Kubernetes Monitoring Guide

sysdig

Contents

About This Guide

With over 30,000 stars on GitHub, over four hundred contributors, and an ecosystem that includes Google, RedHat, 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 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 can you 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 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 in 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 22% of the containers live for 10 seconds or less while 54% of them live less 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 new ones.

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 cannot SSH or look at logs, 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 blackboxes, 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 needs to 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 that we will cover in 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 many 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? Am I 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? Am I 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 that is similar to what you may be used to:

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

Golden Signals are considered the best practice approach on this. We will cover this in section: Monitoring Kubernetes with Golden Signals.

Monitoring Kubernetes deployments and pods

This is a grey 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. 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 Prometheus monitoring at scale

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.

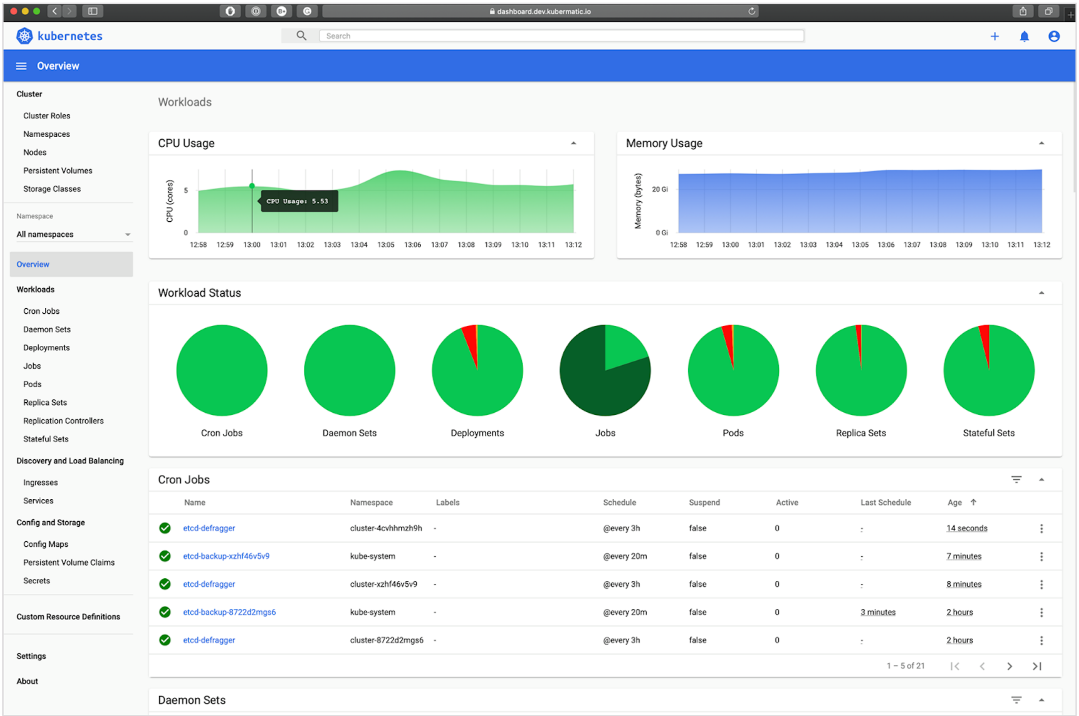
On Kubernetes, the nodes' kubelets, the on-machine Kubernetes agent, install cAdvisor to monitor the resources of the containers inside of each pod. But to go further with this data, you need something that aggregates data across the entire cluster. The most popular option used to be Heapster, but that has been deprecated and Kubernetes consumes metrics through the metric-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.

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.



Cron Jobs

	Name	Namespace	Labels	Schedule	Suspend	Active	Last Schedule	Age
✓	etcd-defragger	cluster-4cvhhmzh9h	-	@every 3h	false	0	-	14 seconds
✓	etcd-backup-xzhf46v5v9	kube-system	-	@every 20m	false	0	-	7 minutes
✓	etcd-defragger	cluster-xzhf46v5v9	-	@every 3h	false	0	-	8 minutes
✓	etcd-backup-8722d2mgs6	kube-system	-	@every 20m	false	0	3 minutes	2 hours
✓	etcd-defragger	cluster-8722d2mgs6	-	@every 3h	false	0	-	2 hours

1 – 5 of 21

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. It requires Kubernetes 1.2+, and unfortunately warns that metric names and tags are unstable and may change.

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. Kubernetes monitoring probes allows 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:

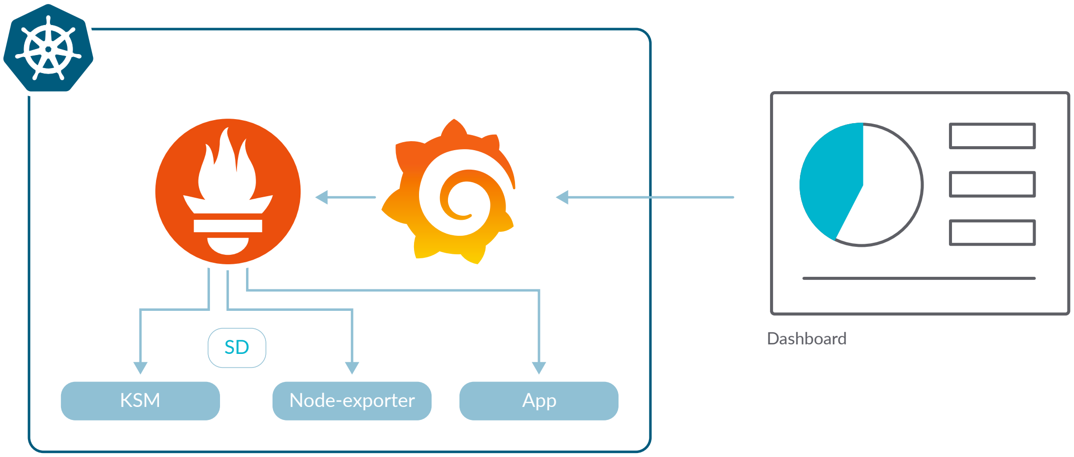
```
#Example Liveness probe for the Sysdig Blog on "Monitoring
Kubernetes"
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    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/
```

A readiness probe is similar to a liveness probe, which executes requests or a command to check if a pod is prepared to handle traffic after (re)starting.

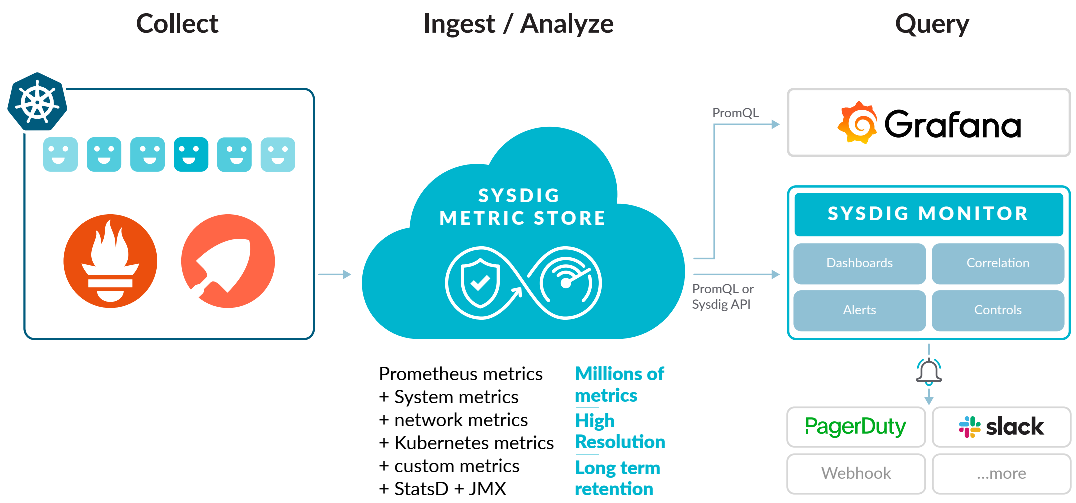
Prometheus for Kubernetes monitoring

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 Prometheus monitoring at scale



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 troubleshooting, correlation and support that we provide as part of Sysdig Monitor for a complete Prometheus monitoring solution.

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 you remain compatible with Prometheus monitoring. And if you are running large scale environments, we have you covered thanks to our Prometheus scale capabilities.

Monitoring Kubernetes With Golden Signals

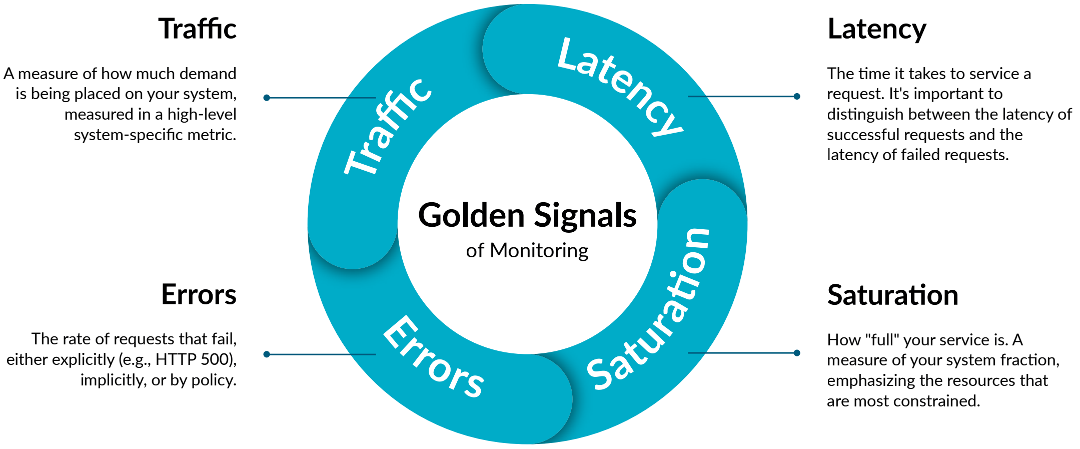
What are Golden Signal metrics? How do you monitor Golden Signals in Kubernetes applications? Golden Signals can help detect issues of a microservices application. These signals are a reduced set of metrics that offer a wide view of a service from a user or consumer perspective, so you can detect potential 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 and that you're unable to detect potential problems. 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 less 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 and, fortunately, 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.



Traffic

A measure of how much demand is being placed on your system, measured in a high-level system-specific metric.

Latency

The time it takes to service a request. It's important to distinguish between the latency of successful requests and the latency of failed requests.

Errors

The rate of requests that fail, either explicitly (e.g., HTTP 500), implicitly, or by policy.

Saturation

How "full" your service is. A measure of your system fraction, emphasizing the resources that are most constrained.

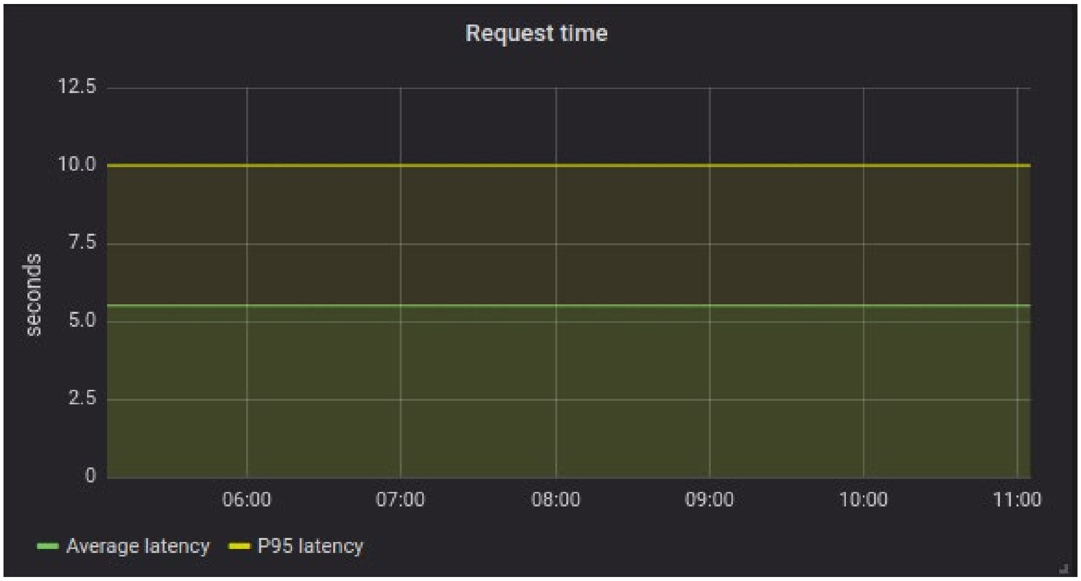
Golden Signals metric: Latency explained

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 of them is 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.

A great tool to avoid this behaviour are histogram metrics. 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, average latency is acceptable. But if you look at percentile, you see a lot of dispersion in the values, giving a better idea of what is the real latency perception. 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 detect not only 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 errors, mean values or percentiles don't make sense. In order to properly see the impact of errors, the best way is to 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.

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.

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 accent 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.

Golden Signals are some of the metrics available out-of-the-box, 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.

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 and then to pod.

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 0 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”
)

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”, prometheus.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)
        }
    }
}
```

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
```

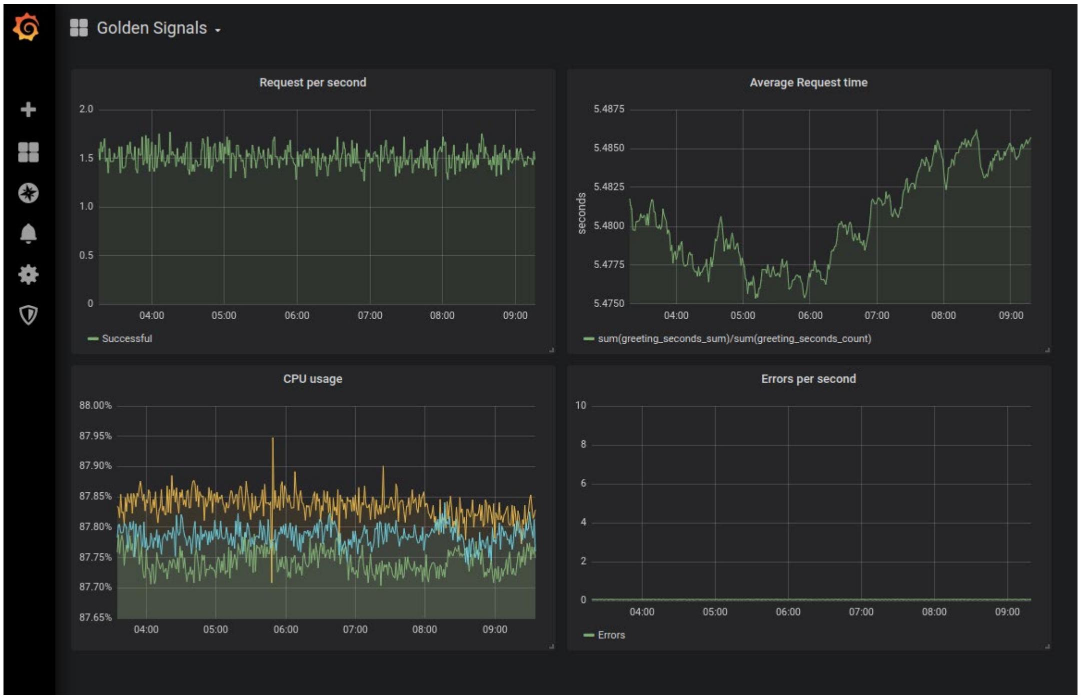
- Errors per second:

