

Kubernetes Monitoring Guide





Table of Contents

01	About This Guide	5
02	Intro to Kubernetes Monitoring	6
	Why is monitoring Kubernetes hard?	6
	Kubernetes increases infrastructure complexity.....	6
	Microservices architecture	6
	Cloud-native explosion and scale requirements	7
	It's hard to see what's inside containers	7
	Best practices for alerting on Kubernetes	8
	General alerting basics	8
	Use cases for Kubernetes monitoring	9
	Cluster Administrator: Monitoring Kubernetes clusters and nodes.....	9
	DevOps: Monitoring Kubernetes applications	9
	Monitoring Kubernetes deployments and pods.....	10
	Kubernetes monitoring tools.....	10
	cAdvisor	11
	Kubernetes metrics server	11
	Kubernetes Dashboard.....	12
	Kubernetes kube-state-metrics	12
	Kubernetes liveness and readiness probes.....	13
	Prometheus for Kubernetes monitoring	14
	Sysdig Monitor for Kubernetes.....	14
	Lessons learned.....	15

03**Monitoring Kubernetes with Golden Signals****16**

Golden Signals, a standard for Kubernetes application monitoring	16
Golden Signals metric: Latency explained	17
Golden Signals metric: Errors explained	18
Golden Signals metric: Traffic / connections explained	19
Golden Signals metric: Saturation explained	19
Golden Signals vs. RED method vs. USE method in Kubernetes	20
Golden Signal Metrics Instrumentation in Kubernetes	21
Instrumenting code with Prometheus metrics / custom metrics	21
Sysdig eBPF system call visibility (no instrumentation)	21
A practical example of Golden Signals in Kubernetes	22
Alerting on application layer metrics	29
Caveats and gotchas of Golden Signals in Kubernetes	30
Lessons learned	30

04**Monitoring Kubernetes Infrastructure and Core Components****31**

Monitoring Kubernetes Infrastructure	31
Alerting on the host or Kubernetes node layer	31
Control Plane	36
Monitoring Kubernetes control plane in Sysdig Monitor	36
How to monitor the Kubernetes API server	37
How to monitor Kubelet	45
How to monitor Controller Manager	56
How to monitor etcd	62
Alerting on the Kubernetes control plane	71
Is Kubernetes etcd running?	71
Is the Kubernetes API server running?	72
Is the latency of Kubelet too high for the start of the pods?	72
Lessons learned	73

05**Monitoring Kubernetes Workloads****74**

Monitoring services running on Kubernetes	74
Kubernetes workloads hierarchy	75
Alerting on services running on Kubernetes	77
Understanding Kubernetes limits and requests by example	79
Resources	80
Namespace quotas	80
Explaining pod requests and limits	81
Choosing pragmatic requests and limits	87
Reducing wasted spending	89
Cost monitoring and optimization	90
Lessons learned	91
How to troubleshoot Kubernetes OOM and CPU Throttle	92
Kubernetes OOM problems	92
OOM kill due to container limit reached	92
Kubernetes OOM kill due to limit overcommit	94
CPU throttling due to CPU limit	94
Troubleshooting OOM Kill and CPU Throttling with Sysdig Monitor	96
Lessons learned	98

06**Conclusion****99**

About This Guide

With over 30,000 stars on GitHub, over 400 contributors, and an ecosystem that includes Google, Red Hat, Intel, and more, Kubernetes has taken the container ecosystem by storm. It's with good reason, too; Kubernetes acts as the brain for your distributed container deployment. It's designed to manage service-oriented applications using containers distributed across clusters of hosts. Kubernetes provides mechanisms for application deployment, service discovery, scheduling, updating, maintenance, and scaling. But what about monitoring Kubernetes environments?

While Kubernetes has the potential to dramatically simplify the act of deploying your application in containers – and across clouds – it also adds a new set of complexities for your day-to-day tasks of managing application performance, gaining visibility into services, and your typical monitoring > alerting > troubleshooting workflow.

New layers of infrastructure complexity are appearing in the hopes of simplifying and automating application deployments including dynamic provisioning via IaaS, automated configuration with configuration management tools, and lately, orchestration platforms like Kubernetes, which sit between your bare metal or virtual infrastructure and the services that empower your applications.

In addition to increased infrastructure complexity, applications are now being designed for microservices, where an order of magnitude more, multiple components are communicating with each other. Each service can be distributed across multiple instances and containers move across your infrastructure as needed. We used to know how many instances we had of each service component and where they were located, but that's no longer the case. How does this affect Kubernetes monitoring methodology and tooling? As described in [Site Reliability Engineering – How Google Runs Production Systems](#), “we need monitoring systems that allow us to alert for high-level service objectives, but retain the granularity to inspect individual components as needed.”

Whether you are a Sysdig customer or not, you will find valuable information in this guide. We have presented information and best practices that apply broadly to most environments. You will learn:

- The basics of Kubernetes monitoring
- How to use Golden Signals
- How to monitor Kubernetes infrastructure
- How to monitor Kubernetes workloads

Throughout the guide we will also provide useful alerts that can be used to notify you when something is not quite right. This guide will help you navigate the complexities of monitoring Kubernetes workloads and show you detailed steps you can take to make sure you have the visibility you need to be successful!

Intro to Kubernetes Monitoring

Monitoring Kubernetes, both the infrastructure platform and the running workloads, is on everyone's checklist as we evolve beyond day zero and head into production. Traditional monitoring tools and processes aren't adequate, as they don't provide visibility into dynamic container environments. Given this, let's take a look at what tools you can use to monitor Kubernetes and your applications.

Why is monitoring Kubernetes hard?

Legacy monitoring tools, collecting metrics from static targets built for monitoring servers that each had their own name, and services that didn't change overnight and worked in the past but won't work well today. This is why those tools fail at monitoring Kubernetes:

Kubernetes increases infrastructure complexity

New layers of infrastructure complexity are appearing in the hopes of simplifying application deployments: dynamic provisioning via IaaS, automated configuration with configuration management tools, and lately, orchestration platforms like Kubernetes, which sit between your bare metal or virtual infrastructure and the services that empower your applications. This is why monitoring the Kubernetes health at the control plane is part of the job.

In addition, Kubernetes has several specific entities with a hierarchy that relates everything. This hierarchy includes many elements linked together that need to be assimilated by the monitoring system. There is no way of monitoring Kubernetes without a good way of reflecting the structure of these objects.

Microservices architecture

In addition to increased infrastructure complexity, new applications are being designed for microservices, where the number of components communicating with each other has increased by an order of magnitude. Each service can be distributed across multiple instances, and containers move across your infrastructure as needed. Monitoring the Kubernetes orchestration state is key to understanding if Kubernetes is doing its job. Trust, but verify, that all the instances of your service are up and running.

Cloud-native explosion and scale requirements

While we adopt cloud native architectures, the changes that they bring carry away an increased amount of smaller components. How does this affect Kubernetes monitoring methodology and tooling? As described on [Site Reliability Engineering – How Google Runs Production Systems](#), “we need monitoring systems that allow us to alert for high-level service objectives, but retain the granularity to inspect individual components as needed.”

The number of metrics simply explodes, and traditional monitoring systems just can't keep up. While we used to know how many instances we had of each service component and where they were located, that's no longer the case. Now, metrics have high cardinality which means they have much more data to store and analyze. Kubernetes adds multidimensional levels like cluster, node, namespace, or service so the different aggregations, or perspectives, that need to be controlled can explode; many labels represent attributes from the logical groups of the microservices, to application version, API endpoint, specific resources or actions, and more.

The containers don't last forever. In our latest [container usage report](#), we found that 23% of the containers live for 10 seconds or less while 72% of them live fewer than five minutes. This creates a high level of churn. Labels like the container id or the pod name change all the time, so we stop seeing old ones while we need to deal with the new.

Now, take all of the combinations of metric names and labels together with the actual value and the timestamp. Over a small period of time, you have thousands of data points, and even in a small size Kubernetes cluster, this will generate hundreds of thousands of time series. This can be millions for a medium-sized infrastructure. This is why Kubernetes monitoring tools need to be ready to scale to hundreds of thousands of metrics.

It's hard to see what's inside containers

Containers are ephemeral. Once the container dies, everything inside is gone. You may not SSH to the container (depending on the container and security policies). Looking at logs, it might be difficult if you try to get data from older instantiations than the most recent one, and most of the tools you are accustomed to using for troubleshooting are not installed. Containers are great for operations as we can package and isolate applications to consistently deploy them everywhere, but at the same time, this makes them black boxes which are hard to troubleshoot. This is why monitoring tools that provide granular visibility through system calls tracing give you visibility down to every process, file, or network connection that happens inside a container to troubleshoot issues faster.

With these considerations in mind, you can now better understand why monitoring Kubernetes is very different from monitoring servers, VMs, or even cloud instances.

Best practices for alerting on Kubernetes

Effective alerting is at the bedrock of a monitoring strategy. Naturally, with the shift to orchestrated container environments and Kubernetes, your alerting strategy will need to evolve as well. This is due to a few core reasons:

- **New infrastructure layers:** Between your services and the host, you have a new layer consisting of the containers and the container orchestrator. These are new internal services that you need to monitor, and your alerting system must be aware of them.
- **Microservices architecture:** Containers aren't coupled with nodes like services were before, so traditional monitoring doesn't work effectively. There is no static number of service instances running (think of a canary deployment or auto-scaling setup). It's fine that a process is being killed in one node because, chances are, it's being rescheduled somewhere else in your infrastructure.
- **New scale and aggregation requirements:** With services spread across multiple containers, monitoring system level and service-specific metrics for those, plus all of the new services that Kubernetes brings in, does your monitoring and alerting system have the ability to ingest all of these metrics at a large scale? You also need to look at the metrics from different perspectives. If you automatically tag metrics with the different labels existing in Kubernetes and our monitoring system understands Kubernetes metadata, you can aggregate or segment metrics as required in each situation.
- **Lack of visibility:** Containers are black boxes. Traditional tools can only check against public monitoring endpoints. If you want to deeply monitor the service in question, you need to be able to look at what's happening inside the containers.

With these issues in mind, let's go through the best practices for alerting on Kubernetes environments.

General alerting basics

Let's first define a set of basic rules you should be following when you set up your alerts in order to improve the efficiency and mental health of your on-call rotation.

- Alerts should be based on symptoms. If an unexpected value in a metric doesn't have any appreciable impact, it shouldn't be an alert. You can check those values and generate maintenance tasks, but don't wake anybody up in the middle of the night!
- Alerts should be actionable. Triggering an alert when there's nothing to do about it will only generate frustration.
- There are several methodologies, like [Chapter 3: Monitoring Kubernetes with Golden Signals](#), that allow a standardization of the way you alert, making the dashboards and alerts much easier to understand.

Use cases for Kubernetes monitoring

Now, you might be wondering, why are we doing this after all? A Kubernetes cluster has multiple components and layers, and across each of them you will find different failure points that you need to monitor. Those are some typical profiles for Kubernetes monitoring:

- **Cluster administrators:** Infrastructure operations teams take care of the cluster's health. This includes monitoring the hosts, Kubernetes components, networking components, capacity planning, and external resources like volumes or load balancers.
- **DevOps:** Responsible for the health of applications running in the cluster. This includes workloads, pods, volume data, and images running. They usually have to monitor application-related metrics, like Golden Signals or custom business metrics.

This is an oversimplification, as there are plenty of gray areas between these two roles and many times issues are interconnected. What if a pod without limits is exhausting the memory in a node? What if an application is giving error messages due to problems in the cluster DNS? Even when there are specific points of responsibility, collaborating and sharing a monitoring environment can provide a lot of advantages like saving time and resources.

In addition, saying who is responsible for detecting an issue is more related to the capacity to understand the symptoms of the problem and not the solution. A problem can appear in the applications but can be caused by a problem in the infrastructure, and the other way around. The person that detects the issue is not necessarily the one to fix it.

Cluster Administrator: Monitoring Kubernetes clusters and nodes

By monitoring the cluster, you get an across-the-board view of the overall platform health and capacity. Specific use cases can be:

- Cluster resource usage: Is the cluster infrastructure underutilized? Are you over capacity?
- Project and team chargeback: What is each project or team resource usage?
- Node availability and health: Are enough nodes available to replicate our applications? Are you going to run out of resources?

DevOps: Monitoring Kubernetes applications

At the end of the day, your applications are what matter most. What is it that you want to look at here? This is the part which is similar to what you may be used to:

- Application availability: Is the application responding?
- Application health and performance: How many requests do you have? What's the responsiveness or latency? Are you having any errors?
- How healthy are the deployments and services that support your application?

[Monitoring using Golden Signals](#) is considered the best practice approach on this. We will cover this in [Chapter 3: Monitoring Kubernetes with Golden Signals](#).

Monitoring Kubernetes deployments and pods

This is a gray area that is mostly the responsibility of DevOps teams, but sometimes the cluster admins can have a key role in finding the source of the problem.

Looking at the Kubernetes constructs like namespaces, deployments, ReplicaSets, or DaemonSets, we can understand whether our applications have been properly deployed. For example:

- Missing and failed pods: Are the pods running for each of our applications? How many pods are dying?
- Running vs. desired instances: How many instances for each service are actually ready? How many do you expect to be ready?
- Pod resource usage against requests and limits: Are CPU and memory requests and limits set? What is the actual usage against those?

Kubernetes monitoring tools

Like most platforms, Kubernetes has a set of rudimentary tools that allow you to monitor your platform, including the Kubernetes constructs that ride on top of physical infrastructure. The term “built-in” may be a little bit of an overstatement. It’s more fair to say that, given the extensible nature of Kubernetes, it’s possible for your inner tinkerer to add additional components to gain visibility into Kubernetes. As you will see in the following sections, some of these monitoring components — like Prometheus, Grafana and kube-state-metrics — are not required anymore while using Sysdig Monitor. These are included out-of-the-box with Sysdig Monitor. Let’s run through the typical pieces of a Kubernetes monitoring setup:

- cAdvisor
- Kubernetes metrics server
- Kubernetes Dashboard
- Kubernetes kube-state-metrics
- Kubernetes liveness and readiness probes
- Prometheus with Grafana
- Sysdig Monitor for Kubernetes

cAdvisor

[cAdvisor](#) is an open source container resource usage collector. It is purposefully built for containers and supports Docker containers natively. Also, it auto-discovers all containers in the given node and collects CPU, memory, filesystem, and network usage statistics. However, cAdvisor is limited in a couple of ways:

- It only collects basic resource utilization since cAdvisor cannot tell you how your applications are actually performing, but only if a container has X% CPU utilization (for example).
- cAdvisor itself doesn't offer any long-term storage or analysis capabilities.

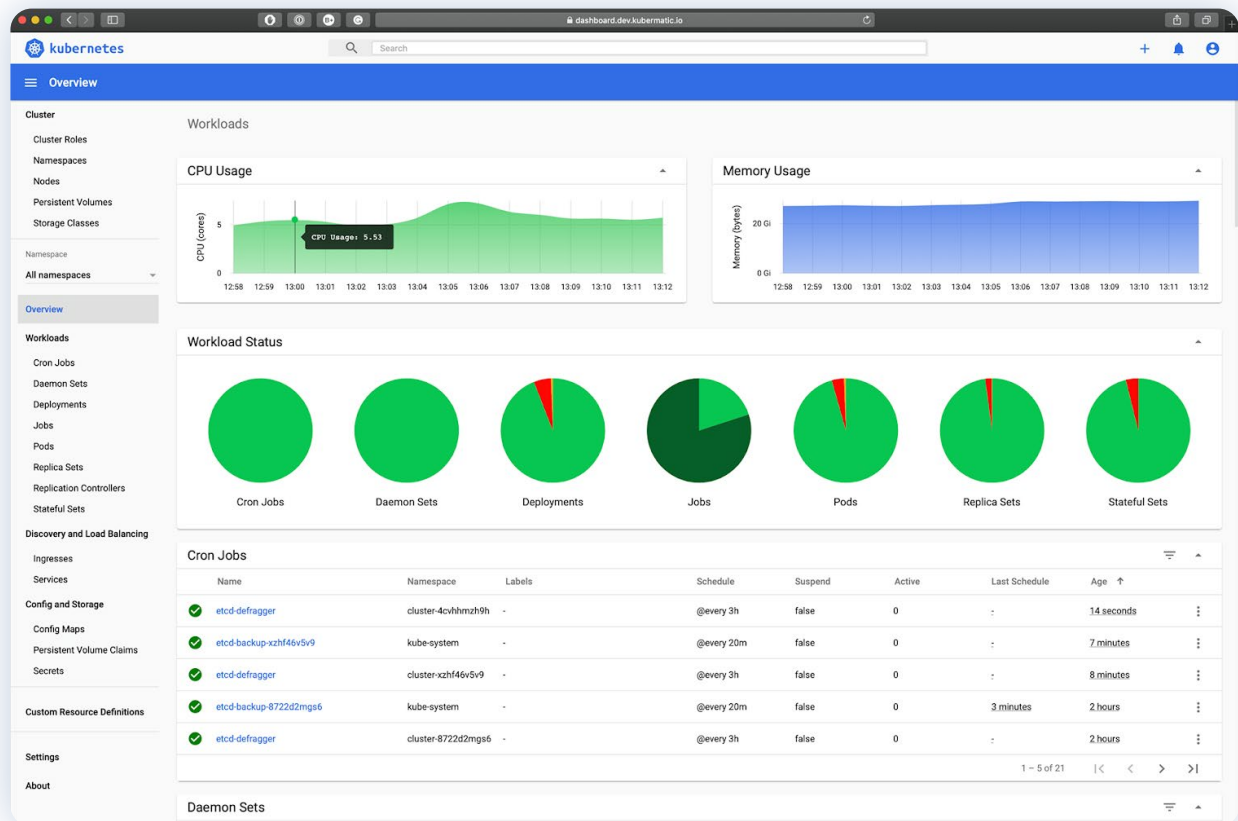
On Kubernetes, the nodes' kubelets, the on-machine Kubernetes agent, install cAdvisor to monitor the resources of the containers inside of each pod. But to go further with this data, you need something that aggregates data across the entire cluster. The most popular option [used to be Heapster](#), but that has been deprecated and Kubernetes consumes metrics through the metrics-server. This data then needs to be pushed to a configurable backend for storage and visualization. Supported backends include InfluxDB, Google Cloud Monitoring, and a few others. Additionally, you must add a visualization layer like Grafana to see your data.

Kubernetes metrics server

Starting from Kubernetes 1.8, the resource usage metrics coming from the kubelets and cAdvisor are available through the [Kubernetes metrics server](#) API, the same way Kubernetes API is exposed. This service doesn't allow you to store values over time either, and lacks visualization or analytics. Kubernetes metrics server is used for Kubernetes advanced orchestration, like [Horizontal Pod Autoscaler for autoscaling](#) and [Vertical Pod Autoscaler](#)

Kubernetes Dashboard

The [Kubernetes Dashboard](#) provides a simple way to see your environment browsing through your Kubernetes namespaces and workloads. Kubernetes Dashboard gives you a consistent way to visualize some of this basic data with only basic CPU / memory data available. You can also manage and take actions from this dashboard, which has been a security concern on multi-tenant clusters as proper RBAC privileges need to be set up.



Kubernetes kube-state-metrics

Another component that you definitely want to consider is [kube-state-metrics](#). It's an add-on service that runs alongside your Kubernetes metrics-server that polls the Kubernetes API and translates characteristics about your Kubernetes constructs into metrics. Some questions kube-state-metrics would answer are:

- How many replicas did you schedule? And how many are currently available?
- How many pods are running / stopped / terminated?
- How many times has this pod restarted?

In general, the model is to take Kubernetes events and convert them to metrics. Kube-state-metrics uses [client-go](#), the [compatibility matrix](#) for client-go, and Kubernetes determines which Kubernetes versions are required for kube-state-metrics.

This at least gives you a sense of the steps you'd take to build reasonable monitoring for your Kubernetes environment. You still wouldn't have detailed application monitoring ("What's the response time for my database service?"), but you could see your underlying hosts, Kubernetes nodes, and some monitoring of the state of your Kubernetes abstractions.

Kubernetes liveness and readiness probes

Kubernetes probes perform the important function of regularly monitoring the health or availability of a pod. Liveness probe is a mechanism that allows the Kubelet to be aware of a deadlock, or any other situation preventing the application from making progress. If the liveness probe fails after matching its own criteria, the Kubelet will restart the container. Kubernetes monitoring probes allow you to arbitrarily define "Liveness" through a request against an endpoint or running a command. Below is a simple example of a liveness probe based on running a cat command:

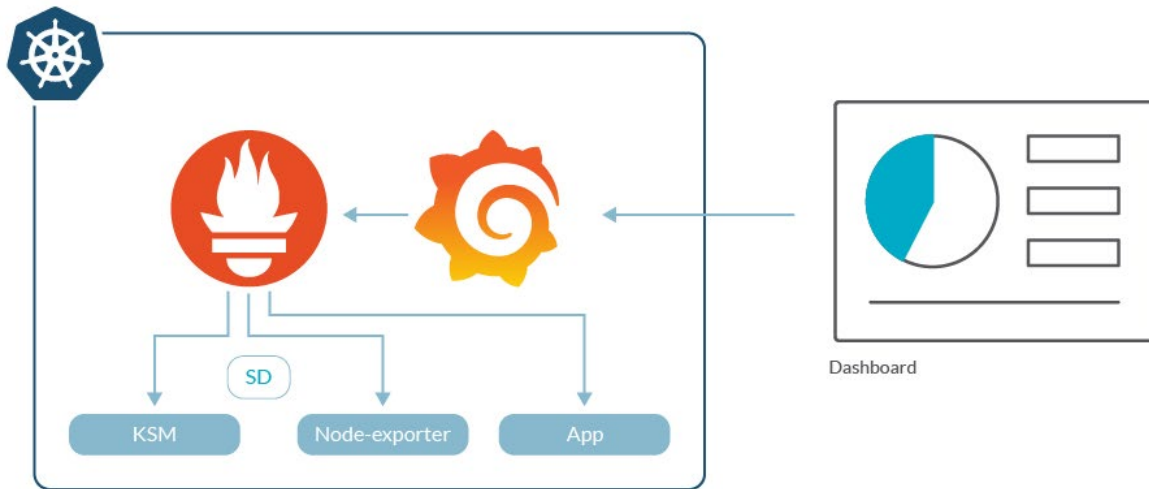
```
#Example Liveness probe for the Sysdig Blog on "Monitoring Kubernetes"
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
    name: liveness-exec
spec:
  containers:
  - name: liveness
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    image: gcr.io/google_containers/busybox
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5

#source https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/
```

The same way a liveness probe checks regularly if a container is alive, readiness probes are responsible for checking and validating when a container is ready to accept traffic. Readiness probes are configured similarly to liveness probes, just replace `livenessProbe` with `readinessProbe`.

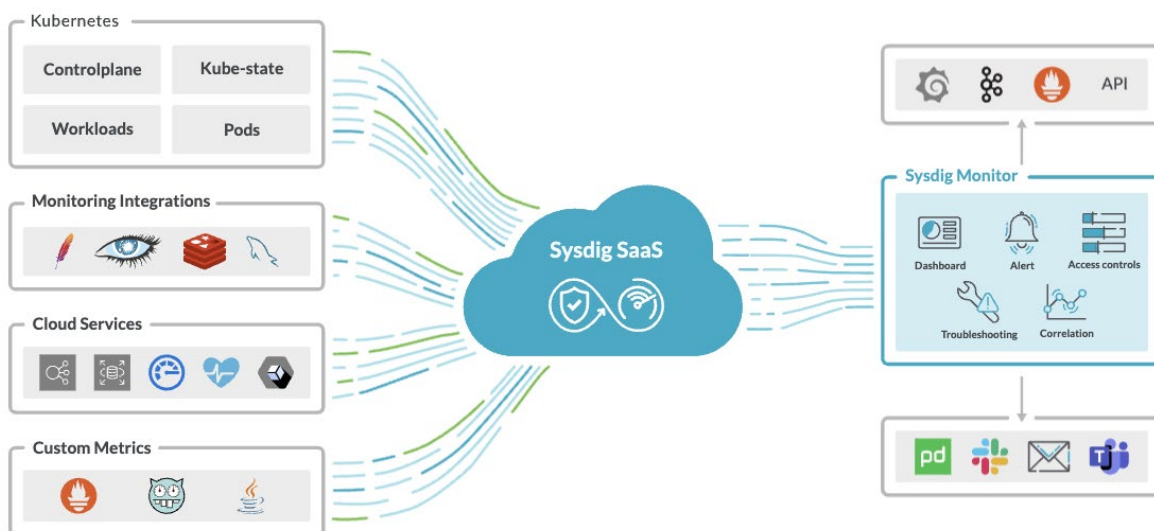
Prometheus for Kubernetes monitoring

Prometheus is a time series database, open source, and, like Kubernetes, is a CNCF project. But Prometheus monitoring is an entire monitoring stack around the Prometheus server that collects and stores the metrics. This includes Grafana for dashboarding and often a number of exporters, including small sidecar containers that transform services metrics into [Prometheus metrics](#) format.



