Scaling Kubernetes: Best Practices

Platform9 Kubernetes Journey eBook series

- Understanding Kubernetes
- Architecting Kubernetes Deployments
- Operating Kubernetes
- Scaling Kubernetes ← This eBook
- Securing Kubernetes

Download at: https://www.platform9.com/k8s-journey-eBooks

TABLE OF CONTENTS

Introduction

Kubernetes is designed to be a highly scalable system, out-of-the-box. It does an excellent job of automatically scaling workloads, allocating resources, balancing application demand and so on.

This does not mean, however, that Kubernetes is a set-it-and-forget-it platform. There are a variety of extra steps that admin teams should take to make sure that Kubernetes performs optimally, especially at scale.

Optimizing the performance of Kubernetes offers a range of benefits. First and foremost, it ensures that applications hosted on Kubernetes are as responsive as possible. It also helps to mitigate infrastructure costs by helping Kubernetes clusters to make the very most of the compute, memory and other resources available to them. And it helps admin teams to keep their Kubernetes management processes as streamlined as possible.

With these goals in mind, Platform9 has prepared this eBook as a resource for teams seeking to optimize their Kubernetes clusters at scale. The following chapters offer insight into a range of topics related to optimization, such as managing cluster and application performance, and addressing networking challenges that arise in large-scale Kubernetes environments. They also offer tips on the best methodologies to use for managing Kubernetes configurations and monitoring Kubernetes environments at scale.

The objective of this guide is to help organizations that leverage Kubernetes to get more out of the platform than what it offers by default. Kubernetes is a powerful tool, out-of-the-box; but it becomes even more effective when tweaked to maintain optimal performance, even in the face of the challenges that arise in large-scale environments.

Kubernetes Multi-Tenancy Best Practices

In a simpler world, all Kubernetes deployments would host just a single tenant – meaning only one workload or application runs on the entire Kubernetes environment.

In the real world, most production Kubernetes deployments have multiple tenants. In a multi-tenant deployment, multiple workloads share the same underlying Kubernetes infrastructure.

Multi-tenant Kubernetes creates some special challenges when it comes to resource sharing and security. Let's walk through what those problems are and some strategies for addressing them.

What is multi-tenant Kubernetes?

Multi-tenant Kubernetes is a Kubernetes deployment where multiple applications or workloads run side-by-side.

Multi-tenancy is a common architecture for organizations that have multiple applications running in the same environment, or where different teams (like developers and IT Ops) share the same Kubernetes environment.

You can think of multi-tenant Kubernetes as being akin to an apartment building, whereas single-tenant Kubernetes is like a single-family house.

Challenges of Kubernetes multi-tenancy

We can take that metaphor further to help explain the challenges of multi-tenant Kubernetes environments.

In an apartment building, you have to ensure that each apartment is sufficiently isolated from the others in order to give each tenant appropriate privacy. You can't have one tenant walking through another tenant's apartment to get to the bathroom, for example.

Similarly, when it comes to resource sharing, you must ensure that each tenant has access to the resources they need. Your building won't function very well if one apartment's water shuts off when another tenant is taking a shower. Of course, none of these things would be an issue in a single-family home, where there is no sharing of resources.

Multi-tenancy Kubnernetes poses similar challenges. Each workload must be isolated so that a security vulnerability or breach in one workload doesn't spill over into another. Likewise, each workload must have fair access to the compute, networking, and other resources provided by Kubernetes. (We say "fair access" because you might not necessarily want each workload to have the same level of access as others, but you do want each workload to be able to access what it needs to do its job.)

Solving Kubernetes multi-tenancy challenges

How can you address these challenges and ensure that each Kubernetes workload has the proper level of isolation and resource availability? The following are some best practices:

Namespace isolation

A basic best practice for handling multiple tenants is to assign each tenant a separate namespace. Kubernetes was

designed for this approach. Most of the isolation features that it provides expect you to have a separate namespace for each entity that you want to isolate.

> *Keep in mind, too, that in some cases it may be desirable to assign multiple namespaces to the same group within your Kubernetes deployment. For example, the same team of developers might need multiple namespaces for hosting different builds of their application.*

Adding namespaces is relatively easy (it takes just a simple *kubectl create namespace your-namespace* command), and it's always better to have the ability to separate workloads in a granular way using namespaces than to try to cram different workloads with different needs into the same namespace.

Block traffic between namespaces

By default, most Kubernetes deployments allow network communication between namespaces. If you need to support multiple tenants, you'll want to change this in order to add isolation to each namespace.

You can do this using Network Policies. Here's an example Network Policy file that will block traffic from external namespaces:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: block-external-namepsace-traffic
spec:
  podSelector:
    matchLabels:
  ingress:
  - from:
    - podSelector: {}
```

After creating the file, apply it with a command like:

```
kubectl apply -f networkpolicy.yaml -n=your-namespace
```

Resource Quotas

When you want to ensure that all Kubernetes tenants have fair access to the resources that they need, Resource Quotas are the solution to use. As the name of this feature implies, it lets you set quotas on how much CPU, storage, memory, and other resources can be consumed by all pods within a namespace.

For example, consider this Resource Quota file:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-demo
spec:
  hard:
    requests.cpu: "1"
    limits.cpu: "2"
```

It can be applied with a command like:

```
kubectl apply -f resourcequota.yaml  -n=your-namespace
```

When applied, the policy will limit all containers within a given namespace to 1 CPU request and 2 total CPUs.

Although you could enforce the same resource quota for all namespaces, you are also totally free to set different quotas for different namespaces. Thus, you can give some tenants access to more resources if desired. Doing so could be helpful for ensuring that production workloads have more resources than tenants that exist only for testing, for example, or if you want to allow some users to pay more for access to additional resources.

Secure your nodes

A final multi-tenancy best practice to keep in mind is the importance of making sure that your master and worker nodes are secure at the level of the host operating system. Node security doesn't reinforce namespace isolation in a direct way; however, since an attacker who is able to compromise a node on the operating system level can potentially use that breach to take control of any workloads that depend on the node, node security is important to keep in mind. (It would be important in a single-tenant environment too, but it's even more important when you have multiple workloads, which makes the security stakes higher.)

Chapter Summary

Ensuring the security and reliability of all of the tenants in a multi-tenant Kubernetes deployment takes some work, but Kubernetes provides several features to help you do it. By leveraging tools like namespaces, Network Policies, and Resource Quotas, you can achieve the right level of isolation and resource sharing whether your Kubernetes deployment hosts a handful of applications or several thousand.

7 Simple Kubernetes Performance Optimization Tips

Kubernetes is a complex tool. As is the case with most complex tools, getting optimal performance out of Kubernetes can be tricky. Most Kubernetes distributions don't come fine-tuned to maximize performance (and even if they did, it's likely that they would not be tweaked in a way that is optimal for your environment).

With that reality in mind, keep reading for tips on Kubernetes performance optimization. We'll focus on simple things you can do to improve Kubernetes performance whether you are just starting out with building your cluster, or already have a production environment up and running.

Add resources to existing worker nodes before you create new worker nodes

Probably the most obvious way to improve Kubernetes performance is to add more worker nodes to your cluster. The more workers you have, the more resources are available to power your workloads. You also get an availability boost, because having more nodes reduces the chances of having so many of them fail that your workloads start failing.

If you want to get the very most out of your worker nodes, however, you'll get more bang for your buck by adding memory and CPU resources to existing worker nodes rather than creating new nodes. In other words, you're better off having 20 nodes with 16 gigabytes of memory each, rather than 40 nodes with 8 gigabytes each.

This is true for two reasons. First, there is a certain amount of overhead for each node due to the host operating system. Having fewer nodes means fewer resources wasted in that way. Second, the more nodes you have, the harder the scheduler, kube-proxy and other components have to work to keep track of everything.

Obviously, you need to take availability into account, and ensure that you have a minimum number of worker nodes to meet your availability goals. But once you cross that threshold, you'll achieve better overall performance efficiency by making sure that each node has as many resources allocated to it as you can afford, rather than trying to maximize the number of overall nodes. Just don't go to extremes (you probably don't want 24 terabytes of memory on a single node, for instance) because you risk losing those resources if a node fails.

Of course, you may or may not have a lot of flexibility in determining the resource allocations of your nodes. If they're virtual machines running in the cloud, you can assign as many resources as you want. If they're on-premise virtual machines or (gasp) physical servers, this is trickier.

Use multiple master nodes

> *The main reason to use multiple masters in a Kubernetes cluster is to achieve high availability. The more masters you have, the less likely it is that they will all fail and bring your cluster down with them.*

However, adding more masters also offers performance benefits because it provides more hosting resources to the essential Kubernetes components (like the scheduler, API server and Etcd) that are hosted on masters. Kubernetes will use the collective resources of all master nodes to power these components.

Thus, adding a master (or two or four) is a good and easy way to give your Kubernetes cluster a performance boost.

Set worker node scoring limits

Part of the job that the Kubernetes scheduler does is to "score" the worker nodes, which means it determines which worker nodes are fit to handle a workload. In clusters with several dozen or more worker nodes, the scheduler can end up wasting time checking every worker node.

