Prometheus Monitoring Guide

sysdig

Table of Contents

01 **About This Guide** 4

02 **Intro to Prometheus Monitoring** 5

- Prometheus Metric Structure 5
- PromQL 6
- Lessons Learned 7

03 **Installing Prometheus** 8

04 **Prometheus Exporters** 11

- Example of Exporter Installation 13

05 **Monitoring Applications with Prometheus** 15

- Prometheus metrics: dot-metrics vs. tagged metrics 16
- Prometheus metrics / OpenMetrics format 17
- Prometheus metrics client libraries 19
- Prometheus metrics / OpenMetrics types 19
- Instrumenting your applications 21
  - Golang code instrumentation with Prometheus metrics / OpenMetrics 21
  - Java code instrumentation with Prometheus metrics / OpenMetrics 26
  - Python code instrumentation with Prometheus metrics / OpenMetrics 29
  - NodeJS / Javascript code instrumentation with Prometheus OpenMetrics 31
  - OpenTelemetry 33

Getting the most out of Prometheus Metrics with Sysdig Monitor 37
Monitoring Prometheus metrics with Sysdig Monitor 38
Alerting on Prometheus metrics with Sysdig Monitor 39
Prometheus metrics for Golden Signals monitoring 40
Troubleshooting issues with Prometheus metrics 42
Exploring your Prometheus metrics 42
Lessons learned 43

06 **Challenges Using Prometheus at Scale** **44**

Prometheus first steps 44
Moving Prometheus into production 45
Keeping global visibility 47
Prometheus horizontal scale 48
Prometheus Data exporting 50
Federation 50
Remote read 51
Remote write 52
Challenges with long-term storage 54
Prometheus metrics cardinality 55
Prometheus metrics costs 57
Scale and simplify Prometheus Monitoring with Sysdig Monitor 57
Addressing Prometheus scalability, availability, and performance challenges 58
Addressing long-term storage challenges 58
Addressing global visibility challenges 59
Addressing Prometheus data exporting challenges 59
Addressing Prometheus metrics cardinality challenges 59
Addressing Prometheus metrics costs challenges 60
Addressing Prometheus exporters challenges 60

07 **Conclusion** **61**

01

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 and its challenges
6. How Sysdig Monitor can help

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.

02

Intro to Prometheus Monitoring

Prometheus is an open source monitoring tool. This means that it mainly stores time series (metrics) of numerical values. Each of these time series are identified by a metric name and a set of key-value pairs (labels), which makes it filter through its query language called PromQL.

But before digging deeper into the Prometheus features that made it the de-facto monitoring standard, let's talk about what Prometheus is not. Prometheus isn't 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 you 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. Using Prometheus that way can lead to serious performance and availability issues.

Now that you know 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 reasons why Prometheus has become so popular is how easy it is to deploy in a Kubernetes cluster. As you'll see in the next section, there are several ways to deploy Prometheus; it's easy and you'll be up and running in minutes. Also, with some annotation in your pods (some of them come already by default), Prometheus – thanks to its service discovery mechanism, – can automatically discover metrics and start to scrape them from the first minute. This autodiscovery service 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, as opposed to dot-separated metrics, is the use of labels to define the time series. The Prometheus approach makes the metrics totally horizontal and flexible, providing a multi-dimensional data model which eliminates 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.The more values for a label, and the more labels in a metric, the more cardinality. Then, cardinality is the number of combinations of labels and values for a metric, and every combination is a new time series metric. Cardinality explosion is a serious problem that may be hard to tackle. Stability, availability, and performance of your observability platform may be in danger due to a high volume of metrics. Costs are also something to take into account with cardinality explosion.

Let's take a look at what a Prometheus metric actually looks like. Prometheus gathers metrics from application endpoints. These applications expose their own 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. You'll find this information included as part of the metrics output. 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 any additional 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 on your own.

PromQL

It's easy to filter labels and fetch data 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 so popular.

You can use the PromQL language for 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 the keys that allows SRE teams to fine tune the alerts to get to that sweet spot of alerting when it is really needed, minimizing the false positive 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. For example, 100 errors doesn't necessarily mean a big problem, depending on the volume of requests the service is handling. It's not the same to serve 500 instead of 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, comparing the 4xx and 5xx response codes with the total volume of responses:

```
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, you 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 observability project with Prometheus. These and other interesting topics will be covered in the following sections.

As discussed in this short introduction, it's easy to see why Prometheus has become the default choice when it comes to monitoring cloud infrastructures, Kubernetes clusters, or applications. In the following chapters, we will go deeper into the details that you need to know if you want to have a Prometheus system up and running reliably in production.

Lessons Learned

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

03

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-2.44.0.darwin-amd64$ ./prometheus
./prometheus
ts=2023-06-05T09:34:12.564Z caller=main.go:531 level=info msg="No time or size
retention was set so using the default time retention" duration=15d
[…]
ts=2023-06-05T09:34:12.637Z caller=main.go:1257 level=info msg="Completed
loading of configuration file" filename=prometheus.yml
totalDuration=58.336584ms db_storage=2.875µs remote_storage=1.375µs web_
handler=458ns query_engine=1.875µs scrape=55.602ms scrape_sd=103.125µs
notify=39.333µs notify_sd=38.584µs rules=13.708µs tracing=733.792µs
ts=2023-06-05T09:34:12.638Z caller=main.go:1001 level=info 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, one of the easiest ways to install Prometheus in Kubernetes is using Helm.

The Prometheus community maintains a Helm chart that makes the Prometheus installation and configuration task really easy. It also deploys and configures other applications that are part of the Prometheus observability ecosystem. To install Prometheus in your Kubernetes cluster with the community Helm chart, run the following commands:

First, 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
```

Once you added the Prometheus Helm repo to your config, you can proceed with the Prometheus installation:

```
# 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 running 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
```

The community Helm chart also deploys node-exporter, kube-state-metrics, and alertmanager along with Prometheus. You will be able to start monitoring your Kubernetes cluster and nodes 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 that manages, configures, and updates other deployments according to its high-level service specifications.

04

Prometheus Exporters

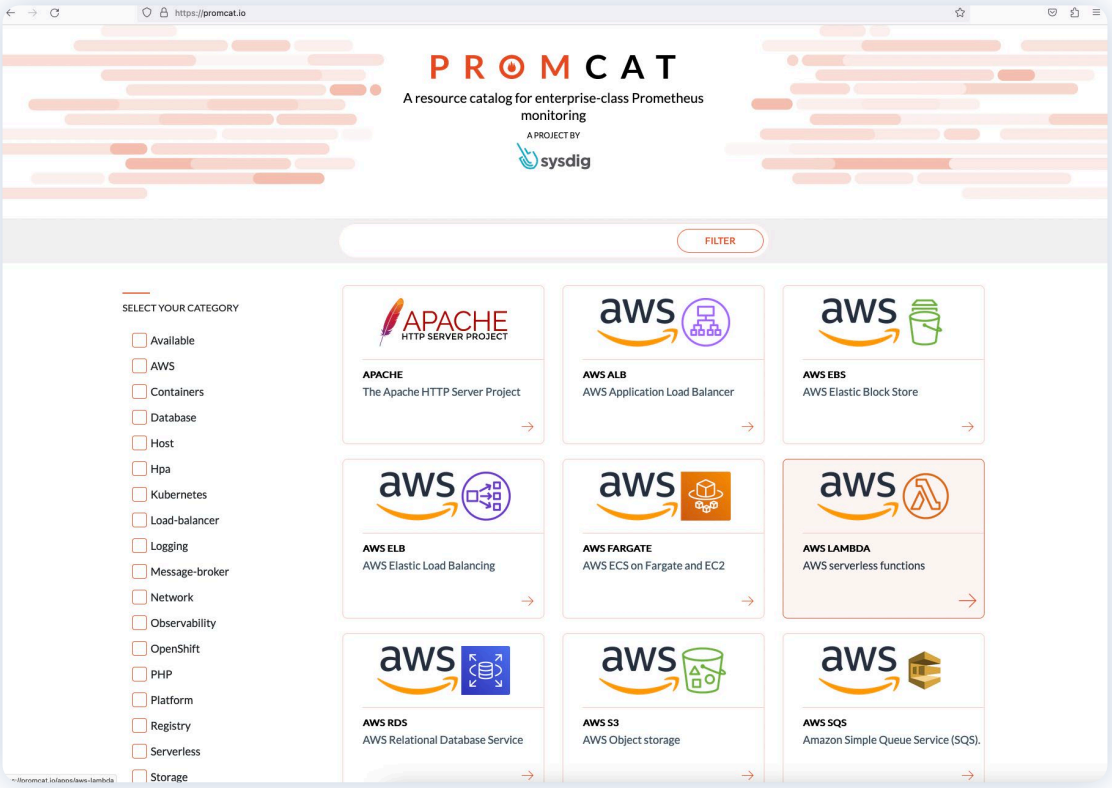
Although some services and applications are already adopting Prometheus metrics format and provide endpoints for this purpose, many popular server applications, like Nginx or PostgreSQL, have been around much longer than Prometheus metrics / OpenMetrics. Some applications may have their own metrics formats 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-publish 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 application that is being monitored, or isolated in their own pod or even running in a different infrastructure. Once the Prometheus exporter is deployed and running, you can collect the service metrics by scraping the exporter that exposes those converted metrics into Prometheus format.

There are hundreds of Prometheus exporters available on the internet. Every exporter is not necessarily like each other. On the contrary, it is designed solely for its monitoring purpose based on the application architecture and design to monitor. 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. Also, sometimes the application needs some tuning or special configuration to allow the exporter to fetch the data and generate metrics.

It is pretty common to see more than one exporter for the same application. Independent of whether they offer different features or not, these exporters may be forked from others, discontinued projects, or even different exporters for different application versions. It's important to correctly identify the application that you want to monitor, the metrics that you need, and the proper exporter that best fits your needs.



