports.java (https://github.com/prometheus/client_java/blob/master/simpleclient_hotspot/src/main/java/io/prometheus/client/hotspot/ 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 (https://github.com/prometheus/blackbox_exporter) and SNMP exporter (https://github.com/prometheus/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.

Machine and process metrics

Many systems, for example Elasticsearch, expose machine metrics such as CPU, memory and filesystem information. As the node exporter (https://github.com/prometheus/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 (https://github.com/prometheus/client_java/blob/master/simpleclient_hotspot/src/main/java/io/prometheus/client/hotspot/ 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 (https://github.com/mesosphere/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 (https://github.com/prometheus/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 wanted to run some business queries against the data to then export. Having an exporter that uses your usual load balancing approach to talk to one replica is the sanest approach.

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 (https://prometheus.io/docs/instrumenting/pushing/) 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.

For service-level metrics, e.g. service-level batch jobs, you should have your exporter push into the Pushgateway and exit after the event rather than handling the state yourself. For instance-level batch metrics, there is no clear pattern yet. The options are either to abuse the 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 (/docs/concepts/jobs_instances/#automatically-generated-labels-and-time-series), 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 (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.

Announcing

Once you're ready to announce your exporter to the world, email the mailing list and send a PR to add it to the list of available exporters (https://github.com/prometheus/docs/blob/main/content/docs/instrumenting/exporters.md).

EXPOSITION FORMATS

Metrics can be exposed to Prometheus using a simple text-based exposition format. There are various client libraries (/docs/instrumenting/clientlibs/) that implement this format for you. If your preferred language doesn't have a client library you can create your own (/docs/instrumenting/writing_clientlibs/).

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	• Verbose • 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`.

The remaining lines describe samples (one per line) using the following syntax (EBNF (https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form)):

```
metric_name [
  "{" 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()` (https://golang.org/pkg/strconv/#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()` (https://golang.org/pkg/strconv/#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"}`.
- Each bucket count of a histogram named `x` is given as a separate sample line with the name `x_bucket` and a label `{le="y"}` (where `y` is the upper bound of the bucket).
- A histogram *must* have a bucket with `{le="+Inf"}`. Its value *must* be identical to the 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\"FILE.TXT\""} 1.45825591

# 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="0.01"} 3102
rpc_duration_seconds{quantile="0.05"} 3272
rpc_duration_seconds{quantile="0.5"} 4773
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 (https://github.com/OpenObservability/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 (https://github.com/OpenObservability/OpenMetrics/blob/main/specification/OpenMetrics.md#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 (https://developers.google.com/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.

Here is a list of experimental features that, once enabled, will configure Prometheus to favor the Protobuf exposition format:

feature flag	**version that introduced it**
native-histograms	2.40.0
created-timestamp-zero-ingestion	2.50.0

Historical versions

For details on historical format versions, see the legacy Client Data Exposition Format (https://docs.google.com/document/d/1ZjyKiKxZV83VI9ZKAXRGKaUKK2BIWCT7oiGBKDBpjEY/edit?usp=sharing) document.

The current version of the original Protobuf format (with the recent extensions for native histograms) is maintained in the prometheus/client_model repository (https://github.com/prometheus/client_model).

SECURITY MODEL

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.

- Automated security scanners
- Prometheus
- Alertmanager
- Pushgateway

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. If you find a *security bug* please report it privately to the maintainers listed in the MAINTAINERS of the relevant repository and CC prometheus-team@googlegroups.com (mailto: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.

- Exporters
- Client Libraries
- Authentication, Authorization, and Encryption
- API Security
- Secrets
- Denial of Service
- Libraries
- Build Process
- Prometheus-Community
- External audits

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.

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 removes this protection, as can certain relabelling setups.

As of Prometheus 2.0, the `--web.enable-admin-api` flag controls access to the 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` enabled, this means anyone with access to the Pushgateway can create any time series in Prometheus.

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` (https://prometheus.io/docs/guides/tls-encryption/).

The Go projects share the same TLS library, based on the Go crypto/tls (https://golang.org/pkg/crypto/tls) library. We default to TLS 1.2 as minimum version. Our policy regarding this is based on Qualys SSL Labs (https://www.ssllabs.com/) 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 (https://golang.org/pkg/crypto/tls/#InsecureCipherSuites) in the crypto/tls (https://golang.org/pkg/crypto/tls) library. If that still does not suit you, the current TLS settings enable you to build a secure tunnel between the servers and reverse proxies with more special requirements.

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 (https://en.wikipedia.org/wiki/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 (https://en.wikipedia.org/wiki/Cross-site_request_forgery) 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 (https://fetch.spec.whatwg.org/#http-cors-protocol) such as `Access-Control-Allow-Origin` in your reverse proxy to prevent XSS (https://en.wikipedia.org/wiki/Cross-site_scripting).

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 (https://grafana.com/docs/grafana/latest/permissions/#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 placed in other configuration fields, as it is common for components to expose their configuration over their HTTP endpoint. It is the responsibility of the user to protect files on disk from unwanted reads and writes.

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 (https://github.com/prometheus-community) organization are supported by third-party maintainers.

If you find a *security bug* in the Prometheus-Community (https://github.com/prometheus-community) organization, please report it privately to the maintainers listed in the MAINTAINERS of the relevant repository and CC prometheus-team@googlegroups.com (mailto:prometheus-team@googlegroups.com).

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 (https://cncf.io) sponsored an external security audit by cure53 (https://cure53.de) which ran from April 2018 to June 2018. For more details, please read the final report of the audit (/assets/downloads/2018-06-11--cure53_security_audit.pdf).
- In 2020, CNCF sponsored a second audit by cure53 (/assets/downloads/2020-07-21--cure53_security_audit_node_exporter.pdf) of Node Exporter.
- In 2023, CNCF sponsored a software supply chain security assessment of Prometheus (/assets/downloads/2023-04-19--chainguard_supply_chain_assessment.pdf) by Chainguard.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

INTEGRATIONS

In addition to client libraries (/docs/instrumenting/clientlibs/) and exporters and related libraries (/docs/instrumenting/exporters/), there are numerous other generic integration points in Prometheus. This page lists some of the integrations with these.

- File Service Discovery
- Remote Endpoints and Storage
- Alertmanager Webhook Receiver
- Management
- Other

Not all integrations are listed here, due to overlapping functionality or still being in development. The exporter default port (https://github.com/prometheus/prometheus/wiki/Default-port-allocations) 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 (/docs/operating/configuration/#%3Cfile_sd_config%3E) provides an interface for integrating.

- Kuma (https://github.com/kumahq/kuma/tree/master/app/kuma-prometheus-sd)
- Lightsail (https://github.com/n888/prometheus-lightsail-sd)
- Netbox (https://github.com/FlxPeters/netbox-prometheus-sd)
- Packet (https://github.com/packethost/prometheus-packet-sd)
- Scaleway (https://github.com/scaleway/prometheus-scw-sd)

Remote Endpoints and Storage

The remote write (/docs/operating/configuration/#remote_write) and remote read (/docs/operating/configuration/#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 (https://github.com/solarwinds/prometheus2appoptics): write
- AWS Timestream (https://github.com/dpattmann/prometheus-timestream-adapter): read and write
- Azure Data Explorer (https://github.com/cosh/PrometheusToAdx): read and write
- Azure Event Hubs (https://github.com/bryanklewis/prometheus-eventhubs-adapter): write
- Chronix (https://github.com/ChronixDB/chronix.ingester): write
- Cortex (https://github.com/cortexproject/cortex): read and write
- CrateDB (https://github.com/crate/crate_adapter): read and write
- Elasticsearch (https://www.elastic.co/guide/en/beats/metricbeat/master/metricbeat-metricset-prometheus-remote_write.html): write
- Gnocchi (https://gnocchi.osci.io/prometheus.html): write
- Google BigQuery (https://github.com/KohlsTechnology/prometheus_bigquery_remote_storage_adapter): read and write
- Google Cloud Spanner (https://github.com/google/truestreet): read and write
- Grafana Mimir (https://github.com/grafana/mimir): read and write
- Graphite (https://github.com/prometheus/prometheus/tree/main/documentation/examples/remote_storage/remote_storage_a write
- GreptimeDB (https://github.com/GreptimeTeam/greptimedb): read and write
- InfluxDB (https://docs.influxdata.com/influxdb/v1.8/supported_protocols/prometheus): read and write
- Instana (https://www.instana.com/docs/ecosystem/prometheus/#remote-write): write
- IRONdb (https://github.com/circonus-labs/irondb-prometheus-adapter): read and write
- Kafka (https://github.com/Telefonica/prometheus-kafka-adapter): write
- M3DB (https://m3db.io/docs/integrations/prometheus/): read and write
- Mezmo (https://docs.mezmo.com/telemetry-pipelines/prometheus-remote-write-pipeline-source): write
- New Relic (https://docs.newrelic.com/docs/set-or-remove-your-prometheus-remote-write-integration): write
- OpenTSDB (https://github.com/prometheus/prometheus/tree/main/documentation/examples/remote_storage/remote_storage_a write
- QuasarDB (https://doc.quasardb.net/master/user-guide/integration/prometheus.html): read and write
- SignalFx (https://github.com/signalfx/metricproxy#prometheus): write
- Splunk (https://github.com/kebe7jun/ropee): read and write
- Sysdig Monitor (https://docs.sysdig.com/en/docs/installation/prometheus-remote-write/): write
- TiKV (https://github.com/bragfoo/TiPrometheus): read and write
- Thanos (https://github.com/thanos-io/thanos): read and write
- VictoriaMetrics (https://github.com/VictoriaMetrics/VictoriaMetrics): write
- Wavefront (https://github.com/wavefrontHQ/prometheus-storage-adapter): write

Prom-migrator (https://github.com/timescale/promscale/tree/master/migration-tool/cmd/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 (/docs/alerting/configuration/#webhook_config) allows for integration.

- alertmanager-webhook-logger (https://github.com/tomtom-international/alertmanager-webhook-logger): logs alerts
- Alertsnitch (https://gitlab.com/yakshaving.art/alertsnitch): saves alerts to a MySQL database
- Asana (https://gitlab.com/lupudu/alertmanager-asana-bridge)
- AWS SNS (https://github.com/DataReply/alertmanager-sns-forwarder)
- Better Uptime (https://docs.betteruptime.com/integrations/prometheus)

- Canopsis (https://git.canopsis.net/canopsis-connectors/connector-prometheus2canopsis)
- DingTalk (https://github.com/timonwong/prometheus-webhook-dingtalk)
- Discord (https://github.com/benjojo/alertmanager-discord)
- GitLab (https://docs.gitlab.com/ee/operations/metrics/alerts.html#external-prometheus-instances)
- Gotify (https://github.com/DRuggeri/alertmanager_gotify_bridge)
- GELF (https://github.com/b-com-software-basis/alertmanager2gelf)
- Heii On-Call (https://heiioncall.com/guides/prometheus-integration)
- Icinga2 (https://github.com/vshn/signalilo)
- iLert (https://docs.ilert.com/integrations/prometheus)
- IRC Bot (https://github.com/multimfi/bot)
- JIRAlert (https://github.com/free/jiralert)
- Matrix (https://github.com/matrix-org/go-neb)
- Phabricator / Maniphest (https://github.com/knyar/phalerts)
- prom2teams (https://github.com/idealista/prom2teams): forwards notifications to Microsoft Teams
- Ansible Tower (https://github.com/pja237/prom2tower): call Ansible Tower (AWX) API on alerts (launch jobs etc.)
- Signal (https://github.com/dgl/alertmanager-webhook-signald)
- SIGNL4 (https://www.signl4.com/blog/portfolio_item/prometheus-alertmanager-mobile-alert-notification-duty-schedule-escalation)
- SMS (https://github.com/messagebird/sachet): supports multiple providers (https://github.com/messagebird/sachet/blob/master/examples/config.yaml)
- SNMP traps (https://github.com/maxwo/snmp_notifier)
- Squadcast (https://support.squadcast.com/docs/prometheus)
- STOMP (https://github.com/thewillyhuman/alertmanager-stomp-forwarder)
- Telegram bot (https://github.com/inCaller/prometheus_bot)
- xMatters (https://github.com/xmatters/xm-labs-prometheus)
- XMPP Bot (https://github.com/jelmer/prometheus-xmpp-alerts)
- Zenduty (https://docs.zenduty.com/docs/prometheus/)
- Zoom (https://github.com/Code2Life/nodess-apps/tree/master/src/zoom-alert-2.0)

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 (https://github.com/coreos/prometheus-operator): Manages Prometheus on top of Kubernetes
- Promgen (https://github.com/line/promgen): Web UI and configuration generator for Prometheus and Alertmanager

Other

- Alert analysis (https://github.com/m0nikasingh/am2ch): Stores alerts into a ClickHouse database and provides alert analysis dashboards
- karma (https://github.com/prymitive/karma): alert dashboard
- PushProx (https://github.com/RobustPerception/PushProx): Proxy to transverse NAT and similar network setups
- Promdump (https://github.com/ihcsim/promdump): kubectl plugin to dump and restore data blocks
- Promregator (https://github.com/promregator/promregator): discovery and scraping for Cloud Foundry applications
- pint (https://github.com/cloudflare/pint): Prometheus rule linter

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

SPECIFICATIONS

ALERTING OVERVIEW

Alerting with Prometheus is separated into two parts. Alerting rules in Prometheus servers send alerts to an Alertmanager. The Alertmanager (../alertmanager/) then manages those alerts, including silencing, inhibition, aggregation and sending out notifications via methods such as email, on-call notification systems, and chat platforms.

The main steps to setting up alerting and notifications are:

- Setup and configure (../configuration/) the Alertmanager
- Configure Prometheus (/docs/prometheus/latest/configuration/configuration/#alertmanager_config) to talk to the Alertmanager
- Create alerting rules (/docs/prometheus/latest/configuration/alerting_rules/) in Prometheus

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

SPECIFICATIONS

ALERTMANAGER

The Alertmanager

- Grouping
- Inhibition
- Silences
- Client behavior
- High Availability

(https://github.com/prometheus/alertmanager) handles alerts sent by client applications such as the Prometheus server. It takes care of deduplicating, grouping, and routing them to the correct receiver integration such as email, PagerDuty, or OpsGenie. It also takes care of silencing and inhibition of alerts.

The following describes the core concepts the Alertmanager implements. Consult the configuration documentation (../configuration/) to learn how to use them in more detail.

Grouping

Grouping categorizes alerts of similar nature into a single notification. This is especially useful during larger outages when many systems fail at once and hundreds to thousands of alerts may be firing simultaneously.

Example: Dozens or hundreds of instances of a service are running in your cluster when a network partition occurs. Half of your service instances can no longer reach the database. Alerting rules in Prometheus were configured to send an alert for each service instance if it cannot communicate with the database. As a result hundreds of alerts are sent to Alertmanager.

As a user, one only wants to get a single page while still being able to see exactly which service instances were affected. Thus one can configure Alertmanager to group alerts by their cluster and alertname so it sends a single

compact notification.

Grouping of alerts, timing for the grouped notifications, and the receivers of those notifications are configured by a routing tree in the configuration file.

Inhibition

Inhibition is a concept of suppressing notifications for certain alerts if certain other alerts are already firing.

Example: An alert is firing that informs that an entire cluster is not reachable. Alertmanager can be configured to mute all other alerts concerning this cluster if that particular alert is firing. This prevents notifications for hundreds or thousands of firing alerts that are unrelated to the actual issue.

Inhibitions are configured through the Alertmanager's configuration file.

Silences

Silences are a straightforward way to simply mute alerts for a given time. A silence is configured based on matchers, just like the routing tree. Incoming alerts are checked whether they match all the equality or regular expression matchers of an active silence. If they do, no notifications will be sent out for that alert.

Silences are configured in the web interface of the Alertmanager.

Client behavior

The Alertmanager has special requirements (../clients/) for behavior of its client. Those are only relevant for advanced use cases where Prometheus is not used to send alerts.

High Availability

Alertmanager supports configuration to create a cluster for high availability. This can be configured using the --cluster-* (https://github.com/prometheus/alertmanager#high-availability) flags.

It's important not to load balance traffic between Prometheus and its Alertmanagers, but instead, point Prometheus to a list of all Alertmanagers.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

CONFIGURATION

Alertmanager (https://github.com/prometheus/alertmanager) is configured via command-line flags and a configuration file. While the command-line flags configure immutable system parameters, the configuration file defines inhibition rules, notification routing and notification receivers.

The visual editor

- `<wechat_config>`
- `<webex_config>`

(https://www.prometheus.io/webtools/alerting/routing-tree-editor) can assist in building routing trees.

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

Alertmanager can reload its configuration at runtime. If the new configuration is not well-formed, the changes will not be applied and an error is logged. A configuration reload is triggered by sending a `SIGHUP` to the process or sending an HTTP POST request to the `/-/reload` endpoint.

Configuration file introduction

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

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

The file is written in the YAML format (https://en.wikipedia.org/wiki/YAML), 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:

- `<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`
- `<labelname>` : a string matching the regular expression `[a-zA-Z_][a-zA-Z0-9_]*`
- `<labelvalue>` : a string of unicode characters
- `<filepath>` : a valid path in the current working directory
- `<boolean>` : a boolean that can take the values `true` or `false`
- `<string>` : a regular string
- `<secret>` : a regular string that is a secret, such as a password
- `<tmpl_string>` : a string which is template-expanded before usage
- `<tmpl_secret>` : a string which is template-expanded before usage that is a secret
- `<int>` : an integer value

- `<regex>` : any valid RE2 regular expression (https://github.com/google/re2/wiki/Syntax) (The regex is anchored on both ends. To un-anchor the regex, use `.*<regex>.*` .)

The other placeholders are specified separately.

A provided valid example file (https://github.com/prometheus/alertmanager/blob/main/doc/examples/simple.yml) shows usage in context.

File layout and global settings

The global configuration specifies parameters that are valid in all other configuration contexts. They also serve as defaults for other configuration sections. The other top-level sections are documented below on this page.

```
global:
  # The default SMTP From header field.
  [ smtp_from: <tmpl_string> ]
  # The default SMTP smarthost used for sending emails, including port number.
  # Port number usually is 25, or 587 for SMTP over TLS (sometimes referred to as ST
  # Example: smtp.example.org:587
  [ smtp_smarthost: <string> ]
  # The default hostname to identify to the SMTP server.
  [ smtp_hello: <string> | default = "localhost" ]
  # SMTP Auth using CRAM-MD5, LOGIN and PLAIN. If empty, Alertmanager doesn't auther
  [ smtp_auth_username: <string> ]
  # SMTP Auth using LOGIN and PLAIN.
  [ smtp_auth_password: <secret> ]
  # SMTP Auth using LOGIN and PLAIN.
  [ smtp_auth_password_file: <string> ]
  # SMTP Auth using PLAIN.
  [ smtp_auth_identity: <string> ]
  # SMTP Auth using CRAM-MD5.
  [ smtp_auth_secret: <secret> ]
  # The default SMTP TLS requirement.
  # Note that Go does not support unencrypted connections to remote SMTP endpoints.
  [ smtp_require_tls: <bool> | default = true ]

  # The API URL to use for Slack notifications.
  [ slack_api_url: <secret> ]
  [ slack_api_url_file: <filepath> ]
  [ victorops_api_key: <secret> ]
  [ victorops_api_key_file: <filepath> ]
  [ victorops_api_url: <string> | default = "https://alert.victorops.com/integration
  [ pagerduty_url: <string> | default = "https://events.pagerduty.com/v2/enqueue" ]
  [ opsgenie_api_key: <secret> ]
  [ opsgenie_api_key_file: <filepath> ]
  [ opsgenie_api_url: <string> | default = "https://api.opsgenie.com/" ]
  [ wechat_api_url: <string> | default = "https://qyapi.weixin.qq.com/cgi-bin/" ]
  [ wechat_api_secret: <secret> ]
  [ wechat_api_corp_id: <string> ]
  [ telegram_api_url: <string> | default = "https://api.telegram.org" ]
  [ webex_api_url: <string> | default = "https://webexapis.com/v1/messages" ]
  # The default HTTP client configuration
  [ http_config: <http_config> ]

  # ResolveTimeout is the default value used by alertmanager if the alert does
  # not include EndsAt, after this time passes it can declare the alert as resolved
  # This has no impact on alerts from Prometheus, as they always include EndsAt.
```

```
  [ resolve_timeout: <duration> | default = 5m ]

# Files from which custom notification template definitions are read.
# The last component may use a wildcard matcher, e.g. 'templates/*.tmpl'.
templates:
  [ - <filepath> ... ]

# The root node of the routing tree.
route: <route>

# A list of notification receivers.
receivers:
  - <receiver> ...

# A list of inhibition rules.
inhibit_rules:
  [ - <inhibit_rule> ... ]

# DEPRECATED: use time_intervals below.
# A list of mute time intervals for muting routes.
mute_time_intervals:
  [ - <mute_time_interval> ... ]

# A list of time intervals for muting/activating routes.
time_intervals:
  [ - <time_interval> ... ]
```

Route-related settings

Routing-related settings allow configuring how alerts are routed, aggregated, throttled, and muted based on time.

`<route>`

A route block defines a node in a routing tree and its children. Its optional configuration parameters are inherited from its parent node if not set.

Every alert enters the routing tree at the configured top-level route, which must match all alerts (i.e. not have any configured matchers). It then traverses the child nodes. If `continue` is set to false, it stops after the first matching child. If `continue` is true on a matching node, the alert will continue matching against subsequent

siblings. If an alert does not match any children of a node (no matching child nodes, or none exist), the alert is handled based on the configuration parameters of the current node.

See Alertmanager concepts (/docs/alerting/alertmanager/#grouping) for more information on grouping.

```
[ receiver: <string> ]
# The labels by which incoming alerts are grouped together. For example,
# multiple alerts coming in for cluster=A and alertname=LatencyHigh would
# be batched into a single group.
#
# To aggregate by all possible labels use the special value '...' as the sole label
# group_by: ['...']
# This effectively disables aggregation entirely, passing through all
# alerts as-is. This is unlikely to be what you want, unless you have
# a very low alert volume or your upstream notification system performs
# its own grouping.
[ group_by: '[' <labelname>, ... ']' ]

# Whether an alert should continue matching subsequent sibling nodes.
[ continue: <boolean> | default = false ]

# DEPRECATED: Use matchers below.
# A set of equality matchers an alert has to fulfill to match the node.
match:
  [ <labelname>: <labelvalue>, ... ]

# DEPRECATED: Use matchers below.
# A set of regex-matchers an alert has to fulfill to match the node.
match_re:
  [ <labelname>: <regex>, ... ]

# A list of matchers that an alert has to fulfill to match the node.
matchers:
  [ - <matcher> ... ]

# How long to initially wait to send a notification for a group
# of alerts. Allows to wait for an inhibiting alert to arrive or collect
# more initial alerts for the same group. (Usually ~0s to few minutes.)
# If omitted, child routes inherit the group_wait of the parent route.
[ group_wait: <duration> | default = 30s ]

# How long to wait before sending a notification about new alerts that
# are added to a group of alerts for which an initial notification has
# already been sent. (Usually ~5m or more.) If omitted, child routes
# inherit the group_interval of the parent route.
[ group_interval: <duration> | default = 5m ]

# How long to wait before sending a notification again if it has already
# been sent successfully for an alert. (Usually ~3h or more). If omitted,
```

```
# child routes inherit the repeat_interval of the parent route.
# Note that this parameter is implicitly bound by Alertmanager's
# `--data.retention` configuration flag. Notifications will be resent after either
# repeat_interval or the data retention period have passed, whichever
# occurs first. `repeat_interval` should be a multiple of `group_interval`.
[ repeat_interval: <duration> | default = 4h ]

# Times when the route should be muted. These must match the name of a
# mute time interval defined in the mute_time_intervals section.
# Additionally, the root node cannot have any mute times.
# When a route is muted it will not send any notifications, but
# otherwise acts normally (including ending the route-matching process
# if the `continue` option is not set.)
mute_time_intervals:
  [ - <string> ...]

# Times when the route should be active. These must match the name of a
# time interval defined in the time_intervals section. An empty value
# means that the route is always active.
# Additionally, the root node cannot have any active times.
# The route will send notifications only when active, but otherwise
# acts normally (including ending the route-matching process
# if the `continue` option is not set).
active_time_intervals:
  [ - <string> ...]

# Zero or more child routes.
routes:
  [ - <route> ... ]
```

Example

```
# The root route with all parameters, which are inherited by the child
# routes if they are not overwritten.
route:
  receiver: 'default-receiver'
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 4h
  group_by: [cluster, alertname]
  # All alerts that do not match the following child routes
  # will remain at the root node and be dispatched to 'default-receiver'.
  routes:
  # All alerts with service=mysql or service=cassandra
  # are dispatched to the database pager.
  - receiver: 'database-pager'
    group_wait: 10s
    matchers:
    - service=~"mysql|cassandra"
  # All alerts with the team=frontend label match this sub-route.
  # They are grouped by product and environment rather than cluster
  # and alertname.
  - receiver: 'frontend-pager'
    group_by: [product, environment]
    matchers:
    - team="frontend"

  # All alerts with the service=inhouse-service label match this sub-route.
  # the route will be muted during offhours and holidays time intervals.
  # even if it matches, it will continue to the next sub-route
  - receiver: 'dev-pager'
    matchers:
      - service="inhouse-service"
    mute_time_intervals:
      - offhours
      - holidays
    continue: true

    # All alerts with the service=inhouse-service label match this sub-route
    # the route will be active only during offhours and holidays time intervals.
  - receiver: 'on-call-pager'
    matchers:
      - service="inhouse-service"
    active_time_intervals:
```

```
      - offhours
      - holidays
```

<time_interval>

A `time_interval` specifies a named interval of time that may be referenced in the routing tree to mute/activate particular routes for particular times of the day.

```
name: <string>
time_intervals:
  [ - <time_interval_spec> ... ]
```

<time_interval_spec>

A `time_interval_spec` contains the actual definition for an interval of time. The syntax supports the following fields:

```
- times:
  [ - <time_range> ...]
  weekdays:
  [ - <weekday_range> ...]
  days_of_month:
  [ - <days_of_month_range> ...]
  months:
  [ - <month_range> ...]
  years:
  [ - <year_range> ...]
  location: <string>
```

All fields are lists. Within each non-empty list, at least one element must be satisfied to match the field. If a field is left unspecified, any value will match the field. For an instant of time to match a complete time interval, all fields must match. Some fields support ranges and negative indices, and are detailed below. If a time zone is not specified, then the times are taken to be in UTC.

`time_range` : Ranges inclusive of the starting time and exclusive of the end time to make it easy to represent times that start/end on hour boundaries. For example, `start_time: '17:00'` and `end_time: '24:00'` will begin at 17:00 and finish

immediately before 24:00. They are specified like so:

```
times:
- start_time: HH:MM
  end_time: HH:MM
```

`weekday_range` : A list of days of the week, where the week begins on Sunday and ends on Saturday. Days should be specified by name (e.g. 'Sunday'). For convenience, ranges are also accepted of the form `<start_day>:<end_day>` and are inclusive on both ends. For example: `['monday:wednesday','saturday', 'sunday']`

`days_of_month_range` : A list of numerical days in the month. Days begin at 1. Negative values are also accepted which begin at the end of the month, e.g. -1 during January would represent January 31. For example: `['1:5', '-3:-1']`. Extending past the start or end of the month will cause it to be clamped. E.g. specifying `['1:31']` during February will clamp the actual end date to 28 or 29 depending on leap years. Inclusive on both ends.

`month_range` : A list of calendar months identified by a case-insensitive name (e.g. 'January') or by number, where January = 1. Ranges are also accepted. For example, `['1:3', 'may:august', 'december']`. Inclusive on both ends.

`year_range` : A numerical list of years. Ranges are accepted. For example, `['2020:2022', '2030']`. Inclusive on both ends.

`location` : A string that matches a location in the IANA time zone database. For example, `'Australia/Sydney'`. The location provides the time zone for the time interval. For example, a time interval with a location of `'Australia/Sydney'` that contained something like:

```
times:
- start_time: 09:00
  end_time: 17:00
weekdays: ['monday:friday']
```

would include any time that fell between the hours of 9:00AM and 5:00PM, between Monday and Friday, using the local time in Sydney, Australia.

You may also use `'Local'` as a location to use the local time of the machine where Alertmanager is running, or `'UTC'` for UTC time. If no timezone is provided, the time interval is taken to be in UTC time.**Note:** On Windows, only `Local` or `UTC` are supported unless you provide a custom time zone database using the `ZONEINFO` environment variable.

Inhibition-related settings

Inhibition allows muting a set of alerts based on the presence of another set of alerts. This allows establishing dependencies between systems or services such that only the most relevant of a set of interconnected alerts are sent out during an outage.

See Alertmanager concepts (/docs/alerting/alertmanager/#inhibition) for more information on inhibition.

`<inhibit_rule>`

An inhibition rule mutes an alert (target) matching a set of matchers when an alert (source) exists that matches another set of matchers. Both target and source alerts must have the same label values for the label names in the `equal` list.

Semantically, a missing label and a label with an empty value are the same thing. Therefore, if all the label names listed in `equal` are missing from both the source and target alerts, the inhibition rule will apply.

To prevent an alert from inhibiting itself, an alert that matches *both* the target and the source side of a rule cannot be inhibited by alerts for which the same is true (including itself). However, we recommend to choose target and source matchers in a way that alerts never match both sides. It is much easier to reason about and does not trigger this special case.

