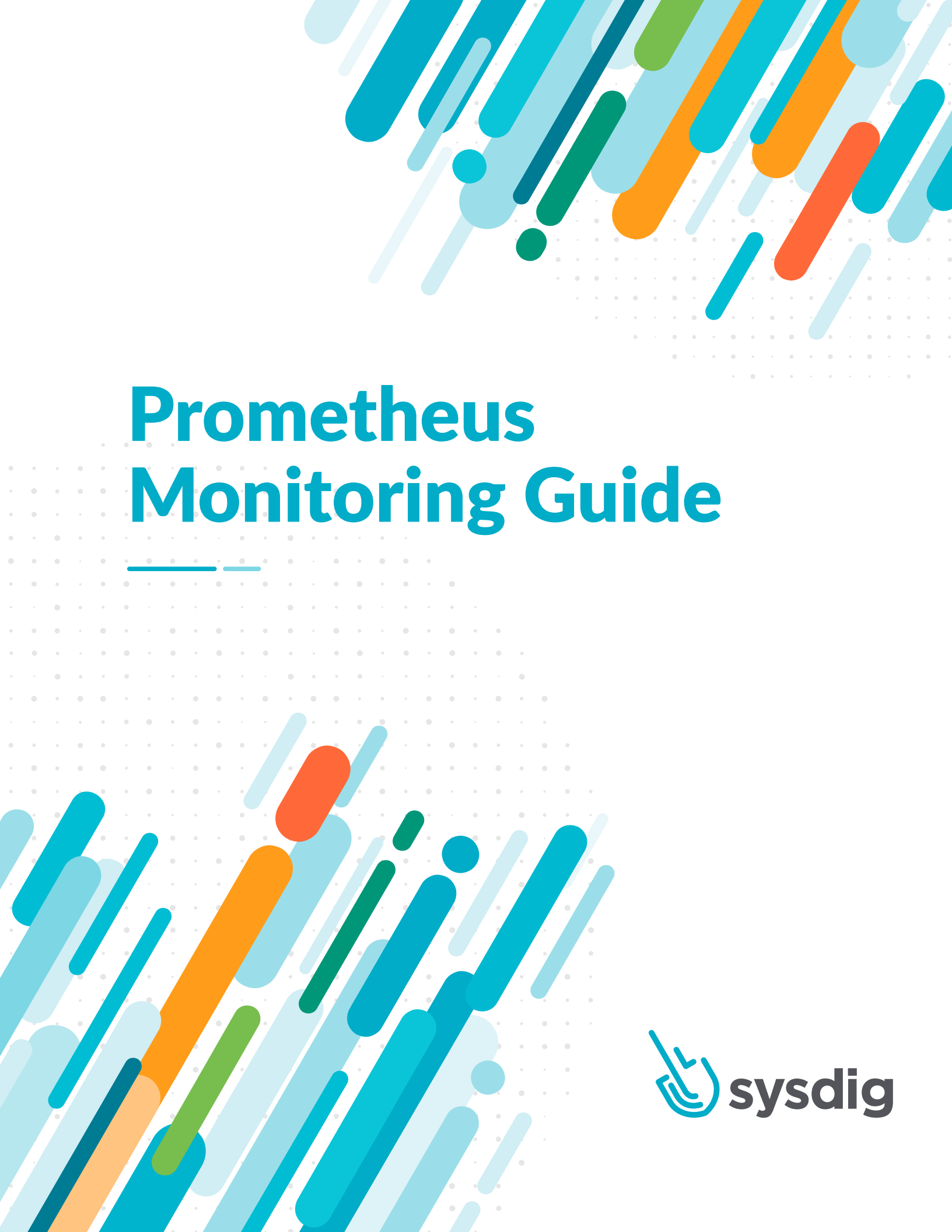
Prometheus Monitoring Guide



Contents

About This Guide

Prometheus is one of the foundations of the cloud-native environment. It has become the de-facto standard for visibility in Kubernetes environments, creating a new category called Prometheus monitoring. The Prometheus journey is usually tied to the Kubernetes journey and the different development stages, from proof of concept to production.

This guide will walk you through:

1. Why Prometheus monitoring is important
2. How to install Prometheus
3. How to use exporters for third-party applications
4. How to monitor your own applications
5. How to scale Prometheus

By learning how to use Prometheus to monitor your environments, you will be able to better understand the complex relationships between containers, services, and applications. These relationships can impact the performance and health of your most critical workloads.

Intro to Prometheus Monitoring

Prometheus is a monitoring tool. This means that it mainly stores time series of numerical values. Each of these time series is identified by a combination of text based labels, which makes it filter through its query language called PromQL.

But before digging deeper into Prometheus features that made it the de-facto monitoring standard, let's talk about what Prometheus is not. Prometheus is not a logging tool. It has no way to store lines of log or parse them. The values of the timeseries are strictly numerical. Also, Prometheus is not a tracing tool. Although, as we will see later, you can instrument your code to expose Prometheus metrics, they are just that: metrics. Finally, Prometheus is not a data lake. Although it has great flexibility and power to gather and index metrics, Prometheus is not designed to store and serve millions of time series over several months or years.

Now that we have defined what Prometheus is and isn't used for, let's dive into some of the features that helped it become the most widespread monitoring tool in cloud native environments. One of the most popular features is how easy it is to have Prometheus up and running in a Kubernetes cluster. Also, with some annotation in your pods (some of them come already by default), Prometheus can automatically discover metrics and start to scrape them literally from the first minute. This autodiscovery is not only limited to Kubernetes, but to platform specific resources like AWS, Azure, GCE, and others.

Prometheus Metric Structure

Another strong point of Prometheus is the use of labels to define the time series, as opposed to dot-separated metrics. The Prometheus approach makes the metrics totally horizontal and flexible, eliminating the restrictions and hierarchy of the dot-separated model. However, one of the consequences of the label-based metric model is the risk of cardinality explosion (a combination of labels that generates an enormous number of different metrics).

Since we are talking about Prometheus metrics, let's examine what they actually look like. Prometheus gathers the metrics from endpoints where the applications expose their metrics in text format. A Prometheus metric usually has a name, a set of labels, and the numeric value for that combination of labels. Here is what a Prometheus metric looks like:

```
prometheus_metric{label_01="label_A", label_02="label_B"} 1234
```

Also, there is usually some additional information about the metric and the type of data that it uses that is displayed in a human readable way. Here's an example:

```
# HELP prometheus_metric This is the information about the metric.
# TYPE prometheus_metric gauge
```

This way of presenting the metrics is very powerful, simple, and useful. You don't need a special program to check if an application is generating metrics. Rather, you can just open the web browser as if it were a usual web page to start seeing metrics. Also, the text-based format makes it easy to explore the metrics and its labels.

PromQL

It's easy to start filtering some labels with the Prometheus query language (promQL). According to the documentation, "Prometheus provides a functional query language called PromQL 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." In some cases, you may find yourself needing to create aggregations to sum the values of all the metrics with a certain label, or starting to make simple mathematical expressions with different metrics to generate your own indicators and rates. With some practice, you will be able to create complex queries by folding and grouping the metrics at will to extract the information that you need. That's the power of the PromQL language and another reason why Prometheus became that popular.

But you can use the PromQL language to make, not only visualization of data, but also alerting. Prometheus allows you to create rules that can be used for alerting. These alerts have all the power of PromQL. You can compare different metrics creating thresholds of rates between different values, multiple conditions, etc. The flexibility that it gives is one of keys that allows the SRE team to fine tune the alerts to get to that sweet spot of alerting when it is really needed, minimizing the false positives alerts while covering all of the potentially disastrous scenarios.

One typical use case is to alert when an error rate has reached a certain threshold. This alert can't be done by just counting the total number of errors, because having 100 errors doesn't necessarily mean the same thing when the system is serving 500 requests and when it's serving 1M requests. PromQL allows you to do mathematical functions between metrics to get that ratio. For example, the following query will get the error ratio of an HTTP server:

```
sum (rate(apache_http_response_codes_total{code=~'4..|5..'}[5m])) /
sum (rate(apache_http_response_codes_total[5m]))
```

Also, by using math with promQL we can alert when the mean process time in a database is over a certain limit. The next query calculates the response time by dividing the processing time by the number of operations performed by the database:

```
sum(rate(redis_commands_duration_seconds_total[5m])) /
sum(rate(redis_commands_processed_total[5m]))
```

There is a rich ecosystem that has grown around Prometheus:

- Libraries for instrumenting the code in different languages.
- Exporters that extract data from applications and generate metrics in Prometheus format.
- Hubs of knowledge where you can find trusted resources and documentation.

All of this makes it easy to find the resources and information that you need when you want to start a new monitoring project with Prometheus. We will cover these in the following sections.

As we have covered in this short introduction, it's easy to see why Prometheus has become the default choice when it comes to monitoring a cloud infrastructure, cluster, or application. In the following chapters, we will go deeper into the details that you need to be aware of if you want to have a Prometheus system up and running reliably in production.

Lessons Learned

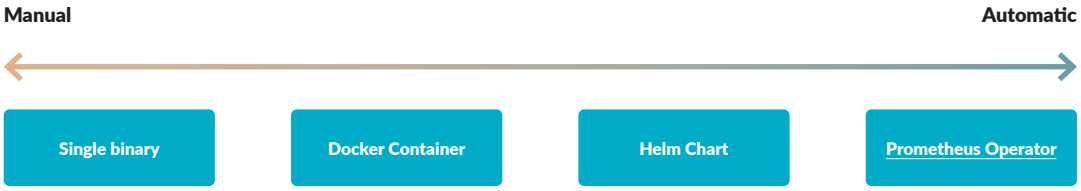
1. Prometheus is a monitoring tool designed for Kubernetes environments that can collect and store time series data.
2. PromQL can be used to query this data in powerful ways to be displayed in graphical form or used by API.