Prometheus is the de facto approach for monitoring Kubernetes. While it is really easy to start monitoring Kubernetes with Prometheus, DevOps teams quickly realize that Prometheus has some roadblocks like scale, RBAC, and teams support for compliance.

Sysdig Monitor for Kubernetes



Sysdig Monitor and Sysdig backend are able to store and query Prometheus native metrics and labels. Additionally, users can use Sysdig in the same way that they use Prometheus. You can leverage Prometheus Query Language queries for dashboards and alerts, or the Prometheus API for scripted queries, as part of a DevOps workflow. This way, your investments in the Prometheus ecosystem can be preserved and extended while improving scale and security. Users can take advantage of the Sysdig's Advisor already included in Sysdig Monitor to troubleshoot issues and correlate them with other events. In addition, thanks to the Sysdig agent running on the hosts, users can also capture live activity data from containers (files, processes, syscalls, network data, etc.), and analyze it with Sysdig Inspect.

Sysdig Monitor is a complete Prometheus managed service that helps users with monitoring Kubernetes environments and troubleshooting issues.

Lessons learned

1. If you have a non-trivial deployment, you must start thinking about monitoring Kubernetes in a way that naturally fits with your orchestrated environment.
2. The de facto standard for Kubernetes monitoring is built up from a number of open source tools like cAdvisor, Kubernetes metrics server, kube-state-metrics, and Prometheus.
3. When monitoring production environments, a commercial tool like Sysdig can provide an opinionated workflow and supported experience for monitoring Kubernetes while remaining compatible with Prometheus monitoring. And if you are running large scale environments, we have you covered thanks to our Prometheus scale capabilities. In addition, Sysdig provides other great tools - adding a huge extra value - allowing you to troubleshoot and analyze data in almost real time.

Monitoring Kubernetes with Golden Signals

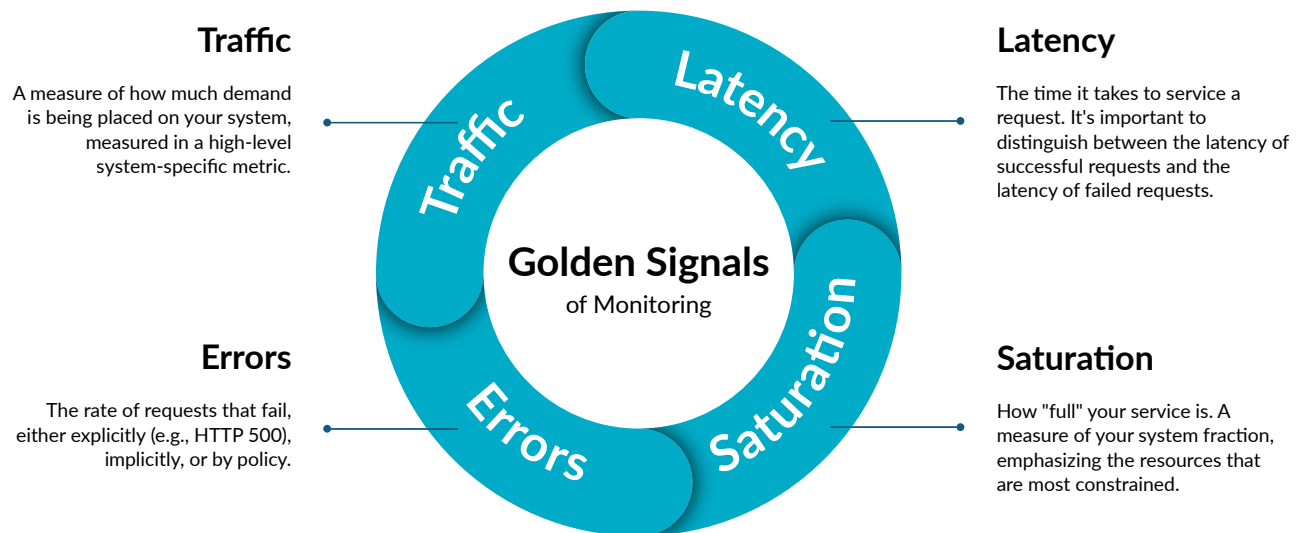
What are Golden Signal metrics? How do you monitor Golden Signals in Kubernetes applications? Golden Signals can help detect issues of a microservices application. These signals are a reduced set of metrics that offer a wide view of a service from a user or consumer perspective, so you can detect potential problems that might be directly affecting the behavior of the application.

Golden Signals, a standard for Kubernetes application monitoring

Congratulations, you have successfully deployed your application in Kubernetes. This is the moment you discover your old monitoring tools are pretty much useless in a modern and scalable environment like Kubernetes, and that you're unable to detect potential problems which look completely different than they do in a more traditional environment. Classic monitoring tools are usually based on static configuration files and were designed to monitor machines, not microservices or containers. But in the container world, things change fast. Containers are created and destroyed at an incredible pace and it's impossible to catch up without specific service discovery functions. Remember, according to the latest [Sysdig Container Usage Report](#), most containers live fewer than five minutes.

Most of the modern monitoring systems offer a huge variety of metrics for many different purposes. It's quite easy to drown in metrics and lose focus on what really is relevant for your application. Setting too many irrelevant alerts can drive you to a permanent emergency status and "alert burn out." Imagine a node that is being heavily used and raising load alerts all the time. You're not doing anything about that as long as the services in the node still work. Having too many alerts is as bad as not having any because important alerts will drown in a sea of irrelevance.

This is a problem that many people have faced. Fortunately, it has already been solved. The answer is the four Golden Signals, a term used first in the [Google SRE handbook](#). Golden Signals are four metrics that will give you a great idea of the real health and performance of your application, as seen by the actors interacting with that service.



Golden Signals metric: Latency explained

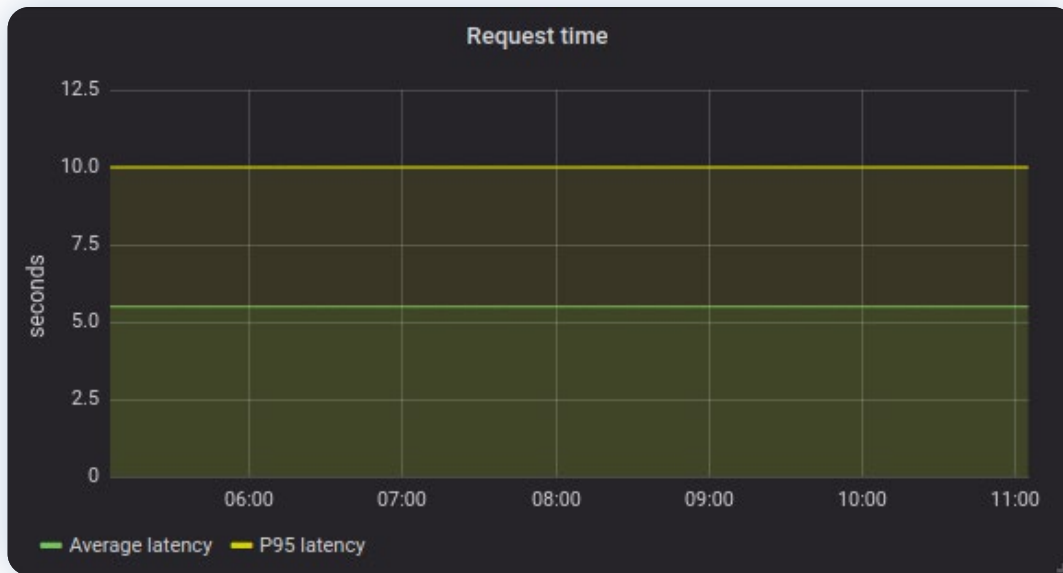
Latency is the time your system takes to serve a request against the service. This is an important sign used to detect a performance degradation problem.

When using latency, it's not enough to use average values since they can be misleading. For example, let's say you have a service that shows an average of 100ms of response time. With only this information, you might consider it pretty good, but the feedback of the users is that it's perceived as slow.

The answer to this contradiction can be discovered using different statistical parameters, like standard deviation, that will give us an idea of the dispersion of the latency values. What if you have two kinds of requests, one very fast and the other slow, because it is more database intensive. If a typical user interaction has one slow request and 10 fast ones, the mean will probably be pretty low, but the application will be slow. Bottleneck analysis is important too, not only mean values.

Histogram metrics are a great tool to avoid this behavior. These indicate the number of requests under different latency thresholds and allow them to aggregate in percentiles. A percentile is a value below which a given percentage of measures falls. For example, p99 says that 99% of my requests have a lower latency value than the percentile.

As you can see in the screenshot below, average latency is acceptable. But if you look at the percentile, you see a lot of dispersion in the values, giving a better idea of what the real latency perception is. Different percentiles express different information; p50 usually expresses general performance degradation and p95, or p99, allows detection of performance issues in specific requests or components of the system.



It may think that a high latency in 1% of the requests is not a big issue, but consider a web application that needs several requests to be fully loaded and displayed. In this common scenario, a high latency in 1% of the requests can affect a much higher rate of final users, because one of these multiple requests is slowing down the performance of the whole application.

Another useful tool to analyze latency values can be the [APDEX score](#) that, given your SLA terms, can provide a general idea of how good your system condition is based on percentiles.

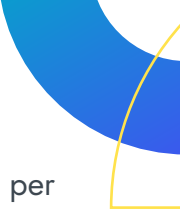
Golden Signals metric: Errors explained

The rate of errors returned by your service is a very good indicator of deeper issues. It's very important to not only detect explicit errors, but implicit errors too.

An explicit error would be any kind of HTTP error code. These are pretty easy to identify as the error code is easily obtained from the reply headers, and they are pretty consistent throughout many systems. Some examples of these errors could be authorization error (503), content not found (404), or server error (500). Error description can be very specific in some cases ([418 - I'm a teapot](#)).

On the other hand, implicit errors can be trickier to detect. How about a request with HTTP response code 200 but with an error message in the content? Different policy violations should be considered as errors too:

- Errors that do not generate HTTP reply, as a request that took longer than the timeout.
- Content error in an apparently successful request.



When using dashboards to analyze HTTP errors, mean values or percentiles don't make sense. In order to properly see the impact of errors, use rates. The percentage of requests that end in errors per second can give detailed information about when the system started to fail and with what impact.

Besides the HTTP errors that we can monitor and analyze in our infrastructure, there are other types of errors that are worth mentioning and, most importantly, detecting and tracking in your Kubernetes environment:

- **CrashLoopBackOff:** A [Kubernetes state representing a restart loop](#) in a Pod. A container running in a Pod is crashing and being restarted permanently.
- **OOM Kill:** A container tries to allocate more memory than allowed, reaching the memory limit. This container is [killed and restarted by Kubernetes](#).
- **CPU Throttling:** This will never be considered as an explicit error but it could rank well among the implicit errors. CPU usage reaches the [limit set for a specific workload](#), and Kubernetes automatically slows down the CPU to consume fewer resources and avoid surpassing the limit. This approach will slow down the response time of the application, likely causing errors in the service itself.
- **Image Pull Error:** A container is failing to start as it [cannot pull the image](#).
- **Liveness and Readiness errors:** These errors can happen during the start-up phase (Readiness) if the container is not ready just before accepting connections, or during the runtime if the container is not responsive when Liveness probe runs.

Golden Signals metric: Traffic / connections explained

Traffic or connections is an indicator of the amount of use of your service per time unit. It can have many different values depending on the nature of the system, like the number of requests to an API or the bandwidth consumed by a streaming app. It can be useful to group the traffic indicators depending on different parameters, like response code or related to business logic.

When talking about Kubernetes, the traffic term can apply to different layers, like Ingress traffic, Egress traffic, or even the internal communications within the cluster through the CNI plugin of your choice. Ensure you properly monitor traffic from all the different layers in your Kubernetes cluster.

Golden Signals metric: Saturation explained

This metric should be the answer to the following question: How full is my system?

Usually, saturation is expressed as a percentage of the maximum capacity, but each system will have different ways to measure saturation. The percentage could mean the number of users, or requests, obtained directly from the application or based upon estimations. Often, saturation is derived from system metrics, like CPU or memory, so they don't rely on instrumentation and are collected directly from the system using different methods, like Prometheus node-exporter. Obtaining system metrics from a Kubernetes node is essentially the same as with any other system. At the end of the day, they are Linux machines.

It's important to choose the adequate metrics and use as few as possible. The key to successfully measuring saturation is to choose the metrics that are constraining the performance of the system. If your application is processor intensive, use CPU load. If it's memory intensive, choose used memory. The process of choosing saturation metrics is often a good exercise to detect bottlenecks in the application.

You should set alerts in order to detect saturation with some margin because usually, the performance drastically falls when saturation exceeds 80%.

Golden Signals vs. RED method vs. USE method in Kubernetes

There are several approaches to design an efficient monitoring system for an application, but commonly, they are based on the four Golden Signals. Some of them, like the RED method, give more importance to organic metrics, like requests rate, errors, and latency. Others, like the USE method, focus on system level metrics and low-level values like use of the CPU, memory, and I/O. When do you need to use each approach?

RED method is focused on parameters of the application, without considering the infrastructure that runs the applications. It's an external view of the service, how the clients see the service. Golden Signals try to add the infra component by adding the saturation value, which will be necessarily implied from system metrics. This way, we have a deeper view, as every service is unavoidably tied to the infrastructure running it. Maybe an external view is fine, but saturation will give you an idea of "how far" the service is from a failure.

USE method puts the emphasis on the utilization of resources, including errors in the requests as the only external indicator of problems. This method could overlook issues that affect some parts of the service. What if the database is slow due to a bad query optimization? That would increase latency but would not be noticeable in saturation. Golden Signals try to get the best of both methods, including external observable and system parameters.

Having said this, all of these methods have a common point — they try to homogenize and simplify complex systems in an effort to make incident detection easier. If you're capable of detecting an issue with a smaller set of metrics, the process of scaling your monitoring to a big number of systems will be almost trivial.

As a positive side effect, reducing the number of metrics involved in incident detection helps to diminish alert fatigue due to arbitrary alerts set on metrics that will undoubtedly become a real issue or do not have a clear direct action path.

As a weakness, any simplification will remove details in the information received. It's important to note that, despite Golden Signals being a good way to detect ongoing or future problems, once the problem is detected, the investigation process will require the use of different inputs to be able to dig deeper into the root cause of the problem. Any tool at hand can be useful for the troubleshooting process, like logs, custom metrics, or different metric aggregation. For example, separate latency per deployment.

Golden Signal Metrics Instrumentation in Kubernetes

Instrumenting code with Prometheus metrics / custom metrics

In order to get Golden Signals with Prometheus, code changes (instrumentation) will be required. This topic is quite vast and addressed in our blog, [Prometheus metrics / OpenMetrics code instrumentation](#).

Prometheus has been positioned as a *de facto* standard for metric collecting, so most of the languages have a library to implement custom metrics in your application in a more convenient way. Nevertheless, instrumenting custom metrics requires a deep understanding of what the application does.

A poorly implemented code instrumentation can end up with time series cardinality bombing and a real chance to collapse your metrics collection systems. Using request id as a label, for example, generates a time series per request (seen in a real use case). Obviously, this is something you don't want in your monitoring system as it increases resources needed to collect the information and can potentially cause downtimes. Choosing the correct aggregation can be key to a successful monitoring approach.

Sysdig eBPF system call visibility (no instrumentation)

Sysdig monitor uses eBPF protocol to gather information of all the system calls directly from the kernel. This way, your application doesn't need any modification in the code or at container runtime. What is running in your nodes is the exact container you built, with the exact version of the libraries and your code (or binaries) intact.

System calls can give information about the processes running, memory allocation, network connections, access to the filesystem, and resource usage, among other things. With this information, it's possible to obtain meaningful metrics that will provide additional information about what is happening in your systems.

Sysdig agent collects the System calls and stores its data in the form of metrics for you. This way, you can correlate all this valuable information received directly from the kernel with Golden Signals, the Kubernetes internal components, or even your own workloads.

Golden Signals are some of the metrics available out-of-the-box with Sysdig monitor, providing latency, requests rate, errors, and saturation with a special added value that all of these metrics are correlated with the information collected from the Kubernetes API. This correlation allows you to do meaningful aggregations and represent the information using multiple dimensions:

- Group latency by node > This will provide information about different problems with your Kubernetes infrastructure.
- Group latency by deployment > This allows to track problems in different microservices or applications.
- Group latency by pod > Perhaps a pod in your deployment is unhealthy.
- Group latency by container > This will give you information about problems in any of your containers.

These different levels of aggregation allow us to slice our data and locate issues, helping with troubleshooting tasks by digging into the different levels of the Kubernetes entities, from cluster, to node, to deployment, to pod, and then to container.

In a new level of abstraction, with Sysdig Monitor, thanks to the correlation between System calls and the Kubernetes API metrics, monitoring data can be represented and grouped by workloads. You might prefer, or it just might be easier for you, to spot issues identifying which workload is involved in some specific problem.

As mentioned earlier, with Sysdig Monitor you can group latencies using different dimensions, but there is much more. Not only latencies, but errors, traffic, and saturation can be aggregated and provide data the same way. Monitor your services and workloads, your Kubernetes or cloud infrastructure, and troubleshoot issues, from top to bottom. It's something you can easily do in a few steps with the out-of-the-box dashboards and tools that Sysdig Monitor provides.

A practical example of Golden Signals in Kubernetes

As an example, to illustrate the use of Golden Signals, let's say you have deployed a simple Go application example with Prometheus instrumentation. This application will apply a random delay between zero and 12 seconds in order to give usable information of latency. Traffic will be generated with curl, with several infinite loops.

If you have included a [histogram](#) to collect metrics related to latency and requests, these metrics will help us obtain the first three Golden Signals: latency, request rate, and error rate. You can obtain saturation directly with Prometheus and node-exporter, using the percentage of CPU in the nodes in this example.

```
package main

import (
    "fmt"
    "log"
    "math/rand"
    "net/http"
    "time"

    "github.com/gorilla/mux"
    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

func main() {
    //Prometheus: Histogram to collect required metrics
    histogram := prometheus.NewHistogramVec(prometheus.HistogramOpts{
        Name: "greeting_seconds",
        Help: "Time take to greet someone",
        Buckets: []float64{1, 2, 5, 6, 10}, //Defining small buckets as this app
should not take more than 1 sec to respond
    }, []string{"code"}) //This will be partitioned by the HTTP code.

    router := mux.NewRouter()
    router.Handle("/sayhello/{name}", Sayhello(histogram))
}
```

```

router.Handle("/metrics", promhttp.Handler()) //Metrics endpoint for scrapping
router.Handle("/{anything}", Sayhello(histogram))
router.Handle("/", Sayhello(histogram))
//Registering the defined metric with Prometheus
prometheus.Register(histogram)

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

func Sayhello(histogram *prometheus.HistogramVec) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {

```

```

//Monitoring how long it takes to respond
start := time.Now()
defer r.Body.Close()
code := 500
defer func() {
    httpDuration := time.Since(start)
    histogram.WithLabelValues(fmt.Sprintf("%d", code)).Observe(httpDuration.
Seconds())
}()

if r.Method == "GET" {
    vars := mux.Vars(r)
    code = http.StatusOK
    if _, ok := vars["anything"]; ok {
        //Sleep random seconds
        rand.Seed(time.Now().UnixNano())
        n := rand.Intn(2) // n will be between 0 and 3
        time.Sleep(time.Duration(n) * time.Second)
        code = http.StatusNotFound
        w.WriteHeader(code)
    }
    //Sleep random seconds
    rand.Seed(time.Now().UnixNano())
    n := rand.Intn(12) //n will be between 0 and 12
    time.Sleep(time.Duration(n) * time.Second)
    name := vars["name"]
    greet := fmt.Sprintf("Hello %s \n", name)
    w.Write([]byte(greet))
} else {
    code = http.StatusBadRequest
    w.WriteHeader(code)
}
}
}

```

You can go through the following procedure to deploy this application on your Kubernetes cluster:
First of all, ensure you have the following packages installed on your system. If not, install them.

```
# yum install -y git make go
```

Create a new namespace for your new application.

```
$ kubectl create namespace prompt
```

Clone the following repo, it contains the example application.

```
$ git clone https://github.com/carillan81/prometheus-example-app.git
Cloning into 'prometheus-example-app'...
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 17 (delta 2), reused 2 (delta 2), pack-reused 9
Receiving objects: 100% (17/17), 5.67 KiB | 5.67 MiB/s, done.
Resolving deltas: 100% (2/2), done.
```

Compile the binary running make.

```
$ cd prometheus-example-app
$ go mod init
$ go get github.com/gorilla/mux
$ go get github.com/prometheus/client_golang/prometheus
$ make
go build -o prometheus-example-app
podman build -t sysdiglabs/prom-example:latest .
STEP 1/3: FROM golang:1.12
STEP 2/3: ADD prometheus-example-app /bin/prometheus-example-app
--> ce977809f0d
STEP 3/3: ENTRYPOINT ["/bin/prometheus-example-app"]
COMMIT sysdiglabs/prom-example:latest
apiVersion: v1
--> bf81147f298
Successfully tagged localhost/sysdiglabs/prom-example:latest
bf81147f298b11837be7d044335d47ab14cedbe490b65d6fe6a4d741bb2fa2ed
```


Tag your new image and push it to your repository to make it public.

```
$ podman tag localhost/sysdiglabs/prom-example:latest myimage/promapp:latest
$ podman push myimage/promapp:latest
Getting image source signatures
Copying blob 7ca605383307 skipped: already exists
Copying blob 8ac92ddf84b3 skipped: already exists
Copying blob dc65f448a2e2 skipped: already exists
Copying blob dea4ecac934f skipped: already exists
Copying blob 346ffb2b67d7 skipped: already exists
Copying blob 020f524b99dd skipped: already exists
Copying blob ed835d26b2a9 done
Copying blob 06036b0307c9 skipped: already exists
Copying config bf81147f29 done
Writing manifest to image destination
Storing signatures
```

Deploy the application creating a simple Pod manifest.

```
apiVersion: v1
kind: Pod
metadata:
  name: promapp
  labels:
    app: promapp
spec:
  containers:
  - name: promapp
    image: myimage/promapp:latest
    ports:
    - containerPort: 8080
```

Create the pod, and a service to make it accessible within your cluster.

```
$ kubectl create -f promapp-pod.yaml -n promapp
$ kubectl get pods -o wide -n promapp
NAME      READY  STATUS   RESTARTS  AGE  IP            NODE                                NOMINATED NODE
promapp   1/1    Running  0          8s   192.169.103.29 k8s-control-1.lab.example.com
<none>    <none>
$ kubectl create svc clusterip promapp --tcp=8080:8080 -n promapp
$ kubectl get svc -n promapp
NAME      TYPE      CLUSTER-IP  EXTERNAL-IP  PORT(S)  AGE
```

```

promapp ClusterIP 10.96.199.55 <none> 8080/TCP 29m
$ kubectl get ep -n promapp
NAME      ENDPOINTS          AGE
promapp   192.169.103.29:8080 29m

```

Forward the port to make the service accessible from your computer and check that the application works properly.

```

$ kubectl port-forward svc/promapp 8080:8080 -n promapp
Forwarding from 127.0.0.1:8080 > 8080
Forwarding from [::1]:8080 > 8080
$ curl http://localhost:8080/sayhello/Scott
Hello Scott

```

Check whether the embedded metrics endpoint is working or not.

```

# curl -s http://localhost:8080/metrics|grep promhttp_metric_handler_requests_
total
# HELP promhttp_metric_handler_requests_total Total number of scrapes by HTTP
status code.
# TYPE promhttp_metric_handler_requests_total counter
promhttp_metric_handler_requests_total{code="200"} 8
promhttp_metric_handler_requests_total{code="500"} 0
promhttp_metric_handler_requests_total{code="503"} 0
# curl -s http://localhost:8080/metrics|grep greeting_seconds_bucket
greeting_seconds_bucket{code="200",le="1"} 1
greeting_seconds_bucket{code="200",le="2"} 3
greeting_seconds_bucket{code="200",le="5"} 4
greeting_seconds_bucket{code="200",le="6"} 5
greeting_seconds_bucket{code="200",le="10"} 5
greeting_seconds_bucket{code="200",le="+Inf"} 6
greeting_seconds_bucket{code="404",le="1"} 0
greeting_seconds_bucket{code="404",le="2"} 0
greeting_seconds_bucket{code="404",le="5"} 0
greeting_seconds_bucket{code="404",le="6"} 0
greeting_seconds_bucket{code="404",le="10"} 0
greeting_seconds_bucket{code="404",le="+Inf"} 1

```

Once you have deployed the application in a Kubernetes cluster with Prometheus and Grafana, and generated a dashboard with Golden Signals, you can obtain the data for the dashboards using these PromQL queries:

- Latency:

```
sum(greeting_seconds_sum)/sum(greeting_seconds_count) //Average  
histogram_quantile(0.95, sum(rate(greeting_seconds_bucket[5m])) by (le)) //  
Percentile p95
```

- Request rate:

```
sum(rate(greeting_seconds_count{}[2m])) //Including errors  
rate(greeting_seconds_count{code="200"}[2m]) //Only 200 OK requests
```

- Errors per second:

```
sum(rate(greeting_seconds_count{code!="200"}[2m]))
```

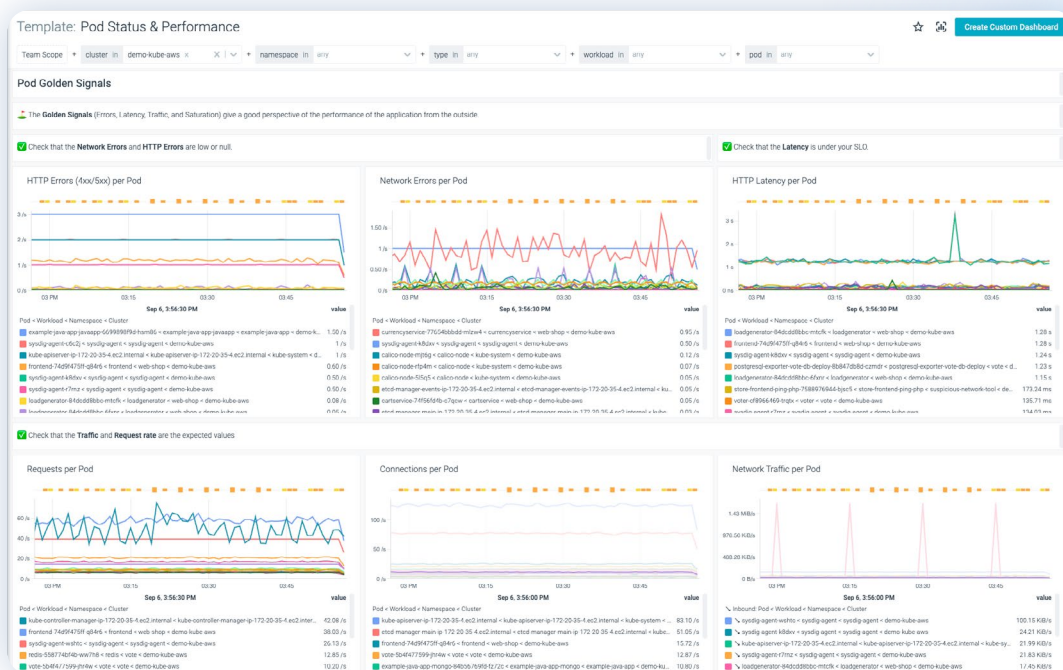
- Saturation: You can use CPU percentage obtained with node-exporter:

```
100 - (avg by (instance) (irate(node_cpu_seconds_total{}[5m])) * 100)
```

This way, you obtain this dashboard with the Golden Signals:



This cluster also has the Sysdig Agent installed. Sysdig allows you to obtain these same Golden Signals without the use of instrumentation (although Sysdig could pull in Prometheus metrics too!). With Sysdig, you could use a default dashboard and obtain the same meaningful information out-of-the-box!