To avoid this inefficiency, you can set the *percentageOfNodesToScore* parameter to a percentage lower than 100. The scheduler will then check only the percentage of nodes that you specify.

Set resource quotas

A simple but very effective way to improve Kubernetes performance, especially in large clusters shared by multiple teams, is to set resource quotas. Resource quotas set limits on the amount of CPU, memory and storage resources that can be used by a given namespace.

Thus, if you divide your cluster into namespaces, give each team a different namespace and set a resource quota for each namespace, you'll help ensure that all workloads get a fair share of resources.

> *Resource quotas aren't a performance optimization per se; they are more of a solution to the noisy-neighbor issue. But they do help ensure that each namespace has the resources it needs to perform its job adequately.*

Set limit ranges

What if you want to limit the resources consumed by a workload, but that workload runs in the same namespace as other workloads? This is what limit ranges do.

Whereas resource quotas set limits on how many resources each namespace can consume, limit ranges do the same thing on a per-pod or per-container basis.

> *For the sake of simplicity, a best practice in most cases is to segment workloads using namespaces and resource quotas. But if you are in a situation where that approach is not practical, limit ranges will allow you to guarantee that individual pods or containers have the resources they need to perform as desired.*

Set up Endpoint Slices

Endpoint Slices are a little-discussed Kubernetes feature that lets you group network endpoints together based on service and port combinations. When they are set up, kube-proxy refers to them when deciding how to route traffic.

In environments with large numbers of endpoints, Endpoint Slices can improve performance by reducing the amount of work that kube-proxy needs to perform in order to route traffic within the cluster.

Use a minimalist host OS

Last but not least is a basic but effective tip: Make sure that the operating system hosting your Kubernetes clusters

is as minimal as possible. Any extra components that aren't strictly necessary for running Kubernetes lead to wasted resources, which in turn degrades the performance of your cluster.

Depending on which Kubernetes distribution you use, you may or may not have the luxury of choosing the host OS. But if you do, choose a Linux distribution with a minimal footprint.

Chapter Summary

Kubernetes is designed to do many things automatically. But it doesn't automate its own performance management. Getting the most performance out of the infrastructure that you dedicate to Kubernetes requires being smart about how you design the infrastructure and how you configure certain Kubernetes components.

GitOps for Kubernetes

When you deploy Kubernetes at scale, managing all of the configuration code and application code that powers your environment is challenging. So is applying and tracking changes in an automated, streamlined way.

That's why many organizations have turned to GitOps to help manage their clusters. Although the GitOps methodology is not unique to Kubernetes (it can be used with virtually any type of platform), it offers critical benefits when dealing with the large-scale, fast-moving environments that are common in the world of Kubernetes.

What is GitOps

GitOps, in layman's terms, can probably be described as DevOps powered by Git, with a noticeable shift in favor of developers.

Think of **Git version control** as a giant indexed folder that magically absorbs all your code into neat laminated sections with colorful post-its sticking out, displaying full audit trails.

> *In the **GitOps model,** Git Version is the only **source of truth**. Anything that happens needs to happen on Git: where it is automatically versioned and can be collaborated on, deployed, rolled-back, and more.*

GitOps involves using version control for **everything from development to deployment.** It's not just your raw application code that is versioned and stored in Git, but also your tests, infrastructure configuration, images, deployment processes, release pipelines and everything else related to the application.

Git becomes the shared, single place where everything exists. The configuration and deployment processes creating and managing your environments are a by-product of code changes, and this code resides right next to your application code and is handled in the same way.

Having everything treated as code and triggered from Git makes life a lot easier for developers and empowers them to perform tasks traditionally associated with operations teams.

Unlike the traditional approach where there is a distinction between Dev and Ops teams (those who develop the code and check-in their code versions on Git, and Ops team who then need to operate the applications in production environments) GitOps lets developers both "build it" and "run it", all while focusing on the code and their familiar processes and repositories.

GitOps in Kubernetes

GitOps in Kubernetes revolves around one key concept that's important to understand -that

> ***Kubernetes is a declarative system.*** *And more than any CI/CD tool that you might have heard of (Jenkins, etc.), Kubernetes and its declarative properties are what are now making GitOps possible for prime-time, and at scale.*

Being a declarative system means that Kubernetes has a final picture of what the ideal cluster configuration needs to look like, and is always working to match that with the current situation. It continually monitors and compares the current status of the system against its desired specified state, and automatically works to ensure that the system is managed to maintain its optimal requested configuration.

> *It's this declarative property of Kubernetes that makes it possible for all configuration files to be version-controlled by Git - as developers specify the "end state" of the system they want to see.*

Standardizing on Git processes makes the entire system a lot easier to manage, and if need be, recover/rolled back to previous versions. This includes both applications and infrastructure - since they're both essentially code and can be versioned with IDs and timestamps and audited for changes.

Best Practices for GitOps in Kubernetes:

Use Kubernetes Operators where possible

Best practice dictates using existing Operators whenever appropriate as opposed to ad-hoc, manual scripting. This is largely because managing stateful applications or other complex services in Kubernetes is quite challenging and requires a lot of knowledge that comes from experience.

Kubernetes Operators, which use Custom Resource Definitions (CRDs), are a lot more efficient than in-house solutions. Re-using vetted operators developed by the community or trusted vendors ensures that you standardize on the best tools for the job and benefit from the contributor's experience in the specific service you're deploying. For example, using an existing Operator to deploy a relational database on your Kubernetes cluster may be a better approach then re-scripting the entire configuration and deployment process yourself, from scratch.

Branches over repositories

While there are advantages to having a multiple-repository setup, (clear ownership, etc.) this model gets unnecessarily complicated as you scale up. Additionally, since each team essentially has separate repositories for development, staging, and production, it is hard to get a good overview of the system as a whole. When each team is only concerned with a single service, people with knowledge of the whole platform become a scarcity.

Reducing the number of repositories benefits developers considerably, as it not only saves time but also makes the tasks of sharing code, debugging and giving feedback a lot more straightforward.

> *Best practice with regard to GitOps is to have one repository per team; and where there's a need for additional repositories, resort to branches instead. Of course, there's no one-size-fits-all when it comes to the number of repositories, but it helps to give each team its own*

repository so that development is not restricted. At the same time, use branches to ensure you don't end up with repository sprawl.

Pull over Push

Gitops is all about using pull requests to publish new code. A pull request is basically like an application you submit in order to have your changes advanced through the workflow, and is typically accompanied by a lot of supporting information. This additional information pertaining to why the changes were made, and by whom, makes the process of debugging and troubleshooting a lot easier.

A best practice is to avoid pushing anything directly to masters, unless absolutely necessary and standardize on Pull requests for the above reasons. In addition, push requests unnecessarily exposes your cluster credentials over the network, while pull requests cause Kubernetes Operators to deploy new images from inside the cluster. The Operator also watches for changes to configuration parameters and begins "convergence" once it finds one.

Automate, automate, automate

When it comes to GitOps and tasks that need to be repeated "exactly" the same way, it's a best practice to keep the "human element" out of the picture. A hands-off approach here is critical, and every single action needs to be performed by a script that has already been tested for bugs. This is accomplished by Git webhooks, which fire in response to application changes that have passed the screening process.

Observability + GitOps

By monitoring the system and the state of your clusters and applications for any anomalies, and tying this data to trigger processes from Git - such as in the case of a failure, bottleneck or other incidents - you can automate response to Production issues and accelerate them to recovery, as well as auto-rollback if needed.

Chapter Summary

GitOps creates a bridge between your repositories and your actual environment, allowing you to use version control to not only manage your application code, but your environment and processes as well. GitOps allows developers to also manage the operations of their code, and essentially be self-sufficient and "operation agnostic."

Using Prometheus and Cortex for Kubernetes Monitoring at Scale

The growing adoption of microservices and distributed applications gave rise to the container revolution. Running on containers necessitated orchestration tooling, like Kubernetes.

But managing the availability, performance, and deployment of containers is not the only challenge. It is important to not only be able to deploy and manage these distributed applications, but also to monitor them. An observability strategy needs to be in place in order to keep track of all the dynamic components in a containerized microservices ecosystem. Such a strategy allows you to see whether your system is operating as expected, and to be alerted when it isn't. You can then drill down for troubleshooting and incident investigation, and view trends over time.

Kubernetes can simplify the management of your containerized applications and services across different cloud services. It can be a double-edged sword, though, as it also adds complexity to your system by introducing a lot of new layers and abstractions, which translates to more components and services that need to be monitored. This makes observability even more critical.

There are a lot of open-source tools that can help you monitor your applications running in your Kubernetes cluster. In this post, we will talk about Prometheus and Cortex, and discuss how to configure them to monitor your Kubernetes applications and services, at scale. Although you can use Prometheus alone, we will discuss the advantages that Cortex brings. We'll also cover how to leverage some of the automated deployment/install methods like Helm.

Kubernetes Monitoring Solutions

Prometheus is an open-source application used for metrics-based monitoring and alerting. It calls out to your application, pulls real-time metrics, compresses and stores them in a time-series database. It offers a powerful data model and a query language and can provide detailed and actionable metrics. Like Kubernetes, the Prometheus project has reached a mature "graduated" stage with CNCF.