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

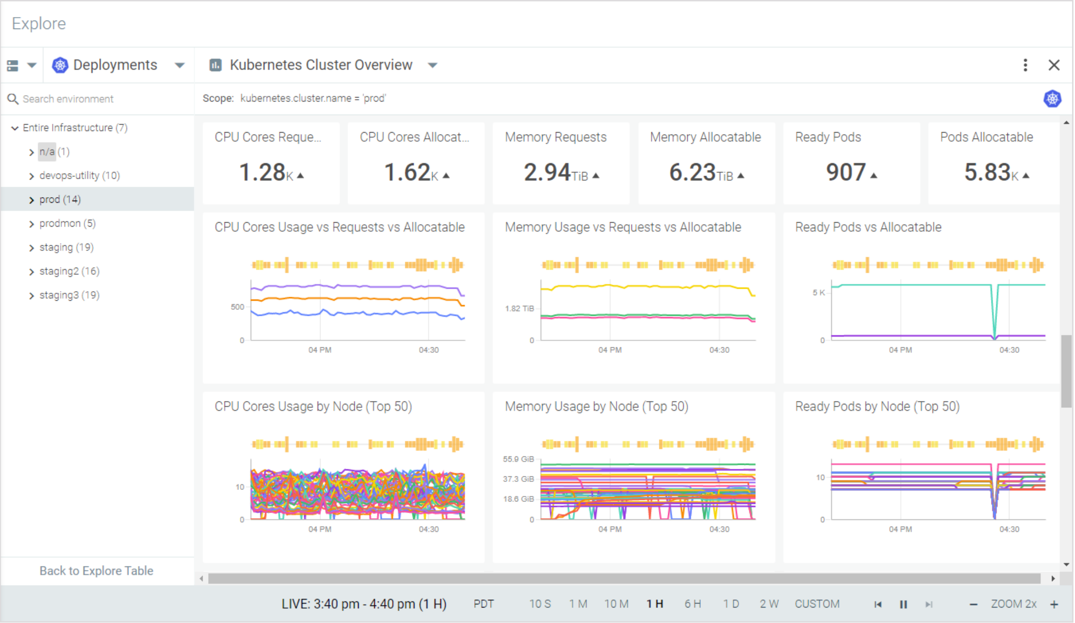
- Saturation: You can use cpu percentage obtained with node-exporter:

```
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 you would 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.

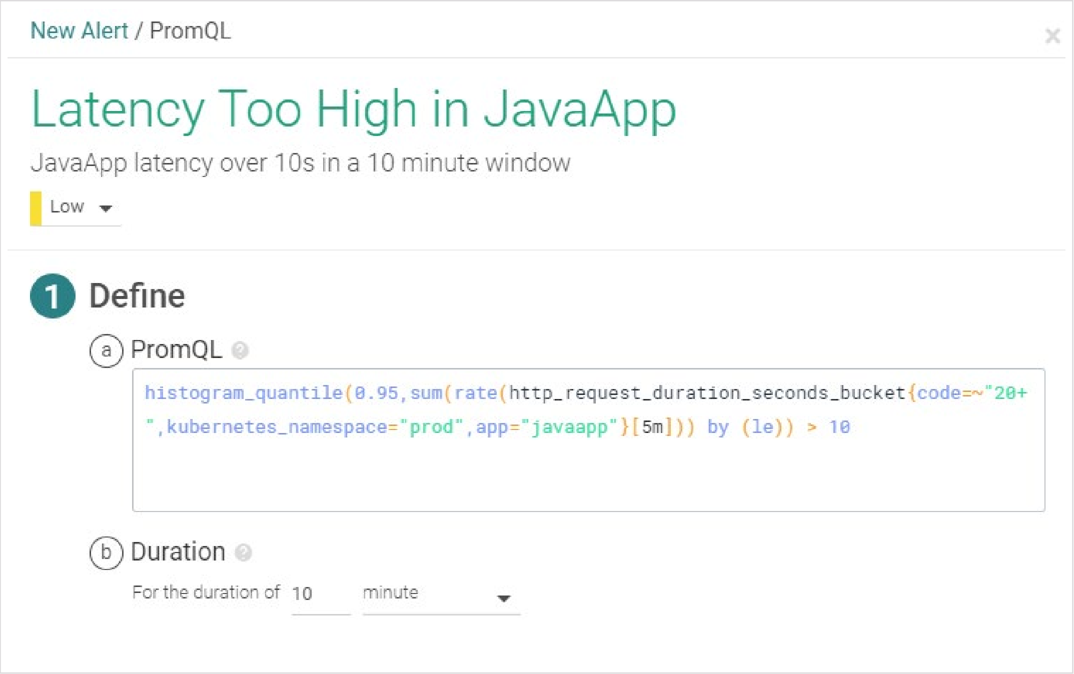
Alerting on application layer metrics

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.



New Alert / PromQL

Latency Too High in JavaApp

JavaApp latency over 10s in a 10 minute window

Low

1 Define

a PromQL

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

b Duration

For the duration of 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 the problem. 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 behaviour 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.

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

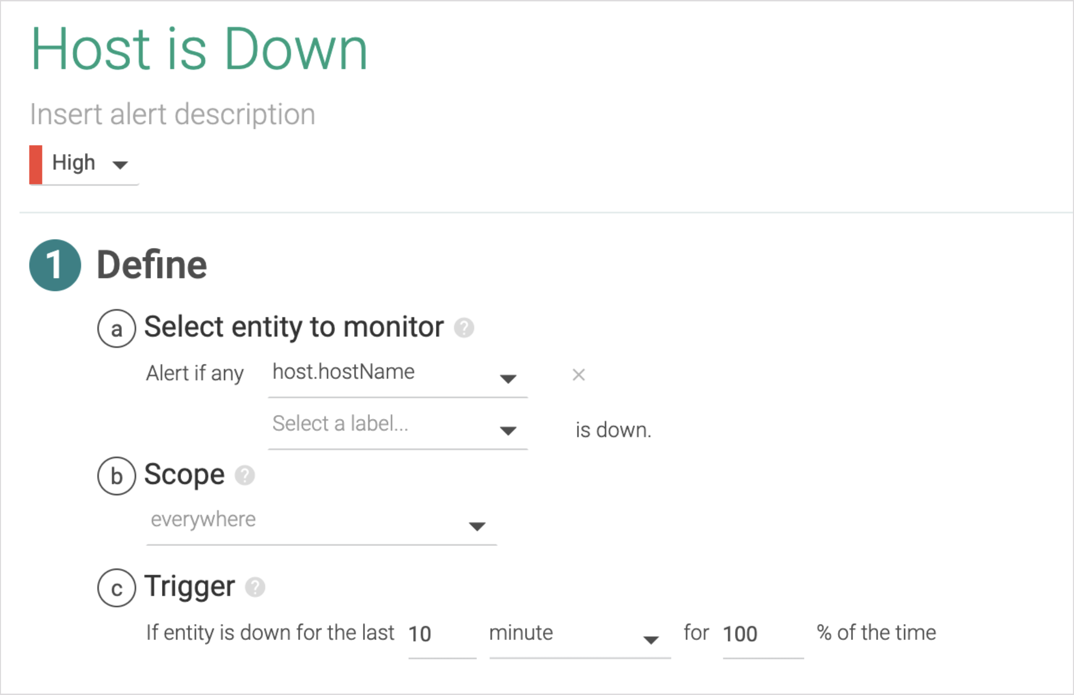
Alerting at the host layer shouldn't be very different from monitoring cloud instances, VMs or bare metal servers. It's going to be mostly about if the host is up or down/unreachable, and resource availability (CPU, memory, disk, etc.).

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 our entire infrastructure. Give it a five minutes 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 host.hostName

Select a label... is down.

b Scope

everywhere

c Trigger

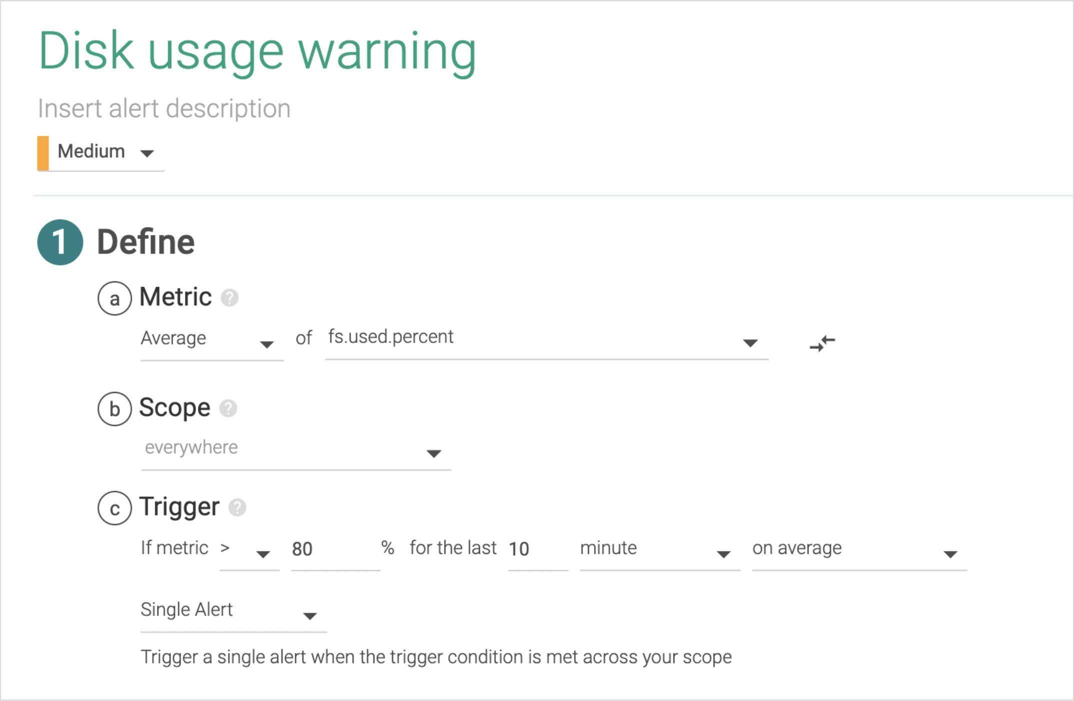
If entity is down for the last 10 minute for 100 % of the time

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

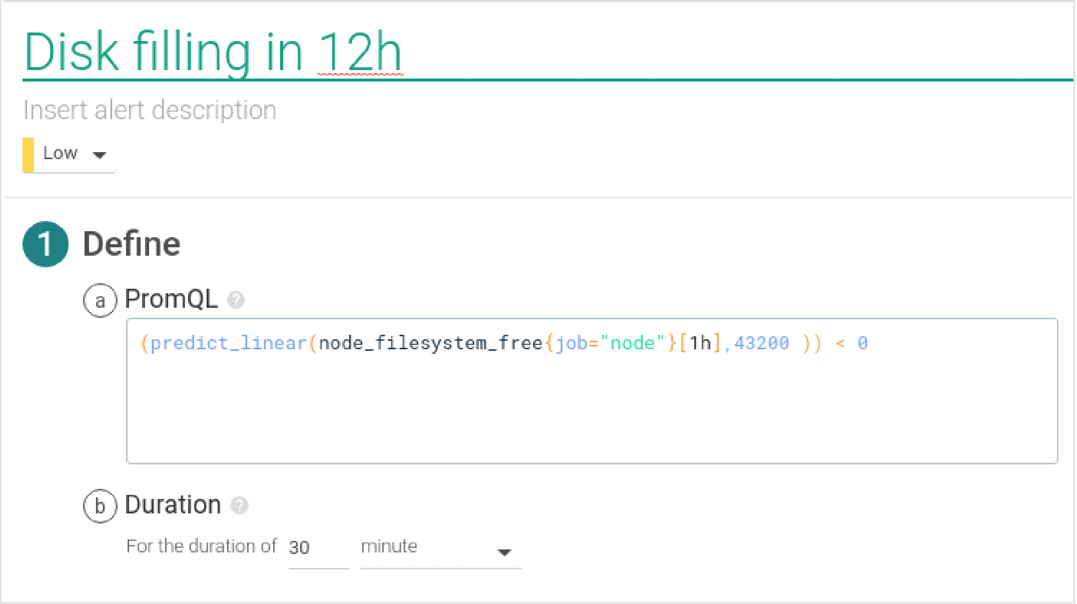
c Trigger

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.



Disk filling in 12h

Insert alert description

Low

1 Define

a PromQL

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

b Duration

For the duration of 30 minute

PromQL query:

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

This alert will trigger in case the disk was 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 I 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 have to add some sections to the agent YAML configuration file and use Prometheus to gather the metrics and filter them. You can choose not to do it but you will save a lot of debugging metrics if you follow this approach. Sysdig can leverage existing Prometheus deployments to collect and filter control plane metrics. So, first of all, you should have a Prometheus server up and running. Prometheus is installed by default in Openshift 4 or when using Kubernetes metrics manager. If you haven't one you don't worry, deploying a new one is as simple as executing 2 commands, the first to create a namespace for the Prometheus and the other just to deploy it with helm 3.

```
kubectl create ns monitoring
helm install -f values.yaml prometheus -n monitoring stable/
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 have to create the rules. These rules create new metrics with a custom tag which will filter the metrics collected by the Sysdig agent. You can find the rules and all steps you have to follow to have the monitoring kubernetes control plane in Sysdig on PromCat.io.

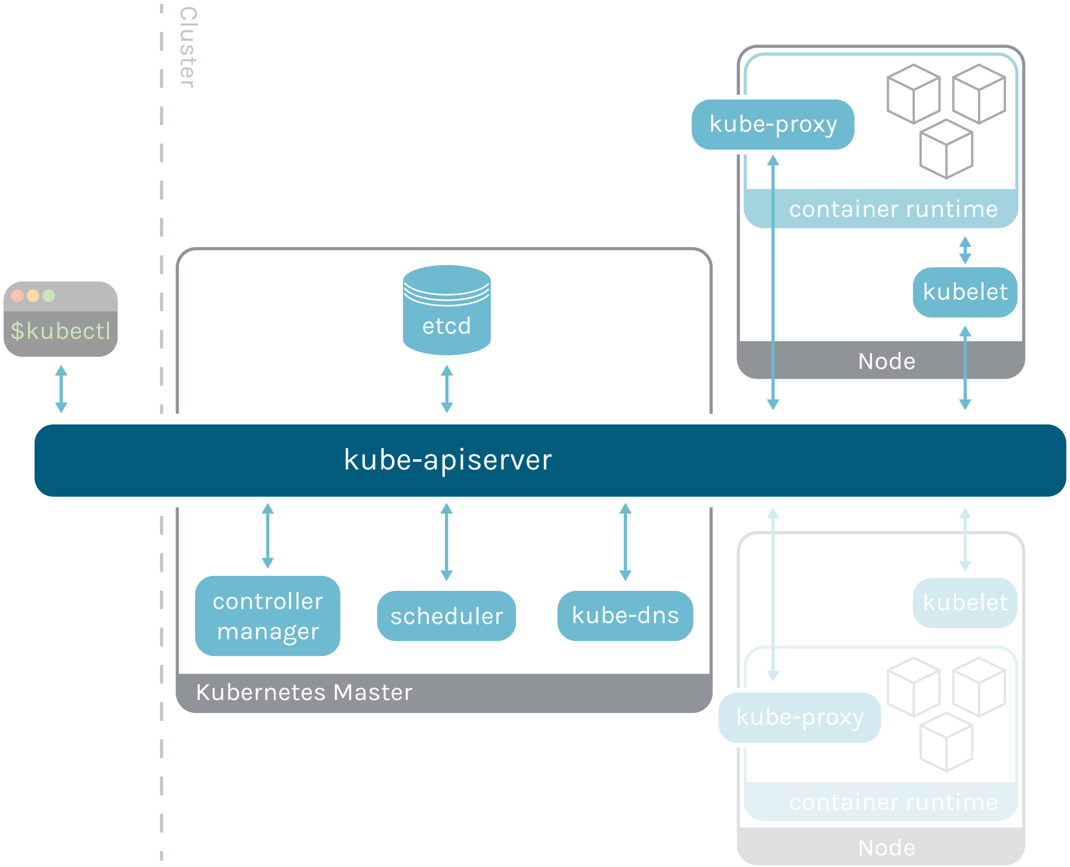
Finally you just need to add some information to the agent configmap:

```
data:
  prometheus.yaml: |-
    global:
      scrape_interval: 15s
      evaluation_interval: 15s
    scrape_configs:
    - job_name: ‘prometheus’ # config for federation
      honor_labels: true
      metrics_path: ‘/federate’
      metric_relabel_configs:
      - regex: ‘kubernetes_pod_name’
        action: labeldrop
      params:
        ‘match[]’:
          - ‘{sysdig=”true”}’
      sysdig_sd_configs:
      - tags:
          namespace: monitoring
          statefulset: prometheus-server
  dragent.yaml: |-
    metrics_excess_log: true
    use_promscrape: true
    prometheus:
      enabled: true
```

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, the rest of the control plane components 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 of 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 a service account bound to a `ClusterRole`, granting GET requests to ***/metrics*** endpoint.