Depending on the nature of the application, it's possible to do different aggregations:

- Response time segmented by response code
- Error rate segmented by response code
- CPU usage per service or deployment

Alerting on application layer metrics

In order to generate application alerts for Golden Signals, you typically need to instrument your application via [Prometheus metrics](#), [statsd](#), or [JMX](#).

Here are some metrics and their alerts that are often found in this category:

- Application availability up/down
- Application response time or latency
- Application error requests rate
- Application requests rate
- Middleware specific metrics: Python uwsgi workers, JVM heap size, etc.
- Database specific metrics: cache hits, indexes, etc.

The following example is a public REST API endpoint monitoring alert for latency over 10 seconds in a 10 minute window. The alert covers the java app deployment in the production namespace prod, using Prometheus custom metrics.

The screenshot shows the 'New Alert / PromQL' configuration interface in Prometheus. The alert title is 'Latency Too High in JavaApp' with a subtitle 'JavaApp latency over 10s in a 10 minute window'. A severity level of 'Low' is selected. Under the '1 Define' section, step (a) 'PromQL' contains the query: `histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket{code=~"20+",kubernetes_namespace="prod",app="javaapp"}[5m])) by (le)) > 10`. Step (b) 'Duration' is set to '10 minute'.

PromQL query:

```
histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket
{code=~"20+",kubernetes_namespace="prod",app="javaapp"}[5m])) by (le)) > 10
```

Caveats and gotchas of Golden Signals in Kubernetes

- Golden Signals are one of the best ways to detect possible problems, but once the problem is detected you will have to use additional metrics and steps to further diagnose it. Detecting issues and resolving them are two different tasks, and they require separate tools and views of the application.
- Mean is not always meaningful. Check standard deviation too, especially with latency. Take into consideration the request path of your application to look for bottlenecks. You should use percentiles instead of averages (or in addition to them).
- Does it make sense to alert every time the CPU or load is high? Probably not. Avoid “alert burnout,” setting alerts only in parameters that are clearly indicative of problems. If it’s not an actionable alert, simply remove it.
- In the situation where a parameter doesn’t look good but it’s not affecting your application directly, don’t set an alert. Instead, create tasks in your backlog to analyze the behavior and avoid possible issues in the long-term.

Lessons learned

1. Knowing the Golden Signals for Kubernetes monitoring enables you to save time by looking at what really matters and avoiding traps that could mask the real problem.
2. By correlating system call information with information collected from the Kubernetes API, allow you to slice your data and locate issues faster. This speeds troubleshooting tasks by exposing performance problems at the different levels of the Kubernetes entities.
3. Sysdig provides a huge collection of out-of-the-box dashboards, not only for Golden Signals but “Pod Golden Signals,” “Pod status”, “resources requested”, and “resources used” dashboards, among others. It facilitates the heavy task of troubleshooting and diagnosing problems.
4. Implicit errors like CPU Throttling, other Kubernetes errors like CrashLoopBackOff, or Image Pull errors are visible thanks to the Sysdig Advisor. You can detect, troubleshoot, and fix these issues almost in real time.

Monitoring Kubernetes Infrastructure and Core Components

Monitoring Kubernetes Infrastructure

Given that Kubernetes adds reliability by adding/moving pods within the cluster, one node is not attached to the applications running on top of it, so the availability of the nodes is transformed into a capacity issue. We have to ensure that nodes work well enough to not be a problem, and that we have enough nodes to run our workloads. If a node fails, the workloads running there automatically are migrated to a different node. As long as there are spare resources to run everything, there will be only a minor interruption, and if the system is well designed, no interruption at all.

Alerting on the host or Kubernetes node layer

Alerting at the host layer shouldn't be very different from monitoring cloud instances, VMs, or bare metal servers. It's going to be mostly about if the host is up or down/unreachable, and resource availability (CPU, memory, disk, etc.).

Now, the main difference is the severity of the alerts. Before, a system down likely meant you had an application down and an incident to handle (barring effective high availability). With Kubernetes, services are now ready to move across hosts and host alerts should never wake you up, as long as you have enough of them to run your apps. You only need to be sure that the dead host has been replaced by a new one.

Let's look at a couple of options that you should still consider:

Host is down

If a host is down or unreachable, you might want to receive a notification. You should apply this single alert across your entire infrastructure. Give it a five-minute wait time in this case, since you don't want to see noisy alerts on network connectivity hiccups. You might want to lower that down to one or two minutes depending on how quickly you want to receive a notification, but you risk flapping notifications if your hosts go up and down too often.

Host is Down

Insert alert description

High ▼

1 Define

(a) Select entity to monitor ?

Alert if any ×

▼ is down.

(b) Scope ?

▼

(c) Trigger ?

If entity is down for the last minute ▼ for % of the time

Later in this document, you will see that since you have another layer in the orchestration that acts as a high availability system, one node failing is not of extreme importance. Anyway, you have to monitor the number of nodes remaining referenced to the load you're running so you can ensure the active nodes can handle the load. In addition, you should be aware of the moment when the failure of another node would provoke a shortage of resources to run all of the workloads.

Disk usage

This is a slightly more complex alert. You can apply this alert across all file systems of our entire infrastructure. You manage to do that setting everywhere using scope and firing a separate evaluation/alert per mount (in Sysdig fs.mountDir).

This is a generic alert that triggers over 80% usage, but you might want different policies like a second higher priority alert with a higher threshold, like 95%, or different thresholds depending on the file system.

Disk usage warning

Insert alert description

Medium

1 Define

a Metric ?

Average of fs.used.percent

b Scope ?

everywhere

c Trigger ?

If metric > 80 % for the last 10 minute on average

Single Alert

Trigger a single alert when the trigger condition is met across your scope

If you want to create different thresholds for different services or hosts, simply change the scope to where you want to apply a particular threshold. If you need more advanced disk alerts, [PromQL has some functions that allow you to do linear predictions](#) and see how fast the disk is filling at the current rate.

The screenshot shows the Sysdig alert configuration interface. At the top, the title is "Disk filling in 12h". Below it is a text input field for "Insert alert description". A severity dropdown menu is set to "Low". The configuration is divided into steps, with the first step "1 Define" being active. Under this step, there are two sub-sections: (a) "PromQL" and (b) "Duration". The PromQL section contains a text area with the query: `(predict_linear(node_filesystem_free{job="node"}[1h],43200)) < 0`. The Duration section shows "For the duration of" with a value of "30" and a unit of "minute".

PromQL query:

```
(predict_linear(node_filesystem_free{job="node"}[1h],43200 )) < 0
```

This alert will trigger in case the disk is going to be full in the next 12 hours at current speed.

Some other resources

Usual suspects in this category are alerts on load, CPU usage, memory, and swap usage. You probably want to send a notification, but not wake anyone, if any of those are significantly higher during a prolonged time frame. A compromise needs to be found between the threshold, the wait time, and how noise can become your alerting system with no actionable alerts.

If you still want to set up metrics for these resources, look at the following metrics names on Sysdig Monitor:

- For load: `load.average.1m`, `load.average.5m`, and `load.average.15m`
- For CPU: `cpu.used.percent`
- For memory: `memory.used.percent` or `memory.bytes.used`
- For swap: `memory.swap.used.percent` or `memory.swap.bytes.used`

In this category, some people also include monitoring the cloud provider resources that are part of their infrastructure.

Do you have enough Kubernetes nodes in the cluster?

A node failure is not a problem in Kubernetes, since the scheduler will spawn the containers from the pods in the failed node into other available nodes. But what if you are running out of nodes, or the resource requirements for the deployed applications overbook existing nodes? And are you hitting a quota limit?

Alerting in these cases is not easy, as it will depend on how many nodes you want to have on standby or how far you want to push oversubscription on your existing nodes. To monitor a node status alert on the metrics: `kube_node_status_ready` and `kube_node_spec_unschedulable`.

An example of this would be the following expression:

```
sum(namespace:kube_pod_container_resource_requests_cpu_cores:sum)
/
sum(node:node_num_cpu:sum)
>
(count(node:node_num_cpu:sum)-1) / count(node:node_num_cpu:sum)
```

This means the alert would trigger in case the sum of all of the requested resources is greater than the capacity of the cluster in case one of the nodes fails.

If you want to alert on capacity, you will have to sum each scheduled pod request for CPU and memory, and then check that it doesn't go over each node: `kube_node_status_capacity_cpu_cores` and `kube_node_status_capacity_memory_bytes`.

For example, this query would alert in case the requested resources are above 90% of the available quota:

```
100 * kube_resourcequota{job="kubernetes-service-endpoints", type="used"}
/ ignoring(instance, job, type)
(kube_resourcequota{job="kubernetes-service-endpoints", type="hard"} > 0)
> 90
```

Control Plane

The main components of the control plane are:

- API Server
- Kubelet
- Controller manager
- etcd
- Kube-proxy
- kube-dns

In this section, we will describe why and how to monitor the API Server, Kubelet, controller manager, and etcd.

Monitoring Kubernetes control plane in Sysdig Monitor

In order to track the Kubernetes control plane in Sysdig Monitor, you only need to have the Sysdig agent up and running, it is already configured to start scraping Kubernetes control plane metrics automatically. On the other hand, you can always use your own Prometheus instance to gather the metrics and filter them. Prometheus is installed by default in Openshift 4 or when using Kubernetes metrics manager. If you don't have one, no need to worry, with Sysdig Monitor it is not required at all. Anyway, deploying a new one is as simple as executing two commands: create a namespace for the Prometheus, and deploy it with helm 3.

```
$ kubectl create ns monitoring
$ helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
$ helm repo update
$ helm install -f values.yaml prometheus -n monitoring prometheus-community/prometheus
```

And use this for values.yaml:

```
server:
  strategy:
    type: Recreate
podAnnotations:
  prometheus.io/scrape: "true"
  prometheus.io/port: "9090"
```

Once Prometheus is up and running, you are ready to start scraping metrics!

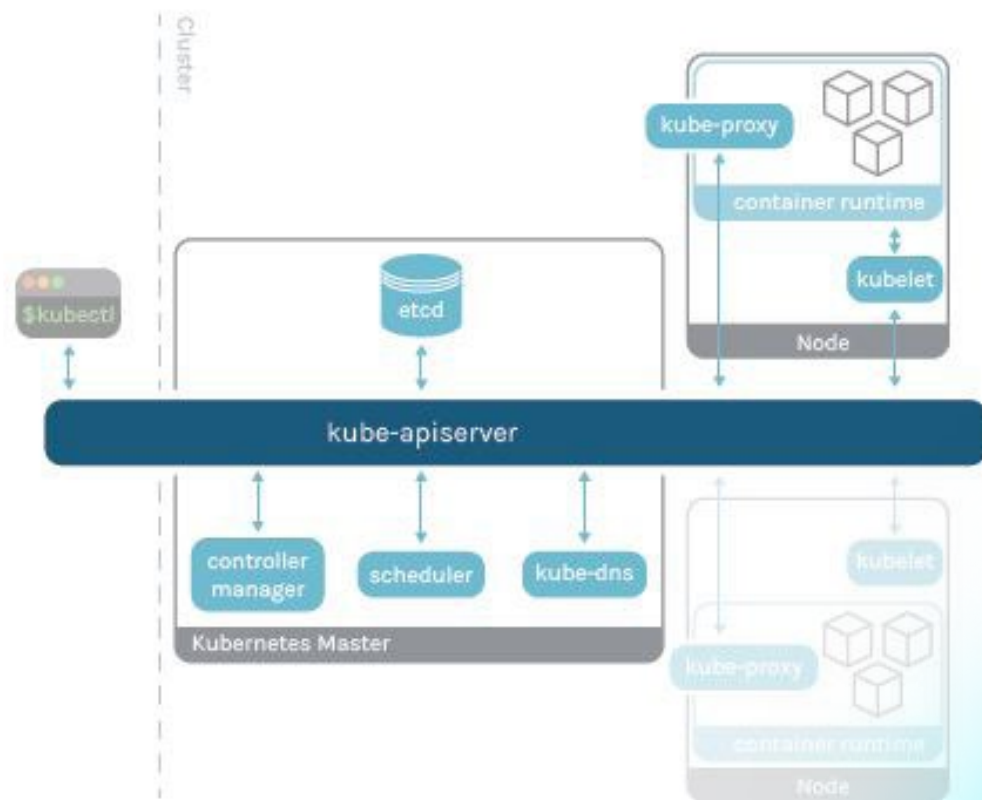
Sysdig Monitor is able to pull the Kubernetes control plane metrics automatically, no additional steps are required.

Disclaimer: Some of the control plane jobs might be disabled by default. If you miss any of the Kubernetes control plane metrics or dashboards, reach out to your customer support representative and request activation.

How to monitor the Kubernetes API server

[Learning how to monitor the Kubernetes API server](#) is of vital importance when running Kubernetes in production. Monitoring kube-apiserver will let you detect and troubleshoot latency, errors, and validate the service performs as expected. Keep reading to learn how you can collect the most important apiserver metrics and use them to monitor this service.

The Kubernetes API server is a foundational component of the Kubernetes control plane. All of the services running inside the cluster use this interface to communicate between each other. The entirety of user interaction is handled through the API as well; kubectl is a wrapper to send requests to the API. While kubectl uses HTTP to connect to the API server, some of the control plane components, like the etcd, use gRPC. You should be ready to monitor both channels.



Like with any other microservice, we're going to take the Golden Signals approach to monitor the Kubernetes API server health and performance:

- Latency
- Request rate
- Errors
- Saturation

Before we dive into the meaning of each one, let's see how you can fetch those metrics.

Getting the metrics to monitor kube-apiserver

The API server has been instrumented and it exposes [Prometheus metrics](#) by default, providing monitoring metrics like latency, requests, errors, and etcd cache status. This endpoint can be easily scraped, obtaining useful information without the need for additional scripts or exporters.

The API server requires authentication to make a request to **/metrics** endpoint, so you need to get credentials with privileges for that. If you're running Prometheus inside the cluster, you can authenticate using the prometheus-server service account already bound to the prometheus-server **ClusterRole**, granting GET requests to **/metrics** endpoint.

```
$ kubectl get clusterrolebinding prometheus-server -n monitoring -o json|jq
".roleRef,.subjects"
{
  "apiGroup": "rbac.authorization.k8s.io",
  "kind": "ClusterRole",
  "name": "prometheus-server"
}
[
  {
    "kind": "ServiceAccount",
    "name": "prometheus-server",
    "namespace": "monitoring"
  }
]
$ kubectl get clusterrole prometheus-server -o json |jq ".rules[2]"
{
  "nonResourceURLs": [
    "/metrics"
  ],
  "verbs": [
    "get"
  ]
}
```

You can access the **/metrics** endpoint using the bearer token from the service account, present in the pod, in **/var/run/secrets/kubernetes.io/serviceaccount**.

You can test the authentication by executing this shell command from within the Prometheus pod:

```
/prometheus $ wget -S --header "Authorization: Bearer $(cat /var/run/secrets/
kubernetes.io/serviceaccount/token)" https://kubernetes.default.svc/metrics
Connecting to kubernetes.default.svc (10.96.0.1:443)
wget: note: TLS certificate validation not implemented
HTTP/1.1 200 OK
Audit-Id: fae26052-f669-4ac0-93ef-40e1d751f1d8
Cache-Control: no-cache, private
Content-Type: text/plain; version=0.0.4; charset=utf-8
X-Kubernetes-Pf-Flowschema-Uid: 2647e24e-e736-4d39-aeb4-a096b29f02ab
X-Kubernetes-Pf-Prioritylevel-Uid: ac1f14db-1afd-4c4a-a195-0e1a692082cb
Date: Fri, 28 Oct 2022 10:17:18 GMT
Connection: close
Transfer-Encoding: chunked
```

It will return a long list of Prometheus metrics (truncated here):

```
/prometheus $ head metrics
# HELP aggregator_openapi_v2_regeneration_count [ALPHA] Counter of OpenAPI v2
spec regeneration count broken down by causing APIService name and reason.
# TYPE aggregator_openapi_v2_regeneration_count counter
aggregator_openapi_v2_regeneration_count{apiservice="*",reason="startup"} 0
aggregator_openapi_v2_regeneration_count{apiservice="k8s_internal_local_
delegation_chain_0000000002",reason="update"} 0
aggregator_openapi_v2_regeneration_count{apiservice="v3.projectcalico.
org",reason="add"} 0
aggregator_openapi_v2_regeneration_count{apiservice="v3.projectcalico.
org",reason="update"} 0
# HELP aggregator_openapi_v2_regeneration_duration [ALPHA] Gauge of OpenAPI v2
spec regeneration duration in seconds.
# TYPE aggregator_openapi_v2_regeneration_duration gauge
aggregator_openapi_v2_regeneration_duration{reason="add"} 0.044517713
aggregator_openapi_v2_regeneration_duration{reason="startup"} 0.01630406
...
```

Configuring Prometheus to scrape the Kubernetes API server endpoint can be done by adding one job to your targets (it is already provided by default by installing Prometheus with the [community Prometheus Helm chart](#)):

```
- job_name: 'kubernetes-apiservers'
  kubernetes_sd_configs:
    - role: endpoints
  scheme: https
  tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  relabel_configs:
    - source_labels: [__meta_kubernetes_namespace, __meta_kubernetes_service_name, __meta_kubernetes_endpoint_port_name]
      action: keep
      regex: default;kubernetes;https
```

Monitoring the Kubernetes API server: What to look for?

You can also use Golden Signals to monitor the Kubernetes API server.

Disclaimer: API server metrics might differ between Kubernetes versions. Here, we used Kubernetes 1.25. You can check the metrics available for your version in the [Kubernetes repo](#).

Latency: Latency can be extracted from the `apiserver_request_duration_seconds` histogram buckets. Data is broken down into different categories, like verb, group, version, resource, component, etc.

```
# HELP apiserver_request_duration_seconds [STABLE] Response latency distribution
in seconds for each verb, dry run value, group, version, resource, subresource,
scope and component.
# TYPE apiserver_request_duration_seconds histogram
apiserver_request_duration_seconds_bucket{component="",dry_run="",group="",resource="",scope="",subresource="/healthz",verb="GET",version="",le="0.005"} 42577
apiserver_request_duration_seconds_bucket{component="",dry_run="",group="",resource="",scope="",subresource="/healthz",verb="GET",version="",le="0.025"} 45964
apiserver_request_duration_seconds_bucket{component="",dry_run="",group="",resource="",scope="",subresource="/healthz",verb="GET",version="",le="0.05"} 46010
apiserver_request_duration_seconds_bucket{component="",dry_run="",group="",resource="",scope="",subresource="/healthz",verb="GET",version="",le="0.1"} 46014
apiserver_request_duration_seconds_bucket{component="",dry_run="",group="",resource="",scope="",subresource=""}
```

```

healthz",verb="GET",version="",le="0.2"} 46014
apiserver_request_duration_seconds_bucket{component="",dry_
run="",group="",resource="",scope="",subresource="/
healthz",verb="GET",version="",le="0.4"} 46014
apiserver_request_duration_seconds_bucket{component="",dry_
run="",group="",resource="",scope="",subresource="/
healthz",verb="GET",version="",le="0.6"} 46014
apiserver_request_duration_seconds_bucket{component="",dry_
run="",group="",resource="",scope="",subresource="/
healthz",verb="GET",version="",le="0.8"} 46014
apiserver_request_duration_seconds_bucket{component="",dry_
run="",group="",resource="",scope="",subresource="/
healthz",verb="GET",version="",le="1"} 46014
apiserver_request_duration_seconds_bucket{component="",dry_
run="",group="",resource="",scope="",subresource="/
healthz",verb="GET",version="",le="1.25"} 46014
apiserver_request_duration_seconds_bucket{component="",dry_
run="",group="",resource="",scope="",subresource="/
healthz",verb="GET",version="",le="1.5"} 46014
apiserver_request_duration_seconds_bucket{component="",dry_
run="",group="",resource="",scope="",subresource="/
healthz",verb="GET",version="",le="2"} 46014
apiserver_request_duration_seconds_bucket{component="",dry_
run="",group="",resource="",scope="",subresource="/
healthz",verb="GET",version="",le="3"} 46014
apiserver_request_duration_seconds_bucket{component="",dry_
run="",group="",resource="",scope="",subresource="/
healthz",verb="GET",version="",le="4"} 46014

```

It's a good idea to use percentiles to understand the latency spread:

```

histogram_quantile(0.99,
sum(rate(apiserver_request_duration_seconds_bucket{job="kubernetes-
apiservers"}[5m])) by (verb, le))

```

Request rate: The metric `apiserver_request_total` can be used to monitor the requests to the service, where they are coming from, as well as to which service, which action, and whether they were successful:

```
# TYPE apiserver_request_count counter
apiserver_request_total{client="Go-http-client/1.1",code="0",contentType="",resource="pods",scope="namespace",subresource="portforward",verb="CONNECT"} 4
apiserver_request_total{client="Go-http-client/2.0",code="200",contentType="application/json",resource="alertmanagers",scope="cluster",subresource="",verb="LIST"} 1
apiserver_request_total{client="Go-http-client/2.0",code="200",contentType="application/json",resource="alertmanagers",scope="cluster",subresource="",verb="WATCH"} 72082
apiserver_request_total{client="Go-http-client/2.0",code="200",contentType="application/json",resource="clusterinformations",scope="cluster",subresource="",verb="LIST"} 1
```

For example, you can get all of the successful requests across the service like this:

```
sum(rate(apiserver_request_total{job="kubernetes-apiservers",code=~"2.."}[5m]))
```

Errors: You can use the same query used for request rate, but filter for 400 and 500 error codes:

```
sum(rate(apiserver_request_total{job="kubernetes-apiservers",code=~"[45].."}[5m]))
```

It is even more interesting monitoring the error rate vs. the total amount of requests. This way, you get the magnitude of the errors for the Kubernetes API server requests.

```
sum(rate(apiserver_request_total{job="kubernetes-apiservers",code=~"[45].."}[5m]))*100/sum(rate(apiserver_request_total{job="kubernetes-apiservers"}[5m]))
```


Saturation: You can monitor saturation through system resource consumption metrics like CPU, memory, and network I/O for this service.