To reduce the maintenance burden needed to find, validate, and configure these exporters, Sysdig has created a site called PromCat.io. Here, Sysdig curates the best exporters, provides detailed configuration examples, and supports 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, you can 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 following 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" }
      ]
   }
)
```

Set a username and password of your own choice.

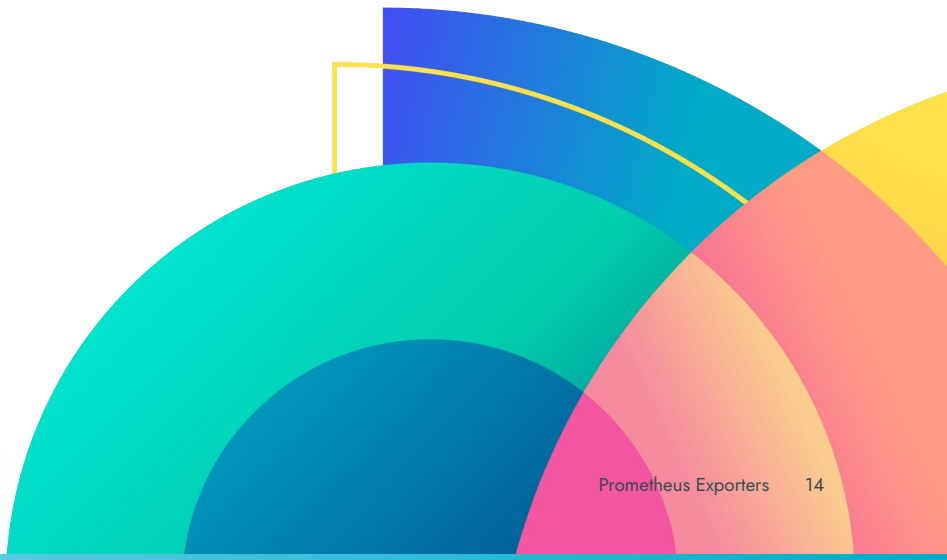
The metrics will be available in the port 9216 of the exporter pod.

Then, install the exporter in your K8s cluster using the following 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:

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



05

Monitoring Applications with Prometheus

In the following example-driven tutorial, you will learn how to use Prometheus metrics / OpenMetrics to instrument your code for multiple programming languages, like 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 open source project includes a collection of client libraries which allow metrics publication for applications, so they can then be collected (or "scraped" using Prometheus' terminology) by the Prometheus metrics server.

Prometheus metrics libraries are widely adopted nowadays, 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 Prometheus metrics in the core Kubernetes components, like the API server, etcd, CoreDNS, kube-controller-manager, Kubelet, 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.

Along with Prometheus, Grafana is one of the OSS projects most used by the community when it comes to visualizing data. Practitioners can import dashboards already built from other community members, or can create their own from scratch. In Sysdig Monitor, there are plenty of out-of-the-box dashboards for third-party software. These are available from the very beginning, and will take your burden off in most cases. It also allows customers to create their own custom dashboards, either through a Form UI based experience or with PromQL queries for most experienced users. In this section, you'll find how data is visualized with Sysdig Monitor dashboards.

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 its name. 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 them. When it comes to metrics aggregation, this model of metrics exposition helps with a faster and easier adoption in exchange for calculating that metric up front and storing 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 your 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 pushing data into it. 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, Ruby, and Rust. 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). The full list of code instrumentation libraries is 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**" which 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

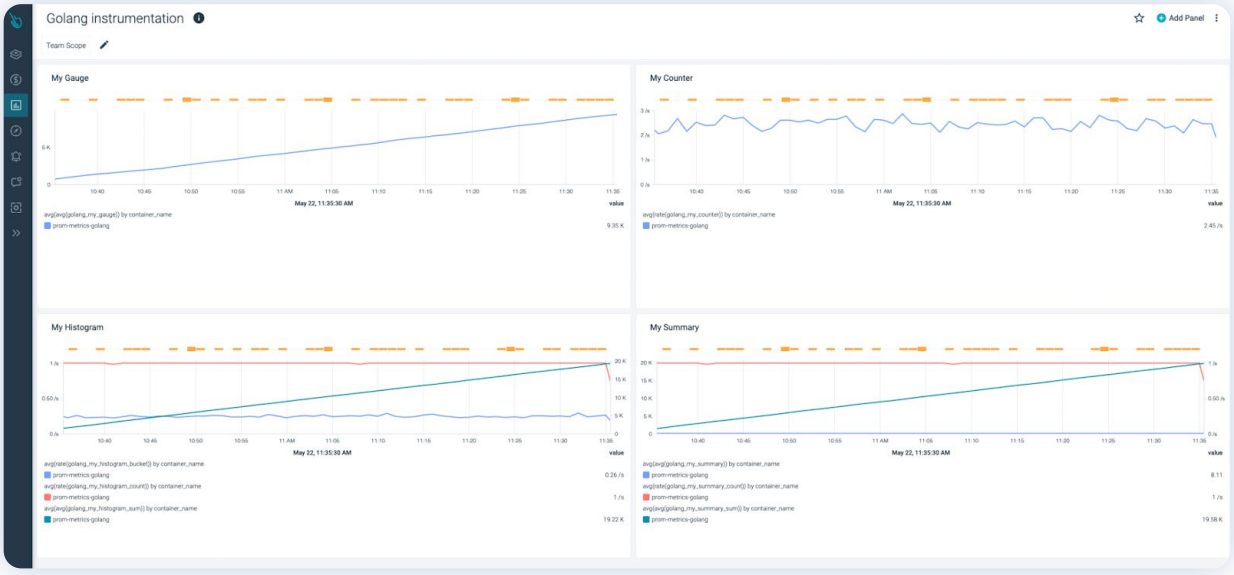
To demonstrate Prometheus metrics code instrumentation in Golang, we're going to use the official Prometheus library to instrument a simple application. You just need to create and register your metrics and update their values. Prometheus will handle the math behind the summaries and expose the metrics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


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 raw prometheus-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:



Deploy this Golang application in Kubernetes

Here, you'll learn how to deploy this Golang application for testing purposes. It's already instrumented and ready to expose Prometheus metrics.

First, clone the repo and go to "prometheus/golang" application path:

```
$ git clone https://github.com/sysdiglabs/custom-metrics-examples
$ cd custom-metrics-examples/prometheus/golang
```

Add this new "Makefile" to the current path:

```
$ cat Makefile
VERSION=latest
all:
   go build -o prometheus-metrics-golang
   podman build -t sysdiglabs/prom-metrics-golang:$(VERSION) .
```

Run “make” to build your Golang application:

```
$ make
go build -o prometheus-metrics-golang
go: downloading github.com/prometheus/client_golang v0.8.0
go: downloading github.com/prometheus/common v0.0.0-20180518154759-7600349dcfe1
go: downloading github.com/prometheus/client_model
v0.0.0-20180712105110-5c3871d89910
go: downloading github.com/golang/protobuf v1.1.0
go: downloading github.com/beorn7/perks v0.0.0-20180321164747-3a771d992973
go: downloading github.com/prometheus/procfs v0.0.0-20180705121852-ae68e2d4c00f
$ podman build -t sysdiglabs/prom-metrics-golang:latest .
[1/2] STEP 1/5: FROM golang:alpine AS builder
… (output truncated)
[2/2] STEP 4/5: EXPOSE 8080
--> 7ec4614c4ac
[2/2] STEP 5/5: CMD [“/main”]
[2/2] COMMIT sysdiglabs/prom-metrics-golang:latest
--> f82023ab740
Successfully tagged localhost/sysdiglabs/prom-metrics-golang:latest
f82023ab740fddd2d22fc038415dc5bf3774d255377dbee77c4bd3731533fd8b
```

Tag and push the new image to your registry:

```
$ podman tag localhost/sysdiglabs/prom-metrics-golang:latest
docker.io/vhernando/prom-metrics-golang:latest
$ podman push docker.io/vhernando/prom-metrics-golang:latest
Getting image source signatures
Copying blob 0534b11b2654 done
Copying blob a68e3cc56821 done
Copying blob 9c87f2223d79 skipped: already exists
Copying config f82023ab74 done
Writing manifest to image destination
Storing signatures
```

Try out your application in Kubernetes by creating a new Pod object:

```
$ cat pod-prom-metrics-golang.yaml
apiVersion: v1
kind: Pod
metadata:
  name: prom-metrics-golang
  labels:
    app: prom-metrics-golang
spec:
  containers:
  - name: prom-metrics-golang
    image: docker.io/vhernando/prom-metrics-golang:latest
    ports:
    - containerPort: 8080

$ kubectl create namespace prom-metrics
$ kubectl create -f pod-prom-metrics-golang.yaml -n prom-metrics
pod/prom-metrics-golang created
$ kubectl get pods -n prom-metrics
NAME                 READY   STATUS    RESTARTS   AGE
prom-metrics-golang   1/1     Running   0          20m
$ kubectl create svc clusterip prom-metrics-golang --tcp=8080:8080 -n prom-metrics
service/prom-metrics-golang created
$ kubectl get svc -n prom-metrics
NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
prom-metrics-golang   ClusterIP   10.96.213.125   <none>        8080/TCP   5s
$ kubectl get ep -n prom-metrics
NAME                 ENDPOINTS             AGE
prom-metrics-golang   192.169.203.59:8080   18s
$ kubectl port-forward svc/prom-metrics-golang 8080:8080 -n prom-metrics
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
Handling connection for 8080

$ curl http://localhost:8080/metrics
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0
go_gc_duration_seconds{quantile="0.25"} 0
go_gc_duration_seconds{quantile="0.5"} 0
go_gc_duration_seconds{quantile="0.75"} 0
go_gc_duration_seconds{quantile="1"} 0
go_gc_duration_seconds_sum 0
go_gc_duration_seconds_count 0
(output truncated)
```

Try it yourself with Podman

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

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

Check your metrics live:

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

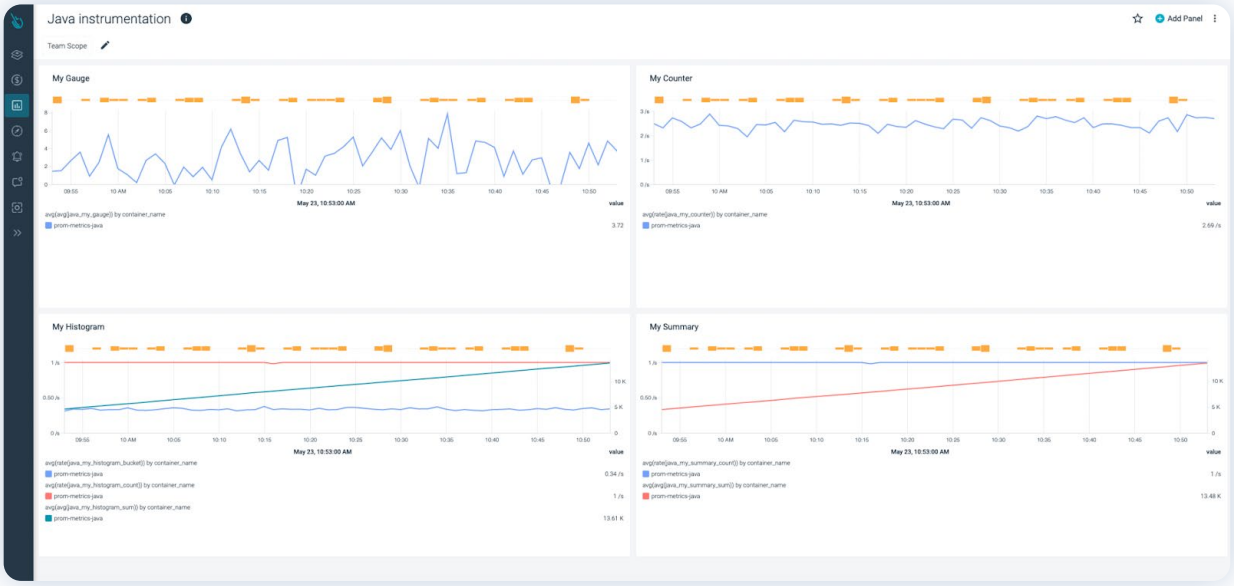
Java code instrumentation with Prometheus metrics / OpenMetrics

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 raw prometheus-metrics-java.java hosted with ❤ by GitHub
```

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



Try it yourself with Podman

Download, build, and run:

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

Check the local metrics endpoint:

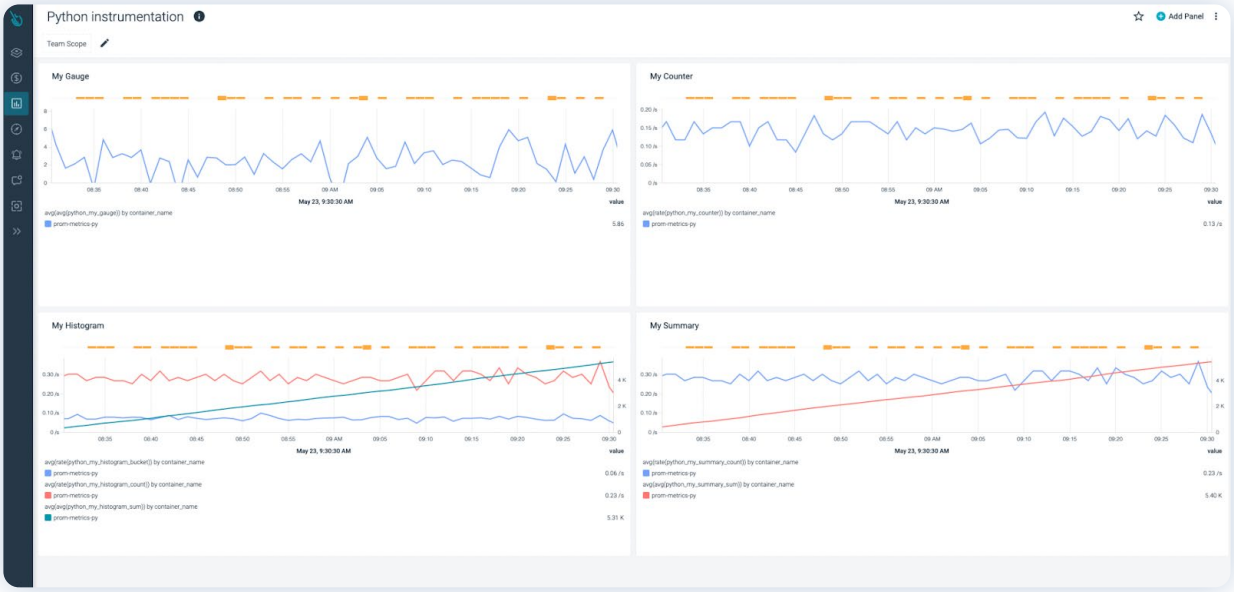
```
$ curl localhost:8080
```

Python code instrumentation with Prometheus metrics / OpenMetrics

This example uses the same application as the previous one, 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 raw prometheus-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:

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

Check the local metrics endpoint:

```
$ curl localhost:8080
```

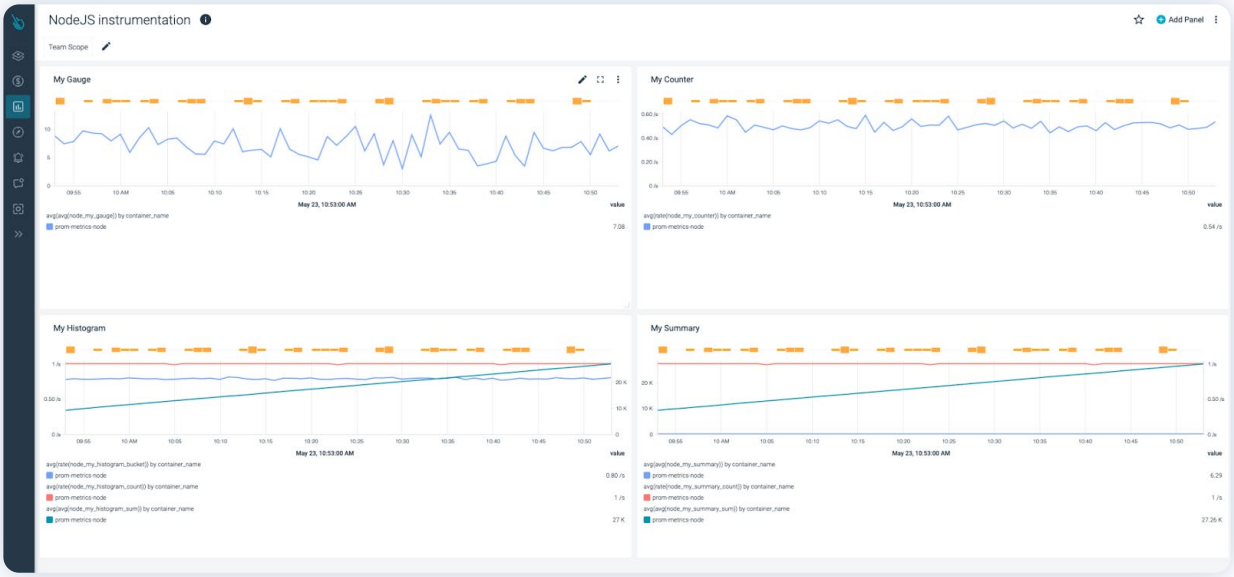
NodeJS / Javascript code instrumentation with Prometheus OpenMetrics

This last example uses the same app 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 raw prometheus-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
$ podman build custom-metrics-examples/prometheus/javascript -t prometheus-node
$ podman run -d --rm --name prometheus-node -p 8080:8080 prometheus-node
```

Check the metrics endpoint:

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

OpenTelemetry

OpenTelemetry, or most commonly known as OTel, is an open source project incubated by the CNCF. OTel aims to provide a set of standardized vendor-agnostic SDKs, APIs, and tools for ingesting, transforming, and sending data to a observability backend.

One of the main advantages of using OTel is there is no need to instrument your applications to get your metrics. OTel can automatically pull telemetry data without any instrumentation in your application, and push it to your observability backend.

In short, with OTel you can either manually instrument your application by coding against the OTel APIs, or rely on the automatic instrumentation which leverages an underlying mechanism that will add the OTel, SDK, and API capabilities to your application.

In this section, we'll explore how to use OTel automatic instrumentation to start pulling metrics from your container.

First off, deploy the OpenTelemetry operator in your Kubernetes cluster. You can use a Helm chart for your ease. Ensure that cert-manager is installed first.

```
$ helm repo add open-telemetry
https://open-telemetry.github.io/opentelemetry-helm-charts
$ helm repo update
$ helm install --namespace opentelemetry \
            opentelemetry-operator open-telemetry/opentelemetry-operator
```

Once the operator Pod is up and running, it's time to start creating a few objects that are needed for automatic application instrumentation, and to ingest and process metrics as well.

```
$ kubectl apply -f - <<EOF
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: demo
  namespace: opentelemetry
spec:
  mode: daemonset
  config: |
    receivers:
      otlp:
        protocols:
          grpc:
          http:
    processors:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 75
        spike_limit_percentage: 15
      batch:
        send_batch_size: 10000
        timeout: 10s
    exporters:
      logging:
      prometheus:
        enable_open_metrics: false
        endpoint: 0.0.0.0:9464
        resource_to_telemetry_conversion:
          enabled: true
    service:
      pipelines:
        traces:
          receivers: [otlp]
          processors: [memory_limiter, batch]
          exporters: [logging]
        metrics:
          receivers: [otlp]
          processors: [memory_limiter, batch]
          exporters: [prometheus]
```

```
        logs:
          receivers: [otlp]
          processors: [memory_limiter, batch]
          exporters: [logging]
EOF
$ kubectl apply -f - <<EOF
apiVersion: opentelemetry.io/v1alpha1
kind: Instrumentation
metadata:
  name: demo-instrumentation
  namespace: opentelemetry
spec:
  exporter:
    endpoint: http://demo-collector.opentelemetry.svc.cluster.local:4317
  propagators:
    - tracecontext
    - baggage
  sampler:
    type: parentbased_traceidratio
    argument: “1”
EOF
```

It’s time to deploy your application and apply the required OpenTelemetry annotations. The annotations required may vary depending on the programming language used. In this example, we deploy the following Java application.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-sagar
  namespace: opentelemetry
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo-sagar
  template:
    metadata:
      labels:
        app: demo-sagar
      annotations:
        instrumentation.opentelemetry.io/inject-java: “true”
        instrumentation.opentelemetry.io/container-names: “spring”
    spec:
      containers:
      - name: spring
        image: sagar27/petclinic-demo
        ports:
        - containerPort: 8080
```

Once the application is deployed, the OpenTelemetry instrumentation CRD injects its SDK to auto instrument this Java application. If you want to start scraping your metrics from a Prometheus instance, you can rely on its autodiscover mechanism. Or, if you prefer a push model, you can either send them out with remote write.

Getting the most out of Prometheus Metrics with Sysdig Monitor

Sysdig Monitor is an observability solution that provides insights into your cloud and Kubernetes workloads to simplify monitoring and reduce costs. Following are some of the benefits of using Sysdig Monitor:

- Its Prometheus compatible SaaS solution with long-term storage allows customers to reduce the complexity of their DIY Prometheus environments and take their burden off. It is also more cost-effective than competition.
- You get immediate granular details and remediation steps to troubleshoot and resolve complex issues in rapidly changing container environments.
- Reduce your wasted spending with Kubernetes and cloud costs visibility.
- Out-of-the-box-dashboards and alerts help Sysdig customers with observability of their Kubernetes and cloud environments.
- Multi-cloud observability platform. Get visibility of your workloads and services running on the main cloud providers (AWS, Google cloud, Azure).
- Sysdig Agent leverages eBPF to get internal Kernel data from nodes and enriches metrics with Kernel insights. Your metrics gain Kubernetes and cloud context.
- Collect all the custom metrics you need for business and applications KPIs at a much lower cost.

