

# Getting Started With Prometheus

COLIN DOMONEY  
API SECURITY RESEARCH SPECIALIST & DEVELOPER ADVOCATE, 42CRUNCH

Prometheus has become the de facto system for the monitoring and alerting of distributed systems and architecture. Key benefits of Prometheus include an extensible design architecture, efficient data storage, high scalability for large environments, an integrated query language, and many excellent integrations.

In this Refcard, we explore the core components of the Prometheus architecture and key concepts — then focus on getting up and running with Prometheus, including configuration and both collecting and working with data.

## WHAT IS PROMETHEUS?

[Prometheus](#) is a modern, open-source metric-based monitoring system designed around a highly efficient time series database and optimized query language. Prometheus has been designed to overcome shortcomings in existing monitoring solutions, including a lack of scalability, deployment challenges requiring the installation of agents, and a lack of advanced query and reporting features. And further, its core architecture, a variety of data collection and discovery services, and third-party integrations help to derive greater value. Grafana dashboards, for example, provide metric visualization.

Prometheus does one job well: monitoring *targets* (anything that can emit metrics via HTTP can be a Prometheus target). It should not be used as a long-term storage system or a business intelligence reporting platform where more niche solutions are preferable.

Central to Prometheus are *metrics* that are ingested from *targets*; let's explore further.

## METRIC TYPES


A **metric** is anything that can be **observed** and **measured** on any target being monitored by Prometheus. The type of metric emitted by a target depends on the target type — a server might report CPU load and memory use, whilst a web server might report requests or errors.

### CONTENTS

- What Is Prometheus?
- Get Started With Prometheus
  - Prometheus Architecture and Components
  - Configuring Prometheus
  - Running Prometheus
  - Configuring Prometheus
- Prometheus Key Methods and Techniques
  - Collecting Data
  - Working With Data
  - Working With Alerts
  - Using Prometheus in Production
- Conclusion

Prometheus excels at efficiently storing metrics in a [time series database](#) (where each entry is timestamped) using, on average, 3.5 bytes per record. There are four types of metrics in Prometheus, namely:

TYPE	PURPOSE	EXAMPLE
Counter	The most elementary metric type; represents an (increasing) value on the target	The uptime of a server
Gauge	Represents a continuous value of a target, though is a snapshot that can go up and down over time	A CPU load on a server
Histogram	A calculated metric that shows the distribution of a metric on an x-axis (the percentile)	CPU load distribution shown by grouping load into 10 buckets in 10% increments
Summary	A calculated metric that performs a computation or transformation on a series to produce a scalar summary (e.g., count, average, min, max)	The average CPU load over the last 10 minutes

 chronosphere

Ready to stop managing your own Prometheus?

Running your own Prometheus deployment requires significant management overhead, and many organizations are exploring moving to a hosted SaaS offering. Learn what important questions to consider in your search.

DOWNLOAD THE GUIDE

# Ready to stop managing your own Prometheus?

Running your own Prometheus deployment requires significant management overhead, and many organizations are exploring moving to a hosted SaaS offering to scale faster and more reliably. Explore the key capabilities to consider when selecting a Prometheus-native monitoring solution.

[DOWNLOAD THE GUIDE](#)



## METRICS NAMES AND LABELS

Metrics are named according to a [loose convention](#) based on the target type and measurement type. Multiple targets may emit metrics to the same label (e.g., the **up** metric indicates if the target is up), and to distinguish the targets, the concept of **labels** is used. A label is a tag applied to a metric to convey additional context to a metric. Label names can be chosen arbitrarily, although certain conventions exist — for example, **instance** is a defined target name in Prometheus.

Using a combination of metrics and labels, precise queries can be executed against the database. As an example, here is a query to display if the **localhost** exporter is up:

```
up{instance="localhost:9090"} 1
```

A note of caution: Each unique combination of metric and label creates a separate time series database that utilizes resources. Be wary of using labels that may have a large number of values, such as UUIDs or customer numbers. This will increase cardinality and potentially overwhelm a Prometheus instance.