```
# DEPRECATED: Use target_matchers below.
# Matchers that have to be fulfilled in the alerts to be muted.
target_match:
  [ <labelname>: <labelvalue>, ... ]
# DEPRECATED: Use target_matchers below.
target_match_re:
  [ <labelname>: <regex>, ... ]

# A list of matchers that have to be fulfilled by the target
# alerts to be muted.
target_matchers:
  [ - <matcher> ... ]

# DEPRECATED: Use source_matchers below.
# Matchers for which one or more alerts have to exist for the
# inhibition to take effect.
source_match:
  [ <labelname>: <labelvalue>, ... ]
# DEPRECATED: Use source_matchers below.
source_match_re:
  [ <labelname>: <regex>, ... ]

# A list of matchers for which one or more alerts have
# to exist for the inhibition to take effect.
source_matchers:
  [ - <matcher> ... ]

# Labels that must have an equal value in the source and target
# alert for the inhibition to take effect.
[ equal: '[' <labelname>, ... ']' ]
```

Label matchers

Label matchers match alerts to routes, silences, and inhibition rules.

Important: Prometheus is adding support for UTF-8 in labels and metrics. In order to also support UTF-8 in the Alertmanager, Alertmanager versions 0.27 and later have a new parser for matchers that has a number of backwards incompatible changes. While most matchers will be forward-compatible, some will not.

Alertmanager is operating a transition period where it supports both UTF-8 and classic matchers, and has provided a number of tools to help you prepare for the transition.

If this is a new Alertmanager installation, we recommend enabling UTF-8 strict mode before creating an Alertmanager configuration file. You can find instructions on how to enable UTF-8 strict mode here.

If this is an existing Alertmanager installation, we recommend running the Alertmanager in the default mode called fallback mode before enabling UTF-8 strict mode. In this mode, Alertmanager will log a warning if you need to make any changes to your configuration file before UTF-8 strict mode can be enabled. Alertmanager will make UTF-8 strict mode the default in the next two versions, so it's important to transition as soon as possible.

Irrespective of whether an Alertmanager installation is a new or existing installation, you can also use `amtool` to validate that an Alertmanager configuration file is compatible with UTF-8 strict mode before enabling it in Alertmanager server. You do not need a running Alertmanager server to do this. You can find instructions on how to validate an Alertmanager configuration file using `amtool` here.

Alertmanager server operational modes

During the transition period, Alertmanager supports three modes of operation. These are known as fallback mode, UTF-8 strict mode and classic mode. Fallback mode is the default mode.

Operators of Alertmanager servers should transition to UTF-8 strict mode before the end of the transition period. Alertmanager will make UTF-8 strict mode the default in the next two versions, so it's important to transition as soon as possible.

Fallback mode

Alertmanager runs in a special mode called fallback mode as its default mode. As operators, you should not experience any difference in how your routes, silences or inhibition rules work.

In fallback mode, configurations are first parsed as UTF-8 matchers, and if incompatible with the UTF-8 parser, are then parsed as classic matchers. If your Alertmanager configuration contains matchers that are incompatible with the UTF-8 parser, Alertmanager will parse them as classic matchers and log a warning. This warning also includes a suggestion on how to change the matchers from classic matchers to UTF-8 matchers. For example:

```
ts=2024-02-11T10:00:00Z caller=parse.go:176 level=warn msg="Alertmanager is moving to a new parser for labels and matchers, and this input is incompatible. Alertmanager has instead parsed the input using the classic matchers parser as a fallback. To make this input compatible with the UTF-8 matchers parser please make sure all regular expressions and values are double-quoted. If you are still seeing this message please open an issue." input="foo=" origin=config err="end of input: expected label value" suggestion="foo=\"\""
```

Here the matcher `foo=` can be made into a valid UTF-8 matcher by double quoting the right hand side of the expression to give `foo=""`. These two matchers are equivalent, however with UTF-8 matchers the right hand side of the matcher is a required field.

In rare cases, a configuration can cause disagreement between the UTF-8 and classic parser. This happens when a matcher is valid in both parsers, but due to added support for UTF-8, results in different parsings depending on which parser is used. If your Alertmanager configuration has disagreement, Alertmanager will use the classic parser and log a warning. For example:

```
ts=2024-02-11T10:00:00Z caller=parse.go:183 level=warn msg="Matchers input has disagreement" input="qux=\"\xf0\x9f\x99\x82\"\n" origin=config
```

Any occurrences of disagreement should be looked at on a case by case basis as depending on the nature of the disagreement, the configuration might not need updating before enabling UTF-8 strict mode. For example `\xf0\x9f\x99\x82` is the byte sequence for the 🙂 emoji. If the intention is to match a literal 🙂 emoji then no change is required. However, if the intention is to match the literal `\xf0\x9f\x99\x82` then the matcher should be changed to `qux="\\xf0\\x9f\\x99\\x82"`.

UTF-8 strict mode

In UTF-8 strict mode, Alertmanager disables support for classic matchers:

```
alertmanager --config.file=config.yml --enable-feature="utf8-strict-mode"
```

This mode should be enabled for new Alertmanager installations, and existing Alertmanager installations once all warnings of incompatible matchers have been resolved. Alertmanager will not start in UTF-8 strict mode until all the warnings of incompatible matchers have been resolved:

```
ts=2024-02-11T10:00:00Z caller=coordinator.go:118 level=error component=configuration msg="Loading configuration file failed" file=config.yml err="end of input: expected label value"
```

UTF-8 strict mode will be the default mode of Alertmanager at the end of the transition period.

Classic mode

Classic mode is equivalent to Alertmanager versions 0.26.0 and older:

```
alertmanager --config.file=config.yml --enable-feature="classic-mode"
```

You can use this mode if you suspect there is an issue with fallback mode or UTF-8 strict mode. In such cases, please open an issue on GitHub with as much information as possible.

Verification

You can use `amtool` to validate that an Alertmanager configuration file is compatible with UTF-8 strict mode before enabling it in Alertmanager server. You do not need a running Alertmanager server to do this.

Just like Alertmanager server, `amtool` will log a warning if the configuration is incompatible or contains disagreement:

```
amtool check-config config.yml
Checking 'config.yml'
level=warn msg="Alertmanager is moving to a new parser for labels and matchers, and
level=warn msg="Matchers input has disagreement" input="qux=\"\\xf0\\x9f\\x99\\x82\"
  SUCCESS
Found:
 - global config
 - route
 - 2 inhibit rules
 - 2 receivers
 - 0 templates
```

You will know if a configuration is compatible with UTF-8 strict mode when no warnings are logged in `amtool`:

```
amtool check-config config.yml
Checking 'config.yml'  SUCCESS
Found:
 - global config
 - route
 - 2 inhibit rules
 - 2 receivers
 - 0 templates
```

You can also use `amtool` in UTF-8 strict mode as an additional level of verification. You will know that a configuration is invalid because the command will fail:

```
amtool check-config config.yml --enable-feature="utf8-strict-mode"
level=warn msg="UTF-8 mode enabled"
Checking 'config.yml'  FAILED: end of input: expected label value

amtool: error: failed to validate 1 file(s)
```

You will know that a configuration is valid because the command will succeed:

```
amtool check-config config.yml --enable-feature="utf8-strict-mode"
level=warn msg="UTF-8 mode enabled"
Checking 'config.yml'  SUCCESS
Found:
 - global config
 - route
 - 2 inhibit rules
 - 2 receivers
 - 0 templates
```

`<matcher>`

UTF-8 matchers

A UTF-8 matcher consists of three tokens:

- An unquoted literal or a double-quoted string for the label name.
- One of `=`, `!=`, `=~`, or `!~`. `=` means equals, `!=` means not equal, `=~` means matches the regular expression and `!~` means doesn't match the regular expression.
- An unquoted literal or a double-quoted string for the regular expression or label value.

Unquoted literals can contain all UTF-8 characters other than the reserved characters. These are whitespace, and all characters in `{ } ! = ~ , \ " ' ` `. For example, `foo`, `[a-zA-Z]+`, and `Προμηθεύς` (Prometheus in Greek) are all examples of valid unquoted literals. However, `foo!` is not a valid literal as `!` is a reserved character.

Double-quoted strings can contain all UTF-8 characters. Unlike unquoted literals, there are no reserved characters. You can even use UTF-8 code points. For example, `"foo!"`, `"bar,baz"`, `"\"baz qux\""` and `"\xf0\x9f\x99\x82"` are valid

double-quoted strings.

Classic matchers

A classic matcher is a string with a syntax inspired by PromQL and OpenMetrics. The syntax of a classic matcher consists of three tokens:

- A valid Prometheus label name.
- One of `=`, `!=`, `=~`, or `!~`. `=` means equals, `!=` means that the strings are not equal, `=~` is used for equality of regex expressions and `!~` is used for un-equality of regex expressions. They have the same meaning as known from PromQL selectors.
- A UTF-8 string, which may be enclosed in double quotes. Before or after each token, there may be any amount of whitespace.

The 3rd token may be the empty string. Within the 3rd token, OpenMetrics escaping rules apply: `\"` for a double-quote, `\n` for a line feed, `\\` for a literal backslash. Unescaped `"` must not occur inside the 3rd token (only as the 1st or last character). However, literal line feed characters are tolerated, as are single `\` characters not followed by `\`, `n`, or `"`. They act as a literal backslash in that case.

Composition of matchers

You can compose matchers to create complex match expressions. When composed, all matchers must match for the entire expression to match. For example, the expression `{alertname="Watchdog", severity=~"warning|critical"}` will match an alert with labels `alertname=Watchdog, severity=critical` but not an alert with labels `alertname=Watchdog, severity=none` as while the alertname is Watchdog the severity is neither warning nor critical.

You can compose matchers into expressions with a YAML list:

```
matchers:
  - alertname = Watchdog
  - severity =~ "warning|critical"
```

or as a PromQL inspired expression where each matcher is comma separated:

```
{alertname="Watchdog", severity=~"warning|critical"}
```

A single trailing comma is permitted:

```
{alertname="Watchdog", severity=~"warning|critical",}
```

The open `{` and close `}` brace are optional:

```
alertname="Watchdog", severity=~"warning|critical"
```

However, both must be either present or omitted. You cannot have incomplete open or close braces:

```
{alertname="Watchdog", severity=~"warning|critical"
```

```
alertname="Watchdog", severity=~"warning|critical"}
```

You cannot have duplicate open or close braces either:

```
{{alertname="Watchdog", severity=~"warning|critical",}}
```

Whitespace (spaces, tabs and newlines) is permitted outside double quotes and has no effect on the matchers themselves. For example:

```
{
   alertname = "Watchdog",
   severity =~ "warning|critical",
}
```

is equivalent to:

```
{alertname="Watchdog",severity=~"warning|critical"}
```

More examples

Here are some more examples:

1. Two equals matchers composed as a YAML list:

```
matchers:
  - foo = bar
  - dings != bums
```

2. Two matchers combined composed as a short-form YAML list:

```
matchers: [ foo = bar, dings != bums ]
```

As shown below, in the short-form, it's better to use double quotes to avoid problems with special characters like commas:

```
matchers: [ "foo = \"bar,baz\"", "dings != bums" ]
```

1. You can also put both matchers into one PromQL-like string. Single quotes work best here:

```
matchers: [ '{foo="bar", dings!="bums"}' ]
```

2. To avoid issues with escaping and quoting rules in YAML, you can also use a YAML block:

```
matchers:
  - |
      {quote=~"She said: \"Hi, all!( How're you…)?\""}
```

General receiver-related settings

These receiver settings allow configuring notification destinations (receivers) and HTTP client options for HTTP-based receivers.

`<receiver>`

Receiver is a named configuration of one or more notification integrations.

Note: As part of lifting the past moratorium on new receivers it was agreed that, in addition to the existing requirements, new notification integrations will be required to have a committed maintainer with push access.

```
# The unique name of the receiver.
name: <string>

# Configurations for several notification integrations.
discord_configs:
  [ - <discord_config>, ... ]
email_configs:
  [ - <email_config>, ... ]
msteams_configs:
  [ - <msteams_config>, ... ]
opsgenie_configs:
  [ - <opsgenie_config>, ... ]
pagerduty_configs:
  [ - <pagerduty_config>, ... ]
pushover_configs:
  [ - <pushover_config>, ... ]
slack_configs:
  [ - <slack_config>, ... ]
sns_configs:
  [ - <sns_config>, ... ]
telegram_configs:
  [ - <telegram_config>, ... ]
victorops_configs:
  [ - <victorops_config>, ... ]
webex_configs:
  [ - <webex_config>, ... ]
webhook_configs:
  [ - <webhook_config>, ... ]
wechat_configs:
  [ - <wechat_config>, ... ]
```

`<http_config>`

An `http_config` allows configuring the HTTP client that the receiver uses to communicate with HTTP-based API services.

```
# Note that `basic_auth` and `authorization` options are mutually exclusive.

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

# Optional the `Authorization` header configuration.
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials 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> ]

# Whether to enable HTTP2.
[ enable_http2: <bool> | 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, http_proxy, HTTPS_PR
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]

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

# Configures the TLS settings.
```

```
tls_config:
  [ <tls_config> ]
```

<oauth2>

OAuth 2.0 authentication using the client credentials grant type. Alertmanager 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.
endpoint_params:
  [ <string>: <string> ... ]

# 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_P
[ proxy_from_environment: <boolean> | default: false ]
# Specifies headers to send to proxies during CONNECT requests.
[ proxy_connect_header:
  [ <string>: [<secret>, ...] ] ]
```

`<tls_config>`

A `tls_config` allows configuring TLS connections.

```
# CA certificate to validate the server certificate with.
[ ca_file: <filepath> ]

# Certificate and key files for client cert authentication to the server.
[ cert_file: <filepath> ]
[ key_file: <filepath> ]

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

# Disable validation of the server certificate.
[ insecure_skip_verify: <boolean> | default = false]

# 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> ]
```

Receiver integration settings

These settings allow configuring specific receiver integrations.

`<discord_config>`

Discord notifications are sent via the Discord webhook API (https://discord.com/developers/docs/resources/webhook). See Discord's "Intro to Webhooks" article (https://support.discord.com/hc/en-us/articles/228383668-Intro-to-Webhooks) to learn how to configure a webhook integration for a channel.

```
# Whether to notify about resolved alerts.
[ send_resolved: <boolean> | default = true ]

# The Discord webhook URL.
# webhook_url and webhook_url_file are mutually exclusive.
webhook_url: <secret>
webhook_url_file: <filepath>

# Message title template.
[ title: <tmpl_string> | default = '{{ template "discord.default.title" . }}' ]

# Message body template.
[ message: <tmpl_string> | default = '{{ template "discord.default.message" . }}' ]

# The HTTP client's configuration.
[ http_config: <http_config> | default = global.http_config ]
```

<email_config>

```
# Whether to notify about resolved alerts.
[ send_resolved: <boolean> | default = false ]

# The email address to send notifications to.
to: <tmpl_string>

# The sender's address.
[ from: <tmpl_string> | default = global.smtp_from ]

# The SMTP host through which emails are sent.
[ smarthost: <string> | default = global.smtp_smarthost ]

# The hostname to identify to the SMTP server.
[ hello: <string> | default = global.smtp_hello ]

# SMTP authentication information.
# auth_password and auth_password_file are mutually exclusive.
[ auth_username: <string> | default = global.smtp_auth_username ]
[ auth_password: <secret> | default = global.smtp_auth_password ]
[ auth_password_file: <string> | default = global.smtp_auth_password_file ]
[ auth_secret: <secret> | default = global.smtp_auth_secret ]
[ auth_identity: <string> | default = global.smtp_auth_identity ]

# The SMTP TLS requirement.
# Note that Go does not support unencrypted connections to remote SMTP endpoints.
[ require_tls: <bool> | default = global.smtp_require_tls ]

# TLS configuration.
tls_config:
  [ <tls_config> ]

# The HTML body of the email notification.
[ html: <tmpl_string> | default = '{{ template "email.default.html" . }}' ]
# The text body of the email notification.
[ text: <tmpl_string> ]

# Further headers email header key/value pairs. Overrides any headers
# previously set by the notification implementation.
[ headers: { <string>: <tmpl_string>, ... } ]
```

<msteams_config>

Microsoft Teams notifications are sent via the Incoming Webhooks (https://learn.microsoft.com/en-us/microsoftteams/platform/webhooks-and-connectors/what-are-webhooks-and-connectors) API endpoint.

```
# Whether to notify about resolved alerts.
[ send_resolved: <boolean> | default = true ]

# The incoming webhook URL.
# webhook_url and webhook_url_file are mutually exclusive.
[ webhook_url: <secret> ]
[ webhook_url_file: <filepath> ]

# Message title template.
[ title: <tmpl_string> | default = '{{ template "msteams.default.title" . }}' ]

# Message summary template.
[ summary: <tmpl_string> | default = '{{ template "msteams.default.summary" . }}' ]

# Message body template.
[ text: <tmpl_string> | default = '{{ template "msteams.default.text" . }}' ]

# The HTTP client's configuration.
[ http_config: <http_config> | default = global.http_config ]
```

<opsgenie_config>

OpsGenie notifications are sent via the OpsGenie API (https://docs.opsgenie.com/docs/alert-api).

```
# Whether to notify about resolved alerts.
[ send_resolved: <boolean> | default = true ]

# The API key to use when talking to the OpsGenie API.
[ api_key: <secret> | default = global.opsgenie_api_key ]

# The filepath to API key to use when talking to the OpsGenie API. Conflicts with ap
[ api_key_file: <filepath> | default = global.opsgenie_api_key_file ]

# The host to send OpsGenie API requests to.
[ api_url: <string> | default = global.opsgenie_api_url ]

# Alert text limited to 130 characters.
[ message: <tmpl_string> | default = '{{ template "opsgenie.default.message" . }}' ]

# A description of the alert.
[ description: <tmpl_string> | default = '{{ template "opsgenie.default.description"

# A backlink to the sender of the notification.
[ source: <tmpl_string> | default = '{{ template "opsgenie.default.source" . }}' ]

# A set of arbitrary key/value pairs that provide further detail
# about the alert.
# All common labels are included as details by default.
[ details: { <string>: <tmpl_string>, ... } ]

# List of responders responsible for notifications.
responders:
  [ - <responder> ... ]

# Comma separated list of tags attached to the notifications.
[ tags: <tmpl_string> ]

# Additional alert note.
[ note: <tmpl_string> ]

# Priority level of alert. Possible values are P1, P2, P3, P4, and P5.
[ priority: <tmpl_string> ]

# Whether to update message and description of the alert in OpsGenie if it already e
# By default, the alert is never updated in OpsGenie, the new message only appears i
[ update_alerts: <boolean> | default = false ]

# Optional field that can be used to specify which domain alert is related to.
```

```
[ entity: <tmpl_string> ]

# Comma separated list of actions that will be available for the alert.
[ actions: <tmpl_string> ]

# The HTTP client's configuration.
[ http_config: <http_config> | default = global.http_config ]
```

<responder>

```
# Exactly one of these fields should be defined.
[ id: <tmpl_string> ]
[ name: <tmpl_string> ]
[ username: <tmpl_string> ]

# "team", "teams", "user", "escalation" or "schedule".
type: <tmpl_string>
```

<pagerduty_config>

PagerDuty notifications are sent via the PagerDuty API (https://developer.pagerduty.com/documentation/integration/events). PagerDuty provides documentation (https://www.pagerduty.com/docs/guides/prometheus-integration-guide/) on how to integrate. There are important differences with Alertmanager's v0.11 and greater support of PagerDuty's Events API v2.

```
# Whether to notify about resolved alerts.
[ send_resolved: <boolean> | default = true ]

# The routing and service keys are mutually exclusive.
# The PagerDuty integration key (when using PagerDuty integration type `Events API v
# It is mutually exclusive with `routing_key_file`.
routing_key: <tmpl_secret>
# Read the Pager Duty routing key from a file.
# It is mutually exclusive with `routing_key`.
routing_key_file: <filepath>
# The PagerDuty integration key (when using PagerDuty integration type `Prometheus`)
# It is mutually exclusive with `service_key_file`.
service_key: <tmpl_secret>
# Read the Pager Duty service key from a file.
# It is mutually exclusive with `service_key`.
service_key_file: <filepath>

# The URL to send API requests to
[ url: <string> | default = global.pagerduty_url ]

# The client identification of the Alertmanager.
[ client:  <tmpl_string> | default = '{{ template "pagerduty.default.client" . }}' ]
# A backlink to the sender of the notification.
[ client_url:  <tmpl_string> | default = '{{ template "pagerduty.default.clientURL"

# A description of the incident.
[ description: <tmpl_string> | default = '{{ template "pagerduty.default.descriptior

# Severity of the incident.
[ severity: <tmpl_string> | default = 'error' ]

# Unique location of the affected system.
[ source: <tmpl_string> | default = client ]

# A set of arbitrary key/value pairs that provide further detail
# about the incident.
[ details: { <string>: <tmpl_string>, ... } | default = {
  firing:       '{{ template "pagerduty.default.instances" .Alerts.Firing }}'
  resolved:     '{{ template "pagerduty.default.instances" .Alerts.Resolved }}'
  num_firing:   '{{ .Alerts.Firing | len }}'
  num_resolved: '{{ .Alerts.Resolved | len }}'
} ]

# Images to attach to the incident.
```

```
images:
  [ <image_config> ... ]

# Links to attach to the incident.
links:
  [ <link_config> ... ]

# The part or component of the affected system that is broken.
[ component: <tmpl_string> ]

# A cluster or grouping of sources.
[ group: <tmpl_string> ]

# The class/type of the event.
[ class: <tmpl_string> ]

# The HTTP client's configuration.
[ http_config: <http_config> | default = global.http_config ]
```

<image_config>

The fields are documented in the PagerDuty API documentation (https://developer.pagerduty.com/docs/events-api-v2/trigger-events/#the-images-property).

```
href: <tmpl_string>
src: <tmpl_string>
alt: <tmpl_string>
```

<link_config>

The fields are documented in the PagerDuty API documentation (https://developer.pagerduty.com/docs/events-api-v2/trigger-events/#the-links-property).

```
href: <tmpl_string>
text: <tmpl_string>
```

`<pushover_config>`

Pushover notifications are sent via the Pushover API (https://pushover.net/api).

```
# Whether to notify about resolved alerts.
[ send_resolved: <boolean> | default = true ]

# The recipient user's key.
# user_key and user_key_file are mutually exclusive.
user_key: <secret>
user_key_file: <filepath>

# Your registered application's API token, see https://pushover.net/apps
# You can also register a token by cloning this Prometheus app:
# https://pushover.net/apps/clone/prometheus
# token and token_file are mutually exclusive.
token: <secret>
token_file: <filepath>

# Notification title.
[ title: <tmpl_string> | default = '{{ template "pushover.default.title" . }}' ]

# Notification message.
[ message: <tmpl_string> | default = '{{ template "pushover.default.message" . }}' ]

# A supplementary URL shown alongside the message.
[ url: <tmpl_string> | default = '{{ template "pushover.default.url" . }}' ]

# Optional device to send notification to, see https://pushover.net/api#device
[ device: <string> ]

# Optional sound to use for notification, see https://pushover.net/api#sound
[ sound: <string> ]

# Priority, see https://pushover.net/api#priority
[ priority: <tmpl_string> | default = '{{ if eq .Status "firing" }}2{{ else }}0{{ er

# How often the Pushover servers will send the same notification to the user.
# Must be at least 30 seconds.
[ retry: <duration> | default = 1m ]

# How long your notification will continue to be retried for, unless the user
# acknowledges the notification.
[ expire: <duration> | default = 1h ]

# Optional time to live (TTL) to use for notification, see https://pushover.net/api#
[ ttl: <duration> ]
```

```
# The HTTP client's configuration.
[ http_config: <http_config> | default = global.http_config ]
```

`<slack_config>`

Slack notifications can be sent via Incoming webhooks (https://api.slack.com/messaging/webhooks) or Bot tokens (https://api.slack.com/authentication/token-types).

If using an incoming webhook then `api_url` must be set to the URL of the incoming webhook, or written to the file referenced in `api_url_file`.

If using Bot tokens then `api_url` must be set to `https://slack.com/api/chat.postMessage` (https://api.slack.com/methods/chat.postMessage), the bot token must be set as the authorization credentials in `http_config`, and `channel` must contain either the name of the channel or Channel ID to send notifications to. If using the name of the channel the # is optional.