Prometheus Limitations and Challenges

While Prometheus is a great solution for your monitoring needs, it is also purposely designed to be simple and versatile. It is meant to store all of its compressed metrics into a single host, in its own time-series database on disk. Prometheus is **not designed for long-term storage** (so you can keep data for a few weeks, but not for months or years), **and the storage layer is not distributed** (all the data is on one machine). Prometheus is great for alerting and short-term trends, but not for more **historical data needs** (i.e. for use cases such as capacity planning or billing – where historical data is important and you typically want to keep data for a long time).

> *Prometheus does not provide **multi-tenancy;** which means that it can scrape many targets, but has no concepts of different users, authorization, or keeping things "separate" between users accessing the metrics. Anyone with access to the query endpoint and web endpoints can*

*see all the data. This is the same for **capacity isolation**. If one user or target sends too many metrics, it may break the Prometheus server for everyone. All of these factors **limit its scalability,** which can make running Prometheus in an enterprise environment challenging.*

Cortex

Cortex is an Open Source project that originated at Weaveworks that seeks to bridge these Prometheus scalability gaps. It is a "reimplementation" of Prometheus in the sense that it reimplements the **storage and querying parts** of Prometheus (but not the scraping parts) in a way that is **horizontally scalable and durable** (safe against data loss resulting from losing a single node). Cortex parallelizes the ingestion and storage of a Prometheus metric, and stores it across a distributed noSQL database. Out of the box, some of the long-term storage options that Cortex supports are: AWS DynamoDB, AWS S3, Apache Cassandra, Google Cloud Bigtable, and others. For users who want to run Cortex on-prem - Cassandra is currently the best choice for installations where a user wants to keep Cortex off of public cloud services. There are also some newer alternatives, and one could also setup a private S3 installation with Minio and use that instead of AWS S3.

A typical Cortex use case is to have a Prometheus instance running on your Kubernetes cluster, which scrapes all of your services and forwards them to a Cortex deployment using the Prometheus remote write API. Cortex then periodically bundles the samples and writes them to a distributed storage. It has multi-tenancy built in, which means that all Prometheus metrics that go through Cortex are associated with a tenant and offers a fully compatible API for making queries in Prometheus. Cortex also offers a multi-tenanted alert management and configuration service for re-implementing Prometheus recording rules and alerts. Instead of being config-file driven like Prometheus, the same config files are persisted via a REST API. You could use Grafana (or other similar tools) alerting capabilities on top of Cortex, as part of your stack.

Deploying Prometheus and Cortex on Kubernetes

Let's deploy these monitoring tools onto Kubernetes and see it in action. When using Cortex, you still need a 'normal' Prometheus setup in order to scrape metrics and forward data to it.

Assuming that you have a Kubernetes cluster up and running with kubectl set up, we can use helm to make our lives easier in installing Prometheus components onto our cluster. Helm is a package manager for Kubernetes that collects sets of templatized Kubernetes manifest files into a unified bundle called a chart, which is organized and versioned. Once the Helm client deploys the chart into the Kubernetes cluster, it can manage the upgrade and rollback of these bundles.

Installing Helm
First install Helm, and then install Tiller (Helm's server-side component) onto our cluster:

```
$ helm init
```

Output:

```
$HELM_HOME has been configured at /home/daisy/.helm.

Tiller (the Helm server-side component) has been installed into your Kubernetes Cluster.

Please note: by default, Tiller is deployed with an insecure 'allow unauthenticated users' policy.
To prevent this, run `helm init` with the --tiller-tls-verify flag.
For more information on securing your installation see: https://docs.helm.sh/using_helm/#securing-your-helm-installation
Happy Helming!
```

Check that Tiller has been installed by typing:

```
$ helm version
```

Output:
```
Client: &version.Version{SemVer:"v2.12.3", GitCommit:"eecf22f77df5f65c823aacd2dbd30ae6c65f186e", GitTreeState:"clean"}
Error: could not find a ready tiller pod
```

If you see this output, this is because the Helm permission model requires us to adjust its default setup so that it can work with the permission structure in Kubernetes, which has RBAC turned on by default. As a result, the service account for each namespace does not have cluster-wide access out-of-the-box, so Tiller (running as a pod in the system-wide `kube-system` namespace) does not have permission to install and manage cluster resources.

To work around this issue, we can elevate permissions for the default service account in the `kube-system` namespace to be cluster-wide:

```
$ kubectl create clusterrolebinding add-on-cluster-admin --clusterrole=cluster-admin --serviceaccount=kube-system:default
```

Note: This is fine for exploring Helm, but you will want a more secure configuration for a production-grade Kubernetes cluster. Consult the official Helm RBAC documentation for more information on securing a Tiller installation.

Type “helm version” again and we should see:
```
Client: &version.Version{SemVer:"v2.12.3", GitCommit:"eecf22f77df5f65c823aacd2dbd30ae6c65f186e", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.12.3", GitCommit:"eecf22f77df5f65c823aacd2dbd30ae6c65f186e", GitTreeState:"clean"}
```

Deploying Prometheus
Now we need to deploy Prometheus onto our cluster. This entails running the Prometheus server in a pod (by using a deployment to ensure that it keeps running), exposing the Prometheus server web UI (by using a NodePort), running the node exporter on each node (using Daemon sets), setting up a service account so that Prometheus can query the Kubernetes API, and configuring the Prometheus server (and storing the configuration in a Config Map).

Helm will take care of all of these steps for us! In your terminal, type:

```
$ helm upgrade prometheus stable/prometheus \
        --install \
        --namespace monitoring \
        --set server.service.type=NodePort \
        --set server.service.nodePort=30090 \
        --set server.persistentVolume.enabled=false
```

This will upgrade the “prometheus” release to the latest version from the stable/prometheus Helm chart, and install it if it doesn’t exist. We are also putting the Prometheus components into a separate monitoring namespace and specifying some values to be set when rendering the charts. In particular, **we are exposing the server with a NodePort.** A NodePort is a service type that is made available on all of our nodes and can be connected to externally. By default, the allocated ports are in the range 30000-32768.

Check the status of your pods by typing:

```
$ kubectl get pods -n monitoring
```

Output:

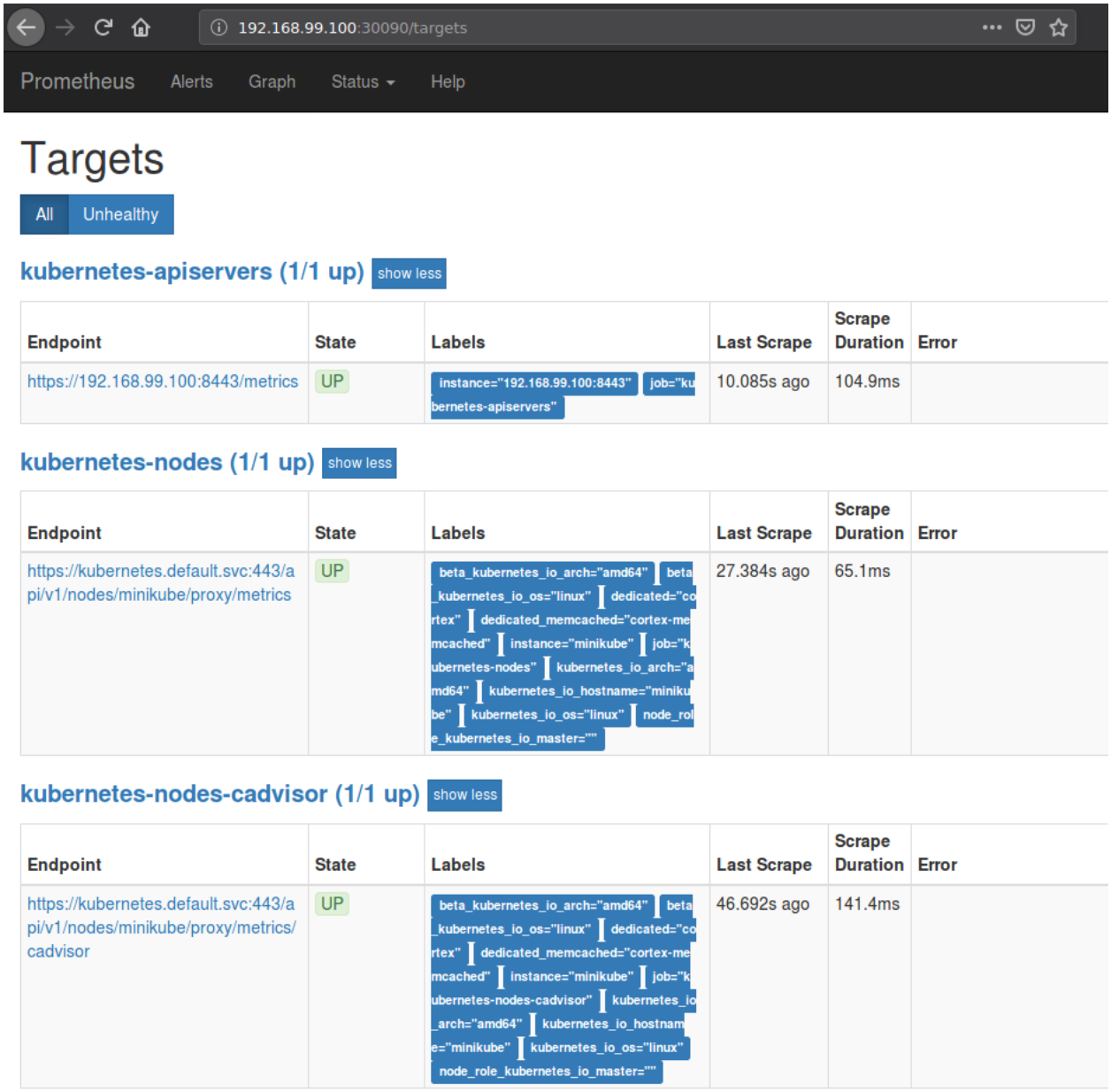
```
NAME                                            READY   STATUS  RESTARTS   AGE
prometheus-alertmanager-5df865b5d7-m6c25        2/2     Running 0          20m
prometheus-kube-state-metrics-88648b885-ppz6c   1/1     Running 0          20m
prometheus-node-exporter-zscn5                  1/1     Running 0          20m
prometheus-pushgateway-58fb574d-tcfqb           1/1     Running 0          20m
prometheus-server-8bb84ddc-snjzp                2/2     Running 0          20m
```

