Getting started

This section lists the different ways to set up and run Kubernetes. When you install Kubernetes, choose an installation type based on: ease of maintenance, security, control, available resources, and expertise required to operate and manage a cluster.

You can download Kubernetes to deploy a Kubernetes cluster on a local machine, into the cloud, or for your own datacenter.

If you don't want to manage a Kubernetes cluster yourself, you could pick a managed service, including certified platforms. There are also other standardized and custom solutions across a wide range of cloud and bare metal environments.

Learning environment

If you're learning Kubernetes, use the tools supported by the Kubernetes community, or tools in the ecosystem to set up a Kubernetes cluster on a local machine. See Install tools.

Production environment

When evaluating a solution for a production environment, consider which aspects of operating a Kubernetes cluster (or *abstractions*) you want to manage yourself and which you prefer to hand off to a provider.

For a cluster you're managing yourself, the officially supported tool for deploying Kubernetes is kubeadm.

What's next

- Download Kubernetes

- Download and install tools including `kubectl`
- Select a container runtime for your new cluster
- Learn about best practices for cluster setup

Kubernetes is designed for its control plane to run on Linux. Within your cluster you can run applications on Linux or other operating systems, including Windows.

- Learn to set up clusters with Windows nodes

1 - Learning environment

2 - Production environment

Create a production-quality Kubernetes cluster

A production-quality Kubernetes cluster requires planning and preparation. If your Kubernetes cluster is to run critical workloads, it must be configured to be resilient. This page explains steps you can take to set up a production-ready cluster, or to promote an existing cluster for production use. If you're already familiar with production setup and want the links, skip to What's next.

Production considerations

Typically, a production Kubernetes cluster environment has more requirements than a personal learning, development, or test environment Kubernetes. A production environment may require secure access by many users, consistent availability, and the resources to adapt to changing demands.

As you decide where you want your production Kubernetes environment to live (on premises or in a cloud) and the amount of management you want to take on or hand to others, consider how your requirements for a Kubernetes cluster are influenced by the following issues:

- *Availability*: A single-machine Kubernetes learning environment has a single point of failure. Creating a highly available cluster means considering:
  - Separating the control plane from the worker nodes.
  - Replicating the control plane components on multiple nodes.
  - Load balancing traffic to the cluster's API server.
  - Having enough worker nodes available, or able to quickly become available, as changing workloads warrant it.
- *Scale*: If you expect your production Kubernetes environment to receive a stable amount of demand, you might be able to set up for the capacity you need and be done. However, if you expect demand to grow over time or change dramatically based on things like season or special events, you need to plan how to scale to relieve increased pressure from more requests to the control plane and worker nodes or scale down to reduce unused resources.
- *Security and access management*: You have full admin privileges on your own Kubernetes learning cluster. But shared clusters with important workloads, and more than one or two users, require a more refined approach to who and what can access cluster resources. You can use role-based access control (RBAC) and other security mechanisms to make sure that users and workloads can get access to the resources they need, while keeping workloads, and the cluster itself, secure. You can set limits on the resources that users and workloads can access by managing policies and container resources.

Before building a Kubernetes production environment on your own, consider handing off some or all of this job to Turnkey Cloud Solutions providers or other Kubernetes Partners. Options include:

- *Serverless*: Just run workloads on third-party equipment without managing a cluster at all. You will be charged for things like CPU usage, memory, and disk requests.
- *Managed control plane*: Let the provider manage the scale and availability of the cluster's control plane, as well as handle patches and upgrades.
- *Managed worker nodes*: Configure pools of nodes to meet your needs, then the provider makes sure those nodes are available and ready to implement upgrades when needed.
- *Integration*: There are providers that integrate Kubernetes with other services you may need, such as storage, container registries, authentication methods, and development tools.

Whether you build a production Kubernetes cluster yourself or work with partners, review the following sections to evaluate your needs as they relate to your cluster's *control plane*, *worker nodes*, *user access*, and *workload resources*.

Production cluster setup

In a production-quality Kubernetes cluster, the control plane manages the cluster from services that can be spread across multiple computers in different ways. Each worker node, however, represents a single entity that is configured to run Kubernetes pods.

Production control plane

The simplest Kubernetes cluster has the entire control plane and worker node services running on the same machine. You can grow that environment by adding worker nodes, as reflected in the diagram illustrated in Kubernetes Components. If the cluster is meant to be available for a short period of time, or can be discarded if something goes seriously wrong, this might meet your needs.

If you need a more permanent, highly available cluster, however, you should consider ways of extending the control plane. By design, one-machine control plane services running on a single machine are not highly available. If keeping the cluster up and running and ensuring that it can be repaired if something goes wrong is important, consider these steps:

- *Choose deployment tools*: You can deploy a control plane using tools such as kubeadm, kops, and kubespray. See Installing Kubernetes with deployment tools to learn tips for production-quality deployments using each of those deployment methods. Different Container Runtimes are available to use with your deployments.
- *Manage certificates*: Secure communications between control plane services are implemented using certificates. Certificates are automatically generated during deployment or you can generate them using your own certificate authority. See PKI certificates and requirements for details.
- *Configure load balancer for apiserver*: Configure a load balancer to distribute external API requests to the apiserver service instances running on different nodes. See Create an External Load Balancer for details.
- *Separate and backup etcd service*: The etcd services can either run on the same machines as other control plane services or run on separate machines, for extra security and availability. Because etcd stores cluster configuration data, backing up the etcd database should be done regularly to ensure that you can repair that database if needed. See the etcd FAQ for details on configuring and using etcd. See Operating etcd clusters for Kubernetes and Set up a High Availability etcd cluster with kubeadm for details.
- *Create multiple control plane systems*: For high availability, the control plane should not be limited to a single machine. If the control plane services are run by an init service (such as systemd), each service should run on at least three machines. However, running control plane services as pods in Kubernetes ensures that the replicated number of services that you request will always be available. The scheduler should be fault tolerant, but not highly available. Some deployment tools set up Raft consensus algorithm to do leader election of Kubernetes services. If the primary goes away, another service elects itself and take over.
- *Span multiple zones*: If keeping your cluster available at all times is critical, consider creating a cluster that runs across multiple data centers, referred to as zones in cloud environments. Groups of zones are referred to as regions. By spreading a cluster across multiple zones in the same region, it can improve the chances that your cluster will continue to function even if one zone becomes unavailable. See Running in multiple zones for details.
- *Manage on-going features*: If you plan to keep your cluster over time, there are tasks you need to do to maintain its health and security. For example, if you installed with kubeadm, there are instructions to help you with Certificate Management and Upgrading kubeadm clusters. See Administer a Cluster for a longer list of Kubernetes administrative tasks.

To learn about available options when you run control plane services, see kube-apiserver, kube-controller-manager, and kube-scheduler component pages. For highly available control plane examples, see Options for Highly Available topology, Creating Highly Available clusters

with kubeadm, and Operating etcd clusters for Kubernetes. See Backing up an etcd cluster for information on making an etcd backup plan.

Production worker nodes

Production-quality workloads need to be resilient and anything they rely on needs to be resilient (such as CoreDNS). Whether you manage your own control plane or have a cloud provider do it for you, you still need to consider how you want to manage your worker nodes (also referred to simply as *nodes*).

- *Configure nodes*: Nodes can be physical or virtual machines. If you want to create and manage your own nodes, you can install a supported operating system, then add and run the appropriate Node services. Consider:
  - The demands of your workloads when you set up nodes by having appropriate memory, CPU, and disk speed and storage capacity available.
  - Whether generic computer systems will do or you have workloads that need GPU processors, Windows nodes, or VM isolation.
- *Validate nodes*: See Valid node setup for information on how to ensure that a node meets the requirements to join a Kubernetes cluster.
- *Add nodes to the cluster*: If you are managing your own cluster you can add nodes by setting up your own machines and either adding them manually or having them register themselves to the cluster's apiserver. See the Nodes section for information on how to set up Kubernetes to add nodes in these ways.
- *Add Windows nodes to the cluster*: Kubernetes offers support for Windows worker nodes, allowing you to run workloads implemented in Windows containers. See Windows in Kubernetes for details.
- *Scale nodes*: Have a plan for expanding the capacity your cluster will eventually need. See Considerations for large clusters to help determine how many nodes you need, based on the number of pods and containers you need to run. If you are managing nodes yourself, this can mean purchasing and installing your own physical equipment.
- *Autoscale nodes*: Most cloud providers support Cluster Autoscaler to replace unhealthy nodes or grow and shrink the number of nodes as demand requires. See the Frequently Asked Questions for how the autoscaler works and Deployment for how it is implemented by different cloud providers. For on-premises, there are some virtualization platforms that can be scripted to spin up new nodes based on demand.
- *Set up node health checks*: For important workloads, you want to make sure that the nodes and pods running on those nodes are healthy. Using the Node Problem Detector daemon, you can ensure your nodes are healthy.

Production user management

In production, you may be moving from a model where you or a small group of people are accessing the cluster to where there may potentially be dozens or hundreds of people. In a learning environment or platform prototype, you might have a single administrative account for everything you do. In production, you will want more accounts with different levels of access to different namespaces.

Taking on a production-quality cluster means deciding how you want to selectively allow access by other users. In particular, you need to select strategies for validating the identities of those who try to access your cluster (authentication) and deciding if they have permissions to do what they are asking (authorization):

- *Authentication*: The apiserver can authenticate users using client certificates, bearer tokens, an authenticating proxy, or HTTP basic auth. You can choose which authentication methods you want to use. Using plugins, the apiserver can leverage your organization's existing authentication methods, such as LDAP or Kerberos. See Authentication for a description of these different methods of authenticating Kubernetes users.
- *Authorization*: When you set out to authorize your regular users, you will probably choose between RBAC and ABAC authorization. See Authorization Overview to review

different modes for authorizing user accounts (as well as service account access to your cluster):

- *Role-based access control* (RBAC): Lets you assign access to your cluster by allowing specific sets of permissions to authenticated users. Permissions can be assigned for a specific namespace (Role) or across the entire cluster (ClusterRole). Then using RoleBindings and ClusterRoleBindings, those permissions can be attached to particular users.
- *Attribute-based access control* (ABAC): Lets you create policies based on resource attributes in the cluster and will allow or deny access based on those attributes. Each line of a policy file identifies versioning properties (apiVersion and kind) and a map of spec properties to match the subject (user or group), resource property, non-resource property (/version or /apis), and readonly. See Examples for details.

As someone setting up authentication and authorization on your production Kubernetes cluster, here are some things to consider:

- *Set the authorization mode*: When the Kubernetes API server (kube-apiserver) starts, the supported authentication modes must be set using the *--authorization-mode* flag. For example, that flag in the *kube-adminserver.yaml* file (in */etc/kubernetes/manifests*) could be set to Node,RBAC. This would allow Node and RBAC authorization for authenticated requests.
- *Create user certificates and role bindings (RBAC)*: If you are using RBAC authorization, users can create a CertificateSigningRequest (CSR) that can be signed by the cluster CA. Then you can bind Roles and ClusterRoles to each user. See Certificate Signing Requests for details.
- *Create policies that combine attributes (ABAC)*: If you are using ABAC authorization, you can assign combinations of attributes to form policies to authorize selected users or groups to access particular resources (such as a pod), namespace, or apiGroup. For more information, see Examples.
- *Consider Admission Controllers*: Additional forms of authorization for requests that can come in through the API server include Webhook Token Authentication. Webhooks and other special authorization types need to be enabled by adding Admission Controllers to the API server.

Set limits on workload resources

Demands from production workloads can cause pressure both inside and outside of the Kubernetes control plane. Consider these items when setting up for the needs of your cluster's workloads:

- *Set namespace limits*: Set per-namespace quotas on things like memory and CPU. See Manage Memory, CPU, and API Resources for details. You can also set Hierarchical Namespaces for inheriting limits.
- *Prepare for DNS demand*: If you expect workloads to massively scale up, your DNS service must be ready to scale up as well. See Autoscale the DNS service in a Cluster.
- *Create additional service accounts*: User accounts determine what users can do on a cluster, while a service account defines pod access within a particular namespace. By default, a pod takes on the default service account from its namespace. See Managing Service Accounts for information on creating a new service account. For example, you might want to:
  - Add secrets that a pod could use to pull images from a particular container registry. See Configure Service Accounts for Pods for an example.
  - Assign RBAC permissions to a service account. See ServiceAccount permissions for details.

What's next

- Decide if you want to build your own production Kubernetes or obtain one from available Turnkey Cloud Solutions or Kubernetes Partners.

- If you choose to build your own cluster, plan how you want to handle certificates and set up high availability for features such as etcd and the API server.
- Choose from kubeadm, kops or Kubespray deployment methods.
- Configure user management by determining your Authentication and Authorization methods.
- Prepare for application workloads by setting up resource limits, DNS autoscaling and service accounts.

2.1 - Container runtimes

You need to install a container runtime into each node in the cluster so that Pods can run there. This page outlines what is involved and describes related tasks for setting up nodes.

Kubernetes 1.23 requires that you use a runtime that conforms with the Container Runtime Interface (CRI).

See CRI version support for more information.

This page lists details for using several common container runtimes with Kubernetes, on Linux:

- containerd
- CRI-O
- Docker Engine
- Mirantis Container Runtime

> **Note:** For other operating systems, look for documentation specific to your platform.

Cgroup drivers

Control groups are used to constrain resources that are allocated to processes.

When systemd is chosen as the init system for a Linux distribution, the init process generates and consumes a root control group (`cgroup`) and acts as a cgroup manager. Systemd has a tight integration with cgroups and allocates a cgroup per systemd unit. It's possible to configure your container runtime and the kubelet to use `cgroupfs` . Using `cgroupfs` alongside systemd means that there will be two different cgroup managers.

A single cgroup manager simplifies the view of what resources are being allocated and will by default have a more consistent view of the available and in-use resources. When there are two cgroup managers on a system, you end up with two views of those resources. In the field, people have reported cases where nodes that are configured to use `cgroupfs` for the kubelet and Docker, but `systemd` for the rest of the processes, become unstable under resource pressure.

Changing the settings such that your container runtime and kubelet use `systemd` as the cgroup driver stabilized the system. To configure this for Docker, set `native.cgroupdriver=systemd` .

> **Caution:**
> Changing the cgroup driver of a Node that has joined a cluster is a sensitive operation. If the kubelet has created Pods using the semantics of one cgroup driver, changing the container runtime to another cgroup driver can cause errors when trying to re-create the Pod sandbox for such existing Pods. Restarting the kubelet may not solve such errors.
>
> If you have automation that makes it feasible, replace the node with another using the updated configuration, or reinstall it using automation.

Cgroup v2

Cgroup v2 is the next version of the cgroup Linux API. Differently than cgroup v1, there is a single hierarchy instead of a different one for each controller.