In addition to API server related metrics, you can access **other relevant metrics**. API server offers:

- From **controller-manager**:
 - **work queue addition rate**: How fast are you scheduling new actions to perform by controller? These actions can include additions, deletions, and modifications of any resource in the cluster (workloads, configmaps, services, etc.).
 - **work queue latency**: How fast is the controller-manager performing these actions?
 - **work queue depth**: How many actions are waiting to be executed?
- From **etcd**:
 - **etcd cache entries**: How many query results have been cached?
 - **etcd cache hit/miss rate**: Is cache being useful?
 - **etcd cache duration**: How long are the cache results stored?

Examples of issues in API server

You detect an increase of latency in the requests to the API.

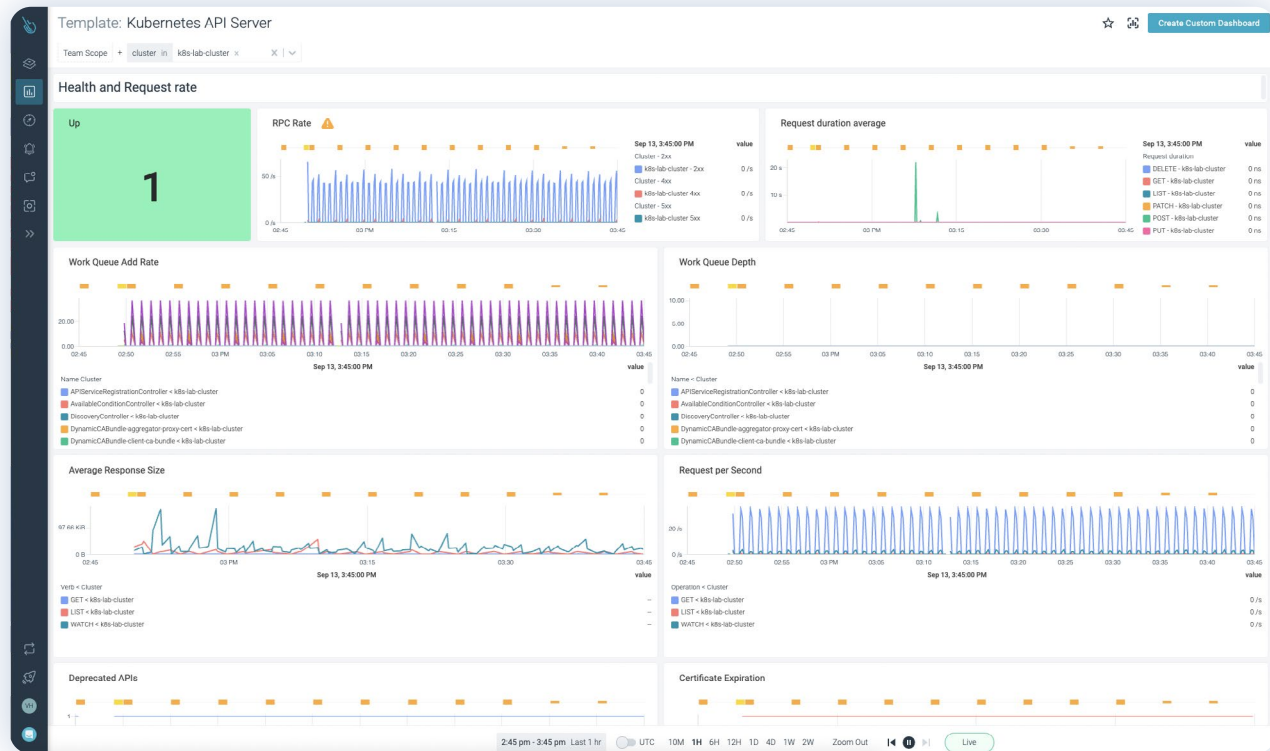
This is typically a sign of overload in the API server. Most likely, your cluster has a lot of load and the API server needs to be scaled out. You can segment the metrics by type of request, resource, or verb. This way, you can detect where the problem is. Maybe you are having issues reading or writing to etcd and need to fix it.

You detect an increase in the depth and latency of the work queue.

You are having issues scheduling actions. You should check that the scheduler is working. Maybe some of your nodes are overloaded and you need to scale out your cluster. Maybe one node is having issues and you want to replace it.

Monitoring Kubernetes API server metrics in Sysdig Monitor

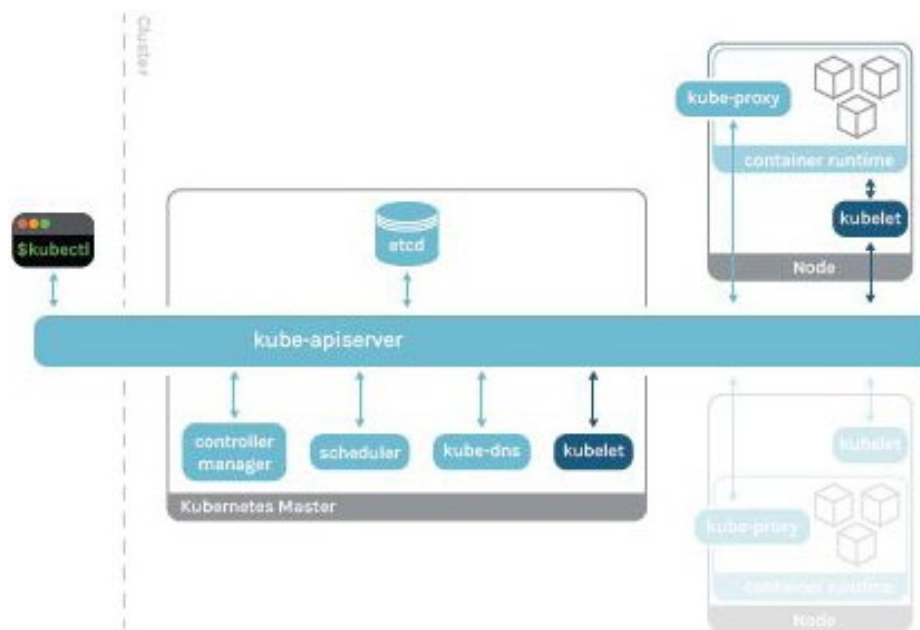
If you want to monitor Kubernetes API server using Sysdig Monitor, just run the Sysdig agent on your Kubernetes nodes, as we explained earlier in the “Monitoring kubernetes control plane in Sysdig Monitor” section. Then, you only need to log into the Sysdig portal and review your data in the out-of-the-box API Server dashboard. Here is an API server dashboard in Sysdig Monitor.



How to monitor Kubelet

[Monitoring Kubelet](#) is essential when running Kubernetes in production. Kubelet is a very important service inside Kubernetes' control plane. It's the component that cares that the containers described by pods are running in the nodes. Kubelet works in a declarative way by receiving PodSpecs and ensuring that the current state matches desired pods.

Kubelet has some differences with other control plane components as it's the only one that runs over the host OS in the nodes, and not as a Kubernetes entity. This makes kubelet monitoring a little special, but you can still rely on [Prometheus service discovery \(node\)](#).



Getting metrics from Kubelet

Kubelet has been instrumented and it exposes [Prometheus metrics](#) by default in the port 10250 of the host, providing information about pods' volumes and internal operations. This endpoint can be easily scraped, obtaining useful information without the need for additional scripts or exporters.

You can scrape Kubelet metrics accessing the port in the node directly by using the secure port and authenticating with your token.

In order to get the Kubelet metrics, get access to the node itself, or ssh into a Pod, this service is listening on `0.0.0.0` address, so in terms of connectivity, there are no restrictions at all. If the Pod has access to the host network, you can access it using `localhost` too.

```
$ curl -k -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" https://localhost:10250/metrics
```

You will get a long list of metrics with this structure (truncated):

```
# HELP apiserver_audit_event_total Counter of audit events generated and sent to
the audit backend.
# TYPE apiserver_audit_event_total counter
apiserver_audit_event_total 0
# HELP apiserver_audit_requests_rejected_total Counter of apiserver requests
rejected due to an error in audit logging backend.
# TYPE apiserver_audit_requests_rejected_total counter
apiserver_audit_requests_rejected_total 0
# HELP apiserver_client_certificate_expiration_seconds Distribution of the
remaining lifetime on the certificate used to authenticate a request.
# TYPE apiserver_client_certificate_expiration_seconds histogram
apiserver_client_certificate_expiration_seconds_bucket{le="0"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="1800"} 0
...
```

If you want to configure Prometheus to scrape Kubelet, you can add this job to your targets:

```
- bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  job_name: kubernetes-nodes
  kubernetes_sd_configs:
    - role: node
  relabel_configs:
    - action: labelmap
      regex: __meta_kubernetes_node_label_(.+)
    - replacement: kubernetes.default.svc:443
      target_label: __address__
    - regex: (.+)
      replacement: /api/v1/nodes/$1/proxy/metrics
      source_labels:
        - __meta_kubernetes_node_name
      target_label: __metrics_path__
    - scheme: https
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
        insecure_skip_verify: true
```

You can customize your own labels and relabeling configuration.

Monitoring Kubelet: What to look for?

Disclaimer: Kubelet metrics might differ between Kubernetes versions. Here, we used Kubernetes 1.25. You can check the metrics available for your version in the [Kubernetes repo](#).

Number of Kubelet instances: This value will give an idea of the general health of the Kubelet in the nodes. The expected value is the number of nodes in the cluster. You can obtain this value counting targets found by Prometheus or by checking the process if you have low-level access to the node. A possible PromQL query for a single stat graph would be:

```
sum(kubelet_node_name)
```

Another way to count the number of **Ready** nodes is:

Note: If you want to get the number of not-ready nodes, filter by **status="false"**.

```
sum(kube_node_status_condition{condition="Ready", status="true"})
```

Number of pods and containers running: Kubelet provides insight to the number of pods and containers really running in the node. You can check this value with the one expected, or reported, by Kubernetes to detect possible issues in the nodes.

```
# HELP kubelet_running_pod_count Number of pods currently running
# TYPE kubelet_running_pod_count gauge
kubelet_running_pods 9
# HELP kubelet_running_container_count Number of containers currently running
# TYPE kubelet_running_container_count gauge
kubelet_running_containers 9
```

Number of volumes: In the system, kubelet mounts the volumes indicated by the controller so it can provide information on them. This can be useful to diagnose issues with volumes that aren't being mounted when a pod is recreated in a statefulSet. It provides two metrics that can be represented together; the number of desired volumes and the number of volumes actually mounted:

```
# HELP volume_manager_total_volumes Number of volumes in Volume Manager
# TYPE volume_manager_total_volumes gauge
volume_manager_total_volumes{plugin_name="kubernetes.io/
configmap",state="actual_state_of_world"} 1
volume_manager_total_volumes{plugin_name="kubernetes.io/
configmap",state="desired_state_of_world"} 1
```

```

volume_manager_total_volumes{plugin_name="kubernetes.io/empty-dir",state="actual_state_of_world"} 1
volume_manager_total_volumes{plugin_name="kubernetes.io/empty-dir",state="desired_state_of_world"} 1
volume_manager_total_volumes{plugin_name="kubernetes.io/host-path",state="actual_state_of_world"} 55
volume_manager_total_volumes{plugin_name="kubernetes.io/host-path",state="desired_state_of_world"} 55
volume_manager_total_volumes{plugin_name="kubernetes.io/secret",state="actual_state_of_world"} 4
volume_manager_total_volumes{plugin_name="kubernetes.io/secret",state="desired_state_of_world"} 4

```

Differences between these two values (outside of transitory phases) can be a good indicator of issues.

Golden Signals of every operation performed by kubelet (Operation rate, operation error rate and operation duration). Saturation can be measured with system metrics and Kubelet offers detailed information of the operations performed by the daemon. Metrics that can be used are:

- **kubelet_runtime_operations_total**: Total count of runtime operations of each type.

```

# HELP kubelet_runtime_operations_total Cumulative number of runtime
operations by operation type.
# TYPE kubelet_runtime_operations_total counter
kubelet_runtime_operations_total{operation_type="container_status"} 225
kubelet_runtime_operations_total{operation_type="create_container"} 44
kubelet_runtime_operations_total{operation_type="exec"} 5
kubelet_runtime_operations_total{operation_type="exec_sync"} 1.050273e+06
...

```

This is a counter metric, you can use the rate function to calculate the average increase rate for Kubelet runtime operations.

```

sum(rate(kubelet_runtime_operations_total{job="kubernetes-nodes\"}[5m]))
by (operation_type, instance)

```


- **kubelet_runtime_operations_errors_total**: Count of errors in the operations. This can be a good indicator of low-level issues in the node, such as problems with container runtime.

```
# HELP kubelet_runtime_operations_errors_total Cumulative number of runtime
operation errors by operation type.
# TYPE kubelet_runtime_operations_errors_total counter
kubelet_runtime_operations_errors_total{operation_type="container_status"}
18
kubelet_runtime_operations_errors_total{operation_type="create_container"}
1
kubelet_runtime_operations_errors_total{operation_type="exec_sync"} 7
```

- **kubelet_runtime_operations_duration_seconds_bucket**: Duration of the operations. Useful to calculate percentiles.

```
# HELP kubelet_runtime_operations_duration_seconds Duration in seconds of
runtime operations. Broken down by operation type.
# TYPE kubelet_runtime_operations_duration_seconds histogram
kubelet_runtime_operations_duration_seconds_bucket{operation_
type="container_status",le="0.005"} 194
kubelet_runtime_operations_duration_seconds_bucket{operation_
type="container_status",le="0.01"} 207
...
```

You may want to calculate the 99th percentile of the Kubelet runtime operations duration by instance and operation type.

```
histogram_quantile(0.99, sum(rate(kubelet_runtime_operations_duration_
seconds_bucket{instance=~".*"}[5m])) by (instance, operation_type, le))
```

Pod start rate and duration: This could indicate issues with container runtime or with access to images.

- **kubelet_pod_start_duration_seconds_count:** Number of pod start operations.

```
# HELP kubelet_pod_start_duration_seconds Duration in seconds for a single
pod to go from pending to running.
# TYPE kubelet_pod_start_duration_seconds histogram
...
kubelet_pod_worker_duration_seconds_count{operation_type="sync"} 196
...
```

- **kubelet_pod_worker_duration_seconds_count:** The number of create, sync, and update operations for a single Pod.

```
# HELP kubelet_pod_worker_duration_seconds Duration in seconds to sync a
single pod. Broken down by operation type: create, update, or sync
# TYPE kubelet_pod_worker_duration_seconds histogram
...
kubelet_pod_worker_duration_seconds_count{operation_type="sync"} 196
...
```

- **kubelet_pod_start_duration_seconds_bucket:** This metric gives you a histogram on the duration, in seconds, from the Kubelet seeing a Pod for the first time to the Pod starting to run.

```
# HELP kubelet_pod_worker_duration_seconds Duration in seconds to sync a
single pod. Broken down by operation type: create, update, or sync
# TYPE kubelet_pod_worker_duration_seconds histogram
kubelet_pod_worker_duration_seconds_bucket{operation_
type="sync",le="0.005"} 194
kubelet_pod_worker_duration_seconds_bucket{operation_type="sync",le="0.01"}
195
...
```

You can get the 95th percentile of the Pod start duration seconds metric by node.

```
histogram_quantile(0.95,sum(rate(kubelet_pod_start_duration_seconds_
bucket{instance=~".*"}[5m])) by (instance, le))
```

- **kubelet_pod_worker_duration_seconds_bucket:** This metric provides the duration in seconds to sync a Pod. The information is broken down into three different types (create, update, and sync).

```
# HELP kubelet_pod_worker_duration_seconds Duration in seconds to sync a
single pod. Broken down by operation type: create, update, or sync
# TYPE kubelet_pod_worker_duration_seconds histogram
kubelet_pod_worker_duration_seconds_bucket{operation_
type="sync",le="0.005"} 194
kubelet_pod_worker_duration_seconds_bucket{operation_type="sync",le="0.01"}
195
...
```

It may be worth checking percentiles for the Kubelet Pod worker duration metric as well. This way, you will get a better understanding of how the different operations are performing across all the nodes.

```
histogram_quantile(0.99, sum(rate(kubelet_pod_worker_duration_seconds_
bucket{instance=~".*"}[5m])) by (instance, operation_type, le))
```

Storage Golden Signals (operation rate, error rate, and duration).

- **storage_operation_duration_seconds_count:** The number of storage operations for every volume_plugin.

```
# HELP storage_operation_duration_seconds Storage operation duration
# TYPE storage_operation_duration_seconds histogram
...
storage_operation_duration_seconds_count{operation_name="verify_controller_
attached_volume",volume_plugin="kubernetes.io/configmap"} 16
...
```

In order to get the storage operation rate, you can use the following query. Get an overview on the different operations performed, the status of each operation, and the volume plugin involved.

```
sum(rate(storage_operation_duration_seconds_count{instance=~".*"}[5m])) by
(instance, operation_name, volume_plugin, status)
```

- **storage_operation_duration_seconds_bucket:** Measures the duration, in seconds, for each storage operation. This information is represented by a histogram.

```
# HELP storage_operation_duration_seconds Storage operation duration
# TYPE storage_operation_duration_seconds histogram
storage_operation_duration_seconds_bucket{operation_name="verify_
controller_attached_volume",volume_plugin="kubernetes.io/
configmap",le="0.1"} 16
storage_operation_duration_seconds_bucket{operation_name="verify_
controller_attached_volume",volume_plugin="kubernetes.io/
configmap",le="0.25"} 16
...
```

This time, you may want to get the 99th percentile of the storage operation duration, grouped by instance, operation name, and volume plugin.

```
histogram_quantile(0.99, sum(rate(storage_operation_duration_seconds_
bucket{instance=~".*"}[5m])) by (instance, operation_name, volume_plugin,
le))
```

Cgroup manager operation rate and duration.

- **kubelet_cgroup_manager_duration_seconds_count:** This metric counts the number of destroy and update operations.

```
# HELP kubelet_cgroup_manager_duration_seconds Duration in seconds for
cgroup manager operations. Broken down by method.
# TYPE kubelet_cgroup_manager_duration_seconds histogram
...
kubelet_cgroup_manager_duration_seconds_count{operation_type="create"} 28
...
```

- **kubelet_cgroup_manager_duration_seconds_bucket:** This metric provides the duration, in seconds, for the cgroup manager operations. This data is broken down into two different methods (destroy and update).

```
# HELP kubelet_cgroup_manager_duration_seconds Duration in seconds for
cgroup manager operations. Broken down by method.
# TYPE kubelet_cgroup_manager_duration_seconds histogram

kubelet_cgroup_manager_duration_seconds_bucket{operation_
type="create",le="0.005"} 11
kubelet_cgroup_manager_duration_seconds_bucket{operation_
type="create",le="0.01"} 21
...
```

Let's see how to get a histogram that represents the 99th percentile of the Kubelet cgroup manager operations. This query will help you better understand the duration of every Kubelet cgroup operation type.

```
histogram_quantile(0.99, sum(rate(kubelet_cgroup_manager_duration_seconds_
bucket{instance=~".*"}[5m])) by (instance, operation_type, le))
```

Pod Lifecycle Event Generator (PLEG): A module in the Kubelet responsible for adjusting the container runtime state. To achieve this task, it relies on a periodic listing to discover container changes. These metrics can be useful for you to determine whether there are errors with latencies at container runtime. Relist rate, relist interval, and relist duration are some of the metrics shown here. Errors or excessive latency in these values can provoke issues in the Kubernetes status of the pods.

- **kubelet_pleg_relist_duration_seconds_count:** The **number of relisting operations** in PLEG.

```
# HELP kubelet_pleg_relist_duration_seconds Duration in seconds for
relisting pods in PLEG.
# TYPE kubelet_pleg_relist_duration_seconds histogram
...
kubelet_pleg_relist_duration_seconds_count 5.344102e+06
...
```

- **kubelet_pleg_relist_interval_seconds_bucket**: This metric provides a histogram with the interval in seconds between relisting operations in PLEG.

```
# HELP kubelet_pleg_relist_interval_seconds Interval in seconds between
relisting in PLEG.
# TYPE kubelet_pleg_relist_interval_seconds histogram
kubelet_pleg_relist_interval_seconds_bucket{le="0.005"} 0
kubelet_pleg_relist_interval_seconds_bucket{le="0.01"} 0
...
```

You can get the 99th percentile of the interval between Kubelet relisting PLEG operations.

```
histogram_quantile(0.99, sum(rate(kubelet_pleg_relist_interval_seconds_
bucket{instance=~".*"}[5m])) by (instance, le))
```

- **kubelet_pleg_relist_duration_seconds_bucket**: This is the **duration in seconds** for relisting Pods in **PLEG**.

```
# HELP kubelet_pleg_relist_duration_seconds Duration in seconds for
relisting pods in PLEG.
# TYPE kubelet_pleg_relist_duration_seconds histogram
kubelet_pleg_relist_duration_seconds_bucket{le="0.005"} 2421
kubelet_pleg_relist_duration_seconds_bucket{le="0.01"} 4.335858e+06
...
```

This time, let's see how to build a histogram representing the 99th percentile of Kubelet PLEG relisting operations.

```
histogram_quantile(0.99, sum(rate(kubelet_pleg_relist_duration_seconds_
bucket{instance=~".*"}[5m])) by (instance, le))
```


Examples of issues in Kubelet

Pods are not starting.

This is typically a sign of Kubelet having problems connecting to the container runtime running below. Check for the **pod start rate and duration** metrics to see if there is latency creating the containers, or if they are in fact starting.

A node doesn't seem to be scheduling new pods.

Check the Kubelet job number. There's a chance that Kubelet has died in a node and is unable to schedule pods.

Kubernetes seems to be slow performing operations.

Check all of the Golden Signals in Kubelet metrics. It may have issues with storage, latency, communicating with the container runtime engine, or load issues.

Monitoring Kubelet metrics in Sysdig Monitor

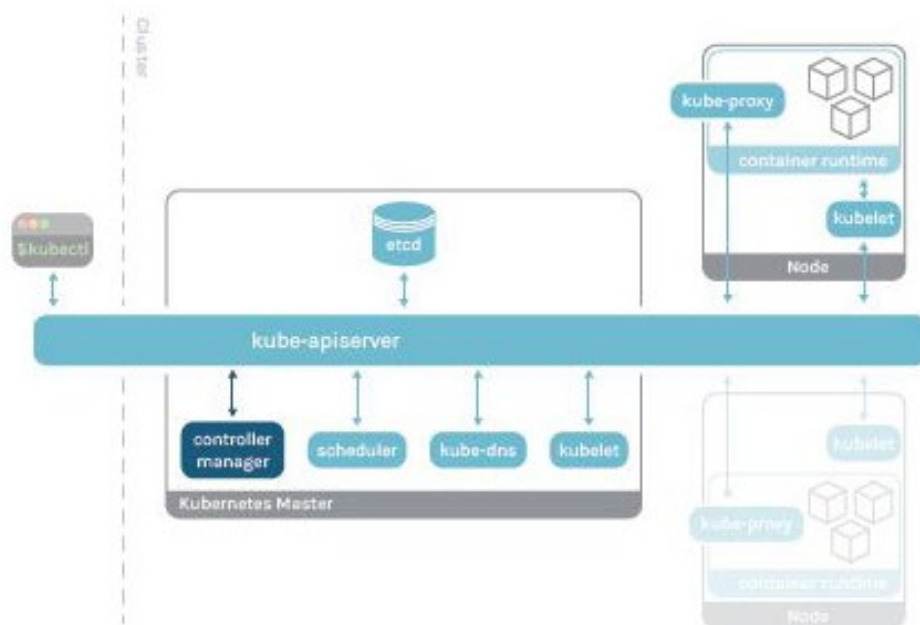
In order to track Kubelet in Sysdig Monitor, just run the Sysdig agent on your Kubernetes nodes, as we explained above in “Monitoring kubernetes control plane in Sysdig Monitor”. Log into the Sysdig portal, and check out the Kubelet out-of-the-box dashboard. Here is a Kubelet dashboard in Sysdig Monitor.



