



# Prometheus Monitoring Guide

## - How to monitor Kubernetes.

Prometheus is fast becoming one of the most popular Docker and Kubernetes monitoring tools to use. Read this eBook to learn about the benefits of Prometheus, and how to use it for monitoring Kubernetes services, clusters and components.

eBOOK



# Kubernetes Monitoring with Prometheus

## - **How to Monitor Kubernetes.**

About Prometheus.....	3
Key Features .....	4
How Prometheus Overcomes Monitoring Challenges .....	5
How to Monitor Kubernetes with Prometheus.....	7
Architecture Overview .....	7
Monitoring Services, Clusters and Components.....	16
Conclusion .....	23

# Kubernetes Monitoring with Prometheus.

## About Prometheus

Prometheus, the open-source systems monitoring and alerting toolkit originally built at SoundCloud, is fast becoming one of the most popular Docker and Kubernetes monitoring tools to use.

The rise of Prometheus can be attributed to two technology shifts that created a need for a new type of monitoring framework:

**1. DevOps Culture:** Prior to the emergence of DevOps, monitoring was comprised of hosts, networks and services. Within the world of DevOps, modern developers are often very involved in the CI/CD pipeline and responsible for their own perform operations debugging. As a result, they require the ability to easily integrate both app and business-related metrics as an organic part of the infrastructure. Monitoring needed to be democratized, made more accessible, and expanded to cover additional layers of the stack.

**2. Containers and Kubernetes:** Container-based infrastructures have radically changed logging, debugging, high-availability and monitoring. Today's IT teams are faced with an enormous number of volatile software entities, services, virtual network addresses and exposed metrics that suddenly appear or vanish. Where traditional monitoring tools fall short, Prometheus delivers.

# Kubernetes Monitoring with Prometheus.

## Key Features

Several features come together to position Prometheus as the tool to use for monitoring containerized environments:

1. **Multi-dimensional data model:** Flexible and accurate time series data powers the Prometheus query language. The model is based on [key-value pairs](#), similar to how Kubernetes organizes infrastructure metadata using labels.
2. **Accessible format and protocols:** Exposing Prometheus metrics is a straightforward task. The metrics are human readable, are in a self-explanatory format, and are published using a standard HTTP transport.
3. **Service discovery:** The Prometheus server periodically scrapes the targets, so that applications and services aren't required to do so. In other words, metrics are pulled, not pushed. A Prometheus server can use one of several methods to auto-discover targets, and some can be configured to filter and match container metadata, which is ideal for ephemeral Kubernetes workloads.
4. **Modular and highly available components:** Composable services, which perform metrics collection, alerting, graphical visualization and other similar functions, are designed to support redundancy and sharding.

# Kubernetes Monitoring with Prometheus.

## How Prometheus Overcomes Monitoring Challenges

Prometheus helps overcome many of the unique challenges that monitoring Kubernetes clusters can present.

### 1. Container visibility

Containers are lightweight, mostly immutable black boxes, which can make them hard to find and to monitor.

While the Kubernetes API and the [kube-state-metrics](#) (which natively uses Prometheus metrics) help to solve for this by exposing Kubernetes internal data (number of desired/running replicas in a deployment, unschedulable nodes, etc.), Prometheus makes it even easier.

With Prometheus, all you need to do is expose a metrics port, which doesn't add much complexity or run additional services. Often, the service itself is already presenting a HTTP interface, and all that's required is an additional path, such as `/metrics`.

If the service is not prepared to serve Prometheus metrics and the code can't be modified to support it, simply deploy a [Prometheus exporter](#) that is bundled with the service, often as a sidecar container of the same pod.

# Kubernetes Monitoring with Prometheus.

## 2. Changing and volatile infrastructures

Ephemeral entities that can start or stop reporting any time are a problem for classical, more static monitoring systems.

Prometheus is a dynamic monitoring tool that has [three helpful autodiscover mechanisms](#):

- **Consul:** A tool for service discovery and configuration. Consul is distributed, highly available, and extremely scalable.
- **Kubernetes:** Kubernetes SD configurations allow retrieving scrape targets from Kubernetes' REST API and always staying synchronized with the cluster state.
- **Prometheus Operator:** Automatically generates monitoring target configurations based on familiar Kubernetes label queries.

## 3. New infrastructure layers

Kubernetes components create new infrastructure layers, and the need to organize monitoring based on groupings, such as microservice performance (with different pods scattered around multiple nodes), namespace, deployment versions, etc.