Let's get the details for the NodePort that has been allocated to the Prometheus server so that we can connect to the Prometheus web UI:

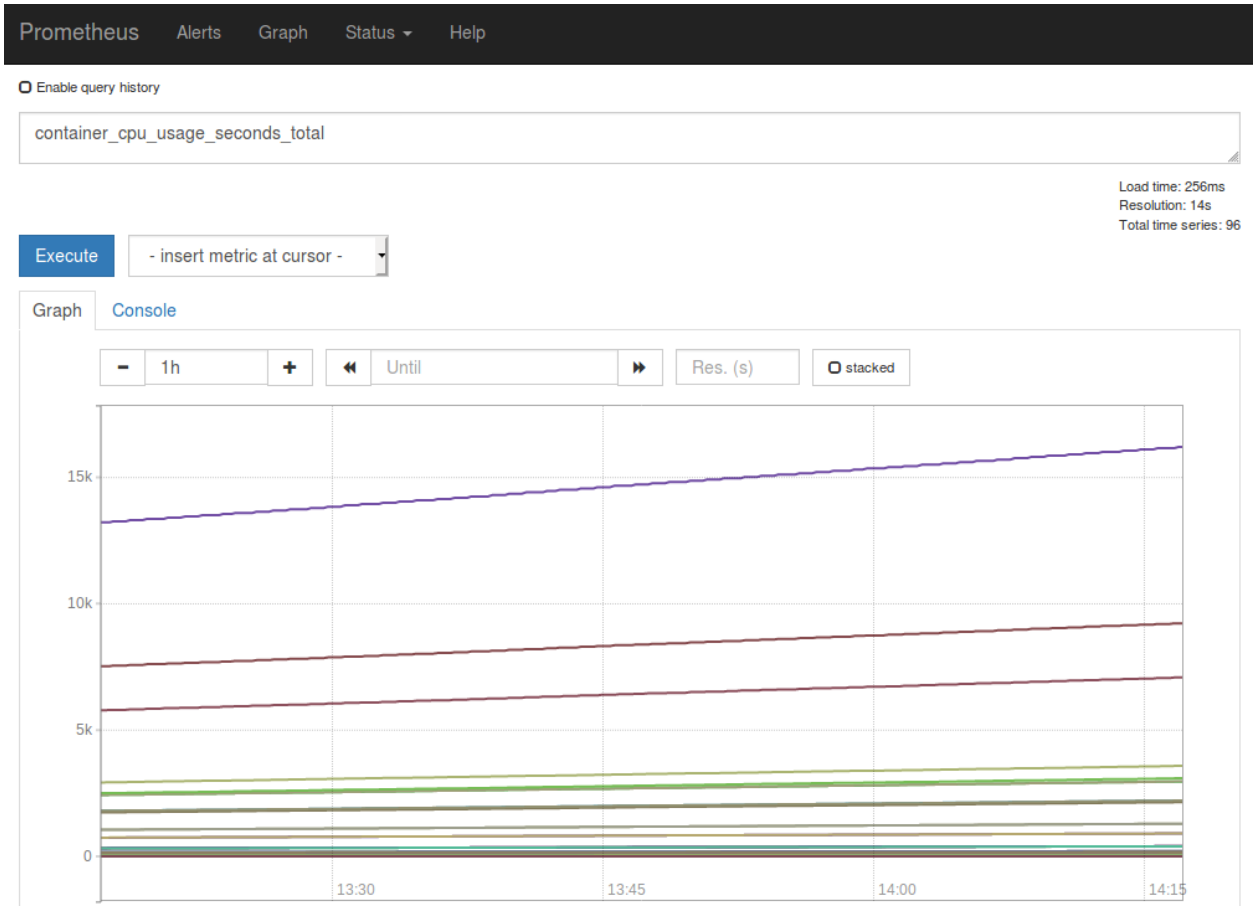
```
$ kubectl get services --all-namespaces | grep prometheus-server
```

```
monitoring      prometheus-server      NodePort    10.104.39.225    <none>   80:30090/TCP    47m
```

Using the IP address of your running cluster, connect to the mapped port specified with your browser:



Now you can query for some metrics with PromQL. Click on the “Graph” tab and in the Expression input box, type “container_cpu_usage_seconds_total” to view CPU usage across all containers.



We will go over other metrics we should pay attention to later on.

Deploying Cortex

Let's move onto installing Cortex onto our cluster. Cortex has conveniently provided a Helm chart that can be found here. Download the chart, configure it to your liking, package it up with "helm package", and install it like this:

```
$ helm install --name cortex --namespace cortex <path-to-cortex-helm-chart>
```

If you would like to override some default values:

```
$ helm install --name cortex --namespace cortex -f my-cortex-values.yaml <path-to-cortex-helm-chart>
```

Make sure that you have **sufficient CPU and memory** resources on your cluster before installing this chart since there are varying resource requirements for the different components. In addition, the default chart settings specify that Cortex pods have node affinity for nodes with the label "dedicated=cortex", and memcached pods have affinity for nodes marked with "dedicated=cortex-memcached".

To label your nodes following this requirement, type:

```
$ kubectl label nodes <name-of-node> dedicated=cortex
$ kubectl label nodes <name-of-node> dedicated=cortex-memcached
```

There are also a few manual steps required in order to install the chart, specified in more detail here. You will need to create a secret "cortex-aws-creds" with credentials to your cloud storage account:

```
$ kubectl -n {{ .Release.Namespace }} create secret generic cortex-aws-creds \
  --from-literal=USER_NAME=<AWS_ACCESS_KEY_ID> \
  --from-literal=USER_PASSWORD=<URL Encoded AWS_SECRET_ACCESS_KEY> \
  --from-literal=AWS_DEFAULT_REGION=<AWS_DEFAULT_REGION>
```

To verify that Cortex is installed correctly, port forward to an nginx Pod like this:

```
$ kubectl -n {{ .Release.Namespace }} port-forward $(kubectl -n {{ .Release.Namespace }} get pods -l "cortex=nginx" -o jsonpath='{.items[0].metadata.name}') 8080:80
```

You should now be able to visit the Prometheus API, which should show a bunch of metrics being collected: http://127.0.0.1:8080/api/prom/api/v1/query?query=up

Key Metrics to Track

Kubernetes introduces several abstractions in its ecosystem, which translates to new considerations when it comes to choosing which metrics to track. They can be separated into two main categories: monitoring the health of the cluster, and monitoring the pods which hold the containers running the services and applications.

There are two types of resources that can be consumed by containers: CPU and RAM. The USE/RED method is a good guideline to follow when tracking metrics in production and states that you should track the utilization, saturation, and error rates for all of your resources. For an excellent explanation of the four golden signals (latency, traffic, errors, saturation), please refer to this chapter of the Google SRE book.

Generally speaking, you should be collecting metrics from your cluster infrastructure, container metrics, and application-specific metrics. Prometheus was built primarily for the third use-case, but the ecosystem around it quickly evolved so it is well suited for all of the above.

There are application-specific metrics that will vary and depend on functionality. These should be custom-instrumented, but will typically export metrics such as number of HTTP requests and its latency and error rate, number of outgoing connections, number of threads, etc.

Chapter Summary

Kubernetes introduces a lot of new layers that need to be taken into account when crafting out an observability strategy. While Prometheus is a great monitoring tool to use with Kubernetes, it was purposely designed to be simple and not built for long term storage out of the box. When it comes to creating a scalable and durable monitoring solution, it is useful to turn to tools like Cortex (which extends Prometheus) and leverage its distributed long term storage features and built-in multi-tenancy.

Kubernetes Networking Challenges at Scale

Kubernetes networking can be noisy, tedious, and complex. This chapter discusses some of the challenges involved with managing and troubleshooting Kubernetes networking for large-scale production deployments. In fact, in a recent survey, 42% of Kubernetes users list networking as their largest Kubernetes challenge. This number often goes up as deployment size increases. Additionally, according to Gartner, more than 75% of global organizations will be running containerized applications in production by 2022.