The notification contains an attachment (https://api.slack.com/messaging/composing/layouts#attachments).

```
# Whether to notify about resolved alerts.
[ send_resolved: <boolean> | default = false ]

# The Slack webhook URL. Either api_url or api_url_file should be set.
# Defaults to global settings if none are set here.
[ api_url: <secret> | default = global.slack_api_url ]
[ api_url_file: <filepath> | default = global.slack_api_url_file ]

# The channel or user to send notifications to.
channel: <tmpl_string>

# API request data as defined by the Slack webhook API.
[ icon_emoji: <tmpl_string> ]
[ icon_url: <tmpl_string> ]
[ link_names: <boolean> | default = false ]
[ username: <tmpl_string> | default = '{{ template "slack.default.username" . }}' ]
# The following parameters define the attachment.
actions:
  [ <action_config> ... ]
[ callback_id: <tmpl_string> | default = '{{ template "slack.default.callbackid" . }
[ color: <tmpl_string> | default = '{{ if eq .Status "firing" }}danger{{ else }}good
[ fallback: <tmpl_string> | default = '{{ template "slack.default.fallback" . }}' ]
fields:
  [ <field_config> ... ]
[ footer: <tmpl_string> | default = '{{ template "slack.default.footer" . }}' ]
[ mrkdwn_in: '[' <string>, ... ']' | default = ["fallback", "pretext", "text"] ]
[ pretext: <tmpl_string> | default = '{{ template "slack.default.pretext" . }}' ]
[ short_fields: <boolean> | default = false ]
[ text: <tmpl_string> | default = '{{ template "slack.default.text" . }}' ]
[ title: <tmpl_string> | default = '{{ template "slack.default.title" . }}' ]
[ title_link: <tmpl_string> | default = '{{ template "slack.default.titlelink" . }}'
[ image_url: <tmpl_string> ]
[ thumb_url: <tmpl_string> ]

# The HTTP client's configuration.
[ http_config: <http_config> | default = global.http_config ]
```

`<action_config>`

The fields are documented in the Slack API documentation for message attachments (https://api.slack.com/messaging/composing/layouts#attachments) and interactive messages (https://api.slack.com/legacy/interactive-message-field-

guide#action_fields).

```
text: <tmpl_string>
type: <tmpl_string>
# Either url or name and value are mandatory.
[ url: <tmpl_string> ]
[ name: <tmpl_string> ]
[ value: <tmpl_string> ]

[ confirm: <action_confirm_field_config> ]
[ style: <tmpl_string> | default = '' ]
```

<action_confirm_field_config>

The fields are documented in the Slack API documentation (https://api.slack.com/legacy/interactive-message-field-guide#confirmation_fields).

```
text: <tmpl_string>
[ dismiss_text: <tmpl_string> | default '' ]
[ ok_text: <tmpl_string> | default '' ]
[ title: <tmpl_string> | default '' ]
```

<field_config>

The fields are documented in the Slack API documentation (https://api.slack.com/messaging/composing/layouts#attachments).

```
title: <tmpl_string>
value: <tmpl_string>
[ short: <boolean> | default = slack_config.short_fields ]
```

<sns_config>

```
# Whether to notify about resolved alerts.
[ send_resolved: <boolean> | default = true ]

# The SNS API URL i.e. https://sns.us-east-2.amazonaws.com.
#  If not specified, the SNS API URL from the SNS SDK will be used.
[ api_url: <tmpl_string> ]

# Configures AWS's Signature Verification 4 signing process to sign requests.
sigv4:
  [ <sigv4_config> ]

# SNS topic ARN, i.e. arn:aws:sns:us-east-2:698519295917:My-Topic
# If you don't specify this value, you must specify a value for the phone_number or
# If you are using a FIFO SNS topic you should set a message group interval longer t
# to prevent messages with the same group key being deduplicated by the SNS default
[ topic_arn: <tmpl_string> ]

# Subject line when the message is delivered to email endpoints.
[ subject: <tmpl_string> | default = '{{ template "sns.default.subject" .}}' ]

# Phone number if message is delivered via SMS in E.164 format.
# If you don't specify this value, you must specify a value for the topic_arn or tar
[ phone_number: <tmpl_string> ]

# The  mobile platform endpoint ARN if message is delivered via mobile notifications
# If you don't specify this value, you must specify a value for the topic_arn or pho
[ target_arn: <tmpl_string> ]

# The message content of the SNS notification.
[ message: <tmpl_string> | default = '{{ template "sns.default.message" .}}' ]

# SNS message attributes.
attributes:
  [ <string>: <string> ... ]

# The HTTP client's configuration.
[ http_config: <http_config> | default = global.http_config ]
```

<sigv4_config>

```
# The AWS region. If blank, the region from the default credentials chain is used.
[ region: <string> ]

# The AWS API keys. Both access_key and secret_key must be supplied or both must be
# If blank the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`
[ 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> ]
```

<telegram_config>

```
# Whether to notify about resolved alerts.
[ send_resolved: <boolean> | default = true ]

# The Telegram API URL i.e. https://api.telegram.org.
# If not specified, default API URL will be used.
[ api_url: <string> | default = global.telegram_api_url ]

# Telegram bot token. It is mutually exclusive with `bot_token_file`.
[ bot_token: <secret> ]

# Read the Telegram bot token from a file. It is mutually exclusive with `bot_token`
[ bot_token_file: <filepath> ]

# ID of the chat where to send the messages.
[ chat_id: <int> ]

# Message template.
[ message: <tmpl_string> default = '{{ template "telegram.default.message" .}}' ]

# Disable telegram notifications
[ disable_notifications: <boolean> | default = false ]

# Parse mode for telegram message, supported values are MarkdownV2, Markdown, HTML a
[ parse_mode: <string> | default = "HTML" ]

# The HTTP client's configuration.
[ http_config: <http_config> | default = global.http_config ]
```

<victorops_config>

VictorOps notifications are sent out via the VictorOps API (https://help.victorops.com/knowledge-base/rest-endpoint-integration-guide/)

```
# Whether to notify about resolved alerts.
[ send_resolved: <boolean> | default = true ]

# The API key to use when talking to the VictorOps API.
# It is mutually exclusive with `api_key_file`.
[ api_key: <secret> | default = global.victorops_api_key ]

# Reads the API key to use when talking to the VictorOps API from a file.
# It is mutually exclusive with `api_key`.
[ api_key_file: <filepath> | default = global.victorops_api_key_file ]

# The VictorOps API URL.
[ api_url: <string> | default = global.victorops_api_url ]

# A key used to map the alert to a team.
routing_key: <tmpl_string>

# Describes the behavior of the alert (CRITICAL, WARNING, INFO).
[ message_type: <tmpl_string> | default = 'CRITICAL' ]

# Contains summary of the alerted problem.
[ entity_display_name: <tmpl_string> | default = '{{ template "victorops.default.ent

# Contains long explanation of the alerted problem.
[ state_message: <tmpl_string> | default = '{{ template "victorops.default.state_mes

# The monitoring tool the state message is from.
[ monitoring_tool: <tmpl_string> | default = '{{ template "victorops.default.monitor

# The HTTP client's configuration.
[ http_config: <http_config> | default = global.http_config ]
```

`<webhook_config>`

The webhook receiver allows configuring a generic receiver.

```
# Whether to notify about resolved alerts.
[ send_resolved: <boolean> | default = true ]

# The endpoint to send HTTP POST requests to.
# url and url_file are mutually exclusive.
url: <secret>
url_file: <filepath>

# The HTTP client's configuration.
[ http_config: <http_config> | default = global.http_config ]

# The maximum number of alerts to include in a single webhook message. Alerts
# above this threshold are truncated. When leaving this at its default value of
# 0, all alerts are included.
[ max_alerts: <int> | default = 0 ]
```

The Alertmanager will send HTTP POST requests in the following JSON format to the configured endpoint:

```
{
  "version": "4",
  "groupKey": <string>,              // key identifying the group of alerts (e.g. to
  "truncatedAlerts": <int>,          // how many alerts have been truncated due to "
  "status": "<resolved|firing>",
  "receiver": <string>,
  "groupLabels": <object>,
  "commonLabels": <object>,
  "commonAnnotations": <object>,
  "externalURL": <string>,           // backlink to the Alertmanager.
  "alerts": [
    {
      "status": "<resolved|firing>",
      "labels": <object>,
      "annotations": <object>,
      "startsAt": "<rfc3339>",
      "endsAt": "<rfc3339>",
      "generatorURL": <string>,      // identifies the entity that caused the alert
      "fingerprint": <string>        // fingerprint to identify the alert
    },
    ...
  ]
}
```

There is a list of integrations (/docs/operating/integrations/#alertmanager-webhook-receiver) with this feature.

`<wechat_config>`

WeChat notifications are sent via the WeChat API (http://admin.wechat.com/wiki/index.php?title=Customer_Service_Messages).

```
# Whether to notify about resolved alerts.
[ send_resolved: <boolean> | default = false ]

# The API key to use when talking to the WeChat API.
[ api_secret: <secret> | default = global.wechat_api_secret ]

# The WeChat API URL.
[ api_url: <string> | default = global.wechat_api_url ]

# The corp id for authentication.
[ corp_id: <string> | default = global.wechat_api_corp_id ]

# API request data as defined by the WeChat API.
[ message: <tmpl_string> | default = '{{ template "wechat.default.message" . }}' ]
# Type of the message type, supported values are `text` and `markdown`.
[ message_type: <string> | default = 'text' ]
[ agent_id: <string> | default = '{{ template "wechat.default.agent_id" . }}' ]
[ to_user: <string> | default = '{{ template "wechat.default.to_user" . }}' ]
[ to_party: <string> | default = '{{ template "wechat.default.to_party" . }}' ]
[ to_tag: <string> | default = '{{ template "wechat.default.to_tag" . }}' ]
```

<webex_config>

```
# Whether to notify about resolved alerts.
[ send_resolved: <boolean> | default = true ]

# The Webex Teams API URL i.e. https://webexapis.com/v1/messages
# If not specified, default API URL will be used.
[ api_url: <string> | default = global.webex_api_url ]

# ID of the Webex Teams room where to send the messages.
room_id: <string>

# Message template.
[ message: <tmpl_string> default = '{{ template "webex.default.message" .}}' ]

# The HTTP client's configuration. You must use this configuration to supply the bot
[ http_config: <http_config> | default = global.http_config ]
```

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

SPECIFICATIONS

SENDING ALERTS

Disclaimer: Prometheus automatically takes care of sending alerts generated by its configured alerting rules (/docs/prometheus/latest/configuration/alerting_rules/). It is highly recommended to configure alerting rules in Prometheus based on time series data rather than implementing a direct client.

The Alertmanager has two APIs, v1 and v2, both listening for alerts. The scheme for v1 is described in the code snipped below. The scheme for v2 is specified as an OpenAPI specification that can be found in the Alertmanager repository (https://github.com/prometheus/alertmanager/blob/master/api/v2/openapi.yaml). Clients are expected to continuously re-send alerts as long as they are still active (usually on the order of 30 seconds to 3 minutes). Clients can push a list of alerts to Alertmanager via a POST request.

The labels of each alert are used to identify identical instances of an alert and to perform deduplication. The annotations are always set to those received most recently and are not identifying an alert.

Both `startsAt` and `endsAt` timestamp are optional. If `startsAt` is omitted, the current time is assigned by the Alertmanager. `endsAt` is only set if the end time of an alert is known. Otherwise it will be set to a configurable timeout period from the time since the alert was last received.

The `generatorURL` field is a unique back-link which identifies the causing entity of this alert in the client.

```
[
  {
    "labels": {
      "alertname": "<requiredAlertName>",
      "<labelname>": "<labelvalue>",
      ...
    },
    "annotations": {
      "<labelname>": "<labelvalue>",
    },
    "startsAt": "<rfc3339>",
    "endsAt": "<rfc3339>",
    "generatorURL": "<generator_url>"
  },
  ...
]
```

NOTIFICATION TEMPLATE REFERENCE

Prometheus creates and sends alerts to the Alertmanager which then sends notifications out to different receivers based on their labels. A receiver can be one of many integrations including: Slack, PagerDuty, email, or a custom integration via the generic webhook interface.

The notifications sent to receivers are constructed via templates. The Alertmanager comes with default templates but they can also be customized. To avoid confusion it's important to note that the Alertmanager templates differ from templating in Prometheus (/docs/visualization/template_reference/), however Prometheus templating also includes the templating in alert rule labels/annotations.

The Alertmanager's notification templates are based on the Go templating (https://golang.org/pkg/text/template) system. Note that some fields are evaluated as text, and others as HTML which will affect escaping.

DATA STRUCTURES

Data

`Data` is the structure passed to notification templates and webhook pushes.

Name	Type	Notes
Receiver	string	Defines the receiver's name that the notification will be sent to (slack, email etc.).
Status	string	Defined as firing if at least one alert is firing, otherwise resolved.
Alerts	Alert	List of all alert objects in this group (see below).
GroupLabels	KV	The labels these alerts were grouped by.
CommonLabels	KV	The labels common to all of the alerts.
CommonAnnotations	KV	Set of common annotations to all of the alerts. Used for longer additional strings of information about the alert.
ExternalURL	string	Backlink to the Alertmanager that sent the notification.

The `Alerts` type exposes functions for filtering alerts:

- `Alerts.Firing` returns a list of currently firing alert objects in this group

- `Alerts.Resolved` returns a list of resolved alert objects in this group

Alert

`Alert` holds one alert for notification templates.

Name	Type	Notes
Status	string	Defines whether or not the alert is resolved or currently firing.
Labels	KV	A set of labels to be attached to the alert.
Annotations	KV	A set of annotations for the alert.
StartsAt	time.Time	The time the alert started firing. If omitted, the current time is assigned by the Alertmanager.
EndsAt	time.Time	Only set if the end time of an alert is known. Otherwise set to a configurable timeout period from the time since the last alert was received.
GeneratorURL	string	A backlink which identifies the causing entity of this alert.
Fingerprint	string	Fingerprint that can be used to identify the alert.

KV

`KV` is a set of key/value string pairs used to represent labels and annotations.

```
type KV map[string]string
```

Annotation example containing two annotations:

```
{
  summary: "alert summary",
  description: "alert description",
}
```

In addition to direct access of data (labels and annotations) stored as KV, there are also methods for sorting, removing, and viewing the LabelSets:

KV methods

Name	Arguments	Returns	Notes
SortedPairs	-	Pairs (list of key/value string pairs.)	Returns a sorted list of key/value pairs.
Remove	[]string	KV	Returns a copy of the key/value map without the given keys.
Names	-	[]string	Returns the names of the label names in the LabelSet.
Values	-	[]string	Returns a list of the values in the LabelSet.

FUNCTIONS

Note the default functions (https://golang.org/pkg/text/template/#hdr-Functions) also provided by Go templating.

Strings

Name	Arguments	Returns	Notes
title	string	strings.Title (https://golang.org/pkg/strings/#Title), capitalises first character of each word.	
toUpper	string	strings.ToUpper (https://golang.org/pkg/strings/#ToUpper), converts all characters to upper case.	
toLower	string	strings.ToLower (https://golang.org/pkg/strings/#ToLower), converts all characters to lower case.	
trimSpace	string	strings.TrimSpace (https://pkg.go.dev/strings#TrimSpace), removes leading and trailing white spaces.	
match	pattern, string	Regexp.MatchString (https://golang.org/pkg/regexp/#MatchString). Match a string using Regexp.	

Name	Arguments	Returns	Notes
reReplaceAll	pattern, replacement, text	Regexp.ReplaceAllString (https://golang.org/pkg/regexp/#Regexp.ReplaceAllString) Regexp substitution, unanchored.	
join	sep string, s []string	strings.Join (https://golang.org/pkg/strings/#Join), concatenates the elements of s to create a single string. The separator string sep is placed between elements in the resulting string. (note: argument order inverted for easier pipelining in templates.)	
safeHtml	text string	html/template.HTML (https://golang.org/pkg/html/template/#HTML), Marks string as HTML not requiring auto-escaping.	
stringSlice	...string	Returns the passed strings as a slice of strings.	

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

SPECIFICATIONS

NOTIFICATION TEMPLATE EXAMPLES

The following are all different examples of alerts and corresponding Alertmanager configuration file setups (alertmanager.yml). Each use the Go templating (https://golang.org/pkg/text/template/) system.

Customizing Slack notifications

In this example we've customised our Slack notification to send a URL to our organisation's wiki on how to deal with the particular alert that's been sent.

```
global:
  # Also possible to place this URL in a file.
  # Ex: `slack_api_url_file: '/etc/alertmanager/slack_url'`
  slack_api_url: '<slack_webhook_url>'

route:
  receiver: 'slack-notifications'
  group_by: [alertname, datacenter, app]

receivers:
- name: 'slack-notifications'
  slack_configs:
  - channel: '#alerts'
    text: 'https://internal.myorg.net/wiki/alerts/{{ .GroupLabels.app }}/{{ .Grou
```

Accessing annotations in CommonAnnotations

In this example we again customize the text sent to our Slack receiver accessing the `summary` and `description` stored in the `CommonAnnotations` of the data sent by the Alertmanager.

Alert

```
groups:
- name: Instances
  rules:
  - alert: InstanceDown
    expr: up == 0
    for: 5m
    labels:
      severity: page
    # Prometheus templates apply here in the annotation and label fields of the a
    annotations:
      description: '{{ $labels.instance }} of job {{ $labels.job }} has been down
      summary: 'Instance {{ $labels.instance }} down'
```

Receiver

```
- name: 'team-x'
  slack_configs:
  - channel: '#alerts'
    # Alertmanager templates apply here.
    text: "<!channel> \nsummary: {{ .CommonAnnotations.summary }}\ndescription: {
```

Ranging over all received Alerts

Finally, assuming the same alert as the previous example, we customize our receiver to range over all of the alerts received from the Alertmanager, printing their respective annotation summaries and descriptions on new lines.

Receiver

```
- name: 'default-receiver'
  slack_configs:
  - channel: '#alerts'
    title: "{{ range .Alerts }}{{ .Annotations.summary }}\n{{ end }}"
    text: "{{ range .Alerts }}{{ .Annotations.description }}\n{{ end }}"
```

Defining reusable templates

Going back to our first example, we can also provide a file containing named templates which are then loaded by Alertmanager in order to avoid complex templates that span many lines. Create a file under `/alertmanager/template/myorg.tmpl` and create a template in it named "slack.myorg.text":

```
{{ define "slack.myorg.text" }}https://internal.myorg.net/wiki/alerts/{{ .GroupLa
```

The configuration now loads the template with the given name for the "text" field and we provide a path to our custom template file:

```
global:
  slack_api_url: '<slack_webhook_url>'

route:
  receiver: 'slack-notifications'
  group_by: [alertname, datacenter, app]

receivers:
- name: 'slack-notifications'
  slack_configs:
  - channel: '#alerts'
    text: '{{ template "slack.myorg.text" . }}'

templates:
- '/etc/alertmanager/templates/myorg.tmpl'
```

This example is explained in further detail in this blogpost (/blog/2016/03/03/custom-alertmanager-templates/).

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

SPECIFICATIONS

MANAGEMENT API

Alertmanager provides a set of management API to ease automation and integrations.

- Health check
- Readiness check
- Reload

Health check

```
GET /-/healthy
HEAD /-/healthy
```

This endpoint always returns 200 and should be used to check Alertmanager health.

Readiness check

```
GET /-/ready
HEAD /-/ready
```

This endpoint returns 200 when Alertmanager is ready to serve traffic (i.e. respond to queries).

Reload

```
POST /-/reload
```

This endpoint triggers a reload of the Alertmanager configuration file.

An alternative way to trigger a configuration reload is by sending a `SIGHUP` to the Alertmanager process.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

SPECIFICATIONS

HTTPS AND AUTHENTICATION

Alertmanager supports basic authentication and TLS. This is **experimental** and might change in the future.

- HTTP Traffic
- Gossip Traffic

Currently TLS is supported for the HTTP traffic and gossip traffic.

HTTP Traffic

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

The file is written in YAML format (https://en.wikipedia.org/wiki/YAML), 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

```
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
  #
  # NOTE: If you want to enable client authentication, you need to use
  # RequireAndVerifyClientCert. Other values are insecure.
  [ client_auth_type: <string> | default = "NoClientCert" ]

  # CA certificate for client certificate authentication to the server.
  [ client_ca_file: <filename> ]

  # 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: <bool> | 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.
  # This can not be changed on the fly.
  [ http2: <boolean> | default = true ]
  # List of headers that can be added to HTTP responses.
  [ 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-Op
    [ 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-
    [ 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> ... ]
```

Gossip Traffic

To specify whether to use mutual TLS for gossip, use the `--cluster.tls-config` flag.

The server and client sides of the gossip are configurable.

```
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
  [ client_auth_type: <string> | default = "NoClientCert" ]

  # CA certificate for client certificate authentication to the server.
  [ client_ca_file: <filename> ]

  # 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
  [ 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: <bool> | 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> ] ]

tls_client_config:
  # Path to the CA certificate with which to validate the server certificate.
  [ ca_file: <filepath> ]

  # Certificate and key files for client cert authentication to the server.
```

```
[ cert_file: <filepath> ]
[ key_file: <filepath> ]

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

# Disable validation of the server certificate.
[ insecure_skip_verify: <boolean> | default = false]
```

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 (/docs/concepts/data_model/#metric-names-and-labels) 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.
  - `http_request_duration_`**`seconds`**
  - `node_memory_usage_`**`bytes`**
  - `http_requests_`**`total`** (for a unit-less accumulating count)
  - `process_cpu_`**`seconds_total`** (for an accumulating count with unit)

    - `foobar_build_info` (for a pseudo-metric that provides metadata (https://www.robustperception.io/exposing-the-software-version-to-prometheus) about the running binary)
    - `data_pipeline_last_record_processed_timestamp_seconds` (for a timestamp that tracks the time of the latest record processed in a data processing pipeline)
- ...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.

Labels

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

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

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

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

Base units

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

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

Family	Base unit	Remark
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.

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 (/docs/visualization/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.

INSTRUMENTATION

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

How to instrument

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

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

The three types of services

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

Online-serving systems

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

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

For counting failed queries, see section Failures below.

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

them individually, so they have to rely on their own stats.

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

Offline processing

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

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

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

Batch jobs

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

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

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

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

Subsystems

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

Libraries

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

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

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

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

Logging

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

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

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

Failures

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

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

Threadpools

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

Caches

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

Collectors

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

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

than tracking multiple durations over time.

Things to watch out for

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

Use labels

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

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

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

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

See also the naming (/docs/practices/naming/) section.

Do not overuse labels

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

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

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

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

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

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

Counter vs. gauge, summary vs. histogram

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

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

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

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

Summaries and histograms are more complex metric types discussed in their own section (/docs/practices/histograms/).

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 (https://github.com/prometheus/client_java/blob/master/benchmark/README.md) 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.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

NOTE: This document predates native histograms (added as an experimental feature in Prometheus v2.40). Once native histograms are closer to becoming a stable feature, this document will be thoroughly updated.

HISTOGRAMS AND SUMMARIES

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
- Count and sum of observations
- Apdex score
- Quantiles
- Errors of quantile estimation
- What can I do if my client library does not support the metric type I need?

Library support

First of all, check the library support for histograms (/docs/concepts/metric_types/#histogram) and summaries (/docs/concepts/metric_types/#summary).

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 positive and one for negative observations (the latter with inverted sign), and combine the results later with suitable PromQL expressions.

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 (https://en.wikipedia.org/wiki/Apdex) 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 (https://en.wikipedia.org/wiki/Histogram#Cumulative_histogram). The `le="0.3"` bucket is also contained in the `le="1.2"` bucket; dividing it by 2 corrects for that.

The calculation does not exactly match the traditional Apdex score, as it includes errors in the satisfied and tolerable parts of the calculation.

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 (/docs/prometheus/latest/querying/functions/#histogram_quantile).

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 and sliding windows cannot be calcu
Client performance	Observations are very cheap as they only need to increment counters.	Observations are expensive due to t calculation.
Server performance	The server has to calculate quantiles. You can use recording rules (/docs/prometheus/latest/configuration/recording_rules/#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 quar
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
Specification of φ-quantile and sliding time-window	Ad-hoc with Prometheus expressions (/docs/prometheus/latest/querying/functions/#histogram_quantile).	Preconfigured by the client.
Aggregation	Ad-hoc with Prometheus expressions (/docs/prometheus/latest/querying/functions/#histogram_quantile).	In general not aggregatable (http://latencytipoftheday.blogspot.c you-cant-average.html).

Note the importance of the last item in the table. Let us return to the SLO of serving 95% of requests within 300ms. This time, you do not want to display the percentage of requests served within 300ms, but instead the 95th percentile, i.e. the request duration within which you have served 95% of requests. To do that, you can either configure a summary with a 0.95-quantile and (for example) a 5-minute decay time, or you configure a 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 (/docs/prometheus/latest/querying/functions/#histogram_quantile).

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

Furthermore, should your SLO change and you now want to plot the 90th percentile, or you want to take into account the last 10 minutes instead of the last 5 minutes, you only have to adjust the 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.

Next step in our thought experiment: A change in backend routing adds a fixed amount of 100ms to all request durations. Now the request duration has its sharp spike at 320ms and almost all observations will fall into the bucket from 300ms to 450ms. The 95th percentile is 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 (http://dimacs.rutgers.edu/%7Egraham/pubs/slides/bquant-long.pdf)). 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.
2. Otherwise, choose a histogram if you have an idea of the range and distribution of values that will be observed. Choose a summary if you need an accurate quantile, no matter what the range and distribution of the values is.

What can I do if my client library does not support the metric type I need?

Implement it! Code contributions are welcome (/community/). 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.

ALERTING

- Naming
- What to alert on
  - Online serving systems
  - Offline processing
  - Batch jobs
  - Capacity
  - Metamonitoring

We recommend that you read My Philosophy on Alerting (https://docs.google.com/a/boxever.com/document/d/199PqyG3UsyXlwieHaqbGiWVa8eMWi8zzAn0YfcApr8Q/edit) 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 (https://monitoring.mixins.dev/) using Camel Case (https://en.wikipedia.org/wiki/Camel_case) for their alert names.

What to alert on

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

Allow for slack in alerting to accommodate small blips.

Online serving systems

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

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

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

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

Offline processing

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

Batch jobs

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

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

Capacity

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

Metamonitoring

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

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

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

RECORDING RULES

A consistent naming scheme for recording rules

(/docs/prometheus/latest/configuration/recording_rules/) makes it easier to interpret the meaning of a rule at a glance. It also avoids mistakes by making incorrect or meaningless calculations stand out.

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

Naming

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

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

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

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

Aggregation

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

Examples

Note the indentation style with outdented operators on their own line between two vectors. To make this style possible in Yaml, block quotes with an indentation indicator (https://yaml.org/spec/1.2/spec.html#style/block/scalar) (e.g. `|2`) are used.

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

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

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

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

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

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

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

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

Calculating average latency over a time period from a Summary:

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

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

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

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

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

```
- record: job:request_latency_seconds_count:avg_rate5m
  expr: avg without (instance, path)(instance:request_latency_seconds_count:rate5
```

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

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 (/docs/instrumenting/pushing/).

Should I be using the Pushgateway?

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

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

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

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

Alternative strategies

If an inbound firewall or NAT is preventing you from pulling metrics from targets, consider moving the Prometheus server behind the network barrier as well. We generally recommend running Prometheus servers on the same network as the monitored instances. Otherwise, consider PushProx (https://github.com/RobustPerception/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 (https://github.com/prometheus/node_exporter) textfile collector (https://github.com/prometheus/node_exporter#textfile-collector) instead of the Pushgateway.

REMOTE WRITE TUNING

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

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

- Remote write characteristics
  - Resource usage
- Parameters
  - `capacity`
  - `max_shards`
  - `min_shards`
  - `max_samples_per_send`
  - `batch_send_deadline`
  - `min_backoff`
  - `max_backoff`

(/docs/prometheus/latest/configuration/configuration/#remote_write)

Remote write characteristics

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

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

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

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

Resource usage

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

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

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

Parameters

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

`capacity`

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

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

`max_shards`

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

`min_shards`

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

`max_samples_per_send`

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

`batch_send_deadline`

Batch send deadline sets the maximum amount of time between sends for a single shard. Even if the queued shards has not reached `max_samples_per_send`, a request will be sent. Batch send deadline can be increased for low volume

systems that are not latency sensitive in order to increase request efficiency.

`min_backoff`

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

`max_backoff`

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

TUTORIALS

SPECIFICATIONS

SECURING PROMETHEUS API AND UI ENDPOINTS USING BASIC AUTH

Prometheus supports basic authentication (https://en.wikipedia.org/wiki/Basic_access_authentication) (aka "basic auth") for connections to the Prometheus expression browser (/docs/visualization/browser) and HTTP API (/docs/prometheus/latest/querying/api).

NOTE: This tutorial covers basic auth connections *to* Prometheus instances. Basic auth is also supported for connections *from* Prometheus instances to scrape targets (../../prometheus/latest/configuration/configuration/#scrape_config).

Hashing a password

Let's say that you want to require a username and password from all users accessing the Prometheus instance. For this example, use `admin` as the username and choose any password you'd like.

First, generate a bcrypt (https://en.wikipedia.org/wiki/Bcrypt) hash of the password. To generate a hashed password, we will use python3-bcrypt.

Let's install it by running `apt install python3-bcrypt`, assuming you are running a debian-like distribution. Other alternatives exist to generate hashed passwords; for testing you can also use bcrypt generators on the web (https://bcrypt-generator.com/).

Here is a python script which uses python3-bcrypt to prompt for a password and hash it:

```
import getpass
import bcrypt

password = getpass.getpass("password: ")
hashed_password = bcrypt.hashpw(password.encode("utf-8"), bcrypt.gensalt())
print(hashed_password.decode())
```

Save that script as `gen-pass.py` and run it:

```
$ python3 gen-pass.py
```

That should prompt you for a password:

```
password:
$2b$12$hNf2lSsxfm0.i4a.1kVpSOVyBCfIB51VRjgBUyv6kdnyTlgWj81Ay
```

In this example, I used "test" as password.

Save that password somewhere, we will use it in the next steps!