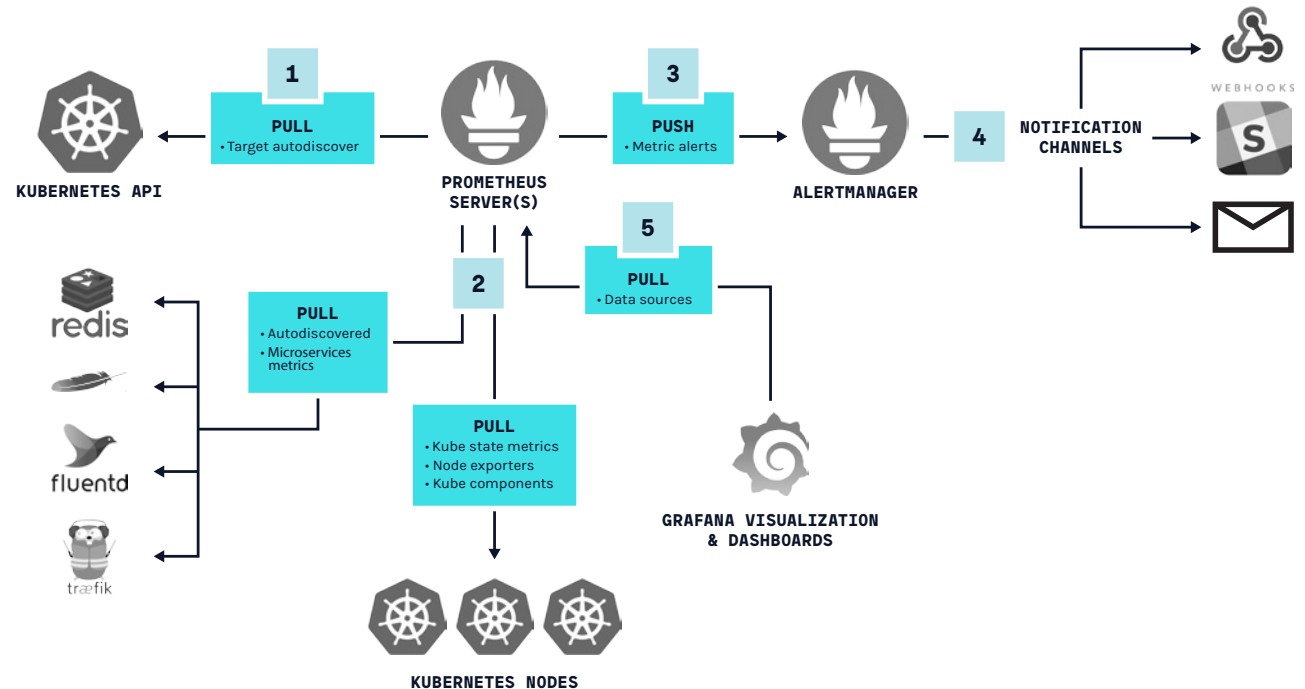
Using the label-based data model of Prometheus together with the [PromQL](#), you can easily adapt to these new scopes.

# Kubernetes Monitoring with Prometheus.

## How to Monitor Kubernetes with Prometheus

### Kubernetes Cluster Architecture Overview

This diagram illustrates the core components and their respective functions within a Kubernetes cluster:



# Kubernetes Monitoring with Prometheus.

## 1. Kubernetes API

Auto discovery of targets by Prometheus can be achieved in a few different ways:

- Consul
- Prometheus Kubernetes SD plugin
- The Prometheus operator and its [Custom Resource Definitions](#)

## 2. Kubernetes Nodes

Prometheus should collect application metrics, as well as metrics related to Kubernetes services, nodes and orchestration status:

- [Node exporter](#), for the classical host-related metrics: CPU, mem, network, etc.
- Kube-state-metrics for orchestration and cluster level metrics: deployments, pod metrics, resource reservation, etc.
- Kube-system metrics from internal components: kubelet, etcd, DNS, scheduler, etc.

## 3. Alerts

Prometheus can configure rules to trigger alerts using PromQL, alertmanager will be in charge of managing alert notification, grouping, inhibition, etc.

## 4. Notification Devices

The alertmanager component configures the receivers, gateways to deliver alert notifications.

## 5. Dashboards

Grafana, the open platform for analytics and reporting, can pull metrics from any number of Prometheus servers and display panels.



# Kubernetes Monitoring with Prometheus.

## Monitoring Kubernetes with Prometheus

There are many components to deploying a Kubernetes with Prometheus monitoring. Here we'll cover the core elements for using Prometheus to monitor end-user apps and Kubernetes cluster endpoints.