Even if you don't think you have a large-scale deployment now, odds are that your deployment will quickly grow. Without consideration early on, you may build technical debt that gets harder to pay off over time. Let's explore some of the network challenges to consider now.

Container Networking Differences

The first challenge of networking with Kubernetes is that it varies enough from traditional networking, yet too many developers try to apply these fundamentals to containers as well. For starters, network addressing is more dynamic with Kubernetes environments – nothing is static – and due to the nature of container migration, classic DHCP can slow things down. You cannot rely on hard-coded addresses or even ports, which makes network security more challenging as well. Additionally, dynamic changes to the networking layout of your Kubernetes deployments aren't persisted since containers are immutable, and can migrate across environments over time. Finally, for large-scale deployments in particular, you will likely have many more addresses to assign and manage than with traditional deployments. Many enterprises run up against the IPv4 limitations for hosts within a subnet, resulting in the deployment of more subnets, or a move to IPv6.

Kubernetes Network Security

Nearly every technology-related list of challenges, regardless of the topic, begins with security (or at least it should). Network security has long been an area of focus, and with Kubernetes it remains so. In many ways, the ease of deployment of container-based applications can make it easy to overlook network security, or deploy applications that have flaws in the design of their Kubernetes managed containers. Fortunately, you can define Kubernetes network policies to help here, but these introduce some of their own challenges (discussed below). However, there are managed solutions, platforms, and cloud-native services available to help with Kubernetes deployment.

Networking Misconceptions

Containers and Kubernetes power today's cloud-native application deployments so well, it makes it easy to go from development to production with little friction. This helps power an efficient DevOps practice, getting features to users quickly. However, due to misconceptions or just lack of understanding of key areas such as networking, with Kubernetes you risk deploying without consideration to proper network design. Sometimes this may be due to convenience, where developers focus on open communication in order to get their job done. Areas of consideration often overlooked involve defining specific Kubernetes network constraints around container-to-container communication, pod-to-pod networking, service communication, and Internet connectivity.

Defining and Changing Network Policies

Kubernetes network policies define how groups of pods communicate with one another. Defining and then changing the network policy for a pod or container requires you to create NetworkPolicy resources, which amounts to configuration files. For large numbers of containers and pods, this can be a daunting task. Further, changing all of these policies can be tedious and error prone. Apstra AOS is an intent-based networking system that creates a software-defined layer to allow you to easily change the policy based on application requirements,

all via a REST API. It also helps to abstract network vendor specific implementations and APIs.

Project Calico is designed to help manage security with Kubernetes networking, while delivering high performance. Its usage is growing in the enterprise due to its layer-3 based scalability, its fine grained segmentation when it comes to role-based network security and policies, and its integration with the Linux kernel for maximum performance.

Lack of Abstraction

According to Network Computing, one of the challenges holding back enterprises from adopting Kubernetes as quickly as they should, is its direct effect on data center design and implementation. For example, one strategy to adopt Kubernetes involves restructuring an existing data center (or changing the deployment plans for new ones) to suit the Kubernetes and container-based application architectures specifically, or buying into other interconnectivity solutions altogether.

As an alternative, consider using a software-defined framework to turn your data center into something similar to a cloud-native environment to support both container and non-container based applications. These solutions help you define load-balancing, security policies, container traffic management, and other network challenges for your Kubernetes deployment through REST APIs. It also helps make your Kubernetes-based applications more portable across data centers and cloud providers, providing an automation layer to integrate with your DevOps practice.

As the usage of Linux containers grows rapidly, enterprise networking is still not well defined. Because of this, container applications, runtimes and orchestrators each attempt to address this problem. The goal of the Container Network Interface (CNI) specification is to address this and avoid and define a common interface between network plugins and containers themselves. The CNI describes how to write plugins to configure network interfaces for Linux containers. CNI focuses only on network connectivity of containers, removing allocated resources when the container is deleted. As a result, CNI is straightforward to implement, and has a wide range of support. In fact, CNI is supported by Project Calico, discussed above.

Battling Complexity

Complexity is one area that containers and Kubernetes were created to help eliminate. However, container-based *networking* complexity can still be an issue, especially at scale.

Tools such as linkerd offer service mesh solutions for container-based and cloud native applications, with agents that provide features such as service discovery, routing, failure handling, and visibility to applications transparently, without requiring code changes. A service mesh is a newer technology that helps network routing and resiliency between services, security, and observability, inspired by microservice-driven application architecture.

Network Communication Reliability

The advantages of microservice and container-based application design are well known. It simplifies development, testing, management of requirements and development team responsibilities, and eases deployment headaches. However, service-to-service communication becomes more critical in this type of architecture. As the number of services (and containers and Kubernetes pods) increases, the complexity of service communication increases, and so does the importance of reliable communications. You need to harden the Kubernetes network policies (and other configurations) to ensure reliable communications, and once you've done that, you need to properly monitor and manage it as well. This last point alone is vital to ensuring optimal service communication performance and reliability.

Combining Virtual Machine Networking

Even after you rise to the challenges of Kubernetes networking, you're often faced with the need to combine the management and monitoring of VM-based deployments with container and Kubernetes-based deployments. The industry is just now beginning to offer solutions to help manage both types of deployments, together. But there

are enough differences between the VMs and containers that it's not quite transparent yet.

KubeVirt, however, offers an API to address the needs of development teams that have adopted Kubernetes for containers that also run with a mix of virtual machines. It provides a unified development platform where developers can build and deploy applications in both Application Containers and Virtual Machines in a common, shared environment. KubeVirt helps organizations with existing virtual machine-based workloads to rapidly containerize applications.

Debugging Network Connectivity Issues

To judge development talent, some people use what is known as the "golf rule". Every golfer wants to hit the ball straight and far every time, but the golfers who know how to get out of trouble (sand traps, the rough, long putts) are the most successful. This applies to technology as well: being able to debug issues effectively defines success. The same goes for Kubernetes network issues, especially across managed containers and containers deployed on geographically disparate clouds. Often, having the right tools can help, and leveraging the right Kubernetes monitoring and management platforms will make you a more effective problem solver.

Chapter Summary: Avoid Network Hell

Containers come and go, they're immutable, and you just can't nail them down, especially when you're running Kubernetes. The networking challenges with Kubernetes at scale can seem overwhelming, but they're manageable with the right planning and tools. From service meshes, new network infrastructure products from Cisco, frameworks and APIs that help drive a software-defined approach, and forthcoming support from heavyweights such as VMware, the future is bright for Kubernetes.

10 Kubernetes Performance Tips

Kubernetes is a scalable and performant engine that orchestrates containers in a server environment. It is highly optimized by default, and it scales nicely in a suitable infrastructure.

However, Kubernetes is also not a very "opinionated" platform by default. That means that there are plenty of customizations for end users to define. This flexibility allows Kubernetes to cover many different use cases and penetrate the market faster, making it extremely popular.

Since server costs can increase quickly, you have to find ways to reduce your bills and get the most of what you have. In this chapter, we will give you ten tips for squeezing every bit of efficiency and performance out of your Kubernetes distribution.

Best tips for Kubernetes Performance

1. **Define Deployment Resources**:
   Kubernetes orchestrates containers at scale, and the heart of this mechanism is the efficient scheduling of pods into nodes. You can actually help the Kubernetes scheduler do this by specifying resource constraints.

   In other words, you can define **request** and **limit** resources such as CPU, memory, or Linux HugePages. For example, let's say you have a Java microservice that acts as an email sender service, so it primarily handles network requests. You can assign the following resource profile:

```
resources:
   requests:
     memory: 1Gi
     cpu: 250m
   limits:
     memory: 2.5Gi
     cpu: 750m
```

   If you were to implement the same application in Go, you would use a different set of resource constraints (mostly memory-related):

```
resources:
   requests:
     memory: 128m
```

```
      cpu: 250m
    limits:
      memory: 1Gi
      cpu: 750m
```

So what do we mean by CPU and memory? That depends, since there are different kinds of CPUs and different kinds of memory chips which vary from provider to provider. Xeon CPUs are better suited for server environments, for example, while DDR4 chips have higher read/write throughput. You can also choose to use either CPU or memory-intensive nodes for specific deployments so that Kubernetes can schedule them appropriately.

Ultimately, when you clearly define the resource requirements in the deployment descriptor, you make it easier for the scheduler to ensure that each resource is allocated to the best available node, which will improve runtime performance.

2. Deploy Clusters closer to customers:

The geographic location of the cluster nodes that Kubernetes manages is closely related to the latency that clients experience. For example, nodes that host pods located in Europe will have faster DNS resolve times and lower latencies for customers in that region.

Before spawning Kubernetes clusters here and there, however, you need to devise a careful plan for handling multi-zone Kubernetes clusters. Each provider has limitations on which zones can be used to provide the best failure tolerations. For example, AKS can use this list of zones, while GKE offers options for multi-zonal or regional-clusters, each with its own list of pros and cons.

It's important to start locally and then expand if you observe issues with the current traffic or if you offer priority routing for some services. This will give you more time to figure out what the major bottlenecks are when serving content to customers.