The new version offers several improvements over cgroup v1, some of these improvements are:

- cleaner and easier to use API

- safe sub-tree delegation to containers
- newer features like Pressure Stall Information

Even if the kernel supports a hybrid configuration where some controllers are managed by cgroup v1 and some others by cgroup v2, Kubernetes supports only the same cgroup version to manage all the controllers.

If systemd doesn't use cgroup v2 by default, you can configure the system to use it by adding `systemd.unified_cgroup_hierarchy=1` to the kernel command line.

```
# This example is for a Linux OS that uses the DNF package manager
# Your system might use a different method for setting the command line
# that the Linux kernel uses.
sudo dnf install -y grubby && \
  sudo grubby \
  --update-kernel=ALL \
  --args="systemd.unified_cgroup_hierarchy=1"
```

If you change the command line for the kernel, you must reboot the node before your change takes effect.

There should not be any noticeable difference in the user experience when switching to cgroup v2, unless users are accessing the cgroup file system directly, either on the node or from within the containers.

In order to use it, cgroup v2 must be supported by the CRI runtime as well.

Migrating to the `systemd` driver in kubeadm managed clusters

Follow this Migration guide if you wish to migrate to the `systemd` cgroup driver in existing kubeadm managed clusters.

CRI version support

Your container runtime must support at least v1alpha2 of the container runtime interface.

Kubernetes 1.23 defaults to using v1 of the CRI API. If a container runtime does not support the v1 API, the kubelet falls back to using the (deprecated) v1alpha2 API instead.

Container runtimes

> **Note:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

containerd

This section contains the necessary steps to use containerd as CRI runtime.

Use the following commands to install Containerd on your system:

Install and configure prerequisites:

```
cat <<EOF | sudo tee /etc/modules-load.d/containerd.conf
overlay
br_netfilter
```

```
EOF

sudo modprobe overlay
sudo modprobe br_netfilter

# Setup required sysctl params, these persist across reboots.
cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF

# Apply sysctl params without reboot
sudo sysctl --system
```

Install containerd:

Linux | Windows (PowerShell)

1. Install the `containerd.io` package from the official Docker repositories. Instructions for setting up the Docker repository for your respective Linux distribution and installing the `containerd.io` package can be found at Install Docker Engine.
2. Configure containerd:

```
sudo mkdir -p /etc/containerd
containerd config default | sudo tee /etc/containerd/config.toml
```

3. Restart containerd:

```
sudo systemctl restart containerd
```

Using the `systemd` cgroup driver

To use the `systemd` cgroup driver in `/etc/containerd/config.toml` with `runc`, set

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
  ...
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
    SystemdCgroup = true
```

If you apply this change make sure to restart containerd again:

```
sudo systemctl restart containerd
```

When using kubeadm, manually configure the cgroup driver for kubelet.

CRI-O

This section contains the necessary steps to install CRI-O as a container runtime.

Use the following commands to install CRI-O on your system:

> **Note:** The CRI-O major and minor versions must match the Kubernetes major and minor versions. For more information, see the CRI-O compatibility matrix.

Install and configure prerequisites:

```
# Create the .conf file to load the modules at bootup
cat <<EOF | sudo tee /etc/modules-load.d/crio.conf
overlay
br_netfilter
EOF

sudo modprobe overlay
sudo modprobe br_netfilter

# Set up required sysctl params, these persist across reboots.
cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF

sudo sysctl --system
```

Debian | Ubuntu | CentOS | openSUSE Tumbleweed | Fedora

To install CRI-O on the following operating systems, set the environment variable `OS` to the appropriate value from the following table:

Operating system	$OS
Debian Unstable	`Debian_Unstable`
Debian Testing	`Debian_Testing`

Then, set `$VERSION` to the CRI-O version that matches your Kubernetes version. For instance, if you want to install CRI-O 1.20, set `VERSION=1.20`. You can pin your installation to a specific release. To install version 1.20.0, set `VERSION=1.20:1.20.0`.

Then run

```
cat <<EOF | sudo tee /etc/apt/sources.list.d/devel:kubic:libcontainers:stable.list
deb https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable/$
EOF
cat <<EOF | sudo tee /etc/apt/sources.list.d/devel:kubic:libcontainers:stable:cri-o:
deb http://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable:/c
EOF

curl -L https://download.opensuse.org/repositories/devel:kubic:libcontainers:stable:
curl -L https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stab

sudo apt-get update
sudo apt-get install cri-o cri-o-runc
```

Start CRI-O:

```
sudo systemctl daemon-reload
sudo systemctl enable crio --now
```

Refer to the CRI-O installation guide for more information.

cgroup driver

CRI-O uses the systemd cgroup driver per default. To switch to the `cgroupfs` cgroup driver, either edit `/etc/crio/crio.conf` or place a drop-in configuration in `/etc/crio/crio.conf.d/02-cgroup-manager.conf`, for example:

```
[crio.runtime]
conmon_cgroup = "pod"
cgroup_manager = "cgroupfs"
```

Please also note the changed `conmon_cgroup`, which has to be set to the value `pod` when using CRI-O with `cgroupfs`. It is generally necessary to keep the cgroup driver configuration of the kubelet (usually done via kubeadm) and CRI-O in sync.

Docker Engine

Docker Engine is the container runtime that started it all. Formerly known just as Docker, this container runtime is available in various forms. Install Docker Engine explains your options for installing this runtime.

Docker Engine is directly compatible with Kubernetes 1.23, using the deprecated `dockershim` component. For more information and context, see the Dockershim deprecation FAQ.

You can also find third-party adapters that let you use Docker Engine with Kubernetes through the supported Container Runtime Interface (CRI).

The following CRI adaptors are designed to work with Docker Engine:

- `cri-dockerd` from Mirantis

Mirantis Container Runtime

Mirantis Container Runtime (MCR) is a commercially available container runtime that was formerly known as Docker Enterprise Edition.

You can use Mirantis Container Runtime with Kubernetes using the open source `cri-dockerd` component, included with MCR.

2.2 - Installing Kubernetes with deployment tools

2.2.1 - Bootstrapping clusters with kubeadm

2.2.1.1 - Installing kubeadm

This page shows how to install the `kubeadm` toolbox. For information on how to create a cluster with kubeadm once you have performed this installation process, see the Using kubeadm to Create a Cluster page.

Before you begin

- A compatible Linux host. The Kubernetes project provides generic instructions for Linux distributions based on Debian and Red Hat, and those distributions without a package manager.
- 2 GB or more of RAM per machine (any less will leave little room for your apps).
- 2 CPUs or more.
- Full network connectivity between all machines in the cluster (public or private network is fine).
- Unique hostname, MAC address, and product_uuid for every node. See here for more details.
- Certain ports are open on your machines. See here for more details.
- Swap disabled. You **MUST** disable swap in order for the kubelet to work properly.

Verify the MAC address and product_uuid are unique for every node

- You can get the MAC address of the network interfaces using the command `ip link` or `ifconfig -a`
- The product_uuid can be checked by using the command `sudo cat /sys/class/dmi/id/product_uuid`

It is very likely that hardware devices will have unique addresses, although some virtual machines may have identical values. Kubernetes uses these values to uniquely identify the nodes in the cluster. If these values are not unique to each node, the installation process may fail.

Check network adapters

If you have more than one network adapter, and your Kubernetes components are not reachable on the default route, we recommend you add IP route(s) so Kubernetes cluster addresses go via the appropriate adapter.

Letting iptables see bridged traffic

Make sure that the `br_netfilter` module is loaded. This can be done by running `lsmod | grep br_netfilter`. To load it explicitly call `sudo modprobe br_netfilter`.

As a requirement for your Linux Node's iptables to correctly see bridged traffic, you should ensure `net.bridge.bridge-nf-call-iptables` is set to 1 in your `sysctl` config, e.g.

```
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
br_netfilter
EOF

cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
sudo sysctl --system
```

For more details please see the Network Plugin Requirements page.

Check required ports

These required ports need to be open in order for Kubernetes components to communicate with each other. You can use telnet to check if a port is open. For example:

```
telnet 127.0.0.1 6443
```

The pod network plugin you use (see below) may also require certain ports to be open. Since this differs with each pod network plugin, please see the documentation for the plugins about what port(s) those need.

Installing runtime

To run containers in Pods, Kubernetes uses a container runtime.

Linux nodes | other operating systems

By default, Kubernetes uses the Container Runtime Interface (CRI) to interface with your chosen container runtime.

If you don't specify a runtime, kubeadm automatically tries to detect an installed container runtime by scanning through a list of well known Unix domain sockets. The following table lists container runtimes and their associated socket paths:

Runtime	Path to Unix domain socket
Docker	`/var/run/dockershim.sock`
containerd	`/run/containerd/containerd.sock`
CRI-O	`/var/run/crio/crio.sock`

If both Docker and containerd are detected, Docker takes precedence. This is needed because Docker 18.09 ships with containerd and both are detectable even if you only installed Docker. If any other two or more runtimes are detected, kubeadm exits with an error.

The kubelet integrates with Docker through the built-in `dockershim` CRI implementation.

See container runtimes for more information.

Installing kubeadm, kubelet and kubectl

You will install these packages on all of your machines:

- `kubeadm` : the command to bootstrap the cluster.

- `kubelet` : the component that runs on all of the machines in your cluster and does things like starting pods and containers.
- `kubectl` : the command line util to talk to your cluster.

kubeadm **will not** install or manage `kubelet` or `kubectl` for you, so you will need to ensure they match the version of the Kubernetes control plane you want kubeadm to install for you. If you do not, there is a risk of a version skew occurring that can lead to unexpected, buggy behaviour. However, *one* minor version skew between the kubelet and the control plane is supported, but the kubelet version may never exceed the API server version. For example, the kubelet running 1.7.0 should be fully compatible with a 1.8.0 API server, but not vice versa.

For information about installing `kubectl` , see Install and set up kubectl.

> **Warning:** These instructions exclude all Kubernetes packages from any system upgrades. This is because kubeadm and Kubernetes require special attention to upgrade.

For more information on version skews, see:

- Kubernetes version and version-skew policy
- Kubeadm-specific version skew policy

Debian-based distributions | Red Hat-based distributions

Without a package manager

1. Update the `apt` package index and install packages needed to use the Kubernetes `apt` repository:

```
sudo apt-get update
sudo apt-get install -y apt-transport-https ca-certificates curl
```

2. Download the Google Cloud public signing key:

```
sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg https://pac
```

3. Add the Kubernetes `apt` repository:

```
echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] https:
```

4. Update `apt` package index, install kubelet, kubeadm and kubectl, and pin their version:

```
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

The kubelet is now restarting every few seconds, as it waits in a crashloop for kubeadm to tell it what to do.

Configuring a cgroup driver

Both the container runtime and the kubelet have a property called "cgroup driver", which is important for the management of cgroups on Linux machines.

> **Warning:**
> Matching the container runtime and kubelet cgroup drivers is required or otherwise the kubelet process will fail.

> See Configuring a cgroup driver for more details.

Troubleshooting

If you are running into difficulties with kubeadm, please consult our troubleshooting docs.

What's next

- Using kubeadm to Create a Cluster

2.2.1.2 - Troubleshooting kubeadm

As with any program, you might run into an error installing or running kubeadm. This page lists some common failure scenarios and have provided steps that can help you understand and fix the problem.

If your problem is not listed below, please follow the following steps:

- If you think your problem is a bug with kubeadm:
  - Go to github.com/kubernetes/kubeadm and search for existing issues.
  - If no issue exists, please open one and follow the issue template.
- If you are unsure about how kubeadm works, you can ask on Slack in `#kubeadm`, or open a question on StackOverflow. Please include relevant tags like `#kubernetes` and `#kubeadm` so folks can help you.

Not possible to join a v1.18 Node to a v1.17 cluster due to missing RBAC

In v1.18 kubeadm added prevention for joining a Node in the cluster if a Node with the same name already exists. This required adding RBAC for the bootstrap-token user to be able to GET a Node object.

However this causes an issue where `kubeadm join` from v1.18 cannot join a cluster created by kubeadm v1.17.

To workaround the issue you have two options:

Execute `kubeadm init phase bootstrap-token` on a control-plane node using kubeadm v1.18. Note that this enables the rest of the bootstrap-token permissions as well.

or

Apply the following RBAC manually using `kubectl apply -f ...`:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kubeadm:get-nodes
rules:
- apiGroups:
  - ""
  resources:
  - nodes
  verbs:
  - get
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: kubeadm:get-nodes
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: kubeadm:get-nodes
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:bootstrappers:kubeadm:default-node-token
```

`ebtables` or some similar executable not found during installation

If you see the following warnings while running `kubeadm init`

```
[preflight] WARNING: ebtables not found in system path
[preflight] WARNING: ethtool not found in system path
```

Then you may be missing `ebtables`, `ethtool` or a similar executable on your node. You can install them with the following commands:

- For Ubuntu/Debian users, run `apt install ebtables ethtool`.
- For CentOS/Fedora users, run `yum install ebtables ethtool`.

kubeadm blocks waiting for control plane during installation

If you notice that `kubeadm init` hangs after printing out the following line:

```
[apiclient] Created API client, waiting for the control plane to become ready
```

This may be caused by a number of problems. The most common are:

- network connection problems. Check that your machine has full network connectivity before continuing.
- the cgroup driver of the container runtime differs from that of the kubelet. To understand how to configure it properly see Configuring a cgroup driver.
- control plane containers are crashlooping or hanging. You can check this by running `docker ps` and investigating each container by running `docker logs`. For other container runtime see Debugging Kubernetes nodes with crictl.

kubeadm blocks when removing managed containers

The following could happen if Docker halts and does not remove any Kubernetes-managed containers:

```
sudo kubeadm reset
```

```
[preflight] Running pre-flight checks
[reset] Stopping the kubelet service
[reset] Unmounting mounted directories in "/var/lib/kubelet"
[reset] Removing kubernetes-managed containers
(block)
```

A possible solution is to restart the Docker service and then re-run `kubeadm reset`:

```
sudo systemctl restart docker.service
sudo kubeadm reset
```

Inspecting the logs for docker may also be useful:

```
journalctl -u docker
```

Pods in `RunContainerError`, `CrashLoopBackOff` or `Error` state

Right after `kubeadm init` there should not be any pods in these states.

- If there are pods in one of these states *right after* `kubeadm init`, please open an issue in the kubeadm repo. `coredns` (or `kube-dns`) should be in the `Pending` state until you have deployed the network add-on.
- If you see Pods in the `RunContainerError`, `CrashLoopBackOff` or `Error` state after deploying the network add-on and nothing happens to `coredns` (or `kube-dns`), it's very likely that the Pod Network add-on that you installed is somehow broken. You might have to grant it more RBAC privileges or use a newer version. Please file an issue in the Pod Network providers' issue tracker and get the issue triaged there.
- If you install a version of Docker older than 1.12.1, remove the `MountFlags=slave` option when booting `dockerd` with `systemd` and restart `docker`. You can see the MountFlags in `/usr/lib/systemd/system/docker.service`. MountFlags can interfere with volumes mounted by Kubernetes, and put the Pods in `CrashLoopBackOff` state. The error happens when Kubernetes does not find `var/run/secrets/kubernetes.io/serviceaccount` files.

`coredns` is stuck in the `Pending` state

This is **expected** and part of the design. kubeadm is network provider-agnostic, so the admin should install the pod network add-on of choice. You have to install a Pod Network before CoreDNS may be deployed fully. Hence the `Pending` state before the network is set up.

`HostPort` services do not work

The `HostPort` and `HostIP` functionality is available depending on your Pod Network provider. Please contact the author of the Pod Network add-on to find out whether `HostPort` and `HostIP` functionality are available.

Calico, Canal, and Flannel CNI providers are verified to support HostPort.

For more information, see the CNI portmap documentation.

If your network provider does not support the portmap CNI plugin, you may need to use the NodePort feature of services or use `HostNetwork=true`.

Pods are not accessible via their Service IP

- Many network add-ons do not yet enable hairpin mode which allows pods to access themselves via their Service IP. This is an issue related to CNI. Please contact the network add-on provider to get the latest status of their support for hairpin mode.
- If you are using VirtualBox (directly or via Vagrant), you will need to ensure that `hostname -i` returns a routable IP address. By default the first interface is connected to a non-routable host-only network. A work around is to modify `/etc/hosts`, see this Vagrantfile for an example.

TLS certificate errors

The following error indicates a possible certificate mismatch.