This can be done by adding one rule to the `ClusterRole` used by Prometheus:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    app: monitor
    component: server
  name: monitor
rules:
- nonResourceURLs:
  - /metrics
  verbs:
  - get
```

This way, 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:

```
#curl  https://kubernetes.default.svc/metrics -H “Authorization:
Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)”
-vvv --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
```

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

```
# TYPE APIServiceOpenAPIAggregationControllerQueue1_adds counter
APIServiceOpenAPIAggregationControllerQueue1_adds 108089
# HELP APIServiceOpenAPIAggregationControllerQueue1_depth Current
depth of workqueue: APIServiceOpenAPIAggregationControllerQueue1
# TYPE APIServiceOpenAPIAggregationControllerQueue1_depth gauge
APIServiceOpenAPIAggregationControllerQueue1_depth 0
# HELP APIServiceOpenAPIAggregationControllerQueue1_
queue_latency How long an item stays in
workqueueAPIServiceOpenAPIAggregationControllerQueue1 before being
requested.
# TYPE APIServiceOpenAPIAggregationControllerQueue1_queue_latency
summary
APIServiceOpenAPIAggregationControllerQueue1_queue_
latency{quantile=”0.5”} 15
...
```

Configuring Prometheus to scrape the Kubernetes API server endpoint can be done by adding one job to your targets:

```
- 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 1.15. You can check the metrics available for your version in the* *Kubernetes repo* (link *for the 1.15.3 version).*

Latency: Latency can be extracted from the apiserver_request_duration_seconds histogram buckets:

```
# TYPE apiserver_request_latencies histogram
apiserver_request_duration_seconds{resource=”adapters”,scope=
”cluster”,subresource=””,verb=”LIST”,le=”125000”} 2
apiserver_request_duration_seconds{resource=”adapters”,scope=
”cluster”,subresource=””,verb=”LIST”,le=”250000”} 2
apiserver_request_duration_seconds{resource=”adapters”,scope=
”cluster”,subresource=””,verb=”LIST”,le=”500000”} 2
apiserver_request_duration_seconds{resource=”adapters”,scope=
”cluster”,subresource=””,verb=”LIST”,le=”1e+06”} 2
apiserver_request_duration_seconds{resource=”adapters”,scope=
”cluster”,subresource=””,verb=”LIST”,le=”2e+06”} 2
apiserver_request_duration_seconds{resource=”adapters”,scope=
”cluster”,subresource=””,verb=”LIST”,le=”4e+06”} 2
apiserver_request_duration_seconds{resource=”adapters”,scope=
”cluster”,subresource=””,verb=”LIST”,le=”8e+06”} 2
apiserver_request_duration_seconds{resource=”adapters”,scope=
”cluster”,subresource=””,verb=”LIST”,le=”+Inf”} 2
apiserver_request_duration_seconds_sum{resource=”adapters”,scope=
”cluster”,subresource=””,verb=”LIST”} 50270
apiserver_request_duration_seconds_count{resource=”adapters”,scope=
”cluster”,subresource=””,verb=”LIST”} 2
```

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

```
histogram_quantile(0.99, sum(rate(apiserver_request_latencies_
count{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]))
```

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.

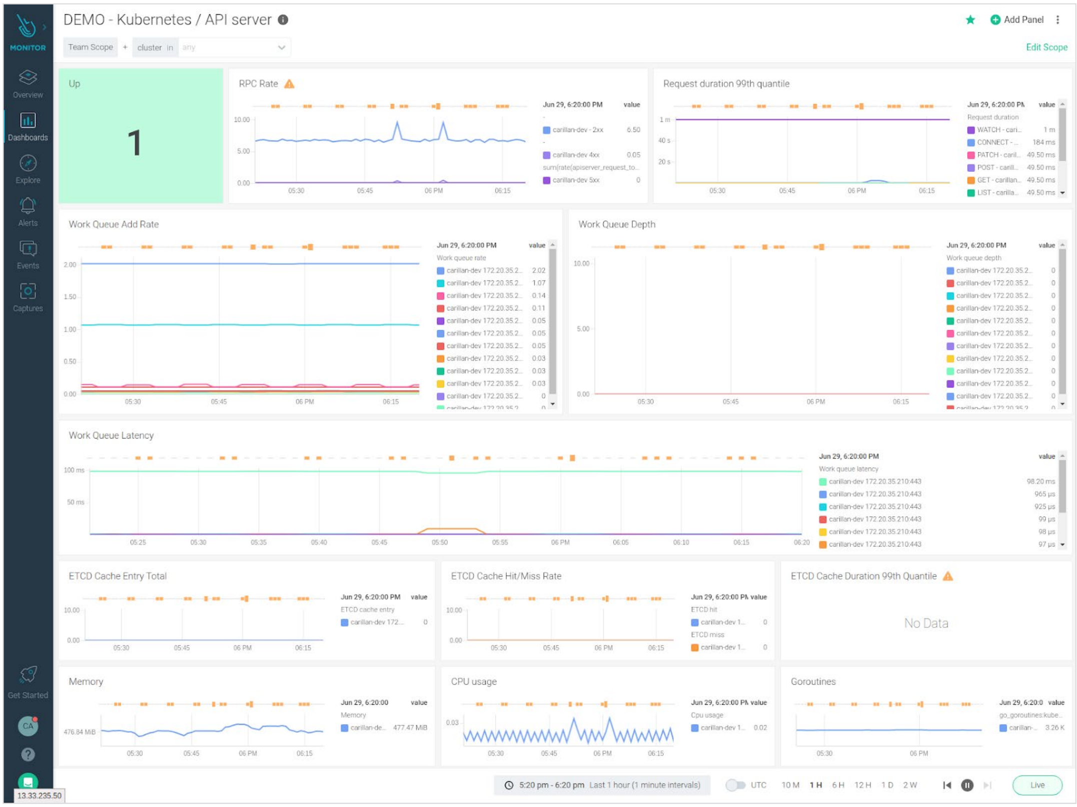
This is typically a sign of overload in the API server. Most likely, your cluster has a lot of load and the API server needs to be scaled out. You can segment the metrics by type of request, resource or verb. This way, you can detect where the problem is. Maybe you are having issues reading or writing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

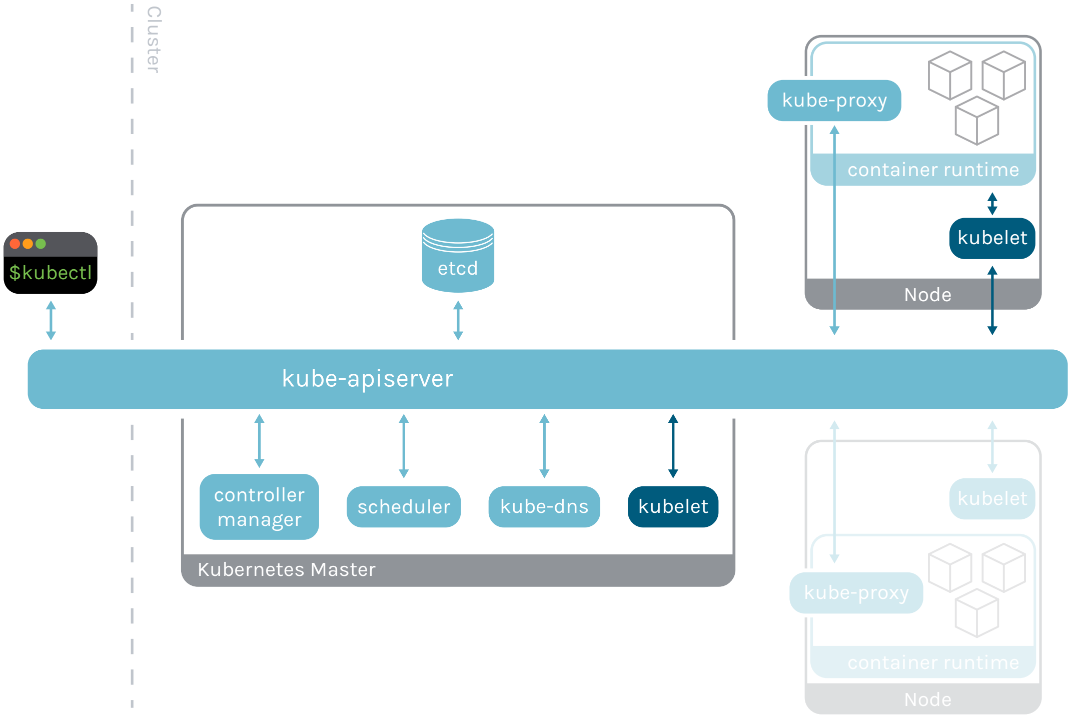
If you want to monitor Kubernetes API server using Sysdig Monitor, you just need to add a couple of sections to the Sysdig agent yaml configuration file as we explained above in "Monitoring kubernetes control plane in Sysdig Monitor": Then you can build custom dashboards using these metrics. Here is what an API server dashboard would look like 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 10255 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 without authentication.

```
curl  http://[Node_Internal_IP]:10255/metrics
```

If the container has access to the host network, you can access using localhost too.

Note that the port and address may vary depending on your particular configuration.

It will return 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 our targets:

```
- job_name: ‘kubernetes-nodes’
    kubernetes_sd_configs:
      - role: node
    relabel_configs:
      - action: labelmap
        regex: __meta_kubernetes_node_label_(.+)
      # Only for Kubernetes ^1.7.3.
      # See: https://github.com/prometheus/prometheus/issues/2916
      - target_label: __address__
        replacement: kubernetes.default.svc:443
      - source_labels: [__meta_kubernetes_node_name]
        regex: (.+)
        target_label: __metrics_path__
        replacement: /api/v1/nodes/${1}/proxy/metrics/cadvisor
      metric_relabel_configs:
        - action: replace
          source_labels: [id]
          regex: ‘^/machine\.slice/machine-rkt\\x2d([^\\]+)\\.+/
([^/]+)\.service$’
          target_label: rkt_container_name
          replacement: ‘${2}-${1}’
        - action: replace
          source_labels: [id]
          regex: ‘^/system\.slice/(.+)\.service$’
          target_label: systemd_service_name
          replacement: ‘${1}’
```

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 1.15. You can check the metrics available for your version in the Kubernetes repo (link for the 1.15.3 version).*

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(up{job=\”kubernetes-nodes\”})
```

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_pod_count 9
# HELP kubelet_running_container_count Number of containers currently
running
# TYPE kubelet_running_container_count gauge
kubelet_running_container_count 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 than 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.

Config errors: This metric acts as a flag for configuration errors in the node.

```
# HELP kubelet_node_config_error This metric is true (1) if the node
is experiencing a configuration-related error, false (0) otherwise.
# TYPE kubelet_node_config_error gauge
kubelet_node_config_error 0
```

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 than 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
...
```

- **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
...
```

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:**

```
# 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**:

```
# 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
...
```

- **Kubelet_pod_worker_duration_seconds_bucket:**

```
# 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
...
```

Storage golden signals (operation rate, error rate and duration).

- **storage_operation_duration_seconds_count:**

```
# 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
…
```

- **storage_operation_errors_total:**

```
# HELP storage_operation_errors_total Storage errors total
# TYPE storage_operation_errors_total counter
storage_operation_errors_total { volume_plugin = “aws-ebs”,
operation_name = “volume_attach” } 0
storage_operation_errors_total { volume_plugin = “aws-ebs”,
operation_name = “volume_detach” } 0
```

- **storage_operation_duration_seconds_bucket:**

```
# 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
...
```

Cgroup manager operation rate and duration.

- **kubelet_cgroup_manager_duration_seconds_count**:

```
# 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**:

```
# 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
...
```

Pod Lifecycle Event Generator (PLEG): relist rate, relist interval and relist duration. Errors or excessive latency in these values can provoke issues in Kubernetes status of the pods.

- **kubelet_pleg_relist_duration_seconds_count**:

```
# 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**:

```
# 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
...
```

```
# 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
...
```

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 check if there is latency creating the containers or if they are in fact starting.