3. Choose better Persistent Storage and Quality of Service:

Just as there are different kinds of CPUs and memory chips, there are different kinds of persistent storage hardware. For example, SSDs offer better read/write performance than HDDs, and NVMe SSDs are even better under heavy workloads.

There are many different types of persistent volumes available to Kubernetes, which is very extensible and less opinionated about storage.

Some Kubernetes providers extend the persistent volume claims schema definitions with Quality of Service levels. That means they prioritize volume read/write quotas for specific deployments, offering better throughput under certain circumstances.

In general, everyone agrees that better hardware offers better performance, but it's important to realize that, in most cases, the order of performance is by a constant factor **c.** So if you were to upgrade from SSD

to NVMe, you would expect to see a percentage increase in read/write operations, but you would not expect to see any difference in network latency.

4. Configure Node Affinities:

Not all nodes run on the same hardware, and not all pods need to run CPU-intensive applications. Specializations for nodes and pods are available to Kubernetes via the Node Affinity and Pod Affinity.

When you have nodes that are suitable for CPU-intensive operations, you want to pair them with CPU-intensive applications to maximize effort. To do that, you would use the **nodeSelector** with the specified label. For example, let's say that you have two nodes: one with **CPUType=HIGHCORE** that offers high CPU core count and frequency, and another with **MemoryType=HIGHMEMORY** that offers the highest and fastest memory available.

The simplest way to do this is to issue a POD deployment to the **HIGHCORE** by adding the following selector in the **spec** section:

```
…
nodeSelector:
        CPUType: HIGHCORE
```

A more expressive yet specific way to do this is to use the **nodeAffinity** of the **affinity** field in the spec section. There are currently two choices:

- **requiredDuringSchedulingIgnoredDuringExecution**: This is a hard preference, and the scheduler will deploy the pods only to specific nodes (and nowhere else).
- **preferredDuringSchedulingIgnoredDuringExecution:** This is a soft preference, which means that the scheduler will try to deploy to the specified nodes, and if that's not feasible, it will try to schedule to the next available node.

You can use specific syndax for controlling the node label, such as **`In, NotIn, Exists, DoesNotExist, Gt, or Lt.`** Bear in mind, however, that complex predicates within big lists of labels will slow down the decision-making process in critical scenarios. In other words, keep it simple.

5. Configure Pod Affinity:

As we mentioned earlier, Kubernetes allows you to configure the Pod affinity in terms of existing running pods. Simply speaking, you can allow certain pods to run along other pods in the same zone or cluster of nodes so that they communicate with each other more frequently.

The available fields under the **podAffinity** of the **affinity** field in the spec section are the same as the ones for **nodeAffinity**: **requiredDuringSchedulingIgnoredDuringExecution** and **preferredDuringSchedulingIgnoredDuringExecution.** The sole difference is that the **matchExpressions** will assign pods to a node that already has a running pod with that label.

Kubernetes also offers a **podAntiAffinity** field that does the opposite of the above: it will not schedule a pod into a node that contains specific pods.

In either case, the same advice applies to **nodeAffinity** expressions: try to keep the rules simple and logical, with fewer labels to match (not thousands). It's remarkably easy to produce a non-matching rule that will create more work for the scheduler down the line and thus diminish the overall performance.

6. Configure Taints:

The ability to customize Pod scheduling options does not end there. When you have thousands of pods, labels, or nodes, it's sometimes very hard not to allow certain pods to land on certain nodes.

This is where taints come in. Taints denote rules for not allowing things to happen, such as not allowing a specific set of nodes to be scheduled to particular places for various reasons. To apply a taint to a specific node, you have to use the **taint** option in the kubectl. You need to specify a key and a value part followed by a taint effect like **NoSchedule** or **NoExecute**:

```
$ kubectl taint nodes backup-node-1=backups-only:NoSchedule
```

Interestingly, you can also override this behavior using tolerations. This comes in handy when you have a node that you tainted, and so nothing was scheduled. Now, you want just the backup jobs to be scheduled there. So how do you schedule them? You need to enable only pods that have a toleration to be scheduled there.

In this example, you would add the following fields in the Pod Spec:

```
spec:
   tolerations:
     - key: "backup-node-1"
        operator: "Equal"
        value: "backups-only"
        effect: "NoSchedule"
```

Since this matches the tainted node, any pod with that spec will be able to be deployed in the **backup-node-1.**

Taints and tolerations give Kubernetes admins the highest level of control, but they require some manual setup before you can make use of them.

7. Build optimized images:

Naturally, it's best to use container optimized images so that Kubernetes can pull them faster and run them more efficiently.

What we mean by optimized is that they:

- Only contain one application or do one thing.
- Have small images, since big images are not so portable over the network.
- Have endpoints for health and readiness checks so that Kubernetes can take action in case of downtimes.
- Use a container-friendly OS (like Alpine or CoreOS) so that they are more resistant to misconfigurations.
- Use multistage builds so that you deploy only the compiled application and not the dev sources that come with it.

Lots of tools and services let you scan and optimize images on the fly. It's important to keep them up-to-date and security-assessed at all times.

8. Configure Pod Priorities:

Just because you have configured Node and Pod affinities does not mean that all of the Pods should be scheduled with the same priority. For example, you may have some pods that need to be deployed before others for various reasons.

In that case, Kubernetes offers the option to assign priorities to Pods. First you need to create a pod, for example **PriorityClass**:

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 9999
globalDefault: false
description: "This priority class should be used for smoke testing
environments before any other pod gets deployed."
```

You can create as many priority classes as you want, although it's recommended that you have just a few levels (low, medium, and high, for example). The higher value number indicates higher priority order. You can add a **priorityClassName** under the Pod spec:

```
priorityClassName: high-priority
```

That way, you can customize in order to efficiently run some deployments as necessary.

9. Configure Kubernetes Features:

Kubernetes runs a feature gate framework that allows administrators to enable or disable features for their environments.

Some of these features could be beneficial for scaling performance, so they are worth investigating:

- **CPUManager**: offers basic core affinity features and constrains workloads to specific CPUs.
- **Accelerators:** enables GPU support.
- **PodOverhead:** handles Pod overhead.

Among other considerations, it's equally important to manage memory overcommitment so that it doesn't panic and so that processes are killed based on priority:

For example, it's worth verifying the following system settings:

```
vm.overcommit_memory=1
vm.panic_on_oom=0
```

In addition, kubelet config has a setting that controls the **pods-per-core,** which is the number of pods that you run on the node based on the number of cores that node has available. For example, for **pods-per-core=10** and a 4-core node, you can have a maximum of 40 pods per node.

Many optimizations could affect the maximum cluster limit for the best performance (typically latency under 1s) and the maximum number of pods per cluster, though this may not be feasible to verify in practice.

10. Optimize Etcd Cluster:

Etcd is the brain of Kubernetes, so it's important to keep it healthy and optimized. Ideally, etcd clusters

should be co-located together with the **kube-apiserver** in order to minimize latency between them. If co-location is not possible, then you should have an optimized routing path with high network throughput between them.

Be careful, though, as scaling up etcd clusters leads to higher availability at the expense of higher performance, so do not overcommit to extra etcd members if it's not necessary.

Chapter Summary

For what it's worth, Kubernetes is very efficient and performant out of the box. The more effort you put into properly configuring your deployments to match your current needs, the more performance you will get in the long run.

You can also make your job easier by utilizing a Managed Kubernetes service like Platform9, which is already optimized, scales well, and offers solutions for every workload. If you want to try out their platform, you can test their sandbox environment here.

Kubernetes Infrastructure Cost Optimization

Kubernetes costs can vary considerably in the enterprise. Depending on whether you decide to host your cluster on Google Kubernetes Engine (GKE), Azure Kubernetes Service (AKS), Amazon Elastic Kubernetes Services (EKS) or on-premise, there are a number of ways to ensure you are spending your money efficiently. While the public cloud experience is all about saving money, a lack of proper management can quickly turn into something quite the opposite. This is mainly because of the significant variation in cost across different providers and the more specific services that they provide.

Similar to how you would pick a mobile phone plan to suit your usage, optimizing how you use resources in the cloud can often help you achieve considerable savings. The flip side of that coin, however, is that just like with a mobile phone, using a service that's not part of your package could result in a rather large bill in the mail.

Cloud cost structures often confuse and intimidate people, especially when terms like "autoscaling" are used.

In this post, we will discuss the major factors that drive up Kubernetes infrastructural costs – like AI, Node type, size, and density – as well as optimizing on-prem resources to meet modern requirements.

Visibility

The public cloud is built for growth; and while cloud providers make it super easy to grow, it's also super easy to grow out of your standard plan and into premium territory. This is why, before you can make any adjustments to how you use Kubernetes resources in the cloud or on-premise, the first step is to gain visibility of your current situation.

Prometheus and Grafana are great tools for this purpose and help you create pretty detailed dashboards through which you can visualize your Kubernetes infrastructure costs.

Resources can generally be classified into three groups: compute, memory and storage. Kubernetes only deals with the first two. While Kubernetes allows us to provision nodes with compute and memory, we still need to make sure they're physically accounted for.

Prometheus is used to first gather metrics using kube-state-metrics, and then analyze those metrics based on namespace, cluster, and pod. They're then displayed through a Grafana dashboard, which is pretty specific and will show you the exact footprint of each part of your cluster – like how much of each resource Istio is using, for example. (Istio is a service mesh that helps manage traffic between microservices with features like load balancing, service-to-service authentication, and monitoring.)

Node size

In production environments in the cloud, Kubernetes nodes exist as instances. Choosing the right size for your master node and worker nodes, in proportion to your cluster, is critical to optimizing cost. You also need to pick the right type of nodes because containers have different requirements. While some may work with general-purpose instances, others might need I/O, memory, or CPU optimized instances. In addition to picking the right size and type, operational hours is another factor that needs to be considered.

Autoscaling is how we tackle that particular problem. While most cloud providers like AWS offer "free" auto-scaling, without proper configuration, you're right back where you started . . . except you have a huge bill from

your cloud provider. To properly configure auto-scaling, you need to be able to make informed decisions on the ideal node size and scaling parameters for your cluster. If your minimum parameters are too high, you risk a big bill. If they're too low, you risk downtime. This is why visibility is key.

Node density

Another critical factor involved in ensuring cost efficiency in Kubernetes is the number of nodes running in a cluster. While this number is typically controlled by the value NUM_NODES in the config.sh file of whatever platform you're using, you can't simply change the value to a very large number. This is because public clouds have limits on the number of resources you can allocate, and you first need to increase your "quota." They do this because containers compete with each other for resources by default and will leave nothing for the system processes that run Kubernetes.

To avoid running into such difficulties while managing node density, a best practice is to first reserve resources for all system daemons, and then set resource limits for all addon containers. It's also a good idea to increase your maximum cloud quota for key parameters like CPU instances, VM instances, In-use IP addresses and firewall rules.

Artificial Intelligence

There are now a number of tools that not only help you gain visibility into your cluster and into how your applications are consuming resources, but they help you manage your applications effectively, as well. This includes tools like Densify that harness machine learning to enable applications to become "self-aware" of their resource needs so they can adjust the corresponding cloud resources accordingly. Similarly, Yotascale uses AI to scan cloud workloads for billing spikes, along with identifying their causes.

Another example is Turbonomic that uses an AI-based decision engine that not only continually monitors instances, but also recommends reserved instances whenever appropriate. CloudSqueeze is a cloud "management" vendor that helps users cut costs by employing deep learning to predict resource demands. While the big cloud providers are yet to provide similar offerings, since such services would obviously be counterproductive, customer demand remains high.

On-prem optimization

Now, as opposed to cloud resources that are rather "elastic" by nature, on-prem nodes are very real and brittle. This is why when we talk about Kubernetes cost optimization on-prem, it's typically with regard to ensuring there is enough juice to meet peak requirements without overdoing it. For high availability on-premise, using redundant instances of all major components is generally considered best practice. These components include the API server, etcd, controller manager and scheduler.

The recoverability of the etcd cluster is a priority with on-premise clusters. A separate five-node etcd cluster is recommended for production environments. A hypervisor is also recommended over the traditional approach of configuring Linux scheduler priorities. This is because it allows for planned resource consumption governance when dealing with single or dual-host deployments, in addition to better workload isolation. With on-premise deployments, in particular, Hypervisors serve to isolate system components and reserve resources, which is critical to protect nodes from exhaustion.

Additionally, disk performance also greatly impacts failed node recovery time, so SSDs are recommended, along with redundant storage and error-correcting memory. With Kubernetes on-premise, it's important to remember that you're not in the cloud and special care has to be taken to avoid overload.

Chapter Summary

With combinations of complex services operating at different scales in different regions with different implications, cloud pricing is anything but straightforward.

The good news is that Kubernetes is a scheduler by nature, and optimizing Kubernetes costs in the cloud or on-premise is all based on how well you know your own requirements. While Grafana and Prometheus are great ways to tackle the problem yourself, cost-calculating open-source tools like Kubecost helps make the process even simpler.

A calculator is only as good as the input it's fed, however; so again, it comes down to how good your information is.

The only thing growing faster than Kubernetes is AI, which is definitely going to be a big part of how we manage our Kubernetes infrastructure in the future.

Calculating the ROI of Your Kubernetes Deployment

Kubernetes is a free and open-source platform. Anyone can download and run it without cost.

Yet there are a variety of costs that could be associated with running Kubernetes. They vary depending on exactly how you design and implement your Kubernetes clusters, but they all play a key role in determining the return you ultimately receive on your investment in Kubernetes.

With that reality in mind, keep reading for an overview of the primary factors to consider when calculating Kubernetes return on investment (ROI). Depending on your Kubernetes architecture and management strategy, not all of these factors may apply to you, but they are certainly worth considering when you are planning how to deploy and run your clusters.

Engineering Staff Costs

If you set up and manage Kubernetes yourself, one of your most significant costs will lie in the engineering staff who implement and maintain your clusters.

The exact cost in this respect will depend on how large your deployment is and how many engineers you need to manage it, of course. To set a baseline, let's imagine you have a small cluster that can be managed by just one engineer. Ideally, that person will be a DevOps engineer, since running Kubernetes successfully requires expertise in both software development and IT operations.

Estimates of the salary of a DevOps engineer range from around $85,000 on average for an entry-level engineer, to $135,000 for a senior DevOps engineer. Add in payroll taxes, benefit costs and any other applicable fully loaded costs, and your total cost to employ that engineer for a year could easily surpass $150,000 – and perhaps run $200,000 or more if you are hiring top talent (which you probably should, because Kubernetes is a complex technology that remains relatively new, and talented engineers who are truly qualified to make the most of it command higher-than-average salaries).

Breaking Down Engineering Costs

What we've presented above is a high-level overview of what a DevOps engineer who implements and maintains a Kubernetes cluster might cost. To provide some more detail and to drill down deeper, here's a breakdown of specific Kubernetes-related setup and maintenance tasks, along with estimates of their cost in "man-months." These estimates reflect how much time you can expect your staff to spend, on average, performing various tasks in a self-implemented and self-hosted Kubernetes cluster.

Key Features for Establishing Production-Ready Kubernetes	Initial Cost (# Man Months)	Recurring Cost (FTE)
Provisioning of k8s Clusters	6	0.75
Monitoring	5	0.45

Upgrading	6	0.5
Security - RBAC	2	0.1
Security - Authn/z	2	0.1
HA and Healing	8	0.5
Load Balancing	4	0.25
Private Registry Support	6	0.15
Persistent File System Storage Support	14	1
Heterogeneous Environment Support	18	2
Federation Support	8	0.5
UI for Multi-Cluster Management	3	0.15
Aggregate	82	6.45

Time-to-Deploy Costs

A third type of cost to consider if you set up Kubernetes with your own staff is that, as reflected in the table above, it will take a long time – several months, in most cases – to get a cluster up and running. That setup time has a large impact on Kubernetes' ROI because you are paying staff during the setup process but don't start reaping the rewards of their work until considerably later.

Infrastructure Costs

Alongside staffing costs, the infrastructure you use to host a Kubernetes cluster is a second major expense that plays a major role in shaping Kubernetes' ROI. Here again, your costs may vary widely depending on the way you implement your Kubernetes clusters. But in general, here are the key costs to consider:

- **On-Premise Servers:** If you use on-premises physical or virtual servers to build your cluster, you must factor hardware acquisition costs into your Kubernetes' ROI calculations. Remember also to factor in electricity and other costs associated with running those servers.
- **Cloud Servers**: If you host Kubernetes on self-provisioned servers in the cloud, the cost of running virtual server instances will be one of your major infrastructure expenses.
- **Networking and Data Transfer**: In the cases of both cloud and (to a lesser extent in most cases) on-premises hosting infrastructure, you need to maintain network connectivity capable of supporting your cluster. In most public cloud environments, you'll also pay data egress fees whenever data leaves the cluster; these fees could be steep depending on how your storage is architected.