Creating web.yml

Let's create a web.yml file (documentation (https://prometheus.io/docs/prometheus/latest/configuration/https/)), with the following content:

```
basic_auth_users:
    admin: $2b$12$hNf2lSsxfm0.i4a.1kVpSOVyBCfIB51VRjgBUyv6kdnyTlgWj81Ay
```

You can validate that file with `promtool check web-config web.yml`

```
$ promtool check web-config web.yml
web.yml SUCCESS
```

You can add multiple users to the file.

Launching Prometheus

You can launch prometheus with the web configuration file as follows:

```
$ prometheus --web.config.file=web.yml
```

Testing

You can use cURL to interact with your setup. Try this request:

```
curl --head http://localhost:9090/graph
```

This will return a `401 Unauthorized` response because you've failed to supply a valid username and password.

To successfully access Prometheus endpoints using basic auth, for example the `/metrics` endpoint, supply the proper username using the `-u` flag and supply the password when prompted:

```
curl -u admin http://localhost:9090/metrics
Enter host password for user 'admin':
```

That should return Prometheus metrics output, which should look something like this:

```
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0.0001343
go_gc_duration_seconds{quantile="0.25"} 0.0002032
go_gc_duration_seconds{quantile="0.5"} 0.0004485
...
```

Summary

In this guide, you stored a username and a hashed password in a `web.yml` file, launched prometheus with the parameter required to use the credentials in that file to authenticate users accessing Prometheus' HTTP endpoints.

MONITORING DOCKER CONTAINER METRICS USING CADVISOR

cAdvisor (https://github.com/google/cadvisor) (short for **c**ontainer **Advisor**) analyzes and exposes resource usage and performance data from running containers. cAdvisor exposes Prometheus metrics out of the box. In this guide, we will:

- create a local multi-container Docker Compose (https://docs.docker.com/compose/) installation that includes containers running Prometheus, cAdvisor, and a Redis (https://redis.io/) server, respectively
- Prometheus configuration
- Docker Compose configuration
- Exploring the cAdvisor web UI
- Exploring metrics in the expression browser
- Other expressions
- Summary

- examine some container metrics produced by the Redis container, collected by cAdvisor, and scraped by Prometheus

Prometheus configuration

First, you'll need to configure Prometheus (/docs/prometheus/latest/configuration/configuration) to scrape metrics from cAdvisor. Create a `prometheus.yml` file and populate it with this configuration:

```
scrape_configs:
- job_name: cadvisor
  scrape_interval: 5s
  static_configs:
  - targets:
    - cadvisor:8080
```

Docker Compose configuration

Now we'll need to create a Docker Compose configuration (https://docs.docker.com/compose/compose-file/) that specifies which containers are part of our installation as well as which ports are exposed by each container, which volumes are used, and so on.

In the same folder where you created the `prometheus.yml` file, create a `docker-compose.yml` file and populate it with this Docker Compose configuration:

```
version: '3.2'
services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    ports:
    - 9090:9090
    command:
    - --config.file=/etc/prometheus/prometheus.yml
    volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml:ro
    depends_on:
    - cadvisor
  cadvisor:
    image: gcr.io/cadvisor/cadvisor:latest
    container_name: cadvisor
    ports:
    - 8080:8080
    volumes:
    - /:/rootfs:ro
    - /var/run:/var/run:rw
    - /sys:/sys:ro
    - /var/lib/docker/:/var/lib/docker:ro
    depends_on:
    - redis
  redis:
    image: redis:latest
    container_name: redis
    ports:
    - 6379:6379
```

This configuration instructs Docker Compose to run three services, each of which corresponds to a Docker (https://docker.com) container:

1. The `prometheus` service uses the local `prometheus.yml` configuration file (imported into the container by the `volumes` parameter).

2. The `cadvisor` service exposes port 8080 (the default port for cAdvisor metrics) and relies on a variety of local volumes (`/` , `/var/run` , etc.).
3. The `redis` service is a standard Redis server. cAdvisor will gather container metrics from this container automatically, i.e. without any further configuration.

To run the installation:

```
docker-compose up
```

If Docker Compose successfully starts up all three containers, you should see output like this:

```
prometheus  | level=info ts=2018-07-12T22:02:40.5195272Z caller=main.go:500 msg="Server is ready to receive web requests."
```

You can verify that all three containers are running using the `ps` (https://docs.docker.com/compose/reference/ps/) command:

```
docker-compose ps
```

Your output will look something like this:

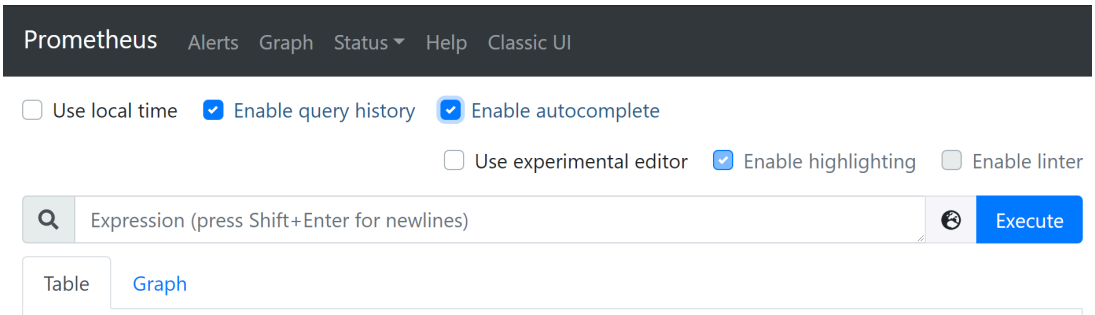
```
   Name                 Command               State          Ports
-------------------------------------------------------------------------
cadvisor     /usr/bin/cadvisor -logtostderr   Up      8080/tcp
prometheus   /bin/prometheus --config.f ...   Up      0.0.0.0:9090->9090/tcp
redis        docker-entrypoint.sh redis ...   Up      0.0.0.0:6379->6379/tcp
```

Exploring the cAdvisor web UI

You can access the cAdvisor web UI (https://github.com/google/cadvisor/blob/master/docs/web.md) at `http://localhost:8080` . You can explore stats and graphs for specific Docker containers in our installation at `http://localhost:8080/docker/<container>` . Metrics for the Redis container, for example, can be accessed at `http://localhost:8080/docker/redis` , Prometheus at `http://localhost:8080/docker/prometheus` , and so on.

Exploring metrics in the expression browser

cAdvisor's web UI is a useful interface for exploring the kinds of things that cAdvisor monitors, but it doesn't provide an interface for exploring container *metrics*. For that we'll need the Prometheus expression browser (/docs/visualization/browser), which is available at `http://localhost:9090/graph` . You can enter Prometheus expressions into the expression bar, which looks like this:



Let's start by exploring the `container_start_time_seconds` metric, which records the start time of containers (in seconds). You can select for specific containers by name using the `name="<container_name>"` expression. The container name corresponds to the `container_name` parameter in the Docker Compose configuration. The

`container_start_time_seconds{name="redis"}` (http://localhost:9090/graph?g0.range_input=1h&g0.expr=container_start_time_seconds%7Bname%3D%22redis%22%7D&g0.tab=1) expression, for example, shows the start time for the `redis` container.

NOTE: A full listing of cAdvisor-gathered container metrics exposed to Prometheus can be found in the cAdvisor documentation (https://github.com/google/cadvisor/blob/master/docs/storage/prometheus.md).

Other expressions

The table below lists some other example expressions

Expression
`rate(container_cpu_usage_seconds_total{name="redis"}[1m])` (http://localhost:9090/graph?g0.range_input=1h&g0.expr=rate(container_cpu_usage_seconds_total%7Bname%3D%22redis%22%7D%5B1m%5D)&g0.tab
`container_memory_usage_bytes{name="redis"}` (http://localhost:9090/graph?g0.range_input=1h&g0.expr=container_memory_usage_bytes%7Bname%3D%22redis%22%7D&g0.tab=1)
`rate(container_network_transmit_bytes_total[1m])` (http://localhost:9090/graph?g0.range_input=1h&g0.expr=rate(container_network_transmit_bytes_total%5B1m%5D)&g0.tab=1)
`rate(container_network_receive_bytes_total[1m])` (http://localhost:9090/graph?g0.range_input=1h&g0.expr=rate(container_network_receive_bytes_total%5B1m%5D)&g0.tab=1)

Summary

In this guide, we ran three separate containers in a single installation using Docker Compose: a Prometheus container scraped metrics from a cAdvisor container which, in turns, gathered metrics produced by a Redis container. We then explored a handful of cAdvisor container metrics using the Prometheus expression browser.

TUTORIALS

SPECIFICATIONS

USE FILE-BASED SERVICE DISCOVERY TO DISCOVER SCRAPE TARGETS

Prometheus offers a variety of service discovery options

(https://github.com/prometheus/prometheus/tree/main/discovery) for discovering scrape targets, including Kubernetes (/docs/prometheus/latest/configuration/configuration/#kubernetes_sd_config), Consul (/docs/prometheus/latest/configuration/configuration/#consul_sd_config), and many others. If you need to use a service discovery system that is not currently supported, your use case may be best served by Prometheus' file-based service discovery (/docs/prometheus/latest/configuration/configuration/#file_sd_config) mechanism, which enables you to list scrape targets in a JSON file (along with metadata about those targets).

In this guide, we will:

- Install and run a Prometheus Node Exporter (../node-exporter) locally

- Create a `targets.json` file specifying the host and port information for the Node Exporter
- Install and run a Prometheus instance that is configured to discover the Node Exporter using the `targets.json` file

Installing and running the Node Exporter

See this section (../node-exporter#installing-and-running-the-node-exporter) of the Monitoring Linux host metrics with the Node Exporter (../node-exporter) guide. The Node Exporter runs on port 9100. To ensure that the Node Exporter is exposing metrics:

```
curl http://localhost:9100/metrics
```

The metrics output should look something like this:

```
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0
go_gc_duration_seconds{quantile="0.25"} 0
go_gc_duration_seconds{quantile="0.5"} 0
...
```

Installing, configuring, and running Prometheus

Like the Node Exporter, Prometheus is a single static binary that you can install via tarball. Download the latest release (/download#prometheus) for your platform and untar it:

```
wget https://github.com/prometheus/prometheus/releases/download/v*/prometheus-*.*
tar xvf prometheus-*.*-amd64.tar.gz
cd prometheus-*.*
```

The untarred directory contains a `prometheus.yml` configuration file. Replace the current contents of that file with this:

```
scrape_configs:
- job_name: 'node'
  file_sd_configs:
  - files:
    - 'targets.json'
```

This configuration specifies that there is a job called `node` (for the Node Exporter) that retrieves host and port information for Node Exporter instances from a `targets.json` file.

Now create that `targets.json` file and add this content to it:

```
[
  {
    "labels": {
      "job": "node"
    },
    "targets": [
      "localhost:9100"
    ]
  }
]
```

NOTE: In this guide we'll work with JSON service discovery configurations manually for the sake of brevity. In general, however, we recommend that you use some kind of JSON-generating process or tool instead.

This configuration specifies that there is a `node` job with one target: `localhost:9100`.

Now you can start up Prometheus:

```
./prometheus
```

If Prometheus has started up successfully, you should see a line like this in the logs:

```
level=info ts=2018-08-13T20:39:24.905651509Z caller=main.go:500 msg="Server is re
```

Exploring the discovered services' metrics

With Prometheus up and running, you can explore metrics exposed by the `node` service using the Prometheus expression browser (/docs/visualization/browser). If you explore the `up{job="node"}` (http://localhost:9090/graph?g0.range_input=1h&g0.expr=up%7Bjob%3D%22node%22%7D&g0.tab=1) metric, for example, you can see that the Node Exporter is being appropriately discovered.

Changing the targets list dynamically

When using Prometheus' file-based service discovery mechanism, the Prometheus instance will listen for changes to the file and automatically update the scrape target list, without requiring an instance restart. To demonstrate this, start up a second Node Exporter instance on port 9200. First navigate to the directory containing the Node Exporter binary and run this command in a new terminal window:

```
./node_exporter --web.listen-address=":9200"
```

Now modify the config in `targets.json` by adding an entry for the new Node Exporter:

```
[
  {
    "targets": [
      "localhost:9100"
    ],
    "labels": {
      "job": "node"
    }
  },
  {
    "targets": [
      "localhost:9200"
    ],
    "labels": {
      "job": "node"
    }
  }
]
```

When you save the changes, Prometheus will automatically be notified of the new list of targets. The `up{job="node"}` (http://localhost:9090/graph?g0.range_input=1h&g0.expr=up%7Bjob%3D%22node%22%7D&g0.tab=1) metric should display two instances with `instance` labels `localhost:9100` and `localhost:9200`.

Summary

In this guide, you installed and ran a Prometheus Node Exporter and configured Prometheus to discover and scrape metrics from the Node Exporter using file-based service discovery.

INSTRUMENTING A GO APPLICATION FOR PROMETHEUS

Prometheus has an official Go client library (https://github.com/prometheus/client_golang) that you can use to instrument Go applications. In this guide, we'll create a simple Go application that exposes Prometheus metrics via HTTP.

NOTE: For comprehensive API documentation, see the GoDoc (https://godoc.org/github.com/prometheus/client_golang) for Prometheus' various Go libraries.

Installation

You can install the `prometheus`, `promauto`, and `promhttp` libraries necessary for the guide using `go get` (https://golang.org/doc/articles/go_command.html):

```
go get github.com/prometheus/client_golang/prometheus
go get github.com/prometheus/client_golang/prometheus/promauto
go get github.com/prometheus/client_golang/prometheus/promhttp
```

How Go exposition works

To expose Prometheus metrics in a Go application, you need to provide a `/metrics` HTTP endpoint. You can use the `prometheus/promhttp` (https://godoc.org/github.com/prometheus/client_golang/prometheus/promhttp) library's HTTP `Handler` (https://godoc.org/github.com/prometheus/client_golang/prometheus/promhttp#Handler) as the handler function.

This minimal application, for example, would expose the default metrics for Go applications via `http://localhost:2112/metrics`:

```
package main

import (
        "net/http"

        "github.com/prometheus/client_golang/prometheus/promhttp"
)

func main() {
        http.Handle("/metrics", promhttp.Handler())
        http.ListenAndServe(":2112", nil)
}
```

To start the application:

```
go run main.go
```

To access the metrics:

```
curl http://localhost:2112/metrics
```

Adding your own metrics

The application above exposes only the default Go metrics. You can also register your own custom application-specific metrics. This example application exposes a `myapp_processed_ops_total` counter (/docs/concepts/metric_types/#counter) that counts the number of operations that have been processed thus far. Every 2 seconds, the counter is incremented by one.

```
package main

import (
        "net/http"
        "time"

        "github.com/prometheus/client_golang/prometheus"
        "github.com/prometheus/client_golang/prometheus/promauto"
        "github.com/prometheus/client_golang/prometheus/promhttp"
)

func recordMetrics() {
        go func() {
                for {
                        opsProcessed.Inc()
                        time.Sleep(2 * time.Second)
                }
        }()
}

var (
        opsProcessed = promauto.NewCounter(prometheus.CounterOpts{
                Name: "myapp_processed_ops_total",
                Help: "The total number of processed events",
        })
)

func main() {
        recordMetrics()

        http.Handle("/metrics", promhttp.Handler())
        http.ListenAndServe(":2112", nil)
}
```

To run the application:

```
go run main.go
```

To access the metrics:

```
curl http://localhost:2112/metrics
```

In the metrics output, you'll see the help text, type information, and current value of the `myapp_processed_ops_total` counter:

```
# HELP myapp_processed_ops_total The total number of processed events
# TYPE myapp_processed_ops_total counter
myapp_processed_ops_total 5
```

You can configure (/docs/prometheus/latest/configuration/configuration/#scrape_config) a locally running Prometheus instance to scrape metrics from the application. Here's an example `prometheus.yml` configuration:

```
scrape_configs:
- job_name: myapp
  scrape_interval: 10s
  static_configs:
  - targets:
    - localhost:2112
```

Other Go client features

In this guide we covered just a small handful of features available in the Prometheus Go client libraries. You can also expose other metrics types, such as gauges (https://godoc.org/github.com/prometheus/client_golang/prometheus#Gauge) and histograms (https://godoc.org/github.com/prometheus/client_golang/prometheus#Histogram), non-global registries (https://godoc.org/github.com/prometheus/client_golang/prometheus#Registry), functions for pushing metrics (https://godoc.org/github.com/prometheus/client_golang/prometheus/push) to Prometheus PushGateways (/docs/instrumenting/pushing/), bridging Prometheus and Graphite (https://godoc.org/github.com/prometheus/client_golang/prometheus/graphite), and more.

Summary

In this guide, you created two sample Go applications that expose metrics to Prometheus---one that exposes only the default Go metrics and one that also exposes a custom Prometheus counter---and configured a Prometheus instance to scrape metrics from those applications.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

SPECIFICATIONS

UNDERSTANDING AND USING THE MULTI-TARGET EXPORTER PATTERN

This guide will introduce you to the multi-target exporter pattern. To achieve this we will:

- describe the multi-target exporter pattern and why it is used,
- run the blackbox (https://github.com/prometheus/blackbox_exporter) exporter as an example of the pattern,
- configure a custom query module for the blackbox exporter,
- let the blackbox exporter run basic metric queries against the Prometheus website (https://prometheus.io),
- examine a popular pattern of configuring Prometheus to scrape exporters using relabeling.

The multi-target exporter pattern?

By multi-target exporter (/docs/instrumenting/exporters/) pattern we refer to a specific design, in which:

- the exporter will get the target's metrics via a network protocol.
- the exporter does not have to run on the machine the metrics are taken from.
- the exporter gets the targets and a query config string as parameters of Prometheus' GET request.
- the exporter subsequently starts the scrape after getting Prometheus' GET requests and once it is done with scraping.
- the exporter can query multiple targets.

This pattern is only used for certain exporters, such as the blackbox (https://github.com/prometheus/blackbox_exporter) and the SNMP exporter (https://github.com/prometheus/snmp_exporter).

The reason is that we either can't run an exporter on the targets, e.g. network gear speaking SNMP, or that we are explicitly interested in the distance, e.g. latency and reachability of a website from a specific point outside of our network, a common use case for the blackbox (https://github.com/prometheus/blackbox_exporter) exporter.

Running multi-target exporters

Multi-target exporters are flexible regarding their environment and can be run in many ways. As regular programs, in containers, as background services, on baremetal, on virtual machines. Because they are queried and do query over network they do need appropriate open ports. Otherwise they are frugal.

Now let's try it out for yourself!

Use Docker (https://www.docker.com/) to start a blackbox exporter container by running this in a terminal. Depending on your system configuration you might need to prepend the command with a `sudo`:

```
docker run -p 9115:9115 prom/blackbox-exporter
```

You should see a few log lines and if everything went well the last one should report `msg="Listening on address"` as seen here:

```
level=info ts=2018-10-17T15:41:35.4997596Z caller=main.go:324 msg="Listening on address
```

Basic querying of multi-target exporters

There are two ways of querying:

1. Querying the exporter itself. It has its own metrics, usually available at `/metrics`.
2. Querying the exporter to scrape another target. Usually available at a "descriptive" endpoint, e.g. `/probe`. This is likely what you are primarily interested in, when using multi-target exporters.

You can manually try the first query type with curl in another terminal or use this link (http://localhost:9115/metrics):

```
curl 'localhost:9115/metrics'
```

The response should be something like this:

```
# HELP blackbox_exporter_build_info A metric with a constant '1' value labeled by versi
# TYPE blackbox_exporter_build_info gauge
blackbox_exporter_build_info{branch="HEAD",goversion="go1.10",revision="4a22506cf0cf139
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0
go_gc_duration_seconds{quantile="0.25"} 0
go_gc_duration_seconds{quantile="0.5"} 0
go_gc_duration_seconds{quantile="0.75"} 0
go_gc_duration_seconds{quantile="1"} 0
go_gc_duration_seconds_sum 0
go_gc_duration_seconds_count 0
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 9

[…]

# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 0.05
# HELP process_max_fds Maximum number of open file descriptors.
# TYPE process_max_fds gauge
process_max_fds 1.048576e+06
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 7
# HELP process_resident_memory_bytes Resident memory size in bytes.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 7.8848e+06
# HELP process_start_time_seconds Start time of the process since unix epoch in seconds
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.54115492874e+09
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.5609856e+07
```

Those are metrics in the Prometheus format (/docs/instrumenting/exposition_formats/#text-format-example). They come from the exporter's instrumentation (/docs/practices/instrumentation/) and tell us about the state of the exporter itself while it is running. This is called whitebox monitoring and very useful in daily ops practice. If you are curious, try out our guide on how to instrument your own applications (https://prometheus.io/docs/guides/go-application/).

For the second type of querying we need to provide a target and module as parameters in the HTTP GET Request. The target is a URI or IP and the module must defined in the exporter's configuration. The blackbox exporter container comes with a meaningful default configuration.
We will use the target `prometheus.io` and the predefined module `http_2xx`. It tells the exporter to make a GET request like a browser would if you go to `prometheus.io` and to expect a 200 OK (https://en.wikipedia.org/wiki/List_of_HTTP_status_codes#2xx_Success) response.

You can now tell your blackbox exporter to query `prometheus.io` in the terminal with curl:

```
curl 'localhost:9115/probe?target=prometheus.io&module=http_2xx'
```

This will return a lot of metrics:

```
# HELP probe_dns_lookup_time_seconds Returns the time taken for probe dns lookup in sec
# TYPE probe_dns_lookup_time_seconds gauge
probe_dns_lookup_time_seconds 0.061087943
# HELP probe_duration_seconds Returns how long the probe took to complete in seconds
# TYPE probe_duration_seconds gauge
probe_duration_seconds 0.065580871
# HELP probe_failed_due_to_regex Indicates if probe failed due to regex
# TYPE probe_failed_due_to_regex gauge
probe_failed_due_to_regex 0
# HELP probe_http_content_length Length of http content response
# TYPE probe_http_content_length gauge
probe_http_content_length 0
# HELP probe_http_duration_seconds Duration of http request by phase, summed over all r
# TYPE probe_http_duration_seconds gauge
probe_http_duration_seconds{phase="connect"} 0
probe_http_duration_seconds{phase="processing"} 0
probe_http_duration_seconds{phase="resolve"} 0.061087943
probe_http_duration_seconds{phase="tls"} 0
probe_http_duration_seconds{phase="transfer"} 0
# HELP probe_http_redirects The number of redirects
# TYPE probe_http_redirects gauge
probe_http_redirects 0
# HELP probe_http_ssl Indicates if SSL was used for the final redirect
# TYPE probe_http_ssl gauge
probe_http_ssl 0
# HELP probe_http_status_code Response HTTP status code
# TYPE probe_http_status_code gauge
probe_http_status_code 0
# HELP probe_http_version Returns the version of HTTP of the probe response
# TYPE probe_http_version gauge
probe_http_version 0
# HELP probe_ip_protocol Specifies whether probe ip protocol is IP4 or IP6
# TYPE probe_ip_protocol gauge
probe_ip_protocol 6
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 0
```

Notice that almost all metrics have a value of `0`. The last one reads `probe_success 0`. This means the prober could not successfully reach `prometheus.io`. The reason is hidden in the metric `probe_ip_protocol` with the value `6`. By default the prober uses

IPv6 (https://en.wikipedia.org/wiki/IPv6) until told otherwise. But the Docker daemon blocks IPv6 until told otherwise. Hence our blackbox exporter running in a Docker container can't connect via IPv6.

We could now either tell Docker to allow IPv6 or the blackbox exporter to use IPv4. In the real world both can make sense and as so often the answer to the question "what is to be done?" is "it depends". Because this is an exporter guide we will change the exporter and take the opportunity to configure a custom module.

Configuring modules

The modules are predefined in a file inside the docker container called `config.yml` which is a copy of blackbox.yml (https://github.com/prometheus/blackbox_exporter/blob/master/blackbox.yml) in the github repo.

We will copy this file, adapt (https://github.com/prometheus/blackbox_exporter/blob/master/CONFIGURATION.md) it to our own needs and tell the exporter to use our config file instead of the one included in the container.

First download the file using curl or your browser:

```
curl -o blackbox.yml https://raw.githubusercontent.com/prometheus/blackbox_exporter/mas
```

Open it in an editor. The first few lines look like this:

```
modules:
  http_2xx:
    prober: http
  http_post_2xx:
    prober: http
    http:
      method: POST
```

YAML (https://en.wikipedia.org/wiki/YAML) uses whitespace indentation to express hierarchy, so you can recognise that two `modules` named `http_2xx` and `http_post_2xx` are defined, and that they both have a prober `http` and for one the

method value is specifically set to `POST`.
You will now change the module `http_2xx` by setting the `preferred_ip_protocol` of the prober `http` explicitly to the string `ip4`.

```
modules:
  http_2xx:
    prober: http
    http:
      preferred_ip_protocol: "ip4"
  http_post_2xx:
    prober: http
    http:
      method: POST
```

If you want to know more about the available probers and options check out the documentation (https://github.com/prometheus/blackbox_exporter/blob/master/CONFIGURATION.md).

Now we need to tell the blackbox exporter to use our freshly changed file. You can do that with the flag `--config.file="blackbox.yml"`. But because we are using Docker, we first must make this file available (https://docs.docker.com/storage/bind-mounts/) inside the container using the `--mount` command.

NOTE: If you are using macOS you first need to allow the Docker daemon to access the directory in which your `blackbox.yml` is. You can do that by clicking on the little Docker whale in menu bar and then on `Preferences` -> `File Sharing` -> `+`. Afterwards press `Apply & Restart`.

First you stop the old container by changing into its terminal and press `ctrl+c`. Make sure you are in the directory containing your `blackbox.yml`. Then you run this command. It is long, but we will explain it:

```
docker \
  run -p 9115:9115 \
  --mount type=bind,source="$(pwd)"/blackbox.yml,target=/blackbox.yml,readonly \
  prom/blackbox-exporter \
  --config.file="/blackbox.yml"
```

With this command, you told `docker` to:

1. `run` a container with the port `9115` outside the container mapped to the port `9115` inside of the container.
2. `mount` from your current directory (`$(pwd)` stands for print working directory) the file `blackbox.yml` into `/blackbox.yml` in `readonly` mode.
3. use the image `prom/blackbox-exporter` from Docker hub (https://hub.docker.com/r/prom/blackbox-exporter/).
4. run the blackbox-exporter with the flag `--config.file` telling it to use `/blackbox.yml` as config file.

If everything is correct, you should see something like this:

```
level=info ts=2018-10-19T12:40:51.650462756Z caller=main.go:213 msg="Starting blackbox_
level=info ts=2018-10-19T12:40:51.653357722Z caller=main.go:220 msg="Loaded config file
level=info ts=2018-10-19T12:40:51.65349635Z caller=main.go:324 msg="Listening on addres
```

Now you can try our new IPv4-using module `http_2xx` in a terminal:

```
curl 'localhost:9115/probe?target=prometheus.io&module=http_2xx'
```

Which should return Prometheus metrics like this:

```
# HELP probe_dns_lookup_time_seconds Returns the time taken for probe dns lookup in sec
# TYPE probe_dns_lookup_time_seconds gauge
probe_dns_lookup_time_seconds 0.02679421
# HELP probe_duration_seconds Returns how long the probe took to complete in seconds
# TYPE probe_duration_seconds gauge
probe_duration_seconds 0.461619124
# HELP probe_failed_due_to_regex Indicates if probe failed due to regex
# TYPE probe_failed_due_to_regex gauge
probe_failed_due_to_regex 0
# HELP probe_http_content_length Length of http content response
# TYPE probe_http_content_length gauge
probe_http_content_length -1
# HELP probe_http_duration_seconds Duration of http request by phase, summed over all r
# TYPE probe_http_duration_seconds gauge
probe_http_duration_seconds{phase="connect"} 0.062076202999999996
probe_http_duration_seconds{phase="processing"} 0.23481845699999998
probe_http_duration_seconds{phase="resolve"} 0.029594103
probe_http_duration_seconds{phase="tls"} 0.163420078
probe_http_duration_seconds{phase="transfer"} 0.002243199
# HELP probe_http_redirects The number of redirects
# TYPE probe_http_redirects gauge
probe_http_redirects 1
# HELP probe_http_ssl Indicates if SSL was used for the final redirect
# TYPE probe_http_ssl gauge
probe_http_ssl 1
# HELP probe_http_status_code Response HTTP status code
# TYPE probe_http_status_code gauge
probe_http_status_code 200
# HELP probe_http_uncompressed_body_length Length of uncompressed response body
# TYPE probe_http_uncompressed_body_length gauge
probe_http_uncompressed_body_length 14516
# HELP probe_http_version Returns the version of HTTP of the probe response
# TYPE probe_http_version gauge
probe_http_version 1.1
# HELP probe_ip_protocol Specifies whether probe ip protocol is IP4 or IP6
# TYPE probe_ip_protocol gauge
probe_ip_protocol 4
# HELP probe_ssl_earliest_cert_expiry Returns earliest SSL cert expiry in unixtime
# TYPE probe_ssl_earliest_cert_expiry gauge
probe_ssl_earliest_cert_expiry 1.581897599e+09
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 1
# HELP probe_tls_version_info Contains the TLS version used
```

```
# TYPE probe_tls_version_info gauge
probe_tls_version_info{version="TLS 1.3"} 1
```

You can see that the probe was successful and get many useful metrics, like latency by phase, status code, ssl status or certificate expiry in Unix time (https://en.wikipedia.org/wiki/Unix_time).
The blackbox exporter also offers a tiny web interface at localhost:9115 (http://localhost:9115) for you to check out the last few probes, the loaded config and debug information. It even offers a direct link to probe `prometheus.io`. Handy if you are wondering why something does not work.

