Cluster Administration

Lower-level detail relevant to creating or administering a Kubernetes cluster.

The cluster administration overview is for anyone creating or administering a Kubernetes cluster. It assumes some familiarity with core Kubernetes concepts.

Planning a cluster

See the guides in Setup for examples of how to plan, set up, and configure Kubernetes clusters. The solutions listed in this article are called *distros*.

> **Note:** Not all distros are actively maintained. Choose distros which have been tested with a recent version of Kubernetes.

Before choosing a guide, here are some considerations:

- Do you want to try out Kubernetes on your computer, or do you want to build a high-availability, multi-node cluster? Choose distros best suited for your needs.
- Will you be using **a hosted Kubernetes cluster**, such as Google Kubernetes Engine, or **hosting your own cluster**?
- Will your cluster be **on-premises**, or **in the cloud (IaaS)**? Kubernetes does not directly support hybrid clusters. Instead, you can set up multiple clusters.
- **If you are configuring Kubernetes on-premises**, consider which networking model fits best.
- Will you be running Kubernetes on **"bare metal" hardware** or on **virtual machines (VMs)**?
- Do you **want to run a cluster**, or do you expect to do **active development of Kubernetes project code**? If the latter, choose an actively-developed distro. Some distros only use binary releases, but offer a greater variety of choices.
- Familiarize yourself with the components needed to run a cluster.

Managing a cluster

- Learn how to manage nodes.

- Learn how to set up and manage the resource quota for shared clusters.

Securing a cluster

- Generate Certificates describes the steps to generate certificates using different tool chains.

- Kubernetes Container Environment describes the environment for Kubelet managed containers on a Kubernetes node.

- Controlling Access to the Kubernetes API describes how Kubernetes implements access control for its own API.

- Authenticating explains authentication in Kubernetes, including the various authentication options.
- Authorization is separate from authentication, and controls how HTTP calls are handled.
- Using Admission Controllers explains plug-ins which intercepts requests to the Kubernetes API server after authentication and authorization.
- Using Sysctls in a Kubernetes Cluster describes to an administrator how to use the `sysctl` command-line tool to set kernel parameters .
- Auditing describes how to interact with Kubernetes' audit logs.

Securing the kubelet

- Control Plane-Node communication
- TLS bootstrapping
- Kubelet authentication/authorization

Optional Cluster Services

- DNS Integration describes how to resolve a DNS name directly to a Kubernetes service.
- Logging and Monitoring Cluster Activity explains how logging in Kubernetes works and how to implement it.

1 - Certificates

To learn how to generate certificates for your cluster, see [Certificates].

2 - Managing Resources

You've deployed your application and exposed it via a service. Now what? Kubernetes provides a number of tools to help you manage your application deployment, including scaling and updating. Among the features that we will discuss in more depth are configuration files and labels.

Organizing resource configurations

Many applications require multiple resources to be created, such as a Deployment and a Service. Management of multiple resources can be simplified by grouping them together in the same file (separated by `---` in YAML). For example:

application/nginx-app.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-svc
  labels:
    app: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Multiple resources can be created the same way as a single resource:

```
kubectl apply -f https://k8s.io/examples/application/nginx-app.yaml
```

```
service/my-nginx-svc created
deployment.apps/my-nginx created
```

The resources will be created in the order they appear in the file. Therefore, it's best to specify the service first, since that will ensure the scheduler can spread the pods associated with the service as they are created by the controller(s), such as Deployment.

`kubectl apply` also accepts multiple `-f` arguments:

```
kubectl apply -f https://k8s.io/examples/application/nginx/nginx-svc.yaml -f https:/
```

And a directory can be specified rather than or in addition to individual files:

```
kubectl apply -f https://k8s.io/examples/application/nginx/
```

`kubectl` will read any files with suffixes `.yaml`, `.yml`, or `.json`.

It is a recommended practice to put resources related to the same microservice or application tier into the same file, and to group all of the files associated with your application in the same directory. If the tiers of your application bind to each other using DNS, you can deploy all of the components of your stack together.

A URL can also be specified as a configuration source, which is handy for deploying directly from configuration files checked into GitHub:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/website/master/content
```

```
deployment.apps/my-nginx created
```

Bulk operations in kubectl

Resource creation isn't the only operation that `kubectl` can perform in bulk. It can also extract resource names from configuration files in order to perform other operations, in particular to delete the same resources you created:

```
kubectl delete -f https://k8s.io/examples/application/nginx-app.yaml
```

```
deployment.apps "my-nginx" deleted
service "my-nginx-svc" deleted
```

In the case of two resources, you can specify both resources on the command line using the resource/name syntax:

```
kubectl delete deployments/my-nginx services/my-nginx-svc
```

For larger numbers of resources, you'll find it easier to specify the selector (label query) specified using `-l` or `--selector`, to filter resources by their labels:

```
kubectl delete deployment,services -l app=nginx
```

```
deployment.apps "my-nginx" deleted
service "my-nginx-svc" deleted
```

Because `kubectl` outputs resource names in the same syntax it accepts, you can chain operations using `$()` or `xargs`:

```
kubectl get $(kubectl create -f docs/concepts/cluster-administration/nginx/ -o name
kubectl create -f docs/concepts/cluster-administration/nginx/ -o name | grep service
```

```
NAME           TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
my-nginx-svc   LoadBalancer   10.0.0.208   <pending>     80/TCP       0s
```

With the above commands, we first create resources under `examples/application/nginx/` and print the resources created with `-o name` output format (print each resource as resource/name). Then we `grep` only the "service", and then print it with `kubectl get`.

If you happen to organize your resources across several subdirectories within a particular directory, you can recursively perform the operations on the subdirectories also, by specifying `--recursive` or `-R` alongside the `--filename,-f` flag.

For instance, assume there is a directory `project/k8s/development` that holds all of the manifests needed for the development environment, organized by resource type:

```
project/k8s/development
├── configmap
│   └── my-configmap.yaml
├── deployment
│   └── my-deployment.yaml
└── pvc
    └── my-pvc.yaml
```

By default, performing a bulk operation on `project/k8s/development` will stop at the first level of the directory, not processing any subdirectories. If we had tried to create the resources in this directory using the following command, we would have encountered an error:

```
kubectl apply -f project/k8s/development
```

```
error: you must provide one or more resources by argument or filename (.json|.yaml|.
```

Instead, specify the `--recursive` or `-R` flag with the `--filename,-f` flag as such:

```
kubectl apply -f project/k8s/development --recursive
```

```
configmap/my-config created
deployment.apps/my-deployment created
persistentvolumeclaim/my-pvc created
```

The `--recursive` flag works with any operation that accepts the `--filename,-f` flag such as: `kubectl {create,get,delete,describe,rollout}` etc.

The `--recursive` flag also works when multiple `-f` arguments are provided:

```
kubectl apply -f project/k8s/namespaces -f project/k8s/development --recursive
```

```
namespace/development created
namespace/staging created
configmap/my-config created
deployment.apps/my-deployment created
persistentvolumeclaim/my-pvc created
```

If you're interested in learning more about `kubectl`, go ahead and read kubectl Overview.

Using labels effectively

The examples we've used so far apply at most a single label to any resource. There are many scenarios where multiple labels should be used to distinguish sets from one another.

For instance, different applications would use different values for the `app` label, but a multi-tier application, such as the guestbook example, would additionally need to distinguish each tier. The frontend could carry the following labels:

```yaml
     labels:
        app: guestbook
        tier: frontend
```

while the Redis master and slave would have different `tier` labels, and perhaps even an additional `role` label:

```yaml
     labels:
        app: guestbook
        tier: backend
        role: master
```

and

```yaml
     labels:
        app: guestbook
        tier: backend
        role: slave
```

The labels allow us to slice and dice our resources along any dimension specified by a label:

```
kubectl apply -f examples/guestbook/all-in-one/guestbook-all-in-one.yaml
kubectl get pods -Lapp -Ltier -Lrole
```

```
NAME                           READY     STATUS    RESTARTS   AGE       APP
guestbook-fe-4nlpb             1/1       Running   0          1m        guestbook
guestbook-fe-ght6d             1/1       Running   0          1m        guestbook
guestbook-fe-jpy62             1/1       Running   0          1m        guestbook
guestbook-redis-master-5pg3b   1/1       Running   0          1m        guestbook
guestbook-redis-slave-2q2yf    1/1       Running   0          1m        guestbook
```

```
guestbook-redis-slave-qgazl   1/1       Running    0          1m        guestbook
my-nginx-divi2                1/1       Running    0          29m       nginx
my-nginx-o0ef1                1/1       Running    0          29m       nginx
```

```
kubectl get pods -lapp=guestbook,role=slave
```

```
NAME                          READY     STATUS    RESTARTS   AGE
guestbook-redis-slave-2q2yf   1/1       Running   0          3m
guestbook-redis-slave-qgazl   1/1       Running   0          3m
```

Canary deployments

Another scenario where multiple labels are needed is to distinguish deployments of different releases or configurations of the same component. It is common practice to deploy a *canary* of a new application release (specified via image tag in the pod template) side by side with the previous release so that the new release can receive live production traffic before fully rolling it out.

For instance, you can use a `track` label to differentiate different releases.

The primary, stable release would have a `track` label with value as `stable`:

```
     name: frontend
     replicas: 3
     ...
     labels:
        app: guestbook
        tier: frontend
        track: stable
     ...
     image: gb-frontend:v3
```

and then you can create a new release of the guestbook frontend that carries the `track` label with different value (i.e. `canary`), so that two sets of pods would not overlap:

```
     name: frontend-canary
     replicas: 1
     ...
     labels:
        app: guestbook
        tier: frontend
        track: canary
     ...
     image: gb-frontend:v4
```

The frontend service would span both sets of replicas by selecting the common subset of their labels (i.e. omitting the `track` label), so that the traffic will be redirected to both applications:

```
  selector:
     app: guestbook
     tier: frontend
```

You can tweak the number of replicas of the stable and canary releases to determine the ratio of each release that will receive live production traffic (in this case, 3:1). Once you're confident, you can update the stable track to the new application release and remove the canary one.

For a more concrete example, check the tutorial of deploying Ghost.

Updating labels

Sometimes existing pods and other resources need to be relabeled before creating new resources. This can be done with `kubectl label`. For example, if you want to label all your nginx pods as frontend tier, run:

```
kubectl label pods -l app=nginx tier=fe
```

```
pod/my-nginx-2035384211-j5fhi labeled
pod/my-nginx-2035384211-u2c7e labeled
pod/my-nginx-2035384211-u3t6x labeled
```

This first filters all pods with the label "app=nginx", and then labels them with the "tier=fe". To see the pods you labeled, run:

```
kubectl get pods -l app=nginx -L tier
```

```
NAME                         READY     STATUS    RESTARTS   AGE       TIER
my-nginx-2035384211-j5fhi    1/1       Running   0          23m       fe
my-nginx-2035384211-u2c7e    1/1       Running   0          23m       fe
my-nginx-2035384211-u3t6x    1/1       Running   0          23m       fe
```

This outputs all "app=nginx" pods, with an additional label column of pods' tier (specified with `-L` or `--label-columns`).

For more information, please see labels and kubectl label.

Updating annotations

Sometimes you would want to attach annotations to resources. Annotations are arbitrary non-identifying metadata for retrieval by API clients such as tools, libraries, etc. This can be done with `kubectl annotate`. For example:

```
kubectl annotate pods my-nginx-v4-9gw19 description='my frontend running nginx'
kubectl get pods my-nginx-v4-9gw19 -o yaml
```

```
apiVersion: v1
kind: pod
metadata:
  annotations:
    description: my frontend running nginx
...
```

For more information, please see annotations and kubectl annotate document.

Scaling your application

When load on your application grows or shrinks, use `kubectl` to scale your application. For instance, to decrease the number of nginx replicas from 3 to 1, do:

```
kubectl scale deployment/my-nginx --replicas=1
```

```
deployment.apps/my-nginx scaled
```

Now you only have one pod managed by the deployment.