```
# kubectl get pods
Unable to connect to the server: x509: certificate signed by unknown authority (possibly
```

- Verify that the `$HOME/.kube/config` file contains a valid certificate, and regenerate a certificate if necessary. The certificates in a kubeconfig file are base64 encoded. The `base64 --decode` command can be used to decode the certificate and `openssl x509 -text -noout` can be used for viewing the certificate information.
- Unset the `KUBECONFIG` environment variable using:

```
unset KUBECONFIG
```

Or set it to the default `KUBECONFIG` location:

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

- Another workaround is to overwrite the existing `kubeconfig` for the "admin" user:

```
mv  $HOME/.kube $HOME/.kube.bak
mkdir $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Kubelet client certificate rotation fails

By default, kubeadm configures a kubelet with automatic rotation of client certificates by using the `/var/lib/kubelet/pki/kubelet-client-current.pem` symlink specified in `/etc/kubernetes/kubelet.conf`. If this rotation process fails you might see errors such as `x509: certificate has expired or is not yet valid` in kube-apiserver logs. To fix the issue you must follow these steps:

1. Backup and delete `/etc/kubernetes/kubelet.conf` and `/var/lib/kubelet/pki/kubelet-client*` from the failed node.
2. From a working control plane node in the cluster that has `/etc/kubernetes/pki/ca.key` execute `kubeadm kubeconfig user --org system:nodes --client-name system:node:$NODE > kubelet.conf`. `$NODE` must be set to the name of the existing failed node in the cluster. Modify the resulted `kubelet.conf` manually to adjust the cluster name and server endpoint, or pass `kubeconfig user --config` (it accepts `InitConfiguration`). If your cluster does not have the `ca.key` you must sign the embedded certificates in the `kubelet.conf` externally.
3. Copy this resulted `kubelet.conf` to `/etc/kubernetes/kubelet.conf` on the failed node.
4. Restart the kubelet (`systemctl restart kubelet`) on the failed node and wait for `/var/lib/kubelet/pki/kubelet-client-current.pem` to be recreated.
5. Manually edit the `kubelet.conf` to point to the rotated kubelet client certificates, by replacing `client-certificate-data` and `client-key-data` with:

```
client-certificate: /var/lib/kubelet/pki/kubelet-client-current.pem
client-key: /var/lib/kubelet/pki/kubelet-client-current.pem
```

6. Restart the kubelet.

7. Make sure the node becomes `Ready`.

Default NIC When using flannel as the pod network in Vagrant

The following error might indicate that something was wrong in the pod network:

```
Error from server (NotFound): the server could not find the requested resource
```

- If you're using flannel as the pod network inside Vagrant, then you will have to specify the default interface name for flannel.

  Vagrant typically assigns two interfaces to all VMs. The first, for which all hosts are assigned the IP address `10.0.2.15`, is for external traffic that gets NATed.

  This may lead to problems with flannel, which defaults to the first interface on a host. This leads to all hosts thinking they have the same public IP address. To prevent this, pass the `--iface eth1` flag to flannel so that the second interface is chosen.

Non-public IP used for containers

In some situations `kubectl logs` and `kubectl run` commands may return with the following errors in an otherwise functional cluster:

```
Error from server: Get https://10.19.0.41:10250/containerLogs/default/mysql-ddc65b868-gl
```

- This may be due to Kubernetes using an IP that can not communicate with other IPs on the seemingly same subnet, possibly by policy of the machine provider.

- DigitalOcean assigns a public IP to `eth0` as well as a private one to be used internally as anchor for their floating IP feature, yet `kubelet` will pick the latter as the node's `InternalIP` instead of the public one.

  Use `ip addr show` to check for this scenario instead of `ifconfig` because `ifconfig` will not display the offending alias IP address. Alternatively an API endpoint specific to DigitalOcean allows to query for the anchor IP from the droplet:

```
curl http://169.254.169.254/metadata/v1/interfaces/public/0/anchor_ipv4/address
```

The workaround is to tell `kubelet` which IP to use using `--node-ip`. When using DigitalOcean, it can be the public one (assigned to `eth0`) or the private one (assigned to `eth1`) should you want to use the optional private network. The `kubeletExtraArgs` section of the kubeadm `NodeRegistrationOptions` structure can be used for this.

Then restart `kubelet`:

```
systemctl daemon-reload
systemctl restart kubelet
```

`coredns` pods have `CrashLoopBackOff` or `Error` state

If you have nodes that are running SELinux with an older version of Docker you might experience a scenario where the `coredns` pods are not starting. To solve that you can try one of the following options:

- Upgrade to a newer version of Docker.
- Disable SELinux.
- Modify the `coredns` deployment to set `allowPrivilegeEscalation` to `true`:

```
kubectl -n kube-system get deployment coredns -o yaml | \
  sed 's/allowPrivilegeEscalation: false/allowPrivilegeEscalation: true/g' | \
  kubectl apply -f -
```

Another cause for CoreDNS to have `CrashLoopBackOff` is when a CoreDNS Pod deployed in Kubernetes detects a loop. A number of workarounds are available to avoid Kubernetes trying to restart the CoreDNS Pod every time CoreDNS detects the loop and exits.

> **Warning:** Disabling SELinux or setting `allowPrivilegeEscalation` to `true` can compromise the security of your cluster.

etcd pods restart continually

If you encounter the following error:

```
rpc error: code = 2 desc = oci runtime error: exec failed: container_linux.go:247: starti
```

this issue appears if you run CentOS 7 with Docker 1.13.1.84. This version of Docker can prevent the kubelet from executing into the etcd container.

To work around the issue, choose one of these options:

- Roll back to an earlier version of Docker, such as 1.13.1-75

```
yum downgrade docker-1.13.1-75.git8633870.el7.centos.x86_64 docker-client-1.13.1-75.git86
```

- Install one of the more recent recommended versions, such as 18.06:

```
sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.rep
yum install docker-ce-18.06.1.ce-3.el7.x86_64
```

Not possible to pass a comma separated list of values to arguments inside a `--component-extra-args` flag

`kubeadm init` flags such as `--component-extra-args` allow you to pass custom arguments to a control-plane component like the kube-apiserver. However, this mechanism is limited due to the underlying type used for parsing the values (`mapStringString`).

If you decide to pass an argument that supports multiple, comma-separated values such as `--apiserver-extra-args "enable-admission-plugins=LimitRanger,NamespaceExists"` this flag will fail with `flag: malformed pair, expect string=string`. This happens because the list of arguments for `--apiserver-extra-args` expects `key=value` pairs and in this case `NamespacesExists` is considered as a key that is missing a value.

Alternatively, you can try separating the `key=value` pairs like so: `--apiserver-extra-args "enable-admission-plugins=LimitRanger,enable-admission-plugins=NamespaceExists"` but this will result in the key `enable-admission-plugins` only having the value of `NamespaceExists`.

A known workaround is to use the kubeadm configuration file.

kube-proxy scheduled before node is initialized by cloud-controller-manager

In cloud provider scenarios, kube-proxy can end up being scheduled on new worker nodes before the cloud-controller-manager has initialized the node addresses. This causes kube-proxy to fail to pick up the node's IP address properly and has knock-on effects to the proxy function managing load balancers.

The following error can be seen in kube-proxy Pods:

```
server.go:610] Failed to retrieve node IP: host IP unknown; known addresses: []
proxier.go:340] invalid nodeIP, initializing kube-proxy with 127.0.0.1 as nodeIP
```

A known solution is to patch the kube-proxy DaemonSet to allow scheduling it on control-plane nodes regardless of their conditions, keeping it off of other nodes until their initial guarding conditions abate:

```
kubectl -n kube-system patch ds kube-proxy -p='{ "spec": { "template": { "spec": { "toler
```

The tracking issue for this problem is here.

`/usr` is mounted read-only on nodes

On Linux distributions such as Fedora CoreOS or Flatcar Container Linux, the directory `/usr` is mounted as a read-only filesystem. For flex-volume support, Kubernetes components like the kubelet and kube-controller-manager use the default path of `/usr/libexec/kubernetes/kubelet-plugins/volume/exec/`, yet the flex-volume directory *must be writeable* for the feature to work. (**Note**: FlexVolume was deprecated in the Kubernetes v1.23 release)

To workaround this issue you can configure the flex-volume directory using the kubeadm configuration file.

On the primary control-plane Node (created using `kubeadm init`) pass the following file using `--config`:

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
nodeRegistration:
  kubeletExtraArgs:
    volume-plugin-dir: "/opt/libexec/kubernetes/kubelet-plugins/volume/exec/"
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
controllerManager:
  extraArgs:
    flex-volume-plugin-dir: "/opt/libexec/kubernetes/kubelet-plugins/volume/exec/"
```

On joining Nodes:

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: JoinConfiguration
nodeRegistration:
  kubeletExtraArgs:
    volume-plugin-dir: "/opt/libexec/kubernetes/kubelet-plugins/volume/exec/"
```

Alternatively, you can modify `/etc/fstab` to make the `/usr` mount writeable, but please be advised that this is modifying a design principle of the Linux distribution.

`kubeadm upgrade plan` prints out `context deadline exceeded` error message

This error message is shown when upgrading a Kubernetes cluster with `kubeadm` in the case of running an external etcd. This is not a critical bug and happens because older versions of kubeadm perform a version check on the external etcd cluster. You can proceed with `kubeadm upgrade apply ...`.

This issue is fixed as of version 1.19.

`kubeadm reset` unmounts `/var/lib/kubelet`

If `/var/lib/kubelet` is being mounted, performing a `kubeadm reset` will effectively unmount it.

To workaround the issue, re-mount the `/var/lib/kubelet` directory after performing the `kubeadm reset` operation.

This is a regression introduced in kubeadm 1.15. The issue is fixed in 1.20.

Cannot use the metrics-server securely in a kubeadm cluster

In a kubeadm cluster, the metrics-server can be used insecurely by passing the `--kubelet-insecure-tls` to it. This is not recommended for production clusters.

If you want to use TLS between the metrics-server and the kubelet there is a problem, since kubeadm deploys a self-signed serving certificate for the kubelet. This can cause the following errors on the side of the metrics-server:

```
x509: certificate signed by unknown authority
x509: certificate is valid for IP-foo not IP-bar
```

See Enabling signed kubelet serving certificates to understand how to configure the kubelets in a kubeadm cluster to have properly signed serving certificates.

Also see How to run the metrics-server securely.

2.2.1.3 - Creating a cluster with kubeadm

Using `kubeadm`, you can create a minimum viable Kubernetes cluster that conforms to best practices. In fact, you can use `kubeadm` to set up a cluster that will pass the Kubernetes Conformance tests. `kubeadm` also supports other cluster lifecycle functions, such as bootstrap tokens and cluster upgrades.

The `kubeadm` tool is good if you need:

- A simple way for you to try out Kubernetes, possibly for the first time.
- A way for existing users to automate setting up a cluster and test their application.
- A building block in other ecosystem and/or installer tools with a larger scope.

You can install and use `kubeadm` on various machines: your laptop, a set of cloud servers, a Raspberry Pi, and more. Whether you're deploying into the cloud or on-premises, you can integrate `kubeadm` into provisioning systems such as Ansible or Terraform.

Before you begin

To follow this guide, you need:

- One or more machines running a deb/rpm-compatible Linux OS; for example: Ubuntu or CentOS.
- 2 GiB or more of RAM per machine--any less leaves little room for your apps.
- At least 2 CPUs on the machine that you use as a control-plane node.
- Full network connectivity among all machines in the cluster. You can use either a public or a private network.

You also need to use a version of `kubeadm` that can deploy the version of Kubernetes that you want to use in your new cluster.

Kubernetes' version and version skew support policy applies to `kubeadm` as well as to Kubernetes overall. Check that policy to learn about what versions of Kubernetes and `kubeadm` are supported. This page is written for Kubernetes v1.23.

The `kubeadm` tool's overall feature state is General Availability (GA). Some sub-features are still under active development. The implementation of creating the cluster may change slightly as the tool evolves, but the overall implementation should be pretty stable.

> **Note:** Any commands under `kubeadm alpha` are, by definition, supported on an alpha level.

Objectives

- Install a single control-plane Kubernetes cluster
- Install a Pod network on the cluster so that your Pods can talk to each other

Instructions

Installing kubeadm on your hosts

See "Installing kubeadm".

> **Note:**
> If you have already installed kubeadm, run `apt-get update && apt-get upgrade` or `yum update` to get the latest version of kubeadm.

> When you upgrade, the kubelet restarts every few seconds as it waits in a crashloop for kubeadm to tell it what to do. This crashloop is expected and normal. After you initialize your control-plane, the kubelet runs normally.

Preparing the required container images

This step is optional and only applies in case you wish `kubeadm init` and `kubeadm join` to not download the default container images which are hosted at `k8s.gcr.io`.

Kubeadm has commands that can help you pre-pull the required images when creating a cluster without an internet connection on its nodes. See Running kubeadm without an internet connection for more details.

Kubeadm allows you to use a custom image repository for the required images. See Using custom images for more details.

Initializing your control-plane node

The control-plane node is the machine where the control plane components run, including etcd (the cluster database) and the API Server (which the kubectl command line tool communicates with).