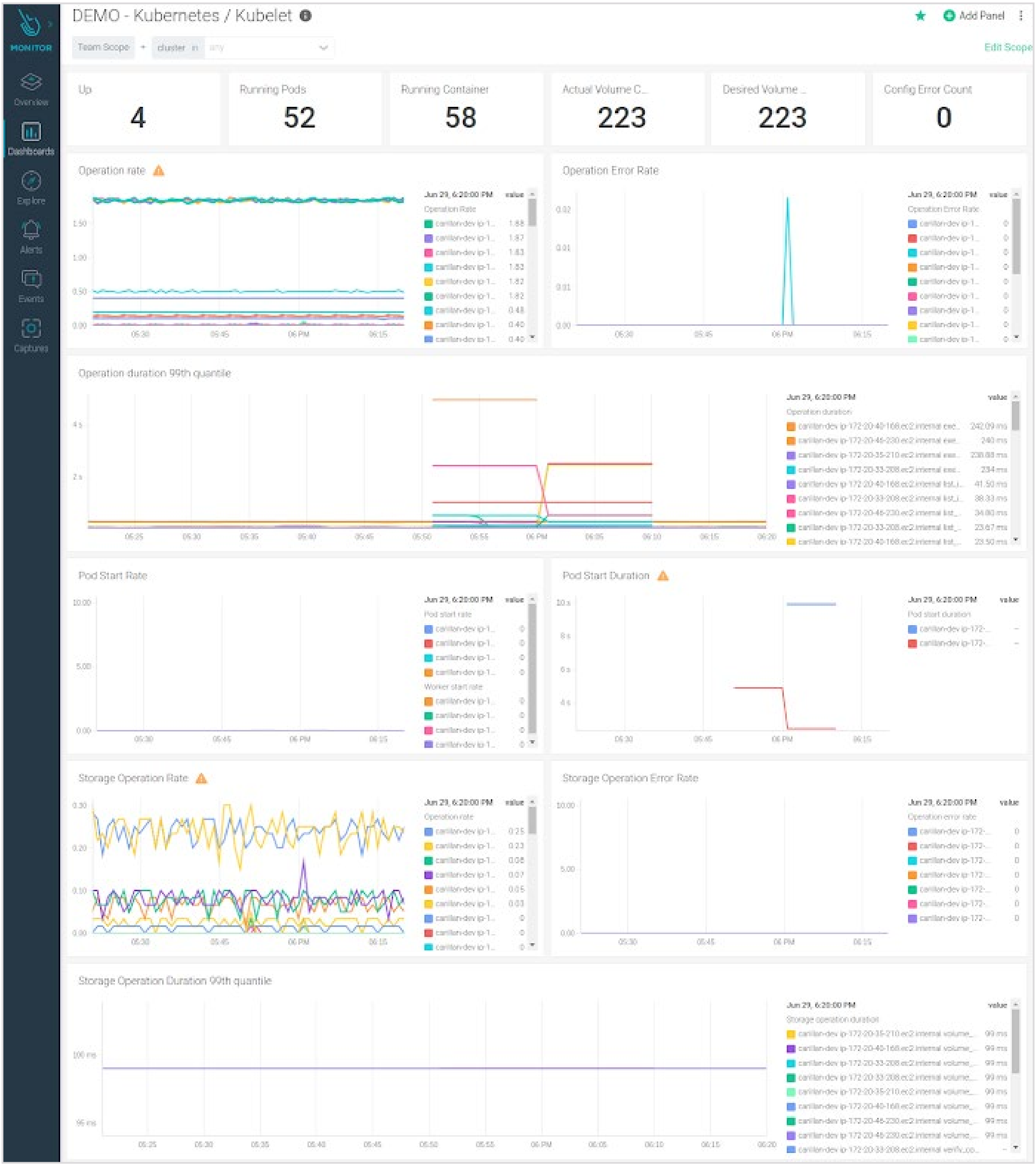
A node doesn’t seem to be scheduling new pods.
Check the Kubelet job number. There’s a chance that Kubelet has died in a node and is unable to schedule pods.

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

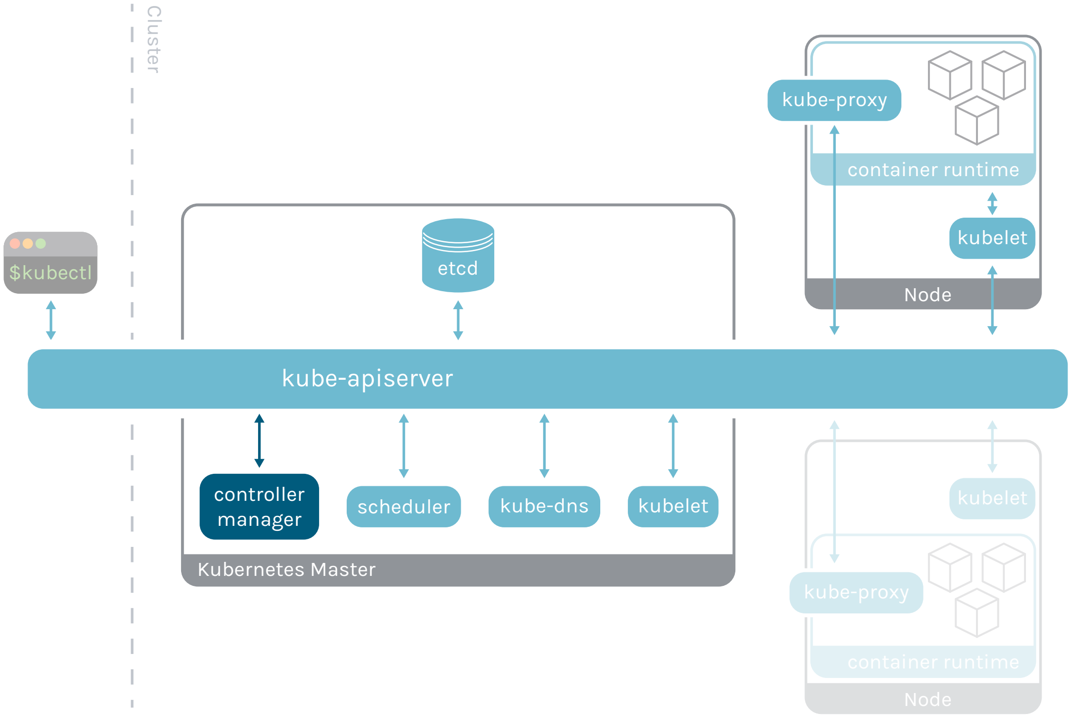
In order to track Kubelet in Sysdig Monitor, you just need to add a couple of sections to the Sysdig agent yaml configuration file as we explained above in “Monitoring kubernetes control plane in Sysdig Monitor”: Then you can build custom dashboards using these metrics. Here is what a Kublet dashboard would look like 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.

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

```
curl  http://localhost:10252/metrics
```

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

```
# HELP ClusterRoleAggregator_adds (Deprecated) Total number of adds
handled by workqueue: ClusterRoleAggregator
# TYPE ClusterRoleAggregator_adds counter
ClusterRoleAggregator_adds 602
# HELP ClusterRoleAggregator_depth (Deprecated) Current depth of
workqueue: ClusterRoleAggregator
# TYPE ClusterRoleAggregator_depth gauge
ClusterRoleAggregator_depth 0
# HELP ClusterRoleAggregator_longest_running_processor_microseconds
(Deprecated) How many microseconds has the longest running processor
for ClusterRoleAggregator been running.
# TYPE ClusterRoleAggregator_longest_running_processor_microseconds
gauge
```

```
ClusterRoleAggregator_longest_running_processor_microseconds 0
# HELP ClusterRoleAggregator_queue_latency (Deprecated) How long an
item stays in workqueueClusterRoleAggregator before being requested.
# TYPE ClusterRoleAggregator_queue_latency summary
ClusterRoleAggregator_queue_latency{quantile=”0.5”} 0
ClusterRoleAggregator_queue_latency{quantile=”0.9”} 0
...
```

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

```
- job_name: ‘kubernetes-pods’
      kubernetes_sd_configs:
        - role: pod
      relabel_configs:
    - action: labelmap
      regex: __meta_kubernetes_pod_label_(.+)
    - source_labels: [__meta_kubernetes_namespace]
      action: replace
      target_label: kubernetes_namespace
    - source_labels: [__meta_kubernetes_pod_name]
      action: replace
      target_label: kubernetes_pod_name
```

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.manifest* and add these under annotations:

```
prometheus.io/scrape: “true”
prometheus.io/port: “10252”
```

Monitoring the controller manager: what to look for?

Disclaimer: *kube-controller-manager metrics might differ between Kubernetes versions. Here, we used Kubernetes 1.15. You can check the metrics available for your version in the Kubernetes repo (link for the 1.15.3 version).*

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{k8s_app=”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{k8s_app=”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{k8s_app=”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.

```
sum(rate(workqueue_depth{k8s_app=”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:

```
histogram_quantile(0.99, sum(rate(rest_client_request_latency_
seconds_bucket{k8s_app=”kube-controller-manager”}[5m])) by (url, le))
```

- Request rate and errors:

```
sum(rate(rest_client_requests_total{k8s_app=”kube-controller-
manager”,code=~\”2..\”}[5m]))
sum(rate(rest_client_requests_total{k8s_app=”kube-controller-
manager”,code=~\”3..\”}[5m]))
sum(rate(rest_client_requests_total{k8s_app=”kube-controller-
manager”,code=~\”4..\”}[5m]))
sum(rate(rest_client_requests_total{k8s_app=”kube-controller-
manager”,code=~\”5..\”}[5m]))
```

Saturation metrics (requires node_exporter):

- CPU usage:

```
rate(process_cpu_seconds_total{k8s_app=”kube-controller-manager”}
[5m])
```

- Memory usage:

```
process_resident_memory_bytes{k8s_app=”kube-controller-manager”}
```

Examples of issues in kube-controller-manager

Workloads desired and current status mismatch

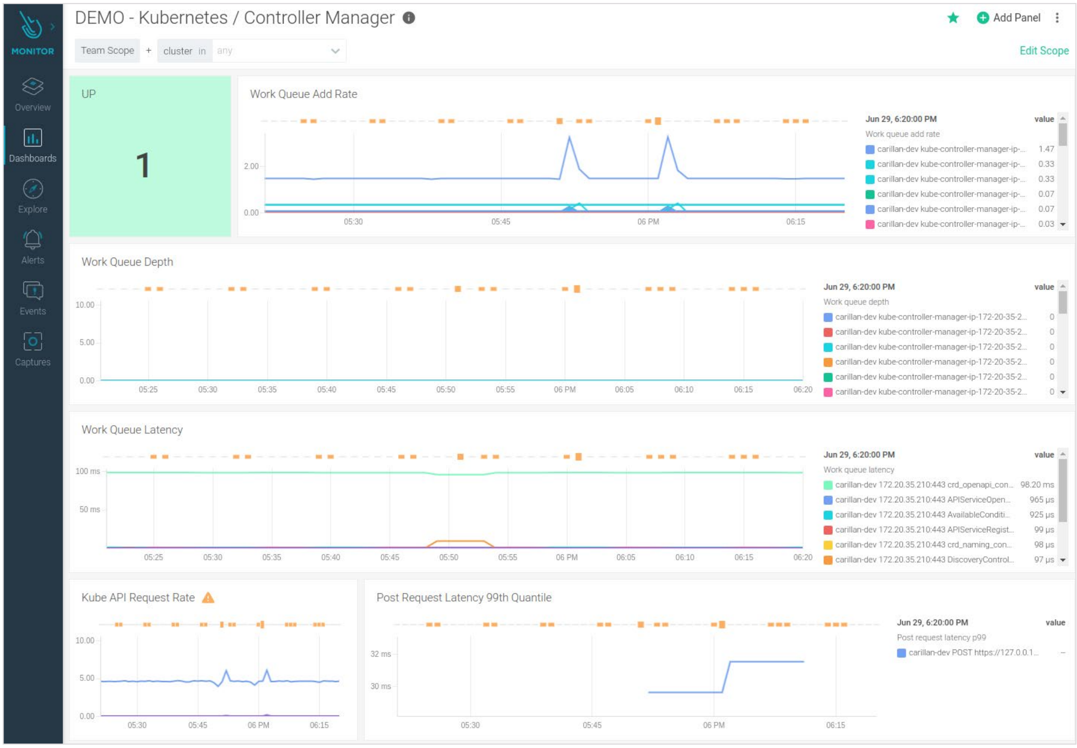
This can be caused by many different issues, but as the kube-controller-manager is the main component responsible with harmonizing current and desired status, we have a possible origin of the issue. Check that the kube-controller-manager instance is up and that the latency of API requests and workqueue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

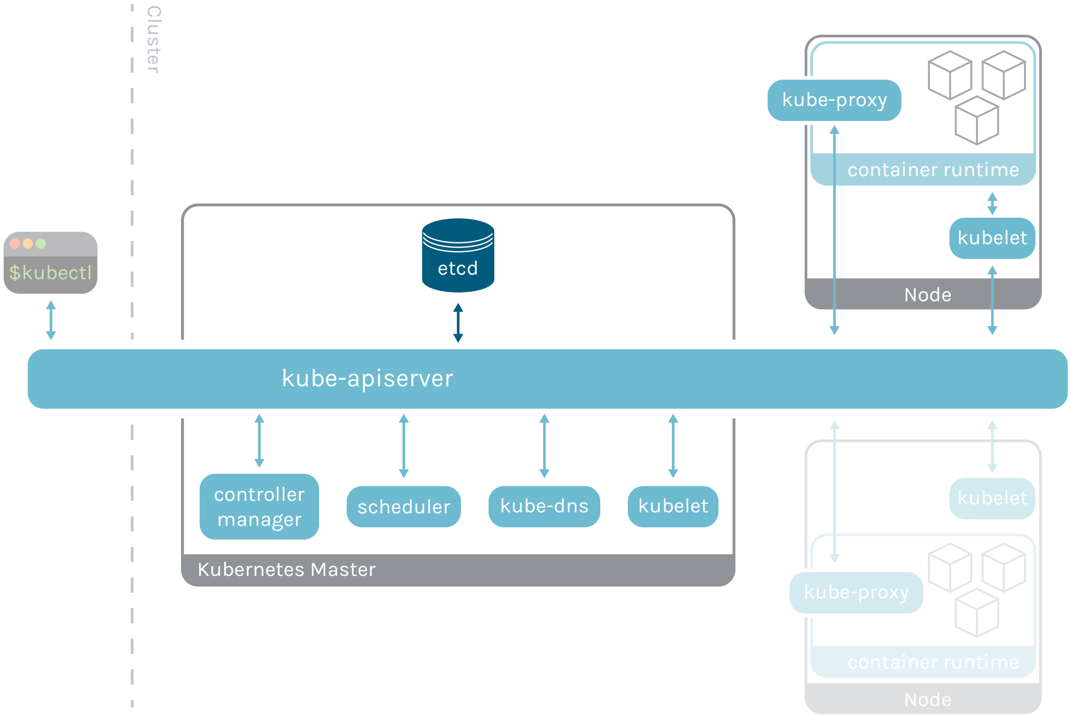
Monitoring kube-controller-manager metrics in Sysdig Monitor

In order to get controller manager monitoring in Sysdig monitor, you just need to add a couple of sections to the Sysdig agent yaml configuration file as we explained above in “Monitoring kubernetes control plane in Sysdig Monitor”: Then you can build custom dashboards using these metrics. Here is what a controller manager dashboard would look like in Sysdig Monitor.


MONITOR
Overview
Dashboards
Explore
Alerts
Events
Captures
DEMO - Kubernetes / Controller Manager
Add Panel
Team Scope
cluster in any
Edit Scope
UP
1
Work Queue Add Rate
2.00
0.00
05:30
05:45
06 PM
06:15
Jun 29, 6:20:00 PM
value
Work queue add rate
carillan-dev kube-controller-manager-ip-... 1.47
carillan-dev kube-controller-manager-ip-... 0.33
carillan-dev kube-controller-manager-ip-... 0.33
carillan-dev kube-controller-manager-ip-... 0.07
carillan-dev kube-controller-manager-ip-... 0.07
carillan-dev kube-controller-manager-ip-... 0.03
Work Queue Depth
10.00
5.00
0.00
05:25
05:30
05:35
05:40
05:45
05:50
05:55
06 PM
06:05
06:10
06:15
06:20
Jun 29, 6:20:00 PM
value
Work queue depth
carillan-dev kube-controller-manager-ip-172-20-35-2... 0
carillan-dev kube-controller-manager-ip-172-20-35-2... 0
carillan-dev kube-controller-manager-ip-172-20-35-2... 0
carillan-dev kube-controller-manager-ip-172-20-35-2... 0
carillan-dev kube-controller-manager-ip-172-20-35-2... 0
carillan-dev kube-controller-manager-ip-172-20-35-2... 0
Work Queue Latency
100 ms
50 ms
05:25
05:30
05:35
05:40
05:45
05:50
05:55
06 PM
06:05
06:10
06:15
06:20
Jun 29, 6:20:00 PM
value
Work queue latency
carillan-dev 172.20.35.210:443 crd_openapi_con... 98.20 ms
carillan-dev 172.20.35.210:443 APIServiceOpen... 965 µs
carillan-dev 172.20.35.210:443 AvailableConditi... 925 µs
carillan-dev 172.20.35.210:443 APIServiceRegist... 99 µs
carillan-dev 172.20.35.210:443 crd_naming_con... 98 µs
carillan-dev 172.20.35.210:443 DiscoveryControl... 97 µs
Kube API Request Rate
10.00
5.00
0.00
05:30
05:45
06 PM
06:15
Post Request Latency 99th Quantile
32 ms
30 ms
05:30
05:45
06 PM
06:15
Jun 29, 6:20:00 PM
value
Post request latency p99
carillan-dev POST https://127.0.0.1... --

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

- Node status can be one of: Follower, Candidate (briefly), Leader.
- If a Follower cannot locate the current Leader, it will become Candidate.
- The voting system will elect a new Leader amongst the Candidates.
- Registry value updates (commits) always go through the Leader.
- Once the Leader has received the ack from the majority of Followers the new value is considered "committed".
- The cluster will survive as long as most of the nodes remain alive.

- Maybe 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, and 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. On more advanced scenarios etcd is often an external service, in these cases you will normally see 3 or more nodes to achieve the required redundancy.

Getting metrics from etcd

Etcd has been instrumented and it exposes Prometheus metrics by default in the port 4001 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 4001 or be in the master itself and you 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-manager-main/etcd-clients-ca.crt** and the key **/etc/kubernetes/pki/etcd-manager-main/etcd-clients-ca.key.**