```
kubectl get pods -l app=nginx
```

```
NAME                        READY     STATUS    RESTARTS   AGE
my-nginx-2035384211-j5fhi   1/1       Running   0          30m
```

To have the system automatically choose the number of nginx replicas as needed, ranging from 1 to 3, do:

```
kubectl autoscale deployment/my-nginx --min=1 --max=3
```

```
horizontalpodautoscaler.autoscaling/my-nginx autoscaled
```

Now your nginx replicas will be scaled up and down as needed, automatically.

For more information, please see kubectl scale, kubectl autoscale and horizontal pod autoscaler document.

In-place updates of resources

Sometimes it's necessary to make narrow, non-disruptive updates to resources you've created.

kubectl apply

It is suggested to maintain a set of configuration files in source control (see configuration as code), so that they can be maintained and versioned along with the code for the resources they configure. Then, you can use `kubectl apply` to push your configuration changes to the cluster.

This command will compare the version of the configuration that you're pushing with the previous version and apply the changes you've made, without overwriting any automated changes to properties you haven't specified.

```
kubectl apply -f https://k8s.io/examples/application/nginx/nginx-deployment.yaml
deployment.apps/my-nginx configured
```

Note that `kubectl apply` attaches an annotation to the resource in order to determine the changes to the configuration since the previous invocation. When it's invoked, `kubectl apply` does a three-way diff between the previous configuration, the provided input and the current configuration of the resource, in order to determine how to modify the resource.

Currently, resources are created without this annotation, so the first invocation of `kubectl apply` will fall back to a two-way diff between the provided input and the current configuration of the resource. During this first invocation, it cannot detect the deletion of properties set when the resource was created. For this reason, it will not remove them.

All subsequent calls to `kubectl apply`, and other commands that modify the configuration, such as `kubectl replace` and `kubectl edit`, will update the annotation, allowing subsequent calls to `kubectl apply` to detect and perform deletions using a three-way diff.

kubectl edit

Alternatively, you may also update resources with `kubectl edit`:

```
kubectl edit deployment/my-nginx
```

This is equivalent to first `get` the resource, edit it in text editor, and then `apply` the resource with the updated version:

```
kubectl get deployment my-nginx -o yaml > /tmp/nginx.yaml
vi /tmp/nginx.yaml
# do some edit, and then save the file

kubectl apply -f /tmp/nginx.yaml
deployment.apps/my-nginx configured

rm /tmp/nginx.yaml
```

This allows you to do more significant changes more easily. Note that you can specify the editor with your `EDITOR` or `KUBE_EDITOR` environment variables.

For more information, please see [kubectl edit](#) document.

kubectl patch

You can use `kubectl patch` to update API objects in place. This command supports JSON patch, JSON merge patch, and strategic merge patch. See [Update API Objects in Place Using kubectl patch](#) and [kubectl patch](#).

Disruptive updates

In some cases, you may need to update resource fields that cannot be updated once initialized, or you may want to make a recursive change immediately, such as to fix broken pods created by a Deployment. To change such fields, use `replace --force`, which deletes and re-creates the resource. In this case, you can modify your original configuration file:

```
kubectl replace -f https://k8s.io/examples/application/nginx/nginx-deployment.yaml -
```

```
deployment.apps/my-nginx deleted
deployment.apps/my-nginx replaced
```

Updating your application without a service outage

At some point, you'll eventually need to update your deployed application, typically by specifying a new image or image tag, as in the canary deployment scenario above. `kubectl` supports several update operations, each of which is applicable to different scenarios.

We'll guide you through how to create and update applications with Deployments.

Let's say you were running version 1.14.2 of nginx:

```
kubectl create deployment my-nginx --image=nginx:1.14.2
```

```
deployment.apps/my-nginx created
```

with 3 replicas (so the old and new revisions can coexist):

```
kubectl scale deployment my-nginx --current-replicas=1 --replicas=3
```

```
deployment.apps/my-nginx scaled
```

To update to version 1.16.1, change `.spec.template.spec.containers[0].image` from `nginx:1.14.2` to `nginx:1.16.1` using the previous kubectl commands.