1. (Recommended) If you have plans to upgrade this single control-plane `kubeadm` cluster to high availability you should specify the `--control-plane-endpoint` to set the shared endpoint for all control-plane nodes. Such an endpoint can be either a DNS name or an IP address of a load-balancer.
2. Choose a Pod network add-on, and verify whether it requires any arguments to be passed to `kubeadm init`. Depending on which third-party provider you choose, you might need to set the `--pod-network-cidr` to a provider-specific value. See Installing a Pod network add-on.
3. (Optional) Since version 1.14, `kubeadm` tries to detect the container runtime on Linux by using a list of well known domain socket paths. To use different container runtime or if there are more than one installed on the provisioned node, specify the `--cri-socket` argument to `kubeadm init`. See Installing a runtime.
4. (Optional) Unless otherwise specified, `kubeadm` uses the network interface associated with the default gateway to set the advertise address for this particular control-plane node's API server. To use a different network interface, specify the `--apiserver-advertise-address=<ip-address>` argument to `kubeadm init`. To deploy an IPv6 Kubernetes cluster using IPv6 addressing, you must specify an IPv6 address, for example `--apiserver-advertise-address=fd00::101`

To initialize the control-plane node run:

```
kubeadm init <args>
```

Considerations about apiserver-advertise-address and ControlPlaneEndpoint

While `--apiserver-advertise-address` can be used to set the advertise address for this particular control-plane node's API server, `--control-plane-endpoint` can be used to set the shared endpoint for all control-plane nodes.

`--control-plane-endpoint` allows both IP addresses and DNS names that can map to IP addresses. Please contact your network administrator to evaluate possible solutions with respect to such mapping.

Here is an example mapping:

```
192.168.0.102 cluster-endpoint
```

Where `192.168.0.102` is the IP address of this node and `cluster-endpoint` is a custom DNS name that maps to this IP. This will allow you to pass `--control-plane-endpoint=cluster-endpoint` to `kubeadm init` and pass the same DNS name to `kubeadm join`. Later you can modify `cluster-endpoint` to point to the address of your load-balancer in an high availability scenario.

Turning a single control plane cluster created without `--control-plane-endpoint` into a highly available cluster is not supported by kubeadm.

More information

For more information about `kubeadm init` arguments, see the kubeadm reference guide.

To configure `kubeadm init` with a configuration file see Using kubeadm init with a configuration file.

To customize control plane components, including optional IPv6 assignment to liveness probe for control plane components and etcd server, provide extra arguments to each component as documented in custom arguments.

To run `kubeadm init` again, you must first tear down the cluster.

If you join a node with a different architecture to your cluster, make sure that your deployed DaemonSets have container image support for this architecture.

`kubeadm init` first runs a series of prechecks to ensure that the machine is ready to run Kubernetes. These prechecks expose warnings and exit on errors. `kubeadm init` then downloads and installs the cluster control plane components. This may take several minutes. After it finishes you should see:

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a Pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  /docs/concepts/cluster-administration/addons/

You can now join any number of machines by running the following on each node
as root:

  kubeadm join <control-plane-host>:<control-plane-port> --token <token> --discovery-toke
```

To make kubectl work for your non-root user, run these commands, which are also part of the `kubeadm init` output:

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Alternatively, if you are the `root` user, you can run:

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

> **Warning:** Kubeadm signs the certificate in the `admin.conf` to have `Subject: O = system:masters, CN = kubernetes-admin`. `system:masters` is a break-glass, super user group that bypasses the authorization layer (e.g. RBAC). Do not share the `admin.conf` file with

anyone and instead grant users custom permissions by generating them a kubeconfig file using the `kubeadm kubeconfig user` command. For more details see Generating kubeconfig files for additional users.

Make a record of the `kubeadm join` command that `kubeadm init` outputs. You need this command to join nodes to your cluster.

The token is used for mutual authentication between the control-plane node and the joining nodes. The token included here is secret. Keep it safe, because anyone with this token can add authenticated nodes to your cluster. These tokens can be listed, created, and deleted with the `kubeadm token` command. See the kubeadm reference guide.

Installing a Pod network add-on

Caution:
This section contains important information about networking setup and deployment order. Read all of this advice carefully before proceeding.

You must deploy a Container Network Interface (CNI) based Pod network add-on so that your Pods can communicate with each other. Cluster DNS (CoreDNS) will not start up before a network is installed.

- Take care that your Pod network must not overlap with any of the host networks: you are likely to see problems if there is any overlap. (If you find a collision between your network plugin's preferred Pod network and some of your host networks, you should think of a suitable CIDR block to use instead, then use that during `kubeadm init` with `--pod-network-cidr` and as a replacement in your network plugin's YAML).
- By default, `kubeadm` sets up your cluster to use and enforce use of RBAC (role based access control). Make sure that your Pod network plugin supports RBAC, and so do any manifests that you use to deploy it.
- If you want to use IPv6--either dual-stack, or single-stack IPv6 only networking--for your cluster, make sure that your Pod network plugin supports IPv6. IPv6 support was added to CNI in v0.6.0.

Note: Kubeadm should be CNI agnostic and the validation of CNI providers is out of the scope of our current e2e testing. If you find an issue related to a CNI plugin you should log a ticket in its respective issue tracker instead of the kubeadm or kubernetes issue trackers.

Several external projects provide Kubernetes Pod networks using CNI, some of which also support Network Policy.

See a list of add-ons that implement the Kubernetes networking model.

You can install a Pod network add-on with the following command on the control-plane node or a node that has the kubeconfig credentials:

```
kubectl apply -f <add-on.yaml>
```

You can install only one Pod network per cluster.

Once a Pod network has been installed, you can confirm that it is working by checking that the CoreDNS Pod is `Running` in the output of `kubectl get pods --all-namespaces`. And once the CoreDNS Pod is up and running, you can continue by joining your nodes.

If your network is not working or CoreDNS is not in the `Running` state, check out the troubleshooting guide for `kubeadm`.

Managed node labels

By default, kubeadm enables the NodeRestriction admission controller that restricts what labels can be self-applied by kubelets on node registration. The admission controller documentation covers what labels are permitted to be used with the kubelet `--node-labels` option. The `node-role.kubernetes.io/control-plane` label is such a restricted label and kubeadm manually applies it using a privileged client after a node has been created. To do that manually you can do the same by using `kubectl label` and ensure it is using a privileged kubeconfig such as the kubeadm managed `/etc/kubernetes/admin.conf`.

Control plane node isolation

By default, your cluster will not schedule Pods on the control-plane node for security reasons. If you want to be able to schedule Pods on the control-plane node, for example for a single-machine Kubernetes cluster for development, run:

```
kubectl taint nodes --all node-role.kubernetes.io/master-
```

With output looking something like:

```
node "test-01" untainted
taint "node-role.kubernetes.io/master:" not found
taint "node-role.kubernetes.io/master:" not found
```

This will remove the `node-role.kubernetes.io/master` taint from any nodes that have it, including the control-plane node, meaning that the scheduler will then be able to schedule Pods everywhere.

Joining your nodes

The nodes are where your workloads (containers and Pods, etc) run. To add new nodes to your cluster do the following for each machine:

- SSH to the machine
- Become root (e.g. `sudo su -`)
- Install a runtime if needed
- Run the command that was output by `kubeadm init`. For example:

```
kubeadm join --token <token> <control-plane-host>:<control-plane-port> --discovery-
```

If you do not have the token, you can get it by running the following command on the control-plane node:

```
kubeadm token list
```

The output is similar to this:

```
TOKEN                    TTL  EXPIRES              USAGES           DESCRIPTION
8ewj1p.9r9hcjoqgajrj4gi  23h  2018-06-12T02:51:28Z authentication,  The default bootstrap
                                                   signing          token generated by
                                                                    'kubeadm init'.
```

By default, tokens expire after 24 hours. If you are joining a node to the cluster after the current token has expired, you can create a new token by running the following command on the control-plane node:

```
kubeadm token create
```

The output is similar to this:

```
5didvk.d09sbcov8ph2amjw
```

If you don't have the value of `--discovery-token-ca-cert-hash`, you can get it by running the following command chain on the control-plane node:

```
openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin -outform der 2>/
   openssl dgst -sha256 -hex | sed 's/^.* //'
```

The output is similar to:

```
8cb2de97839780a412b93877f8507ad6c94f73add17d5d7058e91741c9d5ec78
```

> **Note:** To specify an IPv6 tuple for `<control-plane-host>:<control-plane-port>`, IPv6 address must be enclosed in square brackets, for example: `[fd00::101]:2073`.

The output should look something like:

```
[preflight] Running pre-flight checks

... (log output of join workflow) ...

Node join complete:
* Certificate signing request sent to control-plane and response
  received.
* Kubelet informed of new secure connection details.

Run 'kubectl get nodes' on control-plane to see this machine join.
```

A few seconds later, you should notice this node in the output from `kubectl get nodes` when run on the control-plane node.

> **Note:** As the cluster nodes are usually initialized sequentially, the CoreDNS Pods are likely to all run on the first control-plane node. To provide higher availability, please rebalance the CoreDNS Pods with `kubectl -n kube-system rollout restart deployment coredns` after at least one new node is joined.

(Optional) Controlling your cluster from machines other than the control-plane node

In order to get a kubectl on some other computer (e.g. laptop) to talk to your cluster, you need to copy the administrator kubeconfig file from your control-plane node to your workstation like this:

```
scp root@<control-plane-host>:/etc/kubernetes/admin.conf .
kubectl --kubeconfig ./admin.conf get nodes
```

> **Note:**
> The example above assumes SSH access is enabled for root. If that is not the case, you can copy the `admin.conf` file to be accessible by some other user and `scp` using that other user instead.
>
> The `admin.conf` file gives the user *superuser* privileges over the cluster. This file should be used sparingly. For normal users, it's recommended to generate an unique credential to which you grant privileges. You can do this with the `kubeadm alpha kubeconfig user --client-name <CN>` command. That command will print out a KubeConfig file to STDOUT which you should save to a file and distribute to your user. After that, grant privileges by using `kubectl create (cluster)rolebinding`.

(Optional) Proxying API Server to localhost

If you want to connect to the API Server from outside the cluster you can use `kubectl proxy`:

```
scp root@<control-plane-host>:/etc/kubernetes/admin.conf .
kubectl --kubeconfig ./admin.conf proxy
```

You can now access the API Server locally at `http://localhost:8001/api/v1`

Clean up

If you used disposable servers for your cluster, for testing, you can switch those off and do no further clean up. You can use `kubectl config delete-cluster` to delete your local references to the cluster.

However, if you want to deprovision your cluster more cleanly, you should first drain the node and make sure that the node is empty, then deconfigure the node.

Remove the node

Talking to the control-plane node with the appropriate credentials, run:

```
kubectl drain <node name> --delete-emptydir-data --force --ignore-daemonsets
```

Before removing the node, reset the state installed by `kubeadm`:

```
kubeadm reset
```

The reset process does not reset or clean up iptables rules or IPVS tables. If you wish to reset iptables, you must do so manually:

```
iptables -F && iptables -t nat -F && iptables -t mangle -F && iptables -X
```

If you want to reset the IPVS tables, you must run the following command:

```
ipvsadm -C
```

Now remove the node:

```
kubectl delete node <node name>
```

If you wish to start over, run `kubeadm init` or `kubeadm join` with the appropriate arguments.

Clean up the control plane

You can use `kubeadm reset` on the control plane host to trigger a best-effort clean up.

See the `kubeadm reset` reference documentation for more information about this subcommand and its options.

What's next

- Verify that your cluster is running properly with Sonobuoy
- See Upgrading kubeadm clusters for details about upgrading your cluster using `kubeadm`.
- Learn about advanced `kubeadm` usage in the kubeadm reference documentation
- Learn more about Kubernetes concepts and `kubectl`.
- See the Cluster Networking page for a bigger list of Pod network add-ons.
- See the list of add-ons to explore other add-ons, including tools for logging, monitoring, network policy, visualization & control of your Kubernetes cluster.
- Configure how your cluster handles logs for cluster events and from applications running in Pods. See Logging Architecture for an overview of what is involved.

Feedback

- For bugs, visit the kubeadm GitHub issue tracker
- For support, visit the #kubeadm Slack channel
- General SIG Cluster Lifecycle development Slack channel: #sig-cluster-lifecycle
- SIG Cluster Lifecycle SIG information
- SIG Cluster Lifecycle mailing list: kubernetes-sig-cluster-lifecycle

Version skew policy

The `kubeadm` tool of version v1.23 may deploy clusters with a control plane of version v1.23 or v1.22. `kubeadm` v1.23 can also upgrade an existing kubeadm-created cluster of version v1.22.

Due to that we can't see into the future, kubeadm CLI v1.23 may or may not be able to deploy v1.24 clusters.

These resources provide more information on supported version skew between kubelets and the control plane, and other Kubernetes components:

- Kubernetes version and version-skew policy
- Kubeadm-specific installation guide

Limitations

Cluster resilience

The cluster created here has a single control-plane node, with a single etcd database running on it. This means that if the control-plane node fails, your cluster may lose data and may need to be recreated from scratch.

Workarounds:

- Regularly back up etcd. The etcd data directory configured by kubeadm is at `/var/lib/etcd` on the control-plane node.
- Use multiple control-plane nodes. You can read Options for Highly Available topology to pick a cluster topology that provides high-availability.

Platform compatibility

kubeadm deb/rpm packages and binaries are built for amd64, arm (32-bit), arm64, ppc64le, and s390x following the multi-platform proposal.

Multiplatform container images for the control plane and addons are also supported since v1.12.

Only some of the network providers offer solutions for all platforms. Please consult the list of network providers above or the documentation from each provider to figure out whether the provider supports your chosen platform.

Troubleshooting

If you are running into difficulties with kubeadm, please consult our troubleshooting docs.

2.2.1.4 - Customizing components with the kubeadm API

This page covers how to customize the components that kubeadm deploys. For control plane components you can use flags in the `ClusterConfiguration` structure or patches per-node. For the kubelet and kube-proxy you can use `KubeletConfiguration` and `KubeProxyConfiguration`, accordingly.

All of these options are possible via the kubeadm configuration API. For more details on each field in the configuration you can navigate to our API reference pages.

> **Note:** Customizing the CoreDNS deployment of kubeadm is currently not supported. You must manually patch the `kube-system/coredns` ConfigMap and recreate the CoreDNS Pods after that. Alternatively, you can skip the default CoreDNS deployment and deploy your own variant. For more details on that see Using init phases with kubeadm.

FEATURE STATE: `Kubernetes v1.12 [stable]`

Customizing the control plane with flags in `ClusterConfiguration`

The kubeadm `ClusterConfiguration` object exposes a way for users to override the default flags passed to control plane components such as the APIServer, ControllerManager, Scheduler and Etcd. The components are defined using the following structures:

- `apiServer`
- `controllerManager`
- `scheduler`
- `etcd`

These structures contain a common `extraArgs` field, that consists of `key: value` pairs. To override a flag for a control plane component:

1. Add the appropriate `extraArgs` to your configuration.
2. Add flags to the `extraArgs` field.
3. Run `kubeadm init` with `--config <YOUR CONFIG YAML>`.

> **Note:** You can generate a `ClusterConfiguration` object with default values by running `kubeadm config print init-defaults` and saving the output to a file of your choice.

> **Note:** The `ClusterConfiguration` object is currently global in kubeadm clusters. This means that any flags that you add, will apply to all instances of the same component on different nodes. To apply individual configuration per component on different nodes you can use patches.

> **Note:** Duplicate flags (keys), or passing the same flag `--foo` multiple times, is currently not supported. To workaround that you must use patches.

APIServer flags

For details, see the reference documentation for kube-apiserver.

Example usage:

```
apiVersion: kubeadm.k8s.io/v1beta3
```

```
kind: ClusterConfiguration
kubernetesVersion: v1.16.0
apiServer:
  extraArgs:
    anonymous-auth: "false"
    enable-admission-plugins: AlwaysPullImages,DefaultStorageClass
    audit-log-path: /home/johndoe/audit.log
```

ControllerManager flags

For details, see the reference documentation for kube-controller-manager.

Example usage:

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
kubernetesVersion: v1.16.0
controllerManager:
  extraArgs:
    cluster-signing-key-file: /home/johndoe/keys/ca.key
    deployment-controller-sync-period: "50"
```

Scheduler flags

For details, see the reference documentation for kube-scheduler.