## Kubernetes Services

There are two types of services: those that natively have the ability to expose Prometheus metrics or offers a Prometheus endpoint, and those that don't. The good is news is that many services that don't can easily be made to do so using an exporter, which is a service that collects service stats and translates metrics ready to be scraped to Prometheus.

### 1. Monitoring a service that can expose Prometheus metrics

[Traefik](#) is a reverse proxy designed to be tightly integrated with microservices and containers. A common use case for Traefik is to be used as an Ingress controller or Entrypoint, this is, the bridge between Internet and the specific microservices inside your cluster.

You have several options to [install Traefik](#) and a [Kubernetes-specific install guide](#). If you just want a simple Traefik deployment with Prometheus support up and running quickly, use the following snippet:

```
kubectl create -f https://raw.githubusercontent.com/mateobur  
prometheus-monitoring-guide/master/traefik-prom.yaml
```

# Kubernetes Monitoring with Prometheus.

Once the Traefik pods is running and you can display the service IP:

```
kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	238d
prometheus-example-service	ClusterIP	10.103.108.86	<none>	9090/TCP	5h
traefik	ClusterIP	10.108.71.155	<none>	80/TCP,443/TCP,8080/TCP	35s

You can check that the Prometheus metrics are being exposed just using curl:

```
curl 10.108.71.155:8080/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"} 2.4895e-05
go_gc_duration_seconds{quantile="0.25"} 4.4988e-05
...
```

Now, you need to add the new target to the `prometheus.yml` conf file.

You will notice that Prometheus automatically scrapes itself:

```
- job_name: 'prometheus'

# metrics_path defaults to '/metrics'
# scheme defaults to 'http'.

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

# Kubernetes Monitoring with Prometheus.

Let's add another static endpoint:

```
- job_name: 'traefik'
  static_configs:
    - targets: ['traefik:8080']
```

If the service is in a different namespace you need to use the FQDN (i.e. `traefik.default.svc.cluster.local`)

Of course, this is a bare-minimum configuration, the scrape config supports multiple parameters such as:

- `basic_auth` and `bearer_token`: You endpoints may require authentication over HTTPS, using a classical login/password scheme or a bearer token in the request headers.
- `kubernetes_sd_configs` or `consul_sd_configs`: different endpoint autodiscovery methods.
- `scrape_interval`, `scrape_limit`, `scrape_timeout`: Different tradeoffs between precision, resilience and system load.

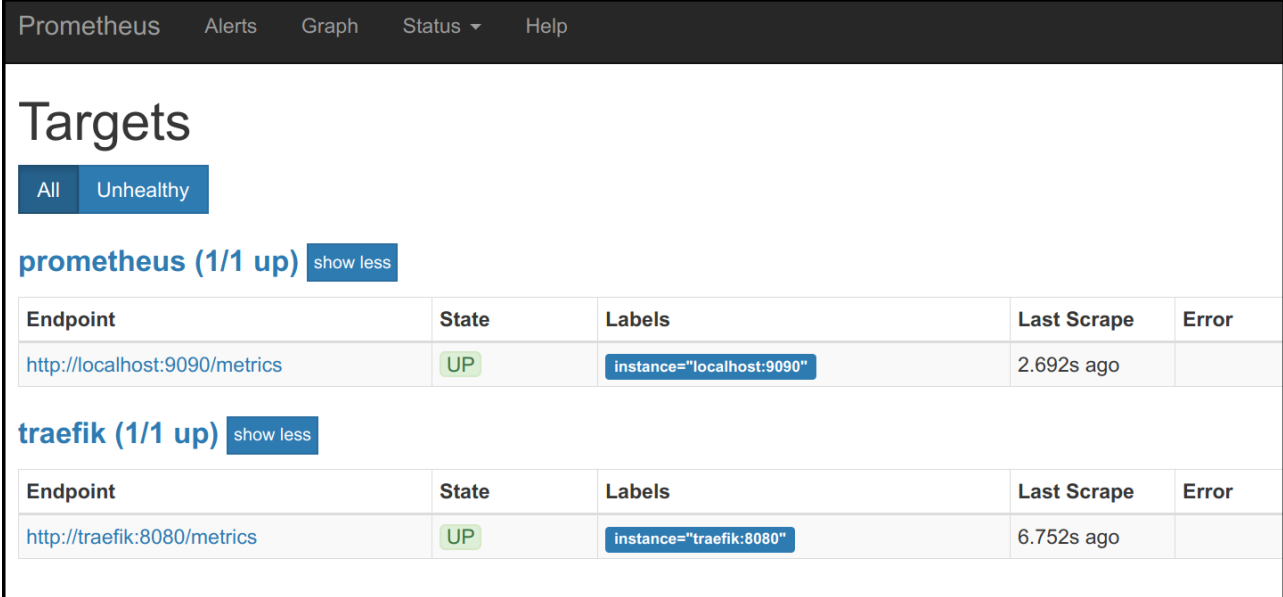
Patch the ConfigMap and Deployment:

```
kubectl create configmap prometheus-example-cm --from-file=prometheus.
yaml -o yaml --dry-run | kubectl apply -f -

kubectl patch deployment prometheus-deployment -p \
  '{"spec":{"template":{"metadata":{"labels":{"date":"'date
  +%s'`"}}}}}'
```