Installing Prometheus

There are different ways to install Prometheus in your host or in your Kubernetes cluster:

- Directly as a single binary running on your hosts, which is fine for learning, testing, and development purposes but not appropriate for a containerized deployment.
- As a Docker container, which has several orchestration options:
  - Raw Docker containers, Kubernetes Deployments / StatefulSets, the Helm Kubernetes package manager, Kubernetes operators, etc.

Depending on the installation method you choose, installing Prometheus can be done manually or by automated deployments:



You can directly download and run the Prometheus binary in your host:

```
prometheus-2.21.0.linux-amd64$ ./prometheus
./prometheus
level=info ts=2020-09-25T10:04:24.911Z caller=main.go:310 msg="No
time or size retention was set so using the default time retention"
duration=15d
[…]
level=info ts=2020-09-25T10:04:24.916Z caller=main.go:673 msg="Server
is ready to receive web requests."
```

This may be nice to get a first impression of the Prometheus web interface (port 9090 by default). However, a better option is to deploy the Prometheus server inside a container:

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

Note that you can easily adapt this Docker container into a proper Kubernetes Deployment object that will mount the configuration from a ConfigMap, expose a service, deploy multiple replicas, etc. Anyway, the easiest way to install Prometheus in Kubernetes is using Helm.

The Prometheus community is maintaining a Helm chart that makes it really easy to install and configure Prometheus and the different applications that form the ecosystem. To install Prometheus in your Kubernetes cluster, just run these commands:

Add the Prometheus charts repository to your helm configuration:

```
helm repo add prometheus-community https://prometheus-community.
github.io/helm-charts
helm repo add stable https://kubernetes-charts.storage.googleapis.
com/
helm repo update
```

Install Prometheus:

```
# Helm 3
helm install [RELEASE_NAME] prometheus-community/prometheus
# Helm 2
helm install --name [RELEASE_NAME] prometheus-community/prometheus
```

After a few seconds, you should see the Prometheus pods in your cluster.

```
NAME                                             READY   STATUS
RESTARTS   AGE
prometheus-kube-state-metrics-66cc6888bd-x9llw   1/1   Running   0
93d
prometheus-node-exporter-h2qx5                   1/1   Running   0
10d
prometheus-node-exporter-k6jvh                   1/1   Running   0
10d
prometheus-node-exporter-thtsr                   1/1   Running   0
10d
prometheus-server-0                              2/2   Running   0
90m
```

Bonus point: Helm chart deploys node-exporter, kube-state-metrics, and alertmanager along with Prometheus, so you will be able to start monitoring nodes and cluster state right away.

A more advanced and automated option is to use the Prometheus operator, which is covered in great detail on the Sysdig blog. You can think of it as a meta-deployment, a deployment that manages other deployments and configures and updates them according to high-level service specifications.

Prometheus exporters

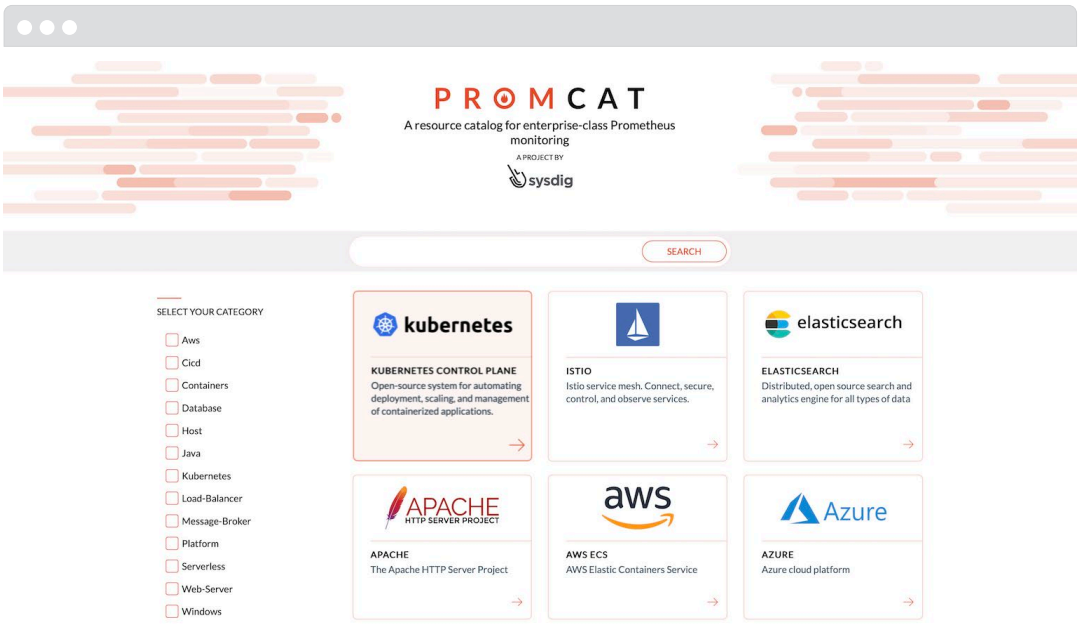
Although some services and applications are already adopting Prometheus metrics format and provide endpoints for this purpose, many popular server applications like Nginx or PostgreSQL have been around much longer than Prometheus metrics / OpenMetrics. They usually have their own metrics formats and exposition methods. If you are trying to unify your metric pipeline across many microservices and hosts using Prometheus metrics, this may be a problem.

To work around this hurdle, the Prometheus community is creating and maintaining a vast collection of Prometheus exporters. An exporter is a "translator" or "adapter" program able to collect the server native metrics (or generate its own data observing the server behavior) and re-publishing these metrics using the Prometheus metrics format and HTTP protocol transports.

These small binaries can be co-located in the same pod as a sidecar of the main server that is being monitored, or isolated in their own pod or even a different infrastructure. Then you can collect the service metrics scraping the exporter that exposes those converted into Prometheus metrics.

There are hundreds of Prometheus exporters available on the internet, and each exporter is as different as the application that they generate metrics for. In most of the cases, the exporter will need an authentication method to access the application and generate metrics. These authentications come in a wide range of forms, from plain text url connection strings to certificates or dedicated users with special permissions inside of the application. In other scenarios, it may need to mount a shared volume with the application to parse logs or files, for example. Also, sometimes the application needs some tuning or special configuration to allow the exporter to obtain the data and generate metrics.

Occasionally, there is more than one exporter for the same application. This can be because they offer different features, they are forked or discontinued projects, or even different exporters for different versions of the application in question. It's important to correctly identify the application that you want to monitor, the metrics that you need, and the proper exporter that can give you the best approach to your monitoring solution.



To reduce the amount of maintenance needed to find, validate, and configure these exporters, Sysdig has created a site called PromCat.io where we curate the best exporters, provide detailed configuration examples, and support our customers who want to use them. Check the up-to-date list of available Prometheus exporters and integrations here.

Example of exporter installation

MongoDB exporter gathers metrics from MongoDB and exposes them in Prometheus format. To install the MongoDB exporter in a Kubernetes cluster, we will use the Helm chart available.

First, create a values.yaml file with the following parameters:

```
fullnameOverride: “mongodb-exporter”
podAnnotations:
  prometheus.io/scrape: “true”
  prometheus.io/port: “9216”
serviceMonitor:
  enabled: false
mongodb:
  uri: mongodb://exporter-user:exporter-pass@mongodb:27017
```

Note that the mongodb.uri parameter is a valid MongoDB URI. In this URI, include the user and password of the exporter. The Helm chart will create a Kubernetes Secret with the URI so it's not visible. The user in MongoDB has to be created in the MongoDB console with the commands:

```
use admin
db.auth("your-admin-user", "your-admin-password")
db.createUser(
   {
      user: "exporter-user",
      pwd: "exporter-pass",
      roles: [
        { role: "clusterMonitor", db: "admin" },
        { role: "read", db: "admin" },
        { role: "read", db: "local" }
      ]
   }
)
```

Change username and password per your taste.

The metrics will be available in the port 9216 of the exporter pod.

Then, install the exporter in your K8s cluster with the commands:

```
helm repo add prometheus-community https://prometheus-community.
github.io/helm-charts
# Helm 3
helm install  mongodb-exporter prometheus-community/prometheus-
mongodb-exporter -f values.yaml
# Helm 2
helm install --name mongodb-exporter prometheus-community/prometheus-
mongodb-exporter -f values.yaml
```

You can list pods and check that the exporter is running:

```
k get pod
NAME                                  READY   STATUS    RESTARTS   AGE
mongodb-exporter-6cf765c9df-lgjsv     1/1     Running   0          3m
```

Monitoring Applications with Prometheus

In the following example-driven tutorial, you will learn how to use Prometheus metrics / OpenMetrics to instrument your code whether you are using Golang, Java, Python, or Javascript. We will cover the different metric types and provide readily executable code snippets.

Prometheus is an open source time series database for monitoring that was originally developed at SoundCloud before being released as an open source project. Nowadays, Prometheus is a completely community-driven project hosted by the Cloud Native Computing Foundation. The Prometheus project includes a collection of client libraries which allow metrics to be published so they can then be collected (or "scraped" using Prometheus' terminology) by the metrics server.

Prometheus metrics libraries have become widely adopted, not only by Prometheus users, but by other monitoring systems including InfluxDB, OpenTSDB, Graphite, and Sysdig Monitor. Many CNCF projects expose out-of-the-box metrics using the Prometheus metrics format. You'll also find them in core Kubernetes components like the API server, etcd, CoreDNS, and more.