Example usage:

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
kubernetesVersion: v1.16.0
scheduler:
  extraArgs:
    config: /etc/kubernetes/scheduler-config.yaml
  extraVolumes:
    - name: schedulerconfig
      hostPath: /home/johndoe/schedconfig.yaml
      mountPath: /etc/kubernetes/scheduler-config.yaml
      readOnly: true
      pathType: "File"
```

Etcd flags

For details, see the etcd server documentation.

Example usage:

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
etcd:
  local:
    extraArgs:
      election-timeout: 1000
```

Customizing the control plane with patches

FEATURE STATE: `Kubernetes v1.22 [beta]`

Kubeadm allows you to pass a directory with patch files to `InitConfiguration` and `JoinConfiguration` on individual nodes. These patches can be used as the last customization step before the control plane component manifests are written to disk.

You can pass this file to `kubeadm init` with `--config <YOUR CONFIG YAML>`:

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
patches:
  directory: /home/user/somedir
```

> **Note:** For `kubeadm init` you can pass a file containing both a `ClusterConfiguration` and `InitConfiguration` separated by `---`.

You can pass this file to `kubeadm join` with `--config <YOUR CONFIG YAML>`:

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: JoinConfiguration
patches:
  directory: /home/user/somedir
```

The directory must contain files named `target[suffix][+patchtype].extension`. For example, `kube-apiserver0+merge.yaml` or just `etcd.json`.

- `target` can be one of `kube-apiserver`, `kube-controller-manager`, `kube-scheduler` and `etcd`.
- `patchtype` can be one of `strategic`, `merge` or `json` and these must match the patching formats supported by kubectl. The default `patchtype` is `strategic`.
- `extension` must be either `json` or `yaml`.
- `suffix` is an optional string that can be used to determine which patches are applied first alpha-numerically.

> **Note:** If you are using `kubeadm upgrade` to upgrade your kubeadm nodes you must again provide the same patches, so that the customization is preserved after upgrade. To do that you can use the `--patches` flag, which must point to the same directory. `kubeadm upgrade` currently does not support a configuration API structure that can be used for the same purpose.

Customizing the kubelet

To customize the kubelet you can add a `KubeletConfiguration` next to the `ClusterConfiguration` or `InitConfiguration` separated by `---` within the same configuration file. This file can then be passed to `kubeadm init`.

> **Note:** kubeadm applies the same `KubeletConfiguration` to all nodes in the cluster. To apply node specific settings you can use kubelet flags as overrides by passing them in the `nodeRegistration.kubeletExtraArgs` field supported by both `InitConfiguration` and `JoinConfiguration`. Some kubelet flags are deprecated, so check their status in the kubelet reference documentation before using them.

For more details see Configuring each kubelet in your cluster using kubeadm

Customizing kube-proxy

To customize kube-proxy you can pass a `KubeProxyConfiguration` next your `ClusterConfiguration` or `InitConfiguration` to `kubeadm init` separated by `---`.

For more details you can navigate to our API reference pages.

> **Note:** kubeadm deploys kube-proxy as a DaemonSet, which means that the `KubeProxyConfiguration` would apply to all instances of kube-proxy in the cluster.

2.2.1.5 - Options for Highly Available Topology

This page explains the two options for configuring the topology of your highly available (HA) Kubernetes clusters.

You can set up an HA cluster:

- With stacked control plane nodes, where etcd nodes are colocated with control plane nodes
- With external etcd nodes, where etcd runs on separate nodes from the control plane

You should carefully consider the advantages and disadvantages of each topology before setting up an HA cluster.

> **Note:** kubeadm bootstraps the etcd cluster statically. Read the etcd Clustering Guide for more details.

Stacked etcd topology

A stacked HA cluster is a topology where the distributed data storage cluster provided by etcd is stacked on top of the cluster formed by the nodes managed by kubeadm that run control plane components.

Each control plane node runs an instance of the `kube-apiserver`, `kube-scheduler`, and `kube-controller-manager`. The `kube-apiserver` is exposed to worker nodes using a load balancer.

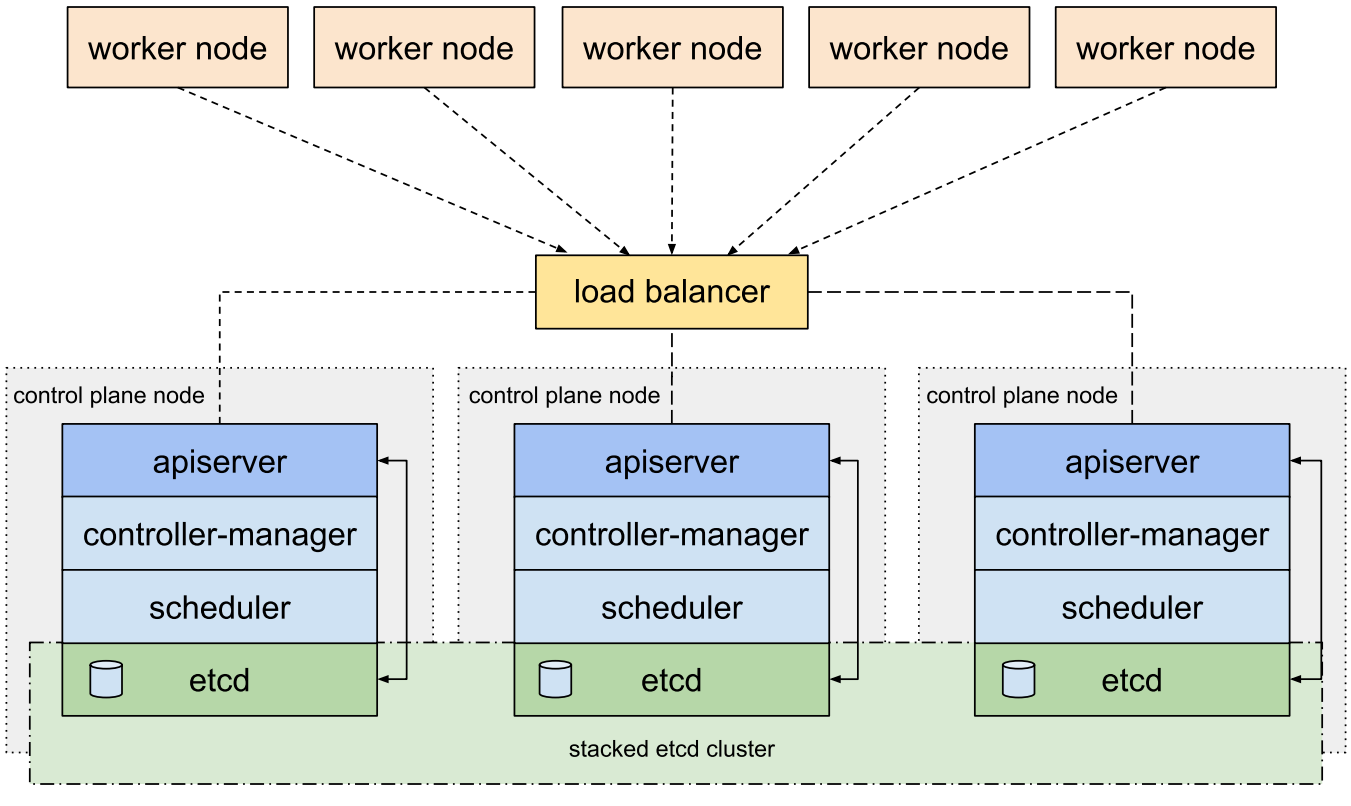
Each control plane node creates a local etcd member and this etcd member communicates only with the `kube-apiserver` of this node. The same applies to the local `kube-controller-manager` and `kube-scheduler` instances.

This topology couples the control planes and etcd members on the same nodes. It is simpler to set up than a cluster with external etcd nodes, and simpler to manage for replication.

However, a stacked cluster runs the risk of failed coupling. If one node goes down, both an etcd member and a control plane instance are lost, and redundancy is compromised. You can mitigate this risk by adding more control plane nodes.

You should therefore run a minimum of three stacked control plane nodes for an HA cluster.

This is the default topology in kubeadm. A local etcd member is created automatically on control plane nodes when using `kubeadm init` and `kubeadm join --control-plane`.



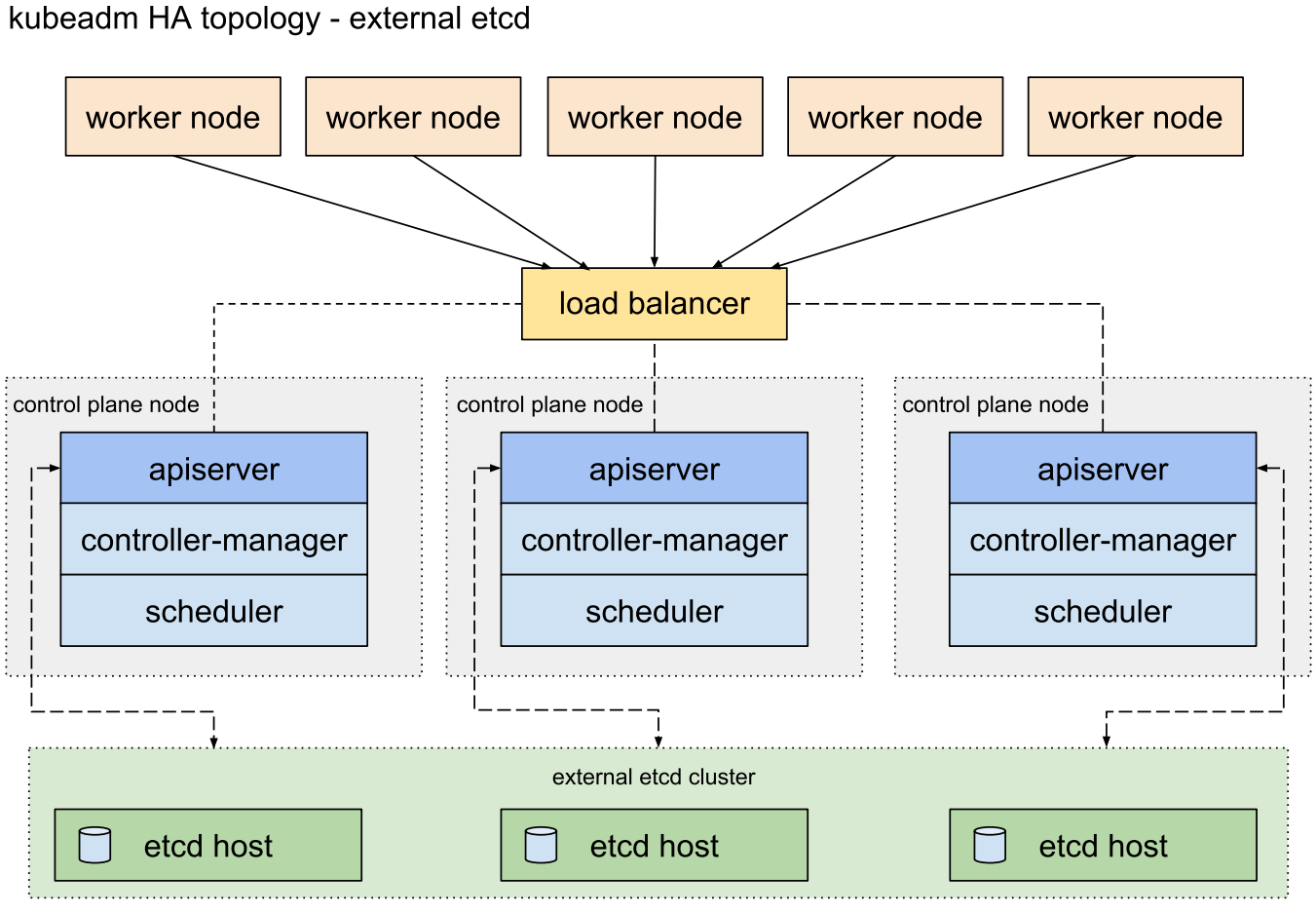
External etcd topology

An HA cluster with external etcd is a topology where the distributed data storage cluster provided by etcd is external to the cluster formed by the nodes that run control plane components.

Like the stacked etcd topology, each control plane node in an external etcd topology runs an instance of the `kube-apiserver`, `kube-scheduler`, and `kube-controller-manager`. And the `kube-apiserver` is exposed to worker nodes using a load balancer. However, etcd members run on separate hosts, and each etcd host communicates with the `kube-apiserver` of each control plane node.

This topology decouples the control plane and etcd member. It therefore provides an HA setup where losing a control plane instance or an etcd member has less impact and does not affect the cluster redundancy as much as the stacked HA topology.

However, this topology requires twice the number of hosts as the stacked HA topology. A minimum of three hosts for control plane nodes and three hosts for etcd nodes are required for an HA cluster with this topology.



What's next

- Set up a highly available cluster with kubeadm

2.2.1.6 - Creating Highly Available Clusters with kubeadm

This page explains two different approaches to setting up a highly available Kubernetes cluster using kubeadm:

- With stacked control plane nodes. This approach requires less infrastructure. The etcd members and control plane nodes are co-located.
- With an external etcd cluster. This approach requires more infrastructure. The control plane nodes and etcd members are separated.

Before proceeding, you should carefully consider which approach best meets the needs of your applications and environment. Options for Highly Available topology outlines the advantages and disadvantages of each.

If you encounter issues with setting up the HA cluster, please report these in the kubeadm issue tracker.

See also the upgrade documentation.

Caution: This page does not address running your cluster on a cloud provider. In a cloud environment, neither approach documented here works with Service objects of type LoadBalancer, or with dynamic PersistentVolumes.