When it comes to pull metrics, the Sysdig Agent is responsible for such a task. It has a Prometheus lightweight instance embedded, so you can **automatically** scrape any of the Prometheus metrics exposed by your containers or pods the same way you'd do with Prometheus. 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

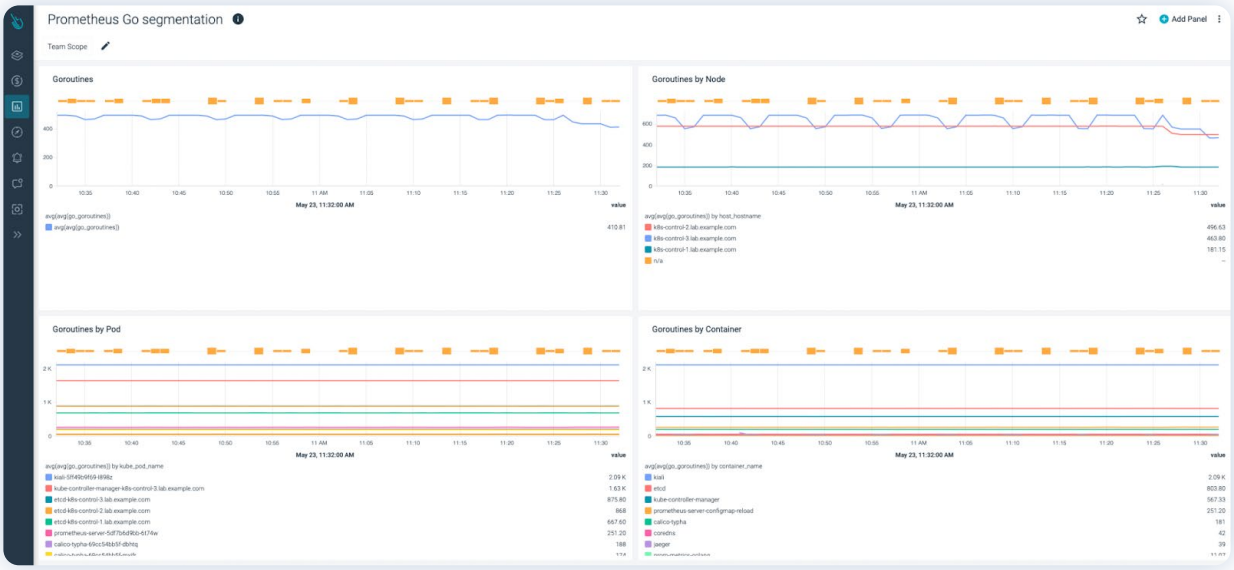
Another immediate advantage of using the Sysdig Agent to collect Prometheus metrics is that the resulting metrics will not only include the labels added in the Prometheus metrics, but also the container and Kubernetes metadata. That's what we know as **metrics enrichment.** Developers won't need to add those labels manually. For example:

- Host, process, and 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. Since they are out of the box and without any further labeling effort from your developers, you can do things like:

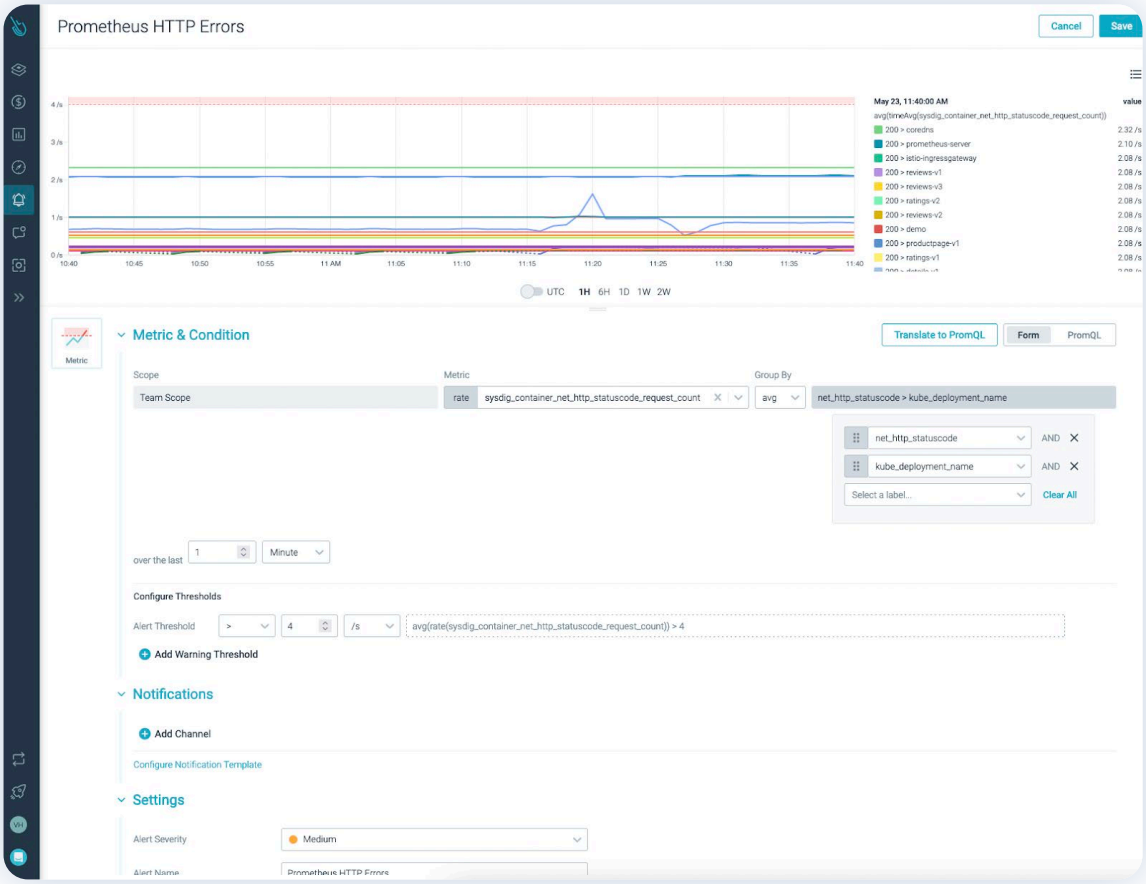
- Measure the overall performance of a service for your entire infrastructure, regardless of the physical hosts or pods, just aggregating the metrics by service label.
- Comparing the number of errors of every deployment in your Kubernetes cluster using a single graph separated in different lines (segmented by deployment label).
- Create a multi-dimensional table with the different HTTP response codes and their frequency (columns), by service (rows).
- Troubleshoot complex issues using extended labels. With Sysdig Monitor, you can dig deeper into Kubernetes and cloud metadata, and even get details of the processes running within a container.

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 sysdig_container_net_http_statuscode_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 kube_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

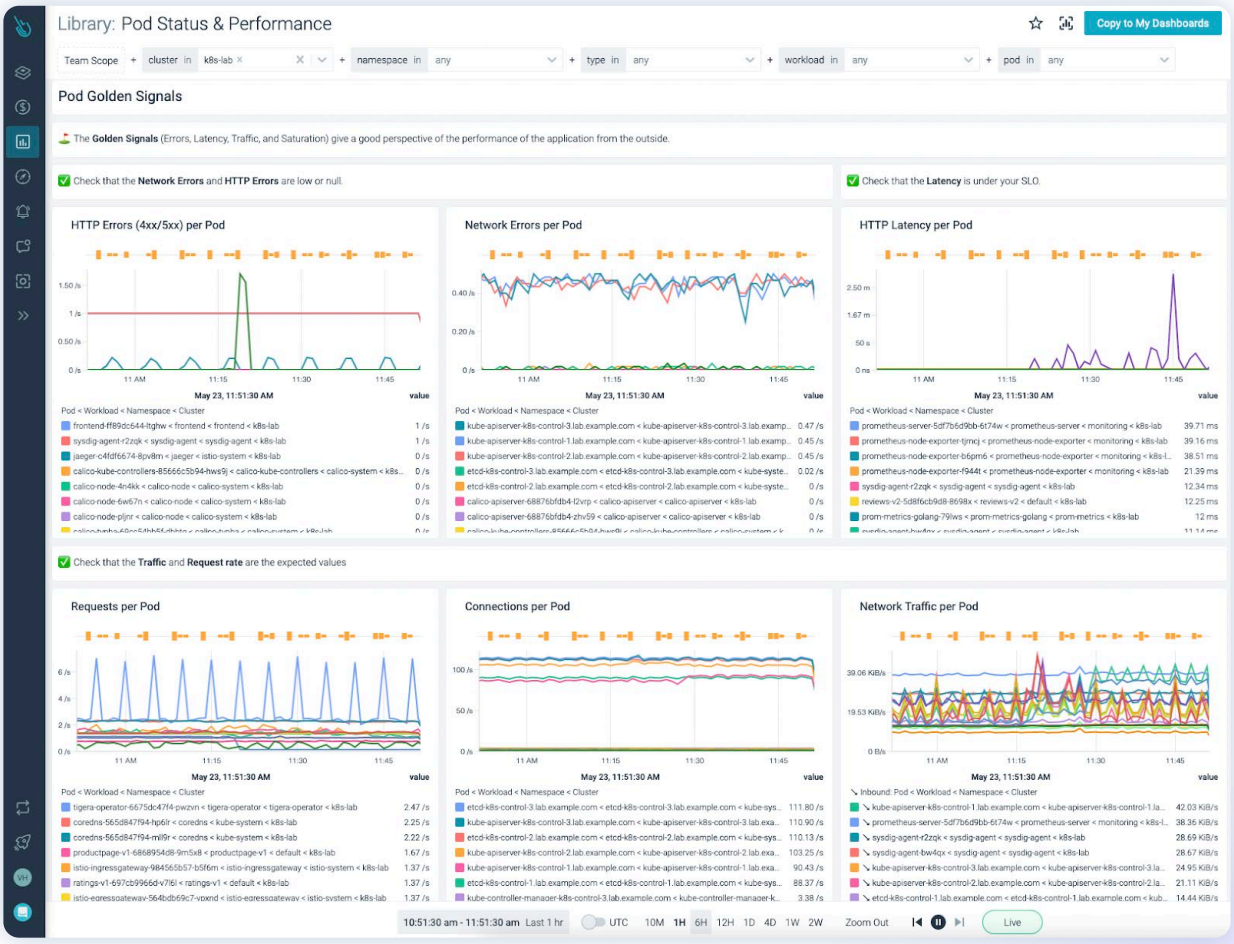
When you instrument your code using Prometheus metrics, one of the first things you'll want to do is to expose these metrics from your application. Prometheus libraries empower you to easily expose the first three: latency or response time, traffic or connections, and errors.

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, nor a Prometheus instance. 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:



Troubleshooting issues with Prometheus metrics

As was already mentioned earlier in this guide, Prometheus metrics are a combination of a given metric identified by a name with a set of key-value pairs. That's all! Part of Sysdig's magic, and one of its more powerful features, is the metric enrichment that is provided to every Prometheus metric collected from your Kubernetes cluster.

Now, you may be wondering: why is this so important? What is this metric enrichment about and what are the benefits that should help me with monitoring my Kubernetes clusters?

In short, your metrics now gain Kubernetes and cloud context. The Sysdig Agent is able to get information about your cloud and Kubernetes infrastructure; it adds and correlates this data with your own Kubernetes and Prometheus metrics. Thanks to this context addition, practitioners get valuable extra information in their own metrics, like container name, deployment name, Kubernetes host, Kubernetes IP, command running within a Pod, Kubernetes service information, cloud instance or VM information, and much more.

Metric enrichment and extended labels helps users and significantly reduces troubleshooting time by accelerating troubleshooting actions.

Exploring your Prometheus metrics

Sysdig Monitor provides multiple tools to explore your metrics on your own. On one hand, practitioners who don't feel comfortable enough using PromQL language can use the metrics explorer to drill through metrics within their infrastructure. In addition, rookies also have a form query user interface to facilitate the task of exploring, understanding, and learning how to get data with PromQL at the same time. On the other hand, most experienced users have the PromQL query tool that gives freedom to run their own PromQL queries, and use their own filters and expressions to get their data directly from metrics.

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. OpenTelemetry allows DevOps teams to either instrument applications manually or rely on the auto instrumentation mechanism to reduce their burden. Sysdig Monitor is able to pull metrics from the OTel collector via the Sysdig Agent, or receive metrics via remote write.
4. 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.
5. Prometheus metrics enrichment and extended labels bring new troubleshooting capabilities to Sysdig Monitor. Users can resolve their issues much faster by reducing their troubleshooting time significantly.