How to monitor Controller Manager

Monitoring kube-controller-manager is important, as it is a main component of Kubernetes control plane. Kube-controller-manager runs in master nodes and it takes care of the different controller processes. These controllers watch the status of the different services deployed through the API and take corrective actions in case real and desired status don't match.

Kube-controller-manager takes care of nodes, workloads (replication controllers), namespaces (namespace controller), and service accounts (serviceaccount controller), among other things.



Getting metrics from kube-controller-manager

Controller-manager has been instrumented and it exposes Prometheus metrics by default, providing information about work-queues and requests to the API. This endpoint can be easily scraped, obtaining all of this information without any calculation.

Disclaimer: The following steps might not work if you used `kubeadm` with default values to deploy your Kubernetes cluster. `Kubeadm` sets the `kube-controller-manager` [bind-address as 127.0.0.1](#) by default. For that reason, a regular Prometheus instance without a network interface in the host network won't be able to reach the metrics endpoint.

We can test the endpoint running a curl from a pod with network access in master nodes:

```
$ curl -k -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" https://localhost:10257/metrics
```

It will return a long list of metrics with this structure (truncated):

```
# HELP apiserver_audit_event_total [ALPHA] Counter of audit events generated and
sent to the audit backend.
# TYPE apiserver_audit_event_total counter
apiserver_audit_event_total 0
# HELP apiserver_audit_requests_rejected_total [ALPHA] Counter of apiserver
requests rejected due to an error in audit logging backend.
# TYPE apiserver_audit_requests_rejected_total counter
apiserver_audit_requests_rejected_total 0
# HELP apiserver_client_certificate_expiration_seconds [ALPHA] Distribution of
the remaining lifetime on the certificate used to authenticate a request.
# TYPE apiserver_client_certificate_expiration_seconds histogram
apiserver_client_certificate_expiration_seconds_bucket{le="0"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="1800"} 0
...
```

If we want to configure a Prometheus to scrape API endpoint, we can add this job to our targets:

```
scrape_configs:
- honor_labels: true
  job_name: kubernetes-pods
  kubernetes_sd_configs:
  - role: pod
  relabel_configs:
  - action: keep
    regex: true
    source_labels:
    - __meta_kubernetes_pod_annotation_prometheus_io_scrape
  - action: replace
    regex: (https?)
    source_labels:
    - __meta_kubernetes_pod_annotation_prometheus_io_scheme
    target_label: __scheme__
  - action: replace
    regex: (.+)
    source_labels:
    - __meta_kubernetes_pod_annotation_prometheus_io_path
```

```

    target_label: __metrics_path__
  - action: replace
    regex: (.+?)(?:\d+)?;(\d+)
    replacement: $1:$2
    source_labels:
      - __address__
      - __meta_kubernetes_pod_annotation_prometheus_io_port
    target_label: __address__
  - action: labelmap
    regex: __meta_kubernetes_pod_annotation_prometheus_io_param_(.+)
    replacement: __param_$1
  - action: labelmap
    regex: __meta_kubernetes_pod_label_(.+)
  - action: replace
    source_labels:
      - __meta_kubernetes_namespace
    target_label: namespace
  - action: replace
    source_labels:
      - __meta_kubernetes_pod_name
    target_label: pod
  - action: drop
    regex: Pending|Succeeded|Failed|Completed
    source_labels:
      - __meta_kubernetes_pod_phase

```

In addition, we need to add annotations to the pod, so we have to modify the manifest in the master node located in `/etc/kubernetes/manifests/kube-controller-manager.yaml` and add these following annotations:

```

prometheus.io/scrape: "true"
prometheus.io/port: "10257"
prometheus.io/scheme: "https"

```

Monitoring the controller manager: What to look for?

Disclaimer: *kube-controller-manager metrics might differ between Kubernetes versions. Here, we used Kubernetes 1.25. You can check the metrics available for your version in the [Kubernetes repo](#).*

Number of kube-controller-manager instances: This value will give an idea of the general health of the kubelet in the nodes. The expected value is the number of nodes in the cluster. You can obtain this value counting targets found by Prometheus or by checking the process if you have low-level access to the node.

A possible PromQL query for a single stat graph would be:

```
sum(up{container="kube-controller-manager"})
```

Workqueue information: It provides metrics with information about workqueue to detect possible bottlenecks or issues processing different commands. We will focus on aggregated metrics from all of the controllers, but you have different metrics available for queues of various controllers, like AWS controller, node controller or service account controller.

- Workqueue latency: It's the time that kube-controller-manager is taking to fulfill the different actions to keep the desired status of the cluster. A good way to represent this are quantiles:

```
histogram_quantile(0.99, sum(rate(workqueue_queue_duration_seconds_
bucket{container="kube-controller-manager"}[5m])) by (instance, name, le))
```

- Workqueue rate: It's the number of required actions per unit time. A high value could indicate problems in the cluster of some of the nodes.

```
sum(rate(workqueue_adds_total{container="kube-controller-manager"}[5m])) by
(instance, name)
```

You may want to check the rate of additions to the kube-controller-manager workqueue. If that's the case, run the following query.

```
sum(rate(workqueue_adds_total{container="kube-controller-manager"}[5m])) by
(instance, name)
```

- Workqueue depth: It's the number of actions waiting in the queue to be performed. It should remain at low values.

The following query will allow you to easily see the increase rate in the kube-controller-manager queue. The bigger the workqueue is, the more that has to process, so a workqueue growing trend may indicate problems in your Kubernetes cluster.

```
sum(rate(workqueue_depth{container="kube-controller-manager"}[5m])) by
(instance, name)
```

Information about requests to Api-server: It provides information about requests performed to the api-server, so you can check that the connectivity is fine and that the api-server is providing the information needed to perform controller operations.

- Latency: Use this query if you want to calculate the 99th percentile of latencies on requests to the Kubernetes API server.

```
histogram_quantile(0.99, sum(rate(rest_client_request_duration_seconds_bucket{container="kube-controller-manager"}[5m])) by (url, le))
```

Request rate and errors: This metric provides the **number of HTTP client requests** for kube-controller-manager by HTTP response code.

```
sum(rate(rest_client_requests_total{container="kube-controller-manager",code=~"2.."}[5m]))
sum(rate(rest_client_requests_total{container="kube-controller-manager",code=~"3.."}[5m]))
sum(rate(rest_client_requests_total{container="kube-controller-manager",code=~"4.."}[5m]))
sum(rate(rest_client_requests_total{container="kube-controller-manager",code=~"5.."}[5m]))
```

Saturation metrics (requires node_exporter):

- CPU usage: The total **CPU time, spent in seconds, for kube-controller-manager by instance.**

```
rate(process_cpu_seconds_total{container="kube-controller-manager"}[5m])
```

- Memory usage: This metric measures the **amount of resident memory size in bytes for kube-controller-manager by instance.**

```
rate(process_resident_memory_bytes{container="kube-controller-manager"}[5m])
```


Examples of issues in kube-controller-manager

Workloads desired and current status mismatch

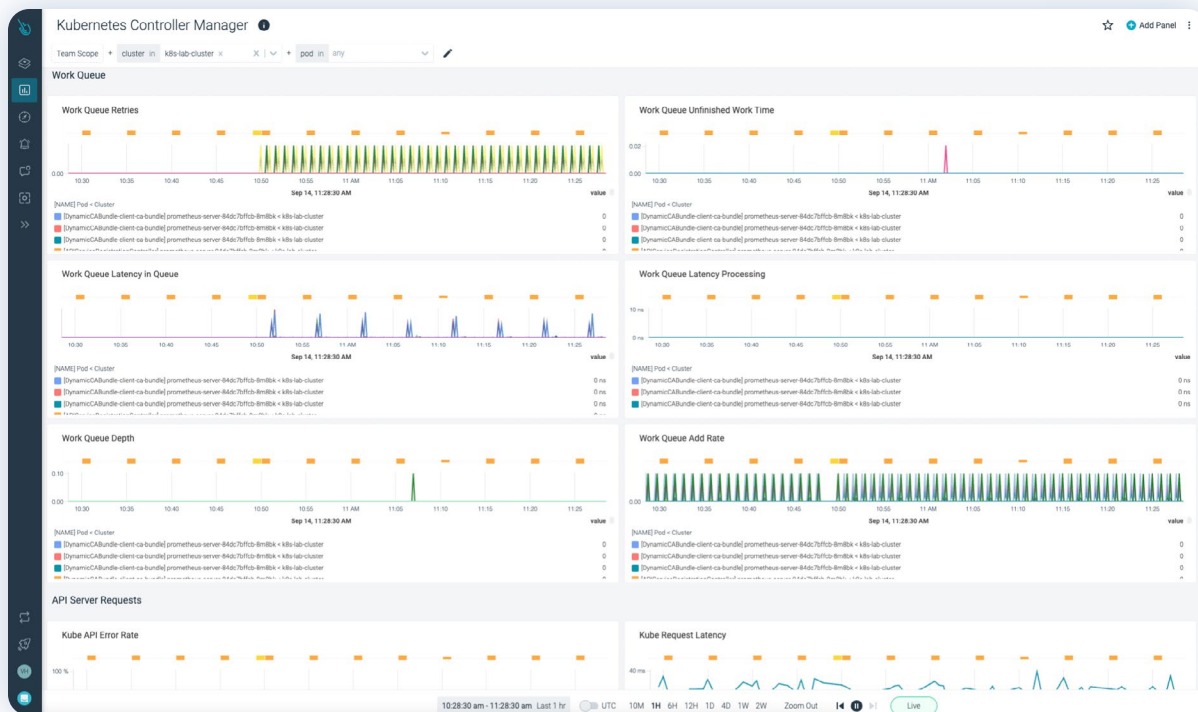
This can be caused by many different issues, but since the kube-controller-manager is the main component responsible with harmonizing current and desired status, we have a possible origin of the issue. Check that the kube-controller-manager instance is up and that the latency of API requests and workqueue are under normal values.

Kubernetes seems to be slow performing operations.

Check the latency and depth of workqueue in kube-controller-manager. It may have issues performing the actions with the API.

Monitoring kube-controller-manager metrics in Sysdig Monitor

In order to get controller manager monitoring in Sysdig Monitor, just run the Sysdig agent on your Kubernetes nodes, as we explained above in “Monitoring kubernetes control plane in Sysdig Monitor”. Log into the Sysdig portal, and check out the kube-controller-manager out-of-the-box dashboard. Here is a controller manager dashboard in Sysdig Monitor.



How to monitor etcd

The motivation of etcd is to provide a distributed key-value dynamic database that maintains a “configuration registry”. This registry is one of the foundations of a Kubernetes cluster service directory, peer discovery, and centralized configuration management. It bears a certain resemblance to a Redis database, classical LDAP configuration backends, or even the Windows Registry if you are more familiar with those technologies.

According to its developers, etcd aims to be:

- Simple: well-defined, user-facing API (JSON and gRPC)
- Secure: automatic TLS with optional client cert authentication
- Fast: benchmarked 10,000 writes/sec
- Reliable: properly distributed using Raft

Kubernetes uses the etcd distributed database to store its REST API objects (under the /registry directory key): pods, secrets, daemonsets, deployments, namespaces, events, etc.

[Raft](#) is a “consensus” algorithm. This is a method to achieve value convergence over a distributed and fault-tolerant set of cluster nodes.

In the event of an etcd quorum lost, and it is unable to elect a new leader, the current pods and workloads would keep running but no new changes can be made. Not even new Pods could be scheduled. [That's why monitoring etcd is crucial](#).

Without going into the gory details that you will find in the referenced articles, here are the basics of what you need to know:

Node status can be one of:

- Follower
- Candidate (briefly)
- Leader

How does the election process work?

1. If a Follower cannot locate the current Leader, it will become Candidate.
2. The voting system will elect a new Leader among the Candidates.
3. Registry value updates (commits) always go through the Leader.
4. Once the Leader has received the ACK from the majority of Followers, the new value is considered “committed”.
5. The cluster will survive as long as most of the nodes remain alive.

Perhaps the most remarkable features of etcd are the straightforward way of accessing the service using REST-like HTTP calls. That makes integrating third-party agents as simple as you can get, as well as its master-master protocol which automatically elects the cluster Leader and provides a fallback mechanism to switch this role if needed.

You can run etcd in Kubernetes, inside Docker containers, or as an independent cluster (in virtual machines or directly bare-metal). Usually, for simple scenarios, etcd is deployed in a Docker container

like other Kubernetes services, such as the API server, controller-manager, scheduler, or Kubelet. In more advanced scenarios, etcd is often an external service. In these cases, you will normally see three or more nodes to achieve the required redundancy.

When running etcd to store your data for your distributed systems or Kubernetes clusters, it is recommended to use an [odd number of nodes](#). Quorum needs a majority of nodes in the cluster to agree on updates to the cluster state. For a cluster with n number of nodes, the quorum needed to make up a cluster is $(n/2)+1$. For example:

- For a three nodes cluster, quorum will be achieved with two nodes. (Failure tolerance one node).
- For a four nodes cluster, quorum will be achieved with three nodes. (Failure tolerance one node).
- For a five nodes cluster, quorum will be achieved with three nodes. (Failure tolerance two nodes).

An odd-size cluster tolerates the same number of failures as an even-size cluster, but with fewer nodes. In addition, in the event of a network partition, an odd number of nodes guarantees that there will always be a majority partition, avoiding the frightening split-brain scenario. This way, the etcd cluster can keep operating and being the source of truth when the network partition is resolved.

Getting metrics from etcd

Etcd has been instrumented and it exposes [Prometheus metrics](#) by default in the port 2379 of the master host, providing information of the storage. This endpoint can be easily scraped, obtaining useful information without the need for additional scripts or exporters.

You can't scrape etcd metrics accessing the port in the node directly without authentication. The etcd is the core of any Kubernetes cluster, so its metrics are securitized too. To get the metrics, you need to have access to the port 2379 or be in the master itself, and you also need to have the client certificates. If you have access to the master node, just do a curl from there with the client certificate paths. The certificate is in `/etc/kubernetes/pki/etcd/server.crt` and the key `/etc/kubernetes/pki/etcd/server.key`.

```
$ curl https://localhost:2379/metrics -k --cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key
```

If you want to connect from outside of the master node, and you got the certificates from the master node and also have the port 2379 open, then you can access it with the IP too.

```
$ curl https://[master_ip]:2379/metrics -k --cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key
```

It will return a long list of metrics with this structure (truncated):

```
# HELP etcd_disk_backend_snapshot_duration_seconds The latency distribution of
backend snapshots.
# TYPE etcd_disk_backend_snapshot_duration_seconds histogram
etcd_disk_backend_snapshot_duration_seconds_bucket{le="0.01"} 0
etcd_disk_backend_snapshot_duration_seconds_bucket{le="0.02"} 0
etcd_disk_backend_snapshot_duration_seconds_bucket{le="0.04"} 0
etcd_disk_backend_snapshot_duration_seconds_bucket{le="0.08"} 0
etcd_disk_backend_snapshot_duration_seconds_bucket{le="0.16"} 0
etcd_disk_backend_snapshot_duration_seconds_bucket{le="0.32"} 3286
etcd_disk_backend_snapshot_duration_seconds_bucket{le="0.64"} 4617
etcd_disk_backend_snapshot_duration_seconds_bucket{le="1.28"} 4620
etcd_disk_backend_snapshot_duration_seconds_bucket{le="2.56"} 4620
etcd_disk_backend_snapshot_duration_seconds_bucket{le="5.12"} 4620
etcd_disk_backend_snapshot_duration_seconds_bucket{le="10.24"} 4620
etcd_disk_backend_snapshot_duration_seconds_bucket{le="20.48"} 4620
etcd_disk_backend_snapshot_duration_seconds_bucket{le="40.96"} 4620
etcd_disk_backend_snapshot_duration_seconds_bucket{le="81.92"} 4620
etcd_disk_backend_snapshot_duration_seconds_bucket{le="163.84"} 4620
etcd_disk_backend_snapshot_duration_seconds_bucket{le="327.68"} 4620
etcd_disk_backend_snapshot_duration_seconds_bucket{le="655.36"} 4620
etcd_disk_backend_snapshot_duration_seconds_bucket{le="+Inf"} 4620
etcd_disk_backend_snapshot_duration_seconds_sum 1397.2374600930025
etcd_disk_backend_snapshot_duration_seconds_count 4620
# HELP etcd_disk_wal_fsync_duration_seconds The latency distributions of fsync
called by wal.
# TYPE etcd_disk_wal_fsync_duration_seconds histogram
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.001"} 4.659349e+06
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.002"} 7.276276e+06
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.004"} 8.589085e+06
```

If you want to configure a Prometheus to scrape etcd, you have to mount the certificates and create the job:

The certificates are located in the master node in **/etc/kubernetes/pki/etcd/server.key** and **/etc/kubernetes/pki/etcd/server.crt**, just download them and create the secrets on Kubernetes with the next command.

Disclaimer: The etcd is the core of any Kubernetes cluster. if you don't take care with the certificates, you can expose the entire cluster and potentially be a target.

```
$ kubectl -n monitoring create secret generic etcd-ca --from-file=server.key
--from-file=server.crt
```

```
$ kubectl -n monitoring patch deployment prometheus-server -p '{"spec":{"template":{"spec":{"volumes":[{"name":"etcd-ca","secret":{"defaultMode":420,"secretName":"etcd-ca"}}]}}}}'
$ kubectl -n monitoring patch deployment prometheus-server -p '{"spec":{"template":{"spec":{"containers":[{"name":"prometheus-server","volumeMounts":[{"mountPath":"/opt/prometheus/secrets","name":"etcd-ca"}]}}]}}}}'
```

```
scrape_configs:
  ...
  - job_name: etcd
    scheme: https
    kubernetes_sd_configs:
      - role: pod
    relabel_configs:
      - action: keep
        source_labels:
          - __meta_kubernetes_namespace
          - __meta_kubernetes_pod_name
        separator: '/'
        regex: 'kube-system/etcd.+'
      - source_labels:
          - __address__
        action: replace
        target_label: __address__
        regex: (.+?)(\\:\\d)?
        replacement: $1:2379
      tls_config:
        insecure_skip_verify: true
    cert_file: /opt/prometheus/secrets/server.crt
    key_file: /opt/prometheus/secrets/server.key
```

You can customize your own labels and relabeling configuration.

Monitoring etcd: What to look for?

Disclaimer: etcd metrics might differ between Kubernetes versions. Here, we used Kubernetes 1.25. You can check the metrics available for your version in the [Kubernetes repo](#).

etcd node availability: An obvious error scenario for any cluster is that you lose one of the nodes. The cluster will continue operating, but it is probably a good idea to receive an alert, diagnose, and recover before you continue losing nodes and risk facing the next scenario: total service failure. The simplest way to check this is with a PromQL query:

```
sum(up{job=\"etcd\"})
```

This should give the number of nodes running. If you detect a missing member or the number is 0, there is a problem.

etcd has a leader: One key metric is to know if all nodes have a leader. If one node does not, it will be unavailable, if all nodes don't have a leader, then the cluster will become totally unavailable. There is a metric that says whether a node has a leader or not.

```
# HELP etcd_server_has_leader Whether or not a leader exists. 1 is existence, 0 is not.
# TYPE etcd_server_has_leader gauge
etcd_server_has_leader 1
```

etcd leader changes: The leader can change over time, but if it changes too often, these changes can impact the performance of the etcd itself. This can also be a signal of the leader being unstable because of connectivity problems, or maybe etcd has too much load.

```
# HELP etcd_server_leader_changes_seen_total The number of leader changes seen.
# TYPE etcd_server_leader_changes_seen_total counter
etcd_server_leader_changes_seen_total 1
```

Consensus proposal: A proposal is a request (i.e., a write request, a configuration change request) that needs to go through raft protocol. The proposal metrics have four different types: committed, applied, pending, and failed. All four can give information about the problems the etcd can face, but the most important is the failed type. If there are proposals failed, it can be for two reasons: the leader election is failing, or there is a loss of quorum.

For example, if we wanted to set an alert to show that there were more than five consensus proposals failed over the course of a 15 minute period, we could use this statement:

```
rate(etcd_server_proposals_failed_total{job=~"etcd"}[15m]) > 5
```

```
# HELP etcd_server_proposals_applied_total The total number of consensus
proposals applied.
# TYPE etcd_server_proposals_applied_total gauge
etcd_server_proposals_applied_total 1.3605153e+07
# HELP etcd_server_proposals_committed_total The total number of consensus
proposals committed.
# TYPE etcd_server_proposals_committed_total gauge
etcd_server_proposals_committed_total 1.3605153e+07
# HELP etcd_server_proposals_failed_total The total number of failed proposals
seen.
# TYPE etcd_server_proposals_failed_total counter
etcd_server_proposals_failed_total 0
# HELP etcd_server_proposals_pending The current number of pending proposals to
commit.
# TYPE etcd_server_proposals_pending gauge
etcd_server_proposals_pending 0
```

Disk sync duration: As etcd is storing all important things about Kubernetes. The speed of committing changes to disk and the health of your storage are key indicators if etcd is working properly. If the disk sync has high latencies, then the disk may have issues or the cluster can become unavailable. The metrics that show this are **wal_fsync_duration_seconds** and **backend_commit_duration_seconds**.

```
# HELP etcd_disk_backend_commit_duration_seconds The latency distributions of
commit called by backend.
# TYPE etcd_disk_backend_commit_duration_seconds histogram
etcd_disk_backend_commit_duration_seconds_bucket{le="0.001"} 0
etcd_disk_backend_commit_duration_seconds_bucket{le="0.002"} 5.402102e+06
etcd_disk_backend_commit_duration_seconds_bucket{le="0.004"} 6.0471e+06
...
etcd_disk_backend_commit_duration_seconds_sum 11017.523900176226
etcd_disk_backend_commit_duration_seconds_count 6.157407e+06
# HELP etcd_disk_wal_fsync_duration_seconds The latency distributions of fsync
called by wal.
# TYPE etcd_disk_wal_fsync_duration_seconds histogram
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.001"} 4.659349e+06
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.002"} 7.276276e+06
```

```
...
etcd_disk_wal_fsync_duration_seconds_sum 11580.35429902582
etcd_disk_wal_fsync_duration_seconds_count 8.786736e+06
```

To know if the duration of the backend commit is good enough, you can visualize in a histogram. With the next command, you can show the time latency in which 99% of requests are covered.

```
histogram_quantile(0.99,
rate(etcd_disk_backend_commit_duration_seconds_bucket{job=~"etcd"}[5m]))
```

Network peer round trip time: This metric is key and **extremely important to measure how your network and etcd nodes are performing**. It indicates **the RTT (round trip time) latency** for etcd to replicate a request between etcd members. A high latency or latencies growing over time **may indicate issues in your network, causing serious trouble** with etcd requests and even loss of quorum. There is one complete histogram per communication (from peer to peer). This value should not exceed 50ms (0.050s).

```
# HELP etcd_network_peer_round_trip_time_seconds Round-Trip-Time histogram
between peers
# TYPE etcd_network_peer_round_trip_time_seconds histogram
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.0001"} 0
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.0002"} 0
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.0004"} 0
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.0008"} 46
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.0016"} 55
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.0032"} 59
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.0064"} 59
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.0128"} 112
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.0256"} 116
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.0512"} 116
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.1024"} 116
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.2048"} 116
```

```

etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.4096"} 116
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="0.8192"} 116
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="1.6384"} 116
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="3.2768"} 116
etcd_network_peer_round_trip_time_seconds_
bucket{To="bf52b1f71736a183",le="+Inf"} 116
etcd_network_peer_round_trip_time_seconds_sum{To="bf52b1f71736a183"}
0.7280363210000003
etcd_network_peer_round_trip_time_seconds_count{To="bf52b1f71736a183"} 116

```

To know if the RTT latencies between etcd nodes are good enough, run the following query to visualize the data in a histogram.

```

histogram_quantile(0.99,
rate(etcd_network_peer_round_trip_time_seconds_bucket[5m]))

```

etcd network peer sent failures total: This metric provides the total number of failures sent by peer or etcd member. It can help to understand better whether a specific node is facing performance or network issues.

etcd network peer received failures total: The same way the previous metric provided sent failures, this time, what is measured is the total received failures by peer.