Before you begin

The prerequisites depend on which topology you have selected for your cluster's control plane:

Stacked etcd | External etcd

You need:

- Three or more machines that meet kubeadm's minimum requirements for the control-plane nodes. Having an odd number of control plane nodes can help with leader selection in the case of machine or zone failure.
  - including a container runtime, already set up and working
- Three or more machines that meet kubeadm's minimum requirements for the workers
  - including a container runtime, already set up and working
- Full network connectivity between all machines in the cluster (public or private network)
- Superuser privileges on all machines using `sudo`
  - You can use a different tool; this guide uses `sudo` in the examples.
- SSH access from one device to all nodes in the system
- `kubeadm` and `kubelet` already installed on all machines.

See Stacked etcd topology for context.

Container images

Each host should have access read and fetch images from the Kubernetes container image registry, `k8s.gcr.io` . If you want to deploy a highly-available cluster where the hosts do not have access to pull images, this is possible. You must ensure by some other means that the correct container images are already available on the relevant hosts.

Command line interface

To manage Kubernetes once your cluster is set up, you should install kubectl on your PC. It is also useful to install the `kubectl` tool on each control plane node, as this can be helpful for troubleshooting.

First steps for both methods

Create load balancer for kube-apiserver

> **Note:** There are many configurations for load balancers. The following example is only one option. Your cluster requirements may need a different configuration.

1. Create a kube-apiserver load balancer with a name that resolves to DNS.
   - In a cloud environment you should place your control plane nodes behind a TCP forwarding load balancer. This load balancer distributes traffic to all healthy control plane nodes in its target list. The health check for an apiserver is a TCP check on the port the kube-apiserver listens on (default value `:6443`).
   - It is not recommended to use an IP address directly in a cloud environment.
   - The load balancer must be able to communicate with all control plane nodes on the apiserver port. It must also allow incoming traffic on its listening port.
   - Make sure the address of the load balancer always matches the address of kubeadm's `ControlPlaneEndpoint`.
   - Read the Options for Software Load Balancing guide for more details.
2. Add the first control plane node to the load balancer, and test the connection:

   ```
   nc -v <LOAD_BALANCER_IP> <PORT>
   ```

   A connection refused error is expected because the API server is not yet running. A timeout, however, means the load balancer cannot communicate with the control plane node. If a timeout occurs, reconfigure the load balancer to communicate with the control plane node.
3. Add the remaining control plane nodes to the load balancer target group.

Stacked control plane and etcd nodes

Steps for the first control plane node

1. Initialize the control plane:

   ```
   sudo kubeadm init --control-plane-endpoint "LOAD_BALANCER_DNS:LOAD_BALANCER_PORT" -
   ```

   - You can use the `--kubernetes-version` flag to set the Kubernetes version to use. It is recommended that the versions of kubeadm, kubelet, kubectl and Kubernetes match.
   - The `--control-plane-endpoint` flag should be set to the address or DNS and port of the load balancer.
   - The `--upload-certs` flag is used to upload the certificates that should be shared across all the control-plane instances to the cluster. If instead, you prefer to copy certs across control-plane nodes manually or using automation tools, please

remove this flag and refer to Manual certificate distribution section below.

> **Note:** The `kubeadm init` flags `--config` and `--certificate-key` cannot be mixed, therefore if you want to use the kubeadm configuration you must add the `certificateKey` field in the appropriate config locations (under `InitConfiguration` and `JoinConfiguration: controlPlane`).

> **Note:** Some CNI network plugins require additional configuration, for example specifying the pod IP CIDR, while others do not. See the CNI network documentation. To add a pod CIDR pass the flag `--pod-network-cidr`, or if you are using a kubeadm configuration file set the `podSubnet` field under the `networking` object of `ClusterConfiguration`.

The output looks similar to:

```
...
You can now join any number of control-plane node by running the following command
    kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discovery-tok

Please note that the certificate-key gives access to cluster sensitive data, keep i
As a safeguard, uploaded-certs will be deleted in two hours; If necessary, you can

Then you can join any number of worker nodes by running the following on each as ro
    kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discovery-tok
```

- Copy this output to a text file. You will need it later to join control plane and worker nodes to the cluster.
- When `--upload-certs` is used with `kubeadm init`, the certificates of the primary control plane are encrypted and uploaded in the `kubeadm-certs` Secret.
- To re-upload the certificates and generate a new decryption key, use the following command on a control plane node that is already joined to the cluster:

  ```
  sudo kubeadm init phase upload-certs --upload-certs
  ```

- You can also specify a custom `--certificate-key` during `init` that can later be used by `join`. To generate such a key you can use the following command:

  ```
  kubeadm certs certificate-key
  ```

> **Note:** The `kubeadm-certs` Secret and decryption key expire after two hours.

> **Caution:** As stated in the command output, the certificate key gives access to cluster sensitive data, keep it secret!

2. Apply the CNI plugin of your choice:
   Follow these instructions to install the CNI provider. Make sure the configuration corresponds to the Pod CIDR specified in the kubeadm configuration file (if applicable).

> **Note:** You must pick a network plugin that suits your use case and deploy it before you move on to next step. If you don't do this, you will not be able to launch your cluster properly.

3. Type the following and watch the pods of the control plane components get started:

```
kubectl get pod -n kube-system -w
```

Steps for the rest of the control plane nodes

For each additional control plane node you should:

1. Execute the join command that was previously given to you by the `kubeadm init` output on the first node. It should look something like this:

```
sudo kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discovery-to
```

   - The `--control-plane` flag tells `kubeadm join` to create a new control plane.
   - The `--certificate-key ...` will cause the control plane certificates to be downloaded from the `kubeadm-certs` Secret in the cluster and be decrypted using the given key.

You can join multiple control-plane nodes in parallel.

External etcd nodes

Setting up a cluster with external etcd nodes is similar to the procedure used for stacked etcd with the exception that you should setup etcd first, and you should pass the etcd information in the kubeadm config file.

Set up the etcd cluster

1. Follow these instructions to set up the etcd cluster.
2. Setup SSH as described here.
3. Copy the following files from any etcd node in the cluster to the first control plane node:

```
export CONTROL_PLANE="ubuntu@10.0.0.7"
scp /etc/kubernetes/pki/etcd/ca.crt "${CONTROL_PLANE}":
scp /etc/kubernetes/pki/apiserver-etcd-client.crt "${CONTROL_PLANE}":
scp /etc/kubernetes/pki/apiserver-etcd-client.key "${CONTROL_PLANE}":
```

   - Replace the value of `CONTROL_PLANE` with the `user@host` of the first control-plane node.

Set up the first control plane node

1. Create a file called `kubeadm-config.yaml` with the following contents:

```
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
kubernetesVersion: stable
controlPlaneEndpoint: "LOAD_BALANCER_DNS:LOAD_BALANCER_PORT" # change this (see bel
etcd:
  external:
    endpoints:
```

```
    - https://ETCD_0_IP:2379 # change ETCD_0_IP appropriately
    - https://ETCD_1_IP:2379 # change ETCD_1_IP appropriately
    - https://ETCD_2_IP:2379 # change ETCD_2_IP appropriately
  caFile: /etc/kubernetes/pki/etcd/ca.crt
  certFile: /etc/kubernetes/pki/apiserver-etcd-client.crt
  keyFile: /etc/kubernetes/pki/apiserver-etcd-client.key
```

Note: The difference between stacked etcd and external etcd here is that the external etcd setup requires a configuration file with the etcd endpoints under the `external` object for `etcd`. In the case of the stacked etcd topology, this is managed automatically.

- Replace the following variables in the config template with the appropriate values for your cluster:
  - `LOAD_BALANCER_DNS`
  - `LOAD_BALANCER_PORT`
  - `ETCD_0_IP`
  - `ETCD_1_IP`
  - `ETCD_2_IP`

The following steps are similar to the stacked etcd setup:

1. Run `sudo kubeadm init --config kubeadm-config.yaml --upload-certs` on this node.
2. Write the output join commands that are returned to a text file for later use.
3. Apply the CNI plugin of your choice.

Note: You must pick a network plugin that suits your use case and deploy it before you move on to next step. If you don't do this, you will not be able to launch your cluster properly.

Steps for the rest of the control plane nodes

The steps are the same as for the stacked etcd setup:

- Make sure the first control plane node is fully initialized.
- Join each control plane node with the join command you saved to a text file. It's recommended to join the control plane nodes one at a time.
- Don't forget that the decryption key from `--certificate-key` expires after two hours, by default.

Common tasks after bootstrapping control plane

Install workers

Worker nodes can be joined to the cluster with the command you stored previously as the output from the `kubeadm init` command:

```
sudo kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discovery-token-ca
```

Manual certificate distribution

If you choose to not use `kubeadm init` with the `--upload-certs` flag this means that you are going to have to manually copy the certificates from the primary control plane node to the joining control plane nodes.

There are many ways to do this. The following example uses `ssh` and `scp`:

SSH is required if you want to control all nodes from a single machine.

1. Enable ssh-agent on your main device that has access to all other nodes in the system:

   ```
   eval $(ssh-agent)
   ```

2. Add your SSH identity to the session:

   ```
   ssh-add ~/.ssh/path_to_private_key
   ```

3. SSH between nodes to check that the connection is working correctly.

   - When you SSH to any node, add the `-A` flag. This flag allows the node that you have logged into via SSH to access the SSH agent on your PC. Consider alternative methods if you do not fully trust the security of your user session on the node.

     ```
     ssh -A 10.0.0.7
     ```

   - When using sudo on any node, make sure to preserve the environment so SSH forwarding works:

     ```
     sudo -E -s
     ```

4. After configuring SSH on all the nodes you should run the following script on the first control plane node after running `kubeadm init`. This script will copy the certificates from the first control plane node to the other control plane nodes:

   In the following example, replace `CONTROL_PLANE_IPS` with the IP addresses of the other control plane nodes.

   ```
   USER=ubuntu # customizable
   CONTROL_PLANE_IPS="10.0.0.7 10.0.0.8"
   for host in ${CONTROL_PLANE_IPS}; do
       scp /etc/kubernetes/pki/ca.crt "${USER}"@$host:
       scp /etc/kubernetes/pki/ca.key "${USER}"@$host:
       scp /etc/kubernetes/pki/sa.key "${USER}"@$host:
       scp /etc/kubernetes/pki/sa.pub "${USER}"@$host:
       scp /etc/kubernetes/pki/front-proxy-ca.crt "${USER}"@$host:
       scp /etc/kubernetes/pki/front-proxy-ca.key "${USER}"@$host:
       scp /etc/kubernetes/pki/etcd/ca.crt "${USER}"@$host:etcd-ca.crt
       # Skip the next line if you are using external etcd
       scp /etc/kubernetes/pki/etcd/ca.key "${USER}"@$host:etcd-ca.key
   done
   ```

   Caution: Copy only the certificates in the above list. kubeadm will take care of generating the rest of the certificates with the required SANs for the joining control-plane instances. If you copy all the certificates by mistake, the creation of additional nodes could fail due to a lack of required SANs.

5. Then on each joining control plane node you have to run the following script before running `kubeadm join`. This script will move the previously copied certificates from the home directory to `/etc/kubernetes/pki`:

```
USER=ubuntu # customizable
mkdir -p /etc/kubernetes/pki/etcd
mv /home/${USER}/ca.crt /etc/kubernetes/pki/
mv /home/${USER}/ca.key /etc/kubernetes/pki/
mv /home/${USER}/sa.pub /etc/kubernetes/pki/
mv /home/${USER}/sa.key /etc/kubernetes/pki/
mv /home/${USER}/front-proxy-ca.crt /etc/kubernetes/pki/
mv /home/${USER}/front-proxy-ca.key /etc/kubernetes/pki/
mv /home/${USER}/etcd-ca.crt /etc/kubernetes/pki/etcd/ca.crt
# Skip the next line if you are using external etcd
mv /home/${USER}/etcd-ca.key /etc/kubernetes/pki/etcd/ca.key
```

2.2.1.7 - Set up a High Availability etcd Cluster with kubeadm

> **Note:** While kubeadm is being used as the management tool for external etcd nodes in this guide, please note that kubeadm does not plan to support certificate rotation or upgrades for such nodes. The long term plan is to empower the tool etcdadm to manage these aspects.

By default, kubeadm runs a local etcd instance on each control plane node. It is also possible to treat the etcd cluster as external and provision etcd instances on separate hosts. The differences between the two approaches are covered in the Options for Highly Available topology page.

This task walks through the process of creating a high availability external etcd cluster of three members that can be used by kubeadm during cluster creation.

Before you begin

- Three hosts that can talk to each other over TCP ports 2379 and 2380. This document assumes these default ports. However, they are configurable through the kubeadm config file.
- Each host must have systemd and a bash compatible shell installed.
- Each host must have a container runtime, kubelet, and kubeadm installed.
- Each host should have access to the Kubernetes container image registry (`k8s.gcr.io`) or list/pull the required etcd image using `kubeadm config images list/pull` . This guide will setup etcd instances as static pods managed by a kubelet.
- Some infrastructure to copy files between hosts. For example `ssh` and `scp` can satisfy this requirement.

Setting up the cluster

The general approach is to generate all certs on one node and only distribute the *necessary* files to the other nodes.

> **Note:** kubeadm contains all the necessary crytographic machinery to generate the certificates described below; no other cryptographic tooling is required for this example.

> **Note:** The examples below use IPv4 addresses but you can also configure kubeadm, the kubelet and etcd to use IPv6 addresses. Dual-stack is supported by some Kubernetes options, but not by etcd. For more details on Kubernetes dual-stack support see Dual-stack support with kubeadm.

1. Configure the kubelet to be a service manager for etcd.

   > **Note:** You must do this on every host where etcd should be running.