06

Challenges Using Prometheus at Scale

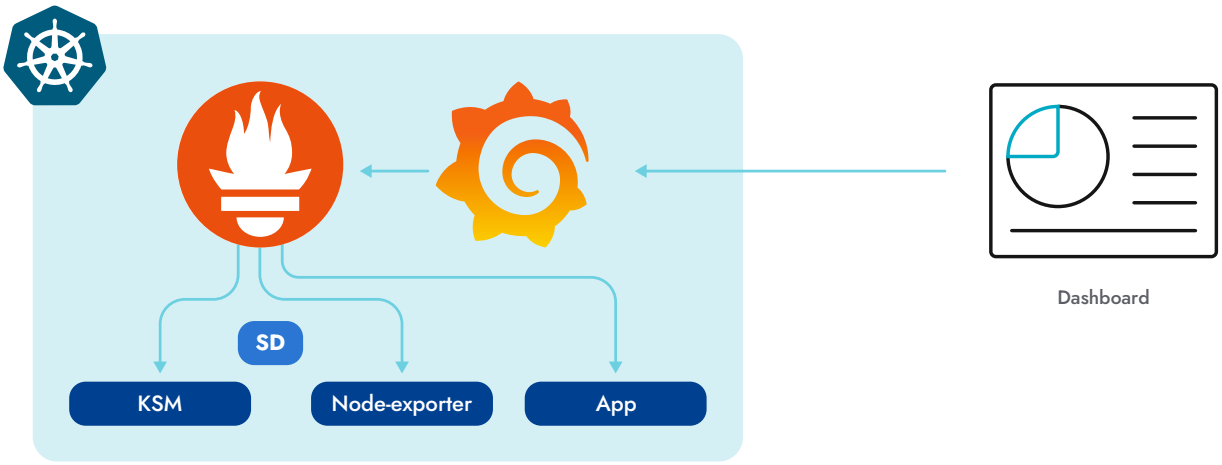
Prometheus first steps

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.

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

Then, dig deeper into Prometheus and discover useful features:

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

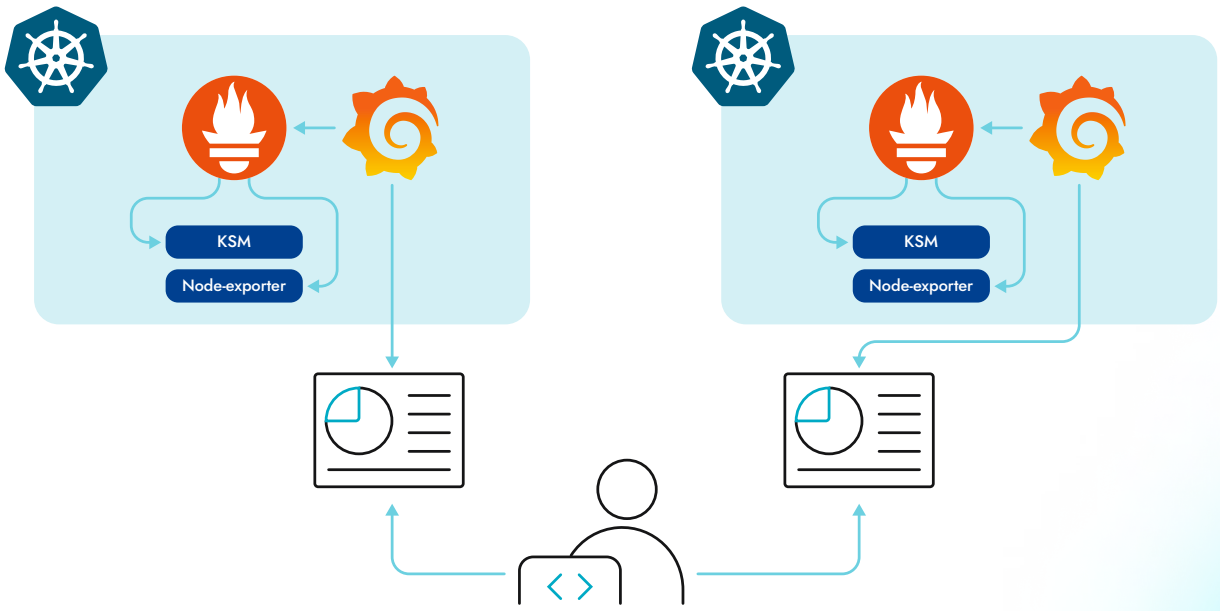


At this point, everything looks great. Kubernetes is your best friend, your apps scale smoothly, and you can watch many metrics from your fancy dashboard. You're ready for the next step.

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.

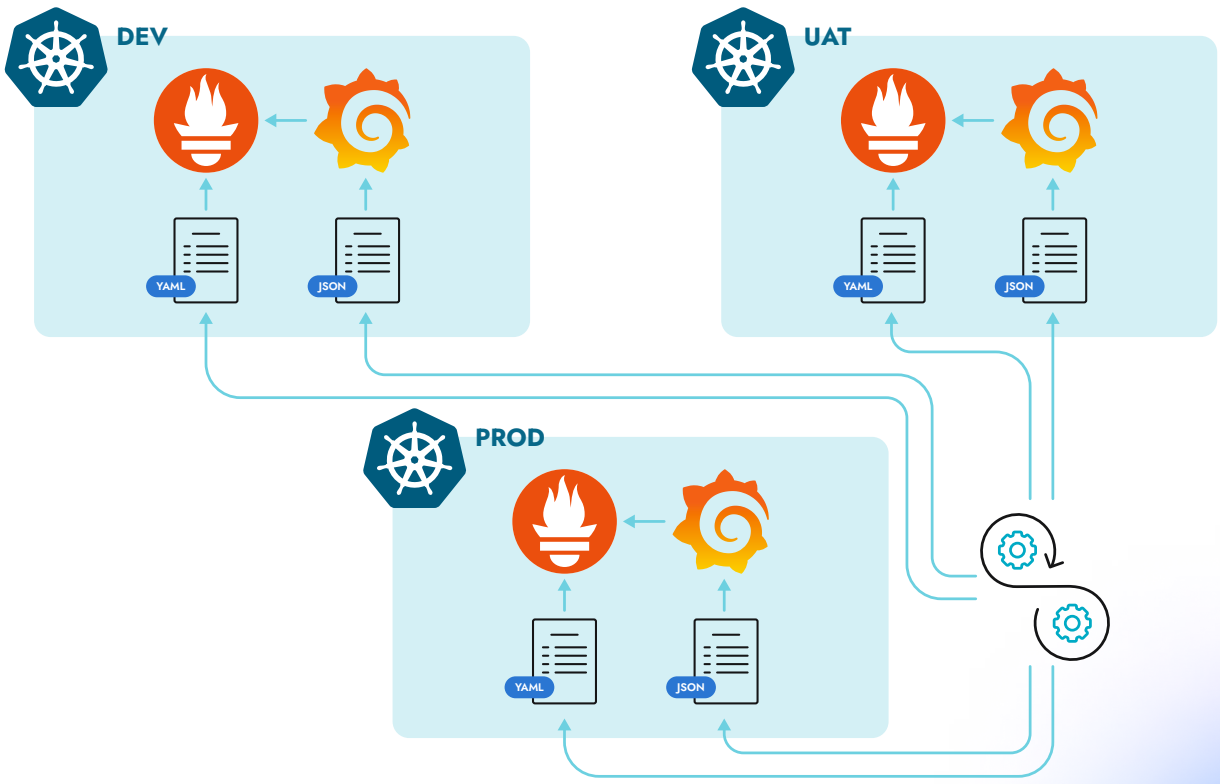
Then the first little problem comes up. Now, you have two different Grafana installations and you need to switch between them to see your metrics from both development and staging environments. Developers aren't too worried about this so far, but it's starting to itch. Everything is checked out and the production release is good to go.



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

The outcome is great! Every team working around Prometheus is excited with the observability platform, so 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 number of applications grows, new requirements show up. New Kubernetes clusters are created to satisfy the demand, allocate workloads in different regions, and increase the service availability. Keeping different Prometheus and Grafana instances for each one becomes a challenge. This time, DevOps and development teams start to complain.



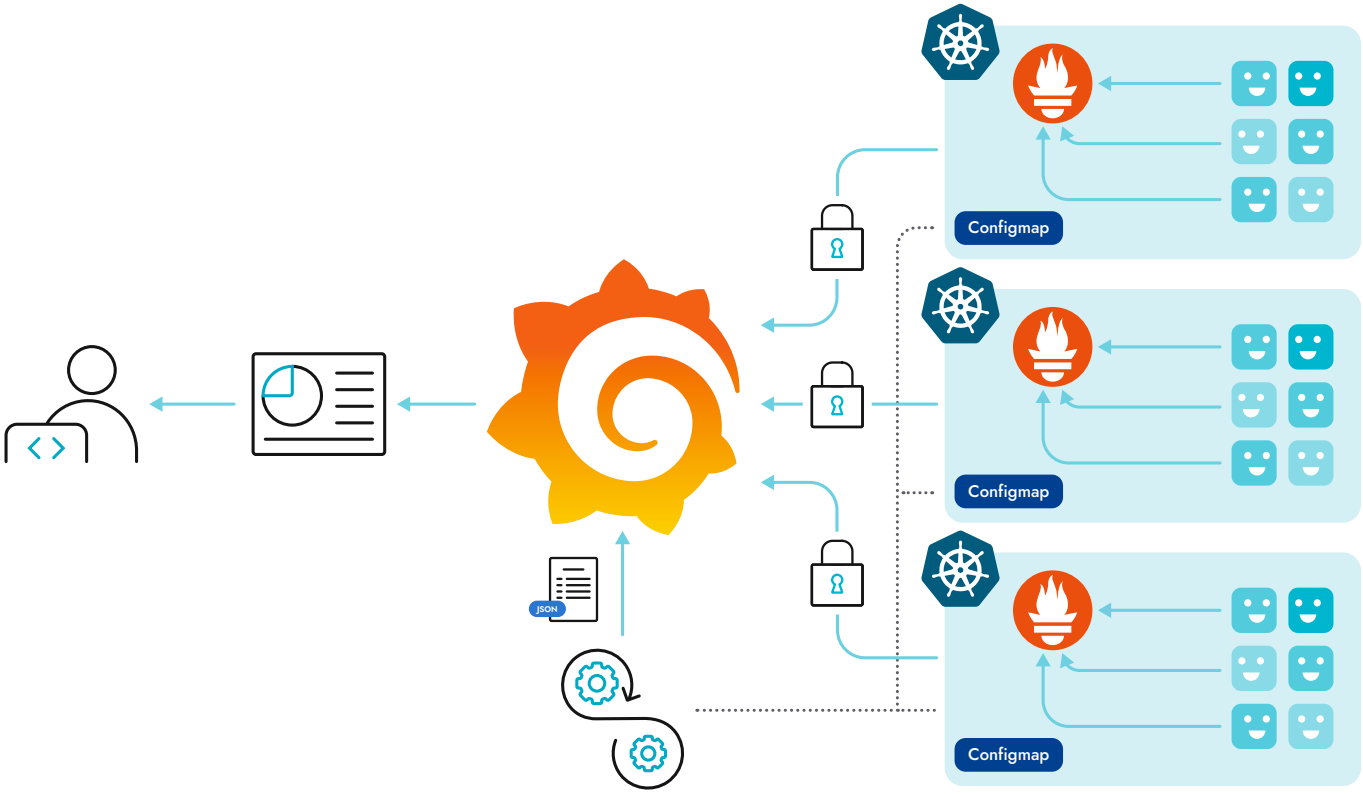
Keeping global visibility

The distributed model presents different challenges:

- There is a lack of global visibility. This is a must if you run several clusters.
- 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 seems really easy. You can deploy a centralized Grafana and plug 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 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.