# Kubernetes Monitoring with Prometheus.

If you access the `/targets` URL in the Prometheus web interface, you should see the Traefik endpoint UP:



The screenshot shows the Prometheus web interface with the following structure:

- Navigation bar: Prometheus, Alerts, Graph, Status ▾, Help
- Section: Targets
- Filters: All (selected), Unhealthy
- Target Group 1: **prometheus (1/1 up)** [show less](#)
- Table 1:

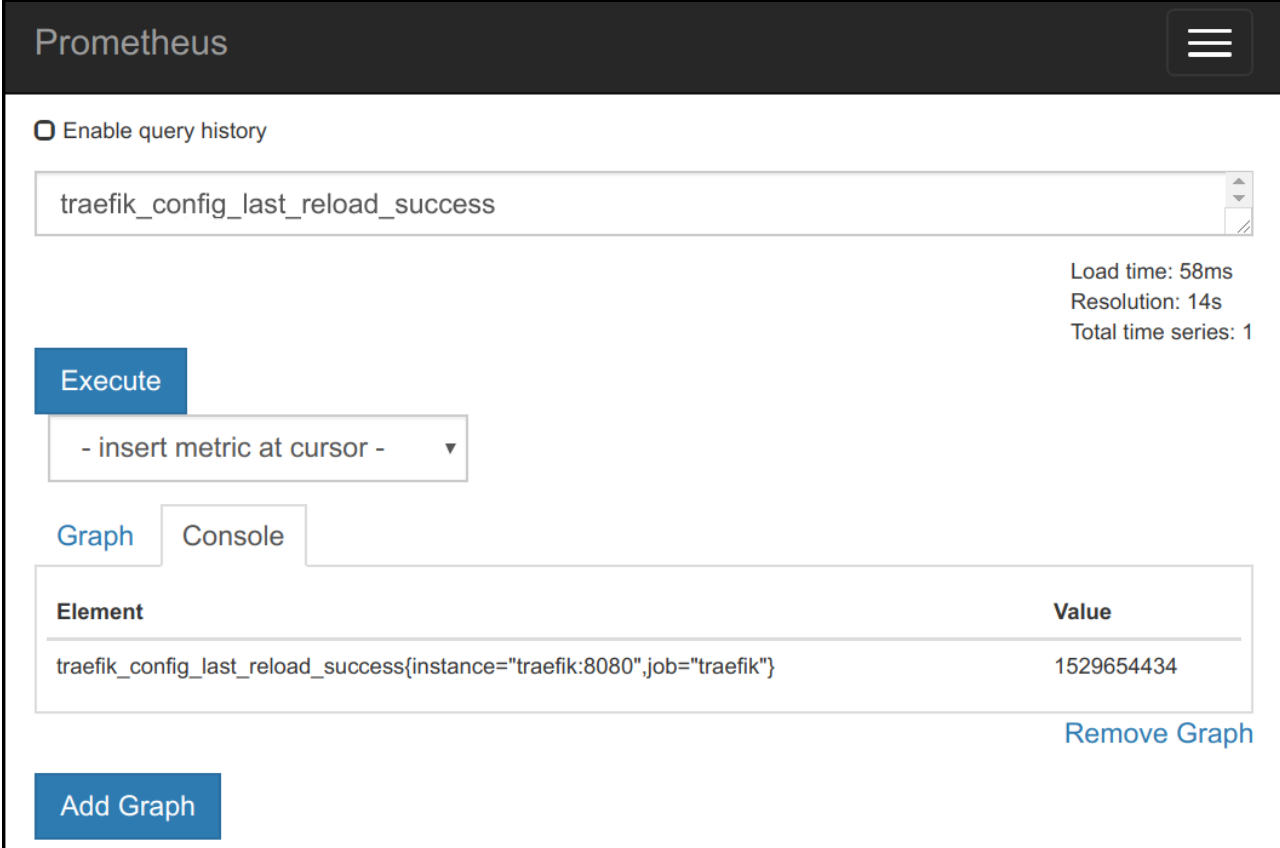
Endpoint	State	Labels	Last Scrape	Error
<a href="http://localhost:9090/metrics">http://localhost:9090/metrics</a>	UP	instance="localhost:9090"	2.692s ago	

