Introduction

- Comparison to alternatives
- Design documents
- Frequently asked questions
- First steps with Prometheus
- Glossary
- Media
- Overview
- Long-term support
- Roadmap

Comparison to alternatives

Scope

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

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

Data model

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

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

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

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

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",instance="<sample1
>"} -> 34
api_server_http_requests_total{method="POST",handler="/tracks",status="500",instance="<sample2
>"} -> 28
api_server_http_requests_total{method="POST",handler="/tracks",status="500",instance="<sample3
>"} -> 31
```

Storage

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

Prometheus's local TSDB keeps recent samples in an in-memory head protected by a write-ahead log, then persists them in time-based block directories containing a chunks directory, an index, and metadata. It accepts samples at arbitrary intervals as scrapes or rule evaluations occur. Retention is configurable, and Prometheus also works well for many short-lived, frequently changing sets of time series.

Summary

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

Prometheus vs. InfluxDB

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

Scope

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

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

Data model / storage

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

InfluxDB uses a variant of a log-structured merge tree for storage with a write ahead log, sharded by time. This is more suitable for event logging, while Prometheus's block-based local TSDB is optimized for monitoring time series.

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

Architecture

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

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

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

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

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

Prometheus vs. OpenTSDB

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

Scope

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

Data model

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

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

Summary

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

Prometheus vs. Nagios

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

Scope

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

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

Data model

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.

Architecture

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

Summary

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

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

Prometheus vs. Sensu

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

Scope

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

Data model

Sensu Events represent service health and/or metrics in a structured data format identified by an entity name (e.g. server, cloud compute instance, container, or service), an event name, and optional key-value metadata called "labels" or "annotations". The Sensu Event payload may include one or more metric `points`, represented as a JSON object containing a `name`, `tags` (key/value pairs), `timestamp`, and `value` (always a float).

Storage

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

Architecture

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

Summary

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

Where Sensu is better:

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

Where Prometheus is better:

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

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.

Design documents

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

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

Problem statements and exploratory documents

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

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

Frequently asked questions

What is Prometheus?

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

How does Prometheus compare against other monitoring systems?

See the comparison page.

What dependencies does Prometheus have?

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

Is this cloud native?

Yes.

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

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

Can Prometheus be made highly available?

Yes, run identical Prometheus servers on two or more separate machines.

Systems like Thanos can deduplicate the data. Many of the external systems that support remote write can also deduplicate the data.

Identical alerts will be deduplicated by the Alertmanager.

Alertmanager supports high availability by interconnecting multiple Alertmanager instances to build an Alertmanager cluster.

I was told Prometheus “doesn't scale”.

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

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

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

What language is Prometheus written in?

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

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

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

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

Why do you pull rather than push?

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.

How to feed logs into Prometheus?

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

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

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

Who wrote Prometheus?

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

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

What license is Prometheus released under?

Prometheus is released under the Apache 2.0 license.

What is the plural of Prometheus?

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

If you cannot 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.

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

Can I create dashboards?

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

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

To avoid any kind of timezone confusion, especially when the so-called daylight saving time is involved, we decided to exclusively use Unix time internally and UTC for display purposes in all

components of Prometheus. A carefully done timezone selection could be introduced into the UI. Contributions are welcome. See issue #500 for the current state of this effort.

Instrumentation

Which languages have instrumentation libraries?

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

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

Can I monitor machines?

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

Can I monitor network devices?

Yes, the SNMP Exporter allows monitoring of devices that support SNMP. For industrial networks, there's also a Modbus exporter.

Can I monitor batch jobs?

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

What applications can Prometheus monitor out of the box?

See the list of exporters and integrations.

Can I monitor JVM applications via JMX?

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

What is the performance impact of instrumentation?

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

Implementation

Why are all sample values 64-bit floats?

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

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.

If you are new to monitoring, or want to learn more about Prometheus' architecture and components before getting hands-on, see the Overview.

Downloading Prometheus

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

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

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

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

The Prometheus monitoring server

. . .
```

Before starting Prometheus, let's configure it.