Examples of issues in etcd

Most of the time, your etcd cluster works so neatly that it is easy to forget its nodes are running. Keep in mind, however, that Kubernetes absolutely needs this registry to function. A major etcd failure will seriously cripple or even take down your container infrastructure. Pods currently running will continue to run, but you cannot execute any further operations on them. When you reconnect etcd and Kubernetes, state incoherences could cause additional malfunction.

Losing quorum

Sometimes, the cluster loses quorum and the etcd goes into a read-only state. Once you lose quorum, you can still see how the cluster is but you cannot take any action because it will be unable to decide if it is permitted.

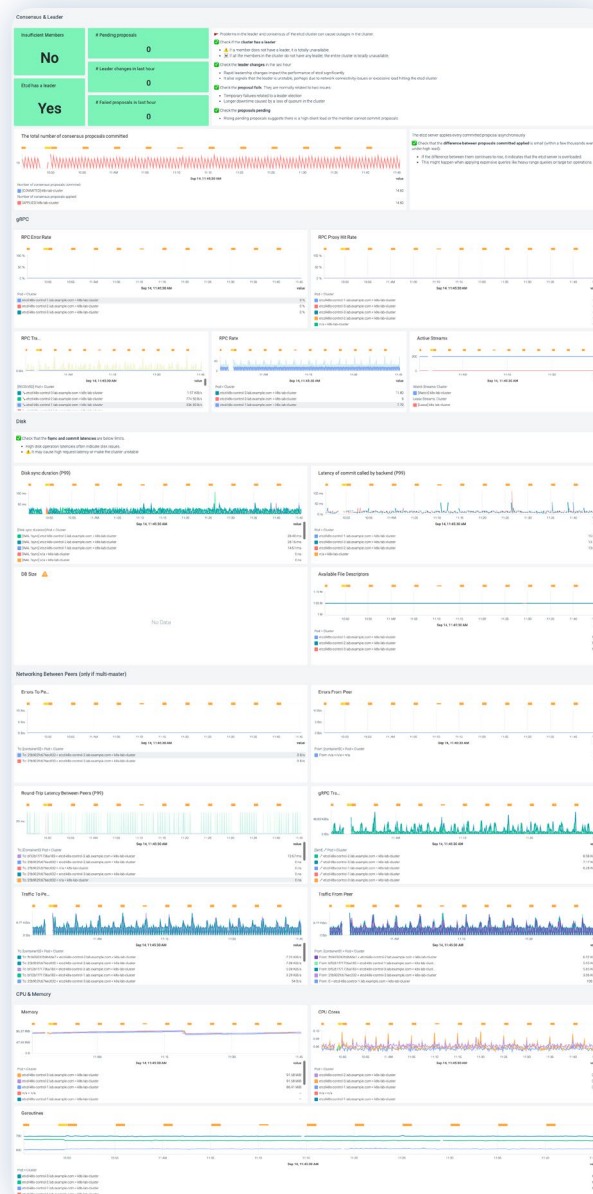
Apply entries took too long

If you see the message *apply entries took too long*, that's because the average apply duration exceeds 100 milliseconds. This issue can be caused by three factors:

- Slow disk
- Cpu throttling
- Slow network

Monitoring etcd metrics in Sysdig Monitor

If you want to monitor etcd using Sysdig Monitor, just run the Sysdig agent on your Kubernetes nodes, as we explained above in “Monitoring kubernetes control plane in Sysdig Monitor”. Log into the Sysdig portal, and check out the etcd out-of-the-box dashboard. Here is the out-of-the-box etcd dashboard in Sysdig Monitor.



Alerting on the Kubernetes control plane

Monitoring and alerting at the container orchestration level is two-fold. On one side, you need to monitor if the services handled by Kubernetes meet the requirements you defined. On the other hand, you need to make sure all of the components of Kubernetes are up and running.

Etcd has Insufficient Peers

Insert alert description

Low

1 Define

a PromQL

```
count(up{job="etcd"} == 0) > (count(up{job="etcd"}) / 2 - 1)
```

b Duration

For the duration of 10 minute

Is Kubernetes etcd running?

etcd is the distributed service discovery, communication, and command channel for Kubernetes. Monitoring etcd can go as deep as monitoring a distributed key value database, but we'll keep things simple here; etcd works if more than half of the configured instances are running, so let's alert this.

PromQL query:

```
count(up{job="etcd"} == 0) > (count(up{job="etcd"} == 0) / 2 - 1)
```

Is the Kubernetes API server running?

The Kubernetes API server is the center of the Kubernetes control plane. Let's configure an alert if the service goes down.

Kube Api Server is Down

Insert alert description

High

1 Define

a PromQL

```
(absent(up{job="kube-apiserver"})) == 1
```

b Duration

For the duration of 10 minute

PromQL query:

```
(absent(up{job="kube-apiserver"})) == 1
```

Is the latency of Kubelet too high for the start of the pods?

Kubelet is a very important service inside Kubernetes' control plane. It's the component that runs the containers described by pods in the nodes. That means you can *Golden Signal* this and check the pod start rate and duration. High latency here could indicate performance degradation on the container runtime, or issues trying to access the container images.

Kubelet Pod Start Up Latency is High

Insert alert description

Low ▼

1 Define

a PromQL ?

```
(histogram_quantile(0.99, sum(rate(kubelet_pod_worker_duration_seconds_bucket{job="kubelet"}[5m])) by (instance, le)) * on(instance) group_left(node) kubelet_node_name) > 60
```

b Duration ?

For the duration of 10 minute ▼

PromQL query:

```
(histogram_quantile(0.99, sum(rate(kubelet_pod_worker_duration_seconds_bucket{job="kubelet"}[5m])) by (instance, le)) * on(instance) group_left(node) kubelet_node_name ) > 60
```

Lessons learned

1. Monitoring the Kubernetes API server is fundamental, since it's a key piece in the cluster operation. Remember, all of the communication between the cluster components is done via **kube-apiserver**.
2. Monitoring Kubelet is important. All of the communication with the container runtime is done through Kubelet. It's the connection between Kubernetes and the OS running behind.
3. Remember that kube-controller-manager is responsible for having the correct number of elements in all of the deployments, daemonsets, persistent volume claims, and many other Kubernetes elements.
4. Some issues in your Kubernetes cluster that appear to be random can be explained by a problem in the API server or Kubelet. Monitoring API server and Kubelet metrics can save you time when these problems come, and they will.

Sysdig helps you follow Kubernetes monitoring best practices, which is just as important as monitoring your workloads and applications running inside the cluster. Don't forget to monitor your control plane!

Monitoring Kubernetes Workloads

Services and Resources

Monitoring services running on Kubernetes

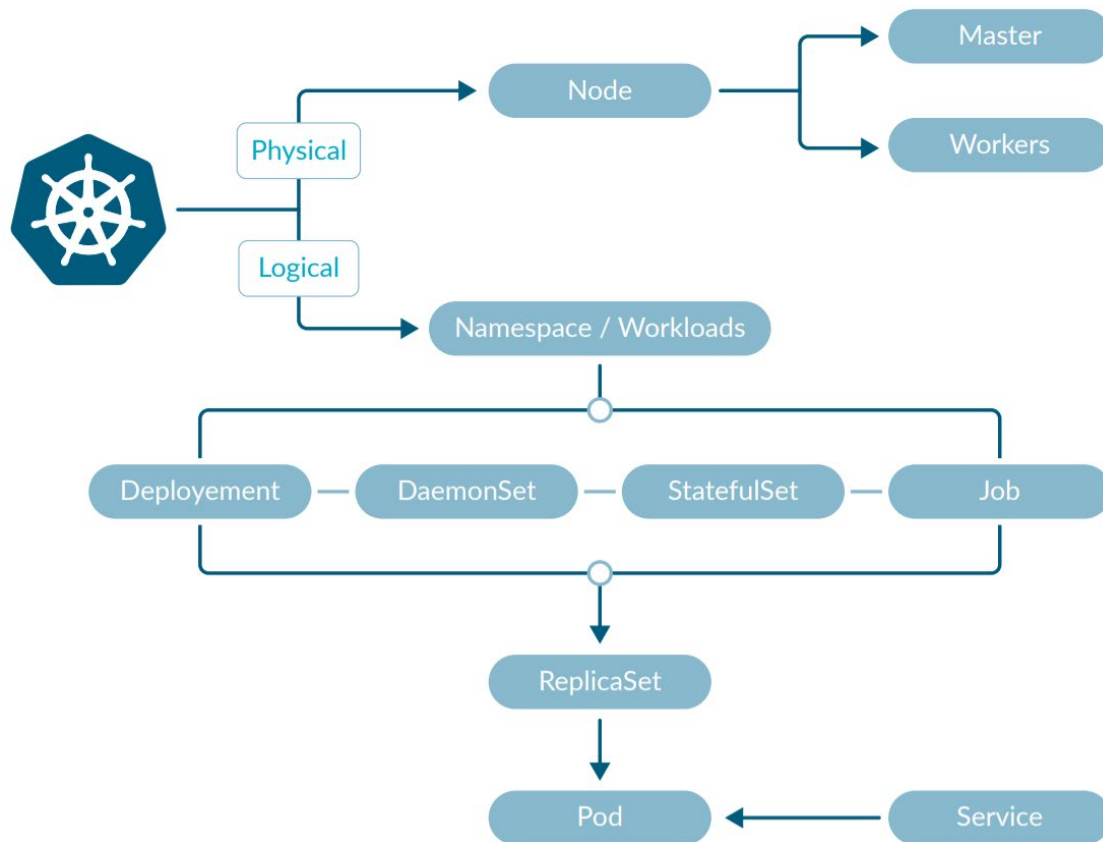
When looking at the service level, it shouldn't be much different from what you were doing before Kubernetes if you had already clustered your services. Think of databases like MySQL/MariaDB or MongoDB, where you will look at the replication status and lag. Is there anything to take into account now?

The answer is yes! If you want to know how your service operates and performs globally, you will need to leverage your monitoring tool capabilities to do metric aggregation and segmentation based on container metadata.

You know Kubernetes tags containers within a deployment or exposed through a service, as we explained in [Chapter 2: Intro to Kubernetes Monitoring](#). Now, you need to take that into account when you define your alerts. For example, scoping alerts only for the production environment, most likely defined by a namespace.

Kubernetes workloads hierarchy

Applications in Kubernetes are not flat. They are structured in several hierarchical layers that define how the services are structured outside the Kubernetes world.



Namespaces

It is the main division of the Kubernetes cluster. Usually namespaces match a logical separation in the different applications running.

Namespaces can be a very good and intuitive way to segment information collected and to delimitate access and responsibilities.

Workloads

These are the basic components of an application. Usually, a microservice is composed of one or more workloads with relations between them.

There are four main kinds of workloads:

- **Deployment:** The most common. They are stateless services that run multiple instances of a pod.
- **DaemonSet:** A special deployment that creates one, and only one, pod in every node in the cluster.
- **StatefulSet:** Specially created for applications that need persistence upon restarts and consistency in the data.
- **Jobs and Cronjobs** designed to run specific jobs that do the task and finish.

ReplicaSet

These different workloads use a Replication Controller to take care of the number of replicas run for every workload. This object takes care of the current state and tries to harmonize that with the desired state.

Pods

It is the main object in Kubernetes. It is one or more containers running together as a unit with a set of configurations and sharing storage.

They are usually controlled by a workload and inherit all the parameters from there.

Services

They are abstractions that allow the cluster control to create the network structure to connect different pods between them and the outside world.

Importance of hierarchy in monitoring Kubernetes

In the Kubernetes world, the fundamental unit is the pod, but it can be a mistake to treat pods as independent objects. Instead, the different pods and elements are strongly coupled and it is important to keep that correlation when interpreting the data.

Information provided by a pod can be misleading if you don't take into account the aggregation of all the pods in a deployment. The state of a pod is not very important as pods can die and they are respawned by the system.

Fundamental parameters must be availability of the service, possible points of failure, and availability of resources.

Alerting on services running on Kubernetes

Do you have enough pods/containers running for each application?

Kubernetes has a few options to handle an application that has multiple pods: Deployments, ReplicaSets, and ReplicationControllers. There are slight differences between them but they can be used to maintain a number of instances in running the same application. There, the number of running instances can be changed dynamically if you scale up and down, and this process can even be automated with auto-scaling.

There are also multiple reasons why the number of running containers can change. That includes rescheduling containers in a different host because a node failed, or because there aren't enough resources and the pod was evicted (don't miss our [Understanding pod evicted](#) blog), a rolling deployment of a new version, and more.

If the number of replicas or instances running during an extended period of time is lower than what you desire, it's a symptom of something not working properly (not enough nodes or resources available, Kubernetes or Docker Engine failure, Docker image broken, etc.).

An alert that evaluates availability across all services is almost a must in any Kubernetes alerting setup.

```
kube_deployment_spec_replicas{job="kubernetes-service-endpoints"}
  !=
kube_deployment_status_replicas_available{job="kubernetes-service-endpoints"}
```

As we mentioned before, this situation is acceptable during container reschedule and migrations, so keep an eye on the configured `.spec.minReadySeconds` value for each container (time from container start until it becomes available in ready status). You might also want to check `.spec.strategy.rollingUpdate.maxUnavailable`, which defines how many containers can be taken offline during a rolling deployment.

The following is an example alert with this condition applied to a deployment `wordpress-wordpress` within a `wordpress` namespace in a cluster with name `kubernetes-dev`.

Wordpress LOW replicas running

There are no replicas running in the Wordpress deployment

warning

1 Define

a Metric
timeAvg(kubernetes.replicaSet.replicas.running) < timeAvg(kubern... [HELP](#)

b Scope

agent.tag.cluster-id	is	kubernetes-dev	and	X
kubernetes.namespace.na...	in	wordpress	and	X
kubernetes.deployment.na...	in	wordpress-wordpr...		X

Select scope

c Trigger
For the last 2 minute(s)
Simple Alert
Trigger a single alert when the trigger condition is met across your scope

Do you have any pod/containers for a given application?

Similar to the previous alert but with higher priority (this example is a candidate for getting paged in the middle of the night), you want to alert if there are no containers running at all for a given application.

In the example below, you apply the alert for the same deployment but trigger if running pods is fewer than one during one minute.

Wordpress NO replicas running

There are no replicas running in the Wordpress deployment

alert

1 Define

a Metric
Average of kubernetes.replicaSet.replicas.running

b Scope

agent.tag.cluster-id	is	kubernetes-dev	and	X
kubernetes.namespace.na...	is	wordpress	and	X
kubernetes.deployment.na...	is	wordpress-wordpr...		X

Select scope

c Trigger
If metric < 1 for the last 1 minute(s) as a rate
Simple Alert
Trigger a single alert when the trigger condition is met across your scope

Is there any pod/container in a restart loop?

When deploying a new version that's broken, if there aren't enough resources available or some requirements/dependencies aren't in place, you might end up with a container or pod continuously restarting in a loop. That's called [CrashLoopBackOff](#). When this happens, pods never get into ready status and, therefore, are counted as unavailable and not as running. This scenario is already captured by the alerts before. Still, it's a good idea to set up an alert that catches this behavior across our entire infrastructure and immediately identifies the specific problem. It's not the kind of alert that interrupts your sleep, but rather one that gives you useful information.

This is an example applied across the entire infrastructure detecting more than four restarts over the last two minutes.

Pod restart count CrashLoopBackOff

Pod restarted too many times, probably in CrashLoopBackOff state

warning

1 Define

a Metric

Average of `kubernetes.pod.restart.count`

b Scope

everywhere

c Trigger

If metric `>` `4` for the last `2` minute(s) on average

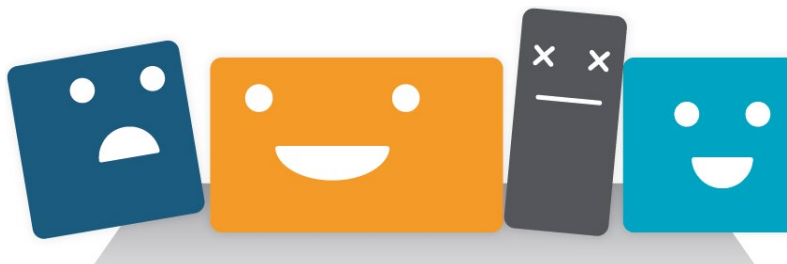
Multiple Alerts

Trigger a separate alert for each `kubernetes.pod.name`

+ Add another

Understanding Kubernetes limits and requests by example

How you set Kubernetes limits and requests is essential in optimizing application and cluster performance. One of the challenges of every distributed system designed to share resources between applications, like Kubernetes, is how to properly share the resources. Applications were typically designed to run standalone in a machine and use all of the resources at hand. It's said that good fences make good neighbors. The new landscape requires sharing the same space with others, and that makes quotas a hard requirement.



Resources

In terms of resources, we can differentiate between CPU resources and memory resources. You can set requests and limits for both types of resources.

CPU resources are measured in CPU units, where one CPU unit is equivalent to 1000m (millicpu). So if you want to set 300m, it will be equivalent to 0.3. In Kubernetes, one CPU unit is equivalent to one Physical CPU core or one Virtual CPU core, depending on the node architecture where Kubernetes is running.

Memory resources are measured in bytes. You can use any of these quantity suffixes (E, P, T, G, M, k). So if you plan to set either a 600M memory request or limit, the equivalent in kb would be 614400k, or 629145600 for bytes.

Namespace quotas

[Kubernetes allows administrators to set quotas, in namespaces](#), as hard limits for resource usage. This has an additional effect; if you set a CPU request quota in a namespace, then all pods need to set a CPU request in their definition, otherwise they will not be scheduled.

Let's look at an example:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-example
spec:
  hard:
    requests.cpu: 2
    requests.memory: 2Gi
    limits.cpu: 3
    limits.memory: 4Gi
```

If you apply this file to a namespace, you will set the following requirements:

- All pod containers have to declare requests and limits for CPU and memory.
- The sum of all the CPU requests can't be higher than two cores.
- The sum of all the CPU limits can't be higher than three cores.
- The sum of all the memory requests can't be higher than two GiB.
- The sum of all the memory limits can't be higher than four GiB.

If you already have 1.9 cores allocated with pods and try to allocate a new pod with 200m of CPU request, the pod will not be scheduled and will remain in a **pending** state.

Explaining pod requests and limits

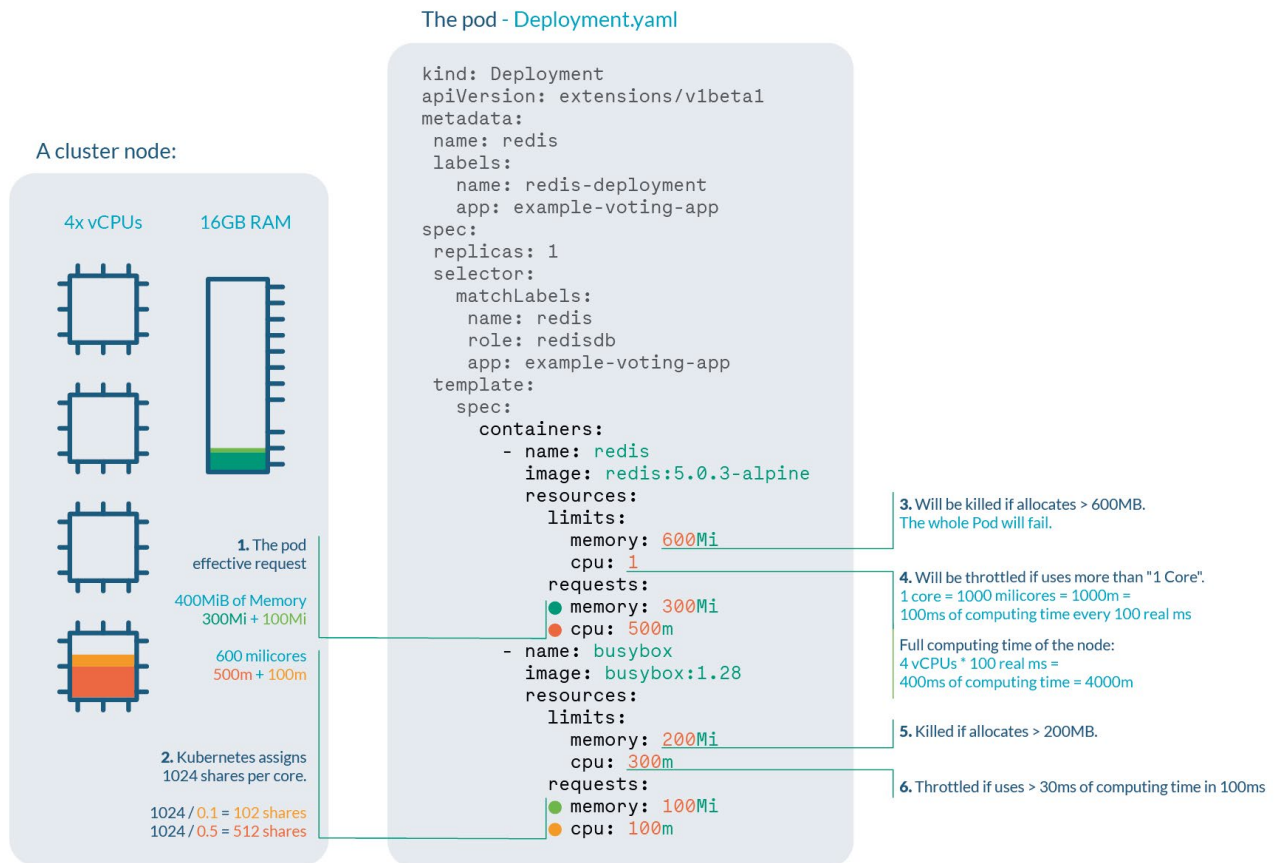
Kubernetes implements two mechanisms to grant and control the usage of resources:

- **Requests** are the resources you want to get guaranteed in a container. The real use of resources can be lower or higher than **Requests**, but never will exceed the **Limits**.
- **Limits** sets the amount of resources the container is allowed to use. If a container needs to exceed and go above this value, it will be denied, causing **OOMKill** for memory and **CPU throttling** when the limit is on the CPU.

Let's consider this example of a deployment:

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: redis
  labels:
    name: redis-deployment
    app: example-voting-app
spec:
  replicas: 1
  selector:
    matchLabels:
      name: redis
      role: redisdb
      app: example-voting-app
  template:
    spec:
      containers:
        - name: redis
          image: redis:5.0.3-alpine
          resources:
            limits:
              memory: 600Mi
              cpu: 1
            requests:
              memory: 300Mi
              cpu: 500m
        - name: busybox
          image: busybox:1.28
          resources:
            limits:
              memory: 200Mi
              cpu: 300m
            requests:
              memory: 100Mi
              cpu: 100m
```

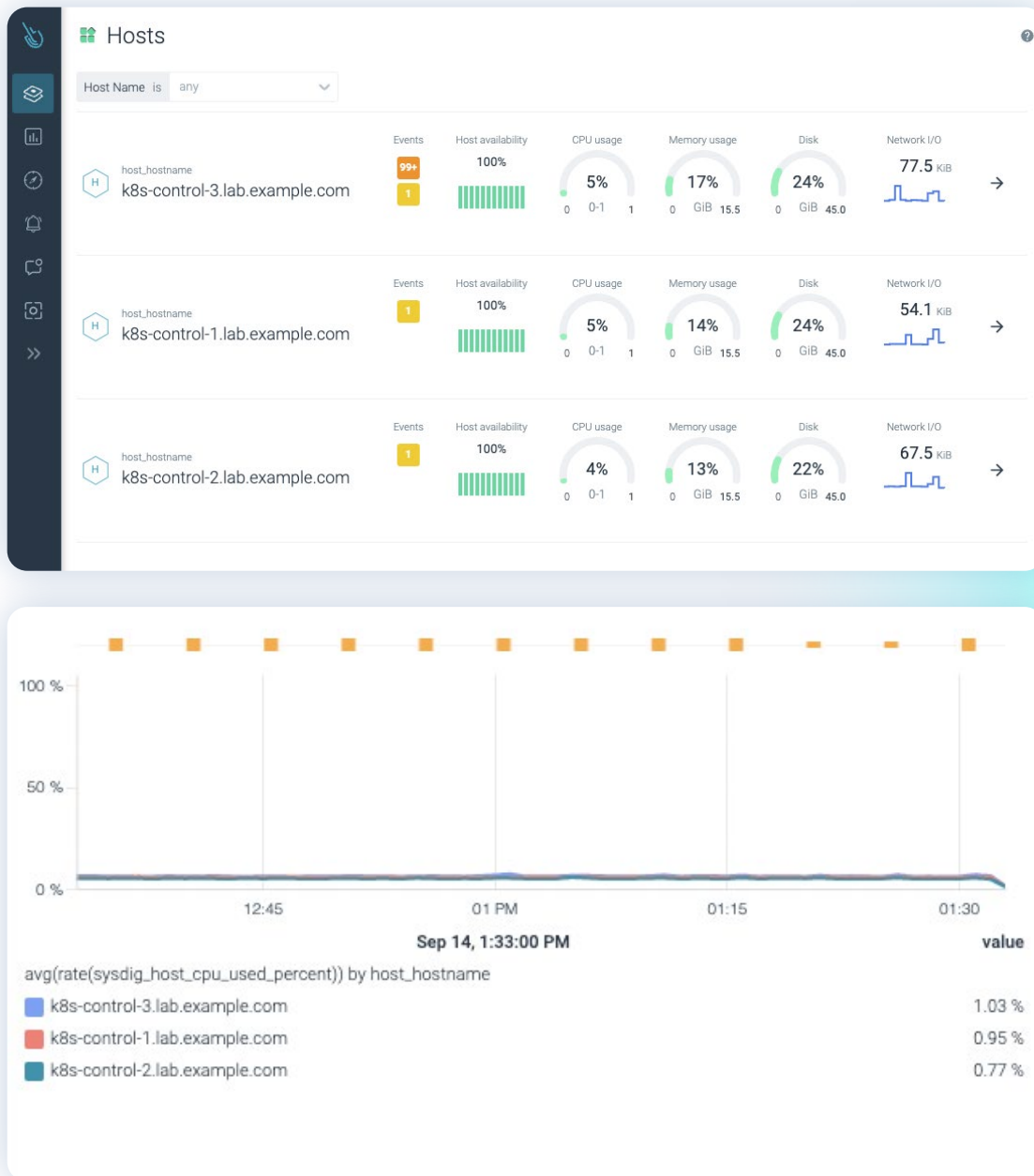
Let's say you are running a cluster with, for example, four cores and 16GB RAM nodes. You can extract a lot of information:



1. **Pod effective request** is 400 MiB of memory and 600 millicores of CPU. You need a node with enough free allocatable space to schedule the pod.
2. **CPU shares** for the redis container will be 512, and 102 for the busybox container. Kubernetes always assign 1,024 shares to every core, so:
 - redis: $1,024 / 0.5 \text{ cores} \approx 512$
 - busybox: $1,024 / 0.1 \text{ cores} \approx 102$
3. **Redis** container will be Out Of Memory (OOM) killed if it tries to allocate more than 600MB of RAM, most likely making the pod fail.
4. **Redis** will suffer **CPU throttle** if it tries to use more than 100ms of CPU in every 100ms, (since you have four cores, available time would be 400ms every 100ms) causing performance degradation.
5. **Busybox** container will be **OOM killed** if it tries to allocate more than 200MB of RAM, resulting in a failed pod.
6. **Busybox** will suffer **CPU throttle** if it tries to use more than 30ms of CPU every 100ms, causing performance degradation.