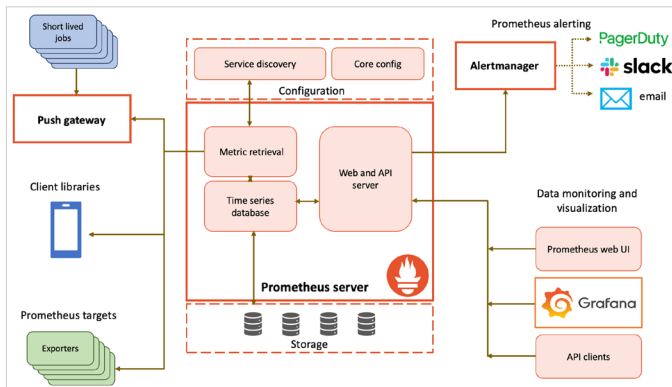
## GET STARTED WITH PROMETHEUS

Now that we know the basics of Prometheus, let's get it up and running.

### PROMETHEUS ARCHITECTURE AND COMPONENTS

Prometheus uses an extensible architecture built around a core server, several extension modules, and an API for integration, as shown here:

Figure 1



Source: Diagram adapted from the [Prometheus Documentation](#)

The **Prometheus server** is a single binary (supporting multiple hosts, including Docker) responsible for retrieving metrics from multiple targets by polling them at an interval (pull model). The samples are stored in a time series database according to their name/label combinations. An integral HTTP server provides a basic web UI capable of showing status and configuration as well as executing queries against the database.

The **Service Discovery** is an important feature that simplifies Prometheus deployment and allows dynamic discovery of targets. The **Pushgateway** allows Prometheus to ingest metrics from short-lived jobs (e.g., a short batch job). The **Alertmanager** is responsible for generating alerts to various back ends based on configurable queries against Prometheus.

## CONFIGURING PROMETHEUS

Prometheus is configured via a YAML configuration file, which is loaded at startup and can be dynamically reloaded by sending the process the **SIGHUP** signal. The **global:** section contains core parameters with sensible default values — of these the most important is the **scrape\_interval:**, which specifies the target polling interval. Too short an interval can cause excessive load on the Prometheus server. The next section is the **scrape\_configs:** section, which contains the configuration of the targets to be polled.

Prometheus excels in its flexible approach to discovering targets with the following options supported:

- Static configurations in the **static\_config:** section typically specify a hostname and port to be polled.
- File-based discovery specifies one or more configuration files that are polled periodically, specifying targets, which allows a dynamic discovery mechanism.
- Several popular services have dedicated service discovery mechanisms — for instance, Prometheus can use introspection on a Kubernetes cluster to discover nodes, pods, endpoints, etc.

Prometheus provides great flexibility in the ingestion of metrics from targets using **relabel\_config:** and **metric\_relabel\_configs:**, which allow metric names and labels to be modified, dropped, or added according to various rules using regular expressions. These rules can be extremely useful in normalizing metrics, adding useful metadata, or suppressing the ingestion of excess data.

## RUNNING PROMETHEUS

Prometheus can be installed from pre-built binaries for popular platforms, built from source via a **Makefile**, run in Docker, or installed using several prominent configuration management tools. Prometheus supports local filesystem storage by default and allows the use of remote filesystems. It is also possible to import existing data in the OpenMetrics format. The Prometheus server also emits its own metrics (on **localhost:9090**), which should be monitored to ensure adequate server performance.

## PROMETHEUS KEY METHODS AND TECHNIQUES

Now that we have Prometheus configured and up and running, let's start working with the key topics of data and alerts.

### COLLECTING DATA

Firstly, let's investigate how to collect data from targets.

### EXPORTERS

Prometheus' popularity as the de facto standard for monitoring is in no small part down to the richness of the exporter ecosystem. Out of the box, Prometheus supports several official exporters, whilst the community has created a portfolio of exporters that cover popular hardware, software, databases, logging tools, etc. An exporter is responsible for extracting native metrics from the underlying system,

packaging them into a set of metrics, and then exposing them over an HTTP endpoint so Prometheus can poll them. A good example is the [node\\_exporter](#), which exports key operating system metrics such as the *uptime*, as shown below:

```
node_boot_time_seconds{instance="host.docker.internal:9100", job="m1pro"}
1657525619.404136
```

To debug exporters, navigate to the **/metrics** endpoint on the target host where the metrics will be displayed in plain text in a browser, or use the **Status | Targets** panel on the Prometheus dashboard, as shown:

Figure 2



Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://host.docker.internal:9100/metrics	UP	instance="host.docker.internal:9100" job="m1pro"	4.900s ago	20.877ms	
http://localhost:9090/metrics	UP	instance="localhost:9090" job="prometheus"	8.702s ago	3.661ms	

In the example above, two targets are configured, and both are currently *up*. This view also shows the associated labels, last scrape time, and scrape duration.