The Prometheus metrics format is so widely adopted that it became an independent project, OpenMetrics, striving to make this metric format specification an industry standard. Sysdig Monitor supports this format out-of-the-box, and it will dynamically detect and scrape Prometheus metrics for you. If you're new to custom metrics, you can start from the beginning in the blog post How to instrument code: Custom Metrics vs APM vs OpenTracing to understand why you need custom application metrics, which use cases they cover, and how they compare with other observability options.

Prometheus metrics: dot-metrics vs tagged metrics

Before describing the Prometheus metrics / OpenMetrics format in particular, let's take a broader look at the two main paradigms used to represent a metric: dot notation and multi-dimensional tagged metrics. Let's start with dot-notated metrics. In essence, everything you need to know about the metric is contained within the name of the metric. For example:

```
production.server5.pod50.html.request.total
production.server5.pod50.html.request.error
```

These metrics provide the detail and the hierarchy needed to effectively utilize your metrics. In order to make it fast and easy to use your metrics, this model of metrics exposition suggests that if you'd like a different aggregation, then you should calculate that metric up front and store it using a separate name. So, with our example above, suppose you were

interested in the “requests” metrics across the entire service. You might rearrange our metrics to look like this:

```
production.service-nginx.html.request.total
production.service-nginx.html.request.error
```

You could imagine many more combinations of metrics that you might need. On the other hand, the Prometheus metric format takes a flat approach to naming metrics. Instead of a hierarchical, dot separated name, you have a name combined with a series of labels or tags:

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

A time series with the metric name *http_requests_total* and the labels *service=”service”*, *server=”pod50”* and *env=”production”* could be written like this:

```
http_requests_total{service=”service”, server=”pod50”,
env=”production”}
```

Highly dimensional data basically means that you can associate any number of context-specific labels to every metric you submit.

Imagine a typical metric like *http_requests_per_second*, every one of your web servers is emitting these metrics. You can then bundle the labels (or dimensions):

- Web Server software (Nginx, Apache)
- Environment (production, staging)
- HTTP method (POST, GET)
- Error code (404, 503)
- HTTP response code (number)
- Endpoint (/webapp1, /webapp2)
- Datacenter zone (east, west)

And voila! Now you have N-dimensional data and can easily derive the following data graphs:

- Total number of requests per web server pod in production
- Number of HTTP errors using the Apache server for webapp2 in staging
- Slowest POST requests segmented by endpoint URL

Prometheus metrics / OpenMetrics format

Prometheus metrics 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.

A metric is composed by several fields:

- Metric name
- Any number of labels (can be 0), represented as a key-value array
- Current metric value
- Optional metric timestamp

A Prometheus metric can be as simple as:

```
http_requests 2
```

Or, it can include all of the mentioned components:

```
http_requests_total{method="post",code="400"}  3   1395066363000
```

Metric output is typically preceded with # HELP and # TYPE metadata lines. The HELP string identifies the metric name and a brief description of it. The TYPE string identifies the type of metric. If there's no TYPE before a metric, the metric is set to untyped. Everything else that starts with a # is parsed as a comment.

```
# HELP metric_name Description of the metric
# TYPE metric_name type
# Comment that's not parsed by prometheus
http_requests_total{method="post",code="400"}  3   1395066363000
```

Prometheus metrics / OpenMetrics represents multi-dimensional data using labels or tags as you saw in the previous section:

```
traefik_entrypoint_request_duration_seconds_
count{code="404",entrypoint="traefik",method="GET",protocol="http"}
44
```

The key advantage of this notation is that all of these dimensional labels can be used by the metric consumer to dynamically perform metric aggregation, scoping, and segmentation. Using these labels and metadata to slice and dice your metrics is an absolute requirement when working with Kubernetes and microservices.

Prometheus metrics client libraries

The Prometheus project maintains four official Prometheus metrics libraries written in Go, Java / Scala, Python, and Ruby. The Prometheus community has created many third-party libraries that you can use to instrument other languages (or just alternative implementations for the same language):

- Bash
- C++
- Common Lisp
- Elixir
- Erlang
- Haskell
- Lua for Nginx
- Lua for Tarantool
- .NET / C#
- Node.js
- Perl
- PHP
- Rust

Full list of code instrumentation libraries here.

Prometheus metrics / OpenMetrics types

Depending on what kind of information you want to collect and expose, you'll have to use a different metric type. Here are your four choices available on the OpenMetrics specification:

1. Counter

This represents a cumulative metric that only increases over time, like the number of requests to an endpoint. Note: instead of using Counter to instrument decreasing values, use Gauges.

```
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated,
even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 3.7156890216e+10
```

2. Gauge

Gauges are instantaneous measurements of a value. They can be arbitrary values which will be recorded. Gauges represent a random value that can increase and decrease randomly such as the load of your system.

```
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 73
```

3. 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 **"http_request_duration_seconds"** exposes multiple time series during a scrape:

```
# HELP http_request_duration_seconds request duration histogram
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{le="0.5"} 0
http_request_duration_seconds_bucket{le="1"} 1
http_request_duration_seconds_bucket{le="2"} 2
http_request_duration_seconds_bucket{le="3"} 3
http_request_duration_seconds_bucket{le="5"} 3
http_request_duration_seconds_bucket{le="+Inf"} 3
http_request_duration_seconds_sum 6
http_request_duration_seconds_count 3
```

4. 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.

Here is a summary with a base metric name of **"go_gc_duration_seconds"** also exposes multiple time series during a scrape:

```
# HELP go_gc_duration_seconds A summary of the GC invocation
durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 3.291e-05
go_gc_duration_seconds{quantile="0.25"} 4.3849e-05
go_gc_duration_seconds{quantile="0.5"} 6.2452e-05
go_gc_duration_seconds{quantile="0.75"} 9.8154e-05
go_gc_duration_seconds{quantile="1"} 0.011689149
go_gc_duration_seconds_sum 3.451780079
go_gc_duration_seconds_count 13118
```

Instrumenting your applications

Golang code instrumentation with Prometheus metrics / OpenMetrics

To demonstrate Prometheus metrics code instrumentation in Golang, we're going to use the official Prometheus library to instrument a simple application. You just need to create and register your metrics and update their values. Prometheus will handle the math behind the summaries and expose the metrics to your HTTP endpoint.

```
package main

import (
 "net/http"
 "github.com/prometheus/client_golang/prometheus"
 "github.com/prometheus/client_golang/prometheus/promhttp"
 "log"
 "time"
 "math/rand"
)

var (
 counter = prometheus.NewCounter(
    prometheus.CounterOpts{
       Namespace: "golang",
       Name:      "my_counter",
       Help:      "This is my counter",
    })
 gauge = prometheus.NewGauge(
    prometheus.GaugeOpts{
       Namespace: "golang",
       Name:      "my_gauge",
       Help:      "This is my gauge",
    })
 histogram = prometheus.NewHistogram(
    prometheus.HistogramOpts{
       Namespace: "golang",
       Name:      "my_histogram",
       Help:      "This is my histogram",
    })
 summary = prometheus.NewSummary(
    prometheus.SummaryOpts{
       Namespace: "golang",
       Name:      "my_summary",
       Help:      "This is my summary",
    })
)
```

```go
func main() {
 rand.Seed(time.Now().Unix())
 http.Handle("/metrics", promhttp.Handler())
 prometheus.MustRegister(counter)
 prometheus.MustRegister(gauge)
 prometheus.MustRegister(histogram)
 prometheus.MustRegister(summary)
 go func() {
    for {
       counter.Add(rand.Float64() * 5)
       gauge.Add(rand.Float64()*15 - 5)
       histogram.Observe(rand.Float64() * 10)
       summary.Observe(rand.Float64() * 10)
       time.Sleep(time.Second)
    }
 }()

 log.Fatal(http.ListenAndServe(":8080", nil))
}
```

view rawprometheus-metrics-golang.go hosted with ❤ by GitHub

This is how these Golang Prometheus metrics look using a Sysdig Monitor dashboard when scraped over a few minutes:

Try it in Docker

To make things easier and because we just love containers, you can directly run this example using Docker:

```
$ git clone https://github.com/sysdiglabs/custom-metrics-examples
$ docker build custom-metrics-examples/prometheus/golang -t
prometheus-golang
$ docker run -d --rm --name prometheus-golang -p 8080:8080
prometheus-golang
```