In order to detect problems, you should be monitoring:

- CPU and memory usage in the node. Memory pressure can trigger OOM kills if the node memory is full, despite all of the containers being under their limits. CPU pressure will throttle processes and affect performance.

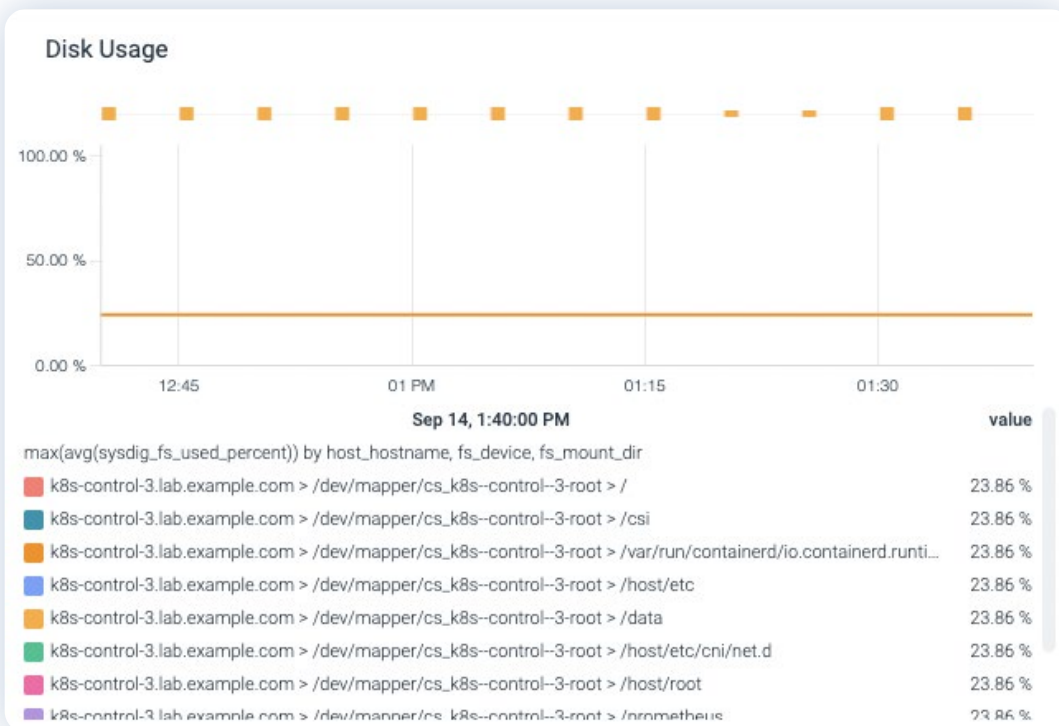


Find these metrics either in Sysdig Monitor in the dashboard:
Host infrastructure > Host Resource usage, or in Sysdig Advisor in Overview > Hosts



Find these metrics in Sysdig Monitor in the dashboard: Host infrastructure > Host Resource usage

- Disk space in the node. If the node runs out of disk, it will try to free disk space with a fair chance of pod eviction.



Find these metrics in Sysdig Monitor in the dashboard: Host infrastructure > Host Resource usage

- Percentage of CPU quota used by every **container**. Monitoring pod CPU usage can lead to errors. Remember, limits are per container, not per pod. Other CPU metrics, like **cpu shares used**, are only valid for allocating, so don't waste time on them if you have performance issues.



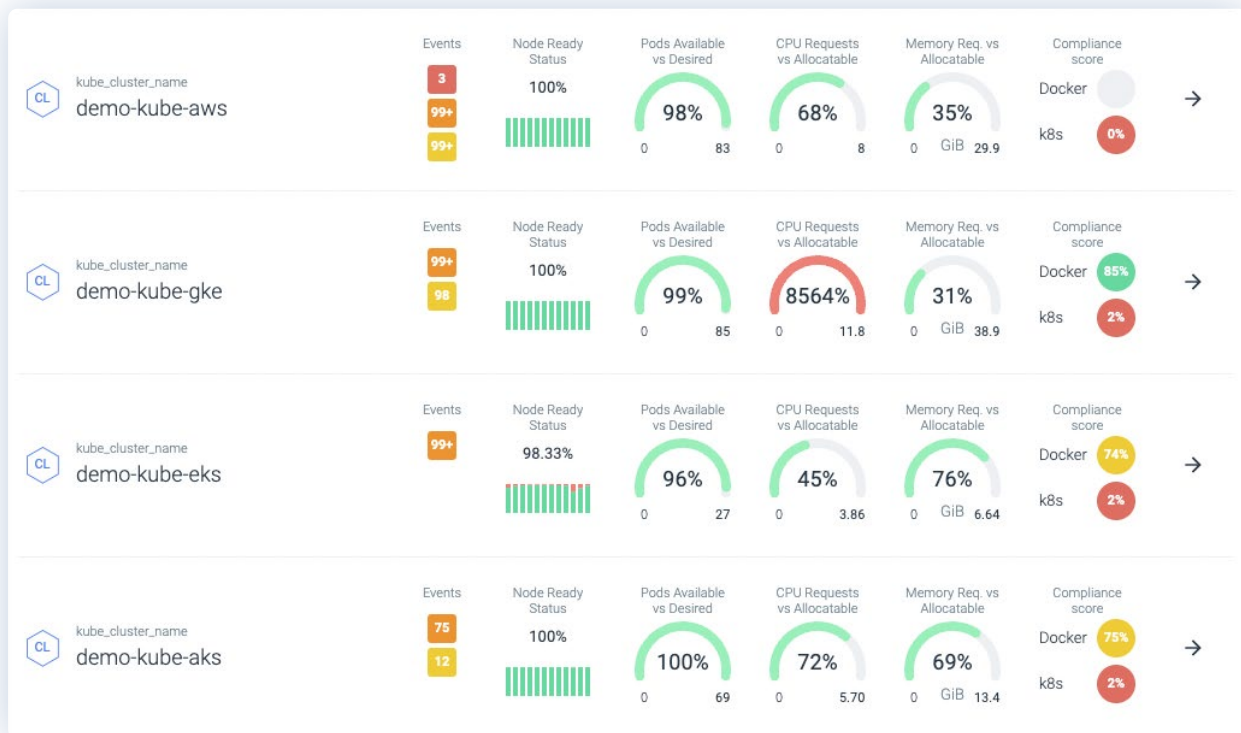
Find these metrics in Sysdig Monitor in the dashboard: Containers > Container CPU & Memory limits

- Memory usage per container. You can relate this value to the limit in the same graph or analyze the percentage of memory limit used. Don't use pod memory usage. A pod in the example can be using 300MiB of RAM, well under the pod effective limit (400MiB), but if the redis container is using 100MiB and the busybox container is using 200MiB, the pod will fail.



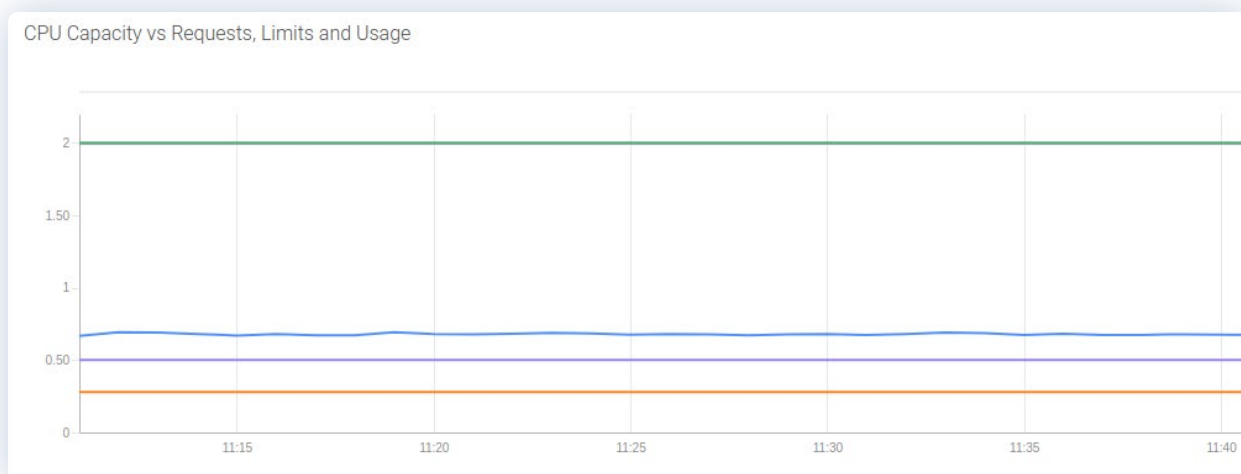
Find these metrics in Sysdig Monitor in the dashboard: Containers > Container CPU & Memory limits

- Percentage of resource allocation in the cluster and the nodes. You can represent this as a percentage of resources allocated from total available resources. A good warning threshold would be $(n-1)/n * 100$, where n is the number of nodes. Over this threshold, in case of a node failure, you wouldn't be able to reallocate your workloads in the rest of the nodes.



Find these metrics in Sysdig Monitor in the Sysdig Advisor Overview feature > clusters

- Limit overcommit (for memory and CPU). The best way to clearly see this is the percentage that the limit represents in the total allocatable resources. This can go over 100% in a normal operation.

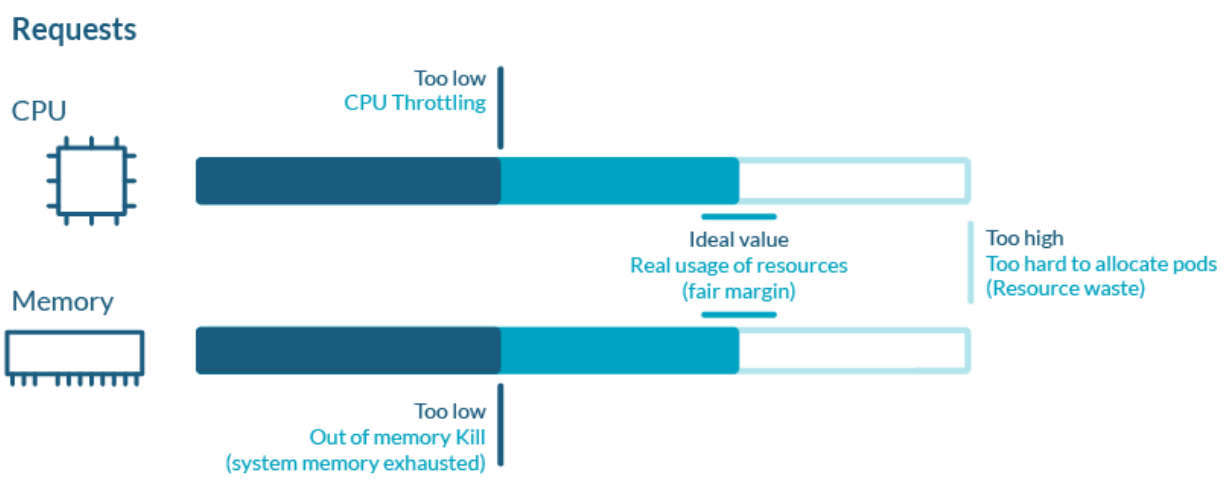


Custom graph showing CPU usage vs. capacity vs. limits vs. requests.

Choosing pragmatic requests and limits

When you have some experience with Kubernetes, you usually understand (the hard way) that properly setting requests and limits is of utmost importance for the performance of the applications and cluster.

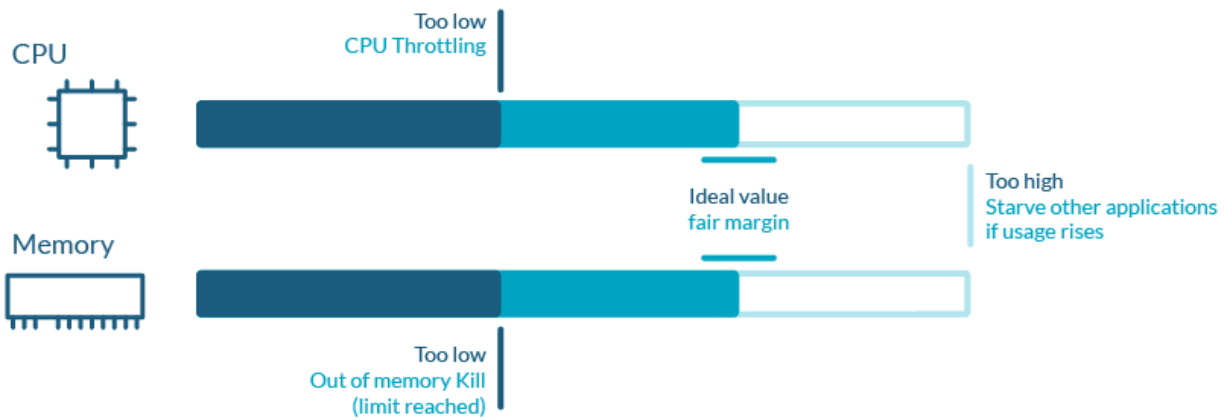
In an ideal world, your pods should be continuously using the exact amount of resources you requested. But the real world is a cold and fickle place, and resource usage is never regular or predictable. Consider a 25% margin up and down the request value as a good situation. If your usage is much lower than your request, you're wasting money. If it's higher, you're risking performance issues in the node.



Requests	Too low	Ideal value	Too High
CPU	CPU throttling	Real use of resources (fair margin)	Hard to allocate pods Resource waste
Memory	OOM kill (system memory exhausted)		

Regarding limits, achieving a good setting is a matter of trial and error. There is no optimal value for everyone as it hardly depends on the nature of the application, the demand model, the tolerance to errors, and many other factors.

Limits



Limits	Too low	Too High
CPU	CPU throttling	Starve other applications if usage rises
Memory	OOM kill (limit reached)	



Another thing to consider is the limit overcommit you allow on your nodes.

Limit overcommit

Conservative



Aggressive



Limit overcommit	Conservative Fewer than 125%	Aggressive More than 150%
Good	Limited risk of resource starvation	Increased resource exploitation
Warning	More chance of resource wasting	More risk of resource starvation in the node

The enforcement of these limits are on the user, as there is no automatic mechanism to tell Kubernetes how much overcommit to allow.

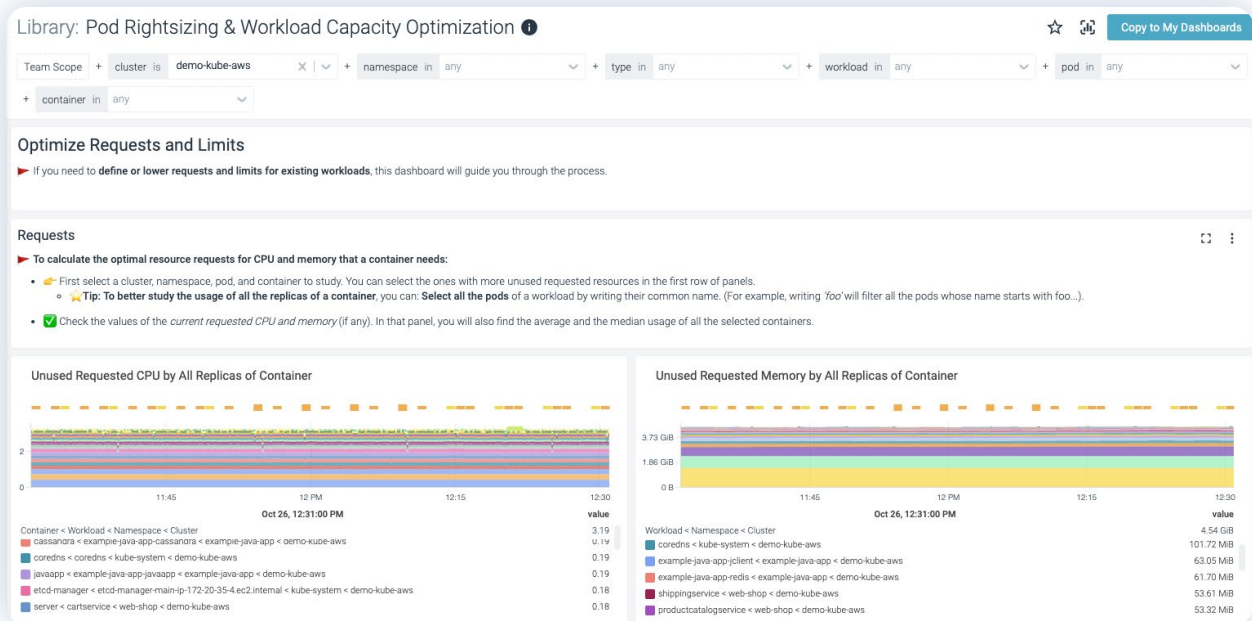
Reducing wasted spending

As mentioned in the previous section, if the usage of your containers' resources is much lower than the requests you set, you are undoubtedly wasting money.

Every time a request is set in a Pod, the kube-scheduler uses this information to decide on which node the Pod should run. As soon as the Pod is scheduled and started, it has the Requests resources guaranteed for its own use.

That's the main reason why it is so important to be accurate when using Requests in a Pod. It can harm your pocket if it is set too high, or can cause serious trouble if it's too low, jeopardizing the node's performance.

In order to avoid wasting resources and save money in your Kubernetes bill, you should consider monitoring the unused CPU and unused memory in your containers.



*Find these metrics in Sysdig Monitor in the dashboard:
Kubernetes > Pod Rightsizing & Workload Capacity Optimization.*

Cost monitoring and optimization

In terms of monitoring costs, according to the [CNCF FinOps for Kubernetes survey](#):

- 68% of people said their Kubernetes costs are rising.
- About 69% of respondents said they have no Kubernetes monitoring tool or only have Kubernetes costs estimates.

Also, according to our [2023 Security and Usage report](#):

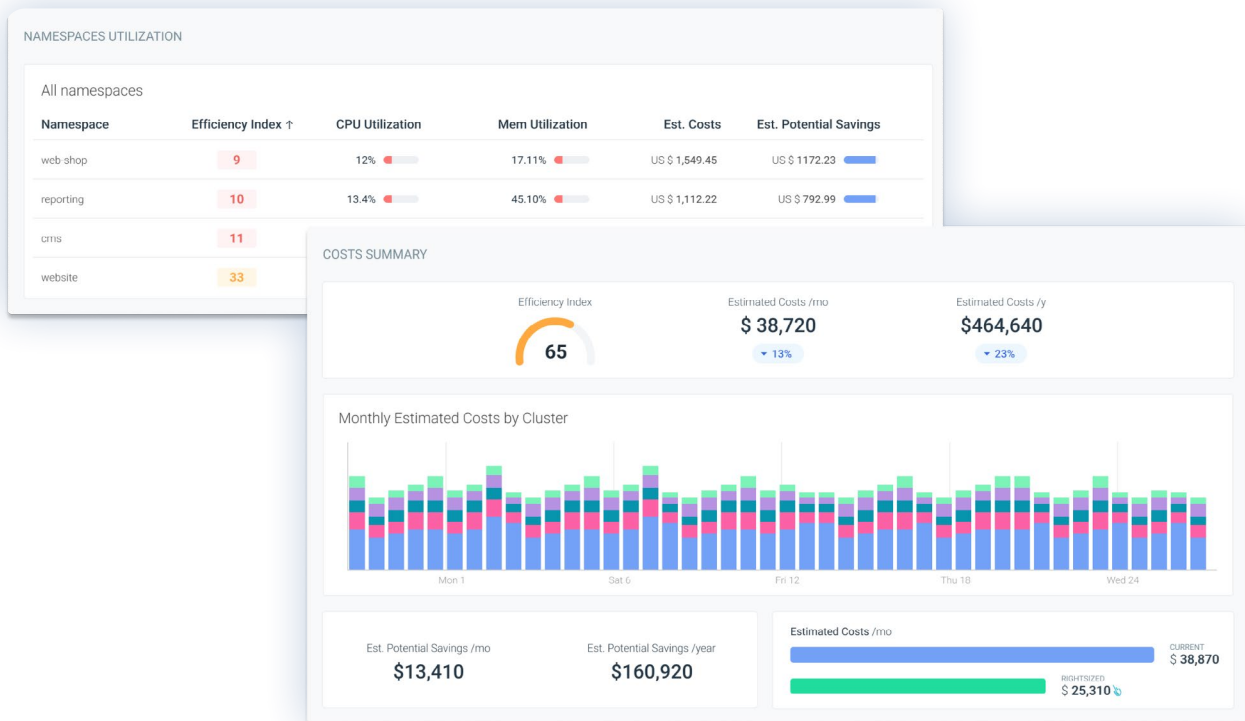
- Organizations with over 1000 nodes could be wasting more than \$10 million annually.
- On average, teams are wasting 69% of the CPU resources they have set for their environments.

Nowadays, many companies are in journeys to move to cloud native, using Kubernetes in one form or another, since in most of the cases, cloud service providers are the preferred option to host their workloads. Not implementing a cost monitoring and optimization strategy can be a huge problem for companies that can't foresee Kubernetes and cloud costs, and even worse, can't remediate and rightsize their workloads effectively.

Sysdig Monitor provides an out-of-the-box tool to help you optimize and rightsize your workloads, reducing the Kubernetes and cloud service providers bill at the end of the month.

[Cost Advisor](#) provides all the information on your resource consumption for your Kubernetes clusters, and automatically elaborates a cost optimization plan for you, providing cost-estimated savings and remediation plans to apply via a one-liner command.

You can export the Cost Advisor information and load it in a chargeback. This way, you can start applying FinOps best practices, promoting a culture of cost discipline where all the teams are reviewing and optimizing cost.