Querying multi-target exporters with Prometheus

So far, so good. Congratulate yourself. The blackbox exporter works and you can manually tell it to query a remote target. You are almost there. Now you need to tell Prometheus to do the queries for us.

Below you find a minimal prometheus config. It is telling Prometheus to scrape the exporter itself as we did before using `curl 'localhost:9115/metrics'`:

NOTE: If you use Docker for Mac or Docker for Windows, you can't use `localhost:9115` in the last line, but must use `host.docker.internal:9115`. This has to do with the virtual machines used to implement Docker on those operating systems. You should not use this in production.

`prometheus.yml` for Linux:

```
global:
  scrape_interval: 5s

scrape_configs:
- job_name: blackbox # To get metrics about the exporter itself
  metrics_path: /metrics
  static_configs:
    - targets:
      - localhost:9115
```

`prometheus.yml` for macOS and Windows:

```
global:
  scrape_interval: 5s

scrape_configs:
- job_name: blackbox # To get metrics about the exporter itself
  metrics_path: /metrics
  static_configs:
    - targets:
      - host.docker.internal:9115
```

Now run a Prometheus container and tell it to mount our config file from above. Because of the way networking on the host is addressable from the container you need to use a slightly different command on Linux than on MacOS and Windows.:

Run Prometheus on Linux (don't use `--network="host"` in production):

```
docker \
  run --network="host"\
  --mount type=bind,source="$(pwd)"/prometheus.yml,target=/prometheus.yml,readonly \
  prom/prometheus \
  --config.file="/prometheus.yml"
```

Run Prometheus on MacOS and Windows:

```
docker \
  run -p 9090:9090 \
  --mount type=bind,source="$(pwd)"/prometheus.yml,target=/prometheus.yml,readonly \
  prom/prometheus \
  --config.file="/prometheus.yml"
```

This command works similarly to running the blackbox exporter using a config file.

If everything worked, you should be able to go to localhost:9090/targets (http://localhost:9090/targets) and see under `blackbox` an endpoint with the state `UP` in green. If you get a red `DOWN` make sure that the blackbox exporter you started above is still running. If you see nothing or a yellow `UNKNOWN` you are really fast and need to give it a few more seconds before reloading your browser's tab.

To tell Prometheus to query `"localhost:9115/probe?target=prometheus.io&module=http_2xx"` you add another scrape job `blackbox-http` where you set the `metrics_path` to `/probe` and the parameters under `params:` in the Prometheus config file `prometheus.yml`:

```
global:
  scrape_interval: 5s

scrape_configs:
- job_name: blackbox # To get metrics about the exporter itself
  metrics_path: /metrics
  static_configs:
    - targets:
      - localhost:9115   # For Windows and macOS replace with - host.docker.internal:91

- job_name: blackbox-http # To get metrics about the exporter’s targets
  metrics_path: /probe
  params:
    module: [http_2xx]
    target: [prometheus.io]
  static_configs:
    - targets:
      - localhost:9115   # For Windows and macOS replace with - host.docker.internal:91
```

After saving the config file switch to the terminal with your Prometheus docker container and stop it by pressing `ctrl+C` and start it again to reload the configuration by using the existing command.

The terminal should return the message `"Server is ready to receive web requests."` and after a few seconds you should start to see colourful graphs in your Prometheus (http://localhost:9090/graph?g0.range_input=5m&g0.stacked=0&g0.expr=probe_http_duration_seconds&g0.tab=0).

This works, but it has a few disadvantages:

1. The actual targets are up in the param config, which is very unusual and hard to understand later.
2. The `instance` label has the value of the blackbox exporter’s address which is technically true, but not what we are interested in.
3. We can’t see which URL we probed. This is unpractical and will also mix up different metrics into one if we probe several URLs.

To fix this, we will use relabeling (/docs/prometheus/latest/configuration/configuration/#relabel_config). Relabeling is useful here because behind the scenes many things in Prometheus are configured with internal labels. The details are complicated and out of scope for this guide. Hence we will limit ourselves to the necessary. But if you want to know more check out this talk (https://www.youtube.com/watch?v=b5-SvvZ7AwI). For now it suffices if you understand this:

- All labels starting with `__` are dropped after the scrape. Most internal labels start with `__`.
- You can set internal labels that are called `__param_<name>`. Those set URL parameter with the key `<name>` for the scrape request.
- There is an internal label `__address__` which is set by the `targets` under `static_configs` and whose value is the hostname for the scrape request. By default it is later used to set the value for the label `instance`, which is attached to each metric and tells you where the metrics came from.

Here is the config you will use to do that. Don't worry if this is a bit much at once, we will go through it step by step:

```
global:
  scrape_interval: 5s

scrape_configs:
- job_name: blackbox # To get metrics about the exporter itself
  metrics_path: /metrics
  static_configs:
    - targets:
      - localhost:9115   # For Windows and macOS replace with - host.docker.internal:91

- job_name: blackbox-http # To get metrics about the exporter's targets
  metrics_path: /probe
  params:
    module: [http_2xx]
  static_configs:
    - targets:
      - http://prometheus.io    # Target to probe with http
      - https://prometheus.io   # Target to probe with https
      - http://example.com:8080 # Target to probe with http on port 8080
  relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
    - source_labels: [__param_target]
      target_label: instance
    - target_label: __address__
      replacement: localhost:9115  # The blackbox exporter's real hostname:port. For Wi
```

So what is new compared to the last config?

`params` does not include `target` anymore. Instead we add the actual targets under `static_configs: targets`. We also use several because we can do that now:

```
  params:
    module: [http_2xx]
  static_configs:
    - targets:
      - http://prometheus.io    # Target to probe with http
      - https://prometheus.io   # Target to probe with https
      - http://example.com:8080 # Target to probe with http on port 8080
```

`relabel_configs` contains the new relabeling rules:

```
relabel_configs:
  - source_labels: [__address__]
    target_label: __param_target
  - source_labels: [__param_target]
    target_label: instance
  - target_label: __address__
    replacement: localhost:9115  # The blackbox exporter's real hostname:port. For Wi
```

Before applying the relabeling rules, the URI of a request Prometheus would make would look like this: `"http://prometheus.io/probe?module=http_2xx"` . After relabeling it will look like this `"http://localhost:9115/probe?target=http://prometheus.io&module=http_2xx"` .

Now let us explore how each rule does that:

First we take the values from the label `__address__` (which contain the values from `targets`) and write them to a new label `__param_target` which will add a parameter `target` to the Prometheus scrape requests:

```
relabel_configs:
  - source_labels: [__address__]
    target_label: __param_target
```

After this our imagined Prometheus request URI has now a target parameter: `"http://prometheus.io/probe?target=http://prometheus.io&module=http_2xx"` .

Then we take the values from the label `__param_target` and create a label instance with the values.

```
relabel_configs:
  - source_labels: [__param_target]
    target_label: instance
```

Our request will not change, but the metrics that come back from our request will now bear a label `instance="http://prometheus.io"` .

After that we write the value `localhost:9115` (the URI of our exporter) to the label `__address__` . This will be used as the hostname and port for the Prometheus scrape requests. So that it queries the exporter and not the target URI directly.

```
  relabel_configs:
    - target_label: __address__
      replacement: localhost:9115  # The blackbox exporter's real hostname:port. For Wi
```

Our request is now `"localhost:9115/probe?target=http://prometheus.io&module=http_2xx"`. This way we can have the actual targets there, get them as `instance` label values while letting Prometheus make a request against the blackbox exporter.

Often people combine these with a specific service discovery. Check out the configuration documentation (/docs/prometheus/latest/configuration/configuration) for more information. Using them is no problem, as these write into the `__address__` label just like `targets` defined under `static_configs`.

That is it. Restart the Prometheus docker container and look at your metrics (http://localhost:9090/graph?g0.range_input=30m&g0.stacked=0&g0.expr=probe_http_duration_seconds&g0.tab=0). Pay attention that you selected the period of time when the metrics were actually collected.

SUMMARY

In this guide, you learned how the multi-target exporter pattern works, how to run a blackbox exporter with a customised module, and to configure Prometheus using relabeling to scrape metrics with prober labels.

MONITORING LINUX HOST METRICS WITH THE NODE EXPORTER

The Prometheus **Node Exporter** (https://github.com/prometheus/node_exporter) exposes a wide variety of hardware- and kernel-related metrics.

In this guide, you will:

- Start up a Node Exporter on `localhost`
- Start up a Prometheus instance on `localhost` that's configured to scrape metrics from the running Node Exporter
- Installing and running the Node Exporter
- Node Exporter metrics
- Configuring your Prometheus instances
- Exploring Node Exporter metrics through the Prometheus expression browser

NOTE: While the Prometheus Node Exporter is for *nix systems, there is the Windows exporter (https://github.com/prometheus-community/windows_exporter) for Windows that serves an analogous purpose.

Installing and running the Node Exporter

The Prometheus Node Exporter is a single static binary that you can install via tarball. Once you've downloaded it from the Prometheus downloads page (/download#node_exporter) extract it, and run it:

```
# NOTE: Replace the URL with one from the above mentioned "downloads" page.
# <VERSION>, <OS>, and <ARCH> are placeholders.
wget https://github.com/prometheus/node_exporter/releases/download/v<VERSION>/node_exporter-<VERSION>.<OS>-<ARCH>.tar.gz
tar xvfz node_exporter-*.*-amd64.tar.gz
cd node_exporter-*.*-amd64
./node_exporter
```

You should see output like this indicating that the Node Exporter is now running and exposing metrics on port 9100:

```
INFO[0000] Starting node_exporter (version=0.16.0, branch=HEAD, revision=d42bd70f4363dced6b77d8fc311ea57b63387e4f)  source="
INFO[0000] Build context (go=go1.9.6, user=root@a67a9bc13a69, date=20180515-15:53:28)  source="node_exporter.go:83"
INFO[0000] Enabled collectors:                           source="node_exporter.go:90"
INFO[0000]  - boottime                                   source="node_exporter.go:97"
...
INFO[0000] Listening on :9100                            source="node_exporter.go:111"
```

Node Exporter metrics

Once the Node Exporter is installed and running, you can verify that metrics are being exported by cURLing the `/metrics` endpoint:

```
curl http://localhost:9100/metrics
```

You should see output like this:

```
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 3.8996e-05
go_gc_duration_seconds{quantile="0.25"} 4.5926e-05
go_gc_duration_seconds{quantile="0.5"} 5.846e-05
# etc.
```

Success! The Node Exporter is now exposing metrics that Prometheus can scrape, including a wide variety of system metrics further down in the output (prefixed with `node_`). To view those metrics (along with help and type information):

```
curl http://localhost:9100/metrics | grep "node_"
```

Configuring your Prometheus instances

Your locally running Prometheus instance needs to be properly configured in order to access Node Exporter metrics. The following `prometheus.yml` (../../prometheus/latest/configuration/configuration/) example configuration file will tell the Prometheus instance to scrape, and how frequently, from the Node Exporter via `localhost:9100` :

```
global:
  scrape_interval: 15s

scrape_configs:
- job_name: node
  static_configs:
  - targets: ['localhost:9100']
```

To install Prometheus, download the latest release (/download) for your platform and untar it:

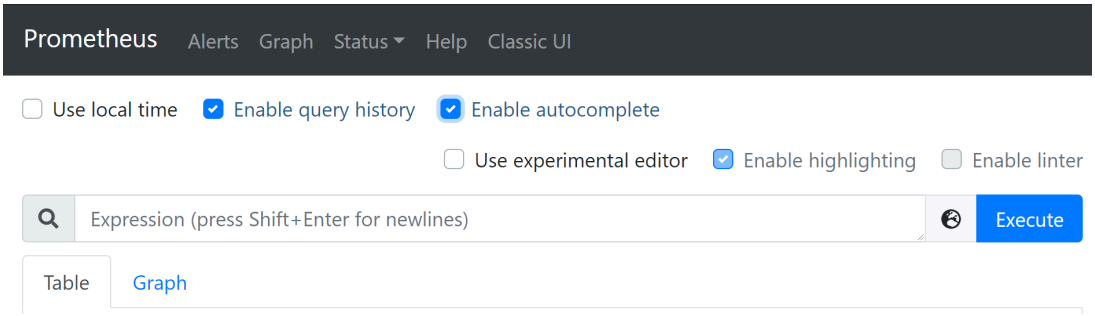
```
wget https://github.com/prometheus/prometheus/releases/download/v*/prometheus-*.*-amd64.tar.gz
tar xvf prometheus-*.*-amd64.tar.gz
cd prometheus-*.*
```

Once Prometheus is installed you can start it up, using the `--config.file` flag to point to the Prometheus configuration that you created above:

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

Exploring Node Exporter metrics through the Prometheus expression browser

Now that Prometheus is scraping metrics from a running Node Exporter instance, you can explore those metrics using the Prometheus UI (aka the expression browser (/docs/visualization/browser)). Navigate to `localhost:9090/graph` in your browser and use the main expression bar at the top of the page to enter expressions. The expression bar looks like this:



Metrics specific to the Node Exporter are prefixed with `node_` and include metrics like `node_cpu_seconds_total` and `node_exporter_build_info`.

Click on the links below to see some example metrics:

Metric	Me
`rate(node_cpu_seconds_total{mode="system"}[1m])` (http://localhost:9090/graph?g0.range_input=1h&g0.expr=rate(node_cpu_seconds_total%7Bmode%3D%22system%22%7D%5B1m%5D)&g0.tab=1)	The ave am of C tim spe syst mo per sec ove last min (in sec
`node_filesystem_avail_bytes` (http://localhost:9090/graph?g0.range_input=1h&g0.expr=node_filesystem_avail_bytes&g0.tab=1)	The files spa ava to r roo use byt
`rate(node_network_receive_bytes_total[1m])` (http://localhost:9090/graph?g0.range_input=1h&g0.expr=rate(node_network_receive_bytes_total%5B1m%5D)&g0.tab=1)	The ave net traf rec per sec ove last min (in b

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

TUTORIALS

SPECIFICATIONS

USING PROMETHEUS AS YOUR OPENTELEMETRY BACKEND

Prometheus supports OTLP (https://opentelemetry.io/docs/specs/otlp) (aka "OpenTelemetry Protocol") ingestion through HTTP

- Enable the OTLP receiver
- Enable out-of-order ingestion
- Promoting resource attributes
- UTF-8
- Delta Temporality

(https://opentelemetry.io/docs/specs/otlp/#otlphttp).

Enable the OTLP receiver

By default, the OTLP receiver is disabled. This is because Prometheus can work without any authentication, so it would not be safe to accept incoming traffic unless explicitly configured.

To enable the receiver you need to toggle the flag `--enable-feature=otlp-write-receiver` .

```
$ prometheus --enable-feature=otlp-write-receiver
```

Enable out-of-order ingestion

There are multiple reasons why you might want to enable out-of-order ingestion.

For example, the OpenTelemetry collector encourages batching and you could have multiple replicas of the collector sending data to Prometheus. Because there is no mechanism ordering those samples they could get out-of-order.

To enable out-of-order ingestion you need to extend the Prometheus configuration file with the following:

```
storage:
  tsdb:
    out_of_order_time_window: 30m
```

30 minutes of out-of-order have been enough for most cases but don't hesitate to adjust this value to your needs.

Promoting resource attributes

Based on experience and conversations with our community, we've found that out of all the commonly seen resource attributes, these are the ones that are most frequently promoted by our users:

```
- service.instance.id
- service.name
- service.namespace
- cloud.availability_zone
- cloud.region
- container.name
- deployment.environment
- k8s.cluster.name
- k8s.container.name
- k8s.cronjob.name
- k8s.daemonset.name
- k8s.deployment.name
- k8s.job.name
- k8s.namespace.name
- k8s.pod.name
- k8s.replicaset.name
- k8s.statefulset.name
```

By default Prometheus won't be promoting any attributes. If you'd like to promote any of them, you can do so in this section of the Prometheus configuration file:

```
otlp:
  resource_attributes:
    - service.instance.id
    - deployment.environment
    - k8s.cluster.name
    - ...
```

UTF-8

The UTF-8 support for Prometheus is not ready yet so both the Prometheus Remote Write Exporter and the OTLP Ingestion endpoint still rely on the Prometheus normalization translator package from OpenTelemetry (https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/pkg/translator/prometheus).

So if you are sending non-valid characters to Prometheus, they will be replaced with an underscore `_` character.

Once the UTF-8 feature is merged into Prometheus, we will revisit this.

Delta Temporality

The OpenTelemetry specification says (https://opentelemetry.io/docs/specs/otel/metrics/data-model/#temporality) that both Delta temporality and Cumulative temporality are supported.

While Delta temporality is common in systems like statsd and graphite, cumulative temporality is the default temporality for Prometheus.

Today Prometheus does not have support for delta temporality but we are learning from the OpenTelemetry community and we are considering adding support for it in the future.

If you are coming from a delta temporality system we recommend that you use the delta to cumulative processor (https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/processor/deltatocumulativeprocessor) in your OTel pipeline.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

DOCKER SWARM

Prometheus can discover targets in a Docker Swarm (https://docs.docker.com/engine/swarm/) cluster, as of v2.20.0. This guide demonstrates how to use that service discovery mechanism.

Docker Swarm service discovery architecture

The Docker Swarm service discovery (https://prometheus.io/docs/prometheus/latest/configuration/configuration/#dockerswarm_sd_config) contains 3 different roles: nodes, services, and tasks.

The first role, **nodes**, represents the hosts that are part of the Swarm. It can be used to automatically monitor the Docker daemons or the Node Exporters who run on the Swarm hosts.

The second role, **tasks**, represents any individual container deployed in the swarm. Each task gets its associated service labels. One service can be backed by one or multiple tasks.

The third one, **services**, will discover the services deployed in the swarm. It will discover the ports exposed by the services. Usually you will want to use the tasks role instead of this one.

Prometheus will only discover tasks and service that expose ports.

> **NOTE:** The rest of this post assumes that you have a Swarm running.

Setting up Prometheus

For this guide, you need to setup Prometheus (https://prometheus.io/docs/prometheus/latest/getting_started/). We will assume that Prometheus runs on a Docker Swarm manager node and has access to the Docker socket at `/var/run/docker.sock`.

Monitoring Docker daemons

Let's dive into the service discovery itself.

Docker itself, as a daemon, exposes metrics (https://docs.docker.com/config/daemon/prometheus/) that can be ingested by a Prometheus server.

You can enable them by editing `/etc/docker/daemon.json` and setting the following properties:

```
{
  "metrics-addr" : "0.0.0.0:9323",
  "experimental" : true
}
```

Instead of `0.0.0.0`, you can set the IP of the Docker Swarm node.

A restart of the daemon is required to take the new configuration into account.

The Docker documentation (https://docs.docker.com/config/daemon/prometheus/) contains more info about this.

Then, you can configure Prometheus to scrape the Docker daemon, by providing the following `prometheus.yml` file:

```
scrape_configs:
  # Make Prometheus scrape itself for metrics.
  - job_name: 'prometheus'
    static_configs:
    - targets: ['localhost:9090']

  # Create a job for Docker daemons.
  - job_name: 'docker'
    dockerswarm_sd_configs:
      - host: unix:///var/run/docker.sock
        role: nodes
    relabel_configs:
      # Fetch metrics on port 9323.
      - source_labels: [__meta_dockerswarm_node_address]
        target_label: __address__
        replacement: $1:9323
      # Set hostname as instance label
      - source_labels: [__meta_dockerswarm_node_hostname]
        target_label: instance
```

For the nodes role, you can also use the `port` parameter of `dockerswarm_sd_configs`. However, using `relabel_configs` is recommended as it enables Prometheus to reuse the same API calls across identical Docker Swarm configurations.

Monitoring Containers

Let's now deploy a service in our Swarm. We will deploy cadvisor (https://github.com/google/cadvisor), which exposes container resources metrics:

```
docker service create --name cadvisor -l prometheus-job=cadvisor \
    --mode=global --publish target=8080,mode=host \
    --mount type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock,ro \
    --mount type=bind,src=/,dst=/rootfs,ro \
    --mount type=bind,src=/var/run,dst=/var/run \
    --mount type=bind,src=/sys,dst=/sys,ro \
    --mount type=bind,src=/var/lib/docker,dst=/var/lib/docker,ro \
    google/cadvisor -docker_only
```

This is a minimal `prometheus.yml` file to monitor it:

```
scrape_configs:
  # Make Prometheus scrape itself for metrics.
  - job_name: 'prometheus'
    static_configs:
    - targets: ['localhost:9090']

  # Create a job for Docker Swarm containers.
  - job_name: 'dockerswarm'
    dockerswarm_sd_configs:
      - host: unix:///var/run/docker.sock
        role: tasks
    relabel_configs:
      # Only keep containers that should be running.
      - source_labels: [__meta_dockerswarm_task_desired_state]
        regex: running
        action: keep
      # Only keep containers that have a `prometheus-job` label.
      - source_labels: [__meta_dockerswarm_service_label_prometheus_job]
        regex: .+
        action: keep
      # Use the prometheus-job Swarm label as Prometheus job label.
      - source_labels: [__meta_dockerswarm_service_label_prometheus_job]
        target_label: job
```

Let's analyze each part of the relabel configuration (https://prometheus.io/docs/prometheus/latest/configuration/configuration/#relabel_config).

```
- source_labels: [__meta_dockerswarm_task_desired_state]
  regex: running
  action: keep
```

Docker Swarm exposes the desired state of the tasks (https://docs.docker.com/engine/swarm/how-swarm-mode-works/swarm-task-states/) over the API. In out example, we only **keep** the targets that should be running. It prevents monitoring tasks that should be shut down.

```
- source_labels: [__meta_dockerswarm_service_label_prometheus_job]
  regex: .+
  action: keep
```

When we deployed our cadvisor, we have added a label `prometheus-job=cadvisor`. As Prometheus fetches the tasks labels, we can instruct it to **only** keep the targets which have a `prometheus-job` label.

```
- source_labels: [__meta_dockerswarm_service_label_prometheus_job]
  target_label: job
```

That last part takes the label `prometheus-job` of the task and turns it into a target label, overwriting the default `dockerswarm` job label that comes from the scrape config.

Discovered labels

The Prometheus Documentation (https://prometheus.io/docs/prometheus/latest/configuration/configuration/#dockerswarm_sd_config) contains the full list of labels, but here are other relabel configs that you might find useful.

Scraping metrics via a certain network only

```
- source_labels: [__meta_dockerswarm_network_name]
  regex: ingress
  action: keep
```

Scraping global tasks only

Global tasks run on every daemon.

```
- source_labels: [__meta_dockerswarm_service_mode]
  regex: global
  action: keep
- source_labels: [__meta_dockerswarm_task_port_publish_mode]
  regex: host
  action: keep
```

Adding a docker_node label to the targets

```
- source_labels: [__meta_dockerswarm_node_hostname]
  target_label: docker_node
```

Connecting to the Docker Swarm

The above `dockerswarm_sd_configs` entries have a field host:

```
host: unix:///var/run/docker.sock
```

That is using the Docker socket. Prometheus offers additional configuration options (https://prometheus.io/docs/prometheus/latest/configuration/configuration/#dockerswarm_sd_config) to connect to Swarm using HTTP and HTTPS, if you prefer that over the unix socket.

Conclusion

There are many discovery labels you can play with to better determine which targets to monitor and how, for the tasks, there is more than 25 labels available. Don't hesitate to look at the "Service Discovery" page of your Prometheus server (under the "Status" menu) to see all the discovered labels.

The service discovery makes no assumptions about your Swarm stack, in such a way that given proper configuration, this should be pluggable to any existing stack.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

TUTORIALS

SPECIFICATIONS

USING THE PROMETHEUS QUERY LOG

Prometheus has the ability to log all the queries run by the engine to a log file, as of 2.16.0. This guide demonstrates how to use that log file, which fields it contains, and provides advanced tips about how to operate the log file.

Enable the query log

The query log can be toggled at runtime. It can therefore be activated when you want to investigate slownesses or high load on your Prometheus instance.

To enable or disable the query log, two steps are needed:

1. Adapt the configuration to add or remove the query log configuration.
2. Reload the Prometheus server configuration.

Logging all the queries to a file

This example demonstrates how to log all the queries to a file called `/prometheus/query.log`. We will assume that `/prometheus` is the data directory and that Prometheus has write access to it.