```
curl https://localhost:4001/metrics -k --cert /etc/kubernetes/pki/
etcd-manager-main/etcd-clients-ca.crt --key /etc/kubernetes/pki/etcd-
manager-main/etcd-clients-ca.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 4001 open, then you can access with the ip too.

```
curl https://[master_ip]:4001/metrics -k --cert /etc/kubernetes/pki/
etcd-manager-main/etcd-clients-ca.crt --key /etc/kubernetes/pki/etcd-
manager-main/etcd-clients-ca.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-manager-main/etcd-clients-ca.key** and **/etc/kubernetes/pki/etcd-manager-main/etcd-clients-ca.crt** , just download the certificates 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 be potentially a target.*

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

```
kubectl -n monitoring patch deployment prometheus-server -p ‘{“spec”:
{“template”:{“spec”:{“volumes”:[{“name”:”etcd-ca”,”secret”:{“defaultM
ode”:420,”secretName”:”etcd-ca”}}]}}}}’
kubectl -n monitoring patch deployment prometheus-server -p
‘{“spec”:{“template”:{“spec”:{“containers”:[{“name”:”prometh
eus-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-manager-main.+’
      - source_labels:
        - __address__
        action: replace
        target_label: __address__
        regex: (.+?)(\\:\\d)?
        replacement: $1:4001
      tls_config:
        insecure_skip_verify: true
        cert_file: /opt/prometheus/secrets/etcd-clients-ca.crt
        key_file: /opt/prometheus/secrets/etcd-clients-ca.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 1.15. You can check the metrics available for your version in the Kubernetes repo (link for the 1.15.3 version).*

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 someone is down or directly 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 has not a leader this node will be unavailable and if all nodes have no leader then the cluster will become totally unavailable. To check this 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 the leader changes too often then these changes can impact the performance of the etcd itself. This also can 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 (for example a write request, a configuration change request) that needs to go through raft protocol. The proposals 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 one. If there are proposals failed can be for two reasons, the leader election is failing or there is a loss of the quorum.