It is also relatively easy to [write custom exporters](#) if you are unable to find one that matches your exact needs. To discover the available exporters, use the [ExporterHub](#) portal, the [PromCat](#) portal, or the official [GitHub repository](#).

## SERVICE DISCOVERY

Once we have configured exporters, Prometheus needs to know how to discover these targets, and this is another area where Prometheus really shines. As discussed in the configuration section above, Prometheus can use static configurations; however, it is the variety of dynamic discovery options that are particularly powerful. Some of the prominent integrated service discovery mechanisms include Docker, Kubernetes, OpenStack, Azure, EC2, and GCE.

As an example, let's look at monitoring a Docker instance: Firstly, Docker must be [configured to emit metrics](#), and then a [configuration needs to be made](#) to the Prometheus configuration by adding the **docker\_sd\_configs**: to reference the Docker host. These are the two necessary steps to take for monitoring and hosting containers.

## APPLICATION INSTRUMENTATION

There is one more important source of metric data: via application instrumentation libraries. Since Prometheus is agnostic to the meaning of the metrics, application developers can use client libraries to emit any application metric of their choosing to Prometheus — for example, the number of failed accesses to a database. By monitoring application-level metrics, developers are able to diagnose both runtime errors and performance bottlenecks.

Official support is provided for the following languages: Go, Java or Scala, Python, Ruby, and Rust. Many other [unofficial libraries](#) also exist for additional popular languages.

The use of client libraries is best illustrated with a [client code sample](#), in this case, Python:

```
from prometheus_client import start_http_server,
Summary
import random
import time

# Create a metric to track time spent and requests
made.
REQUEST_TIME = Summary('request_processing_seconds',
'Time spent processing request')

# Decorate function with metric.
@REQUEST_TIME.time()
def process_request(t):
    """A dummy function that takes some time."""
    time.sleep(t)

if __name__ == '__main__':
    # Start up the server to expose the metrics.
    start_http_server(8000)
    # Generate some requests.
    while True:
        process_request(random.random())
```

This simple code snippet shows what is necessary to emit metrics from Python, namely:

- Import the **prometheus\_client** library
- Create a metric called **'request\_processing\_seconds'**
- Use a Python decorator, **@REQUEST\_TIME.time()**, to measure the **process\_request()** method
- Call the **start\_http\_server(8000)** method to start the metric server

The metrics are now exposed on **http://localhost:8000/**.

## PUSH GATEWAY

Finally, there is one more method to serve metrics to Prometheus: the [Pushgateway](#). This solves a specific problem related to the Prometheus pull model, specifically, how to ingest metrics from short-lived tasks, which will disappear before Prometheus is able to poll them. To use the Pushgateway, it is necessary to run an instance of the server on-premises, configure Prometheus to pull from that, and then use the appropriate Pushgateway client code for emitting to the Pushgateway (push model).

The Prometheus team doesn't recommend the widespread use of this model as it adds complexity and single point of failure.

## WORKING WITH DATA

Now that we looked at the various ways data can be ingested, let's explore how we can work with this data.

### QUERYING DATA WITH PROMQL

Core to Prometheus is the [dedicated query language](#), PromQL, which is designed specifically for querying a time series database. If you come from a background of using structured query languages (SQL), the query structure will take some getting used to but is easily mastered by practice or using the [many samples provided](#).

A query returns one of four data types, namely:

- **Instant vector** – A range of time series comprising a single sample for each time series, all taken at the same instant in time. Typically, this retrieves the current value.
- **Range vector** – A range of time series comprising a series of data points over a specified time for each time series. Think of this as a two-dimensional array.
- **Scalar** – A simple numeric floating-point value (e.g., sum, average).
- **String** – A simple string value (not in general use at present).

To illustrate basic PromQL queries and results, I will use an instance of Prometheus in Docker and the `node_exporter` running on the host to produce demo data.

The most basic query retrieves the last sample in a series, for example:

```
go_threads
```

A typical result is returned below, in this case indicating that the 16 threads are active:

```
go_threads{instance="host.docker.internal:9100",
job="m1pro"} 16
go_threads{instance="localhost:9090",
job="prometheus"} 11
```

Note that two times series are returned since two targets are being monitored, both of which export `go_threads`.