- Target Group 2: **traefik (1/1 up)** [show less](#)
- Table 2:

Endpoint	State	Labels	Last Scrape	Error
<a href="http://traefik:8080/metrics">http://traefik:8080/metrics</a>	UP	instance="traefik:8080"	6.752s ago	

# Kubernetes Monitoring with Prometheus.

Using the main web interface, we can locate some traefik metrics (very few of them, because we don't have any Traefik frontends or backends configured for this example) and retrieve its values:



The screenshot shows the Prometheus web interface. At the top, the word "Prometheus" is displayed. Below it, there is a checkbox for "Enable query history". A text input field contains the query "traefik\_config\_last\_reload\_success". To the right of the input field, the following statistics are shown: "Load time: 58ms", "Resolution: 14s", and "Total time series: 1". Below the input field is a blue "Execute" button. Underneath the button is a dropdown menu with the text "- insert metric at cursor -". Below the dropdown are two tabs: "Graph" (which is active) and "Console". The "Graph" tab displays a table with two columns: "Element" and "Value". The table contains one row with the element "traefik\_config\_last\_reload\_success{instance='traefik:8080',job='traefik'}" and the value "1529654434". To the right of the table is a blue link that says "Remove Graph". At the bottom left of the interface is a blue "Add Graph" button.

Element	Value
traefik_config_last_reload_success{instance="traefik:8080",job="traefik"}	1529654434

# Kubernetes Monitoring with Prometheus.

## 2. Monitoring a service using Prometheus exporters

If an application does not natively expose Prometheus metrics, you need to bundle a [Prometheus exporter](#) in order to retrieve the state/logs/other metric formats of the main service and expose this information as Prometheus metrics.

You can deploy a pod containing the Redis server and a Prometheus sidecar container with the following command:

```
# Clone the repo if you don't have it already
git clone git@github.com:mateobur/prometheus-monitoring-guide.git
kubectl create -f prometheus-monitoring-guide/redis_prometheus_exporter.yaml
```

If you display the redis pod, you will notice it has two containers inside:

```
kubectl get pod redis-546f6c4c9c-lmf6z
```