First, adapt the `prometheus.yml` configuration file:

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s
  query_log_file: /prometheus/query.log
scrape_configs:
- job_name: 'prometheus'
  static_configs:
  - targets: ['localhost:9090']
```

Then, reload (/docs/prometheus/latest/management_api/#reload) the Prometheus configuration:

```
$ curl -X POST http://127.0.0.1:9090/-/reload
```

Or, if Prometheus is not launched with `--web.enable-lifecycle`, and you're not running on Windows, you can trigger the reload by sending a SIGHUP to the Prometheus process.

The file `/prometheus/query.log` should now exist and all the queries will be logged to that file.

To disable the query log, repeat the operation but remove `query_log_file` from the configuration.

Verifying if the query log is enabled

Prometheus conveniently exposes metrics that indicates if the query log is enabled and working:

```
# HELP prometheus_engine_query_log_enabled State of the query log.
# TYPE prometheus_engine_query_log_enabled gauge
prometheus_engine_query_log_enabled 0
# HELP prometheus_engine_query_log_failures_total The number of query log failure
# TYPE prometheus_engine_query_log_failures_total counter
prometheus_engine_query_log_failures_total 0
```

The first metric, `prometheus_engine_query_log_enabled` is set to 1 of the query log is enabled, and 0 otherwise. The second one, `prometheus_engine_query_log_failures_total`, indicates the number of queries that could not be logged.

Format of the query log

The query log is a JSON-formatted log. Here is an overview of the fields present for a query:

```
{
    "params": {
        "end": "2020-02-08T14:59:50.368Z",
        "query": "up == 0",
        "start": "2020-02-08T13:59:50.368Z",
        "step": 5
    },
    "stats": {
        "timings": {
            "evalTotalTime": 0.000447452,
            "execQueueTime": 7.599e-06,
            "execTotalTime": 0.000461232,
            "innerEvalTime": 0.000427033,
            "queryPreparationTime": 1.4177e-05,
            "resultSortTime": 6.48e-07
        }
    },
    "ts": "2020-02-08T14:59:50.387Z"
}
```

- `params`: The query. The start and end timestamp, the step and the actual query statement.
- `stats`: Statistics. Currently, it contains internal engine timers.
- `ts`: The timestamp when the query ended.

Additionally, depending on what triggered the request, you will have additional fields in the JSON lines.

API Queries and consoles

HTTP requests contain the client IP, the method, and the path:

```
{
    "httpRequest": {
        "clientIP": "127.0.0.1",
        "method": "GET",
        "path": "/api/v1/query_range"
    }
}
```

The path will contain the web prefix if it is set, and can also point to a console.

The client IP is the network IP address and does not take into consideration the headers like `X-Forwarded-For`. If you wish to log the original caller behind a proxy, you need to do so in the proxy itself.

Recording rules and alerts

Recording rules and alerts contain a ruleGroup element which contains the path of the file and the name of the group:

```
{
    "ruleGroup": {
        "file": "rules.yml",
        "name": "partners"
    }
}
```

Rotating the query log

Prometheus will not rotate the query log itself. Instead, you can use external tools to do so.

One of those tools is logrotate. It is enabled by default on most Linux distributions.

Here is an example of file you can add as `/etc/logrotate.d/prometheus`:

```
/prometheus/query.log {
    daily
    rotate 7
    compress
    delaycompress
    postrotate
        killall -HUP prometheus
    endscript
}
```

That will rotate your file daily and keep one week of history.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

TUTORIALS

SPECIFICATIONS

SECURING PROMETHEUS API AND UI ENDPOINTS USING TLS ENCRYPTION

Prometheus supports Transport Layer Security (https://en.wikipedia.org/wiki/Transport_Layer_Security) (TLS) encryption for connections to Prometheus instances (i.e. to the expression browser or HTTP API (../../prometheus/latest/querying/api)). If you would like to enforce TLS for those connections, you would need to create a specific web configuration file.

NOTE: This guide is about TLS connections *to* Prometheus instances. TLS is also supported for connections *from* Prometheus instances to scrape targets (../../prometheus/latest/configuration/configuration/#tls_config).

Pre-requisites

Let's say that you already have a Prometheus instance up and running, and you want to adapt it. We will not cover the initial Prometheus setup in this guide.

Let's say that you want to run a Prometheus instance served with TLS, available at the `example.com` domain (which you own).

Let's also say that you've generated the following using OpenSSL (https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs) or an analogous tool:

- an SSL certificate at `/home/prometheus/certs/example.com/example.com.crt`
- an SSL key at `/home/prometheus/certs/example.com/example.com.key`

You can generate a self-signed certificate and private key using this command:

```
mkdir -p /home/prometheus/certs/example.com && cd /home/prometheus/certs/certs/ex
openssl req \
  -x509 \
  -newkey rsa:4096 \
  -nodes \
  -keyout example.com.key \
  -out example.com.crt
```

Fill out the appropriate information at the prompts, and make sure to enter `example.com` at the `Common Name` prompt.

Prometheus configuration

Below is an example `web-config.yml` (https://prometheus.io/docs/prometheus/latest/configuration/https/) configuration file. With this configuration, Prometheus will serve all its endpoints behind TLS.

```
tls_server_config:
  cert_file: /home/prometheus/certs/example.com/example.com.crt
  key_file: /home/prometheus/certs/example.com/example.com.key
```

To make Prometheus use this config, you will need to call it with the flag `--web.config.file` .

```
prometheus \
  --config.file=/path/to/prometheus.yml \
  --web.config.file=/path/to/web-config.yml \
  --web.external-url=https://example.com/
```

The `--web.external-url=` flag is optional here.

Testing

If you'd like to test out TLS locally using the `example.com` domain, you can add an entry to your `/etc/hosts` file that re-routes `example.com` to `localhost`:

```
127.0.0.1     example.com
```

You can then use cURL to interact with your local Prometheus setup:

```
curl --cacert /home/prometheus/certs/example.com/example.com.crt \
  https://example.com/api/v1/label/job/values
```

You can connect to the Prometheus server without specifying certs using the `--insecure` or `-k` flag:

```
curl -k https://example.com/api/v1/label/job/values
```

WHAT IS PROMETHEUS ?

Prometheus is a system monitoring and alerting system. It was opensourced by SoundCloud in 2012 and is the second project both to join and to graduate within Cloud Native Computing Foundation after Kubernetes. Prometheus stores all metrics data as time series, i.e metrics information is stored along with the timestamp at which it was recorded, optional key-value pairs called as labels can also be stored along with metrics.

WHAT ARE METRICS AND WHY IS IT IMPORTANT?

Metrics in layperson terms is a standard for measurement. What we want to measure depends from application to application. For a web server it can be request times, for a database it can be CPU usage or number of active connections etc.

Metrics play an important role in understanding why your application is working in a certain way. If you run a web application and someone comes up to you and says that the application is slow, you will need some information to find out what is happening with your application. For example the application can become slow when the number of requests are high. If you have the request count metric you can spot the reason and increase the number of servers to handle the heavy load. Whenever you are defining the metrics for your application you must put on your detective hat and ask this question **what all information will be important for me to debug if any issue occurs in my application?**

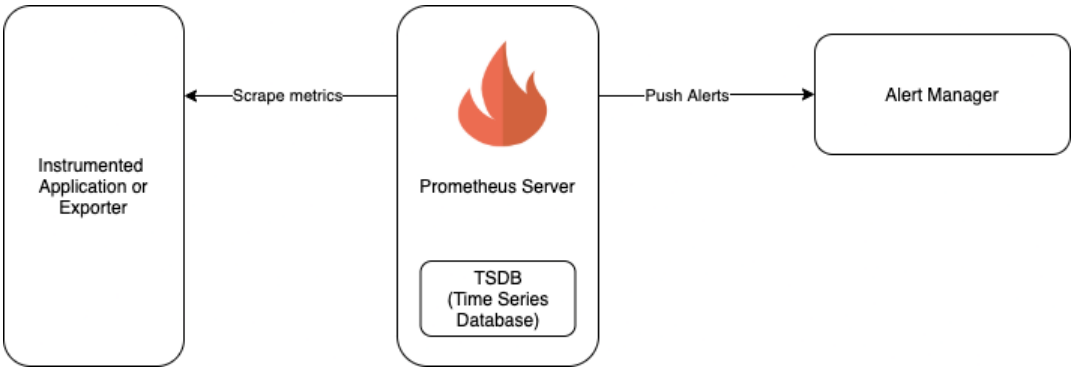
BASIC ARCHITECTURE OF PROMETHEUS

The basic components of a Prometheus setup are:

- Prometheus Server (the server which scrapes and stores the metrics data).
- Targets to be scraped, for example an instrumented application that exposes its metrics, or an exporter that exposes metrics of another

application.
- Alertmanager to raise alerts based on preset rules.

(Note: Apart from this Prometheus has push_gateway which is not covered here).



(/assets/tutorial/architecture.png)

Let's consider a web server as an example application and we want to extract a certain metric like the number of API calls processed by the web server. So we add certain instrumentation code using the Prometheus client library and expose the metrics information. Now that our web server exposes its metrics we can configure Prometheus to scrape it. Now Prometheus is configured to fetch the metrics from the web server which is listening on xyz IP address port 7500 at a specific time interval, say, every minute.

At 11:00:00 when I make the server public for consumption, the application calculates the request count and exposes it, Prometheus simultaneously scrapes the count metric and stores the value as 0.

By 11:01:00 one request is processed. The instrumentation logic in the server increments the count to 1. When Prometheus scrapes the metric the value of count is 1 now.

By 11:02:00 two more requests are processed and the request count is 1+2 = 3 now. Similarly metrics are scraped and stored.

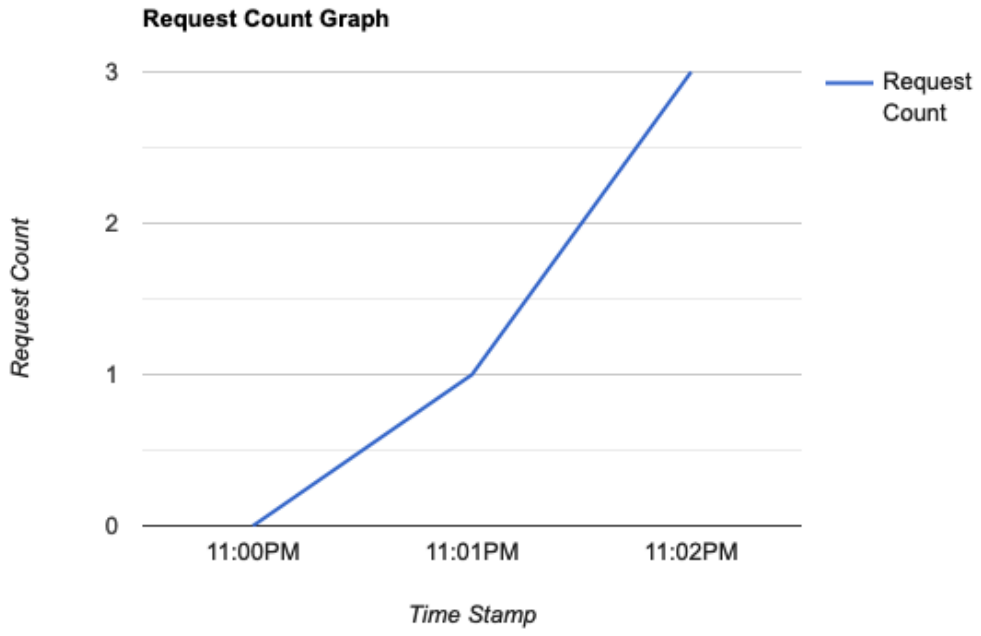
The user can control the frequency at which metrics are scraped by Prometheus.

Time Stamp	Request Count (metric)
11:00:00	0
11:01:00	1
11:02:00	3

(Note: This table is just a representation for understanding purposes. Prometheus doesn't store the values in this exact format)

Prometheus also has an API which allows to query metrics which have been stored by scraping. This API is used to query the metrics, create dashboards/charts on it etc. PromQL is used to query these metrics.

A simple Line chart created on the Request Count metric will look like this



(/assets/tutorial/sample_graph.png)

One can scrape multiple useful metrics to understand what is happening in the application and create multiple charts on them. Group the charts into a dashboard and use it to get an overview of the application.

SHOW ME HOW IT IS DONE

Let’s get our hands dirty and setup Prometheus. Prometheus is written using Go (https://golang.org/) and all you need is the binary compiled for your operating system. Download the binary corresponding to your operating system from here (https://prometheus.io/download/) and add the binary to your path.

Prometheus exposes its own metrics which can be consumed by itself or another Prometheus server.

Now that we have Prometheus installed, the next step is to run it. All that we need is just the binary and a configuration file. Prometheus uses yaml files for configuration.

```
global:
  scrape_interval: 15s

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

In the above configuration file we have mentioned the `scrape_interval` , i.e how frequently we want Prometheus to scrape the metrics. We have added `scrape_configs` which has a name and target to scrape the metrics from. Prometheus by default listens on port 9090. So add it to targets.

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

Now we have Prometheus up and running and scraping its own metrics every 15s. Prometheus has standard exporters available to export metrics. Next we will run a node exporter which is an exporter for machine metrics and scrape the same using Prometheus. (Download node metrics exporter. (https://prometheus.io/download/#node_exporter))

Run the node exporter in a terminal.

```
./node_exporter
```

(/assets/tutorial/node_exporter.png)

Next, add node exporter to the list of scrape_configs:

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: ["localhost:9090"]
  - job_name: node_exporter
    static_configs:
      - targets: ["localhost:9100"]
```

In this tutorial we discussed what are metrics and why they are important, basic architecture of Prometheus and how to run Prometheus.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

TYPES OF METRICS.

Prometheus supports four types of metrics, which are - Counter - Gauge - Histogram - Summary

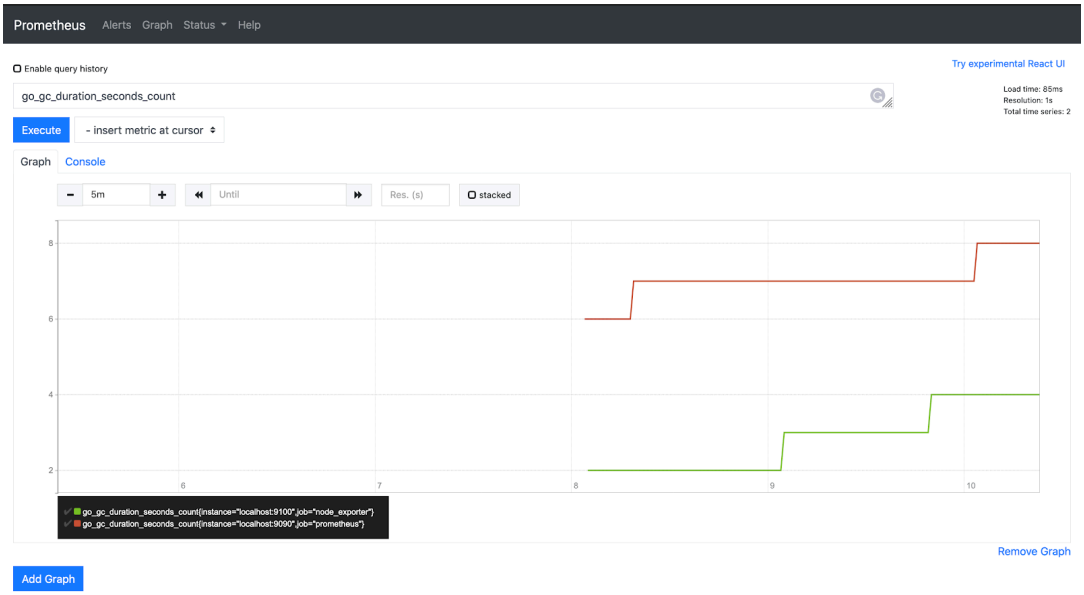
- Counter
- Gauge
- Histogram
- Summary

Counter

Counter is a metric value that can only increase or reset i.e. the value cannot reduce than the previous value. It can be used for metrics like the number of requests, no of errors, etc.

Type the below query in the query bar and click execute.

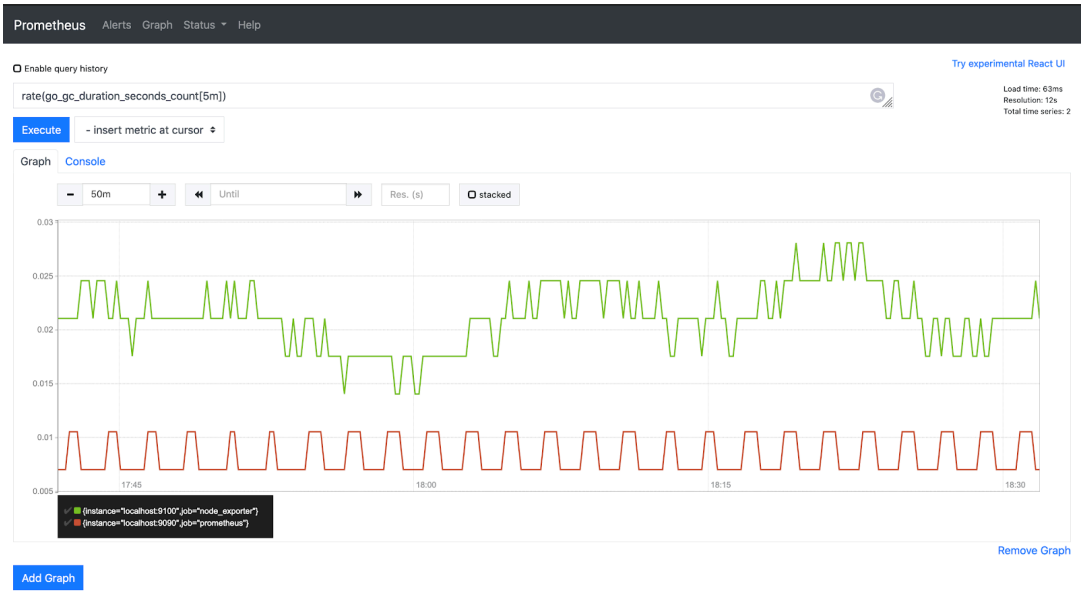
```
go_gc_duration_seconds_count
```



(/assets/tutorial/counter_example.png)

The rate() function in PromQL takes the history of metrics over a time frame and calculates how fast the value is increasing per second. Rate is applicable on counter values only.

```
rate(go_gc_duration_seconds_count[5m])
```

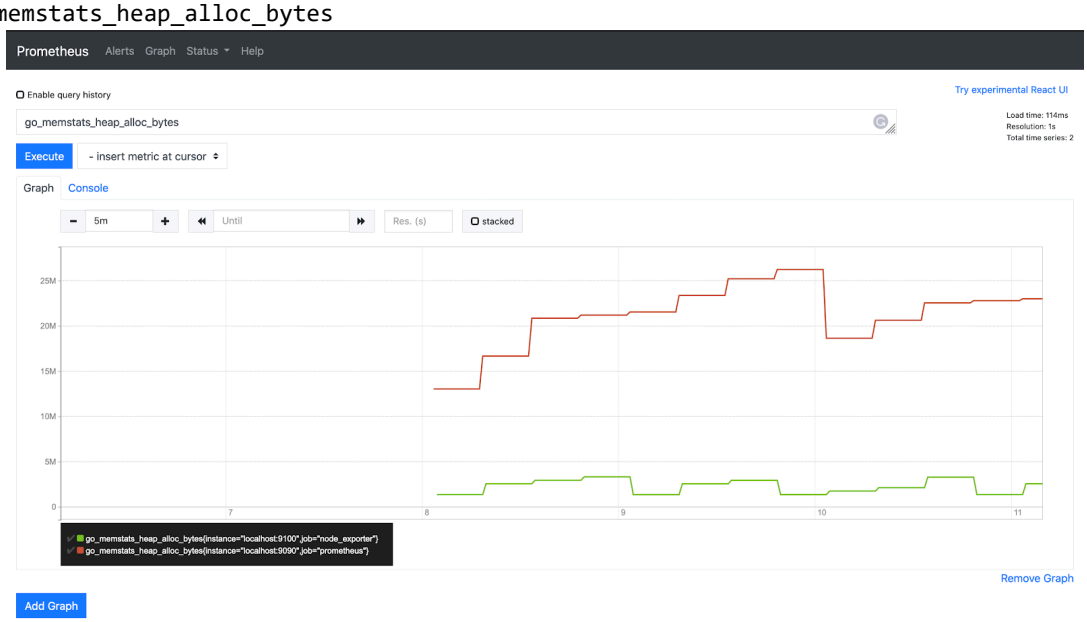


(/assets/tutorial/rate_example.png)

Gauge

Gauge is a number which can either go up or down. It can be used for metrics like the number of pods in a cluster, the number of events in a queue, etc.

```
go_memstats_heap_alloc_bytes
```



(/assets/tutorial/gauge_example.png)

PromQL functions like `max_over_time`, `min_over_time` and `avg_over_time` can be used on gauge metrics

Histogram

Histogram is a more complex metric type when compared to the previous two. Histogram can be used for any calculated value which is counted based on bucket values. Bucket boundaries can be configured by the developer. A common example would be the time it takes to reply to a request, called latency.

Example: Let's assume we want to observe the time taken to process API requests. Instead of storing the request time for each request, histograms allow us to store them in buckets. We define buckets for time taken, for example `lower or equal 0.3`, `le 0.5`, `le 0.7`, `le 1`, and `le 1.2`. So these are our buckets and once the time taken for a request is calculated it is added to the count of all the buckets whose bucket boundaries are higher than the measured value.

Let's say Request 1 for endpoint “/ping” takes 0.25 s. The count values for the buckets will be.

> /ping

Bucket	**Count**
0 - 0.3	1
0 - 0.5	1
0 - 0.7	1
0 - 1	1
0 - 1.2	1
0 - +Inf	1

Note: +Inf bucket is added by default.

(Since the histogram is a cumulative frequency 1 is added to all the buckets that are greater than the value)

Request 2 for endpoint “/ping” takes 0.4s The count values for the buckets will be this.

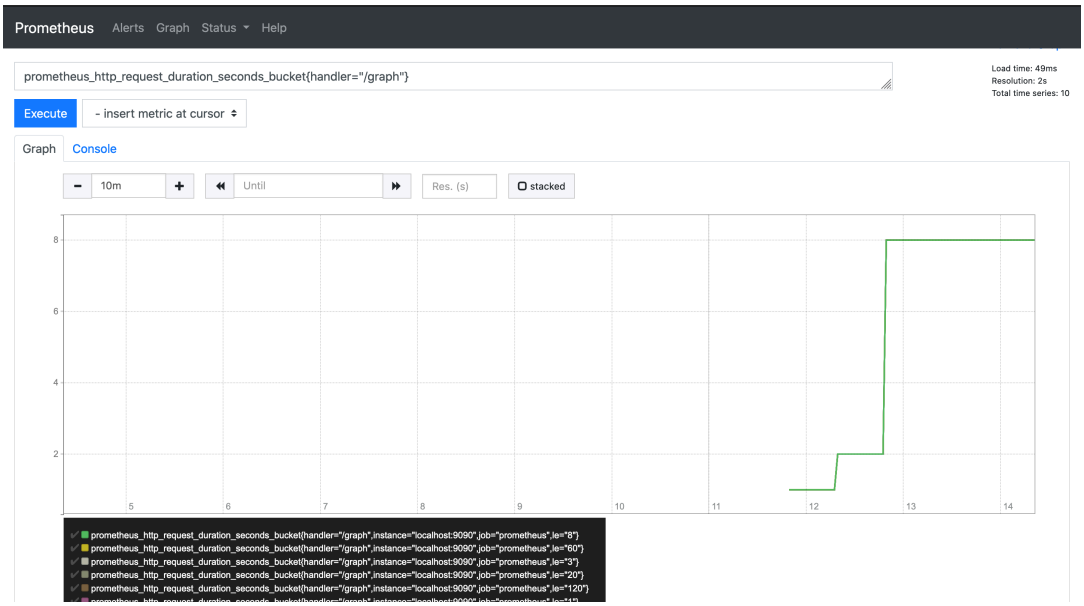
> /ping

Bucket	Count
0 - 0.3	1
0 - 0.5	2
0 - 0.7	2
0 - 1	2
0 - 1.2	2
0 - +Inf	2

Since 0.4 is below 0.5, all buckets up to that boundary increase their counts.

Let's explore a histogram metric from the Prometheus UI and apply a few functions.

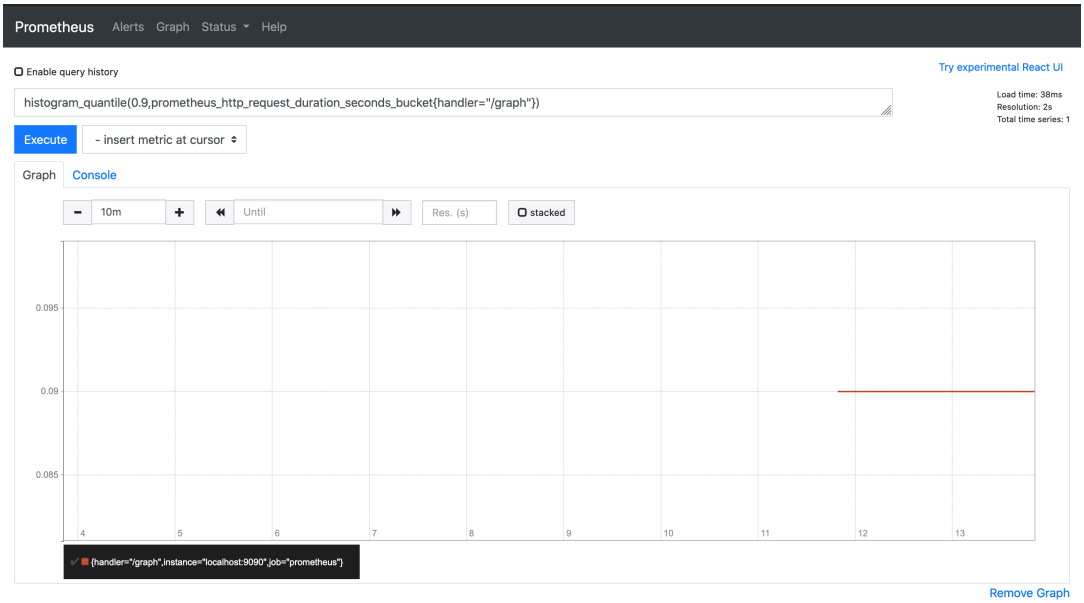
`prometheus_http_request_duration_seconds_bucket{handler="/graph"}`



(/assets/tutorial/histogram_example.png)

`histogram_quantile()` function can be used to calculate quantiles from a histogram

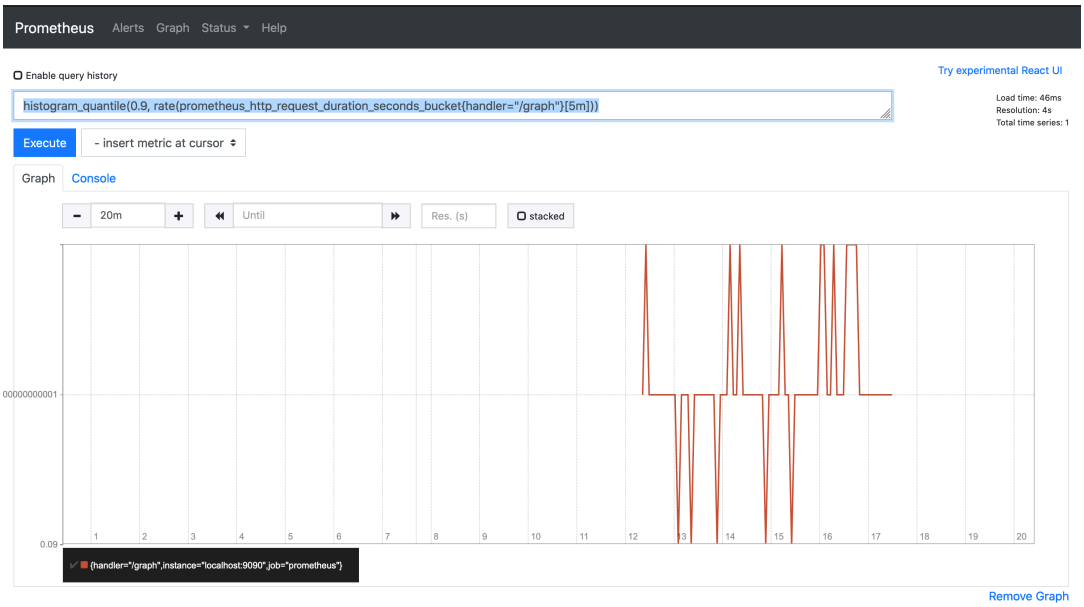
`histogram_quantile(0.9,prometheus_http_request_duration_seconds_bucket{handler="/graph"})`



(/assets/tutorial/histogram_quantile_example.png)

The graph shows that the 90th percentile is 0.09, To find the histogram_quantile over the last 5m you can use the rate() and time frame

```
histogram_quantile(0.9,
rate(prometheus_http_request_duration_seconds_bucket{handler="/graph"}[5m]))
```



(/assets/tutorial/histogram_rate_example.png)

Summary

Summaries also measure events and are an alternative to histograms. They are cheaper but lose more data. They are calculated on the application level hence aggregation of metrics from multiple instances of the same process is not possible. They are used when the buckets of a metric are not known beforehand, but it is highly recommended to use histograms over summaries whenever possible.

In this tutorial, we covered the types of metrics in detail and a few PromQL operations like rate, histogram_quantile, etc.

In this tutorial we will create a simple Go HTTP server and instrumentation it by adding a counter metric to keep count of the total number of requests processed by the server.

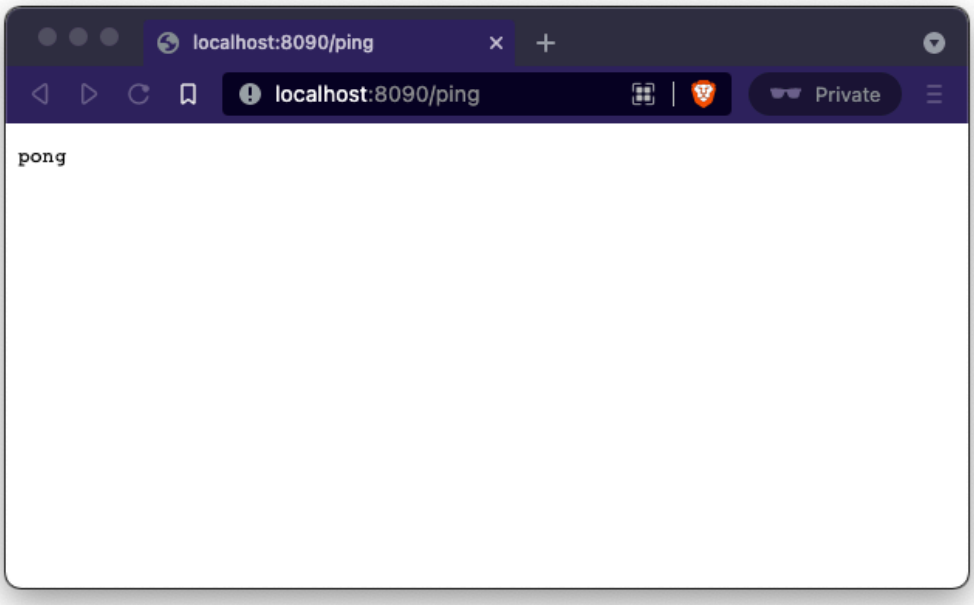
Here we have a simple HTTP server with `/ping` endpoint which returns `pong` as response.