We can apply a label selector to the query to further refine the result set as follows:

```
go_threads{job="m1pro"}
```

In my demo instance, the following result is returned:

```
go_threads{instance="host.docker.internal:9100",
job="m1pro"} 16
```

Selectors can be combined and support regular expressions.

Now, let's add a range selector to return a data series over a 1-minute window:

```
go_threads{job="m1pro"}[1m]
```

In my demo instance, the following result is returned:

```
go_threads{instance="host.docker.internal:9100",
job="m1pro"}
16 @1657708139.523
16 @1657708154.523
16 @1657708169.525
16 @1657708184.526
```

We now have a time series of data for 1 minute (using a 15-second poll interval) rather than the scalar values received previously.

PromQL supports a number of [standard binary operators](#) (e.g., addition, subtraction, division, multiplication), which can be used to produce calculated metrics. Beware of mixing different incompatible result types — for example, mixing a range vector with a scalar.

In addition to these binary operators, Prometheus also supports several aggregation operators, which can produce summary results such as `sum()`, `min()`, `max()`, `average()`, etc.

Additionally, Prometheus has powerful [built-in functions](#) that can perform mathematical and statistical operations on time series data.

### RECORDING RULES

Prometheus provides a feature called [recording rules](#) to perform precomputed queries on a time series to produce a new, derived time series. The primary advantage of a precomputed time series is to avoid repeated, computationally intensive queries — it is more efficient to perform the computation once and store the result.

This is especially useful for dashboards, which need to query the same expression repeatedly every time they refresh.

### VISUALIZING DATA WITH GRAFANA

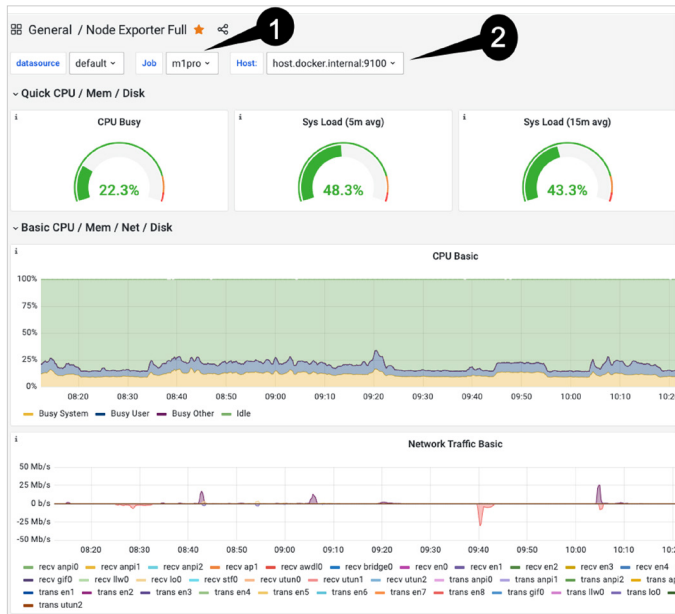
The jewel in the crown for visualizing Prometheus metrics is via integration with the equally popular open-source tool, Grafana. Recent versions of Grafana provide native support for Prometheus servers, allowing developers to run Prometheus queries within Grafana to produce dashboards that include graphs, heatmaps, trends, dials, etc.

In Figure 3 on the following page is an example Grafana dashboard showing local `node_exporter`.

SEE FIGURE ON NEXT PAGE



Figure 3



The only steps required for connecting Grafana to Prometheus are to add a data source and then to add the widgets or dashboards as needed. Grafana provides numerous [ready-made dashboards](#) to visualize common scenarios, and Prometheus is well supported.

## WORKING WITH ALERTS

The key to the Prometheus philosophy is focusing on what it does best and delegating other features, such as visualization with Grafana. This holds true for managing and generating alerts from the underlying Prometheus data series.

### USING ALERTMANAGER

The standalone component, Alertmanager, performs the key functions related to alerts, such as deduplication of alerts, grouping of alerts to relevant receivers, integration with standard alert providers, and suppression of noisy alerts as necessary. Additionally, as a separate component, [Alertmanager](#) is designed with high availability (HA) in mind. Alertmanager is also configured via a configuration file that specifies the following:

- Global settings such as SMTP and various timeouts
- Configuration of routes for alerts to be sent
- Configuration of one or more receivers that will receive the alerts

Common receivers include email, Slack, OpsGenie, PagerDuty, Telegram, and standard webhooks.