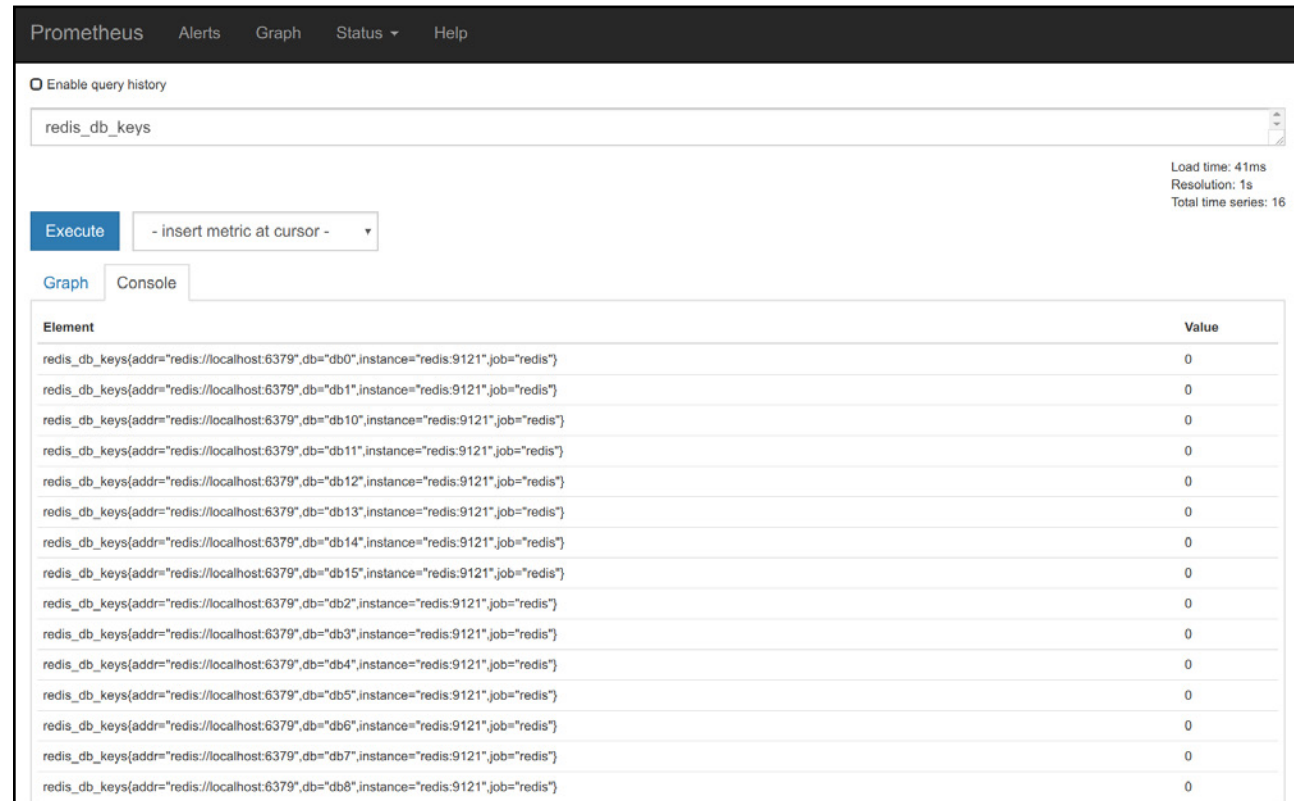
NAME	READY	STATUS	RESTARTS	AGE
redis-546f6c4c9c-lmf6z	2/2	Running	0	2m

Now, you just need to update the Prometheus configuration and reload like we did in the last section:

```
- job_name: 'redis'
  static_configs:
    - targets: ['redis:9121']
```

# Kubernetes Monitoring with Prometheus.

To obtain all the redis service metrics:



# Kubernetes Monitoring with Prometheus.

## Kubernetes Clusters

In addition to monitoring the services deployed in the cluster, you also want to monitor the Kubernetes cluster itself.

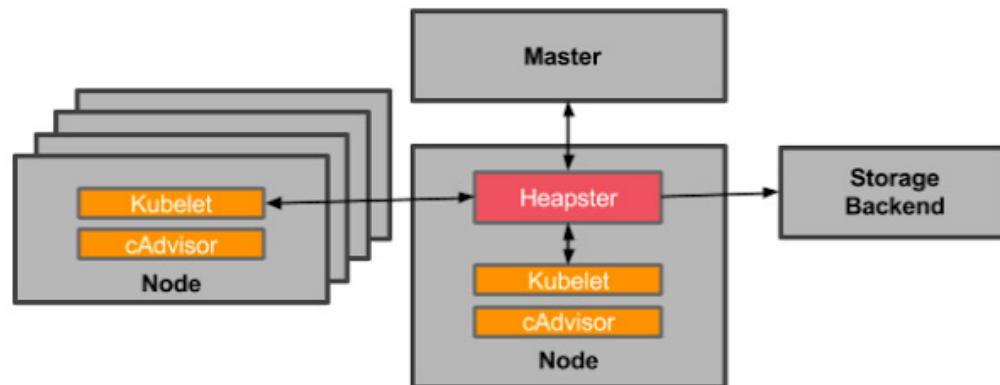
Three aspects of cluster monitoring to consider are:

1. The Kubernetes hosts (nodes) – classical sysadmin metrics such as CPU, load, disk, memory, etc.
2. Orchestration level metrics – Deployment state, resource requests, scheduling and API server latency, etc.
3. Internal kube-system components – Detailed service metrics for the scheduler, controller manager, dns service, etc.

The Kubernetes internal monitoring architecture has experienced some changes recently that we will try to summarize here, for more information you can read its [design proposal](#).

## Kubernetes Components

**Heapster:** Heapster monitors the Kubernetes cluster. It is a cluster-wide aggregator of monitoring and event data that runs as a pod in the cluster.





# Kubernetes Monitoring with Prometheus.

Apart from the Kubelets/cAdvisor endpoints, you can append additional metrics sources to Heapster like kube-state-metrics (see below).

Heapster is now **DEPRECATED**, its replacement is the metrics-server.