Configuring Prometheus

Prometheus configuration is YAML. 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.

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.

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. Give it about 30 seconds to collect data about itself from its own HTTP metrics endpoint.

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

Using the expression browser

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

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

```
promhttp_metric_handler_requests_total
```

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

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

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

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

```
count(promhttp_metric_handler_requests_total)
```

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

Using the graphing interface

To graph expressions, navigate to http://localhost:9090/query 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, let's add a second target: the node exporter, which exposes a wide variety of Linux or macOS host metrics like CPU, memory, and disk usage.

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

```
tar xvfz node_exporter-*.tar.gz
cd node_exporter-*
./node_exporter
```

The node exporter serves metrics on port 9100 by default. To have Prometheus scrape it, add a second job to the `scrape_configs` section of `prometheus.yml`:

```
scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: ['localhost:9090']
  - job_name: node_exporter
    static_configs:
      - targets: ['localhost:9100']
```

Restart Prometheus to pick up the new configuration. After about 30 seconds, the node exporter's metrics are available in the expression browser, for example:

```
node_memory_MemAvailable_bytes
```

To explore other exporters and what they monitor, see the exporters and integrations documentation. The Monitoring Linux or macOS host metrics using a node exporter guide goes into

more depth about monitoring hosts with the node exporter.

Summary

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

Glossary

Alert

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

Alertmanager

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

Bridge

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

Client library

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

Collector

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

Direct instrumentation

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.

Mixin

A mixin is a reusable and extensible set of Prometheus alerts, recording rules, and Grafana dashboards for a specific component or system. Mixins are typically packaged using Jsonnet and can be combined to create comprehensive monitoring configurations. They enable standardized monitoring across similar infrastructure components.

Notification

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

Promdash

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

Prometheus

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

PromQL

PromQL is the Prometheus Query Language. It allows for a wide range of operations including aggregation, slicing and dicing, prediction and joins.

Pushgateway

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

Recording Rules

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

Remote Read

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

Remote Read Adapter

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

Remote Read Endpoint

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

Remote Write

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

Remote Write Adapter

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.

Media

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

The following selection of resources are particularly useful to get started with Prometheus.

Awesome Prometheus contains a more comprehensive community-maintained list of resources.

Blogs

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

Tutorials

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

Podcasts and interviews

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

Recorded talks

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

- What is your application doing right now? – Matthias Gruter, Transmode, at DevOps Stockholm Meetup.
- Prometheus workshop – Jamie Wilkinson at Monitorama PDX 2015 (slides).
- Monitoring Hadoop with Prometheus – Brian Brazil at the Hadoop User Group Ireland (slides).
- In German: Monitoring mit Prometheus – Michael Stapelberg at Easterhegg 2016.
- In German: Prometheus in der Praxis – Jonas Große Sundrup at MRMCD 2016

Presentation slides

General

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

Docker

- Prometheus and Docker – Brian Brazil at Docker Galway.

Python

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

Overview

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

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

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

Features

Prometheus's main features are:

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

What are metrics?

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

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

Architecture

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

Prometheus architecture

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

When does it fit?

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

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

Long-term support

Prometheus LTS are selected releases of Prometheus that receive selected bug fixes for an extended period of one year.

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

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

Prometheus LTS version will only receive fixes for selected issues with high severity, i.e. a CVSS rating of 7.0 or higher. Additional changes for bug fixes or documentation fixes might be added. The build toolchain will also be kept up-to-date. This is **a best effort commitment by the community**, contributions are welcome. Support periods of two subsequent LTS version should overlap by at least one month. This allows users that rely on Prometheus to limit the upgrade risks while still having a Prometheus server maintained by the community.

List of LTS releases

Limitations of LTS support

Some features are excluded from LTS support:

- Unstable features as listed in our API stability guarantees.
- Experimental features.
- OpenBSD support.

Roadmap

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

Server-side metric metadata support

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

Adopt OpenMetrics

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

Retroactive rule evaluations

Add support for retroactive rule evaluations making use of backfill.

TLS and authentication in HTTP serving endpoints

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

Support the Ecosystem

Prometheus 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.