Check your metrics live:

```
$ curl localhost:8080/metrics
```

Java code instrumentation with Prometheus metrics / OpenMetrics

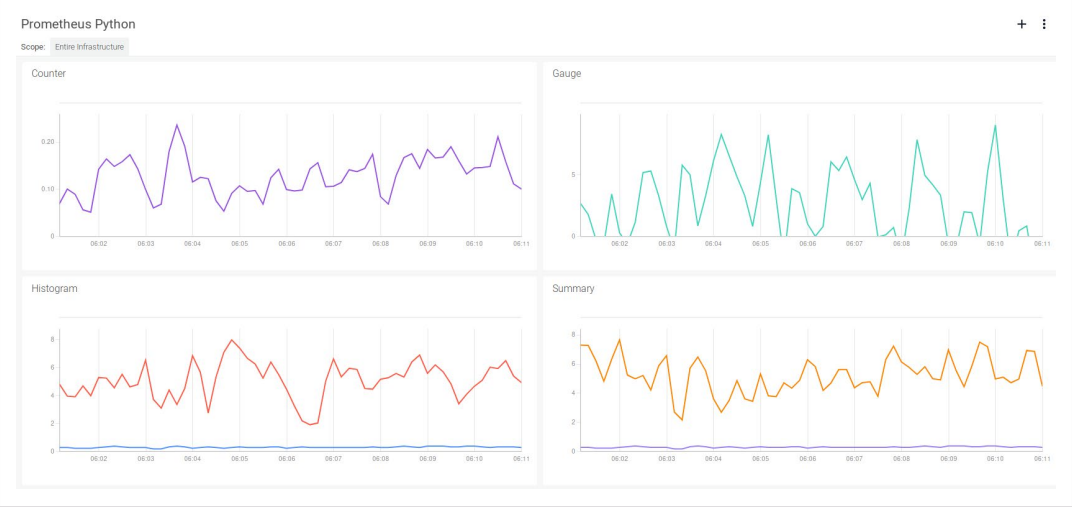
Using the official Java client library, we created this small example of Java code instrumentation with Prometheus metrics:

```
import io.prometheus.client.Counter;
import io.prometheus.client.Gauge;
import io.prometheus.client.Histogram;
import io.prometheus.client.Summary;
import io.prometheus.client.exporter.HTTPServer;
import java.io.IOException;
import java.util.Random;

public class Main {

  private static double rand(double min, double max) {
      return min + (Math.random() * (max - min));
  }

  public static void main(String[] args) {
      Counter counter = Counter.build().namespace("java").name("my_
counter").help("This is my counter").register();
      Gauge gauge = Gauge.build().namespace("java").name("my_gauge").
help("This is my gauge").register();
      Histogram histogram = Histogram.build().namespace("java").
name("my_histogram").help("This is my histogram").register();
      Summary summary = Summary.build().namespace("java").name("my_
summary").help("This is my summary").register();
```

```
        Thread bgThread = new Thread(() -> {
            while (true) {
                try {
                    counter.inc(rand(0, 5));
                    gauge.set(rand(-5, 10));
                    histogram.observe(rand(0, 5));
                    summary.observe(rand(0, 5));
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        bgThread.start();

        try {

            HTTPServer server = new HTTPServer(8080);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

view rawprometheus-metrics-java.java hosted with ❤ by GitHub

This is how the Java Prometheus metrics look using a Sysdig Monitor dashboard:



Try it in Docker

Download, build, and run (make sure you have ports 8080 and 80 free in your host or change the redirected port):

```
$ git clone https://github.com/sysdiglabs/custom-metrics-examples
$ docker build custom-metrics-examples/prometheus/java -t prometheus-
java
$ docker run -d --rm --name prometheus-java -p 8080:8080 -p 80:80
prometheus-java
```

Check that the endpoint is working:

```
$ curl -v localhost
* Rebuilt URL to: localhost/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 80 (#0)
> GET / HTTP/1.1
> Host: localhost
> User-Agent: curl/7.61.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Mon, 08 Oct 2018 13:17:07 GMT
< Transfer-encoding: chunked
<
* Connection #0 to host localhost left intact
```

Check the local metrics endpoint:

```
$ curl localhost:8080
```

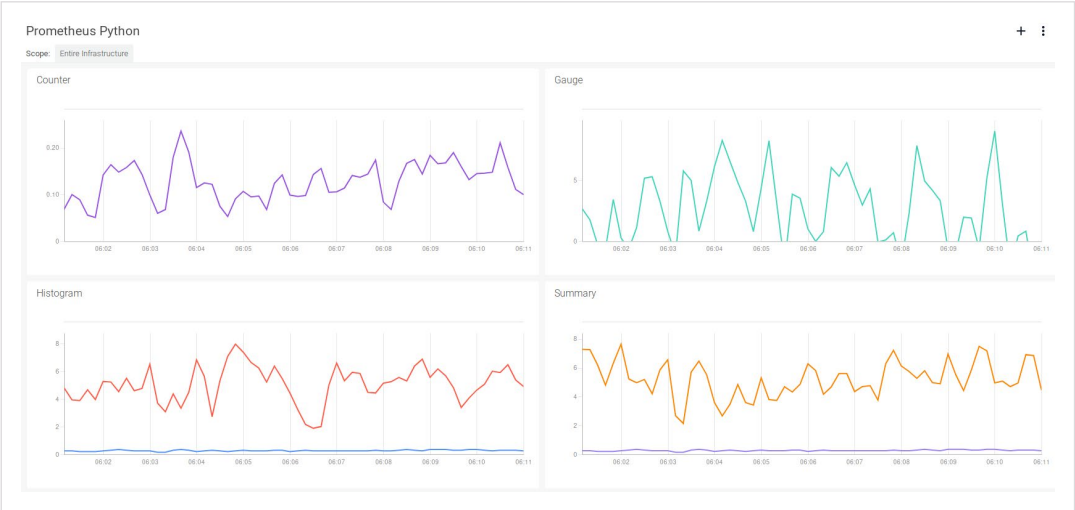
Python code instrumentation with Prometheus metrics / OpenMetrics

This example uses the same application as the previous example, but this time it's written in Python using the official Python client library:

```
import prometheus_client as prom
import random
import time

req_summary = prom.Summary('python_my_req_example', 'Time spent
processing a request')

@req_summary.time()
def process_request(t):
  time.sleep(t)
if __name__ == '__main__':

  counter = prom.Counter('python_my_counter', 'This is my counter')
  gauge = prom.Gauge('python_my_gauge', 'This is my gauge')
  histogram = prom.Histogram('python_my_histogram', 'This is my
histogram')
  summary = prom.Summary('python_my_summary', 'This is my summary')
  prom.start_http_server(8080)

  while True:
      counter.inc(random.random())
      gauge.set(random.random() * 15 - 5)
      histogram.observe(random.random() * 10)
      summary.observe(random.random() * 10)
      process_request(random.random() * 5)

      time.sleep(1)
```

view rawprometheus-metrics-python.py hosted with ❤ by GitHub

This is how Python Prometheus metrics look in the Sysdig Monitor dashboard:



Try it in Docker

Download, build, and run (make sure you have ports 8080 and 80 free in your host or change the redirected port):

```
$ git clone https://github.com/sysdiglabs/custom-metrics-examples
$ docker build custom-metrics-examples/prometheus/python -t
prometheus-python
$ docker run -d --rm --name prometheus-python -p 8080:8080 -p 80:80
prometheus-python
```

Check the local metrics endpoint:

```
$ curl -v localhost
* Rebuilt URL to: localhost/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 80 (#0)
> GET / HTTP/1.1
> Host: localhost
> User-Agent: curl/7.61.1
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: text/html; charset=utf-8
< Content-Length: 2
< Server: Werkzeug/0.14.1 Python/3.7.0
< Date: Mon, 08 Oct 2018 13:21:54 GMT
<
* Closing connection 0
```

Check the local metrics endpoint:

```
$ curl localhost:8080
```

NodeJS / Javascript code instrumentation with Prometheus OpenMetrics

This last example uses the same app again written in Javascript and running in Node.js. We're using an unofficial client library that can be installed via npm: npm i prom-client:

```
const client = require('prom-client');
const express = require('express');
const server = express();
const register = new client.Registry();

// Probe every 5th second.
const intervalCollector = client.collectDefaultMetrics({prefix:
'node_', timeout: 5000, register});

const counter = new client.Counter({
  name: "node_my_counter",
  help: "This is my counter"
});

const gauge = new client.Gauge({
  name: "node_my_gauge",
  help: "This is my gauge"
});

const histogram = new client.Histogram({
  name: "node_my_histogram",
  help: "This is my histogram",
  buckets: [0.1, 5, 15, 50, 100, 500]
});

const summary = new client.Summary({
  name: "node_my_summary",
  help: "This is my summary",
  percentiles: [0.01, 0.05, 0.5, 0.9, 0.95, 0.99, 0.999]
});

register.registerMetric(counter);
register.registerMetric(gauge);
register.registerMetric(histogram);
register.registerMetric(summary);
```

```
const rand = (low, high) => Math.random() * (high - low) + low;

setInterval(() => {
  counter.inc(rand(0, 1));
  gauge.set(rand(0, 15));
  histogram.observe(rand(0,10));
  summary.observe(rand(0, 10));
}, 1000);

server.get('/metrics', (req, res) => {
  res.set('Content-Type', register.contentType);
  res.end(register.metrics());
});

console.log('Server listening to 8080, metrics exposed on /metrics
endpoint');
server.listen(8080);
view rawprometheus-metrics-javascript.js hosted with ❤ by GitHub
```

This is how Node.js/Javascript Prometheus metrics will look using a Sysdig Monitor dashboard:

Try it in Docker

Download, build, and run (make sure you have port 8080 free in your host or change the redirected port):

```
$ git clone https://github.com/sysdiglabs/custom-metrics-examples
$ docker build custom-metrics-examples/prometheus/javascript -t
prometheus-node
$ docker run -d --rm --name prometheus-node -p 8080:8080 prometheus-node
```

Check the metrics endpoint:

```
$ curl localhost:8080/metrics
```

Prometheus metrics and Sysdig Monitor

Using Sysdig Monitor, you can automatically scrape any of the Prometheus metrics exposed by your containers or pods. Following the Prometheus autodiscovery labeling protocol, the Sysdig agent will look for the following annotations:

- *prometheus.io/scrape: "true"* (adds this container to the list of entities to scrape)
- *prometheus.io/port: "endpoint-TCP-port"* (defaults to 8080)
- *prometheus.io/path: "/endpoint-url"* (defaults to /metrics)

Using the standard Prometheus notation has the advantage of having to annotate your containers or pods only once, whether you want to use a Prometheus server, a Sysdig Monitor agent, or both. To dig deeper into the details about our customizable configurations, visit our Sysdig agent configuration for Prometheus metrics support page.

Monitoring Prometheus metrics with Sysdig Monitor

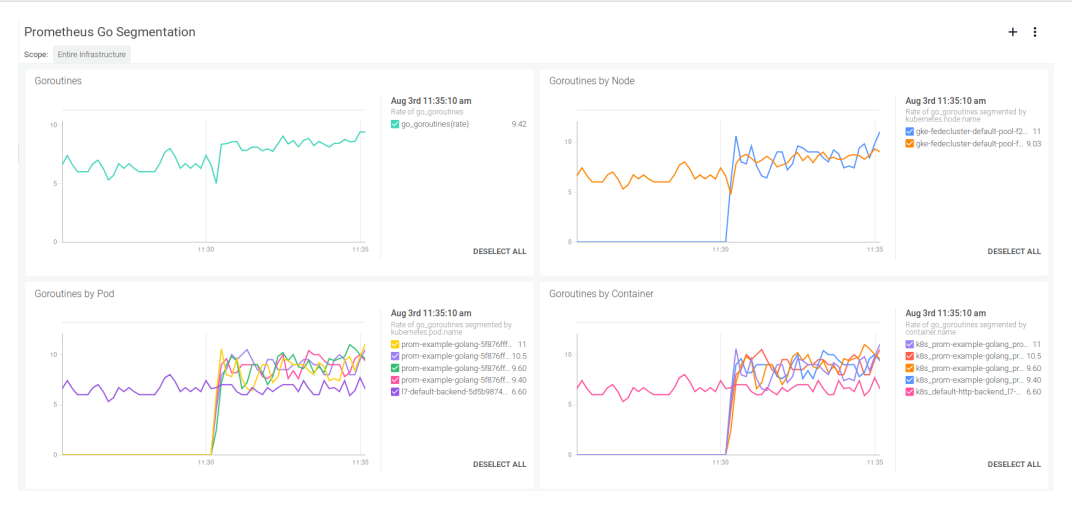
Another immediate advantage of using the Sysdig agent to collect Prometheus metrics is that the resulting metrics will not only include the labels added in the Prometheus metrics, but also the full Docker container and Kubernetes metadata. Developers won't need to add those labels manually. For example:

- Host, process, container runtime labels like *proc.name="nginx"*
- Kubernetes metadata (namespace, deployment, pod, etc) like *kubernetes.namespace.name="frontend"*
- Cloud provider metadata (region, AZ, securityGroups, etc) like *cloudProvider.region="us-east-1"*

These extra dimensions are extremely valuable when monitoring microservices / container oriented environments. These dimensions are even more valuable if you are using an orchestrator like Kubernetes or OpenShift. Out-of-the-box and without any further labeling effort from your developers, you can do things like:

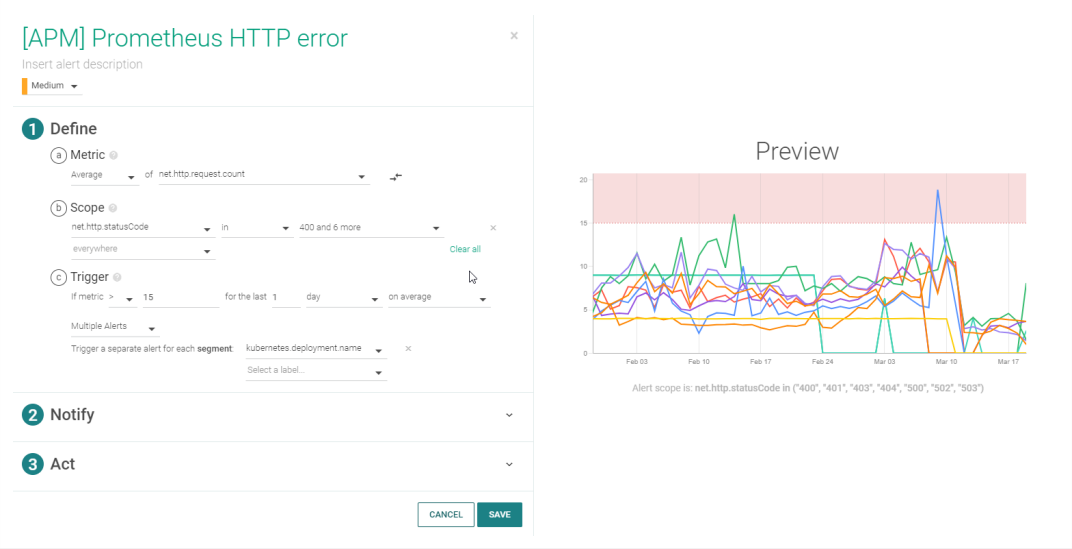
- Measure the overall performance of a service for your entire infrastructure, regardless of the physical hosts or pods, just aggregating the metrics by service label.
- Comparing the number of errors of every deployment in your Kubernetes cluster using a single graph separated in different lines (segmented by deployment label).
- Create a multi-dimensional table with the different HTTP response codes and their frequency (columns), by service (rows).

Here's what Prometheus metrics exported by the Go app and segmented by Kubernetes pod look like with Sysdig:



Alerting on Prometheus metrics with Sysdig Monitor

On top of that, you can also use Prometheus metrics with Sysdig Monitor to configure alerts and notifications. For example using the metric net.http.request.count(base metric), setting the scope using the label net.http.statusCode(Aggregate only for error values like 4xx or 5xx), and segmenting by the kubernetes.deployment.name label (that's why you can see a different line per deployment):



Prometheus metrics for Golden Signals monitoring

As we mentioned earlier in this document, these are the four most important metrics to monitor any microservices application:

- Latency or response time
- Traffic or connections
- Errors
- Saturation

When you instrument your code using Prometheus metrics, one of the first things you'll want to do is to expose these metrics from your app. Prometheus libraries empower you to easily expose the first three: latency or response time, traffic or connections, and errors.

Exposing HTTP metrics can be as easy as importing the instrumenting library, like promhttp in our previous Golang example:

```
import (
...
  "github.com/prometheus/client_golang/prometheus/promhttp"
...
)
...
  http.Handle("/metrics", promhttp.Handler())
...
```

Sysdig Monitor makes this process even easier by automatically discovering and collecting these application performance metrics without any instrumentation required, Prometheus or otherwise. The Sysdig agent decodes any known protocol directly from the kernel system calls and translates this huge amount of low-level information into high-level metrics, giving you a degree of visibility that's typically only found on APMs. Although Prometheus instrumentation can provide more accurate metrics, this is great when you don't have the possibility of instrumenting code, like in legacy apps or when the issues to troubleshoot are already happening.

Here is what your Golden Signals monitoring may look like with Sysdig Monitor — including a topology map of your microservices application:

Lessons learned

1. Prometheus metrics / OpenMetrics facilitate a clean and mostly frictionless interface between developers and operators, making code instrumentation easy and standardized.
2. Libraries already exist for the most popular languages and more are being developed by the community. In addition, the labeling feature makes it a great choice for instrumenting custom metrics if you plan to use containers and microservices.
3. By using Sysdig Monitor on top of applications instrumented with Prometheus, you will go one step further, automatically enhancing all of the collected metrics with container, orchestrator, and cloud provider metadata, as well as enabling golden signal metrics and dashboards without additional instrumentation.

Challenges using Prometheus at scale

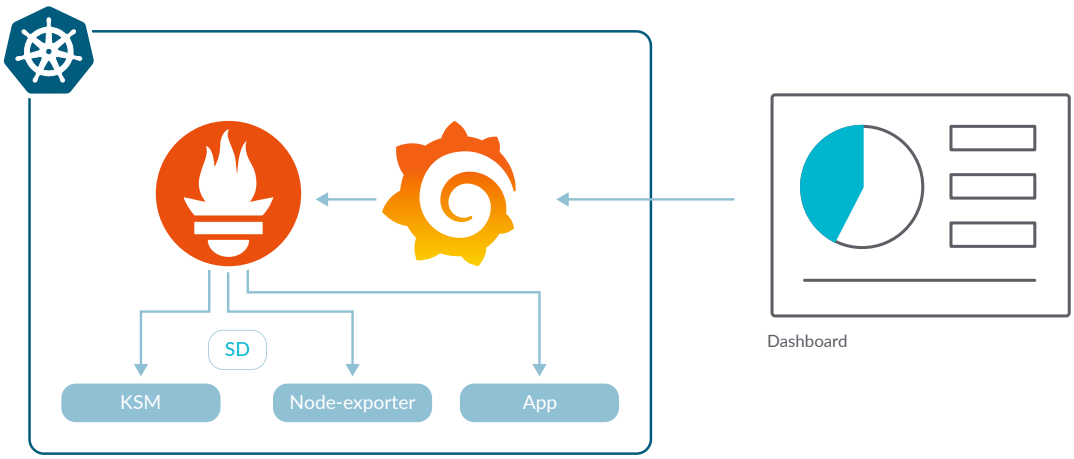
Prometheus first steps

Often, Prometheus comes hand in hand with the first Kubernetes cluster, which is during the development stage. After deploying some apps, you may realize that you need insight into the cluster and your old host-based toolset doesn't work. After exhaustive research (first Google result), you discover Prometheus.

Prometheus is easy to begin with; after deploying the server, some basic exporters, and Grafana, you can get metrics to start building dashboards and alerts.

Then, you dig deeper into Prometheus and discover some very good features:

- Service discovery to make configuration easier.
- Hundreds of exporters built by the open-source community.
- Instrumentation libraries to get custom metrics from your application.
- It is OSS, under the CNCF umbrella, being the second project to graduate after Kubernetes.
- Kubernetes is already instrumented and exposes Prometheus metrics in all of its services.

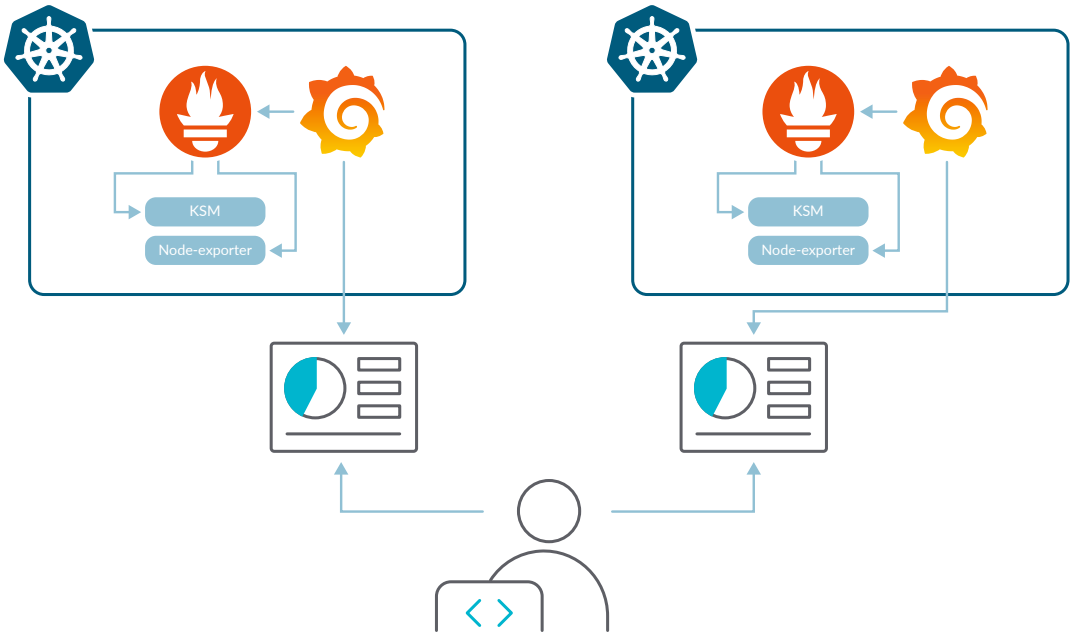


At this point, everything looks great. Kubernetes is your best friend, your apps scale smoothly, and you can watch many metrics from your fancy dashboard. You're ready for the next step.

Moving Prometheus into production

Your second Kubernetes cluster is here and you follow the same process; Prometheus, exporters, and Grafana are swiftly deployed. The staging cluster is usually bigger than development, but Prometheus seems to cope with it well. There is only a small set of applications, and the number of dashboards and users is low so migration is easy too.

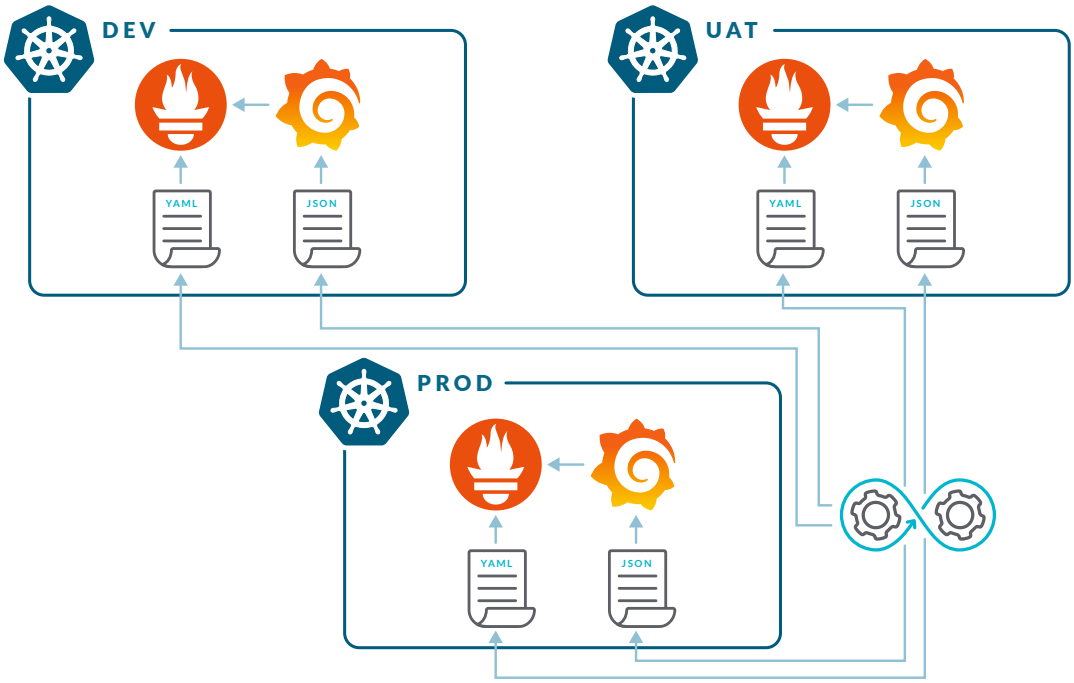
The first little problem arises. Now, you have two different Grafana installations and need to switch between them to see metrics from development and staging. At this moment, developers aren't too worried about this, but it's starting to itch. Everything checks out and production release is good to go.



There it is. Your third cluster. Follow the same process again; deploy Prometheus, exporters. and Grafana, and migrate dashboards and users. Users aren't happy about having a third dashboard portal, but it's not a show stopper.

Results are so good that the company decides to migrate more applications to Kubernetes. New users, configurations, dashboards, and alerts have to be added in each cluster one by one. Managing configuration in a consistent way starts to require some effort and organization.

As the applications grow in numbers, new requirements arise, and new clusters are created in order to host different applications, serve different regions, and increase availability. Keeping different Prometheus and Grafana instances for each one becomes a challenge, and DevOps and development teams start to complain.



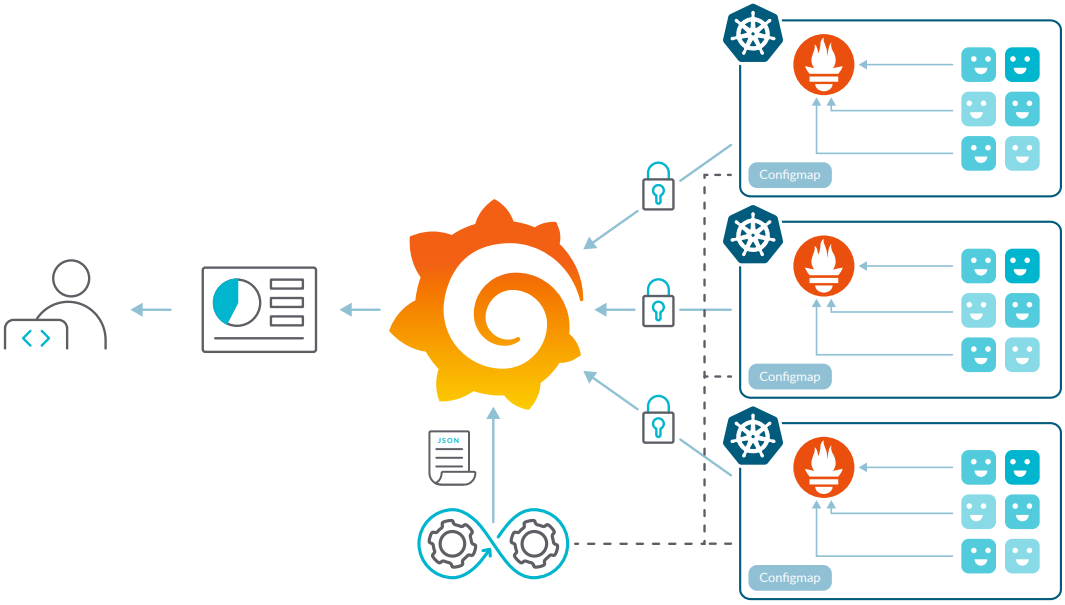
Keeping global visibility

The distributed model presents different challenges:

- It doesn't allow global visibility. Several production clusters would make this a new requirement.
- Operations can be cumbersome and time consuming.
- This model can complicate governance and regulatory compliance.

In addition to making access to different clusters easier, a centralized visualization tool allows you to visualize and correlate information from services across multiple clusters in the same dashboard.

At first glance, this can seem to be easy: You deploy a centralized Grafana and add the different Prometheus servers running in the clusters as *datasources*.



This approach has some hidden challenges:

- Security is not a feature included in Prometheus. As long as the communication between Prometheus and Grafana is intracluster, this isn't an issue. However, as you get Grafana out of the cluster, you need to implement something on top of Prometheus to secure the connection and control access to the metrics. There are many solutions to this issue but they require some effort of implementation and maintenance (manage certificates, create ingress controllers, configure security in Grafana, etc.).