**cAdvisor:** [cAdvisor](#) is an open source container resource usage and performance analysis agent. It is purpose-built for containers and supports Docker containers natively. In Kubernetes, cAdvisor runs as part of the Kubelet binary, any aggregator retrieving node local and Docker metrics will directly scrape the Kubelet Prometheus endpoints.

**Kube-state-metrics:** [kube-state-metrics](#) is a simple service that listens to the Kubernetes API server and generates metrics about the state of the objects such as deployments, nodes and pods. It is important to note that kube-state-metrics is just a metrics endpoint, other entity needs to scrape it and provide long term storage (i.e. the Prometheus server).

**Metrics-server:** Metrics Server is a cluster-wide aggregator of resource usage data. It is intended to be the default Heapster replacement. Again, the metrics server will only present the last datapoints and it's not in charge of long term storage.

Thus:

- Kube-state metrics is focused on orchestration metadata: deployment, pod, replica status, etc.
- Metrics-server is focused on implementing the [resource metrics API](#): CPU, file descriptors, memory, request latencies, etc.

# Kubernetes Monitoring with Prometheus.

## Kubernetes nodes

The Kubernetes nodes or hosts need to be monitored, we have plenty of tools to monitor a Linux host. In this guide we are going to use the Prometheus [node-exporter](#):

- Its hosted by the Prometheus project itself
- Is the one that will be automatically deployed when we use the Prometheus operator in the next chapters
- Can be deployed as a DaemonSet and thus, will automatically scale if you add or remove nodes from your cluster.

You have several options to deploy this service, for example, using the DaemonSet in this repo:

```
kubectl create ns monitoring
kubectl create -f https://raw.githubusercontent.com/bakins/minikube-prometheus-demo/master/node-exporter-daemonset.yml
```

Or using Helm / Tiller:

If you want to use Helm, remember to create the RBAC roles and service accounts for the tiller component before proceeding.

```
helm init --service-account tiller
helm install --name node-exporter stable/prometheus-node-exporter
```

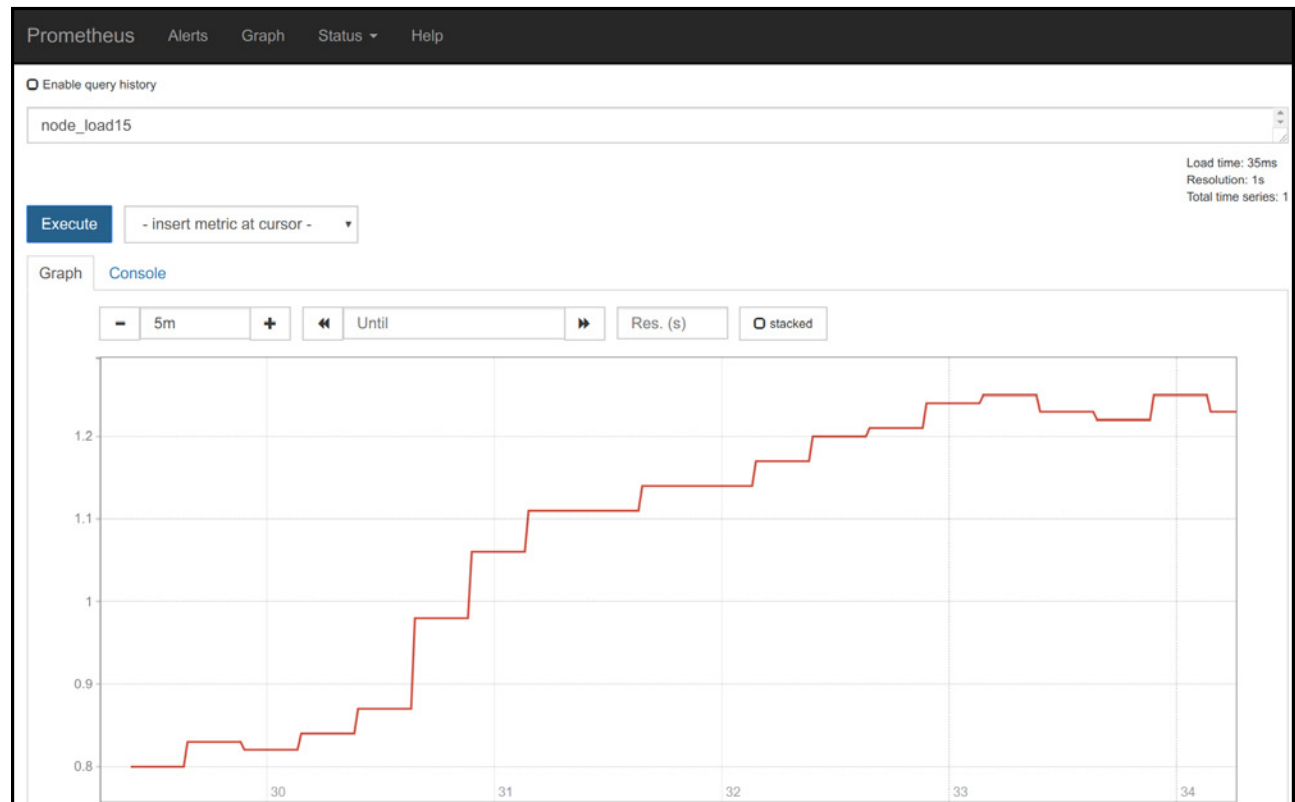
# Kubernetes Monitoring with Prometheus.