```
kubectl edit deployment/my-nginx
```

That's it! The Deployment will declaratively update the deployed nginx application progressively behind the scene. It ensures that only a certain number of old replicas may be down while they are being updated, and only a certain number of new replicas may be created above the desired number of pods. To learn more details about it, visit Deployment page.

What's next

- Learn about how to use `kubectl` for application introspection and debugging.
- See Configuration Best Practices and Tips.

3 - Cluster Networking

Networking is a central part of Kubernetes, but it can be challenging to understand exactly how it is expected to work. There are 4 distinct networking problems to address:

1. Highly-coupled container-to-container communications: this is solved by Pods and `localhost` communications.
2. Pod-to-Pod communications: this is the primary focus of this document.
3. Pod-to-Service communications: this is covered by services.
4. External-to-Service communications: this is covered by services.

Kubernetes is all about sharing machines between applications. Typically, sharing machines requires ensuring that two applications do not try to use the same ports. Coordinating ports across multiple developers is very difficult to do at scale and exposes users to cluster-level issues outside of their control.

Dynamic port allocation brings a lot of complications to the system - every application has to take ports as flags, the API servers have to know how to insert dynamic port numbers into configuration blocks, services have to know how to find each other, etc. Rather than deal with this, Kubernetes takes a different approach.

The Kubernetes network model

Every `Pod` gets its own IP address. This means you do not need to explicitly create links between `Pods` and you almost never need to deal with mapping container ports to host ports. This creates a clean, backwards-compatible model where `Pods` can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- pods on a node can communicate with all pods on all nodes without NAT
- agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node

Note: For those platforms that support `Pods` running in the host network (e.g. Linux):

- pods in the host network of a node can communicate with all pods on all nodes without NAT

This model is not only less complex overall, but it is principally compatible with the desire for Kubernetes to enable low-friction porting of apps from VMs to containers. If your job previously ran in a VM, your VM had an IP and could talk to other VMs in your project. This is the same basic model.

Kubernetes IP addresses exist at the `Pod` scope - containers within a `Pod` share their network namespaces - including their IP address and MAC address. This means that containers within a `Pod` can all reach each other's ports on `localhost`. This also means that containers within a `Pod` must coordinate port usage, but this is no different from processes in a VM. This is called the "IP-per-pod" model.

How this is implemented is a detail of the particular container runtime in use.

It is possible to request ports on the `Node` itself which forward to your `Pod` (called host ports), but this is a very niche operation. How that forwarding is implemented is also a detail of the container runtime. The `Pod` itself is blind to the existence or non-existence of host ports.

How to implement the Kubernetes networking model

There are a number of ways that this network model can be implemented. This document is not an exhaustive study of the various methods, but hopefully serves as an introduction to various technologies and serves as a jumping-off point.

The following networking options are sorted alphabetically - the order does not imply any preferential status.

> **Caution:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects. This page follows CNCF website guidelines by listing projects alphabetically. To add a project to this list, read the content guide before submitting a change.

ACI

Cisco Application Centric Infrastructure offers an integrated overlay and underlay SDN solution that supports containers, virtual machines, and bare metal servers. ACI provides container networking integration for ACI. An overview of the integration is provided here.

Antrea

Project Antrea is an opensource Kubernetes networking solution intended to be Kubernetes native. It leverages Open vSwitch as the networking data plane. Open vSwitch is a high-performance programmable virtual switch that supports both Linux and Windows. Open vSwitch enables Antrea to implement Kubernetes Network Policies in a high-performance and efficient manner. Thanks to the "programmable" characteristic of Open vSwitch, Antrea is able to implement an extensive set of networking and security features and services on top of Open vSwitch.

AOS from Apstra

AOS is an Intent-Based Networking system that creates and manages complex datacenter environments from a simple integrated platform. AOS leverages a highly scalable distributed design to eliminate network outages while minimizing costs.

The AOS Reference Design currently supports Layer-3 connected hosts that eliminate legacy Layer-2 switching problems. These Layer-3 hosts can be Linux servers (Debian, Ubuntu, CentOS) that create BGP neighbor relationships directly with the top of rack switches (TORs). AOS automates the routing adjacencies and then provides fine grained control over the route health injections (RHI) that are common in a Kubernetes deployment.

AOS has a rich set of REST API endpoints that enable Kubernetes to quickly change the network policy based on application requirements. Further enhancements will integrate the AOS Graph model used for the network design with the workload provisioning, enabling an end to end management system for both private and public clouds.

AOS supports the use of common vendor equipment from manufacturers including Cisco, Arista, Dell, Mellanox, HPE, and a large number of white-box systems and open network operating systems like Microsoft SONiC, Dell OPX, and Cumulus Linux.

Details on how the AOS system works can be accessed here: https://www.apstra.com/products/how-it-works/

AWS VPC CNI for Kubernetes

The AWS VPC CNI offers integrated AWS Virtual Private Cloud (VPC) networking for Kubernetes clusters. This CNI plugin offers high throughput and availability, low latency, and minimal network jitter. Additionally, users can apply existing AWS VPC networking and security best practices for building Kubernetes clusters. This includes the ability to use VPC flow logs, VPC routing policies, and security groups for network traffic isolation.

Using this CNI plugin allows Kubernetes pods to have the same IP address inside the pod as they do on the VPC network. The CNI allocates AWS Elastic Networking Interfaces (ENIs) to each Kubernetes node and using the secondary IP range from each ENI for pods on the node. The CNI includes controls for pre-allocation of ENIs and IP addresses for fast pod startup times and enables large clusters of up to 2,000 nodes.

Additionally, the CNI can be run alongside Calico for network policy enforcement. The AWS VPC CNI project is open source with documentation on GitHub.

Azure CNI for Kubernetes

Azure CNI is an open source plugin that integrates Kubernetes Pods with an Azure Virtual Network (also known as VNet) providing network performance at par with VMs. Pods can connect to peered VNet and to on-premises over Express Route or site-to-site VPN and are also directly reachable from these networks. Pods can access Azure services, such as storage and SQL, that are protected by Service Endpoints or Private Link. You can use VNet security policies and routing to filter Pod traffic. The plugin assigns VNet IPs to Pods by utilizing a pool of secondary IPs pre-configured on the Network Interface of a Kubernetes node.

Azure CNI is available natively in the Azure Kubernetes Service (AKS).

Big Cloud Fabric from Big Switch Networks

Big Cloud Fabric is a cloud native networking architecture, designed to run Kubernetes in private cloud/on-premises environments. Using unified physical & virtual SDN, Big Cloud Fabric tackles inherent container networking problems such as load balancing, visibility, troubleshooting, security policies & container traffic monitoring.

With the help of the Big Cloud Fabric's virtual pod multi-tenant architecture, container orchestration systems such as Kubernetes, RedHat OpenShift, Mesosphere DC/OS & Docker Swarm will be natively integrated alongside with VM orchestration systems such as VMware, OpenStack & Nutanix. Customers will be able to securely inter-connect any number of these clusters and enable inter-tenant communication between them if needed.

BCF was recognized by Gartner as a visionary in the latest Magic Quadrant. One of the BCF Kubernetes on-premises deployments (which includes Kubernetes, DC/OS & VMware running on multiple DCs across different geographic regions) is also referenced here.

Calico

Calico is an open source networking and network security solution for containers, virtual machines, and native host-based workloads. Calico supports multiple data planes including: a pure Linux eBPF dataplane, a standard Linux networking dataplane, and a Windows HNS dataplane. Calico provides a full networking stack but can also be used in conjunction with cloud provider CNIs to provide network policy enforcement.

Cilium

Cilium is open source software for providing and transparently securing network connectivity between application containers. Cilium is L7/HTTP aware and can enforce network policies on L3-L7 using an identity based security model that is decoupled from network addressing, and it can be used in combination with other CNI plugins.

CNI-Genie from Huawei

CNI-Genie is a CNI plugin that enables Kubernetes to simultaneously have access to different implementations of the Kubernetes network model in runtime. This includes any implementation that runs as a CNI plugin, such as Flannel, Calico, Romana, Weave-net.

CNI-Genie also supports assigning multiple IP addresses to a pod, each from a different CNI plugin.

cni-ipvlan-vpc-k8s

cni-ipvlan-vpc-k8s contains a set of CNI and IPAM plugins to provide a simple, host-local, low latency, high throughput, and compliant networking stack for Kubernetes within Amazon Virtual Private Cloud (VPC) environments by making use of Amazon Elastic Network Interfaces (ENI) and binding AWS-managed IPs into Pods using the Linux kernel's IPvlan driver in L2 mode.

The plugins are designed to be straightforward to configure and deploy within a VPC. Kubelets boot and then self-configure and scale their IP usage as needed without requiring the often recommended complexities of administering overlay networks, BGP, disabling source/destination checks, or adjusting VPC route tables to provide per-instance subnets to each host (which is limited to 50-100 entries per VPC). In short, cni-ipvlan-vpc-k8s significantly reduces the network complexity required to deploy Kubernetes at scale within AWS.

Coil

Coil is a CNI plugin designed for ease of integration, providing flexible egress networking. Coil operates with a low overhead compared to bare metal, and allows you to define arbitrary egress NAT gateways for external networks.

Contiv

Contiv provides configurable networking (native l3 using BGP, overlay using vxlan, classic l2, or Cisco-SDN/ACI) for various use cases. Contiv is all open sourced.

Contrail / Tungsten Fabric

Contrail, based on Tungsten Fabric, is a truly open, multi-cloud network virtualization and policy management platform. Contrail and Tungsten Fabric are integrated with various orchestration systems such as Kubernetes, OpenShift, OpenStack and Mesos, and provide different isolation modes for virtual machines, containers/pods and bare metal workloads.

DANM

DANM is a networking solution for telco workloads running in a Kubernetes cluster. It's built up from the following components:

- A CNI plugin capable of provisioning IPVLAN interfaces with advanced features
- An in-built IPAM module with the capability of managing multiple, cluster-wide, discontinuous L3 networks and provide a dynamic, static, or no IP allocation scheme on-demand
- A CNI metaplugin capable of attaching multiple network interfaces to a container, either through its own CNI, or through delegating the job to any of the popular CNI solution like SRI-OV, or Flannel in parallel
- A Kubernetes controller capable of centrally managing both VxLAN and VLAN interfaces of all Kubernetes hosts
- Another Kubernetes controller extending Kubernetes' Service-based service discovery concept to work over all network interfaces of a Pod

With this toolset DANM is able to provide multiple separated network interfaces, the possibility to use different networking back ends and advanced IPAM features for the pods.

Flannel

Flannel is a very simple overlay network that satisfies the Kubernetes requirements. Many people have reported success with Flannel and Kubernetes.

Google Compute Engine (GCE)

For the Google Compute Engine cluster configuration scripts, advanced routing is used to assign each VM a subnet (default is `/24` - 254 IPs). Any traffic bound for that subnet will be routed directly to the VM by the GCE network fabric. This is in addition to the "main" IP address assigned to the VM, which is NAT'ed for outbound internet access. A linux bridge (called `cbr0`) is configured to exist on that subnet, and is passed to docker's `--bridge` flag.

Docker is started with:

```
DOCKER_OPTS="--bridge=cbr0 --iptables=false --ip-masq=false"
```

This bridge is created by Kubelet (controlled by the `--network-plugin=kubenet` flag) according to the `Node` 's `.spec.podCIDR` .

Docker will now allocate IPs from the `cbr-cidr` block. Containers can reach each other and `Nodes` over the `cbr0` bridge. Those IPs are all routable within the GCE project network.

GCE itself does not know anything about these IPs, though, so it will not NAT them for outbound internet traffic. To achieve that an iptables rule is used to masquerade (aka SNAT - to make it seem as if packets came from the `Node` itself) traffic that is bound for IPs outside the GCE project network (10.0.0.0/8).

```
iptables -t nat -A POSTROUTING ! -d 10.0.0.0/8 -o eth0 -j MASQUERADE
```

Lastly IP forwarding is enabled in the kernel (so the kernel will process packets for bridged containers):

```
sysctl net.ipv4.ip_forward=1
```

The result of all this is that all `Pods` can reach each other and can egress traffic to the internet.

Jaguar

Jaguar is an open source solution for Kubernetes's network based on OpenDaylight. Jaguar provides overlay network using vxlan and Jaguar CNIPlugin provides one IP address per pod.

k-vswitch

k-vswitch is a simple Kubernetes networking plugin based on Open vSwitch. It leverages existing functionality in Open vSwitch to provide a robust networking plugin that is easy-to-operate, performant and secure.

Knitter

Knitter is a network solution which supports multiple networking in Kubernetes. It provides the ability of tenant management and network management. Knitter includes a set of end-to-end NFV container networking solutions besides multiple network planes, such as keeping IP address for applications, IP address migration, etc.

Kube-OVN

Kube-OVN is an OVN-based kubernetes network fabric for enterprises. With the help of OVN/OVS, it provides some advanced overlay network features like subnet, QoS, static IP allocation, traffic mirroring, gateway, openflow-based network policy and service proxy.

Kube-router

Kube-router is a purpose-built networking solution for Kubernetes that aims to provide high performance and operational simplicity. Kube-router provides a Linux LVS/IPVS-based service proxy, a Linux kernel forwarding-based pod-to-pod networking solution with no overlays, and iptables/ipset-based network policy enforcer.

L2 networks and linux bridging

If you have a "dumb" L2 network, such as a simple switch in a "bare-metal" environment, you should be able to do something similar to the above GCE setup. Note that these instructions have only been tried very casually - it seems to work, but has not been thoroughly tested. If you use this technique and perfect the process, please let us know.

Follow the "With Linux Bridge devices" section of this very nice tutorial from Lars Kellogg-Stedman.

Multus (a Multi Network plugin)

Multus is a Multi CNI plugin to support the Multi Networking feature in Kubernetes using CRD based network objects in Kubernetes.

Multus supports all reference plugins (eg. Flannel, DHCP, Macvlan) that implement the CNI specification and 3rd party plugins (eg. Calico, Weave, Cilium, Contiv). In addition to it, Multus supports SRIOV, DPDK, OVS-DPDK & VPP workloads in Kubernetes with both cloud native and NFV based applications in Kubernetes.

OVN4NFV-K8s-Plugin (OVN based CNI controller & plugin)

OVN4NFV-K8S-Plugin is OVN based CNI controller plugin to provide cloud native based Service function chaining(SFC), Multiple OVN overlay networking, dynamic subnet creation, dynamic creation of virtual networks, VLAN Provider network, Direct provider network and pluggable with other Multi-network plugins, ideal for edge based cloud native workloads in Multi-cluster networking

NSX-T

VMware NSX-T is a network virtualization and security platform. NSX-T can provide network virtualization for a multi-cloud and multi-hypervisor environment and is focused on emerging application frameworks and architectures that have heterogeneous endpoints and technology stacks. In addition to vSphere hypervisors, these environments include other hypervisors such as KVM, containers, and bare metal.

NSX-T Container Plug-in (NCP) provides integration between NSX-T and container orchestrators such as Kubernetes, as well as integration between NSX-T and container-based CaaS/PaaS platforms such as Pivotal Container Service (PKS) and OpenShift.

Nuage Networks VCS (Virtualized Cloud Services)

Nuage provides a highly scalable policy-based Software-Defined Networking (SDN) platform. Nuage uses the open source Open vSwitch for the data plane along with a feature rich SDN Controller built on open standards.

The Nuage platform uses overlays to provide seamless policy-based networking between Kubernetes Pods and non-Kubernetes environments (VMs and bare metal servers). Nuage's policy abstraction model is designed with applications in mind and makes it easy to declare fine-grained policies for applications.The platform's real-time analytics engine enables visibility and security monitoring for Kubernetes applications.

OpenVSwitch

OpenVSwitch is a somewhat more mature but also complicated way to build an overlay network. This is endorsed by several of the "Big Shops" for networking.

OVN (Open Virtual Networking)

OVN is an opensource network virtualization solution developed by the Open vSwitch community. It lets one create logical switches, logical routers, stateful ACLs, load-balancers etc to build different virtual networking topologies. The project has a specific Kubernetes plugin and documentation at ovn-kubernetes.

Romana

Romana is an open source network and security automation solution that lets you deploy Kubernetes without an overlay network. Romana supports Kubernetes Network Policy to provide isolation across network namespaces.

Weave Net from Weaveworks

Weave Net is a resilient and simple to use network for Kubernetes and its hosted applications. Weave Net runs as a CNI plug-in or stand-alone. In either version, it doesn't require any configuration or extra code to run, and in both cases, the network provides one IP address per pod - as is standard for Kubernetes.

What's next

The early design of the networking model and its rationale, and some future plans are described in more detail in the networking design document.

4 - Logging Architecture

Application logs can help you understand what is happening inside your application. The logs are particularly useful for debugging problems and monitoring cluster activity. Most modern applications have some kind of logging mechanism. Likewise, container engines are designed to support logging. The easiest and most adopted logging method for containerized applications is writing to standard output and standard error streams.

However, the native functionality provided by a container engine or runtime is usually not enough for a complete logging solution. For example, you may want access your application's logs if a container crashes; a pod gets evicted; or a node dies. In a cluster, logs should have a separate storage and lifecycle independent of nodes, pods, or containers. This concept is called *cluster-level logging*.

Cluster-level logging architectures require a separate backend to store, analyze, and query logs. Kubernetes does not provide a native storage solution for log data. Instead, there are many logging solutions that integrate with Kubernetes. The following sections describe how to handle and store logs on nodes.

Basic logging in Kubernetes

This example uses a `Pod` specification with a container to write text to the standard output stream once per second.

debug/counter-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args: [/bin/sh, -c,
            'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done']
```