For example, if we wanted to set an alert to show that there were more than 5 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 commiting changes to disk and the health of your storage is a key indicator 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]))
```

Examples of issues in etcd

Most of the time your etcd cluster works so neatly, 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 re-connect etcd and Kubernetes again, state incoherences could cause additional malfunction.

Losing the quorum

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

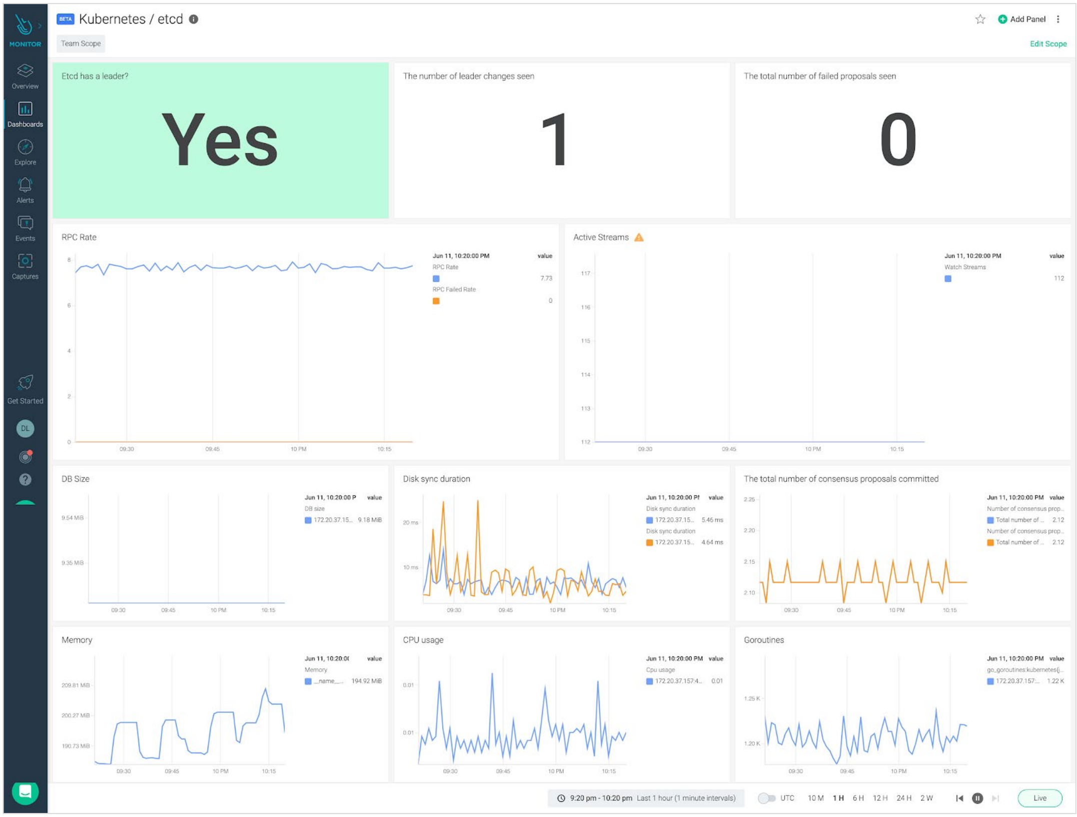
Apply entries took too long

If you see the message *apply entries took too long* is 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, you just need to add a couple of sections to the Sysdig agent yaml configuration file as we explained above in “Monitoring kubernetes control plane in Sysdig Monitor”. Then you can build custom dashboards using these metrics. Here is what an etcd dashboard would look like in Sysdig Monitor.

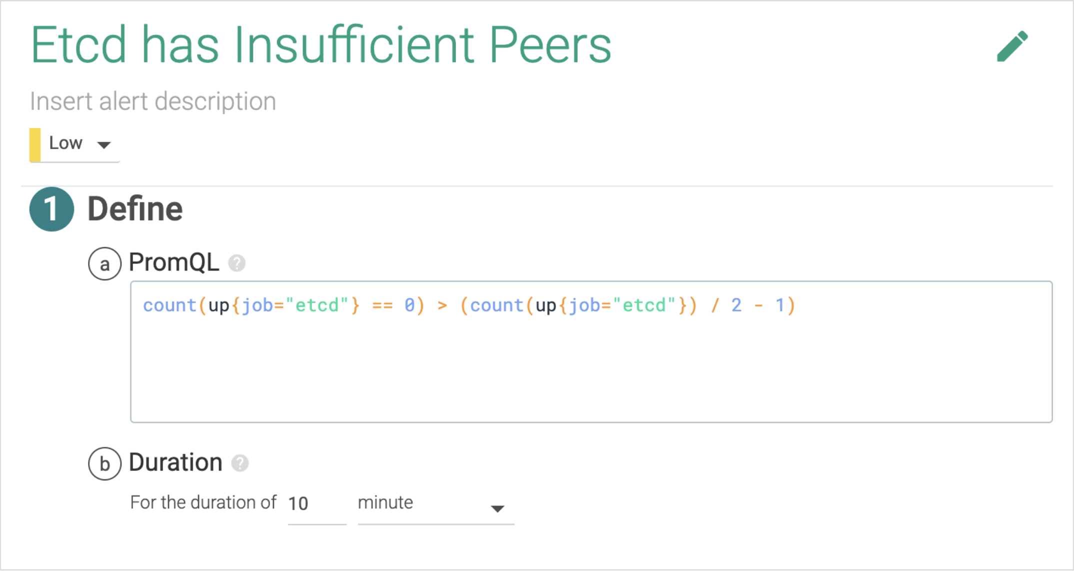

Kubernetes / etcd
Team Scope
Add Panel
Edit Scope
Etcd has a leader?
Yes
The number of leader changes seen
1
The total number of failed proposals seen
0
RPC Rate
Active Streams
DB Size
Disk sync duration
The total number of consensus proposals committed
Memory
CPU usage
Goroutines
Overview
Dashboards
Explore
Alerts
Events
Captures
Get Started
9:20 pm - 10:20 pm Last 1 hour (1 minute intervals)
UTC 10 M 1 H 6 H 12 H 24 H 2 W
Live

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.

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.



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

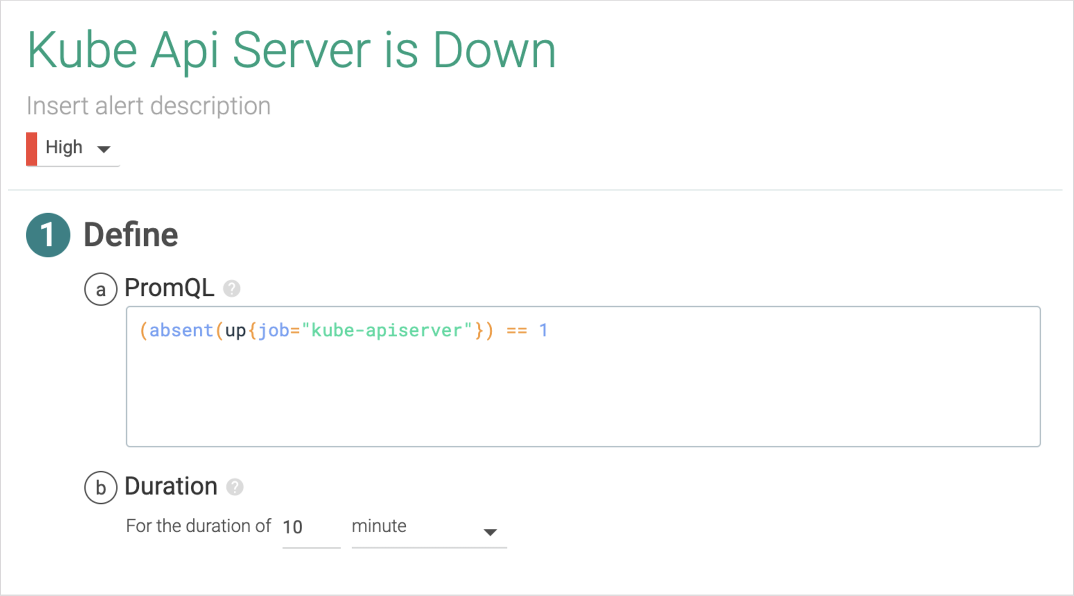
For the duration of 10 minute

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

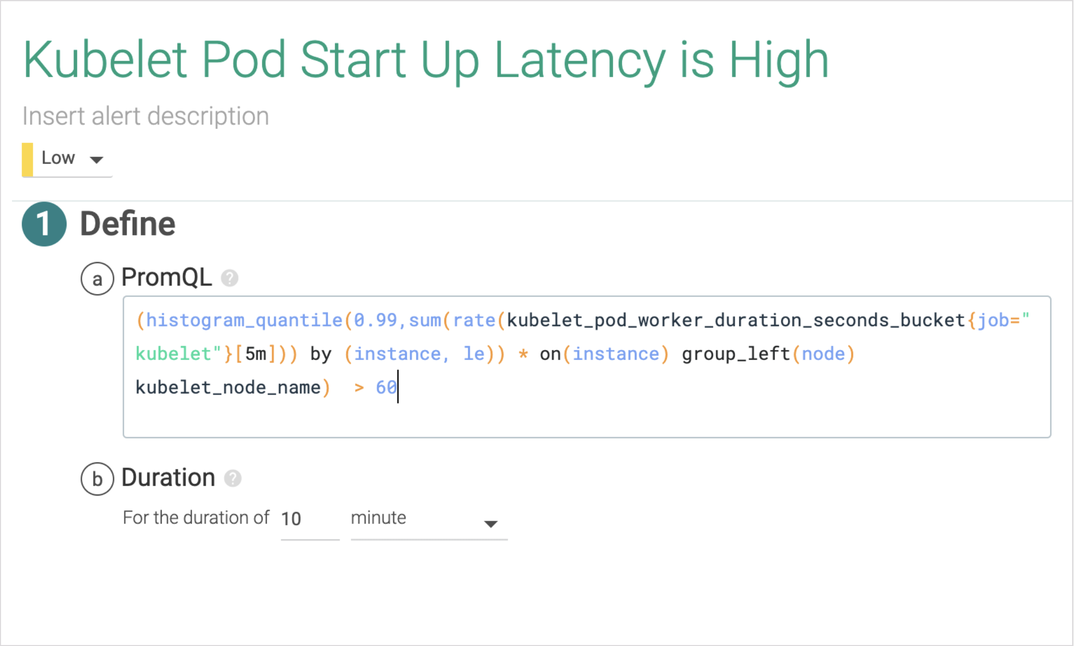
For the duration of 10 minute

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

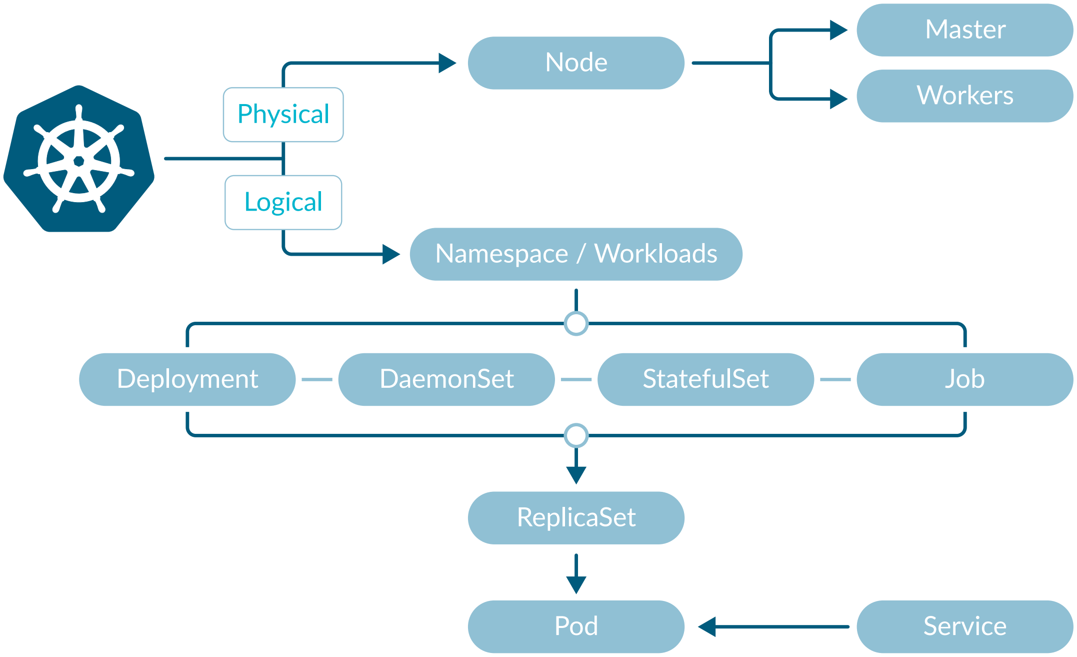
When looking at the service level, it shouldn't be much different from what you were doing before Kubernetes if you had already clustered your services. Think of databases like MySQL/MariaDB or MongoDB, where you will look at the replication status and lag. Is there anything to take into account now?

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 the section 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 have in 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 respawn 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 I 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 the three 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.

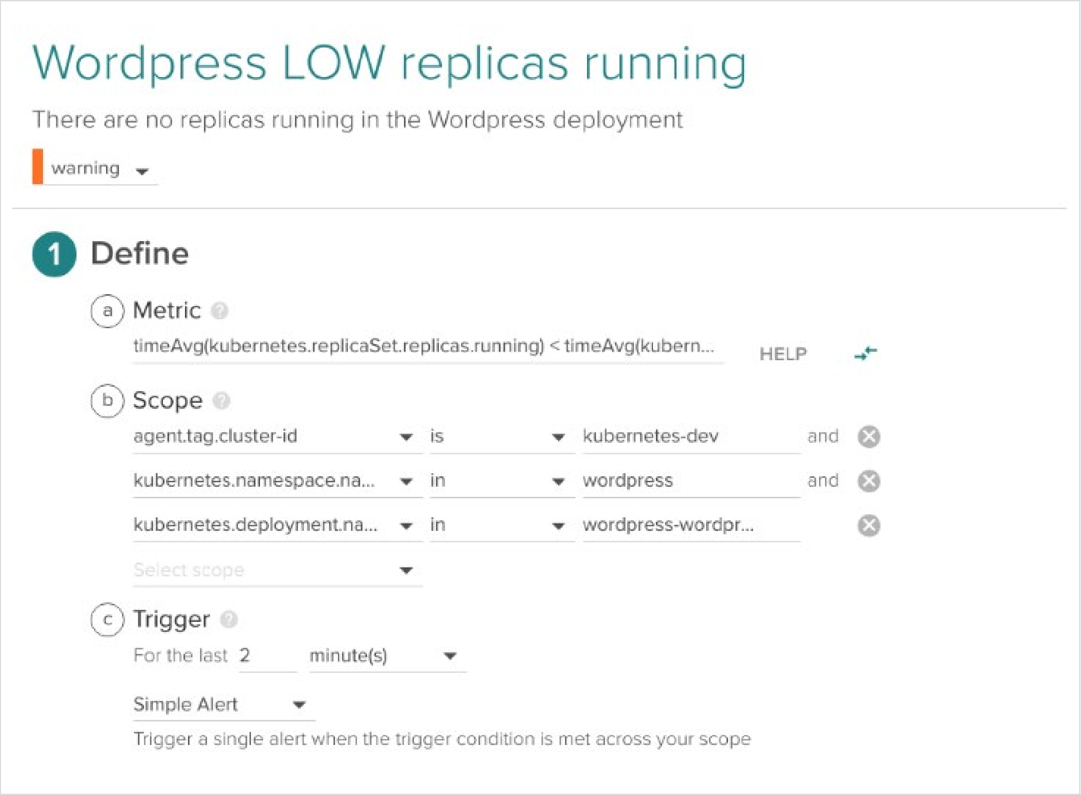
If the number of replicas or instances running during an extended period of time is lower than the number of replicas you desire, it's a symptom of something not working properly (not enough nodes or resources available, Kubernetes or Docker Engine failure, Docker image broken, etc.).

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 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...

b Scope

agent.tag.cluster-id is kubernetes-dev and

kubernetes.namespace.na... in wordpress and

kubernetes.deployment.na... in wordpress-wordpr...

Select scope

c Trigger

For the last 2 minute(s)

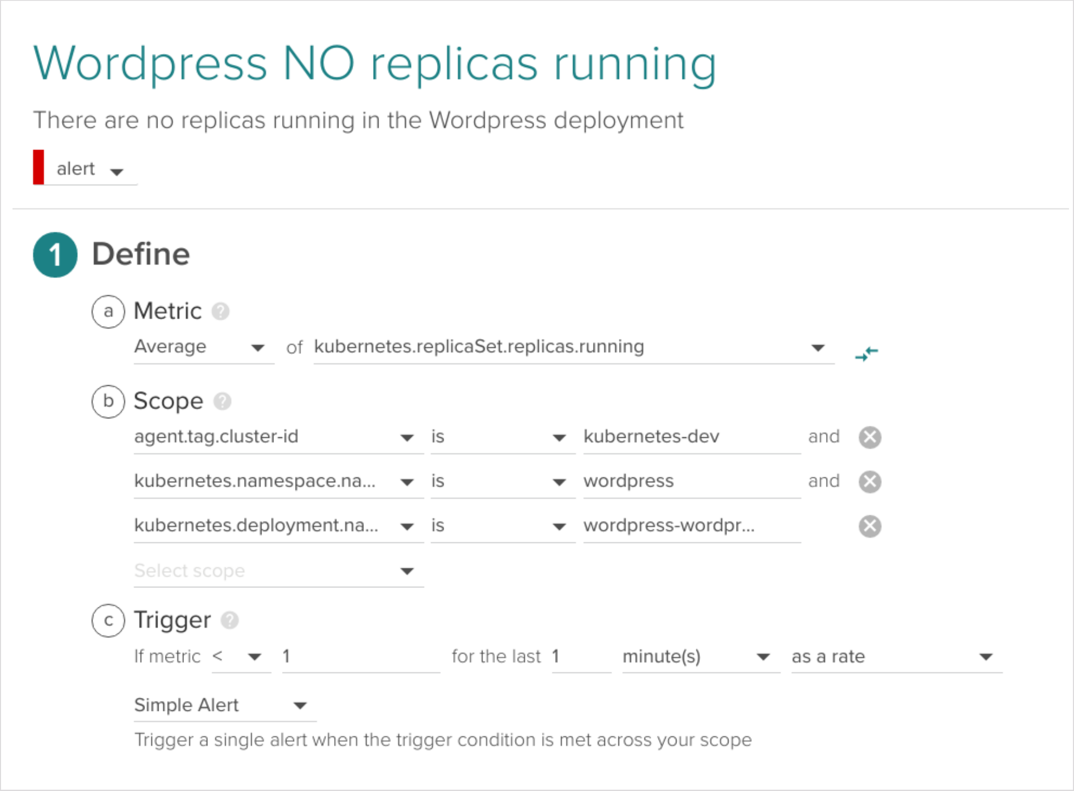
Simple Alert

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

Do I 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 following example, you apply the alert for the same deployment but trigger if running pods is < 1 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
kubernetes.namespace.na...	is	wordpress	and
kubernetes.deployment.na...	is	wordpress-wordpr...	

Select scope

c Trigger

If metric < 1 for the last 1 minute(s) as a rate

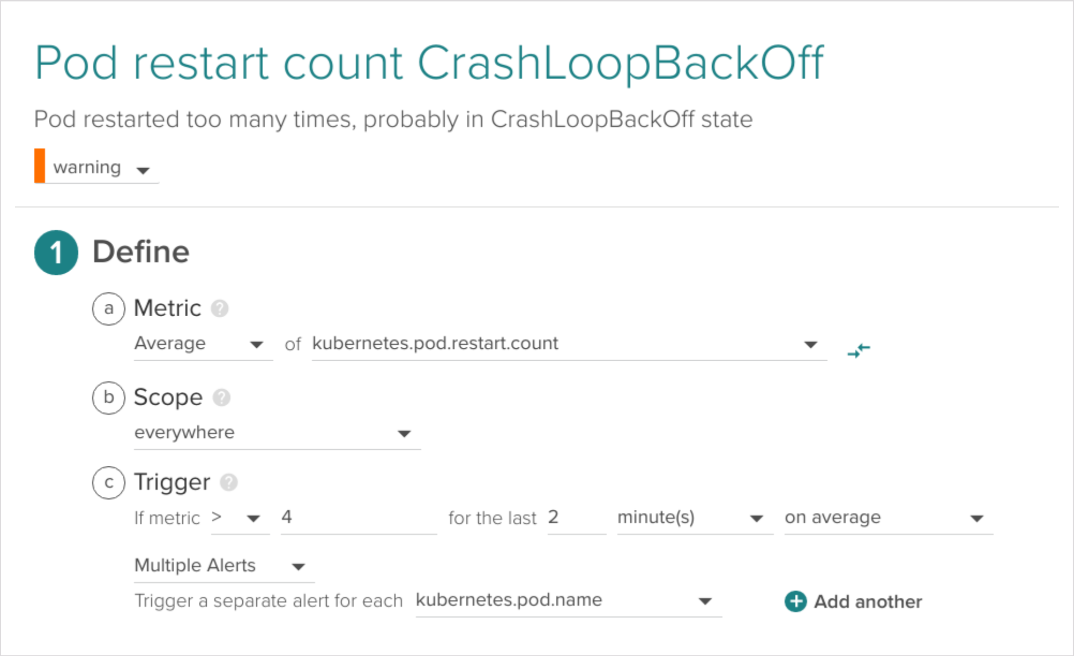
Simple Alert

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

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, paradoxically, 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.

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 2 cores.
- The sum of all the CPU limits can't be higher than 3 cores.
- The sum of all the memory requests can't be higher than 2 GiB.
- The sum of all the memory limits can't be higher than 4 GiB.

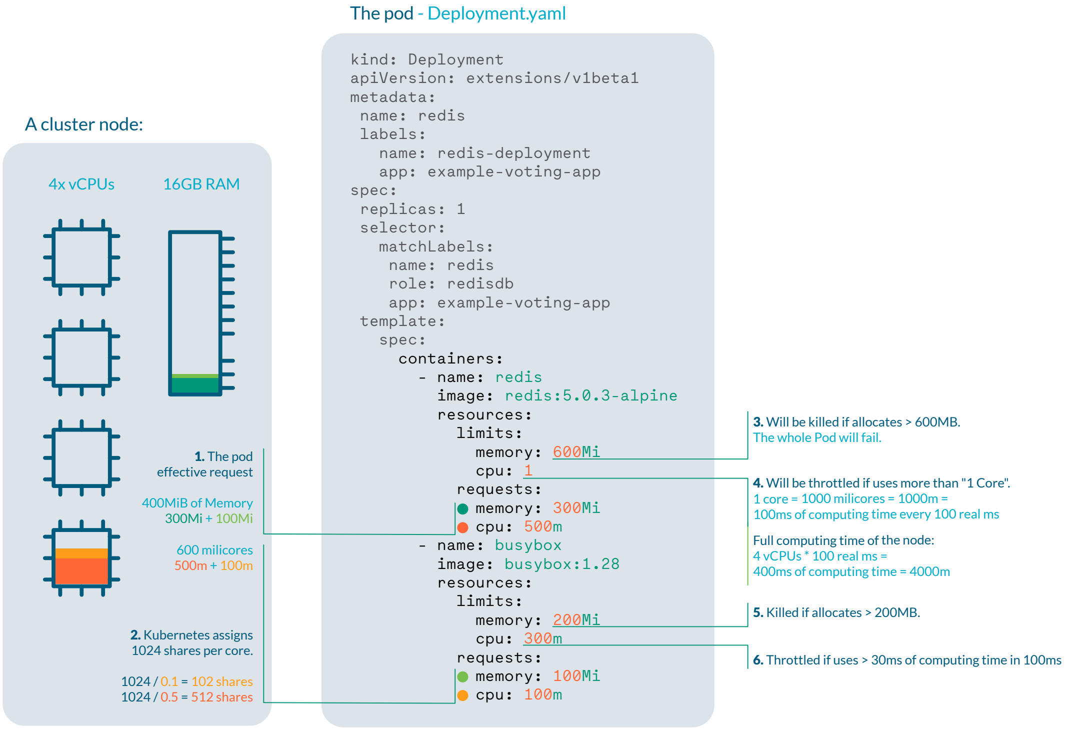
If you already have 1.9 cores allocated with pods and try to allocate a new pod with 200m of CPU request, the pod will not be scheduled and will remain in a ***pending*** state.

Explaining pod requests and limits

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, 4 cores and 16GB RAM nodes. You can extract a lot of information:



```yaml
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
```

1. The pod effective request

400MiB of Memory
300Mi + 100Mi

600 milicores
500m + 100m

2. Kubernetes assigns 1024 shares per core.

1024 / 0.1 = 102 shares
1024 / 0.5 = 512 shares

3. Will be killed if allocates > 600MB.
The whole Pod will fail.

4. Will be throttled if uses more than "1 Core".
1 core = 1000 milicores = 1000m =
100ms of computing time every 100 real ms

Full computing time of the node:
4 vCPUs * 100 real ms =
400ms of computing time = 4000m

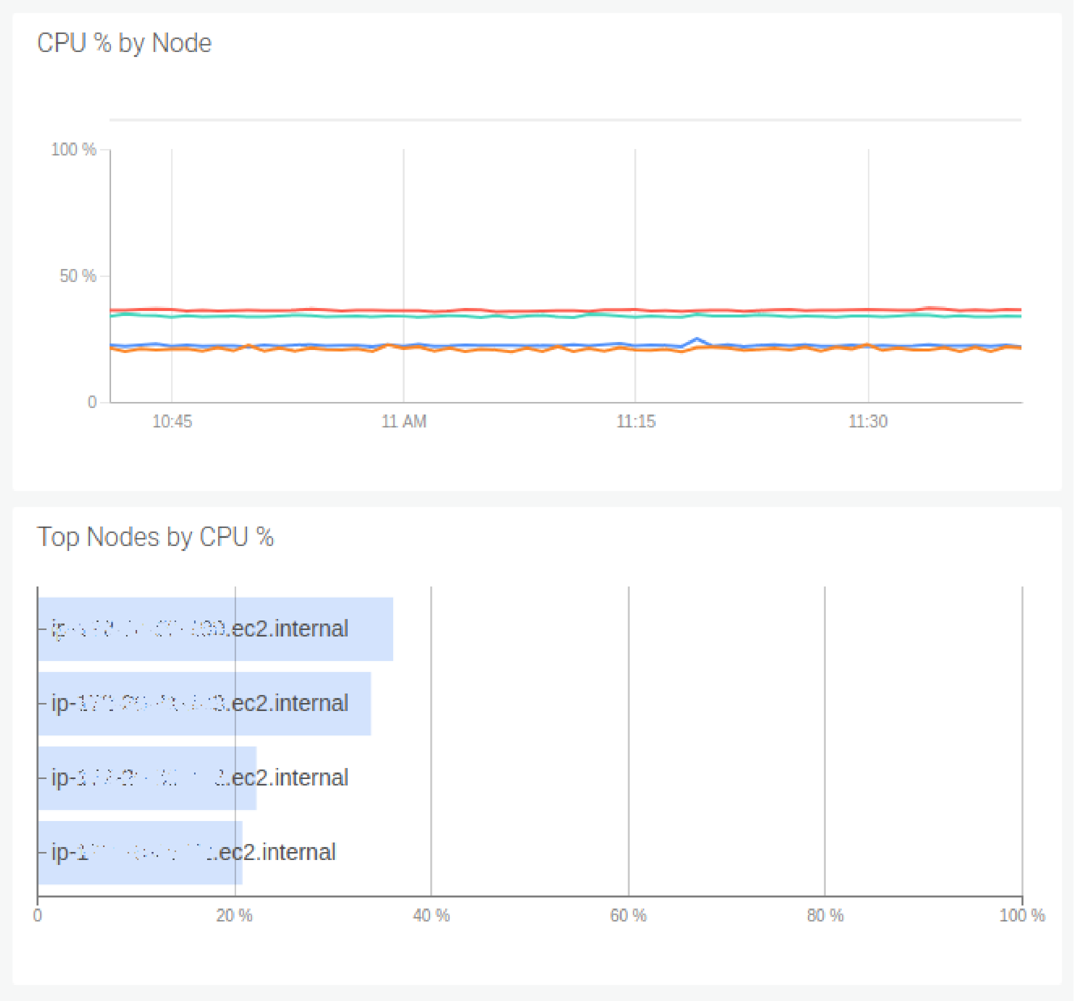
5. Killed if allocates > 200MB.

6. Throttled if uses > 30ms of computing time in 100ms

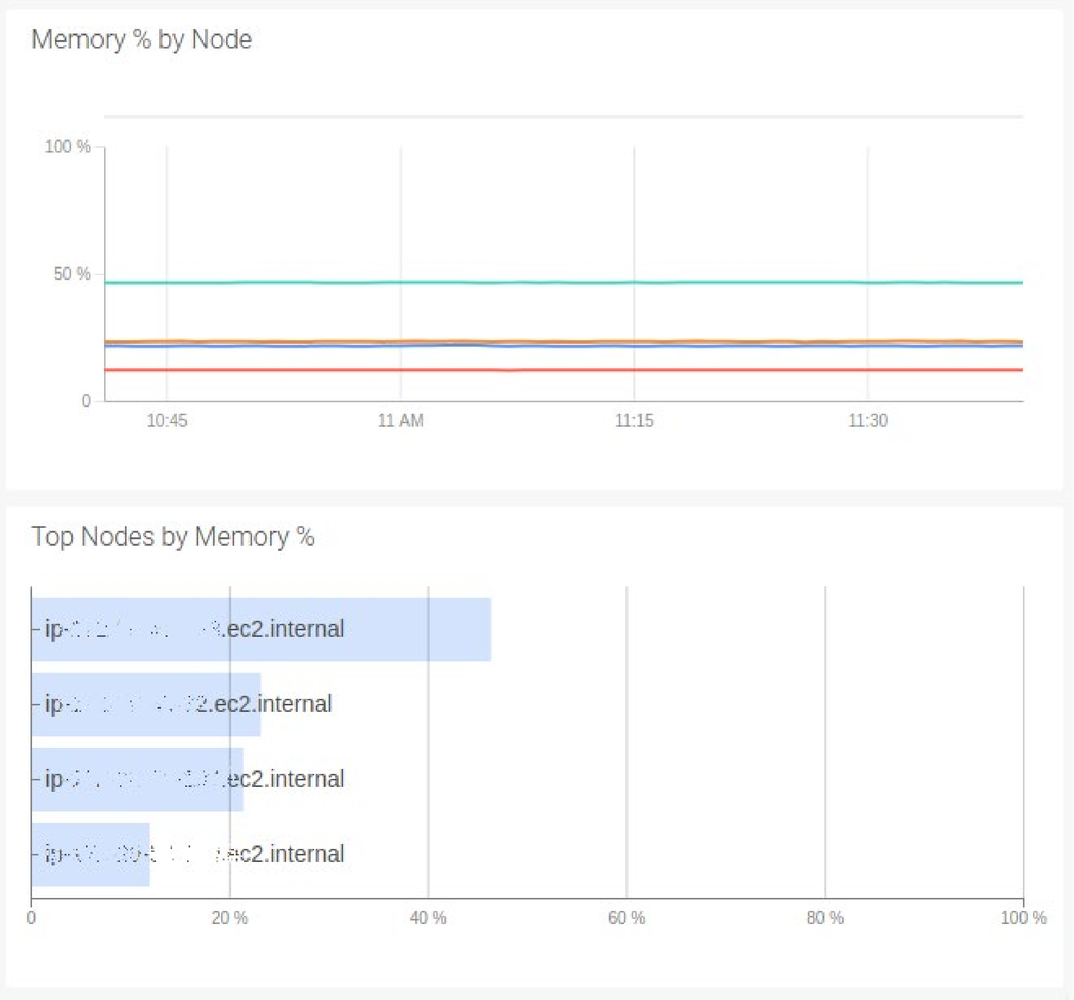
1. **Pod effective request** is 400 MiB of memory and 600 millicores of CPU. You need a node with enough free allocatable space to schedule the pod.
2. **CPU shares** for the redis container will be 512, and 102 for the busybox container. Kubernetes always assign 1024 shares to every core, so:
    - redis: 1024 / 0.5 cores ≅ 512
    - busybox: 1024 / 0.1cores ≅ 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 4 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.

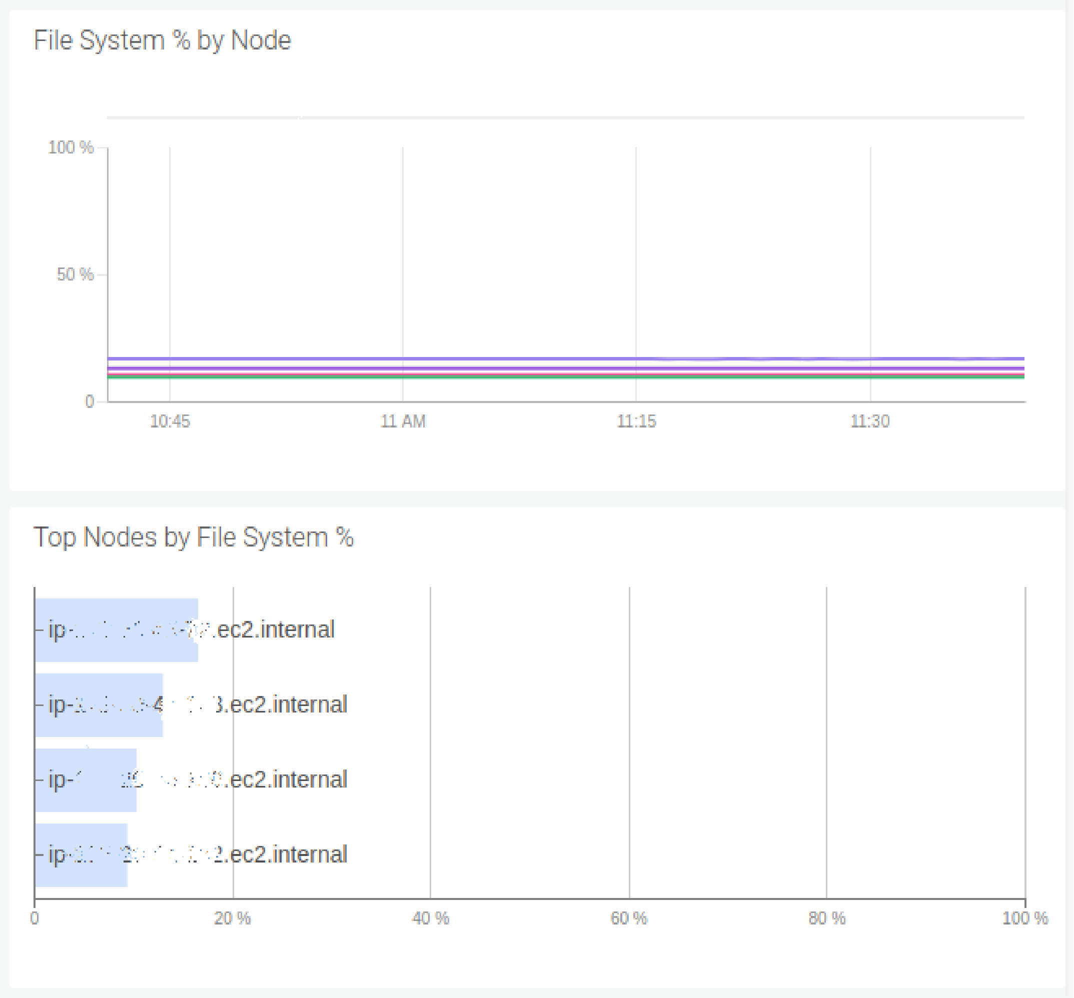

CPU % by Node
100 %
50 %
0
10:45
11 AM
11:15
11:30
Top Nodes by CPU %
ip-[redacted].ec2.internal
ip-[redacted].ec2.internal
ip-[redacted].ec2.internal
ip-[redacted].ec2.internal
0
20 %
40 %
60 %
80 %
100 %

Find these metrics in Sysdig Monitor in the dashboard: Kubernetes → Resource usage → Kubernetes node health



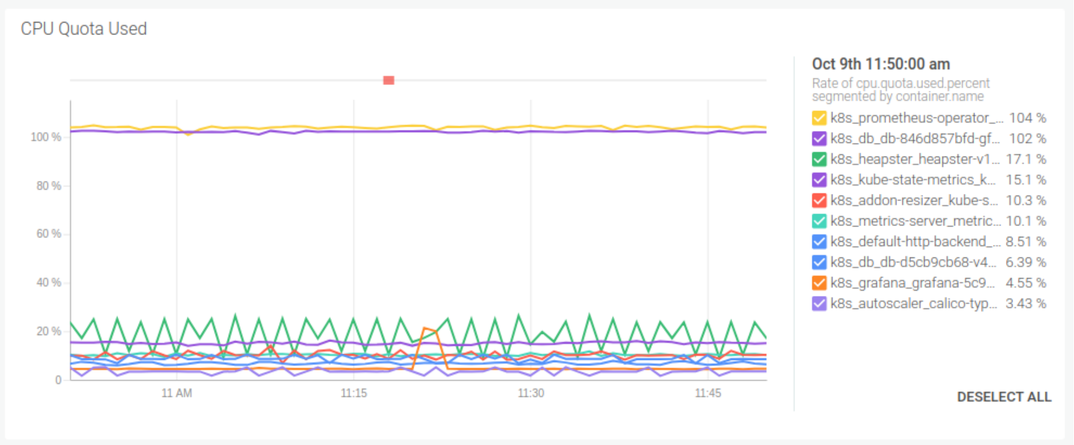
Find these metrics in Sysdig Monitor in the dashboard: Kubernetes → Resource usage → Kubernetes node health

- 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.


File System % by Node
100 %
50 %
0
10:45
11 AM
11:15
11:30
Top Nodes by File System %
ip-[redacted].ec2.internal
ip-[redacted].ec2.internal
ip-[redacted].ec2.internal
ip-[redacted].ec2.internal
0
20 %
40 %
60 %
80 %
100 %

Find these metrics in Sysdig Monitor in the dashboard: Kubernetes → Resource usage → Kubernetes node health

- 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.



CPU Quota Used

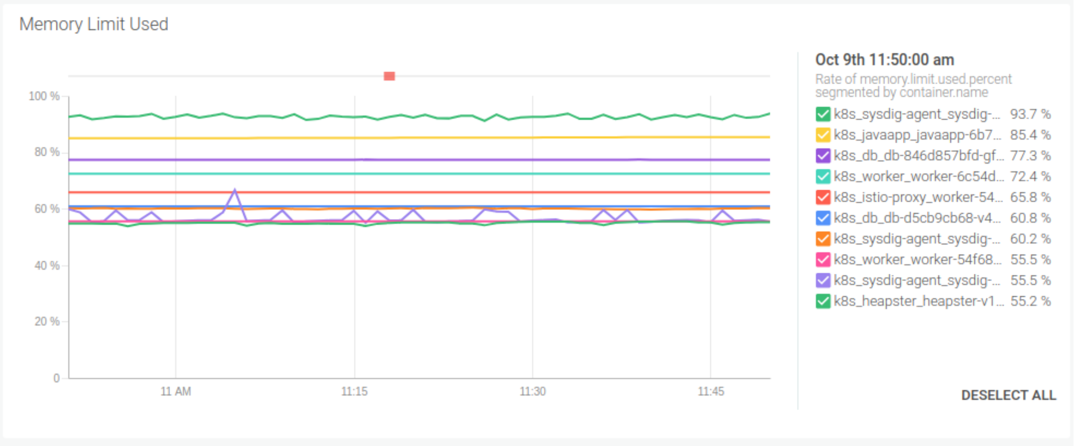
Oct 9th 11:50:00 am

Rate of cpu.quota.used.percent segmented by container.name

☑ k8s_prometheus-operator_...	104 %
☑ k8s_db_db-846d857bfd-gf...	102 %
☑ k8s_heapster_heapster-v1...	17.1 %
☑ k8s_kube-state-metrics_k...	15.1 %
☑ k8s_addon-resizer_kube-s...	10.3 %
☑ k8s_metrics-server_metric...	10.1 %
☑ k8s_default-http-backend_...	8.51 %
☑ k8s_db_db-d5cb9cb68-v4...	6.39 %
☑ k8s_grafana_grafana-5c9...	4.55 %
☑ k8s_autoscaler_calico-typ...	3.43 %

Find these metrics in Sysdig Monitor in the dashboard: Hosts & containers → Container 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 redis container is using 100MiB and busybox container is using 200MiB, the pod will fail.



Memory Limit Used

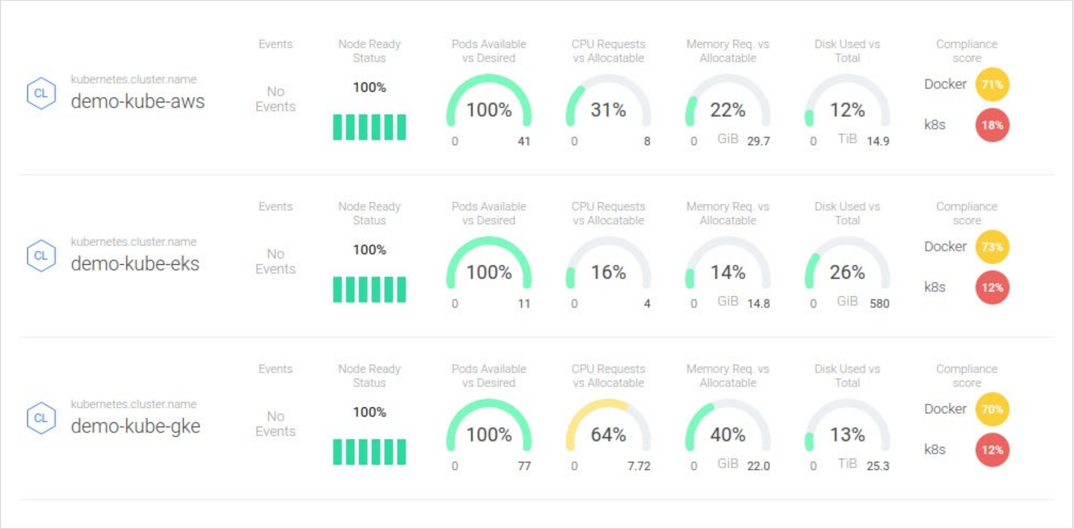
Oct 9th 11:50:00 am

Rate of memory.limit.used.percent segmented by container.name

☑ k8s_sysdig-agent_sysdig-...	93.7 %
☑ k8s_javaapp_javaapp-6b7...	85.4 %
☑ k8s_db_db-846d857bfd-gf...	77.3 %
☑ k8s_worker_worker-6c54d...	72.4 %
☑ k8s_istio-proxy_worker-54...	65.8 %
☑ k8s_db_db-d5cb9cb68-v4...	60.8 %
☑ k8s_sysdig-agent_sysdig-...	60.2 %
☑ k8s_worker_worker-54f68...	55.5 %
☑ k8s_sysdig-agent_sysdig-...	55.5 %
☑ k8s_heapster_heapster-v1...	55.2 %

Find these metrics in Sysdig Monitor in the dashboard: Hosts & containers → Container limits

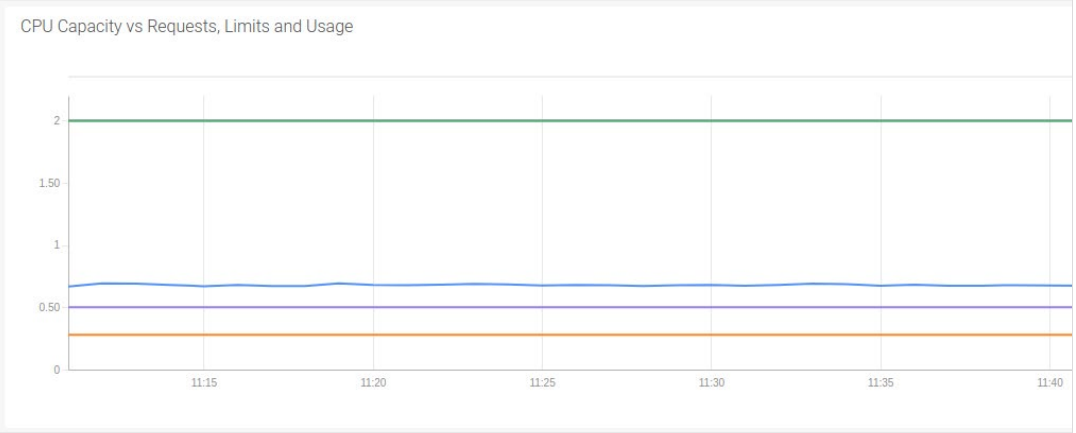
- Percentage of resource allocation in the cluster and the nodes. You can represent this as a percentage of resources allocated from total available resources. A good warning threshold would be **(n-1)/n * 100**, where **n** is the number of nodes. Over this threshold, in case of a node failure, you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



kubernetes.cluster.name	Events	Node Ready Status	Pods Available vs Desired	CPU Requests vs Allocatable	Memory Req. vs Allocatable	Disk Used vs Total	Compliance score
demo-kube-aws	No Events	100%	100% 0 41	31% 0 8	22% 0 GiB 29.7	12% 0 TiB 14.9	Docker 71% k8s 18%
demo-kube-eks	No Events	100%	100% 0 11	16% 0 4	14% 0 GiB 14.8	26% 0 GiB 580	Docker 73% k8s 12%
demo-kube-gke	No Events	100%	100% 0 77	64% 0 7.72	40% 0 GiB 22.0	13% 0 TiB 25.3	Docker 70% k8s 12%

Find these metrics in Sysdig Monitor in the 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.

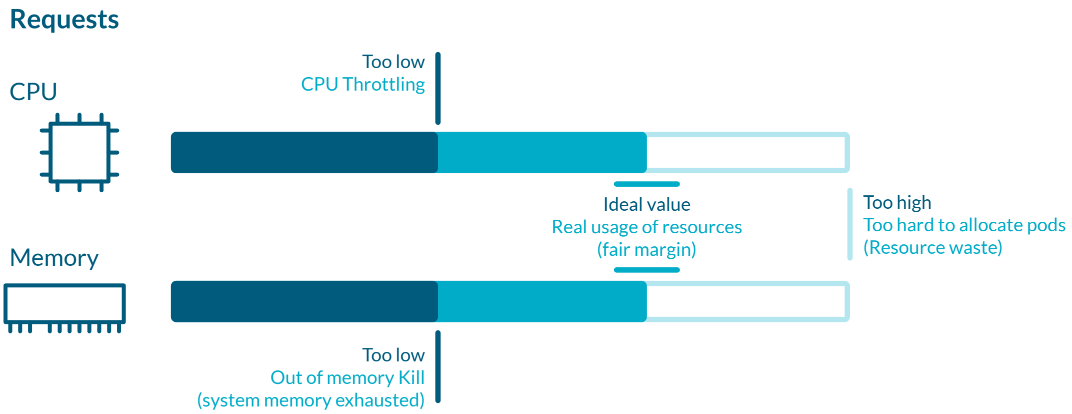


Custom graph showing cpu usage vs. capacity vs. limits vs. requests.

Choosing pragmatic requests and limits

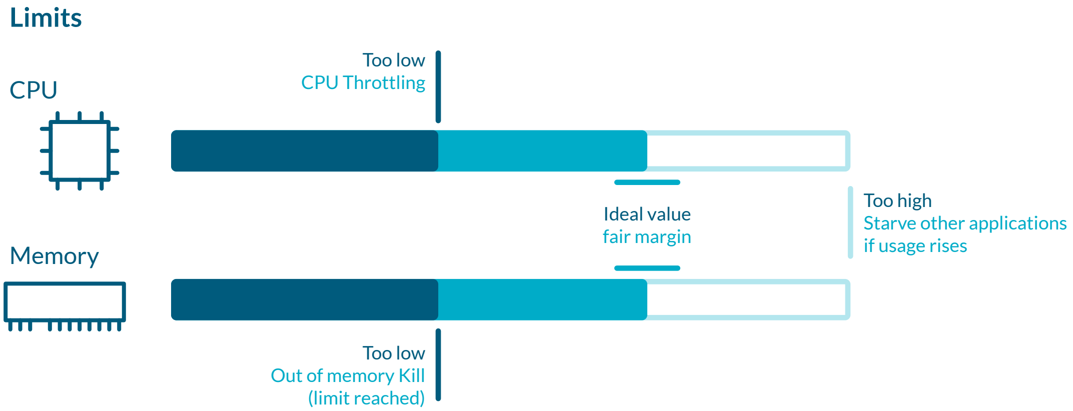
When you have some experience with Kubernetes, you usually understand (the hard way) that properly setting requests and limits is of utmost importance for the performance of the applications and cluster.

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.



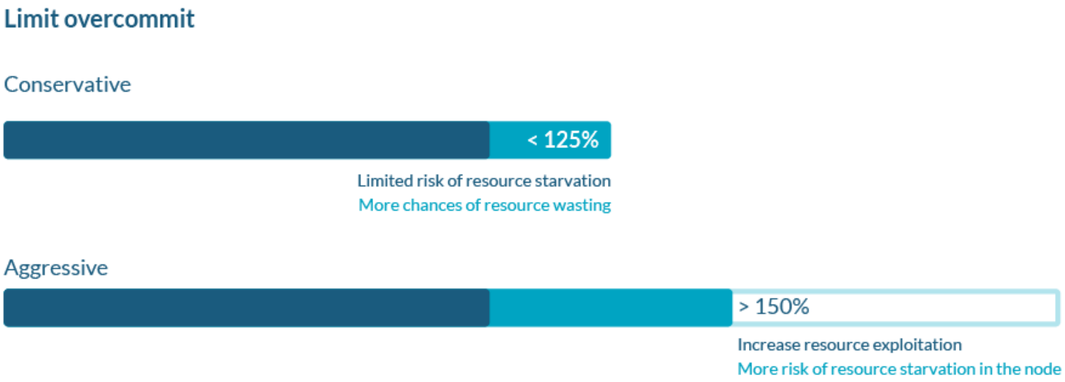
Requests	Too low	Ideal value	Too high
CPU	CPU throttling	Real use of resources (fair margin)	Hard to allocate pods Resource waste
Memory	OOM kill (system memory exhausted)		

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	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
< 125%
Limited risk of resource starvation
More chances of resource wasting
Aggressive
> 150%
Increase resource exploitation
More risk of resource starvation in the node

Limit overcommit	Conservative Less than 125%	Aggressive More than 150%
Good	Limited risk of resource starvation	Increase resource exploitation
Warning	More chances 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.

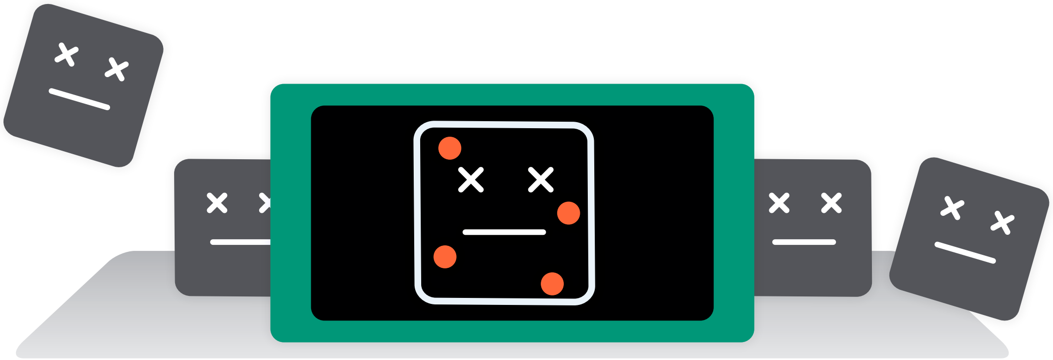
Lessons learned

Some lessons you should learn from this are:

- Set requests and limits in your workloads.
- Setting a namespace quota will enforce all of the workloads in the namespace to have a request and limit in every container.
- 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.

How to troubleshoot Kubernetes OOM and CPU Throttle

Experience Kubernetes OOM kills can be very frustrating. Why is my application struggling if I 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

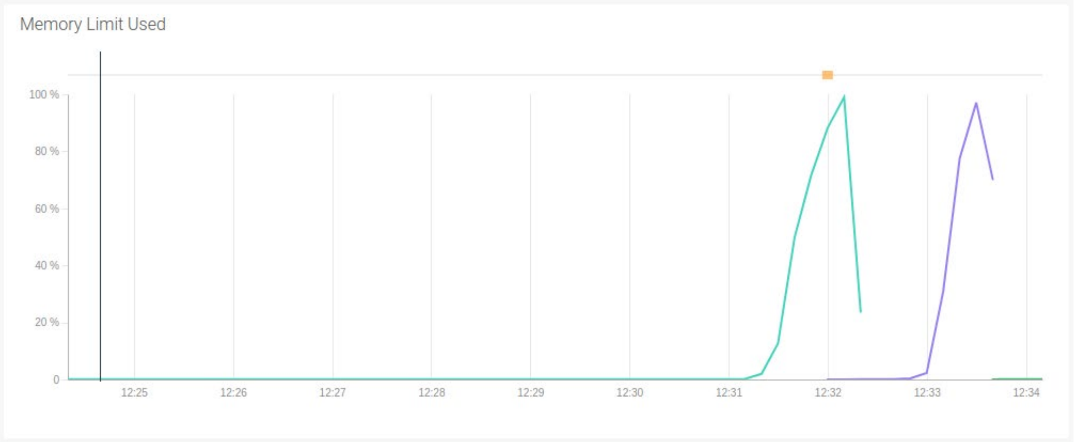
This is by far the most simple memory error you can have in a pod. You set a memory limit, one container tries to allocate more memory than allowed, and it gets an error. This usually ends up with a container dying, one pod unhealthy and Kubernetes restarting that pod.