- **Storage:** Unless all of your Kubernetes workloads are stateless, you need storage for hosting persistent data. (And even with stateless applications, you may need some persistent storage for information such as log data.) On-premises, that storage will probably take the form of disk drives that you have to purchase, power and maintenance. In the cloud, it will be a cloud storage service, where you pay fees based not just on how much storage you consume, but also data-transfer rates and (in some cases) the data performance level you require.

Managed Kubernetes Fees

Most of the costs described in the preceding two sections are relevant only if you set up and maintain Kubernetes yourself. An alternative model is to use a fully-managed Kubernetes service.

Under that approach, you consume Kubernetes using an SaaS model, with all of the requisite infrastructure, cluster setup and management processes provided for you.

There are fewer costs to factor in when using a managed Kubernetes model. The main ones include:

- Software licensing fees to use the core product. Some managed Kubernetes vendors charge a set fee based on the number of deployments (which is the most predictable cost model); others bill based on the time your cluster and/or individual components of it (such as containers) are running.
- "Upgrade" costs associated with running a larger-than-standard deployment (this cost applies if you select a managed Kubernetes service that charges a fixed fee).
- If applicable, training costs required to teach your staff how to work with the managed Kubernetes service. These costs will typically be substantially lower than those required to set up and manage a cluster in-house, but there is still typically some investment necessary to train your staff in using the managed service to full effect.

Because there are fewer costs associated with fully-managed Kubernetes and less variability within those costs, this Kubernetes deployment model makes it much easier to predict both your Kubernetes startup costs and ongoing maintenance costs. It also offers the benefit of virtually no setup delay; instead of waiting for months before starting to see any return on your Kubernetes investment, you can begin using your clusters from the day you sign up for the managed service.

Chapter Summary

To sum up, there are three main categories of costs associated with Kubernetes: staff costs, infrastructure costs, and software licensing costs. The former two categories mostly apply only to companies that take a DIY approach to Kubernetes; the latter applies if you use a fully-managed Kubernetes service.

Each strategy has its pros and cons, but from a cost perspective, fully-managed Kubernetes certainly makes costs more predictable. In many cases, it may also lead to higher Kubernetes ROI over the long term because the managed Kubernetes vendor can set up and maintain clusters and infrastructure more efficiently (and, therefore, at a lower overall cost) than your in-house engineers are likely to be able to achieve on their own, given that they will have a range of additional responsibilities and, unlike managed Kubernetes vendors, do not exclusively work with Kubernetes.

How Large Should a Kubernetes Cluster Be?

When it comes to Kubernetes clusters, size matters. The number of nodes in your cluster plays an important role in determining the overall availability and performance of your workloads. So does the number of namespaces, in a way.

This does not mean, however, that bigger is always better. A Kubernetes cluster

sizing strategy that aims to maximize node count will not always deliver the best results – certainly not from a cost perspective, and perhaps not from an overall availability or performance perspective, either. And maximizing namespaces is hardly ever a smart strategy.

Instead, calculating the number of nodes to include in a cluster requires careful consideration of a variety of factors. Keep reading for an overview – if not a precise recommendation on how large your cluster should be, because only you can decide that for yourself.

Why Kubernetes Cluster Size Matters

The size of your Kubernetes cluster (in terms of the number of nodes) shapes both performance and availability in critical ways.

With regard to performance, more nodes generally mean better performance. This isn't because node count itself promotes better performance, but because having more nodes usually means there are more resources available for the cluster to consume. So, node count in this sense is a proxy for performance.

As for availability, node count plays a more direct role in shaping this characteristic. The more nodes you have, the smaller the chance that you'll experience a node failure so large that it disrupts your cluster's availability.

Of course, a variety of other factors beyond node count shape performance and availability. Resource allocations among pods and namespaces, network quality, the reliability of your underlying infrastructure and the proximity of nodes to each other on the network (to name just a few factors) also affect performance and availability in significant ways.

Why More Nodes are Not Always Better

You may be tempted to assume that the more nodes you can add to your cluster, the better it will be. That's not at all the case, for several reasons.

Not All Nodes are Created Equal

First and foremost is the fact that there is a tremendous amount of variation in what constitutes a node.

Some nodes contribute many more hardware resources to the cluster than others, and therefore do more to improve performance. In this respect, the overall node count is a very weak representation of your cluster's performance. A cluster that has 5,000 nodes (the maximum that Kubernetes can currently support), each with minimal resource allocation, may perform worse than a cluster composed of 100 high-end nodes.

In some cases, some nodes are also more likely to remain available than others. A physical server sitting in your local data center with no power backup is a less reliable node than virtual machines hosted in the cloud (which tends to be a lot more reliable than on-premises infrastructure). Thus, node count is not an exact measure of cluster availability.

Physical vs. Virtual Machine Nodes

Along similar lines, the mix of physical and virtual machines within a Kubernetes cluster impacts its performance and availability in key ways.

In Kubernetes, both physical servers and virtual machines can serve as nodes. Neither is inherently more

reliable or higher-performing than the other. However, a cluster that consists of many virtual machine nodes that are running on just a handful of physical servers is unlikely to be as reliable as one where there are more physical servers in the mix. Whether the physical servers are serving directly as nodes, or as hosts for virtual machine nodes, having more physical servers reduces the impact of the failure of any one server.

To put this another way: If you have 100 virtual machine nodes that are hosted on just five physical servers, the failure of a single physical server would reduce your node count by 20 percent. That's a huge hit, so it's better to have more physical servers in the mix.

That said, taking things to the opposite extreme is not ideal, either. If you were to make every physical server its own node, the failure of one server would deprive your cluster of the total resources that the server contributed. For availability and performance purposes, it would be better to run at least a few virtual machines on each physical server and have those virtual machines connect to the cluster as nodes. That way, if one of the nodes fails, or simply takes too long to start, only part of the resources of the underlying physical server are lost.

The bottom line: The ratio of physical machines to virtual machines within your cluster affects performance and availability in complex ways. There is no simple formula for finding the right ratio, but you should seek a healthy middle ground.

More Nodes Means More Complexity

It's worth noting, too, that the more nodes you have, the harder it is to manage and keep track of them all.

Given that so much in Kubernetes is automated, having a large node count is not a huge obstacle in this respect. But it's still a factor to consider. You will have to provision, monitor and secure every node. If your ability to do these things is limited, that is a consideration in favor of keeping your cluster smaller.

Performance and Availability are Relative

A final fact to remember is that performance and availability are always relative. You're never going to maximize either, no matter how many nodes you have (or don't have), or how perfectly your cluster is configured.

I mention this to emphasize that you can end up shooting yourself in the foot if you obsess over maximizing node count. You should aim to achieve levels of performance and availability that are acceptable for your needs, then move on. Beyond this, you end up with diminishing returns on your investment in nodes (not to mention unnecessary complexity to manage).

Rightsizing your Kubernetes Cluster

So, how do you find the sweet spot? How do you make sure you have enough nodes but not too many, and that your mix of physical and virtual machines is just right?

Obviously, there's no simple or universal answer to that question. You need to consider a variety of factors and their impact on your particular needs.

How Reliable is your Physical Infrastructure?

If the physical infrastructure that forms the foundation for your nodes is ultra reliable, then you can have fewer nodes. Generally speaking, a cloud-based Kubernetes deployment can have fewer nodes than an on-premises one for this reason. (No matter how reliable you think your on-premises data center is, it's probably not as reliable as the modern cloud.)

How Many Resources Does Each Node Have?

The hardware profiles of your nodes (whether they are physical or virtual hardware) is a key factor, too. From a performance perspective, you don't need as many nodes if each node offers a relatively high amount of hardware resources.

How Many Master Nodes Do You Have?

When it comes to overall cluster availability and performance, master nodes matter much more than worker nodes. You could have several worker nodes fail and see no major impact. But the failure of a master node could be catastrophic if it is your only master node. Even if it's not, it will still have a higher impact than the failure of a single worker.

So, consider how many master nodes your cluster contains and perhaps focus on increasing the number of masters before you worry about adding more workers. More masters reduce your need for workers.

How Many Workloads Does Your Cluster Host?

The total number of workloads is a key consideration for determining how large to make your cluster. Although Kubernetes namespaces make it easy to divide clusters into isolated zones for individual workloads (or groups of workloads), there is a point where you are better off simply breaking your cluster into smaller clusters than trying to add more namespaces.

Each namespace adds management overhead. It also increases the challenge of the "noisy neighbor" issue (which can be solved with resource quotas, but you have to set those up manually, so they're not a scalable solution).

Some *Extremely Basic* Rules of Thumb for Cluster Sizing

In case you've been reading this post looking for specific guidance on how large to make your cluster, let me reiterate that there is nothing close to a one-size-fits-all answer.

Still, I'm willing to make some very basic, exceptionally general, sweepingly oversimplified recommendations:

- For a production namespace or cluster, you should have at least one node per container. (This doesn't mean you should run each container on its own node – *au contraire* – but that the minimum total number of nodes available to host container instances should be equal to the total containers.)
- For each pod in a production namespace or cluster, you should have a physical machine. Whether it runs as its own node or hosts virtual machines that serve as nodes doesn't matter. The point is to increase the availability of your cluster by having sufficient underlying physical machines.
- If your namespaces in a single cluster exceed six, it's time to think about breaking the cluster out into smaller clusters.

Again, these are very basic rules. (Also, keep in mind, I'd cut my numbers in half if dealing with a dev/test environment, where performance and availability concerns are generally not as great.) Your mileage will certainly vary,but if you wanted some specific numbers, there you have them.

Chapter Summary: Sizing is more an art than science

Sizing Kubernetes clusters is more of an art than a science. There is a complex mix of factors at play, ranging from what type of infrastructure is hosting your nodes, to how many master nodes you have set up, to the ratio of physical to virtual machines. Treat the pointers above as very general guidelines, and be prepared to

size your cluster based on your specific needs.

Conclusion

Getting the most out of Kubernetes is hard work. Despite all of the built-in automation features that Kubernetes offers for managing workloads intelligently, achieving optimal performance, cost and reliability at scale requires careful planning and tuning of Kubernetes environments. So does the implementation of a process for managing Kubernetes that is efficient and automated, such as one based on GitOps.

We hope this guide has offered some insight into scaling and optimizing your Kubernetes clusters. To make the Kubernetes management process even simpler, we invite you to request a free trial of Platform9, a fully managed Kubernetes distribution where we do the hard work for you to keep your clusters running optimally, at scale.