To run this pod, use the following command:

```
kubectl apply -f https://k8s.io/examples/debug/counter-pod.yaml
```

The output is:

```
pod/counter created
```

To fetch the logs, use the `kubectl logs` command, as follows:

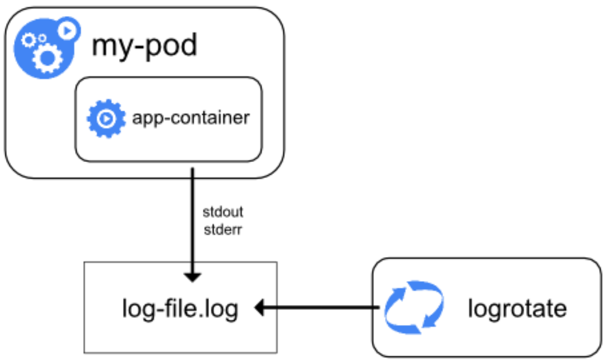
```
kubectl logs counter
```

The output is:

```
0: Mon Jan  1 00:00:00 UTC 2001
1: Mon Jan  1 00:00:01 UTC 2001
2: Mon Jan  1 00:00:02 UTC 2001
...
```

You can use `kubectl logs --previous` to retrieve logs from a previous instantiation of a container. If your pod has multiple containers, specify which container's logs you want to access by appending a container name to the command. See the `kubectl logs` documentation for more details.

Logging at the node level



A container engine handles and redirects any output generated to a containerized application's `stdout` and `stderr` streams. For example, the Docker container engine redirects those two streams to a logging driver, which is configured in Kubernetes to write to a file in JSON format.

> **Note:** The Docker JSON logging driver treats each line as a separate message. When using the Docker logging driver, there is no direct support for multi-line messages. You need to handle multi-line messages at the logging agent level or higher.

By default, if a container restarts, the kubelet keeps one terminated container with its logs. If a pod is evicted from the node, all corresponding containers are also evicted, along with their logs.

An important consideration in node-level logging is implementing log rotation, so that logs don't consume all available storage on the node. Kubernetes is not responsible for rotating logs, but rather a deployment tool should set up a solution to address that. For example, in Kubernetes clusters, deployed by the `kube-up.sh` script, there is a `logrotate` tool configured to run each hour. You can also set up a container runtime to rotate an application's logs automatically.

As an example, you can find detailed information about how `kube-up.sh` sets up logging for COS image on GCP in the corresponding `configure-helper` script.

When you run `kubectl logs` as in the basic logging example, the kubelet on the node handles the request and reads directly from the log file. The kubelet returns the content of the log file.

> **Note:** If an external system has performed the rotation, only the contents of the latest log file will be available through `kubectl logs`. For example, if there's a 10MB file, `logrotate` performs the rotation and there are two files: one file that is 10MB in size and a second file that is empty. `kubectl logs` returns the latest log file which in this example is an empty response.

System component logs

There are two types of system components: those that run in a container and those that do not run in a container. For example:

- The Kubernetes scheduler and kube-proxy run in a container.
- The kubelet and container runtime do not run in containers.

On machines with systemd, the kubelet and container runtime write to journald. If systemd is not present, the kubelet and container runtime write to `.log` files in the `/var/log` directory. System components inside containers always write to the `/var/log` directory, bypassing the default logging mechanism. They use the `klog` logging library. You can find the conventions for logging severity for those components in the development docs on logging.

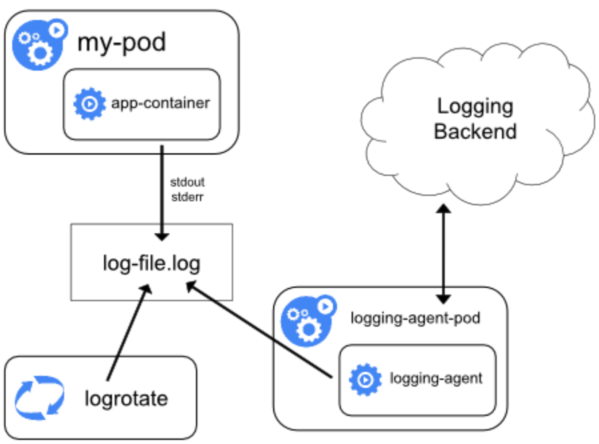
Similar to the container logs, system component logs in the `/var/log` directory should be rotated. In Kubernetes clusters brought up by the `kube-up.sh` script, those logs are configured to be rotated by the `logrotate` tool daily or once the size exceeds 100MB.

Cluster-level logging architectures

While Kubernetes does not provide a native solution for cluster-level logging, there are several common approaches you can consider. Here are some options:

- Use a node-level logging agent that runs on every node.
- Include a dedicated sidecar container for logging in an application pod.
- Push logs directly to a backend from within an application.

Using a node logging agent



You can implement cluster-level logging by including a *node-level logging agent* on each node. The logging agent is a dedicated tool that exposes logs or pushes logs to a backend. Commonly, the logging agent is a container that has access to a directory with log files from all of the application containers on that node.

Because the logging agent must run on every node, it is recommended to run the agent as a `DaemonSet`.

Node-level logging creates only one agent per node and doesn't require any changes to the applications running on the node.

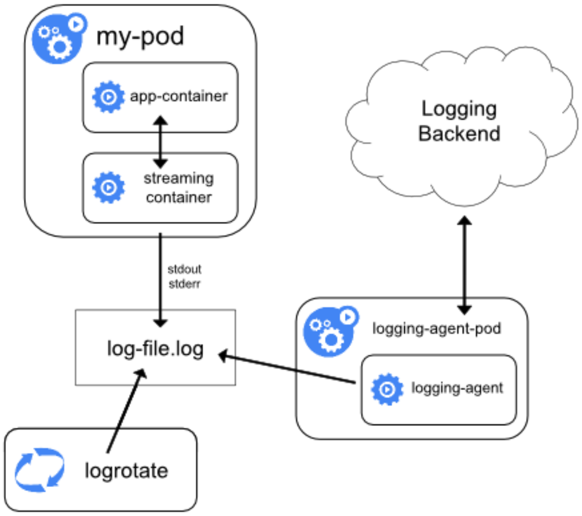
Containers write stdout and stderr, but with no agreed format. A node-level agent collects these logs and forwards them for aggregation.

Using a sidecar container with the logging agent

You can use a sidecar container in one of the following ways:

- The sidecar container streams application logs to its own `stdout`.
- The sidecar container runs a logging agent, which is configured to pick up logs from an application container.

Streaming sidecar container



By having your sidecar containers write to their own `stdout` and `stderr` streams, you can take advantage of the kubelet and the logging agent that already run on each node. The sidecar containers read logs from a file, a socket, or journald. Each sidecar container prints a log to its own `stdout` or `stderr` stream.

This approach allows you to separate several log streams from different parts of your application, some of which can lack support for writing to `stdout` or `stderr`. The logic behind redirecting logs is minimal, so it's not a significant overhead. Additionally, because `stdout` and `stderr` are handled by the kubelet, you can use built-in tools like `kubectl logs`.

For example, a pod runs a single container, and the container writes to two different log files using two different formats. Here's a configuration file for the Pod:

admin/logging/two-files-counter-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  volumes:
  - name: varlog
    emptyDir: {}
```

It is not recommended to write log entries with different formats to the same log stream, even if you managed to redirect both components to the `stdout` stream of the container. Instead, you can create two sidecar containers. Each sidecar container could tail a particular log file from a shared volume and then redirect the logs to its own `stdout` stream.

Here's a configuration file for a pod that has two sidecar containers:

admin/logging/two-files-counter-pod-streaming-sidecar.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-log-1
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/1.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-log-2
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/2.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  volumes:
  - name: varlog
    emptyDir: {}
```

Now when you run this pod, you can access each log stream separately by running the following commands:

```
kubectl logs counter count-log-1
```

The output is:

```
0: Mon Jan  1 00:00:00 UTC 2001
1: Mon Jan  1 00:00:01 UTC 2001
2: Mon Jan  1 00:00:02 UTC 2001
...
```

```
kubectl logs counter count-log-2
```

The output is:

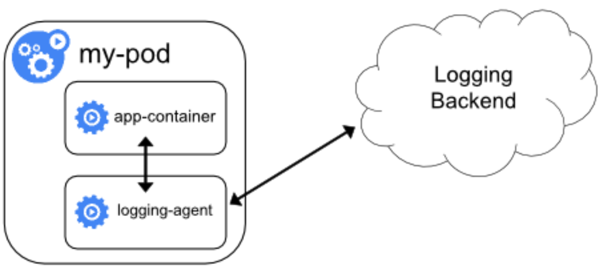
```
Mon Jan  1 00:00:00 UTC 2001 INFO 0
Mon Jan  1 00:00:01 UTC 2001 INFO 1
Mon Jan  1 00:00:02 UTC 2001 INFO 2
...
```

The node-level agent installed in your cluster picks up those log streams automatically without any further configuration. If you like, you can configure the agent to parse log lines depending on the source container.

Note, that despite low CPU and memory usage (order of a couple of millicores for cpu and order of several megabytes for memory), writing logs to a file and then streaming them to `stdout` can double disk usage. If you have an application that writes to a single file, it's recommended to set `/dev/stdout` as the destination rather than implement the streaming sidecar container approach.