- If the clusters are dynamic or change, you often need to implement a way to automatically add data sources to Grafana every time you deploy a Prometheus in a cluster.
- This allows us to mix in dashboard panels from different sources, but you still can't query services across different clusters and perform really global queries.
- Controlling who is accessing what data becomes more important, and an RBAC system may be required. Integration with identity services is most likely necessary in order to keep the user base updated, and this might become a compliance requirement.

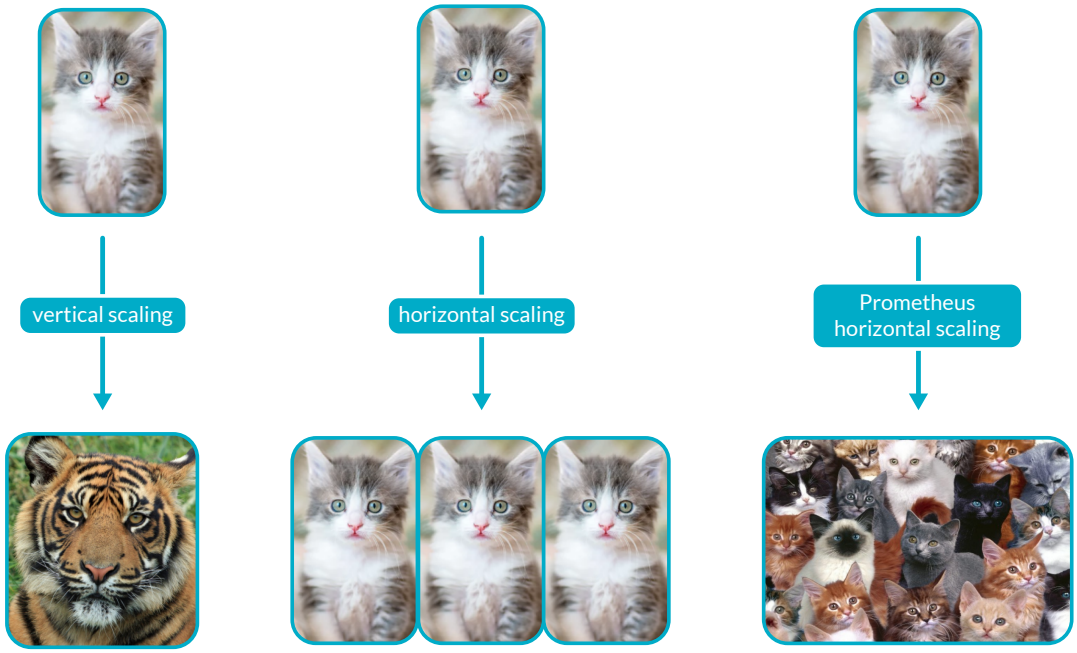
All of these problems aren't blockers, but they require an effort of architecture design, development, and implementation. Maintenance of this entire structure also requires significant resources.

Prometheus horizontal scale

With some effort from your team, you have a decent centralized system and everything seems perfect, until a new problem arises. Developers understand the power of custom metrics with business information, and instrument the code to obtain them. While your organization grows, the number of services in Kubernetes increase and metrics usage rises. Clusters are upscaled and the services have more and more replicas.

Your Prometheus servers start dying and with some research, you clearly see a memory problem. Memory usage in Prometheus is directly proportional to the number of time series stored, and as your timeseries grow in numbers, you start to have OOM kills. You raise the resource quota limits but you can't do this ad infinitum. You will eventually reach the memory capacity of a node, and no pod can go beyond that. A Prometheus with millions of metrics can use more than 100GB of RAM, and that can be an issue running in Kubernetes. You must scale out to absorb capacity, but the short answer is that you can't. Prometheus isn't designed to be scaled horizontally. Once you hit the limit of vertical scaling, you're done.

There are some workarounds for this issue, like sharding different metrics throughout several Prometheus servers, but it's a tricky process. It adds complexity to the setup and can make troubleshooting difficult.



Prometheus Data exporting

There are several methods to use when exporting data from Prometheus that could help with some of the issues we have seen in this guide:

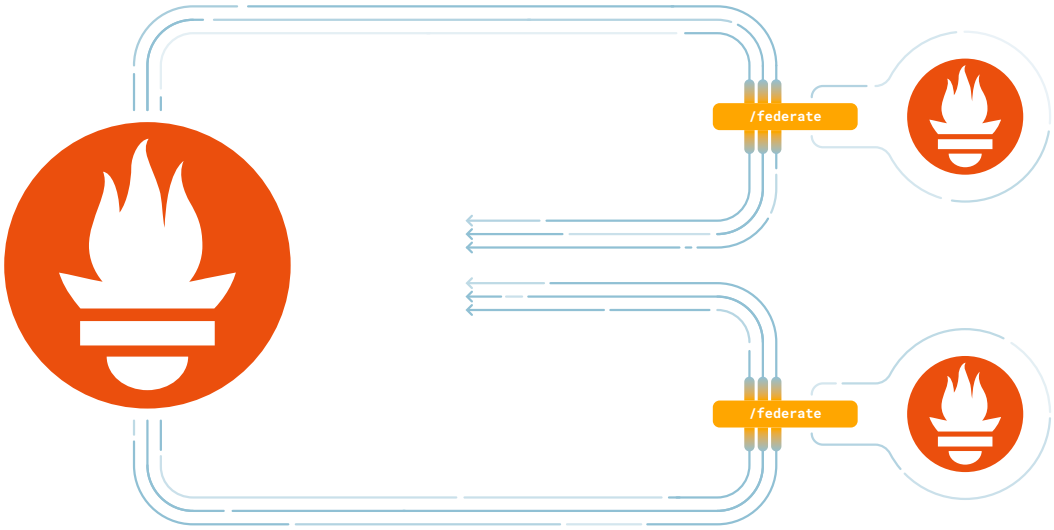
Federation

In this case, Prometheus exposes the information in an endpoint where other higher level Prometheus servers can pull the metrics then consolidate and aggregate the data.

This method is quite simple but has some limitations:

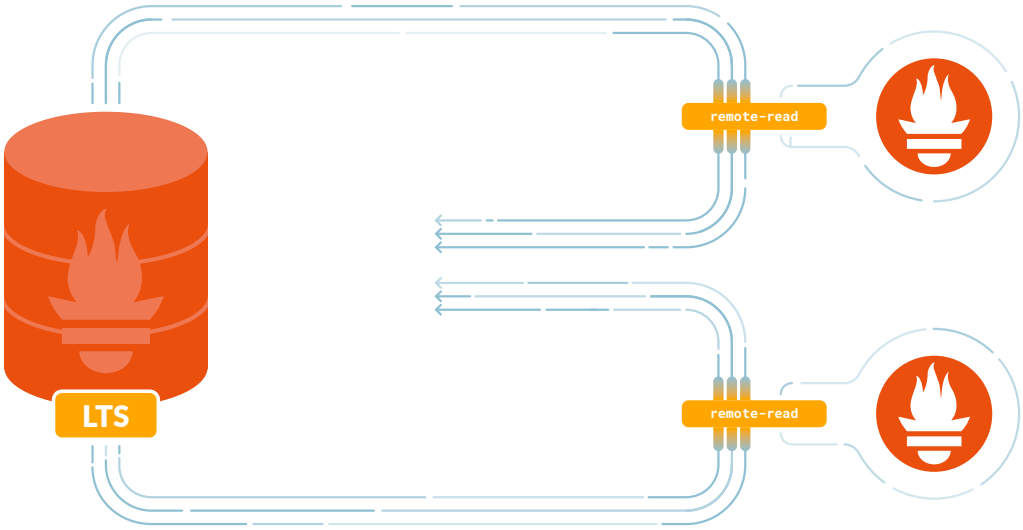
- The amount of data that can be sent is limited as the scale problem is even worse. You are receiving all the data in one Prometheus instance.
- The aggregations must be done first in Prometheus itself, then the requests for the federation endpoint need filtering and explicit information.
- Maintenance can be tricky as you need to configure all the Prometheus servers and keep an updated list as your environment grows.

As you can see, federation can be a good tool in some situations but won't address most of the scale issues.



Remote read

This case is similar to federation in the sense of architecture as the metrics are pulled from other Prometheus servers, but is designed to be more efficient and interoperable. Remote read improves the amount of data that can be pulled and aggregated since they are defined in the original Prometheus instance. That frees the target system from doing that part of the job.



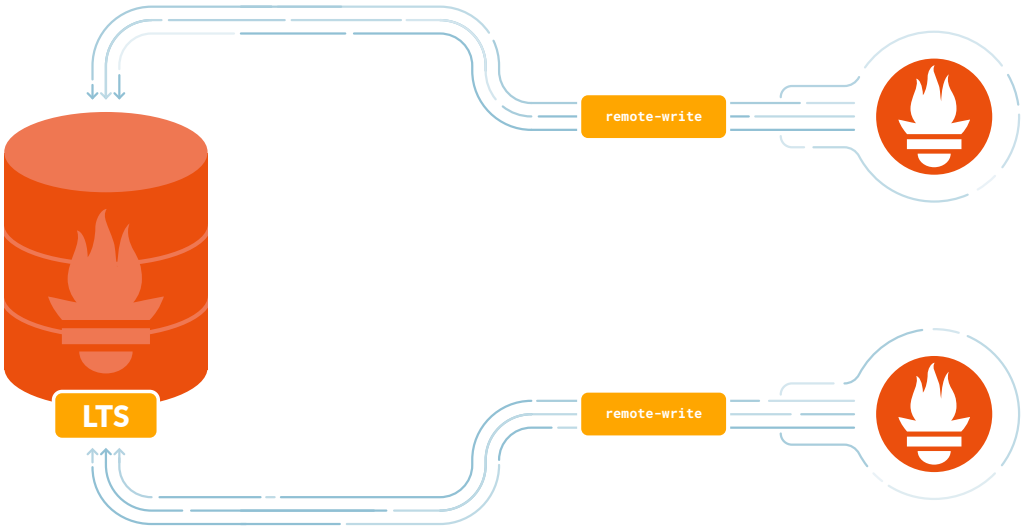
The main problem of the remote read is networking and security. The long term storage or the target system that will pull the metrics needs access to the Prometheus endpoints and that means you will need to deploy some security. In a simple environment this is not hard to do, but in a multi-cloud and/or multi-region environment, this can be a real challenge.

Remote write

It is the most used method to export metrics from Prometheus to other systems. It is especially well suited for Long Term Storage (LTS) as it allows you to push metrics to a single endpoint.

This method has several advantages:

- The metrics to be sent are configured in the Prometheus instance, so every team or operator can choose which metrics are being sent to the LTS and the level of aggregation.
- Configuration is easy and remote_write allows you to send different sets of metrics to different endpoints.
- The networking is easy to configure as only the remote_write endpoint of the LTS instance needs to be open to the public (or available to the other Prometheus server at least).
- It has some security features built in that make the security part much easier:
- TLS communications
- Authentication: basic, token, headers, etc
- The throughput of the solution is high so a lot of metrics can be sent, depending on the LTS capacity.
- The protocol has been implemented in a lot of different applications. this makes the remote_write a really valuable tool to interoperate with other systems (kafka or other queue systems for buffering, transformation, enrichment, etc)



Most of the common LTS systems rely mostly on remote_write to receive metrics from multiple Prometheus instances at the same time.

An example of remote_write configuration would be:

```
remote_write:
- url: “https://mylts.com/remote_write”
  authorization:
    credentials: “g4ghj5ikuwhxf9ñojw0wfegwrtg2”
    type: “Bearer”
  tls_config:
    ca_file: /etc/pki/myca.crt
```

Check the list of different integrations available for remote read and remote write in the Prometheus docs.

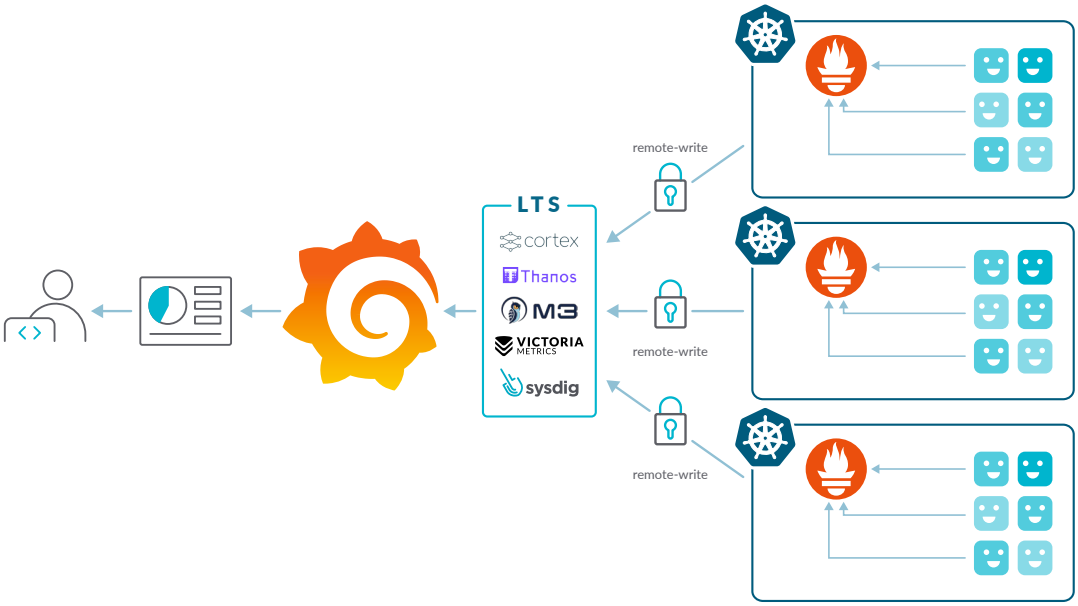
Sysdig Monitor is also fully compatible with Prometheus remote_write. You can offload long term storage of metric to Sysdig in the same way described above. This way Sysdig can handle all the infrastructure and you can take advantage of:

- A fully managed Prometheus service
- Form Query user interface and simplified PromQL queries
- A Monitoring integrations manager based on Prometheus exporters
- Built-in Prometheus alerting best practices
- eBPF based troubleshooting data
- Enterprise access controls
- SOC II compliant SaaS platform

Challenges with long-term storage

Many people have faced these same issues before. Given that Prometheus claims it's not a metrics storage, the expected outcome was that somebody would eventually create that long-term storage for Prometheus metrics.

Currently, there are several open-source projects to provide long term storage (LTS). These community projects are ahead of the rest: Cortex, Thanos, and M3.



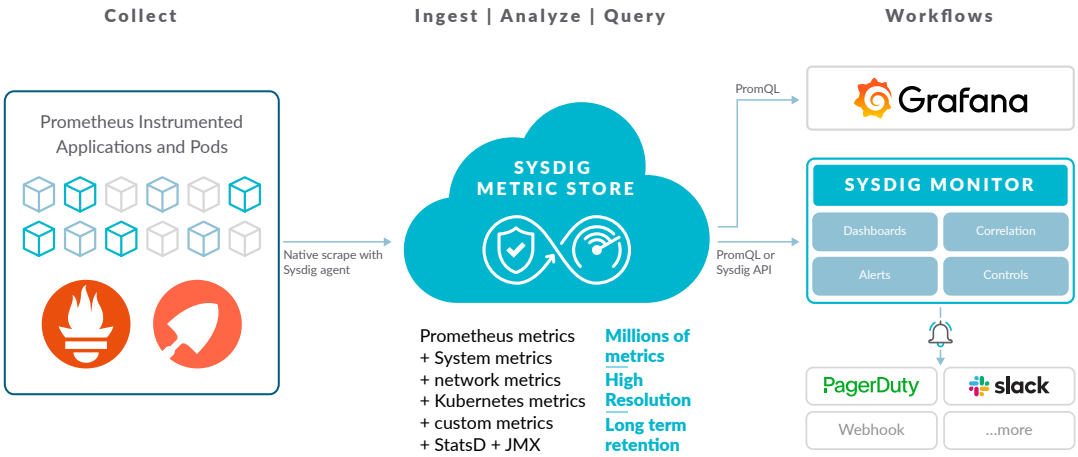
These services are not free of complications, however:

- Architecture is complex and can require a lot of effort to deploy, optimize, and maintain the monitoring self-run service.
- Operation costs can be high. You can leverage managed databases like DynamoDB, but you still need to scale the rest of the monitoring services and the metrics throughout can be very high.
- These projects are still in early stages of development and can be tricky to run in production, even with a dedicated team.

What's next?

Prometheus has been a game changer in the monitoring landscape for cloud-native applications, like Kubernetes has been to container orchestration. However, even large tech companies find the challenges of scaling Prometheus in a DIY fashion daunting. Every full-time equivalent resource that is dedicated to maintaining this approach is one less developer that could be writing innovative code to transform your business.

To be an early adopter and explore new fields is exciting, but sometimes it's nice when you can just pay for a good service and avoid a path full of pain.



To try to relieve the pain that many of our customers are feeling, Sysdig has been working steadily to evolve our platform. We're now **fully compatible** with Prometheus while simultaneously improving our ability to scale to millions of time series and retain data for longer periods. With Sysdig, you can now take advantage of a supported platform that delivers a scalable, secure solution for Prometheus monitoring. At the same time, we're bringing that same joy of visibility to developers when they first discover what Prometheus could do.

Lessons learned

1. Prometheus is easy to initiate, but doesn't scale well for a high number of metrics or for long-term storage. It can add unwanted complexity when you grow to the point that a centralized solution is desired.
2. Using open source federation projects like Cortex or Thanos can help, but they require maintenance and resource management. They also will not help with enterprise requirements like access controls and 24/7 support.
3. A fully Prometheus compatible commercial offering like Sysdig Monitor can be used to solve the issues that often arise with a DIY approach.

Conclusion

In this guide we have presented:

- Why Prometheus monitoring is important
- How to install Prometheus
- How to use exporters for third-party applications
- How to monitor your own applications
- How to scale Prometheus

We hope that you found this information useful as you navigate the best way to monitor your Kubernetes workloads with Prometheus. Still, this can be a complex journey and Prometheus monitoring can take years to master and be difficult to scale. As you have seen in some of the examples in this guide, Sysdig tries to take some of the maintenance burden out of your way with our community site called PromCat.io. On PromCat.io, you will find:

- **Prometheus and third-party exporters,** packaged as container images with deployment manifests for Kubernetes.
- Both **Grafana and Sysdig dashboards.** Most of our customers love Sysdig integrated dashboards, with Team scope and RBAC, which are available alongside our troubleshooting and security functionality. But some advanced users prefer Grafana with Sysdig metrics so they can customize every detail of their dashboarding experience. The catalog supports both.
- Both Prometheus alert rules and Sysdig PromQL **alert definition.**
- **Recording rules,** to pre-calculate metrics when you have tons of them.

The Sysdig Secure DevOps Platform is fully compatible with Prometheus, including PromQL. It can be used to scale your Prometheus Monitoring while providing access to Prometheus metrics to new users that may not be familiar with PromQL yet. Sysdig uses the same data to monitor and secure, so you can correlate system activity with Kubernetes services. This enables you to identify where a problem occurred and why it happened — and you can use this single source of truth to investigate and troubleshoot performance and security issues. If you are also interested in learning how to secure your Kubernetes environments, we have written a companion guide called the Kubernetes Security Guide with detailed information about image scanning, control plane security, RBAC for Kubernetes, and runtime security.

With both of these guides in hand, you will be well on your way to understanding how to both monitor **and** secure your Kubernetes environments!

Find out how the Sysdig Secure DevOps Platform can help you and your teams confidently run cloud-native apps in production. Contact us for additional details about the platform, or to arrange a personalized demo.

www.sysdig.com