```
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: Hosts & containers → Container limits

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. 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 only every several seconds. 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

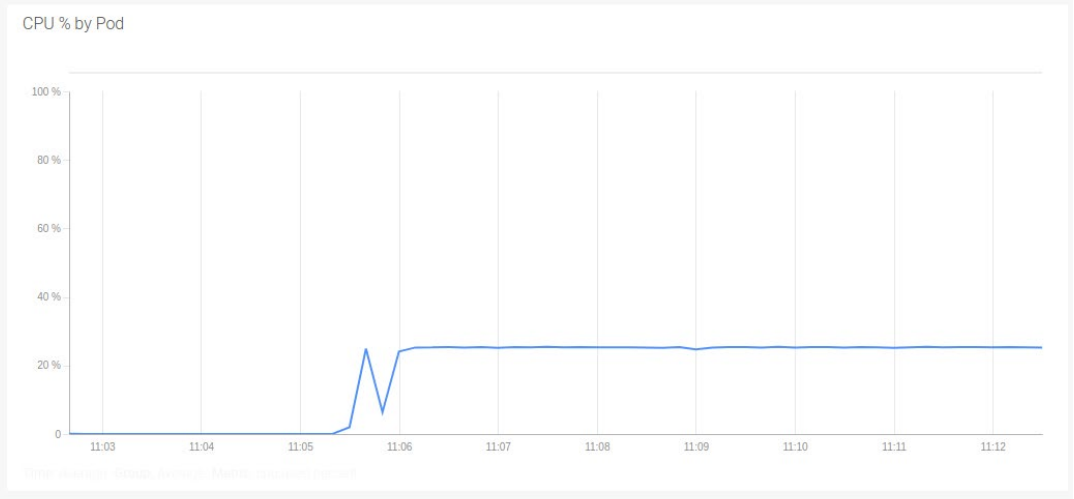
There are many differences on how CPU and memory requests and limits are treated in Kubernetes. A container using more memory than the limit will most likely die, but using CPU can never be the reason that Kubernetes kills a container. CPU management is delegated to the system scheduler, and it uses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

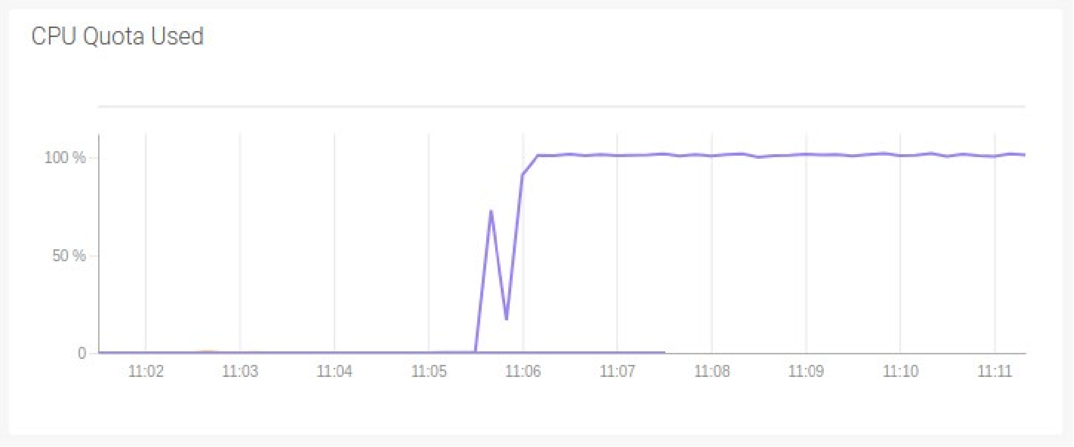
On the other hand, limits are treated differently. Limits are managed with the CPU quota system. This works by dividing the CPU time in 100ms periods and assigning a limit on the containers with the same percentage that the limit represents to the total CPU in the node.

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: Hosts & containers → Container limits



Find these metrics in Sysdig Monitor in the dashboard: Hosts & containers → Container 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.

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

www.sysdig.com