Sidecar containers can also be used to rotate log files that cannot be rotated by the application itself. An example of this approach is a small container running `logrotate` periodically. However, it's recommended to use `stdout` and `stderr` directly and leave rotation and retention policies to the kubelet.

Sidecar container with a logging agent



If the node-level logging agent is not flexible enough for your situation, you can create a sidecar container with a separate logging agent that you have configured specifically to run with your application.

> **Note:** Using a logging agent in a sidecar container can lead to significant resource consumption. Moreover, you won't be able to access those logs using `kubectl logs` because they are not controlled by the kubelet.

Here are two configuration files that you can use to implement a sidecar container with a logging agent. The first file contains a [`ConfigMap`] to configure fluentd.

admin/logging/fluentd-sidecar-config.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: fluentd-config
data:
  fluentd.conf: |
    <source>
      type tail
      format none
      path /var/log/1.log
      pos_file /var/log/1.log.pos
      tag count.format1
    </source>

    <source>
      type tail
```

```
    format none
    path /var/log/2.log
    pos_file /var/log/2.log.pos
    tag count.format2
  </source>

  <match **>
    type google_cloud
  </match>
```

Note: For information about configuring fluentd, see the fluentd documentation.

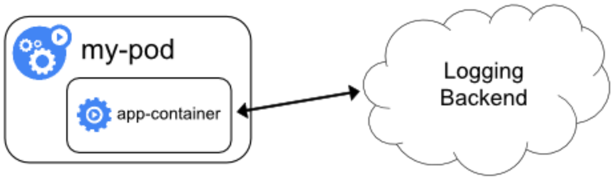
The second file describes a pod that has a sidecar container running fluentd. The pod mounts a volume where fluentd can pick up its configuration data.

admin/logging/two-files-counter-pod-agent-sidecar.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-agent
    image: k8s.gcr.io/fluentd-gcp:1.30
    env:
    - name: FLUENTD_ARGS
      value: -c /etc/fluentd-config/fluentd.conf
    volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: config-volume
      mountPath: /etc/fluentd-config
  volumes:
  - name: varlog
    emptyDir: {}
  - name: config-volume
    configMap:
      name: fluentd-config
```

In the sample configurations, you can replace fluentd with any logging agent, reading from any source inside an application container.

Exposing logs directly from the application



Cluster-logging that exposes or pushes logs directly from every application is outside the scope of Kubernetes.

5 - Metrics For Kubernetes System Components

System component metrics can give a better look into what is happening inside them. Metrics are particularly useful for building dashboards and alerts.

Kubernetes components emit metrics in [Prometheus format](). This format is structured plain text, designed so that people and machines can both read it.

Metrics in Kubernetes

In most cases metrics are available on `/metrics` endpoint of the HTTP server. For components that doesn't expose endpoint by default it can be enabled using `--bind-address` flag.

Examples of those components:

- kube-controller-manager
- kube-proxy
- kube-apiserver
- kube-scheduler
- kubelet

In a production environment you may want to configure [Prometheus Server]() or some other metrics scraper to periodically gather these metrics and make them available in some kind of time series database.

Note that kubelet also exposes metrics in `/metrics/cadvisor`, `/metrics/resource` and `/metrics/probes` endpoints. Those metrics do not have same lifecycle.

If your cluster uses RBAC, reading metrics requires authorization via a user, group or ServiceAccount with a ClusterRole that allows accessing `/metrics`. For example:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
  - nonResourceURLs:
      - "/metrics"
    verbs:
      - get
```

Metric lifecycle

Alpha metric → Stable metric → Deprecated metric → Hidden metric → Deleted metric

Alpha metrics have no stability guarantees. These metrics can be modified or deleted at any time.

Stable metrics are guaranteed to not change. This means:

- A stable metric without a deprecated signature will not be deleted or renamed
- A stable metric's type will not be modified

Deprecated metrics are slated for deletion, but are still available for use. These metrics include an annotation about the version in which they became deprecated.

For example:

- Before deprecation

```
# HELP some_counter this counts things
# TYPE some_counter counter
some_counter 0
```

- After deprecation

```
# HELP some_counter (Deprecated since 1.15.0) this counts things
# TYPE some_counter counter
some_counter 0
```

Hidden metrics are no longer published for scraping, but are still available for use. To use a hidden metric, please refer to the Show hidden metrics section.

Deleted metrics are no longer published and cannot be used.

Show hidden metrics

As described above, admins can enable hidden metrics through a command-line flag on a specific binary. This intends to be used as an escape hatch for admins if they missed the migration of the metrics deprecated in the last release.

The flag `show-hidden-metrics-for-version` takes a version for which you want to show metrics deprecated in that release. The version is expressed as x.y, where x is the major version, y is the minor version. The patch version is not needed even though a metrics can be deprecated in a patch release, the reason for that is the metrics deprecation policy runs against the minor release.

The flag can only take the previous minor version as it's value. All metrics hidden in previous will be emitted if admins set the previous version to `show-hidden-metrics-for-version`. The too old version is not allowed because this violates the metrics deprecated policy.

Take metric `A` as an example, here assumed that `A` is deprecated in 1.n. According to metrics deprecated policy, we can reach the following conclusion:

- In release `1.n`, the metric is deprecated, and it can be emitted by default.
- In release `1.n+1`, the metric is hidden by default and it can be emitted by command line `show-hidden-metrics-for-version=1.n`.
- In release `1.n+2`, the metric should be removed from the codebase. No escape hatch anymore.

If you're upgrading from release `1.12` to `1.13`, but still depend on a metric `A` deprecated in `1.12`, you should set hidden metrics via command line: `--show-hidden-metrics=1.12` and remember to remove this metric dependency before upgrading to `1.14`

Disable accelerator metrics

The kubelet collects accelerator metrics through cAdvisor. To collect these metrics, for accelerators like NVIDIA GPUs, kubelet held an open handle on the driver. This meant that in order to perform infrastructure changes (for example, updating the driver), a cluster administrator needed to stop the kubelet agent.

The responsibility for collecting accelerator metrics now belongs to the vendor rather than the kubelet. Vendors must provide a container that collects metrics and exposes them to the metrics service (for example, Prometheus).

The `DisableAcceleratorUsageMetrics` feature gate disables metrics collected by the kubelet, with a timeline for enabling this feature by default.

Component metrics

kube-controller-manager metrics

Controller manager metrics provide important insight into the performance and health of the controller manager. These metrics include common Go language runtime metrics such as go_routine count and controller specific metrics such as etcd request latencies or Cloudprovider (AWS, GCE, OpenStack) API latencies that can be used to gauge the health of a cluster.

Starting from Kubernetes 1.7, detailed Cloudprovider metrics are available for storage operations for GCE, AWS, Vsphere and OpenStack. These metrics can be used to monitor health of persistent volume operations.

For example, for GCE these metrics are called:

```
cloudprovider_gce_api_request_duration_seconds { request = "instance_list"}
cloudprovider_gce_api_request_duration_seconds { request = "disk_insert"}
cloudprovider_gce_api_request_duration_seconds { request = "disk_delete"}
cloudprovider_gce_api_request_duration_seconds { request = "attach_disk"}
cloudprovider_gce_api_request_duration_seconds { request = "detach_disk"}
cloudprovider_gce_api_request_duration_seconds { request = "list_disk"}
```

kube-scheduler metrics

FEATURE STATE: `Kubernetes v1.20 [alpha]`

The scheduler exposes optional metrics that reports the requested resources and the desired limits of all running pods. These metrics can be used to build capacity planning dashboards, assess current or historical scheduling limits, quickly identify workloads that cannot schedule due to lack of resources, and compare actual usage to the pod's request.

The kube-scheduler identifies the resource requests and limits configured for each Pod; when either a request or limit is non-zero, the kube-scheduler reports a metrics timeseries. The time series is labelled by:

- namespace
- pod name
- the node where the pod is scheduled or an empty string if not yet scheduled
- priority
- the assigned scheduler for that pod
- the name of the resource (for example, `cpu`)
- the unit of the resource if known (for example, `cores`)

Once a pod reaches completion (has a `restartPolicy` of `Never` or `OnFailure` and is in the `Succeeded` or `Failed` pod phase, or has been deleted and all containers have a terminated state) the series is no longer reported since the scheduler is now free to schedule other pods to run. The two metrics are called `kube_pod_resource_request` and `kube_pod_resource_limit`.

The metrics are exposed at the HTTP endpoint `/metrics/resources` and require the same authorization as the `/metrics` endpoint on the scheduler. You must use the `--show-hidden-metrics-for-version=1.20` flag to expose these alpha stability metrics.

What's next

- Read about the Prometheus text format for metrics
- Read about the Kubernetes deprecation policy

6 - System Logs

System component logs record events happening in cluster, which can be very useful for debugging. You can configure log verbosity to see more or less detail. Logs can be as coarse-grained as showing errors within a component, or as fine-grained as showing step-by-step traces of events (like HTTP access logs, pod state changes, controller actions, or scheduler decisions).

Klog

klog is the Kubernetes logging library. klog generates log messages for the Kubernetes system components.

For more information about klog configuration, see the Command line tool reference.

An example of the klog native format:

```
I1025 00:15:15.525108       1 httplog.go:79] GET /api/v1/namespaces/kube-system/pods
```

Structured Logging

FEATURE STATE: `Kubernetes v1.19 [alpha]`

> **Warning:**
> Migration to structured log messages is an ongoing process. Not all log messages are structured in this version. When parsing log files, you must also handle unstructured log messages.
>
> Log formatting and value serialization are subject to change.

Structured logging introduces a uniform structure in log messages allowing for programmatic extraction of information. You can store and process structured logs with less effort and cost. New message format is backward compatible and enabled by default.

Format of structured logs:

```
<klog header> "<message>" <key1>="<value1>" <key2>="<value2>" ...
```

Example:

```
I1025 00:15:15.525108       1 controller_utils.go:116] "Pod status updated" pod="kub
```

JSON log format

FEATURE STATE: `Kubernetes v1.19 [alpha]`

> **Warning:**
> JSON output does not support many standard klog flags. For list of unsupported klog flags, see the Command line tool reference.
>
> Not all logs are guaranteed to be written in JSON format (for example, during process start). If you intend to parse logs, make sure you can handle log lines that are not JSON as well.
>
> Field names and JSON serialization are subject to change.

The `--logging-format=json` flag changes the format of logs from klog native format to JSON format. Example of JSON log format (pretty printed):