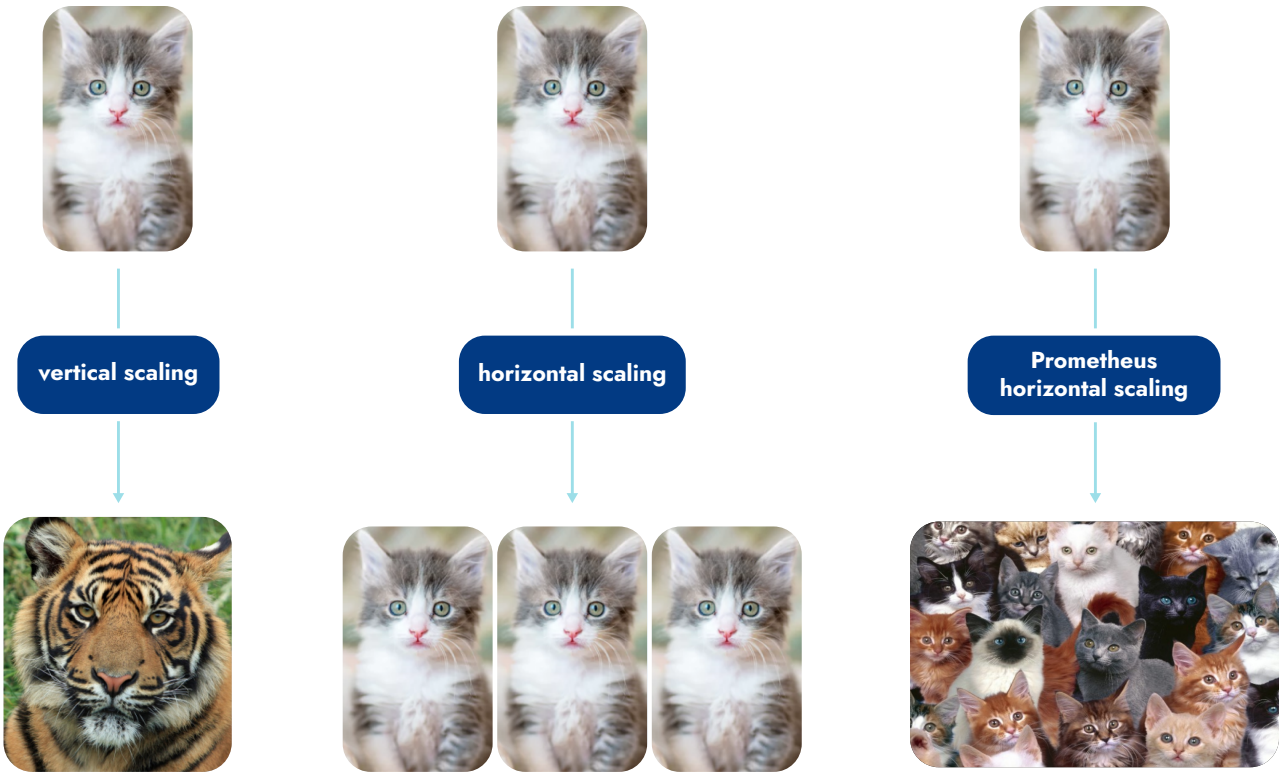
Prometheus horizontal scale

With some effort from your team, you have a decent centralized system and everything seems perfect – until a new problem comes up. Developers embrace the power of business custom metrics and instrument applications to get that key business data. While your organization grows, the number of Kubernetes services increase and metrics usage rises. Clusters are upscaled and Kubernetes services have more and more replicas.

Suddenly, your Prometheus servers start becoming unresponsive and stop working. After some research, you find out a memory problem. Memory usage in Prometheus is directly proportional to the number of time series stored; as your time series grows in numbers, you start to face OOM kills. In order to mitigate this issue, 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 instance with millions of metrics can use more than 100GB of RAM, and that can be an issue when running Prometheus in Kubernetes. You must scale out to absorb capacity, but that isn't so easy. 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:

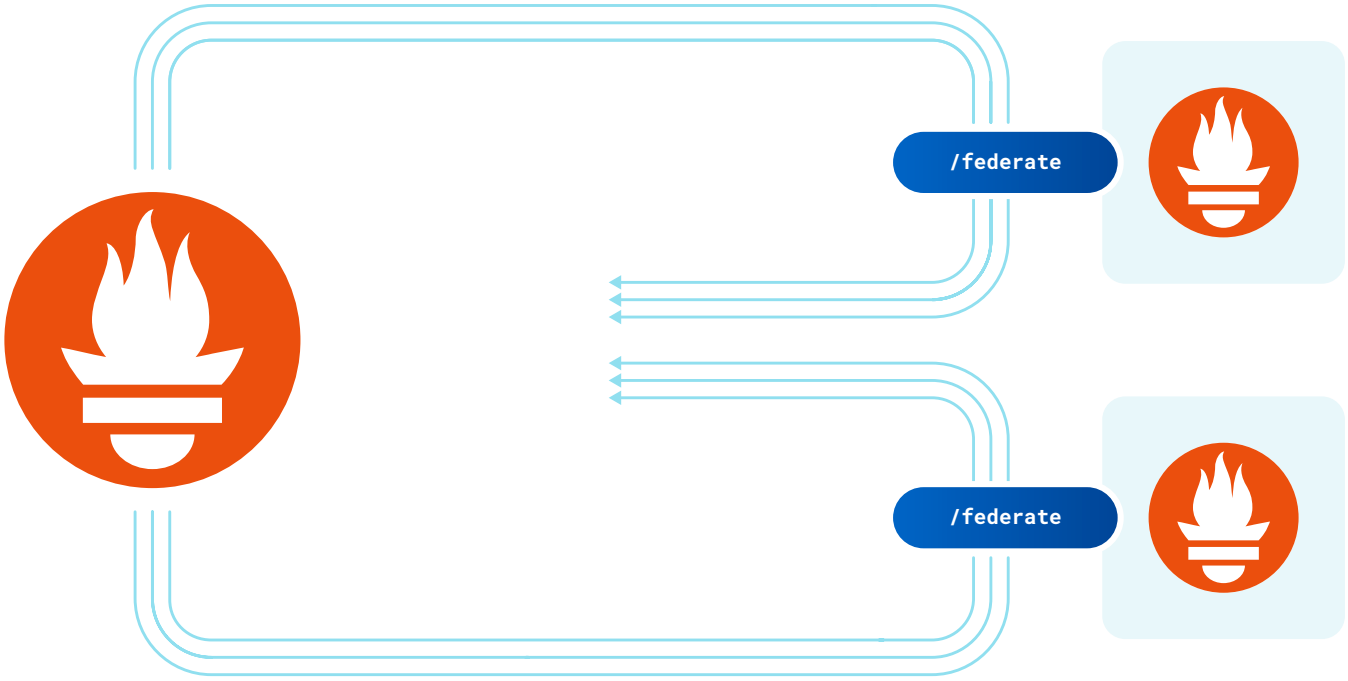
Federation

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:

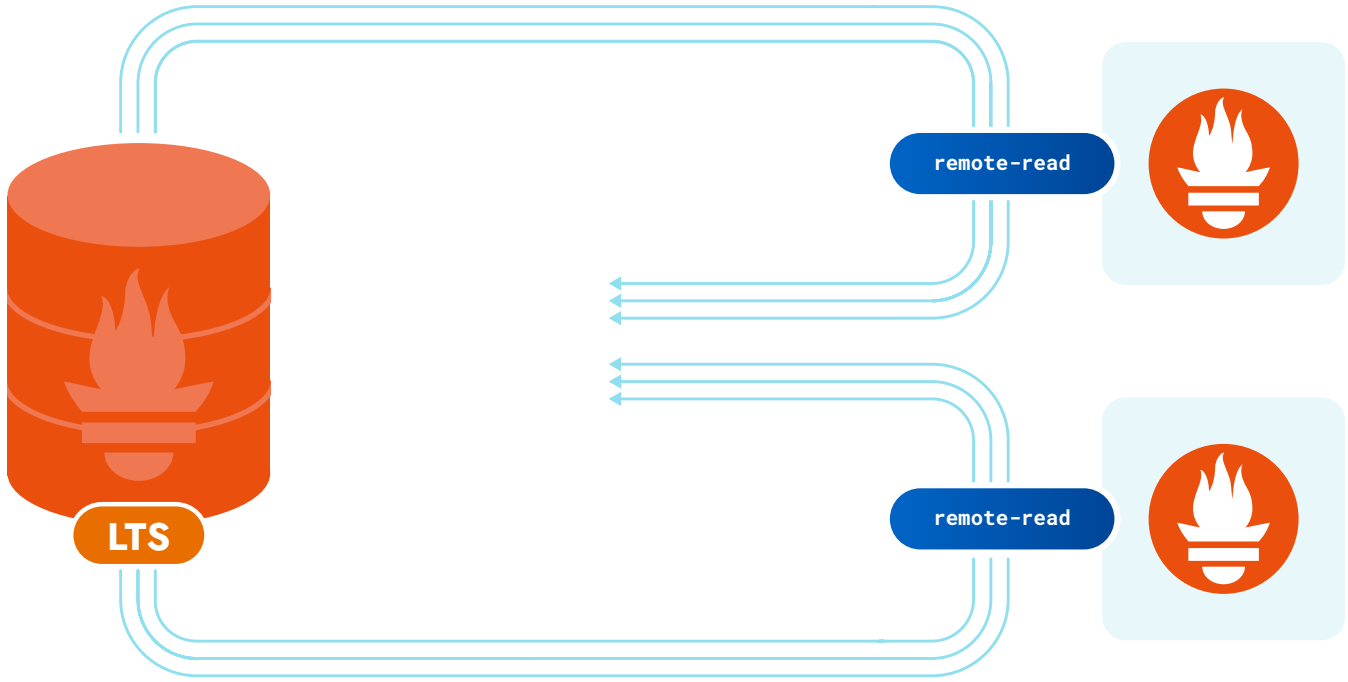
- The amount of data that can be sent is limited as the scale problem is even worse. You are receiving all the data in one Prometheus instance.
- The aggregations must be done first in Prometheus itself, then the requests for the federation endpoint need filtering and explicit information.
- Maintenance 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