```
package main

import (
   "fmt"
   "net/http"
)

func ping(w http.ResponseWriter, req *http.Request){
   fmt.Fprintf(w,"pong")
}

func main() {
   http.HandleFunc("/ping",ping)

   http.ListenAndServe(":8090", nil)
}
```

Compile and run the server

```
go build server.go
./server
```

Now open `http://localhost:8090/ping` in your browser and you must see `pong` .



(/assets/tutorial/server.png)

Now lets add a metric to the server which will instrument the number of requests made to the ping endpoint,the counter metric type is suitable for this as we know the request count doesn't go down and only increases.

Create a Prometheus counter

```
var pingCounter = prometheus.NewCounter(
   prometheus.CounterOpts{
       Name: "ping_request_count",
       Help: "No of request handled by Ping handler",
   },
)
```

Next lets update the ping Handler to increase the count of the counter using `pingCounter.Inc()` .

```
func ping(w http.ResponseWriter, req *http.Request) {
   pingCounter.Inc()
   fmt.Fprintf(w, "pong")
}
```

Then register the counter to the Default Register and expose the metrics.

```
func main() {
   prometheus.MustRegister(pingCounter)
   http.HandleFunc("/ping", ping)
   http.Handle("/metrics", promhttp.Handler())
   http.ListenAndServe(":8090", nil)
}
```

The `prometheus.MustRegister` function registers the pingCounter to the default Register. To expose the metrics the Go Prometheus client library provides the promhttp package. `promhttp.Handler()` provides a `http.Handler` which exposes the metrics registered in the Default Register.

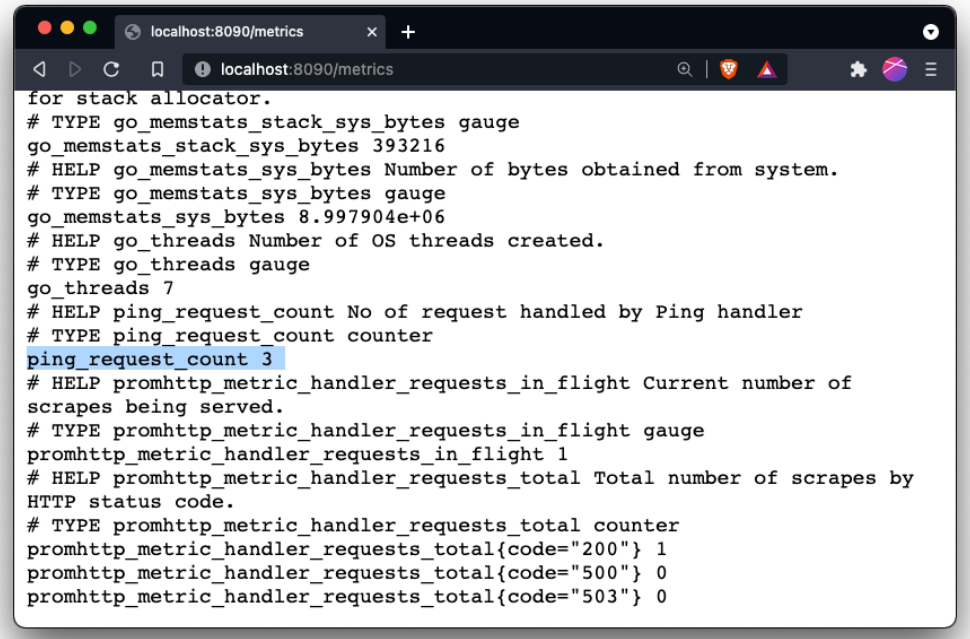
The sample code depends on the

```
package main

import (
   "fmt"
   "net/http"

   "github.com/prometheus/client_golang/prometheus"
   "github.com/prometheus/client_golang/prometheus/promhttp"
)

var pingCounter = prometheus.NewCounter(
   prometheus.CounterOpts{
       Name: "ping_request_count",
       Help: "No of request handled by Ping handler",
   },
)

func ping(w http.ResponseWriter, req *http.Request) {
   pingCounter.Inc()
   fmt.Fprintf(w, "pong")
}

func main() {
   prometheus.MustRegister(pingCounter)

   http.HandleFunc("/ping", ping)
   http.Handle("/metrics", promhttp.Handler())
   http.ListenAndServe(":8090", nil)
}
```

Run the example

```
go mod init prom_example
go mod tidy
go run server.go
```

Now hit the localhost:8090/ping endpoint a couple of times and sending a request to localhost:8090 will provide the metrics.



(/assets/tutorial/ping_metric.png)

Here the `ping_request_count` shows that `/ping` endpoint was called 3 times.

The Default Register comes with a collector for go runtime metrics and that is why we see other metrics like `go_threads`, `go_goroutines` etc.

We have built our first metric exporter. Let's update our Prometheus config to scrape the metrics from our server.

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: ["localhost:9090"]
  - job_name: simple_server
    static_configs:
      - targets: ["localhost:8090"]
```

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

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

VISUALIZING METRICS.

In this tutorial we will create a simple dashboard using Grafana (https://github.com/grafana/grafana) to visualize the `ping_request_count` metric that we instrumented in the previous tutorial (../instrumenting_http_server_in_go).

If you are wondering why one should use a tool like Grafana when one can query and see the graphs using Prometheus, the answer is that the graph that we see when we run queries on Prometheus is to run ad-hoc queries. Grafana and Console Templates (https://prometheus.io/docs/visualization/consoles/) are two recommended ways of creating graphs.

Installing and Setting up Grafana.

Install and Run Grafana by following the steps from here (https://grafana.com/docs/grafana/latest/installation/requirements/#supported-operating-systems) for your operating system.

Once Grafana is installed and run, navigate to http://localhost:3000 (http://localhost:3000) in your browser. Use the default credentials, username as `admin` and password as `admin` to log in and setup new credentials.

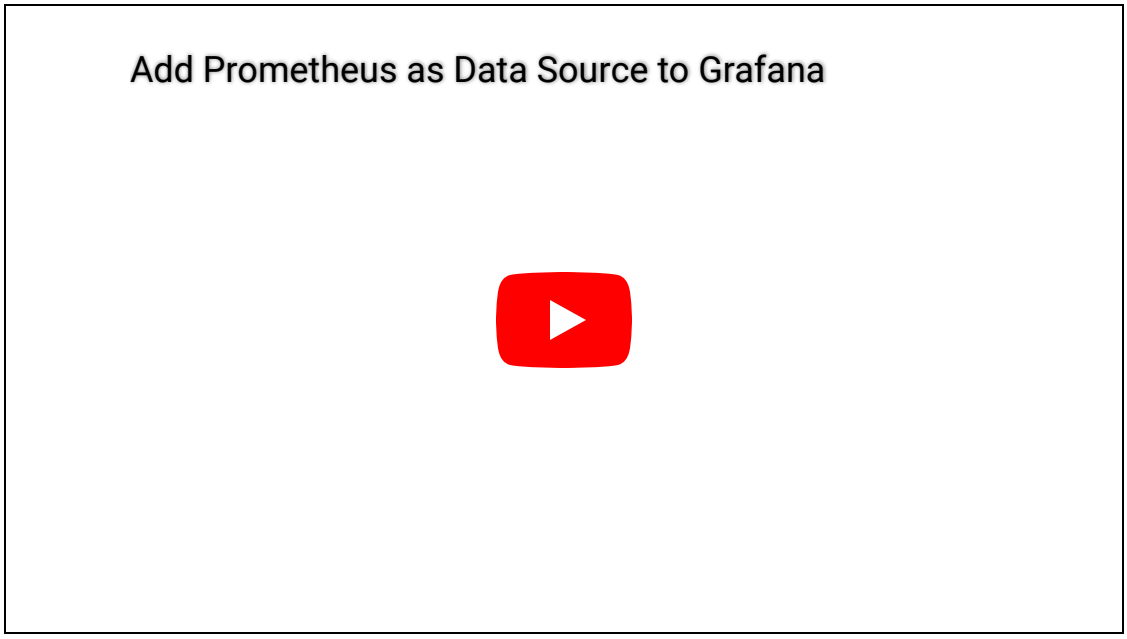
Adding Prometheus as a Data Source in Grafana.

Let's add a datasource to Grafana by clicking on the gear icon in the side bar and select `Data Sources`

> ⚙ > Data Sources

In the Data Sources screen you can see that Grafana supports multiple data sources like Graphite, PostgreSQL etc. Select Prometheus to set it up.

Enter the URL as http://localhost:9090 (http://localhost:9090) under the HTTP section and click on `Save and Test`.



Creating our first dashboard.

Now we have successfully added Prometheus as a data source, Next we will create our first dashboard for the `ping_request_count` metric that we instrumented in the previous tutorial.

1. Click on the `+` icon in the side bar and select `Dashboard`.
2. In the next screen, Click on the `Add new panel` button.
3. In the `Query` tab type the PromQL query, in this case just type `ping_request_count`.
4. Access the `ping` endpoint few times and refresh the Graph to verify if it is working as expected.
5. In the right hand section under `Panel Options` set the `Title` as `Ping Request Count`.
6. Click on the Save Icon in the right corner to Save the dashboard.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

ALERTING BASED ON METRICS

In this tutorial we will create alerts on the `ping_request_count` metric that we instrumented earlier in the Instrumenting HTTP server written in Go (./instrumenting_http_server_in_go/) tutorial.

For the sake of this tutorial we will alert when the `ping_request_count` metric is greater than 5, Checkout real world best practices (../../practices/alerting) to learn more about alerting principles.

Download the latest release of Alertmanager for your operating system from here (https://github.com/prometheus/alertmanager/releases)

Alertmanager supports various receivers like `email`, `webhook`, `pagerduty`, `slack` etc through which it can notify when an alert is firing. You can find the list of receivers and how to configure them here (../../alerting/latest/configuration). We will use `webhook` as a receiver for this tutorial, head over to webhook.site (https://webhook.site) and copy the webhook URL which we will use later to configure the Alertmanager.

First let's setup Alertmanager with webhook receiver.

alertmanager.yml

```
global:
  resolve_timeout: 5m
route:
  receiver: webhook_receiver
receivers:
    - name: webhook_receiver
      webhook_configs:
        - url: '<INSERT-YOUR-WEBHOOK>'
          send_resolved: false
```

Replace `<INSERT-YOUR-WEBHOOK>` with the webhook that we copied earlier in the alertmanager.yml file and run the Alertmanager using the following command.

```
alertmanager --config.file=alertmanager.yml
```

Once the Alertmanager is up and running navigate to http://localhost:9093 (http://localhost:9093) and you should be able to access it.

Now that we have configured the Alertmanager with webhook receiver let's add the rules to the Prometheus config.

prometheus.yml

```
global:
 scrape_interval: 15s
 evaluation_interval: 10s
rule_files:
  - rules.yml
alerting:
  alertmanagers:
  - static_configs:
    - targets:
       - localhost:9093
scrape_configs:
 - job_name: prometheus
   static_configs:
       - targets: ["localhost:9090"]
 - job_name: simple_server
   static_configs:
       - targets: ["localhost:8090"]
```

If you notice the `evaluation_interval` , `rule_files` and `alerting` sections are added to the Prometheus config, the `evaluation_interval` defines the intervals at which the rules are evaluated, `rule_files` accepts an array of yaml files that defines the rules and the `alerting` section defines the Alertmanager configuration. As mentioned in the beginning of this tutorial we will create a basic rule where we want to raise an alert when the `ping_request_count` value is greater than 5.

rules.yml

```
groups:
 - name: Count greater than 5
   rules:
   - alert: CountGreaterThan5
     expr: ping_request_count > 5
     for: 10s
```

Now let's run Prometheus using the following command.

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

Open http://localhost:9090/rules (http://localhost:9090/rules) in your browser to see the rules. Next run the instrumented ping server and visit the http://localhost:8090/ping (http://localhost:8090/ping) endpoint and refresh the page atleast 6 times. You can check the ping count by navigating to http://localhost:8090/metrics (http://localhost:8090/metrics) endpoint. To see the status of the alert visit http://localhost:9090/alerts (http://localhost:9090/alerts). Once the condition `ping_request_count > 5` is true for more than 10s the `state` will become `FIRING`. Now if you navigate back to your `webhook.site` URL you will see the alert message.

Similarly Alertmanager can be configured with other receivers to notify when an alert is firing.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

PROMETHEUS REMOTE-WRITE SPECIFICATION

- Version: 2.0-rc.3
- Status: **Experimental**
- Date: May 2024

The Remote-Write specification, in general, is intended to document the standard for how Prometheus and Prometheus Remote-Write compatible senders send data to Prometheus or Prometheus Remote-Write compatible receivers.

This document is intended to define a second version of the Prometheus Remote-Write (./remote_write_spec.md) API with minor changes to protocol and semantics. This second version adds a new Protobuf Message with new features enabling more use cases and wider adoption on top of performance and cost savings. The second version also deprecates the previous Protobuf Message from a 1.0 Remote-Write specification (./remote_write_spec.md#protocol) and adds mandatory `X-Prometheus-Remote-Write-*-Written` HTTP response headersfor reliability purposes. Finally, this spec outlines how to implement backwards-compatible senders and receivers (even under a single endpoint) using existing basic content negotiation request headers. More advanced, automatic content negotiation mechanisms might come in a future minor version if needed. For the rationales behind the 2.0 specification, see the formal proposal (https://github.com/prometheus/proposals/pull/35).

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 (https://datatracker.ietf.org/doc/html/rfc2119).

NOTE: This is a release candidate for Remote-Write 2.0 specification. This means that this specification is currently in an experimental state--no major changes are expected, but we reserve the right to break the compatibility if it's necessary, based on the early adopters' feedback. The potential feedback, questions and suggestions should be added as comments to the PR with the open proposal (https://github.com/prometheus/proposals/pull/35).

Introduction

Background

The Remote-Write protocol is designed to make it possible to reliably propagate samples in real-time from a sender to a receiver, without loss.

The Remote-Write protocol is designed to be stateless; there is strictly no inter-message communication. As such the protocol is not considered "streaming". To achieve a streaming effect multiple messages should be sent over the same connection using e.g. HTTP/1.1 or HTTP/2. "Fancy" technologies such as gRPC were considered, but at the time were not widely adopted, and it was challenging to expose gRPC services to the internet behind load balancers such as an AWS EC2 ELB.

The Remote-Write protocol contains opportunities for batching, e.g. sending multiple samples for different series in a single request. It is not expected that multiple samples for the same series will be commonly sent in the same request, although there is support for this in the Protobuf Message.

A test suite can be found at https://github.com/prometheus/compliance/tree/main/remote_write_sender (https://github.com/prometheus/compliance/tree/main/remote_write_sender). The compliance tests for remote write 2.0 compatibility are still in progress (https://github.com/prometheus/compliance/issues/101).

Glossary

In this document, the following definitions are followed:

- `Remote-Write` is the name of this Prometheus protocol.
- a `Protocol` is a communication specification that enables the client and server to transfer metrics.
- a `Protobuf Message` (or Proto Message) refers to the content type (https://www.rfc-editor.org/rfc/rfc9110.html#name-content-type) definition of the data structure for this Protocol. Since the specification uses Google Protocol Buffers ("protobuf") (https://protobuf.dev/) exclusively, the schema is defined in a "proto" file (https://protobuf.dev/programming-guides/proto3/) and represented by a single Protobuf "message" (https://protobuf.dev/programming-guides/proto3/#simple).
- a `Wire Format` is the format of the data as it travels on the wire (i.e. in a network). In the case of Remote-Write, this is always the compressed binary protobuf format.
- a `Sender` is something that sends Remote-Write data.
- a `Receiver` is something that receives (writes) Remote-Write data. The meaning of `Written` is up to the Receiver e.g. usually it means storing received data in a database, but also just validating, splitting or enhancing it.
- `Written` refers to data the `Receiver` has received and is accepting. Whether or not it has ingested this data to persistent storage, written it to a WAL, etc. is up to the `Receiver`. The only distinction is that the `Receiver` has accepted this data rather than explicitly rejecting it with an error response.
- a `Sample` is a pair of (timestamp, value).
- a `Histogram` is a pair of (timestamp, histogram value (https://github.com/prometheus/docs/blob/b9657b5f5b264b81add39f6db2f1df36faf03efe/content/docs/concepts/nati
- a `Label` is a pair of (key, value).
- a `Series` is a list of samples, identified by a unique set of labels.

Definitions

Protocol

The Remote-Write Protocol MUST consist of RPCs with the request body serialized using a Google Protocol Buffers and then compressed.

The protobuf serialization MUST use either of the following Protobuf Messages:

- The `prometheus.WriteRequest` introduced in the Remote-Write 1.0 specification (./remote_write_spec.md#protocol). As of 2.0, this message is deprecated. It SHOULD be used only for compatibility reasons. Senders and Receivers MAY NOT support the `prometheus.WriteRequest`.
- The `io.prometheus.write.v2.Request` introduced in this specification and defined below. Senders and Receivers SHOULD use this message when possible. Senders and Receivers MUST support the `io.prometheus.write.v2.Request`.

Protobuf Message MUST use binary Wire Format. Then, MUST be compressed with Google's Snappy (https://github.com/google/snappy). Snappy's block format (https://github.com/google/snappy/blob/2c94e11145f0b7b184b831577c93e5a41c4c0346/format_description.txt) MUST be used -- the framed format (https://github.com/google/snappy/blob/2c94e11145f0b7b184b831577c93e5a41c4c0346/framing_format.txt) MUST NOT be used.

Senders MUST send a serialized and compressed Protobuf Message in the body of an HTTP POST request and send it to the Receiver via HTTP at the provided URL path. Receivers MAY specify any HTTP URL path to receive metrics.

Senders MUST send the following reserved headers with the HTTP request:

- `Content-Encoding`
- `Content-Type`
- `X-Prometheus-Remote-Write-Version`
- `User-Agent`

Senders MAY allow users to add custom HTTP headers; they MUST NOT allow users to configure them in such a way as to send reserved headers.

Content-Encoding

```
Content-Encoding: <compression>
```

Content encoding request header MUST follow the RFC 9110 (https://www.rfc-editor.org/rfc/rfc9110.html#name-content-encoding). Senders MUST use the `snappy` value. Receivers MUST support `snappy` compression. New, optional compression algorithms might come in 2.x or beyond.

Content-Type

```
Content-Type: application/x-protobuf
Content-Type: application/x-protobuf;proto=<fully qualified name>
```

Content type request header MUST follow the RFC 9110 (https://www.rfc-editor.org/rfc/rfc9110.html#name-content-type). Senders MUST use `application/x-protobuf` as the only media type. Senders MAY add `;proto=` parameter to the header's value to indicate the fully qualified name of the Protobuf Message that was used, from the two mentioned above. As a result, Senders MUST send any of the three supported header values:

For the deprecated message introduced in PRW 1.0, identified by `prometheus.WriteRequest`:

- `Content-Type: application/x-protobuf`
- `Content-Type: application/x-protobuf;proto=prometheus.WriteRequest`

For the message introduced in PRW 2.0, identified by `io.prometheus.write.v2.Request`:

- `Content-Type: application/x-protobuf;proto=io.prometheus.write.v2.Request`

When talking to 1.x Receivers, Senders SHOULD use `Content-Type: application/x-protobuf` for backward compatibility. Otherwise, Senders SHOULD use `Content-Type: application/x-protobuf;proto=io.prometheus.write.v2.Request`. More Protobuf Messages might come in 2.x or beyond.

Receivers MUST use the content type header to identify the Protobuf Message schema to use. Accidental wrong schema choices may result in non-deterministic behaviour (e.g. corruptions).

> **NOTE:** Thanks to reserved fields in `io.prometheus.write.v2.Request`, Receiver accidental use of wrong schema with `prometheus.WriteRequest` will result in empty message. This is generally for convenience to avoid surprising errors, but don't rely on it -- future Protobuf Messages might not have this feature.

X-Prometheus-Remote-Write-Version

```
X-Prometheus-Remote-Write-Version: <Remote-Write spec major and minor version>
```

When talking to 1.x Receivers, Senders MUST use `X-Prometheus-Remote-Write-Version: 0.1.0` for backward compatibility. Otherwise, Senders SHOULD use the newest Remote-Write version it is compatible with e.g. `X-Prometheus-Remote-Write-Version: 2.0.0`.

User-Agent

```
User-Agent: <name & version of the Sender>
```

Senders MUST include a user agent header that SHOULD follow the RFC 9110 User-Agent header format (https://www.rfc-editor.org/rfc/rfc9110.html#name-user-agent).

Response

Receivers that written all data successfully MUST return a success 2xx HTTP status code (https://www.rfc-editor.org/rfc/rfc9110.html#name-successful-2xx). In such a successful case, the response body from the Receiver SHOULD be empty and the status code SHOULD be 204 HTTP No Content (https://www.rfc-editor.org/rfc/rfc9110.html#name-204-no-content); Senders MUST ignore the response body. The response body is RESERVED for future use.

Receivers MUST NOT return a 2xx HTTP status code if any of the pieces of sent data known to the Receiver (e.g. Samples, Histograms, Exemplars) were NOT written successfully (both partial write or full write rejection). In such a case, the Receiver MUST provide a human-readable error message in the response body. The Receiver's error SHOULD contain information about the amount of the samples being rejected and for what reasons. Senders MUST NOT try and interpret the error message and SHOULD log it as is.

The following subsections specify Sender and Receiver semantics around headers and different write error cases.

Required `Written` Response Headers

Upon a successful content negotiation, Receivers process (write) the received batch of data. Once completed (with success or failure) for each important piece of data (currently Samples, Histograms and Exemplars) Receivers MUST send a dedicated HTTP `X-Prometheus-Remote-Write-*-Written` response header with the precise number of successfully written elements.

Each header value MUST be a single 64-bit integer. The header names MUST be as follows:

```
X-Prometheus-Remote-Write-Samples-Written <count of all successfully written Samples>
X-Prometheus-Remote-Write-Histograms-Written <count of all successfully written Histogram samples>
X-Prometheus-Remote-Write-Exemplars-Written <count of all successfully written Exemplars>
```

Upon receiving a 2xx or a 4xx status code, Senders CAN assume that any missing `X-Prometheus-Remote-Write-*-Written` response header means no element from this category (e.g. Sample) was written by the Receiver (count of `0`). Senders MUST NOT assume the same when using the deprecated `prometheus.WriteRequest` Protobuf Message due to the risk of hitting 1.0 Receiver without this feature.

Senders MAY use those headers to confirm which parts of data were successfully written by the Receiver. Common use cases:

- Better handling of the Partial Write failure situations: Senders MAY use those headers for more accurate client instrumentation and error handling.
- Detecting broken 1.0 Receiver implementations: Senders SHOULD assume 415 HTTP Unsupported Media Type (https://www.rfc-editor.org/rfc/rfc9110.html#name-415-unsupported-media-type) status code when sending the data using `io.prometheus.write.v2.Request` request and receiving 2xx HTTP status code, but none of the `X-Prometheus-Remote-Write-*-Written` response headers from the Receiver. This is a common issue for the 1.0 Receivers that do not check the `Content-Type` request header; accidental decoding of the `io.prometheus.write.v2.Request` payload with `prometheus.WriteRequest` schema results in empty result and no decoding errors.
- Detecting other broken implementations or issues: Senders MAY use those headers to detect broken Sender and Receiver implementations or other problems.

Senders MUST NOT assume what Remote Write specification version the Receiver implements from the remote write response headers.

More (optional) headers might come in the future, e.g. when more entities or fields are added and worth confirming.

Partial Write

Senders SHOULD use Remote-Write to send samples for multiple series in a single request. As a result, Receivers MAY write valid samples within a write request that also contains some invalid or otherwise unwritten samples, which represents a partial write case. In such a case, the Receiver MUST return non-2xx status code following the Invalid Samples and Retry on Partial Writes sections.

Unsupported Request Content

Receivers MUST return 415 HTTP Unsupported Media Type (https://www.rfc-editor.org/rfc/rfc9110.html#name-415-unsupported-media-type) status code if they don't support a given content type or encoding provided by Senders.

Senders SHOULD expect 400 HTTP Bad Request (https://www.rfc-editor.org/rfc/rfc9110.html#name-400-bad-request) for the above reasons from 1.x Receivers, for backwards compatibility.

Invalid Samples

Receivers MAY NOT support certain metric types or samples (e.g. a Receiver might reject sample without metadata type specified or without created timestamp, while another Receiver might accept such sample.). It’s up to the Receiver what sample is invalid. Receivers MUST return a 400 HTTP Bad Request (https://www.rfc-editor.org/rfc/rfc9110.html#name-400-bad-request) status code for write requests that contain any invalid samples unless the partial retriable write occurs.

Senders MUST NOT retry on a 4xx HTTP status codes (other than 429 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/429)), which MUST be used by Receivers to indicate that the write operation will never be able to succeed and should not be retried. Senders MAY retry on the 415 HTTP status code with a different content type or encoding to see if the Receiver supports it.

Retries & Backoff

Receivers MAY return a 429 HTTP Too Many Requests (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/429) status code to indicate the overloaded server situation. Receivers MAY return the Retry-After (https://www.rfc-editor.org/rfc/rfc9110.html#name-retry-after) header to indicate the time for the next write attempt. Receivers MAY return a 5xx HTTP status code to represent internal server errors.

Senders MAY retry on a 429 HTTP status code. Senders MUST retry write requests on 5xx HTTP. Senders MUST use a backoff algorithm to prevent overwhelming the server. Senders MAY handle the Retry-After response header (https://www.rfc-editor.org/rfc/rfc9110.html#name-retry-after) to estimate the next retry time.

The difference between 429 vs 5xx handling is due to the potential situation of a Sender "falling behind" when the Receiver cannot keep up with the request volume, or the Receiver choosing to rate limit the Sender to protect its availability. As a result, Senders has the option to NOT retry on 429, which allows progress to be made when there are Sender side errors (e.g. too much traffic), while the data is not lost when there are Receiver side errors (5xx).

Retries on Partial Writes

Receivers MAY return a 5xx HTTP or 429 HTTP status code on partial write or partial invalid sample cases when it expects Senders to retry the whole request. In that case, the Receiver MUST support idempotency as Senders MAY retry with the same request.

Backward and Forward Compatibility

The protocol follows semantic versioning 2.0 (https://semver.org/): any 2.x compatible Receiver MUST be able to read any 2.x compatible Senders and vice versa. Breaking or backwards incompatible changes will result in a 3.x version of the spec.

The Protobuf Messages (in Wire Format) themselves are forward / backward compatible, in some respects:

- Removing fields from the Protobuf Message requires a major version bump.
- Adding (optional) fields can be done in a minor version bump.

In other words, this means that future minor versions of 2.x MAY add new optional fields to `io.prometheus.write.v2.Request`, new compressions, Protobuf Messages and negotiation mechanisms, as long as they are backwards compatible (e.g. optional to both Receiver and Sender).