```
{
   "ts": 1580306777.04728,
   "v": 4,
   "msg": "Pod status updated",
   "pod":{
      "name": "nginx-1",
      "namespace": "default"
   },
   "status": "ready"
}
```

Keys with special meaning:

- `ts` - timestamp as Unix time (required, float)
- `v` - verbosity (required, int, default 0)
- `err` - error string (optional, string)
- `msg` - message (required, string)

List of components currently supporting JSON format:

- kube-controller-manager
- kube-apiserver
- kube-scheduler
- kubelet

Log sanitization

FEATURE STATE: `Kubernetes v1.20 [alpha]`

> **Warning:** Log sanitization might incur significant computation overhead and therefore should not be enabled in production.

The `--experimental-logging-sanitization` flag enables the klog sanitization filter. If enabled all log arguments are inspected for fields tagged as sensitive data (e.g. passwords, keys, tokens) and logging of these fields will be prevented.

List of components currently supporting log sanitization:

- kube-controller-manager
- kube-apiserver
- kube-scheduler
- kubelet

> **Note:** The Log sanitization filter does not prevent user workload logs from leaking sensitive data.

Log verbosity level

The `-v` flag controls log verbosity. Increasing the value increases the number of logged events. Decreasing the value decreases the number of logged events. Increasing verbosity settings logs increasingly less severe events. A verbosity setting of 0 logs only critical events.

Log location

There are two types of system components: those that run in a container and those that do not run in a container. For example:

- The Kubernetes scheduler and kube-proxy run in a container.
- The kubelet and container runtime, for example Docker, do not run in containers.

On machines with systemd, the kubelet and container runtime write to journald. Otherwise, they write to `.log` files in the `/var/log` directory. System components inside containers always write to `.log` files in the `/var/log` directory, bypassing the default logging mechanism. Similar to the container logs, you should rotate system component logs in the `/var/log` directory. In Kubernetes clusters created by the `kube-up.sh` script, log rotation is configured by the `logrotate` tool. The `logrotate` tool rotates logs daily, or once the log size is greater than 100MB.

What's next

- Read about the Kubernetes Logging Architecture
- Read about Structured Logging
- Read about the Conventions for logging severity

7 - Garbage collection for container images

Garbage collection is a helpful function of kubelet that will clean up unused images and unused containers. Kubelet will perform garbage collection for containers every minute and garbage collection for images every five minutes.

External garbage collection tools are not recommended as these tools can potentially break the behavior of kubelet by removing containers expected to exist.

Image Collection

Kubernetes manages lifecycle of all images through imageManager, with the cooperation of cadvisor.

The policy for garbage collecting images takes two factors into consideration: `HighThresholdPercent` and `LowThresholdPercent`. Disk usage above the high threshold will trigger garbage collection. The garbage collection will delete least recently used images until the low threshold has been met.

Container Collection

The policy for garbage collecting containers considers three user-defined variables. `MinAge` is the minimum age at which a container can be garbage collected. `MaxPerPodContainer` is the maximum number of dead containers every single pod (UID, container name) pair is allowed to have. `MaxContainers` is the maximum number of total dead containers. These variables can be individually disabled by setting `MinAge` to zero and setting `MaxPerPodContainer` and `MaxContainers` respectively to less than zero.

Kubelet will act on containers that are unidentified, deleted, or outside of the boundaries set by the previously mentioned flags. The oldest containers will generally be removed first. `MaxPerPodContainer` and `MaxContainer` may potentially conflict with each other in situations where retaining the maximum number of containers per pod (`MaxPerPodContainer`) would go outside the allowable range of global dead containers (`MaxContainers`). `MaxPerPodContainer` would be adjusted in this situation: A worst case scenario would be to downgrade `MaxPerPodContainer` to 1 and evict the oldest containers. Additionally, containers owned by pods that have been deleted are removed once they are older than `MinAge`.

Containers that are not managed by kubelet are not subject to container garbage collection.

User Configuration

You can adjust the following thresholds to tune image garbage collection with the following kubelet flags :

1. `image-gc-high-threshold`, the percent of disk usage which triggers image garbage collection. Default is 85%.
2. `image-gc-low-threshold`, the percent of disk usage to which image garbage collection attempts to free. Default is 80%.

You can customize the garbage collection policy through the following kubelet flags:

1. `minimum-container-ttl-duration`, minimum age for a finished container before it is garbage collected. Default is 0 minute, which means every finished container will be garbage collected.
2. `maximum-dead-containers-per-container`, maximum number of old instances to be retained per container. Default is 1.
3. `maximum-dead-containers`, maximum number of old instances of containers to retain globally. Default is -1, which means there is no global limit.

Containers can potentially be garbage collected before their usefulness has expired. These containers can contain logs and other data that can be useful for troubleshooting. A sufficiently large value for `maximum-dead-containers-per-container` is highly recommended to allow at least 1 dead container to be retained per expected container. A larger value for `maximum-dead-containers` is also recommended for a similar reason. See [this issue] for more details.

Deprecation

Some kubelet Garbage Collection features in this doc will be replaced by kubelet eviction in the future.

Including:

Existing Flag	New Flag	Rationale
`--image-gc-high-threshold`	`--eviction-hard` or `--eviction-soft`	existing eviction signals can trigger image garbage collection
`--image-gc-low-threshold`	`--eviction-minimum-reclaim`	eviction reclaims achieve the same behavior
`--maximum-dead-containers`		deprecated once old logs are stored outside of container's context
`--maximum-dead-containers-per-container`		deprecated once old logs are stored outside of container's context
`--minimum-container-ttl-duration`		deprecated once old logs are stored outside of container's context
`--low-diskspace-threshold-mb`	`--eviction-hard` or `eviction-soft`	eviction generalizes disk thresholds to other resources
`--outofdisk-transition-frequency`	`--eviction-pressure-transition-period`	eviction generalizes disk pressure transition to other resources

What's next

See [Configuring Out Of Resource Handling] for more details.

8 - Proxies in Kubernetes

This page explains proxies used with Kubernetes.

Proxies

There are several different proxies you may encounter when using Kubernetes:

1. The kubectl proxy:
    - runs on a user's desktop or in a pod
    - proxies from a localhost address to the Kubernetes apiserver
    - client to proxy uses HTTP
    - proxy to apiserver uses HTTPS
    - locates apiserver
    - adds authentication headers
2. The apiserver proxy:
    - is a bastion built into the apiserver
    - connects a user outside of the cluster to cluster IPs which otherwise might not be reachable
    - runs in the apiserver processes
    - client to proxy uses HTTPS (or http if apiserver so configured)
    - proxy to target may use HTTP or HTTPS as chosen by proxy using available information
    - can be used to reach a Node, Pod, or Service
    - does load balancing when used to reach a Service
3. The kube proxy:
    - runs on each node
    - proxies UDP, TCP and SCTP
    - does not understand HTTP
    - provides load balancing
    - is only used to reach services
4. A Proxy/Load-balancer in front of apiserver(s):
    - existence and implementation varies from cluster to cluster (e.g. nginx)
    - sits between all clients and one or more apiservers
    - acts as load balancer if there are several apiservers.
5. Cloud Load Balancers on external services:
    - are provided by some cloud providers (e.g. AWS ELB, Google Cloud Load Balancer)
    - are created automatically when the Kubernetes service has type `LoadBalancer`
    - usually supports UDP/TCP only
    - SCTP support is up to the load balancer implementation of the cloud provider
    - implementation varies by cloud provider.

Kubernetes users will typically not need to worry about anything other than the first two types. The cluster admin will typically ensure that the latter types are setup correctly.

Requesting redirects

Proxies have replaced redirect capabilities. Redirects have been deprecated.

9 - API Priority and Fairness

FEATURE STATE: `Kubernetes v1.20 [beta]`

Controlling the behavior of the Kubernetes API server in an overload situation is a key task for cluster administrators. The kube-apiserver has some controls available (i.e. the `--max-requests-inflight` and `--max-mutating-requests-inflight` command-line flags) to limit the amount of outstanding work that will be accepted, preventing a flood of inbound requests from overloading and potentially crashing the API server, but these flags are not enough to ensure that the most important requests get through in a period of high traffic.

The API Priority and Fairness feature (APF) is an alternative that improves upon aforementioned max-inflight limitations. APF classifies and isolates requests in a more fine-grained way. It also introduces a limited amount of queuing, so that no requests are rejected in cases of very brief bursts. Requests are dispatched from queues using a fair queuing technique so that, for example, a poorly-behaved controller need not starve others (even at the same priority level).

> **Caution:** Requests classified as "long-running" — primarily watches — are not subject to the API Priority and Fairness filter. This is also true for the `--max-requests-inflight` flag without the API Priority and Fairness feature enabled.

Enabling/Disabling API Priority and Fairness

The API Priority and Fairness feature is controlled by a feature gate and is enabled by default. See Feature Gates for a general explanation of feature gates and how to enable and disable them. The name of the feature gate for APF is "APIPriorityAndFairness". This feature also involves an API Group with: (a) a `v1alpha1` version, disabled by default, and (b) a `v1beta1` version, enabled by default. You can disable the feature gate and API group v1beta1 version by adding the following command-line flags to your `kube-apiserver` invocation:

```
kube-apiserver \
--feature-gates=APIPriorityAndFairness=false \
--runtime-config=flowcontrol.apiserver.k8s.io/v1beta1=false \
 # …and other flags as usual
```

Alternatively, you can enable the v1alpha1 version of the API group with `--runtime-config=flowcontrol.apiserver.k8s.io/v1alpha1=true`.

The command-line flag `--enable-priority-and-fairness=false` will disable the API Priority and Fairness feature, even if other flags have enabled it.

Concepts

There are several distinct features involved in the API Priority and Fairness feature. Incoming requests are classified by attributes of the request using *FlowSchemas*, and assigned to priority levels. Priority levels add a degree of isolation by maintaining separate concurrency limits, so that requests assigned to different priority levels cannot starve each other. Within a priority level, a fair-queuing algorithm prevents requests from different *flows* from starving each other, and allows for requests to be queued to prevent bursty traffic from causing failed requests when the average load is acceptably low.

Priority Levels

Without APF enabled, overall concurrency in the API server is limited by the `kube-apiserver` flags `--max-requests-inflight` and `--max-mutating-requests-inflight`. With APF enabled, the concurrency limits defined by these flags are summed and then the sum is divided up among a configurable set of *priority levels*. Each incoming request is assigned to a single priority level, and each priority level will only dispatch as many concurrent requests as its configuration allows.

The default configuration, for example, includes separate priority levels for leader-election requests, requests from built-in controllers, and requests from Pods. This means that an ill-behaved Pod that floods the API server with requests cannot prevent leader election or actions by the built-in controllers from succeeding.

Queuing

Even within a priority level there may be a large number of distinct sources of traffic. In an overload situation, it is valuable to prevent one stream of requests from starving others (in particular, in the relatively common case of a single buggy client flooding the kube-apiserver with requests, that buggy client would ideally not have much measurable impact on other clients at all). This is handled by use of a fair-queuing algorithm to process requests that are assigned the same priority level. Each request is assigned to a *flow*, identified by the name of the matching FlowSchema plus a *flow distinguisher* — which is either the requesting user, the target resource's namespace, or nothing — and the system attempts to give approximately equal weight to requests in different flows of the same priority level.

After classifying a request into a flow, the API Priority and Fairness feature then may assign the request to a queue. This assignment uses a technique known as shuffle sharding, which makes relatively efficient use of queues to insulate low-intensity flows from high-intensity flows.

The details of the queuing algorithm are tunable for each priority level, and allow administrators to trade off memory use, fairness (the property that independent flows will all make progress when total traffic exceeds capacity), tolerance for bursty traffic, and the added latency induced by queuing.

Exempt requests

Some requests are considered sufficiently important that they are not subject to any of the limitations imposed by this feature. These exemptions prevent an improperly-configured flow control configuration from totally disabling an API server.

Defaults

The Priority and Fairness feature ships with a suggested configuration that should suffice for experimentation; if your cluster is likely to experience heavy load then you should consider what configuration will work best. The suggested configuration groups requests into five priority classes:

- The `system` priority level is for requests from the `system:nodes` group, i.e. Kubelets, which must be able to contact the API server in order for workloads to be able to schedule on them.
- The `leader-election` priority level is for leader election requests from built-in controllers (in particular, requests for `endpoints`, `configmaps`, or `leases` coming from the `system:kube-controller-manager` or `system:kube-scheduler` users and service accounts in the `kube-system` namespace). These are important to isolate from other traffic because failures in leader election cause their controllers to fail and restart, which in turn causes more expensive traffic as the new controllers sync their informers.
- The `workload-high` priority level is for other requests from built-in controllers.
- The `workload-low` priority level is for requests from any other service account, which will typically include all requests from controllers running in Pods.
- The `global-default` priority level handles all other traffic, e.g. interactive `kubectl` commands run by nonprivileged users.

Additionally, there are two PriorityLevelConfigurations and two FlowSchemas that are built in and may not be overwritten:

- The special `exempt` priority level is used for requests that are not subject to flow control at all: they will always be dispatched immediately. The special `exempt` FlowSchema classifies all requests from the `system:masters` group into this priority level. You may define other

FlowSchemas that direct other requests to this priority level, if appropriate.

- The special `catch-all` priority level is used in combination with the special `catch-all` FlowSchema to make sure that every request gets some kind of classification. Typically you should not rely on this catch-all configuration, and should create your own catch-all FlowSchema and PriorityLevelConfiguration (or use the `global-default` configuration that is installed by default) as appropriate. To help catch configuration errors that miss classifying some requests, the mandatory `catch-all` priority level only allows one concurrency share and does not queue requests, making it relatively likely that traffic that only matches the `catch-all` FlowSchema will be rejected with an HTTP 429 error.

Health check concurrency exemption

The suggested configuration gives no special treatment to the health check requests on kube-apiservers from their local kubelets --- which tend to use the secured port but supply no credentials. With the suggested config, these requests get assigned to the `global-default` FlowSchema and the corresponding `global-default` priority level, where other traffic can crowd them out.

If you add the following additional FlowSchema, this exempts those requests from rate limiting.

> **Caution:** Making this change also allows any hostile party to then send health-check requests that match this FlowSchema, at any volume they like. If you have a web traffic filter or similar external security mechanism to protect your cluster's API server from general internet traffic, you can configure rules to block any health check requests that originate from outside your cluster.

priority-and-fairness/health-for-strangers.yaml

```yaml
apiVersion: flowcontrol.apiserver.k8s.io/v1beta1
kind: FlowSchema
metadata:
  name: health-for-strangers
spec:
  matchingPrecedence: 1000
  priorityLevelConfiguration:
    name: exempt
  rules:
  - nonResourceRules:
    - nonResourceURLs:
      - "/healthz"
      - "/livez"
      - "/readyz"
      verbs:
      - "*"
    subjects:
    - kind: Group
      group:
        name: system:unauthenticated
```

Resources

The flow control API involves two kinds of resources. PriorityLevelConfigurations define the available isolation classes, the share of the available concurrency budget that each can handle, and allow for fine-tuning queuing behavior. FlowSchemas are used to classify individual inbound requests, matching each to a single PriorityLevelConfiguration. There is also a `v1alpha1` version of the same API group, and it has the same Kinds with the same syntax and semantics.

PriorityLevelConfiguration

A PriorityLevelConfiguration represents a single isolation class. Each PriorityLevelConfiguration has an independent limit on the number of outstanding requests, and limitations on the number of queued requests.

Concurrency limits for PriorityLevelConfigurations are not specified in absolute number of requests, but rather in "concurrency shares." The total concurrency limit for the API Server is distributed among the existing PriorityLevelConfigurations in proportion with these shares. This allows a cluster administrator to scale up or down the total amount of traffic to a server by restarting `kube-apiserver` with a different value for `--max-requests-inflight` (or `--max-mutating-requests-inflight`), and all PriorityLevelConfigurations will see their maximum allowed concurrency go up (or down) by the same fraction.

> **Caution:** With the Priority and Fairness feature enabled, the total concurrency limit for the server is set to the sum of `--max-requests-inflight` and `--max-mutating-requests-inflight`. There is no longer any distinction made between mutating and non-mutating requests; if you want to treat them separately for a given resource, make separate FlowSchemas that match the mutating and non-mutating verbs respectively.

When the volume of inbound requests assigned to a single PriorityLevelConfiguration is more than its permitted concurrency level, the `type` field of its specification determines what will happen to extra requests. A type of `Reject` means that excess traffic will immediately be rejected with an HTTP 429 (Too Many Requests) error. A type of `Queue` means that requests above the threshold will be queued, with the shuffle sharding and fair queuing techniques used to balance progress between request flows.

The queuing configuration allows tuning the fair queuing algorithm for a priority level. Details of the algorithm can be read in the enhancement proposal, but in short:

- Increasing `queues` reduces the rate of collisions between different flows, at the cost of increased memory usage. A value of 1 here effectively disables the fair-queuing logic, but still allows requests to be queued.

- Increasing `queueLengthLimit` allows larger bursts of traffic to be sustained without dropping any requests, at the cost of increased latency and memory usage.

- Changing `handSize` allows you to adjust the probability of collisions between different flows and the overall concurrency available to a single flow in an overload situation.

  > **Note:** A larger `handSize` makes it less likely for two individual flows to collide (and therefore for one to be able to starve the other), but more likely that a small number of flows can dominate the apiserver. A larger `handSize` also potentially increases the amount of latency that a single high-traffic flow can cause. The maximum number of queued requests possible from a single flow is `handSize * queueLengthLimit`.

Following is a table showing an interesting collection of shuffle sharding configurations, showing for each the probability that a given mouse (low-intensity flow) is squished by the elephants (high-intensity flows) for an illustrative collection of numbers of elephants. See https://play.golang.org/p/Gi0PLgVHiUg , which computes this table.

HandSize	Queues	1 elephant	4 elephants	16 elephants
12	32	4.428838398950118e-09	0.11431348830099144	0.9935089607656024
10	32	1.550093439632541e-08	0.0626479840223545	0.9753101519027554
10	64	6.601827268370426e-12	0.00045571320990370776	0.49999929150089345
9	64	3.6310049976037345e-11	0.00045501212304112273	0.4282314876454858

HandSize	Queues	1 elephant	4 elephants	16 elephants
8	64	2.25929199850899e-10	0.0004886697053040446	0.35935114681123076
8	128	6.994461389026097e-13	3.4055790161620863e-06	0.02746173137155063
7	128	1.0579122850901972e-11	6.960839379258192e-06	0.02406157386340147
7	256	7.597695465552631e-14	6.728547142019406e-08	0.0006709661542533682
6	256	2.7134626662687968e-12	2.9516464018476436e-07	0.0008895654642000348
6	512	4.116062922897309e-14	4.982983350480894e-09	2.26025764343413e-05
6	1024	6.337324016514285e-16	8.09060164312957e-11	4.517408062903668e-07

FlowSchema

A FlowSchema matches some inbound requests and assigns them to a priority level. Every inbound request is tested against every FlowSchema in turn, starting with those with numerically lowest --- which we take to be the logically highest --- `matchingPrecedence` and working onward. The first match wins.

> **Caution:** Only the first matching FlowSchema for a given request matters. If multiple FlowSchemas match a single inbound request, it will be assigned based on the one with the highest `matchingPrecedence`. If multiple FlowSchemas with equal `matchingPrecedence` match the same request, the one with lexicographically smaller `name` will win, but it's better not to rely on this, and instead to ensure that no two FlowSchemas have the same `matchingPrecedence`.

A FlowSchema matches a given request if at least one of its `rules` matches. A rule matches if at least one of its `subjects` *and* at least one of its `resourceRules` or `nonResourceRules` (depending on whether the incoming request is for a resource or non-resource URL) matches the request.

For the `name` field in subjects, and the `verbs`, `apiGroups`, `resources`, `namespaces`, and `nonResourceURLs` fields of resource and non-resource rules, the wildcard `*` may be specified to match all values for the given field, effectively removing it from consideration.

A FlowSchema's `distinguisherMethod.type` determines how requests matching that schema will be separated into flows. It may be either `ByUser`, in which case one requesting user will not be able to starve other users of capacity, or `ByNamespace`, in which case requests for resources in one namespace will not be able to starve requests for resources in other namespaces of capacity, or it may be blank (or `distinguisherMethod` may be omitted entirely), in which case all requests matched by this FlowSchema will be considered part of a single flow. The correct choice for a given FlowSchema depends on the resource and your particular environment.

Diagnostics

Every HTTP response from an API server with the priority and fairness feature enabled has two extra headers: `X-Kubernetes-PF-FlowSchema-UID` and `X-Kubernetes-PF-PriorityLevel-UID`, noting the flow schema that matched the request and the priority level to which it was assigned, respectively. The API objects' names are not included in these headers in case the requesting user does not have permission to view them, so when debugging you can use a command like