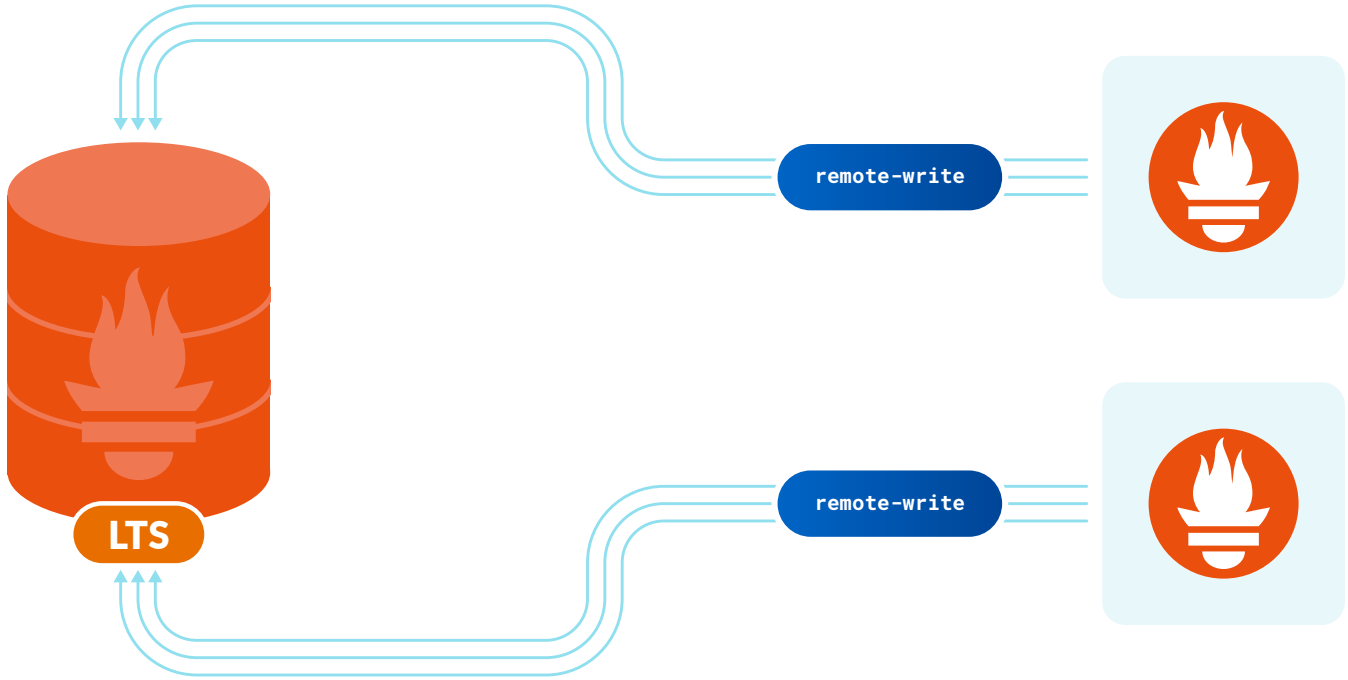
This case is similar to federation in the sense of architecture, as the metrics are pulled from other Prometheus servers, but is designed to be more efficient and interoperable. Remote read improves the amount of data that can be pulled and aggregated since it is defined in the original Prometheus instance.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

his 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 primarily on remote_write to receive metrics from multiple Prometheus instances at the same time.

An example of remote_write configuration would be:

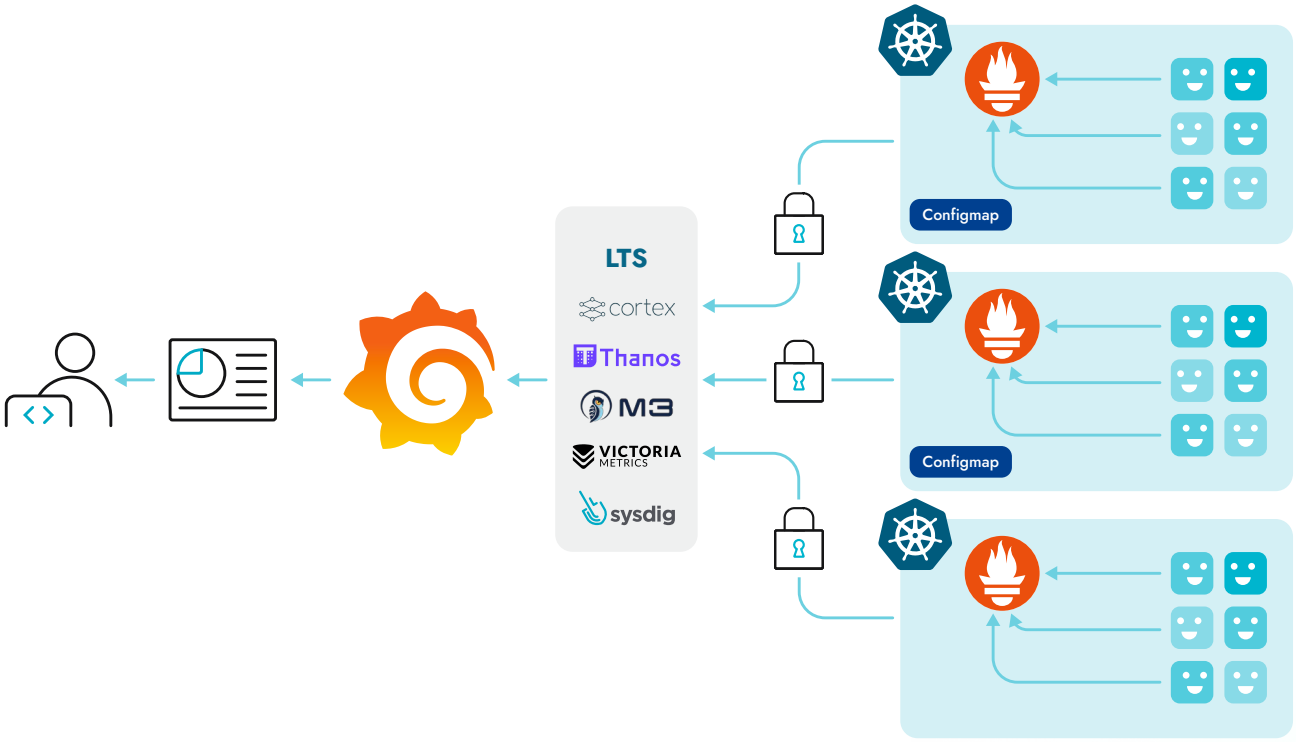
```
remote_write:
- url: "https://mylts.com/remote_write"
  authorization:
    credentials: "g4ghj5ikuwhxf9ñojw0wfegwrtg2"
    type: "Bearer"
  tls_config:
    ca_file: /etc/pki/myca.crt
```

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

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 long-term storage for its metrics.

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



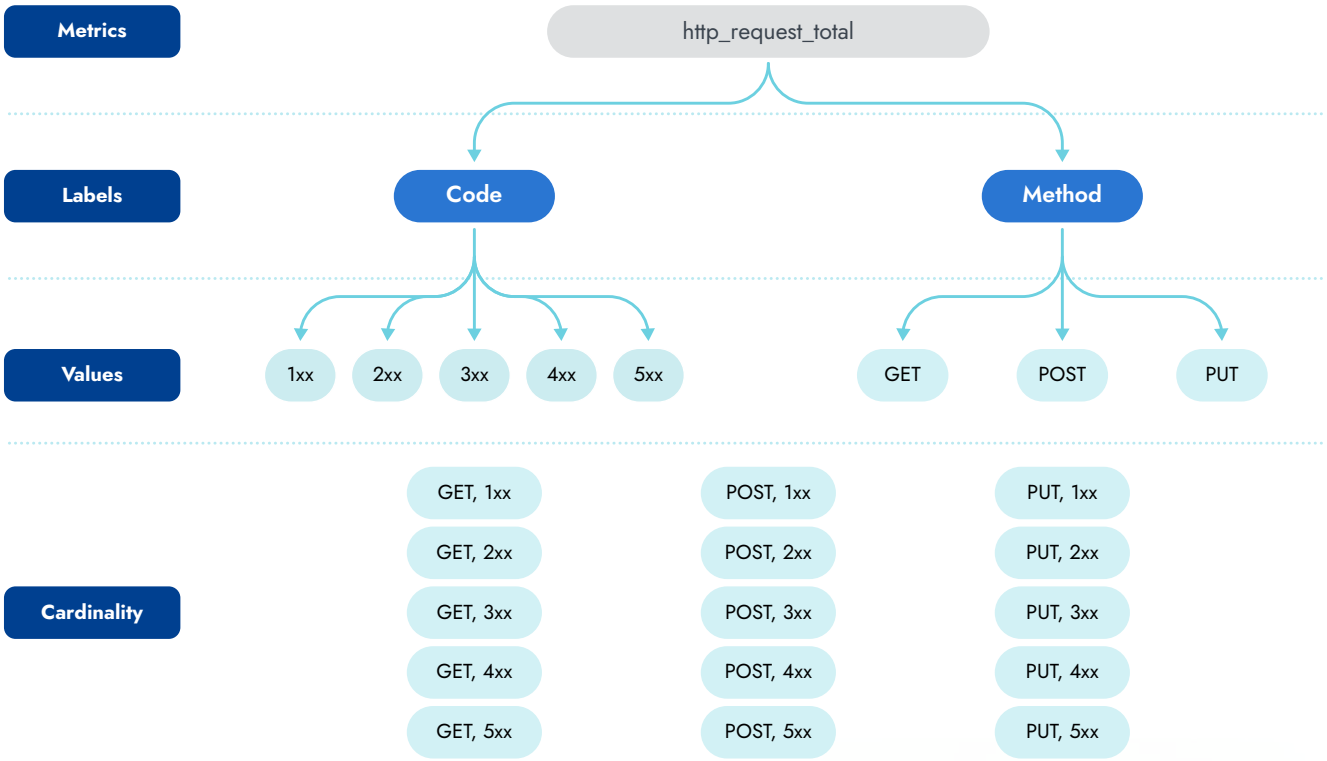
These services are not free of complications, however:

- 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.

Prometheus metrics cardinality

The definition of cardinality is “the number of elements in a given mathematical set.” But, what does exactly cardinality mean when talking about Prometheus metrics?

A metric is identified by a name, and it contains one or more labels. At the same time, a label can have one or more values. We can say cardinality is the total number of combinations of unique values for a given metric. In the following example, considering all the possible combinations for **`http_requests_total metric`**, we can say it has a cardinality of 15.



Let's continue with the example above. Imagine the http_requests_total includes a new label called "instance." This instance label contains the internal container SDN IP that has served the http request. Containers are ephemeral; you may start with a couple of them, but as time goes by, you may end up scaling up to hundreds or thousands for the same application.

```
http_requests_total{code="200", method="get", instance="10.128.1.23"}
http_requests_total{code="404", method="get", instance="10.128.9.46"}
http_requests_total{code="503", method="get", instance="10.128.1.23"}
…
```

As soon as new labels and new values are created, metrics cardinality starts to grow exponentially. Quite often, Prometheus users start collecting metrics and storing them into their Prometheus instance without paying attention to how cardinality evolves and how it affects their observability platform. After some time using Prometheus, users missing cardinality growth may start noticing occasional performance issues while inspecting their metrics like higher latency or large response times. Next, outages will come up. Your Prometheus instance fails to serve and becomes unresponsive not only against requests from users, but also when pulling data from your infrastructure. That's why taking care of metrics cardinality is so important. Otherwise, you may end up with a collapsed and useless observability platform.