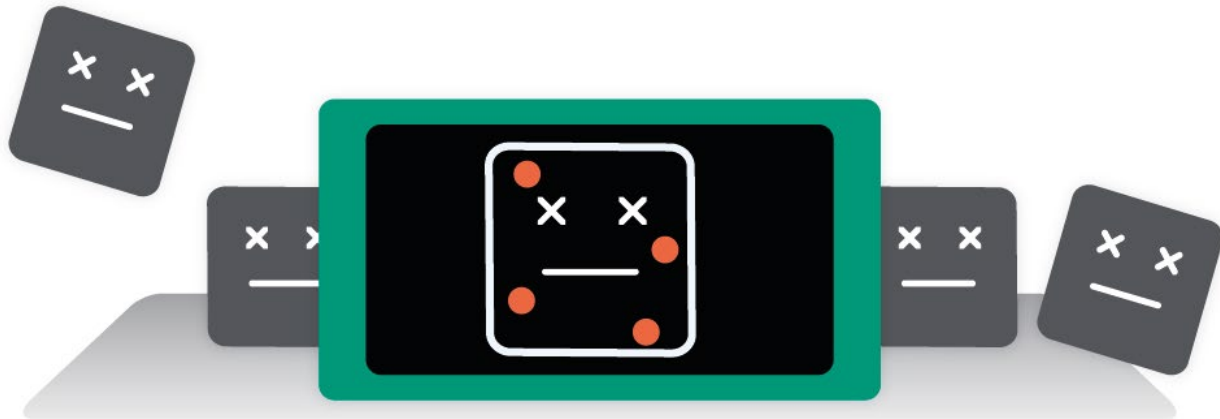
Lessons learned

Some lessons you should learn from this are:

1. Set requests and limits in your workloads. While it is highly recommended, be accurate and permanently monitor both requests and limits on your containers. Setting inappropriate values can cause serious problems in your applications and your Kubernetes clusters.
2. Setting a namespace quota will enforce all of the workloads in the namespace to have a request and limit in every container.
3. Quotas are a necessity to properly share resources. If someone tells you that you can use any shared service without limits, they're either lying or the system will eventually collapse, to no fault of your own.
4. Measuring unused resources, and rightsizing your workloads afterwards, will allow you to reduce wasted spending in your Kubernetes environments.

How to troubleshoot Kubernetes OOM and CPU Throttle

Experiencing Kubernetes OOM kills can be very frustrating. Why is your application struggling if you have plenty of CPU in the node? Managing Kubernetes pod resources can be a challenge. Many issues can arise, possibly due to an incorrect configuration of requests and limits, so it is important to be able to detect the most common issues related to the usage of resources.



Kubernetes OOM problems

When any Unix-based system runs out of memory, OOM safeguard kicks in and kills certain processes based on obscure rules only accessible to level 12 dark sysadmins ([chaotic neutral](#)). Kubernetes OOM management tries to avoid the system running behind by triggering its own rules. When the node is low on memory, Kubernetes eviction policy enters the game and stops pods as failed. If they are managed by a ReplicaSet, these pods are scheduled in a different node. This frees memory to relieve the memory pressure.

OOM kill due to container limit reached

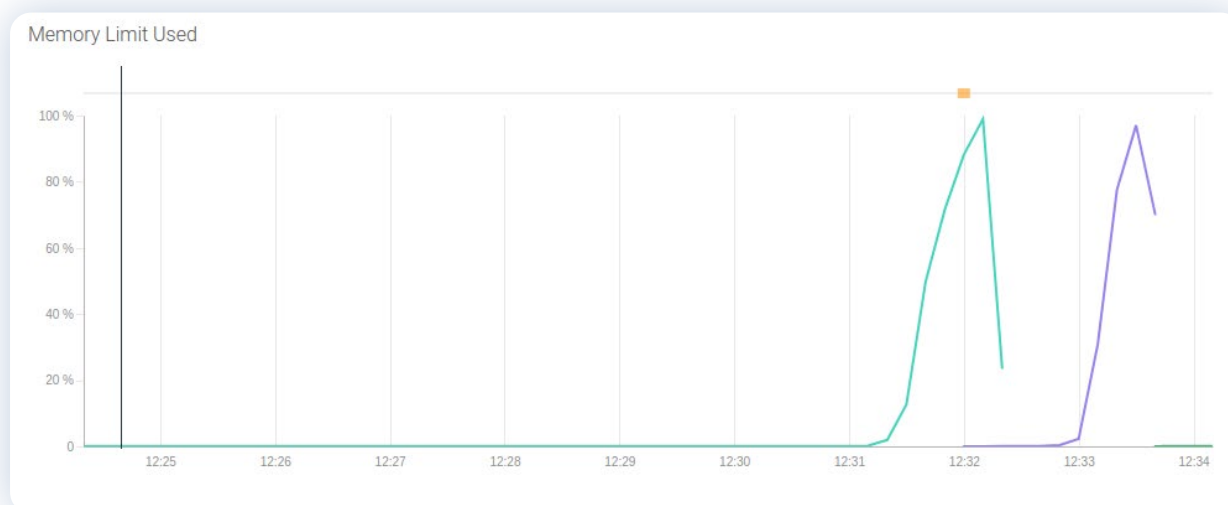
This is by far the most simple memory error you can have in a pod. You set a memory limit, one container tries to allocate more memory than allowed, and it gets an error. This usually ends with a container dying, one pod unhealthy, and Kubernetes restarting that pod.

```
test      frontend    0/1    Terminating    0    9m21s
```

`kubectl describe pods` output would show something like this:

```
State:      Running
  Started:   Thu, 10 Oct 2019 11:14:13 +0200
Last State:  Terminated
  Reason:    OOMKilled
  Exit Code: 137
  Started:   Thu, 10 Oct 2019 11:04:03 +0200
  Finished:  Thu, 10 Oct 2019 11:14:11 +0200
...
Events:
Type Reason      Age      From      Message
----
Normal Scheduled 6m39s    default-scheduler Successfully assigned test/frontend to gke-lab-kube-gke-default-pool-02126501-7nqc
Normal SandboxChanged 2m57s    kubelet, gke-lab-kube-gke-default-pool-02126501-7nqc Pod sandbox changed, it will be killed and re-created.
Normal Killing 2m56s    kubelet, gke-lab-kube-gke-default-pool-02126501-7nqc Killing container with id docker://db:Need to kill Pod
```

The **Exit Code: 137** is important because it means that the system terminated the container as it tried to use more memory than its limit. In order to monitor this, you always have to look at the use of memory compared to the limit. Percentage of the node memory used by a pod is usually a bad indicator as it gives no indication on how close to the limit the memory usage is. In Kubernetes, limits are applied to containers, not pods, so monitor the memory usage of a container vs. the limit of that container.



Find these metrics in Sysdig Monitor in the dashboard: Containers > Container resource usage

Kubernetes OOM kill due to limit overcommit

Memory requests are granted to the containers so they can always use that memory, right? Well, it's complicated. Kubernetes will not allocate pods that sum to more memory requested than memory available in a node. But limits can be higher than requests, so the sum of all limits can be higher than node capacity. This is called "overcommit" and it's very common. In practice, if all containers use more memory than requested, it can exhaust the memory in the node. This usually causes the death of some pods in order to free some memory.

Memory management in Kubernetes is complex, as it has many facets. Many parameters enter the equation at the same time:

- Memory request of the container.
- Memory limit of the container.
- Lack of those settings.
- Free memory in the system.
- Memory used by the different containers.

With these parameters, a blender, and some math, Kubernetes elaborates a score. The last in the table is killed or evicted. The pod can be restarted depending on the policy, so that doesn't mean the pod will be removed entirely.

Despite this mechanism, you can still finish with system OOM kills as Kubernetes memory management runs every several seconds only. If the system memory fills too quickly, the system can kill Kubernetes control processes, making the node unstable. This scenario should be avoided as it will most likely require a complicated troubleshooting process, ending with a root-cause analysis based on hypothesis and a node restart.

In day-to-day operation, this means that in case of overcommitting resources, pods without limits will often be killed, containers using more resources than requested have a chance to die, and guaranteed containers will most likely be fine.

CPU throttling due to CPU limit

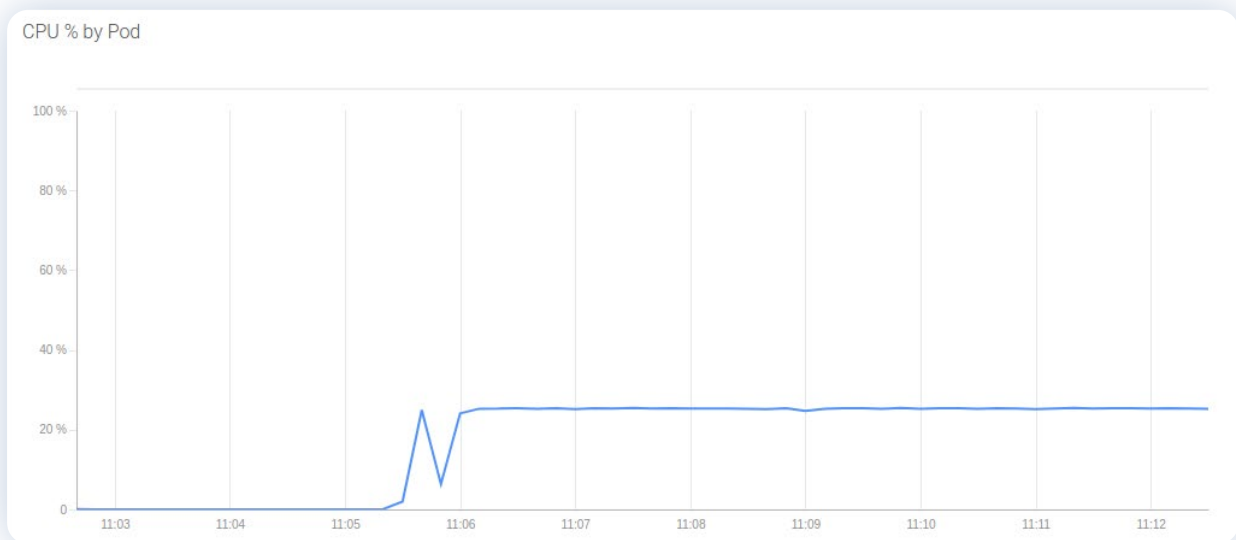
There are many differences on how CPU and memory requests and limits are treated in Kubernetes. A container using more memory than the limit will most likely die, but using CPU can never be the reason that Kubernetes kills a container. CPU management is delegated to the system scheduler, and it uses two different mechanisms for the requests and the limits enforcement.

CPU requests are managed using the shares system. This means that the resources in the CPU are prioritized depending on the value of shares. Each CPU core is divided into 1,024 shares and the resources with more shares have more CPU time reserved. Be careful, because in moments of CPU starvation, shares won't ensure your app has enough resources as it can be affected by bottlenecks and general collapse. If a container has a limit of 100m, the container will have 102 shares. These values are only used for pod allocation. Monitoring the shares in a pod doesn't give any idea of a problem related to CPU throttling.

On the other hand, limits are treated differently. Limits are managed with the CPU quota system. This works by dividing the CPU time in 100ms periods and assigning a limit on the containers with the same percentage that the limit represents to the total CPU in the node.

If you set a limit of 100m, the process can use 10ms of each period of processing. The system will throttle the process if it tries to use more time than the quota, causing possible performance issues. A pod will never be terminated or evicted for trying to use more CPU than its quota. Rather, the system will just limit the CPU.

If you want to know if your pod is suffering from CPU throttling, you have to look at the percentage of the quota assigned that is being used. Absolute CPU use can be treacherous, as you can see in the following graphs. CPU use of the pod is around 25%, but as that is the quota assigned, it is using 100% and consequently suffering CPU throttling.



Find these metrics in Sysdig Monitor in the dashboard: Kubernetes > Node status & performance



Find these metrics in Sysdig Monitor in the dashboard: Containers > Container CPU & Memory limits

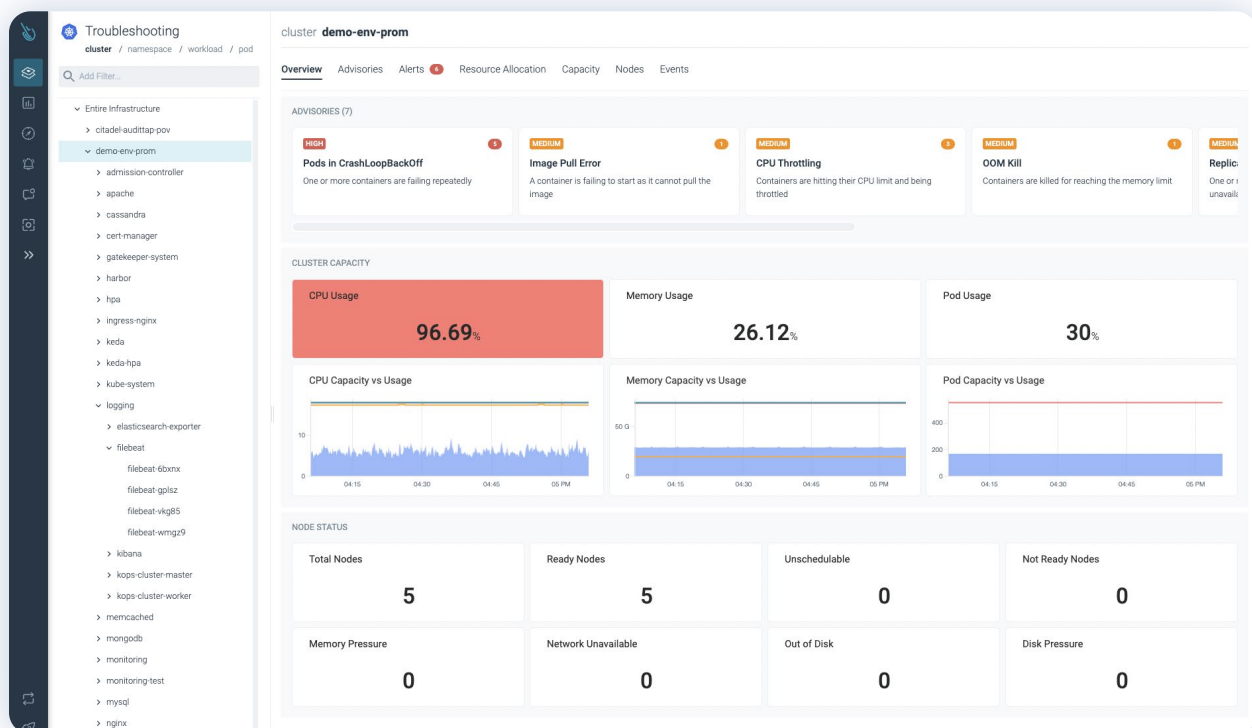
There is a great difference between CPU and memory quota management. Regarding memory, a pod without requests and limits is considered burstable and is the first of the list to OOM kill. With the CPU, this is not the case. A pod without CPU limits is free to use all of the CPU resources in the node. The CPU is there to be used, but if you can't control which process is using your resources, you can end up with a lot of problems due to CPU starvation of key processes.

Troubleshooting OOM Kill and CPU Throttling with Sysdig Monitor

Advisor is a tool integrated in Sysdig Monitor that allows you to easily spot and troubleshoot critical common issues, like OOM Kill and CPU Throttling. Unlike what you usually had to do with traditional troubleshooting, you don't have to go to the CLI, and maybe to the internet, to dig deep on what a certain error means. Thanks to the Advisor, everything that you may need to troubleshoot these kinds of issues are already there, in a centralized console bringing all the data from your Kubernetes clusters.

Using Sysdig's Advisor can accelerate troubleshooting by up to 10 times.

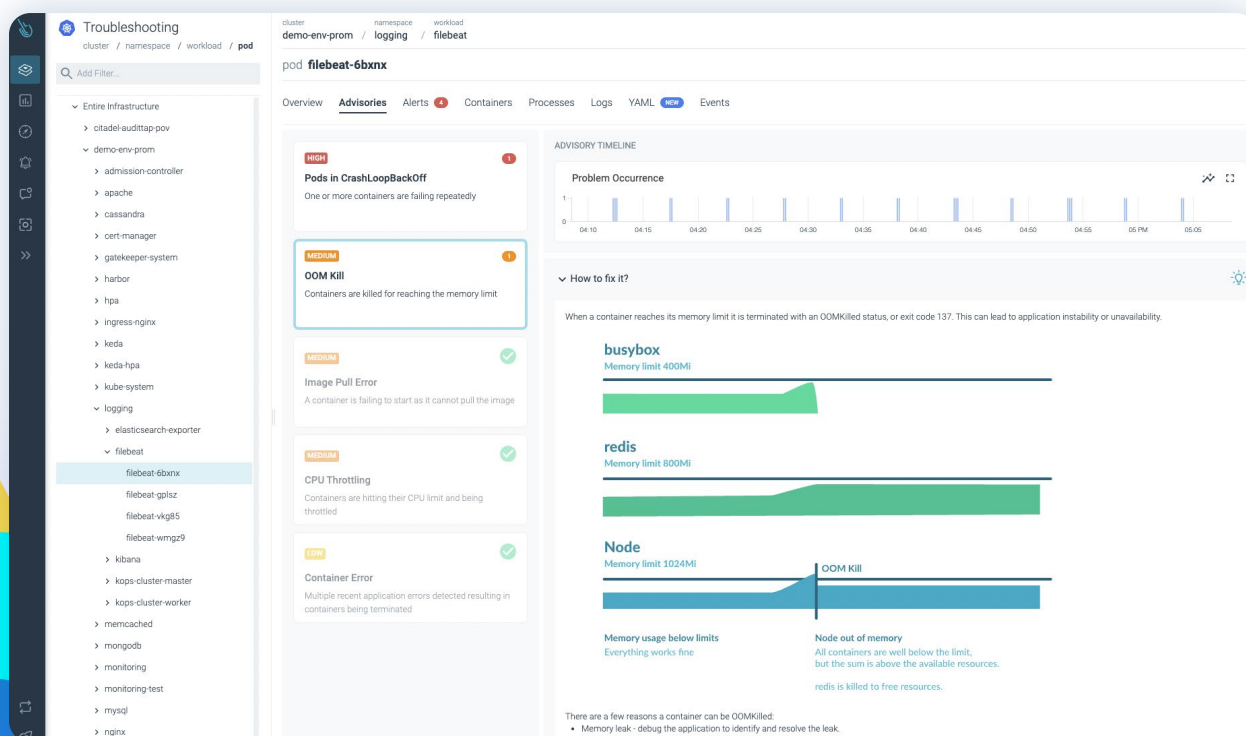
The overview tab allows Sysdig users to check the current status of clusters and workloads at a glance. Advisories are enabled and provided out-of-the-box, alerting Sysdig users of new issues in a proactive way.

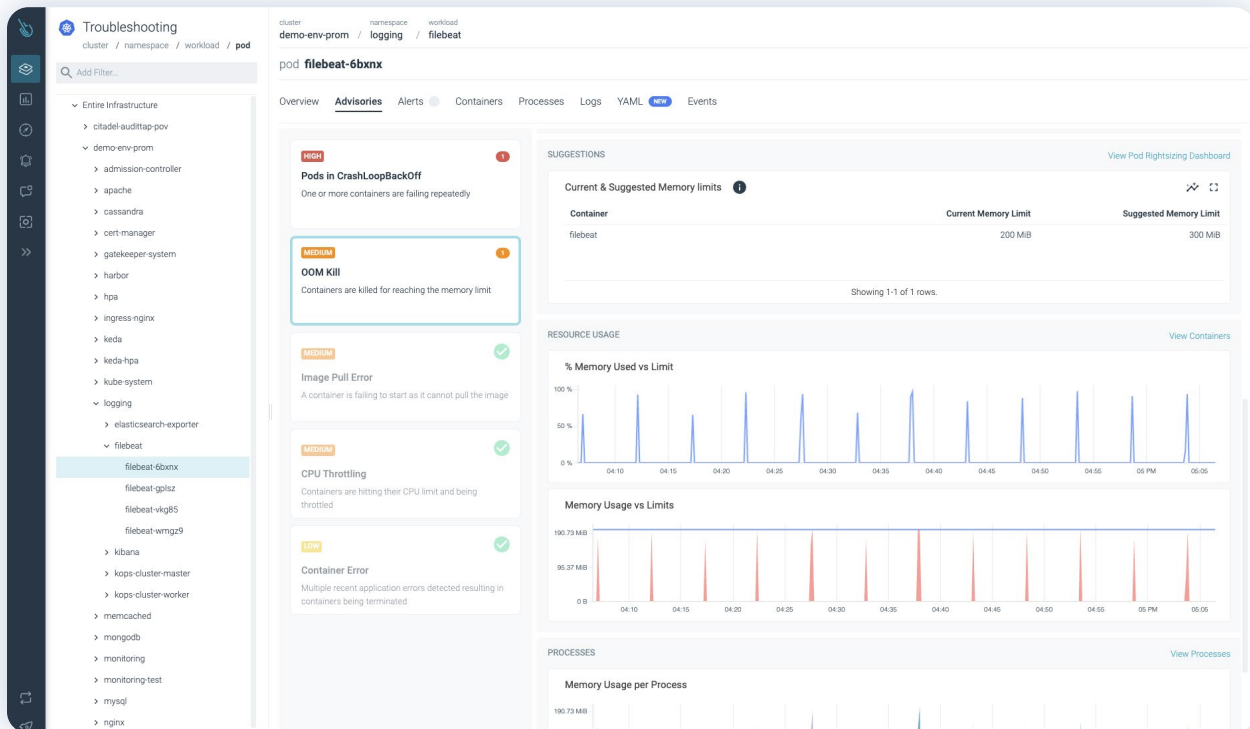


Every advisory has its own priority classification. This makes issues identification and valuation much easier. Red issues are the ones with Higher priority, orange is the color for medium priorities, while green was designated to the issues with lowest priority.

Advisor provides tons of data from your applications. This will help you to determine and find the root cause of the issue. In order to fix the issue as soon as possible, Advisor provides some valuable information, like:

- Brief description about the error and how to fix it.
- Problem occurrence, for valuing how this issue is impacting your business.
- Resources usage data.
- Suggestions in the form of possible remediations towards a quick resolution.
- Live logs access from containers.
- YAML inspect tool, to check the actual YAML definition of the Pod.
- Etc.





Lessons learned

1. Knowing how to monitor resource usage in your workloads is of vital importance. This will allow you to discover different issues that can affect the health of the applications running in the cluster.
2. Understanding that your resource usage can compromise your application and affect other applications in the cluster is the crucial first step. You have to properly configure your quotas.
3. Monitoring the resources and how they are related to the limits and requests will help you set reasonable values and avoid Kubernetes OOM kills. This will result in a better performance of all the applications in the cluster, as well as a fair sharing of resources.
4. Your Kubernetes alerting strategy can't just focus on the infrastructure layer. It needs to understand the entire stack, from the hosts and Kubernetes nodes at the bottom, up to the top where the application workloads and its metrics live.
5. Being able to leverage Kubernetes and cloud providers metadata to aggregate and segment metrics and alerts will be a requirement for effective alerting across all layers.

Conclusion

In this guide, we have presented:

- The basics of Kubernetes monitoring
- How to use Golden Signals
- How to monitor Kubernetes infrastructure
- How to monitor Kubernetes workloads
- Useful alerts to use to become more proactive

We hope that you found this information useful as you navigate the best way to monitor your Kubernetes workloads. Still, this can be a complex journey and Kubernetes monitoring can take years to master. As you have seen with the examples in this guide, Sysdig tries to take some of this complexity out of your way with our [Sysdig Monitor](#) product. With Sysdig Monitor, you can:

- Easily instrument your Kubernetes environment.
- Get granular details from system calls, along with Kubernetes context.
- Correlate system, network, and custom metrics to the health of your Kubernetes workloads for faster troubleshooting.
- Quickly increase Kubernetes visibility with out-of-the-box dashboards and alerts.
- Scale Prometheus to millions of metrics with long-term retention and PromQL compatibility.

A great place to learn about Kubernetes monitoring is the [Sysdig blog](#), where our experts are always posting new tips and recommendations. We encourage you to take advantage of our [free trial](#) to see if Sysdig Monitor can help you meet your performance and availability goals.

The Sysdig Secure DevOps Platform converges security and compliance with performance and capacity monitoring to create a secure DevOps workflow. It uses the same data to monitor and secure, so you can correlate system activity with Kubernetes services. This enables you to identify where a problem occurred and why it happened — and you can use this single source of truth to investigate and troubleshoot performance and security issues. If you are also interested in learning how to secure your Kubernetes environments, we have written a companion guide, called the [Kubernetes Security Guide](#), with detailed information about image scanning, control plane security, RBAC for Kubernetes, and runtime security.

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

Find out how Sysdig Monitor can help you and your teams monitor and troubleshoot cloud-native applications in production. Contact us for additional details about the platform, or to arrange a personalized demo. If you can't wait request a 30 days trial account and try it for free.

sysdig.com/start-free