```
kubectl get flowschemas -o custom-columns="uid:{metadata.uid},name:{metadata.name}"
kubectl get prioritylevelconfigurations -o custom-columns="uid:{metadata.uid},name:
```

to get a mapping of UIDs to names for both FlowSchemas and PriorityLevelConfigurations.

Observability

Metrics

> **Note:** In versions of Kubernetes before v1.20, the labels `flow_schema` and `priority_level` were inconsistently named `flowSchema` and `priorityLevel`, respectively. If you're running Kubernetes versions v1.19 and earlier, you should refer to the documentation for your version.

When you enable the API Priority and Fairness feature, the kube-apiserver exports additional metrics. Monitoring these can help you determine whether your configuration is inappropriately throttling important traffic, or find poorly-behaved workloads that may be harming system health.

- `apiserver_flowcontrol_rejected_requests_total` is a counter vector (cumulative since server start) of requests that were rejected, broken down by the labels `flow_schema` (indicating the one that matched the request), `priority_level` (indicating the one to which the request was assigned), and `reason`. The `reason` label will be have one of the following values:
    - `queue-full`, indicating that too many requests were already queued,
    - `concurrency-limit`, indicating that the PriorityLevelConfiguration is configured to reject rather than queue excess requests, or
    - `time-out`, indicating that the request was still in the queue when its queuing time limit expired.
- `apiserver_flowcontrol_dispatched_requests_total` is a counter vector (cumulative since server start) of requests that began executing, broken down by the labels `flow_schema` (indicating the one that matched the request) and `priority_level` (indicating the one to which the request was assigned).
- `apiserver_current_inqueue_requests` is a gauge vector of recent high water marks of the number of queued requests, grouped by a label named `request_kind` whose value is `mutating` or `readOnly`. These high water marks describe the largest number seen in the one second window most recently completed. These complement the older `apiserver_current_inflight_requests` gauge vector that holds the last window's high water mark of number of requests actively being served.
- `apiserver_flowcontrol_read_vs_write_request_count_samples` is a histogram vector of observations of the then-current number of requests, broken down by the labels `phase` (which takes on the values `waiting` and `executing`) and `request_kind` (which takes on the values `mutating` and `readOnly`). The observations are made periodically at a high rate.
- `apiserver_flowcontrol_read_vs_write_request_count_watermarks` is a histogram vector of high or low water marks of the number of requests broken down by the labels `phase` (which takes on the values `waiting` and `executing`) and `request_kind` (which takes on the values `mutating` and `readOnly`); the label `mark` takes on values `high` and `low`. The water marks are accumulated over windows bounded by the times when an observation was added to `apiserver_flowcontrol_read_vs_write_request_count_samples`. These water marks show the range of values that occurred between samples.
- `apiserver_flowcontrol_current_inqueue_requests` is a gauge vector holding the instantaneous number of queued (not executing) requests, broken down by the labels `priority_level` and `flow_schema`.

- `apiserver_flowcontrol_current_executing_requests` is a gauge vector holding the instantaneous number of executing (not waiting in a queue) requests, broken down by the labels `priority_level` and `flow_schema`.

- `apiserver_flowcontrol_priority_level_request_count_samples` is a histogram vector of observations of the then-current number of requests broken down by the labels `phase` (which takes on the values `waiting` and `executing`) and `priority_level`. Each histogram gets observations taken periodically, up through the last activity of the relevant sort. The observations are made at a high rate.

- `apiserver_flowcontrol_priority_level_request_count_watermarks` is a histogram vector of high or low water marks of the number of requests broken down by the labels `phase` (which takes on the values `waiting` and `executing`) and `priority_level`; the label `mark` takes on values `high` and `low`. The water marks are accumulated over windows bounded by the times when an observation was added to `apiserver_flowcontrol_priority_level_request_count_samples`. These water marks show the range of values that occurred between samples.

- `apiserver_flowcontrol_request_queue_length_after_enqueue` is a histogram vector of queue lengths for the queues, broken down by the labels `priority_level` and `flow_schema`, as sampled by the enqueued requests. Each request that gets queued contributes one sample to its histogram, reporting the length of the queue immediately after the request was added. Note that this produces different statistics than an unbiased survey would.

  > **Note:** An outlier value in a histogram here means it is likely that a single flow (i.e., requests by one user or for one namespace, depending on configuration) is flooding the API server, and being throttled. By contrast, if one priority level's histogram shows that all queues for that priority level are longer than those for other priority levels, it may be appropriate to increase that PriorityLevelConfiguration's concurrency shares.

- `apiserver_flowcontrol_request_concurrency_limit` is a gauge vector holding the computed concurrency limit (based on the API server's total concurrency limit and PriorityLevelConfigurations' concurrency shares), broken down by the label `priority_level`.

- `apiserver_flowcontrol_request_wait_duration_seconds` is a histogram vector of how long requests spent queued, broken down by the labels `flow_schema` (indicating which one matched the request), `priority_level` (indicating the one to which the request was assigned), and `execute` (indicating whether the request started executing).

  > **Note:** Since each FlowSchema always assigns requests to a single PriorityLevelConfiguration, you can add the histograms for all the FlowSchemas for one priority level to get the effective histogram for requests assigned to that priority level.

- `apiserver_flowcontrol_request_execution_seconds` is a histogram vector of how long requests took to actually execute, broken down by the labels `flow_schema` (indicating which one matched the request) and `priority_level` (indicating the one to which the request was assigned).

Debug endpoints

When you enable the API Priority and Fairness feature, the kube-apiserver serves the following additional paths at its HTTP[S] ports.

- `/debug/api_priority_and_fairness/dump_priority_levels` - a listing of all the priority levels and the current state of each. You can fetch like this:

  ```
  kubectl get --raw /debug/api_priority_and_fairness/dump_priority_levels
  ```

  The output is similar to this:

```
PriorityLevelName, ActiveQueues, IsIdle, IsQuiescing, WaitingRequests, Executin
workload-low,      0,            true,   false,       0,               0,
global-default,    0,            true,   false,       0,               0,
exempt,            <none>,       <none>, <none>,      <none>,          <none>,
catch-all,         0,            true,   false,       0,               0,
system,            0,            true,   false,       0,               0,
leader-election,   0,            true,   false,       0,               0,
workload-high,     0,            true,   false,       0,               0,
```

- `/debug/api_priority_and_fairness/dump_queues` - a listing of all the queues and their current state. You can fetch like this:

```
kubectl get --raw /debug/api_priority_and_fairness/dump_queues
```

The output is similar to this:

```
PriorityLevelName, Index,  PendingRequests, ExecutingRequests, VirtualStart,
workload-high,     0,      0,                0,                 0.0000,
workload-high,     1,      0,                0,                 0.0000,
workload-high,     2,      0,                0,                 0.0000,
...
leader-election,   14,     0,                0,                 0.0000,
leader-election,   15,     0,                0,                 0.0000,
```

- `/debug/api_priority_and_fairness/dump_requests` - a listing of all the requests that are currently waiting in a queue. You can fetch like this:

```
kubectl get --raw /debug/api_priority_and_fairness/dump_requests
```

The output is similar to this:

```
PriorityLevelName, FlowSchemaName, QueueIndex, RequestIndexInQueue, FlowDisting
exempt,            <none>,         <none>,     <none>,              <none>,
system,            system-nodes,   12,         0,                   system:node
```

In addition to the queued requests, the output includes one phantom line for each priority level that is exempt from limitation.

You can get a more detailed listing with a command like this:

```
kubectl get --raw '/debug/api_priority_and_fairness/dump_requests?includeReques
```

The output is similar to this:

```
PriorityLevelName, FlowSchemaName, QueueIndex, RequestIndexInQueue, FlowDisting
system,            system-nodes,   12,         0,                   system:node
system,            system-nodes,   12,         1,                   system:node
```

What's next

For background information on design details for API priority and fairness, see the enhancement proposal. You can make suggestions and feature requests via SIG API Machinery or the feature's slack channel.

10 - Installing Addons

> **Caution:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects. This page follows CNCF website guidelines by listing projects alphabetically. To add a project to this list, read the content guide before submitting a change.

Add-ons extend the functionality of Kubernetes.

This page lists some of the available add-ons and links to their respective installation instructions.

Networking and Network Policy

- ACI provides integrated container networking and network security with Cisco ACI.
- Calico is a networking and network policy provider. Calico supports a flexible set of networking options so you can choose the most efficient option for your situation, including non-overlay and overlay networks, with or without BGP. Calico uses the same engine to enforce network policy for hosts, pods, and (if using Istio & Envoy) applications at the service mesh layer.
- Canal unites Flannel and Calico, providing networking and network policy.
- Cilium is a L3 network and network policy plugin that can enforce HTTP/API/L7 policies transparently. Both routing and overlay/encapsulation mode are supported, and it can work on top of other CNI plugins.
- CNI-Genie enables Kubernetes to seamlessly connect to a choice of CNI plugins, such as Calico, Canal, Flannel, Romana, or Weave.
- Contiv provides configurable networking (native L3 using BGP, overlay using vxlan, classic L2, and Cisco-SDN/ACI) for various use cases and a rich policy framework. Contiv project is fully open sourced. The installer provides both kubeadm and non-kubeadm based installation options.
- Contrail, based on Tungsten Fabric, is an open source, multi-cloud network virtualization and policy management platform. Contrail and Tungsten Fabric are integrated with orchestration systems such as Kubernetes, OpenShift, OpenStack and Mesos, and provide isolation modes for virtual machines, containers/pods and bare metal workloads.
- Flannel is an overlay network provider that can be used with Kubernetes.
- Knitter is a plugin to support multiple network interfaces in a Kubernetes pod.
- Multus is a Multi plugin for multiple network support in Kubernetes to support all CNI plugins (e.g. Calico, Cilium, Contiv, Flannel), in addition to SRIOV, DPDK, OVS-DPDK and VPP based workloads in Kubernetes.
- OVN-Kubernetes is a networking provider for Kubernetes based on OVN (Open Virtual Network), a virtual networking implementation that came out of the Open vSwitch (OVS) project. OVN-Kubernetes provides an overlay based networking implementation for Kubernetes, including an OVS based implementation of load balancing and network policy.
- OVN4NFV-K8S-Plugin is OVN based CNI controller plugin to provide cloud native based Service function chaining(SFC), Multiple OVN overlay networking, dynamic subnet creation, dynamic creation of virtual networks, VLAN Provider network, Direct provider network and pluggable with other Multi-network plugins, ideal for edge based cloud native workloads in Multi-cluster networking
- NSX-T Container Plug-in (NCP) provides integration between VMware NSX-T and container orchestrators such as Kubernetes, as well as integration between NSX-T and container-based CaaS/PaaS platforms such as Pivotal Container Service (PKS) and OpenShift.
- Nuage is an SDN platform that provides policy-based networking between Kubernetes Pods and non-Kubernetes environments with visibility and security monitoring.
- Romana is a Layer 3 networking solution for pod networks that also supports the NetworkPolicy API. Kubeadm add-on installation details available here.
- Weave Net provides networking and network policy, will carry on working on both sides of a network partition, and does not require an external database.

Service Discovery

- CoreDNS is a flexible, extensible DNS server which can be installed as the in-cluster DNS for pods.

Visualization & Control

- Dashboard is a dashboard web interface for Kubernetes.
- Weave Scope is a tool for graphically visualizing your containers, pods, services etc. Use it in conjunction with a Weave Cloud account or host the UI yourself.

Infrastructure

- KubeVirt is an add-on to run virtual machines on Kubernetes. Usually run on bare-metal clusters.

Legacy Add-ons

There are several other add-ons documented in the deprecated cluster/addons directory.

Well-maintained ones should be linked to here. PRs welcome!