Once the chart is installed and running, you can display the service that you need to scrape:

```
kubectl get svc
```

NAME	Type	Cluster-IP	External-IP	Port(s)
node-exporter-prometheus-node-exporter	ClusterIP	10.101.57.207	<none>	9100/TCP

Once you add the scrape config like we did in the previous sections, you can start collecting and displaying the node metrics:



# Kubernetes Monitoring with Prometheus.

## Kube-state-metrics

Deploying and monitoring the kube-state-metrics is also a fairly straightforward task. You can deploy directly like in the example below or use a Helm chart.

```
git clone https://github.com/kubernetes/kube-state-metrics.git
kubectl apply -f kube-state-metrics/kubernetes/
...
kubectl get svc -n kube-system
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kube-dns	ClusterIP	10.96.0.10	<none>	53/UDP,53/TCP	13h
kube-state-metrics	ClusterIP	10.102.12.190	<none>	8080/TCP,8081/TCP	1h

Simply scrape that service (port 8080) in the Prometheus config. Remember to use the FQDN this time:

```
- job_name: 'kube-state-metrics'
  static_configs:
    - targets: ['kube-state-metrics.kube-system.svc.cluster.local:8080']
```

# Kubernetes Monitoring with Prometheus.

## Internal Components

There are several Kubernetes components including etcd, kube-scheduler or kube-controller that can expose its internal performance metrics using Prometheus.

Monitoring them is quite similar to monitoring any other Prometheus endpoint with two particularities:

- Which network interfaces are these processes using to listen, http scheme and security (HTTP, HTTPS, RBAC) depend on your deployment method and configuration templates.
  - Frequently, these services are only listening at localhost in the hosting node, making them difficult to reach from the Prometheus pods.
- These components may not have a Kubernetes service pointing to the pods, but you can always create it.

Depending on your deployment method and configuration, the Kubernetes services may be listening on the local host only, to make things easier on this example we are going to use [minikube](#).

Minikube let's you spawn a local single-node Kubernetes virtual machine in minutes.

This will work as well on your hosted cluster, GKE, AWS, etc., but you will need to reach the service port either by modifying the configuration and restarting the services or providing additional network routes.

Installing minikube is a fairly [straightforward process](#). First install the binary, next create a cluster that exposes the kube-scheduler service on all interfaces:

```
minikube start --memory=4096 --bootstrapper=kubeadm --extra-config=kubelet.authentication-token-webhook=true --extra-config=kubelet.authorization-mode=Webhook --extra-config=scheduler.address=0.0.0.0 --extra-config=controller-manager.address=0.0.0.0
```

# Kubernetes Monitoring with Prometheus.

Create a service that will point to the kube-scheduler pod:

```
kind: Service
apiVersion: v1
metadata:
  name: scheduler-service
  namespace: kube-system
spec:
  selector:
    component: kube-scheduler
  ports:
    - name: scheduler
      protocol: TCP
      port: 10251
      targetPort: 10251
```

Now you will be able to scrape the endpoint: `scheduler-service.kube-system.svc.cluster.local:10251`

# Kubernetes Monitoring with Prometheus.

## Conclusion

Prometheus is fast becoming one of the most popular Docker and Kubernetes monitoring tools to use. Having read this eBook, you're familiar with its benefits and possibly have already installed Prometheus and deployed it with end-user apps and Kubernetes cluster endpoints.

Ready to take the next step? Visit the [Sysdig blog](#) for information on additional components that are typically deployed together with the Prometheus service, including using the PromQL language to aggregate metrics, fire alerts and generate visualization dashboards.