That phenomenon is commonly known as cardinality explosion. Sometimes, this may suddenly come up after changes or additions to any of the actual metrics in your system. For example, you add a new label to add more context to your business sales workflow. Thanks to this new addition, you start tracking the product ID in some of your business core sales transactions metrics. If you have thousands of products and millions of new transactions every day, adding this new dimension may have a huge impact. Cardinality will significantly grow suddenly, negatively impacting your observability platform, and leading to an excessive resource consumption. This causes instability and even outages due to memory errors and system crashes.

Prometheus metrics costs

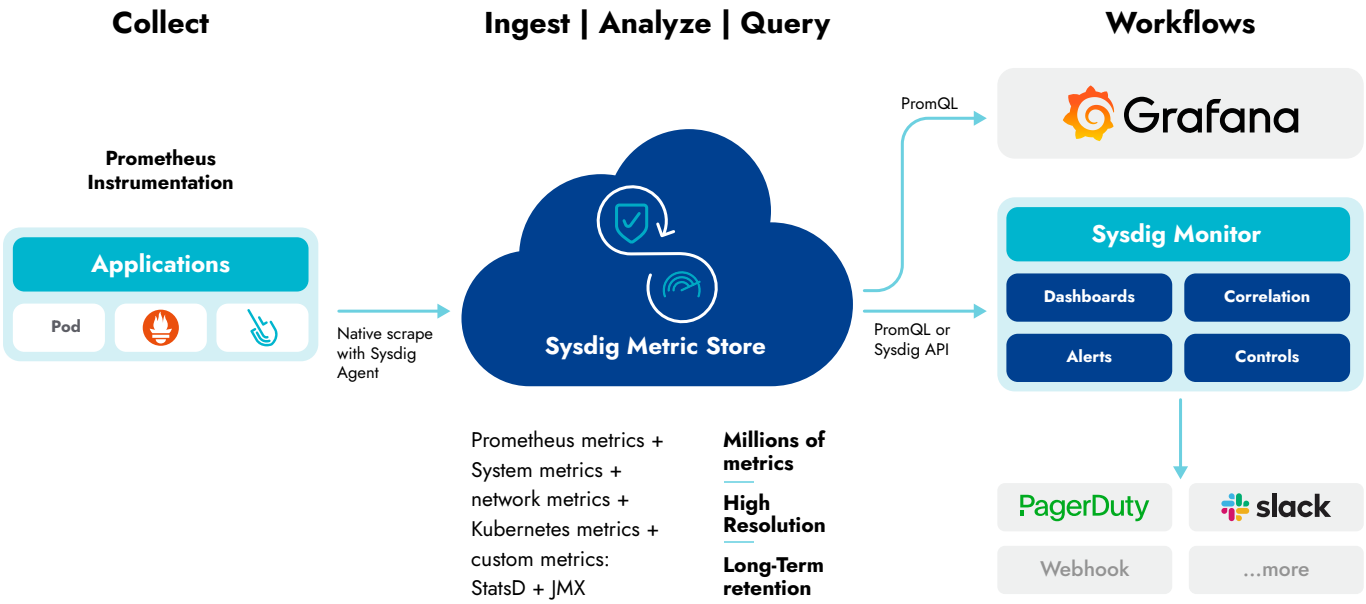
Cardinality explosion is one of the main culprits of increasing Prometheus metrics costs. It will make you pay more for the underlying resources (instances, cpu, memory, storage, networking, etc.). Businesses that delegate metrics ingestion and processing to some of the actual managed services for Prometheus offerings available in the market are not exempt from this problem. Cardinality explosions can definitely cause a lot of trouble with costs. The more volume of time series, the more you pay for your metrics. A sudden growth of time series will likely skyrocket your observability bill.

As discussed earlier, it is key to observe and monitor your metrics cardinality and the total volume of time series you are handling constantly. That way, if you spot any sudden metrics volume growth or a cardinality spike, you can react rapidly and prevent unnecessary (in most cases) wasted spending.

Scale and simplify Prometheus Monitoring with Sysdig Monitor

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 started to work steadily a long time ago to provide a complete multi-cloud and Kubernetes observability solution. Following, you'll find more information on how Sysdig Monitor helps customers with resolving the challenges described earlier.

Addressing Prometheus scalability, availability, and performance challenges

As you already saw in this chapter, sooner or later OSS Prometheus users start facing performance and availability problems. Prometheus doesn't scale well when the metrics volume grows. Sysdig Monitor is a **fully Prometheus compatible** SaaS solution with long-term storage, which is permanently being improved to scale to millions of time series and retain data for longer periods. You won't need to worry about performance, stability, and availability anymore – everything is covered by Sysdig. With Sysdig, you can take advantage of a supported platform that delivers a scalable, secure solution for Prometheus monitoring.

Addressing long-term storage challenges

OSS Prometheus is not designed for storing high volumes of data nor to provide enough retention for most customer needs. There are few OSS projects that help with metrics long-term storage, but implementation, deployment, and operation is complex and costly. In terms of long- term storage retention, Sysdig gives 12 months of data retention at no extra cost and it is cheaper than competition. There is no limit on the metrics volume you manage, just push your metrics and let the Sysdig team take care of your data.

Addressing global visibility challenges

As you saw in the "Challenges using Prometheus at scale" section, one of the pain points is to maintain and operate across different Prometheus and observability instances. The more clusters you run, the more observability interfaces you have to watch your data. This can be unmanageable when running a certain number of Kubernetes clusters. It will lead to wasting your time going back and forth, making mistakes.

With Sysdig Monitor, you get full visibility of your whole infrastructure from the very beginning; your cloud-native applications, custom metrics, Kubernetes clusters, and cloud environments, from a single pane of glass. Just log into the Sysdig Monitor portal and start watching your metrics and troubleshooting your applications right away.

DevOps teams will benefit from out-of-the-box dashboards and alerts to start diving into your data. A PromQL interface is available for experienced users that prefer to query directly metrics data with their own queries. A form query user interface to help and educate newbies with PromQL language is also available for Sysdig Monitor customers.

Addressing Prometheus data exporting challenges

Sysdig Monitor is quite versatile and supports multiple ways to ingest data from endpoints. That way, you can overcome certain limitations like when monitoring Windows VMs. You can ingest data through:

- Sysdig Agent. It is deployed on every Kubernetes node, and pulls not only your Prometheus metrics but Kernel insights from your nodes.
- Remote write. If you want to push Prometheus metrics from environments where the Sysdig Agent can not be installed.
- Cloud integrations. Sysdig Monitor can connect to any of the three main cloud providers (AWS, Google Cloud, Azure) to pull cloud metrics and other data.

In addition, Sysdig Monitor can also push your metrics to other external Prometheus instances or even plug into external Grafana dashboards.

Addressing Prometheus metrics cardinality challenges

We already talked about what cardinality explosion is and why you should take care of it. If you fail to take control of this, you may run into serious trouble such as performance and availability issues, and costs may go unnoticed until you have a huge bill on your table.

With Sysdig, you don't have to worry about performance and availability issues with cardinality explosion. Sysdig Monitor is a SaaS solution, and there is a dedicated team taking care of its reliability. In terms of costs, Sysdig Monitor provides a dashboard to monitor your metrics ingestion and the volumes you manage. In almost real time, customers can watch how their metrics volume grows.

Addressing Prometheus metrics costs challenges

If you are interested in custom metrics costs, Sysdig provides mechanisms to track your custom metrics ingestion volume and avoid issues with cardinality explosion costs. In addition, for those customers interested in Kubernetes and cloud costs, Sysdig Advisor helps customers with monitoring your actual costs and providing remediation steps to start lowering them.

Addressing Prometheus exporters challenges

Prometheus service discovery allows you to easily get your Prometheus metrics automatically without making any extra effort. But there are still a lot of third-party applications across the vast cloud universe not yet instrumented for exporting Prometheus metrics. Fortunately, there are a lot of Prometheus exporters for a huge number of applications supported and maintained by the community. The downside is that for a single application, you may find several exporters. Each of these is designed differently, requiring its own particular configuration, and sometimes providing different functionality. In many cases these are not maintained at all. That may lead you to face several issues: from the simplest, like struggling with the deployment and configuration or having to troubleshoot a exporter because it suddenly stopped working, to the most severe problems, like being attacked through a security breach because of a container image that was neither maintained nor updated.

Promcat.io comes to solve all these problems. Sysdig curates and maintains a whole catalog of Prometheus exporters to take your burden off. In short, these exporters just work, and customers don't have to worry about maintenance or security issues. Sysdig Monitor customers have direct access to Promcat integrations from the Sysdig Monitor portal. For those applications that have an integration, Sysdig detects that match, and offers customers to deploy the exporter following a few guided steps. Along with the exporter that is responsible for collecting metrics, Sysdig also provides out-of-the-box dashboards and alerts to start monitoring right away.

07

Conclusion

OSS Prometheus is easy to deploy. After following a few steps, you have an observability platform up and running in minutes, integrated in your cloud-native applications environment. While it is really easy to kick off, and completely valid for many users that just started in their Kubernetes and cloud observability journey, it may not be the best solution as they grow. As time goes by, organizations start to grow, business requirements evolve rapidly, and more applications and new data points are spread across multiple cloud infrastructures, requiring more and more resources to analyze their own KPIs. Businesses relying on OSS Prometheus start to face challenges at that point. Prometheus doesn't scale well for a high number of metrics, when long-term storage is required, or for multi-cloud environments, which are usually some of the main requirements. There are other OSS federation projects that may help with scaling and long-term storage, like Cortex and Thanos, but they require maintenance and resource management. In addition, these can't help with enterprise business requirements like access control and 24x7 support.

If you are one of these practitioners that are just starting to face some of the Prometheus issues we presented in this guide, or you are starting to see how your infrastructure and your Prometheus metrics volume are growing quickly, don't wait any longer. – things can easily go from bad to worse. In this guide, we presented some of the challenges when using Prometheus at scale, and how to overcome those issues and problems with Sysdig Monitor. In addition, 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.

If you want to learn more about how Sysdig can help you to overcome these and other challenges, check out the following resources:

- Prometheus and Kubernetes Metrics Ingestion
- Prometheus Exporters in Sysdig Monitor
- Troubleshooting Application Issues with Extended Labels
- How Much Does Your Managed Service for Prometheus Cost?
- Did Your Datadog Bill Explode? A Custom Metrics Cost Comparison
- Best Practices for Reducing the Cost of Custom Metrics
- Cut Custom Metrics Cost by 75% and Observe More
- Kubernetes Monitoring Guide
- Migrating from Prometheus, Grafana, and Alert Manager to Sysdig Monitor

How Sysdig Monitor Helps

Sysdig provides a fully compatible Prometheus SaaS solution with long-term support which is permanently being improved to scale to millions of time series. It is also much more cost effective than other Prometheus solutions available in the market. You can request a 30 days trial account and try it for free.

sysdig.com/start-free