2.x vs 1.x Compatibility

The 2.x protocol is breaking compatibility with 1.x by introducing a new, mandatory `io.prometheus.write.v2.Request` Protobuf Message and deprecating the `prometheus.WriteRequest`.

2.x Senders MAY support 1.x Receivers by allowing users to configure what content type Senders should use. 2.x Senders also MAY automatically fall back to different content types, if the Receiver returns 415 HTTP status code.

Protobuf Message

`io.prometheus.write.v2.Request`

The `io.prometheus.write.v2.Request` references the new Protobuf Message that's meant to replace and deprecate the Remote-Write 1.0's `prometheus.WriteRequest` message.

The full schema and source of the truth is in Prometheus repository in `prompb/io/prometheus/write/v2/types.proto` (https://github.com/prometheus/prometheus/blob/remote-write-2.0/prompb/io/prometheus/write/v2/types.proto#L32). The `gogo` dependency and options CAN be ignored (will be removed eventually (https://github.com/prometheus/prometheus/issues/11908)). They are not part of the specification as they don't impact the serialized format.

The simplified version of the new `io.prometheus.write.v2.Request` is presented below.

```
// Request represents a request to write the given timeseries to a remote destination.
message Request {
  // Since Request supersedes 1.0 spec's prometheus.WriteRequest, we reserve the top-down message
  // for the deterministic interop between those two.
  // Generally it's not needed, because Receivers must use the Content-Type header, but we want to
  // be sympathetic to adopters with mistaken implementations and have deterministic error (empty
  // message if you use the wrong proto schema).
  reserved 1 to 3;

  // symbols contains a de-duplicated array of string elements used for various
  // items in a Request message, like labels and metadata items. For the sender's convenience
  // around empty values for optional fields like unit_ref, symbols array MUST start with
  // empty string.
  //
  // To decode each of the symbolized strings, referenced, by "ref(s)" suffix, you
  // need to lookup the actual string by index from symbols array. The order of
  // strings is up to the sender. The receiver should not assume any particular encoding.
  repeated string symbols = 4;
  // timeseries represents an array of distinct series with 0 or more samples.
  repeated TimeSeries timeseries = 5;
}

// TimeSeries represents a single series.
message TimeSeries {
  // labels_refs is a list of label name-value pair references, encoded
  // as indices to the Request.symbols array. This list's length is always
  // a multiple of two, and the underlying labels should be sorted lexicographically.
  //
  // Note that there might be multiple TimeSeries objects in the same
  // Requests with the same labels e.g. for different exemplars, metadata
  // or created timestamp.
  repeated uint32 labels_refs = 1;

  // Timeseries messages can either specify samples or (native) histogram samples
  // (histogram field), but not both. For a typical sender (real-time metric
  // streaming), in healthy cases, there will be only one sample or histogram.
  //
  // Samples and histograms are sorted by timestamp (older first).
  repeated Sample samples = 2;
  repeated Histogram histograms = 3;

  // exemplars represents an optional set of exemplars attached to this series' samples.
  repeated Exemplar exemplars = 4;

  // metadata represents the metadata associated with the given series' samples.
  Metadata metadata = 5;

  // created_timestamp represents an optional created timestamp associated with
  // this series' samples in ms format, typically for counter or histogram type
  // metrics. Created timestamp represents the time when the counter started
  // counting (sometimes referred to as start timestamp), which can increase
  // the accuracy of query results.
  //
  // Note that some receivers might require this and in return fail to
  // write such samples within the Request.
  //
  // For Go, see github.com/prometheus/prometheus/model/timestamp/timestamp.go
  // for conversion from/to time.Time to Prometheus timestamp.
  //
  // Note that the "optional" keyword is omitted due to
  // https://cloud.google.com/apis/design/design_patterns.md#optional_primitive_fields
  // Zero value means value not set. If you need to use exactly zero value for
  // the timestamp, use 1 millisecond before or after.
  int64 created_timestamp = 6;
}
```

```
// Exemplar represents additional information attached to some series' samples.
message Exemplar {
  // labels_refs is an optional list of label name-value pair references, encoded
  // as indices to the Request.symbols array. This list's len is always
  // a multiple of 2, and the underlying labels should be sorted lexicographically.
  // If the exemplar references a trace it should use the `trace_id` label name, as a best practice.
  repeated uint32 labels_refs = 1;
  // value represents an exact example value. This can be useful when the exemplar
  // is attached to a histogram, which only gives an estimated value through buckets.
  double value = 2;
  // timestamp represents the timestamp of the exemplar in ms.
  // For Go, see github.com/prometheus/prometheus/model/timestamp/timestamp.go
  // for conversion from/to time.Time to Prometheus timestamp.
  int64 timestamp = 3;
}

// Sample represents series sample.
message Sample {
  // value of the sample.
  double value = 1;
  // timestamp represents timestamp of the sample in ms.
  int64 timestamp = 2;
}

// Metadata represents the metadata associated with the given series' samples.
message Metadata {
  enum MetricType {
    METRIC_TYPE_UNSPECIFIED    = 0;
    METRIC_TYPE_COUNTER        = 1;
    METRIC_TYPE_GAUGE          = 2;
    METRIC_TYPE_HISTOGRAM      = 3;
    METRIC_TYPE_GAUGEHISTOGRAM = 4;
    METRIC_TYPE_SUMMARY        = 5;
    METRIC_TYPE_INFO           = 6;
    METRIC_TYPE_STATESET       = 7;
  }
  MetricType type = 1;
  // help_ref is a reference to the Request.symbols array representing help
  // text for the metric. Help is optional, reference should point to an empty string in
  // such a case.
  uint32 help_ref = 3;
  // unit_ref is a reference to the Request.symbols array representing a unit
  // for the metric. Unit is optional, reference should point to an empty string in
  // such a case.
  uint32 unit_ref = 4;
}

// A native histogram, also known as a sparse histogram.
// See https://github.com/prometheus/prometheus/blob/remote-write-2.0/prompb/io/prometheus/write/v2/types.proto#L142
// for a full message that follows the native histogram spec for both sparse
// and exponential, as well as, custom bucketing.
message Histogram { ... }
```

All timestamps MUST be int64 counted as milliseconds since the Unix epoch. Sample's values MUST be float64.

For every `TimeSeries` message:

- `labels_refs` MUST be provided.
- At least one element in `samples` or in `histograms` MUST be provided. A `TimeSeries` MUST NOT include both `samples` and `histograms`. For series which (rarely) would mix float and histogram samples, a separate `TimeSeries` message MUST be used.

- `metadata` sub-fields SHOULD be provided. Receivers MAY reject series with unspecified `Metadata.type`.
- Exemplars SHOULD be provided if they exist for a series.
- `created_timestamp` SHOULD be provided for metrics that follow counter semantics (e.g. counters and histograms). Receivers MAY reject those series without `created_timestamp` being set.

The following subsections define some schema elements in detail.

Symbols

The `io.prometheus.write.v2.Request` Protobuf Message is designed to intern all strings (https://en.wikipedia.org/wiki/String_interning) for the proven additional compression and memory efficiency gains on top of the standard compressions.

The `symbols` table MUST be provided and it MUST contain deduplicated strings used in series, exemplar labels, and metadata strings. The first element of the `symbols` table MUST be an empty string, which is used to represent empty or unspecified values such as when `Metadata.unit_ref` or `Metadata.help_ref` are not provided. References MUST point to the existing index in the `symbols` string array.

Series Labels

The complete set of labels MUST be sent with each `Sample` or `Histogram` sample. Additionally, the label set associated with samples:

- SHOULD contain a `__name__` label.
- MUST NOT contain repeated label names.
- MUST have label names sorted in lexicographical order.
- MUST NOT contain any empty label names or values.

Metric names, label names, and label values MUST be any sequence of UTF-8 characters.

Metric names SHOULD adhere to the regex `[a-zA-Z_:]([a-zA-Z0-9_:])*`.

Label names SHOULD adhere to the regex `[a-zA-Z_]([a-zA-Z0-9_])*`.

Names that do not adhere to the above, might be harder to use for PromQL users (see the UTF-8 proposal for more details (https://github.com/prometheus/proposals/blob/main/proposals/2023-08-21-utf8.md)).

Label names beginning with "__" are RESERVED for system usage and SHOULD NOT be used, see Prometheus Data Model (https://prometheus.io/docs/concepts/data_model/).

Receivers also MAY impose limits on the number and length of labels, but this is receiver-specific and is out of the scope of this document.

Samples and Histogram Samples

Senders MUST send `samples` (or `histograms`) for any given `TimeSeries` in timestamp order. Senders MAY send multiple requests for different series in parallel.

Senders SHOULD send stale markers when a time series will no longer be appended to. Senders MUST send stale markers if the discontinuation of time series is possible to detect, for example:

- For series that were pulled (scraped), unless explicit timestamp was used.
- For series that is resulted by a recording rule evaluation.

Generally, not sending stale markers for series that are discontinued can lead to the Receiver non-trivial query time alignment issues (https://prometheus.io/docs/prometheus/latest/querying/basics/#staleness).

Stale markers MUST be signalled by the special NaN value `0x7ff0000000000002`. This value MUST NOT be used otherwise.

Typically, Senders can detect when a time series will no longer be appended using the following techniques:

1. Detecting, using service discovery, that the target exposing the series has gone away.
2. Noticing the target is no longer exposing the time series between successive scrapes.
3. Failing to scrape the target that originally exposed a time series.
4. Tracking configuration and evaluation for recording and alerting rules.
5. Tracking discontinuation of metrics for non-scrape source of metric (e.g. in k6 when the benchmark has finished for series per benchmark, it could emit a stale marker).

Metadata

Metadata SHOULD follow the official Prometheus guidelines for Type (https://prometheus.io/docs/instrumenting/writing_exporters/#types) and Help (https://prometheus.io/docs/instrumenting/writing_exporters/#help-strings).

Metadata MAY follow the official OpenMetrics guidelines for Unit (https://github.com/OpenObservability/OpenMetrics/blob/main/specification/OpenMetrics.md#unit).

Exemplars

Each exemplar, if attached to a `TimeSeries`:

- MUST contain a value.
- MAY contain labels e.g. referencing trace or request ID. If the exemplar references a trace it SHOULD use the `trace_id` label name, as a best practice.
- MUST contain a timestamp. While exemplar timestamps are optional in Prometheus/Open Metrics exposition formats, the assumption is that a timestamp is assigned at scrape time in the same way a timestamp is assigned to the scrape sample. Receivers require exemplar timestamps to reliably handle (e.g. deduplicate) incoming exemplars.

Out of Scope

The same as in 1.0 (./remote_write_spec.md#out-of-scope).

Future Plans

This section contains speculative plans that are not considered part of protocol specification yet but are mentioned here for completeness. Note that 2.0 specification completed 2 of 3 future plans in the 1.0 (./remote_write_spec.md#future-plans).

- **Transactionality** There is still no transactionality defined for 2.0 specification, mostly because it makes a scalable Sender implementation difficult. Prometheus Sender aims at being "transactional" - i.e. to never expose a partially scraped target to a query. We intend to do the same with Remote-Write -- for instance, in the future we would like to "align" Remote-Write with scrapes, perhaps such that all the samples, metadata and exemplars for a single scrape are sent in a single Remote-Write request.

However, Remote-Write 2.0 specification solves an important transactionality problem for the classic histogram buckets (https://docs.google.com/document/d/1mpcSWH1B82q-BtJza-eJ8xMLlKt6EJ9oFGH325vtY1Q/edit#heading=h.ueg7q07wymku). This is done thanks to the native histograms supporting custom bucket-ing possible with the `io.prometheus.write.v2.Request` wire format. Senders might translate all classic histograms to native histograms this way, but it's out of this specification to mandate this. However, for this reason, Receivers MAY ignore certain metric types (e.g. classic histograms).

- **Alternative wire formats**. The OpenTelemetry community has shown the validity of Apache Arrow (and potentially other columnar formats) for over-wire data transfer with their OTLP protocol. We would like to do experiments to confirm the compatibility of a similar format with Prometheus' data model and include benchmarks of any resource usage changes. We would potentially maintain both a protobuf and columnar format long term for compatibility reasons and use our content negotiation to add different Protobuf Messages for this purpose.

- **Global symbols**. Pre-defined string dictionary for interning The protocol could pre-define a static dictionary of ref->symbol that includes strings that are considered common, e.g. “namespace”, “le”, “job”, “seconds”, “bytes”, etc. Senders could refer to these without the need to include them in the request’s symbols table. This dictionary could incrementally grow with minor version releases of this protocol.

Related

FAQ

Why did you not use gRPC? Because the 1.0 protocol does not use gRPC, breaking it would increase friction in the adoption. See 1.0 reason (./remote_write_spec.md#faq).

Why not stream protobuf messages? If you use persistent HTTP/1.1 connections, they are pretty close to streaming. Of course, headers have to be re-sent, but that is less expensive than a new TCP set up.

Why do we send samples in order? The in-order constraint comes from the encoding we use for time series data in Prometheus, the implementation of which is optimized for append-only workloads. However, this requirement is also shared across many other databases and vendors in the ecosystem. In fact, Prometheus with OOO feature enabled (https://youtu.be/qYsycK3nTSQ?t=1321), allows out-of-order writes, but with the performance penalty, thus reserved for rare events. To sum up, Receivers may support out-of-order write, though it is not permitted by the specification. In the future e.g. 2.x spec versions, we could extend content type to negotiate the out-of-order writes, if needed.

How can we parallelise requests with the in-order constraint? Samples must be in-order *for a given series*. However, even if a Receiver does not support out-of-order write, the Remote-Write requests can be sent in parallel as long as they are for different series. Prometheus shards the samples by their labels into separate queues, and then writes happen sequentially in each queue. This guarantees samples for the same series are delivered in order, but samples for different series are sent in parallel - and potentially "out of order" between different series.

What are the differences between Remote-Write 2.0 and OpenTelemetry's OTLP protocol? OpenTelemetry OTLP (https://github.com/open-telemetry/opentelemetry-proto/blob/a05597bff803d3d9405fcdd1e1fb1f42bed4eb7a/docs/specification.md) is a protocol for transporting of telemetry data (such as metrics, logs, traces and profiles) between telemetry sources, intermediate nodes and telemetry backends. The recommended transport involves gRPC with protobuf, but HTTP with protobuf or JSON are also described. It was designed from scratch with the intent to support a variety of different observability signals, data types and extra information. For metrics (https://github.com/open-telemetry/opentelemetry-proto/blob/main/opentelemetry/proto/metrics/v1/metrics.proto) that means additional non-identifying labels, flags, temporal aggregations types, resource or scoped metrics, schema URLs and more. OTLP also requires the semantic convention (https://opentelemetry.io/docs/concepts/semantic-conventions/) to be used.

Remote-Write was designed for simplicity, efficiency and organic growth. The first version was officially released in 2023, when already dozens of battle-tested adopters in the CNCF ecosystem (./remote_write_spec.md#compatible-senders-and-receivers) had been using this protocol for years. Remote-Write 2.0 iterates on the previous protocol by adding a few new elements (metadata, exemplars, created timestamp and native histograms) and string interning. Remote-Write 2.0 is always stateless, focuses only on metrics and is opinionated; as such it is scoped down to elements that the Prometheus community considers enough to have a robust metric solution. The intention is to ensure the Remote-Write is a stable protocol that is cheaper and simpler to adopt and use than the alternatives in the observability ecosystem.

PROMETHEUS REMOTE-WRITE SPECIFICATION

- Version: 1.0
- Status: Published
- Date: April 2023

This document is intended to define and standardise the API, wire format, protocol and semantics of the existing, widely and organically adopted protocol, and not to propose anything new.

The remote write specification is intended to document the standard for how Prometheus and Prometheus remote-write-compatible agents send data to a Prometheus or Prometheus remote-write compatible receiver.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 (https://datatracker.ietf.org/doc/html/rfc2119).

Introduction

Background

The remote write protocol is designed to make it possible to reliably propagate samples in real-time from a sender to a receiver, without loss.

The Remote-Write protocol is designed to be stateless; there is strictly no inter-message communication. As such the protocol is not considered "streaming". To achieve a streaming effect multiple messages should be sent over the same connection using e.g. HTTP/1.1 or HTTP/2. "Fancy" technologies such as gRPC were considered, but at the time were not widely adopted, and it was challenging to expose gRPC services to the internet behind load balancers such as an AWS EC2 ELB.

The remote write protocol contains opportunities for batching, e.g. sending multiple samples for different series in a single request. It is not expected that multiple samples for the same series will be commonly sent in the same request, although there is support for this in the protocol.

The remote write protocol is not intended for use by applications to push metrics to Prometheus remote-write-compatible receivers. It is intended that a Prometheus remote-write-compatible sender scrapes instrumented applications or exporters and sends remote write messages to a server.

A test suite can be found at https://github.com/prometheus/compliance/tree/main/remote_write_sender (https://github.com/prometheus/compliance/tree/main/remote_write_sender).

Glossary

For the purposes of this document the following definitions MUST be followed:

- a "Sender" is something that sends Prometheus Remote Write data.
- a "Receiver" is something that receives Prometheus Remote Write data.
- a "Sample" is a pair of (timestamp, value).
- a "Label" is a pair of (key, value).
- a "Series" is a list of samples, identified by a unique set of labels.

Definitions

Protocol

The Remote Write Protocol MUST consist of RPCs with the following signature:

```
func Send(WriteRequest)

message WriteRequest {
  repeated TimeSeries timeseries = 1;
  // Cortex uses this field to determine the source of the write request.
  // We reserve it to avoid any compatibility issues.
  reserved  2;

  // Prometheus uses this field to send metadata, but this is
  // omitted from v1 of the spec as it is experimental.
  reserved  3;
}

message TimeSeries {
  repeated Label labels   = 1;
  repeated Sample samples = 2;
}

message Label {
  string name  = 1;
  string value = 2;
}

message Sample {
  double value    = 1;
  int64 timestamp = 2;
}
```

Remote write Senders MUST encode the Write Request in the body of a HTTP POST request and send it to the Receivers via HTTP at a provided URL path. The Receiver MAY specify any HTTP URL path to receive metrics.

Timestamps MUST be int64 counted as milliseconds since the Unix epoch. Values MUST be float64.

The following headers MUST be sent with the HTTP request:

- `Content-Encoding: snappy`
- `Content-Type: application/x-protobuf`
- `User-Agent: <name & version of the sender>`
- `X-Prometheus-Remote-Write-Version: 0.1.0`

Clients MAY allow users to send custom HTTP headers; they MUST NOT allow users to configure them in such a way as to send reserved headers. For more info see https://github.com/prometheus/prometheus/pull/8416 (https://github.com/prometheus/prometheus/pull/8416).

The remote write request in the body of the HTTP POST MUST be compressed with Google's Snappy (https://github.com/google/snappy). The block format MUST be used - the framed format MUST NOT be used.

The remote write request MUST be encoded using Google Protobuf 3, and MUST use the schema defined above. Note the Prometheus implementation (https://github.com/prometheus/prometheus/blob/v2.24.0/prompb/remote.proto) uses gogoproto optimisations (https://github.com/gogo/protobuf) - for receivers written in languages other than Golang the gogoproto types MAY be substituted for line-level equivalents.

The response body from the remote write receiver SHOULD be empty; clients MUST ignore the response body. The response body is RESERVED for future use.

Backward and forward compatibility

The protocol follows semantic versioning 2.0 (https://semver.org/): any 1.x compatible receivers MUST be able to read any 1.x compatible sender and so on. Breaking/backwards incompatible changes will result in a 2.x version of the spec.

The proto format itself is forward / backward compatible, in some respects:

- Removing fields from the proto will mean a major version bump.
- Adding (optional) fields will be a minor version bump.

Negotiation:

- Senders MUST send the version number in a headers.
- Receivers MAY return the highest version number they support in a response header ("X-Prometheus-Remote-Write-Version").
- Senders who wish to send in a format >1.x MUST start by sending an empty 1.x, and see if the response says the receiver supports something else. The Sender MAY use any supported version . If there is no version header in the response, senders MUST assume 1.x compatibility only.

Labels

The complete set of labels MUST be sent with each sample. Whatsmore, the label set associated with samples:

- SHOULD contain a `__name__` label.
- MUST NOT contain repeated label names.
- MUST have label names sorted in lexicographical order.
- MUST NOT contain any empty label names or values.

Senders MUST only send valid metric names, label names, and label values:

- Metric names MUST adhere to the regex `[a-zA-Z_:]([a-zA-Z0-9_:])*`.
- Label names MUST adhere to the regex `[a-zA-Z_]([a-zA-Z0-9_])*`.
- Label values MAY be any sequence of UTF-8 characters .

Receivers MAY impose limits on the number and length of labels, but this will be receiver-specific and is out of scope for this document.

Label names beginning with "__" are RESERVED for system usage and SHOULD NOT be used, see Prometheus Data Model (https://prometheus.io/docs/concepts/data_model/).

Remote write Receivers MAY ingest valid samples within a write request that otherwise contains invalid samples. Receivers MUST return a HTTP 400 status code ("Bad Request") for write requests that contain any invalid samples. Receivers SHOULD provide a human readable error message in the response body. Senders MUST NOT try and interpret the error message, and SHOULD log it as is.

Ordering

Prometheus Remote Write compatible senders MUST send samples for any given series in timestamp order. Prometheus Remote Write compatible Senders MAY send multiple requests for different series in parallel.

Retries & Backoff

Prometheus Remote Write compatible senders MUST retry write requests on HTTP 5xx responses and MUST use a backoff algorithm to prevent overwhelming the server. They MUST NOT retry write requests on HTTP 2xx and 4xx responses other than 429. They MAY retry on HTTP 429 responses, which could result in senders "falling behind" if the server cannot keep up. This is done to ensure data is not lost when there are server side errors, and progress is made when there are client side errors.

Prometheus remote Write compatible receivers MUST respond with a HTTP 2xx status code when the write is successful. They MUST respond with HTTP status code 5xx when the write fails and SHOULD be retried. They MUST respond with HTTP status code 4xx when the request is invalid, will never be able to succeed and should not be retried.

Stale Markers

Prometheus remote write compatible senders MUST send stale markers when a time series will no longer be appended to.

Stale markers MUST be signalled by the special NaN value 0x7ff0000000000002. This value MUST NOT be used otherwise.

Typically the sender can detect when a time series will no longer be appended to using the following techniques:

1. Detecting, using service discovery, that the target exposing the series has gone away
2. Noticing the target is no longer exposing the time series between successive scrapes
3. Failing to scrape the target that originally exposed a time series
4. Tracking configuration and evaluation for recording and alerting rules

Out of Scope

This document does not intend to explain all the features required for a fully Prometheus-compatible monitoring system. In particular, the following areas are out of scope for the first version of the spec:

The "up" metric The definition and semantics of the "up" metric are beyond the scope of the remote write protocol and should be documented separately.

HTTP Path The path for HTTP handler can be anything - and MUST be provided by the sender. Generally we expect the whole URL to be specified in config.

Persistence It is recommended that Prometheus Remote Write compatible senders should persistently buffer sample data in the event of outages in the receiver.

Authentication & Encryption as remote write uses HTTP, we consider authentication & encryption to be a transport-layer problem. Senders and receivers should support all the usual suspects (Basic auth, TLS etc) and are free to add potentially custom authentication options. Support for custom authentication in the Prometheus remote write sender and eventual agent should not be assumed, but we will endeavour to support common and widely used auth protocols, where feasible.

Remote Read this is a separate interface that has already seen some iteration, and is less widely used.

Sharding the current sharding scheme in Prometheus for remote write parallelisation is very much an implementation detail, and isn't part of the spec. When senders do implement parallelisation they MUST preserve per-series sample ordering.

Backfill The specification does not place a limit on how old series can be pushed, however server/implementation specific constraints may exist.

Limits Limits on the number and length of labels, batch sizes etc are beyond the scope of this document, however it is expected that implementation will impose reasonable limits.

Push-based Prometheus Applications pushing metrics to Prometheus Remote Write compatible receivers was not a design goal of this system, and should be explored in a separate doc.

Labels Every series MAY include a "job" and/or "instance" label, as these are typically added by service discovery in the Sender. These are not mandatory.

Future Plans

This section contains speculative plans that are not considered part of protocol specification, but are mentioned here for completeness.

Transactionality Prometheus aims at being "transactional" - i.e. to never expose a partially scraped target to a query. We intend to do the same with remote write - for instance, in the future we would like to "align" remote write with scrapes, perhaps such that all the samples, metadata and exemplars for a single scrape are sent in a single remote write request. This is yet to be designed.

Metadata and Exemplars In line with above, we also send metadata (type information, help text) and exemplars along with the scraped samples. We plan to package this up in a single remote write request - future versions of the spec may insist on this. Prometheus currently has experimental support for sending metadata and exemplars.

Optimizations We would like to investigate various optimizations to reduce message size by eliminating repetition of label names and values.

Related

Compatible Senders and Receivers

The spec is intended to describe how the following components interact (as of April 2023):

- Prometheus (https://github.com/prometheus/prometheus/tree/master/storage/remote) (as both a "sender" and a "receiver")
- Avalanche (https://github.com/prometheus-community/avalanche) (as a "sender") - A Load Testing Tool Prometheus Metrics.
- Cortex (https://github.com/cortexproject/cortex/blob/master/pkg/util/push/push.go#L20) (as a "receiver")
- Elastic Agent (https://docs.elastic.co/integrations/prometheus#prometheus-server-remote-write) (as a "receiver")
- Grafana Agent (https://github.com/grafana/agent) (as both a "sender" and a "receiver")
- GreptimeDB (https://github.com/greptimeTeam/greptimedb) (as a "receiver" (https://docs.greptime.com/user-guide/write-data/prometheus#prometheus))
- InfluxData's Telegraf agent. (as a sender (https://github.com/influxdata/telegraf/tree/master/plugins/serializers/prometheusremotewrite), and as a receiver (https://github.com/influxdata/telegraf/pull/8967))
- M3 (https://m3db.io/docs/integrations/prometheus/#prometheus-configuration) (as a "receiver")
- Mimir (https://github.com/grafana/mimir) (as a "receiver")
- OpenTelemetry Collector (https://github.com/open-telemetry/opentelemetry-collector-releases/) (as a "sender" (https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/exporter/prometheusremotewriteexporter#readme) and eventually as a "receiver")
- Thanos (https://thanos.io/tip/components/receive.md/) (as a "receiver")
- Vector (as a "sender" (https://vector.dev/docs/reference/configuration/sinks/prometheus_remote_write/) and a "receiver" (https://vector.dev/docs/reference/configuration/sources/prometheus_remote_write/))
- VictoriaMetrics (https://github.com/VictoriaMetrics/VictoriaMetrics) (as a "receiver" (https://docs.victoriametrics.com/#prometheus-setup))

FAQ

Why did you not use gRPC? Funnily enough we initially used gRPC, but switched to Protos atop HTTP as in 2016 it was hard to get them past ELBs: https://github.com/prometheus/prometheus/issues/1982

(https://github.com/prometheus/prometheus/issues/1982)

Why not streaming protobuf messages? If you use persistent HTTP/1.1 connections, they are pretty close to streaming... Of course headers have to be re-sent, but yes that is less expensive than a new TCP set up.

Why do we send samples in order? The in-order constraint comes from the encoding we use for time series data in Prometheus, the implementation of which is append only. It is possible to remove this constraint, for instance by buffering samples and reordering them before encoding. We can investigate this in future versions of the protocol.

How can we parallelise requests with the in-order constraint? Samples must be in-order *for a given series*. Remote write requests can be sent in parallel as long as they are for different series. In Prometheus, we shard the samples by their labels into separate queues, and then writes happen sequentially in each queue. This guarantees samples for the same series are delivered in order, but samples for different series are sent in parallel - and potentially "out of order" between different series.

We believe this is necessary as, even if the receiver could support out-of-order samples, we can't have agents sending out of order as they would never be able to send to Prometheus, Cortex and Thanos. We're doing this to ensure the integrity of the ecosystem and to prevent confusing/forking the community into "prometheus-agents-that-can-write-to-prometheus" and those that can't.