### ALERTING RULES

To generate alerts, Prometheus is [configured with alerting rules](#), which use exactly the same format as the recording rules discussed previously — they are simply Prometheus queries. Template rules can be used to prettify the alert content sent by using the Go templating engine to enrich the alerts. Again, the community shines with a fine [selection of alerting rules](#).

## USING PROMETHEUS IN PRODUCTION

Finally, let's review some considerations for using Prometheus in production.

### SCALING AND AUGMENTING PROMETHEUS

Prometheus is used in major enterprise organizations and can scale to enormous capacity. To understand your scaling requirements, consider the following questions:

- How many metrics can your monitoring system ingest, and how many do you need?
- What's your metrics cardinality? Cardinality is the number of labels that each metric can have. This is a common issue with metrics coming from dynamic environments, where containers get a different ID or name every time they start, restart, or are moved between nodes.
- Do you need high availability?
- How long do you need to keep metrics, and with what resolution?

Implementing HA is tricky because clustering requires third-party add-ons to the Prometheus server, you need to deal with backups and restores, and storing metrics for an extended period will make your database grow exponentially.

### LONG-TERM STORAGE FOR PROMETHEUS

Prometheus servers provide persistent storage, but Prometheus was not created for distributed metrics storage across multiple nodes in a cluster with replication and healing capabilities. This is known as long-term storage, which is a requirement in a few common use cases, such as:

- Capacity planning to monitor how your infrastructure needs to evolve
- Chargebacks, so you can account for and bill different teams or departments for their specific use of the infrastructure
- Analyzing usage trends
- High availability and resiliency
- Adhering to regulations for certain verticals like banking, insurance, etc.

### SECURITY

A note on [security considerations for operating Prometheus](#) in production environments — presently, Prometheus and associated components allow open, unauthenticated access, including to raw data and administration functionality. As such, users should make sure to restrict access to the network segment associated with Prometheus, and restrict physical access to the servers hosting Prometheus.

At the time of writing, Prometheus had added [experimental support for basic authentication and TLS](#) on the web UI. According to the [Prometheus team](#), "In the future, server-side TLS support will be rolled out to the different Prometheus projects. Those projects include Prometheus, Alertmanager, Pushgateway, and the official exporters. Authentication of clients by TLS client certs will also be supported."

## CONCLUSION

Prometheus is, without a doubt, the go-to open-source platform for infrastructure and application monitoring. The benefits include its flexible, extensible architecture; a rich feature set; several out-of-the-box integrations; and above all, widespread adoption and community support. This Refcard has provided a quick overview of the key concepts in Prometheus, equipping you with the basics for understanding how to get started with Prometheus. Hopefully, you'll take the next step on your Prometheus journey — grab the Docker image and the `node_exporter` for your platform and start monitoring!

Additional resources and further reading:

- Prometheus Docs – <https://prometheus.io/docs/introduction/overview>
- Awesome Prometheus – <https://project-awesome.org/roaldnefs/awesome-prometheus>
- Prometheus GitHub – <https://github.com/prometheus/prometheus>
- "Don't let Prometheus Steal your Fire" – <https://jfrog.com/blog/dont-let-prometheus-steal-your-fire>
- "Full-Stack Observability Essentials" Refcard – <https://dzone.com/refcardz/full-stack-observability-essentials>

### WRITTEN BY COLIN DOMONEY,

API SECURITY RESEARCH SPECIALIST & DEVELOPER  
ADVOCATE, 42CRUNCH



Colin Domoney oversees the development of the 42Crunch community and curates the [apisecurity.io](https://apisecurity.io) industry newsletter. Colin is an experienced DevSecOps professional, having worked with Cyberproof, Veracode, and CA, oversaw Deutsche Bank's global AppSec program, and consulted with numerous Fortune 500 companies. Colin is also a regular conference speaker and DevOps instructor.



600 Park Offices Drive, Suite 300  
Research Triangle Park, NC 27709  
888.678.0399 | 919.678.0300

At DZone, we foster a collaborative environment that empowers developers and tech professionals to share knowledge, build skills, and solve problems through content, code, and community. We thoughtfully — and with intention — challenge the status quo and value diverse perspectives so that, as one, we can inspire positive change through technology.

Copyright © 2022 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means of electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.