   Since etcd was created first, you must override the service priority by creating a new unit file that has higher precedence than the kubeadm-provided kubelet unit file.

```
cat << EOF > /etc/systemd/system/kubelet.service.d/20-etcd-service-manager.conf
[Service]
ExecStart=
# Replace "systemd" with the cgroup driver of your container runtime. The default v
```

```
# Replace the value of "--container-runtime-endpoint" for a different container run
ExecStart=/usr/bin/kubelet --address=127.0.0.1 --pod-manifest-path=/etc/kubernetes/
Restart=always
EOF

systemctl daemon-reload
systemctl restart kubelet
```

Check the kubelet status to ensure it is running.

```
systemctl status kubelet
```

2. Create configuration files for kubeadm.

   Generate one kubeadm configuration file for each host that will have an etcd member running on it using the following script.

```
# Update HOST0, HOST1 and HOST2 with the IPs of your hosts
export HOST0=10.0.0.6
export HOST1=10.0.0.7
export HOST2=10.0.0.8

# Update NAME0, NAME1 and NAME2 with the hostnames of your hosts
export NAME0="infra0"
export NAME1="infra1"
export NAME2="infra2"

# Create temp directories to store files that will end up on other hosts.
mkdir -p /tmp/${HOST0}/ /tmp/${HOST1}/ /tmp/${HOST2}/

HOSTS=(${HOST0} ${HOST1} ${HOST2})
NAMES=(${NAME0} ${NAME1} ${NAME2})

for i in "${!HOSTS[@]}"; do
HOST=${HOSTS[$i]}
NAME=${NAMES[$i]}
cat << EOF > /tmp/${HOST}/kubeadmcfg.yaml
---
apiVersion: "kubeadm.k8s.io/v1beta3"
kind: InitConfiguration
nodeRegistration:
    name: ${NAME}
localAPIEndpoint:
    advertiseAddress: ${HOST}
---
apiVersion: "kubeadm.k8s.io/v1beta3"
kind: ClusterConfiguration
etcd:
    local:
        serverCertSANs:
        - "${HOST}"
        peerCertSANs:
        - "${HOST}"
        extraArgs:
            initial-cluster: ${NAMES[0]}=https://${HOSTS[0]}:2380,${NAMES[1]}=https
            initial-cluster-state: new
            name: ${NAME}
            listen-peer-urls: https://${HOST}:2380
            listen-client-urls: https://${HOST}:2379
            advertise-client-urls: https://${HOST}:2379
            initial-advertise-peer-urls: https://${HOST}:2380
EOF
done
```

3. Generate the certificate authority

   If you already have a CA then the only action that is copying the CA's `crt` and `key` file to `/etc/kubernetes/pki/etcd/ca.crt` and `/etc/kubernetes/pki/etcd/ca.key`. After those files have been copied, proceed to the next step, "Create certificates for each member".

   If you do not already have a CA then run this command on `$HOST0` (where you generated the configuration files for kubeadm).

   ```
   kubeadm init phase certs etcd-ca
   ```

   This creates two files

   - `/etc/kubernetes/pki/etcd/ca.crt`
   - `/etc/kubernetes/pki/etcd/ca.key`

4. Create certificates for each member

   ```
   kubeadm init phase certs etcd-server --config=/tmp/${HOST2}/kubeadmcfg.yaml
   kubeadm init phase certs etcd-peer --config=/tmp/${HOST2}/kubeadmcfg.yaml
   kubeadm init phase certs etcd-healthcheck-client --config=/tmp/${HOST2}/kubeadmcfg.
   kubeadm init phase certs apiserver-etcd-client --config=/tmp/${HOST2}/kubeadmcfg.ya
   cp -R /etc/kubernetes/pki /tmp/${HOST2}/
   # cleanup non-reusable certificates
   find /etc/kubernetes/pki -not -name ca.crt -not -name ca.key -type f -delete

   kubeadm init phase certs etcd-server --config=/tmp/${HOST1}/kubeadmcfg.yaml
   kubeadm init phase certs etcd-peer --config=/tmp/${HOST1}/kubeadmcfg.yaml
   kubeadm init phase certs etcd-healthcheck-client --config=/tmp/${HOST1}/kubeadmcfg.
   kubeadm init phase certs apiserver-etcd-client --config=/tmp/${HOST1}/kubeadmcfg.ya
   cp -R /etc/kubernetes/pki /tmp/${HOST1}/
   find /etc/kubernetes/pki -not -name ca.crt -not -name ca.key -type f -delete

   kubeadm init phase certs etcd-server --config=/tmp/${HOST0}/kubeadmcfg.yaml
   kubeadm init phase certs etcd-peer --config=/tmp/${HOST0}/kubeadmcfg.yaml
   kubeadm init phase certs etcd-healthcheck-client --config=/tmp/${HOST0}/kubeadmcfg.
   kubeadm init phase certs apiserver-etcd-client --config=/tmp/${HOST0}/kubeadmcfg.ya
   # No need to move the certs because they are for HOST0

   # clean up certs that should not be copied off this host
   find /tmp/${HOST2} -name ca.key -type f -delete
   find /tmp/${HOST1} -name ca.key -type f -delete
   ```

5. Copy certificates and kubeadm configs

   The certificates have been generated and now they must be moved to their respective hosts.

   ```
   USER=ubuntu
   HOST=${HOST1}
   scp -r /tmp/${HOST}/* ${USER}@${HOST}:
   ssh ${USER}@${HOST}
   USER@HOST $ sudo -Es
   root@HOST $ chown -R root:root pki
   root@HOST $ mv pki /etc/kubernetes/
   ```

6. Ensure all expected files exist

   The complete list of required files on `$HOST0` is:

```
/tmp/${HOST0}
└── kubeadmcfg.yaml
---
/etc/kubernetes/pki
├── apiserver-etcd-client.crt
├── apiserver-etcd-client.key
└── etcd
    ├── ca.crt
    ├── ca.key
    ├── healthcheck-client.crt
    ├── healthcheck-client.key
    ├── peer.crt
    ├── peer.key
    ├── server.crt
    └── server.key
```

On `$HOST1`:

```
$HOME
└── kubeadmcfg.yaml
---
/etc/kubernetes/pki
├── apiserver-etcd-client.crt
├── apiserver-etcd-client.key
└── etcd
    ├── ca.crt
    ├── healthcheck-client.crt
    ├── healthcheck-client.key
    ├── peer.crt
    ├── peer.key
    ├── server.crt
    └── server.key
```

On `$HOST2`

```
$HOME
└── kubeadmcfg.yaml
---
/etc/kubernetes/pki
├── apiserver-etcd-client.crt
├── apiserver-etcd-client.key
└── etcd
    ├── ca.crt
    ├── healthcheck-client.crt
    ├── healthcheck-client.key
    ├── peer.crt
    ├── peer.key
    ├── server.crt
    └── server.key
```

7. Create the static pod manifests

   Now that the certificates and configs are in place it's time to create the manifests. On each host run the `kubeadm` command to generate a static manifest for etcd.

```
root@HOST0 $ kubeadm init phase etcd local --config=/tmp/${HOST0}/kubeadmcfg.yaml
root@HOST1 $ kubeadm init phase etcd local --config=$HOME/kubeadmcfg.yaml
root@HOST2 $ kubeadm init phase etcd local --config=$HOME/kubeadmcfg.yaml
```

8. Optional: Check the cluster health

```
docker run --rm -it \
--net host \
```

```
-v /etc/kubernetes:/etc/kubernetes k8s.gcr.io/etcd:${ETCD_TAG} etcdctl \
--cert /etc/kubernetes/pki/etcd/peer.crt \
--key /etc/kubernetes/pki/etcd/peer.key \
--cacert /etc/kubernetes/pki/etcd/ca.crt \
--endpoints https://${HOST0}:2379 endpoint health --cluster
...
https://[HOST0 IP]:2379 is healthy: successfully committed proposal: took = 16.2833
https://[HOST1 IP]:2379 is healthy: successfully committed proposal: took = 19.4440
https://[HOST2 IP]:2379 is healthy: successfully committed proposal: took = 35.9264
```

- Set `${ETCD_TAG}` to the version tag of your etcd image. For example `3.4.3-0`. To see the etcd image and tag that kubeadm uses execute `kubeadm config images list --kubernetes-version ${K8S_VERSION}`, where `${K8S_VERSION}` is for example `v1.17.0`
- Set `${HOST0}` to the IP address of the host you are testing.

What's next

Once you have a working 3 member etcd cluster, you can continue setting up a highly available control plane using the external etcd method with kubeadm.

2.2.1.8 - Configuring each kubelet in your cluster using kubeadm

FEATURE STATE: `Kubernetes v1.11 [stable]`

The lifecycle of the kubeadm CLI tool is decoupled from the kubelet, which is a daemon that runs on each node within the Kubernetes cluster. The kubeadm CLI tool is executed by the user when Kubernetes is initialized or upgraded, whereas the kubelet is always running in the background.

Since the kubelet is a daemon, it needs to be maintained by some kind of an init system or service manager. When the kubelet is installed using DEBs or RPMs, systemd is configured to manage the kubelet. You can use a different service manager instead, but you need to configure it manually.

Some kubelet configuration details need to be the same across all kubelets involved in the cluster, while other configuration aspects need to be set on a per-kubelet basis to accommodate the different characteristics of a given machine (such as OS, storage, and networking). You can manage the configuration of your kubelets manually, but kubeadm now provides a `KubeletConfiguration` API type for managing your kubelet configurations centrally.

Kubelet configuration patterns

The following sections describe patterns to kubelet configuration that are simplified by using kubeadm, rather than managing the kubelet configuration for each Node manually.

Propagating cluster-level configuration to each kubelet

You can provide the kubelet with default values to be used by `kubeadm init` and `kubeadm join` commands. Interesting examples include using a different CRI runtime or setting the default subnet used by services.

If you want your services to use the subnet `10.96.0.0/12` as the default for services, you can pass the `--service-cidr` parameter to kubeadm:

```
kubeadm init --service-cidr 10.96.0.0/12
```

Virtual IPs for services are now allocated from this subnet. You also need to set the DNS address used by the kubelet, using the `--cluster-dns` flag. This setting needs to be the same for every kubelet on every manager and Node in the cluster. The kubelet provides a versioned, structured API object that can configure most parameters in the kubelet and push out this configuration to each running kubelet in the cluster. This object is called `KubeletConfiguration`. The `KubeletConfiguration` allows the user to specify flags such as the cluster DNS IP addresses expressed as a list of values to a camelCased key, illustrated by the following example:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
clusterDNS:
- 10.96.0.10
```

For more details on the `KubeletConfiguration` have a look at this section.

Providing instance-specific configuration details

Some hosts require specific kubelet configurations due to differences in hardware, operating system, networking, or other host-specific parameters. The following list provides a few examples.

- The path to the DNS resolution file, as specified by the `--resolv-conf` kubelet configuration flag, may differ among operating systems, or depending on whether you are using `systemd-resolved`. If this path is wrong, DNS resolution will fail on the Node whose kubelet is configured incorrectly.
- The Node API object `.metadata.name` is set to the machine's hostname by default, unless you are using a cloud provider. You can use the `--hostname-override` flag to override the default behavior if you need to specify a Node name different from the machine's hostname.
- Currently, the kubelet cannot automatically detect the cgroup driver used by the CRI runtime, but the value of `--cgroup-driver` must match the cgroup driver used by the CRI runtime to ensure the health of the kubelet.
- Depending on the CRI runtime your cluster uses, you may need to specify different flags to the kubelet. For instance, when using Docker, you need to specify flags such as `--network-plugin=cni`, but if you are using an external runtime, you need to specify `--container-runtime=remote` and specify the CRI endpoint using the `--container-runtime-endpoint=<path>`.

You can specify these flags by configuring an individual kubelet's configuration in your service manager, such as systemd.

Configure kubelets using kubeadm

It is possible to configure the kubelet that kubeadm will start if a custom `KubeletConfiguration` API object is passed with a configuration file like so `kubeadm ... --config some-config-file.yaml`.

By calling `kubeadm config print init-defaults --component-configs KubeletConfiguration` you can see all the default values for this structure.

Also have a look at the reference for the KubeletConfiguration for more information on the individual fields.

Workflow when using `kubeadm init`

When you call `kubeadm init`, the kubelet configuration is marshalled to disk at `/var/lib/kubelet/config.yaml`, and also uploaded to a ConfigMap in the cluster. The ConfigMap is named `kubelet-config-1.X`, where `X` is the minor version of the Kubernetes version you are initializing. A kubelet configuration file is also written to `/etc/kubernetes/kubelet.conf` with the baseline cluster-wide configuration for all kubelets in the cluster. This configuration file points to the client certificates that allow the kubelet to communicate with the API server. This addresses the need to propagate cluster-level configuration to each kubelet.

To address the second pattern of providing instance-specific configuration details, kubeadm writes an environment file to `/var/lib/kubelet/kubeadm-flags.env`, which contains a list of flags to pass to the kubelet when it starts. The flags are presented in the file like this:

```
KUBELET_KUBEADM_ARGS="--flag1=value1 --flag2=value2 ..."
```

In addition to the flags used when starting the kubelet, the file also contains dynamic parameters such as the cgroup driver and whether to use a different CRI runtime socket (`--cri-socket`).

After marshalling these two files to disk, kubeadm attempts to run the following two commands, if you are using systemd:

```
systemctl daemon-reload && systemctl restart kubelet
```

If the reload and restart are successful, the normal `kubeadm init` workflow continues.

Workflow when using `kubeadm join`

When you run `kubeadm join`, kubeadm uses the Bootstrap Token credential to perform a TLS bootstrap, which fetches the credential needed to download the `kubelet-config-1.X` ConfigMap and writes it to `/var/lib/kubelet/config.yaml`. The dynamic environment file is generated in exactly the same way as `kubeadm init`.

Next, `kubeadm` runs the following two commands to load the new configuration into the kubelet:

```
systemctl daemon-reload && systemctl restart kubelet
```

After the kubelet loads the new configuration, kubeadm writes the `/etc/kubernetes/bootstrap-kubelet.conf` KubeConfig file, which contains a CA certificate and Bootstrap Token. These are used by the kubelet to perform the TLS Bootstrap and obtain a unique credential, which is stored in `/etc/kubernetes/kubelet.conf`.

When the `/etc/kubernetes/kubelet.conf` file is written, the kubelet has finished performing the TLS Bootstrap. Kubeadm deletes the `/etc/kubernetes/bootstrap-kubelet.conf` file after completing the TLS Bootstrap.

The kubelet drop-in file for systemd

`kubeadm` ships with configuration for how systemd should run the kubelet. Note that the kubeadm CLI command never touches this drop-in file.

This configuration file installed by the `kubeadm` DEB or RPM package is written to `/etc/systemd/system/kubelet.service.d/10-kubeadm.conf` and is used by systemd. It augments the basic `kubelet.service` for RPM or `kubelet.service` for DEB:

```
[Service]
Environment="KUBELET_KUBECONFIG_ARGS=--bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kub
--kubeconfig=/etc/kubernetes/kubelet.conf"
Environment="KUBELET_CONFIG_ARGS=--config=/var/lib/kubelet/config.yaml"
# This is a file that "kubeadm init" and "kubeadm join" generate at runtime, populating
the KUBELET_KUBEADM_ARGS variable dynamically
EnvironmentFile=-/var/lib/kubelet/kubeadm-flags.env
# This is a file that the user can use for overrides of the kubelet args as a last resort
# the user should use the .NodeRegistration.KubeletExtraArgs object in the configuration
# KUBELET_EXTRA_ARGS should be sourced from this file.
EnvironmentFile=-/etc/default/kubelet
ExecStart=
ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM
```

This file specifies the default locations for all of the files managed by kubeadm for the kubelet.

- The KubeConfig file to use for the TLS Bootstrap is `/etc/kubernetes/bootstrap-kubelet.conf`, but it is only used if `/etc/kubernetes/kubelet.conf` does not exist.
- The KubeConfig file with the unique kubelet identity is `/etc/kubernetes/kubelet.conf`.
- The file containing the kubelet's ComponentConfig is `/var/lib/kubelet/config.yaml`.
- The dynamic environment file that contains `KUBELET_KUBEADM_ARGS` is sourced from `/var/lib/kubelet/kubeadm-flags.env`.

- The file that can contain user-specified flag overrides with `KUBELET_EXTRA_ARGS` is sourced from `/etc/default/kubelet` (for DEBs), or `/etc/sysconfig/kubelet` (for RPMs). `KUBELET_EXTRA_ARGS` is last in the flag chain and has the highest priority in the event of conflicting settings.

Kubernetes binaries and package contents

The DEB and RPM packages shipped with the Kubernetes releases are:

Package name	Description
`kubeadm`	Installs the `/usr/bin/kubeadm` CLI tool and the kubelet drop-in file for the kubelet.
`kubelet`	Installs the kubelet binary in `/usr/bin` and CNI binaries in `/opt/cni/bin`.
`kubectl`	Installs the `/usr/bin/kubectl` binary.
`cri-tools`	Installs the `/usr/bin/crictl` binary from the cri-tools git repository.

2.2.1.9 - Dual-stack support with kubeadm

FEATURE STATE: `Kubernetes v1.23 [stable]`

Your Kubernetes cluster includes dual-stack networking, which means that cluster networking lets you use either address family. In a cluster, the control plane can assign both an IPv4 address and an IPv6 address to a single Pod or a Service.

Before you begin

You need to have installed the kubeadm tool, following the steps from Installing kubeadm.

For each server that you want to use as a node, make sure it allows IPv6 forwarding. On Linux, you can set this by running run `sysctl -w net.ipv6.conf.all.forwarding=1` as the root user on each server.

You need to have an IPv4 and and IPv6 address range to use. Cluster operators typically use private address ranges for IPv4. For IPv6, a cluster operator typically chooses a global unicast address block from within `2000::/3`, using a range that is assigned to the operator. You don't have to route the cluster's IP address ranges to the public internet.

The size of the IP address allocations should be suitable for the number of Pods and Services that you are planning to run.

> **Note:** If you are upgrading an existing cluster with the `kubeadm upgrade` command, `kubeadm` does not support making modifications to the pod IP address range ("cluster CIDR") nor to the cluster's Service address range ("Service CIDR").

Create a dual-stack cluster

To create a dual-stack cluster with `kubeadm init` you can pass command line arguments similar to the following example:

```
# These address ranges are examples
kubeadm init --pod-network-cidr=10.244.0.0/16,2001:db8:42:0::/56 --service-cidr=10.96.0.
```

To make things clearer, here is an example kubeadm configuration file `kubeadm-config.yaml` for the primary dual-stack control plane node.

```
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
networking:
  podSubnet: 10.244.0.0/16,2001:db8:42:0::/56
  serviceSubnet: 10.96.0.0/16,2001:db8:42:1::/112
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
localAPIEndpoint:
  advertiseAddress: "10.100.0.1"
  bindPort: 6443
nodeRegistration:
  kubeletExtraArgs:
    node-ip: 10.100.0.2,fd00:1:2:3::2
```

`advertiseAddress` in InitConfiguration specifies the IP address that the API Server will advertise it is listening on. The value of `advertiseAddress` equals the `--apiserver-advertise-address` flag of `kubeadm init`

Run kubeadm to initiate the dual-stack control plane node:

```
kubeadm init --config=kubeadm-config.yaml
```

The kube-controller-manager flags `--node-cidr-mask-size-ipv4|--node-cidr-mask-size-ipv6` are set with default values. See configure IPv4/IPv6 dual stack.

> **Note:** The `--apiserver-advertise-address` flag does not support dual-stack.

Join a node to dual-stack cluster

Before joining a node, make sure that the node has IPv6 routable network interface and allows IPv6 forwarding.

Here is an example kubeadm configuration file `kubeadm-config.yaml` for joining a worker node to the cluster.

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: JoinConfiguration
discovery:
  bootstrapToken:
    apiServerEndpoint: 10.100.0.1:6443
    token: "clvldh.vjjwg16ucnhp94qr"
    caCertHashes:
    - "sha256:a4863cde706cfc580a439f842cc65d5ef112b7b2be31628513a9881cf0d9fe0e"
    # change auth info above to match the actual token and CA certificate hash for your c
nodeRegistration:
  kubeletExtraArgs:
    node-ip: 10.100.0.3,fd00:1:2:3::3
```

Also, here is an example kubeadm configuration file `kubeadm-config.yaml` for joining another control plane node to the cluster.

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: JoinConfiguration
controlPlane:
  localAPIEndpoint:
    advertiseAddress: "10.100.0.2"
    bindPort: 6443
discovery:
  bootstrapToken:
    apiServerEndpoint: 10.100.0.1:6443
    token: "clvldh.vjjwg16ucnhp94qr"
    caCertHashes:
    - "sha256:a4863cde706cfc580a439f842cc65d5ef112b7b2be31628513a9881cf0d9fe0e"
    # change auth info above to match the actual token and CA certificate hash for your c
nodeRegistration:
  kubeletExtraArgs:
    node-ip: 10.100.0.4,fd00:1:2:3::4
```

`advertiseAddress` in JoinConfiguration.controlPlane specifies the IP address that the API Server will advertise it is listening on. The value of `advertiseAddress` equals the `--apiserver-advertise-address` flag of `kubeadm join` .

```
kubeadm join --config=kubeadm-config.yaml
```

Create a single-stack cluster

> **Note:** Dual-stack support doesn't mean that you need to use dual-stack addressing. You can deploy a single-stack cluster that has the dual-stack networking feature enabled.

To make things more clear, here is an example kubeadm configuration file `kubeadm-config.yaml` for the single-stack control plane node.

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
networking:
  podSubnet: 10.244.0.0/16
  serviceSubnet: 10.96.0.0/16
```

What's next

- Validate IPv4/IPv6 dual-stack networking
- Read about Dual-stack cluster networking
- Learn more about the kubeadm configuration format

2.2.2 - Installing Kubernetes with kops

This quickstart shows you how to easily install a Kubernetes cluster on AWS. It uses a tool called `kops` .

kops is an automated provisioning system:

- Fully automated installation
- Uses DNS to identify clusters
- Self-healing: everything runs in Auto-Scaling Groups
- Multiple OS support (Debian, Ubuntu 16.04 supported, CentOS & RHEL, Amazon Linux and CoreOS) - see the images.md
- High-Availability support - see the high_availability.md
- Can directly provision, or generate terraform manifests - see the terraform.md

Before you begin

- You must have kubectl installed.
- You must install `kops` on a 64-bit (AMD64 and Intel 64) device architecture.
- You must have an AWS account, generate IAM keys and configure them. The IAM user will need adequate permissions.

Creating a cluster

(1/5) Install kops

Installation

Download kops from the releases page (it is also convenient to build from source):

macOS Linux

Download the latest release with the command:

```
curl -LO https://github.com/kubernetes/kops/releases/download/$(curl -s https://api.
```

To download a specific version, replace the following portion of the command with the specific kops version.

```
$(curl -s https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag_na
```

For example, to download kops version v1.20.0 type:

```
curl -LO https://github.com/kubernetes/kops/releases/download/v1.20.0/kops-darwin-am
```

Make the kops binary executable.

```
chmod +x kops-darwin-amd64
```

Move the kops binary in to your PATH.

```
sudo mv kops-darwin-amd64 /usr/local/bin/kops
```

You can also install kops using Homebrew.

```
brew update && brew install kops
```

(2/5) Create a route53 domain for your cluster

kops uses DNS for discovery, both inside the cluster and outside, so that you can reach the kubernetes API server from clients.

kops has a strong opinion on the cluster name: it should be a valid DNS name. By doing so you will no longer get your clusters confused, you can share clusters with your colleagues unambiguously, and you can reach them without relying on remembering an IP address.

You can, and probably should, use subdomains to divide your clusters. As our example we will use `useast1.dev.example.com`. The API server endpoint will then be `api.useast1.dev.example.com`.

A Route53 hosted zone can serve subdomains. Your hosted zone could be `useast1.dev.example.com`, but also `dev.example.com` or even `example.com`. kops works with any of these, so typically you choose for organization reasons (e.g. you are allowed to create records under `dev.example.com`, but not under `example.com`).

Let's assume you're using `dev.example.com` as your hosted zone. You create that hosted zone using the normal process, or with a command such as `aws route53 create-hosted-zone --name dev.example.com --caller-reference 1`.

You must then set up your NS records in the parent domain, so that records in the domain will resolve. Here, you would create NS records in `example.com` for `dev`. If it is a root domain name you would configure the NS records at your domain registrar (e.g. `example.com` would need to be configured where you bought `example.com`).

Verify your route53 domain setup (it is the #1 cause of problems!). You can double-check that your cluster is configured correctly if you have the dig tool by running:

```
dig NS dev.example.com
```

You should see the 4 NS records that Route53 assigned your hosted zone.

(3/5) Create an S3 bucket to store your clusters state

kops lets you manage your clusters even after installation. To do this, it must keep track of the clusters that you have created, along with their configuration, the keys they are using etc. This information is stored in an S3 bucket. S3 permissions are used to control access to the bucket.

Multiple clusters can use the same S3 bucket, and you can share an S3 bucket between your colleagues that administer the same clusters - this is much easier than passing around kubecfg files. But anyone with access to the S3 bucket will have administrative access to all your clusters, so you don't want to share it beyond the operations team.

So typically you have one S3 bucket for each ops team (and often the name will correspond to the name of the hosted zone above!)

In our example, we chose `dev.example.com` as our hosted zone, so let's pick `clusters.dev.example.com` as the S3 bucket name.

- Export `AWS_PROFILE` (if you need to select a profile for the AWS CLI to work)
- Create the S3 bucket using `aws s3 mb s3://clusters.dev.example.com`
- You can `export KOPS_STATE_STORE=s3://clusters.dev.example.com` and then kops will use this location by default. We suggest putting this in your bash profile or similar.

(4/5) Build your cluster configuration

Run `kops create cluster` to create your cluster configuration:

```
kops create cluster --zones=us-east-1c useast1.dev.example.com
```

kops will create the configuration for your cluster. Note that it *only* creates the configuration, it does not actually create the cloud resources - you'll do that in the next step with a `kops update cluster`. This give you an opportunity to review the configuration or change it.

It prints commands you can use to explore further:

- List your clusters with: `kops get cluster`
- Edit this cluster with: `kops edit cluster useast1.dev.example.com`
- Edit your node instance group: `kops edit ig --name=useast1.dev.example.com nodes`
- Edit your master instance group: `kops edit ig --name=useast1.dev.example.com master-us-east-1c`

If this is your first time using kops, do spend a few minutes to try those out! An instance group is a set of instances, which will be registered as kubernetes nodes. On AWS this is implemented via auto-scaling-groups. You can have several instance groups, for example if you wanted nodes that are a mix of spot and on-demand instances, or GPU and non-GPU instances.

(5/5) Create the cluster in AWS

Run "kops update cluster" to create your cluster in AWS:

```
kops update cluster useast1.dev.example.com --yes
```

That takes a few seconds to run, but then your cluster will likely take a few minutes to actually be ready. `kops update cluster` will be the tool you'll use whenever you change the configuration of your cluster; it applies the changes you have made to the configuration to your cluster - reconfiguring AWS or kubernetes as needed.

For example, after you `kops edit ig nodes`, then `kops update cluster --yes` to apply your configuration, and sometimes you will also have to `kops rolling-update cluster` to roll out the configuration immediately.

Without `--yes`, `kops update cluster` will show you a preview of what it is going to do. This is handy for production clusters!

Explore other add-ons

See the list of add-ons to explore other add-ons, including tools for logging, monitoring, network policy, visualization, and control of your Kubernetes cluster.

Cleanup

- To delete your cluster: `kops delete cluster useast1.dev.example.com --yes`

What's next

- Learn more about Kubernetes concepts and `kubectl`.
- Learn more about `kops` advanced usage for tutorials, best practices and advanced configuration options.
- Follow `kops` community discussions on Slack: community discussions
- Contribute to `kops` by addressing or raising an issue GitHub Issues

2.2.3 - Installing Kubernetes with Kubespray

This quickstart helps to install a Kubernetes cluster hosted on GCE, Azure, OpenStack, AWS, vSphere, Packet (bare metal), Oracle Cloud Infrastructure (Experimental) or Baremetal with Kubespray.

Kubespray is a composition of Ansible playbooks, inventory, provisioning tools, and domain knowledge for generic OS/Kubernetes clusters configuration management tasks. Kubespray provides:

- a highly available cluster
- composable attributes
- support for most popular Linux distributions
  - Ubuntu 16.04, 18.04, 20.04
  - CentOS/RHEL/Oracle Linux 7, 8
  - Debian Buster, Jessie, Stretch, Wheezy
  - Fedora 31, 32
  - Fedora CoreOS
  - openSUSE Leap 15
  - Flatcar Container Linux by Kinvolk
- continuous integration tests

To choose a tool which best fits your use case, read this comparison to kubeadm and kops.

Creating a cluster

(1/5) Meet the underlay requirements

Provision servers with the following requirements:

- **Ansible v2.9 and python-netaddr are installed on the machine that will run Ansible commands**
- **Jinja 2.11 (or newer) is required to run the Ansible Playbooks**
- The target servers must have access to the Internet in order to pull docker images. Otherwise, additional configuration is required (See Offline Environment)
- The target servers are configured to allow **IPv4 forwarding**
- **Your ssh key must be copied** to all the servers in your inventory
- **Firewalls are not managed by kubespray**. You'll need to implement appropriate rules as needed. You should disable your firewall in order to avoid any issues during deployment
- If kubespray is ran from a non-root user account, correct privilege escalation method should be configured in the target servers and the `ansible_become` flag or command parameters `--become` or `-b` should be specified

Kubespray provides the following utilities to help provision your environment:

- Terraform scripts for the following cloud providers:
  - AWS
  - OpenStack
  - Packet

(2/5) Compose an inventory file

After you provision your servers, create an inventory file for Ansible. You can do this manually or via a dynamic inventory script. For more information, see "Building your own inventory".

(3/5) Plan your cluster deployment

Kubespray provides the ability to customize many aspects of the deployment:

- Choice deployment mode: kubeadm or non-kubeadm
- CNI (networking) plugins
- DNS configuration
- Choice of control plane: native/binary or containerized
- Component versions
- Calico route reflectors
- Component runtime options
  - Docker
  - containerd
  - CRI-O
- Certificate generation methods

Kubespray customizations can be made to a variable file. If you are getting started with Kubespray, consider using the Kubespray defaults to deploy your cluster and explore Kubernetes.

(4/5) Deploy a Cluster

Next, deploy your cluster:

Cluster deployment using ansible-playbook.

```
ansible-playbook -i your/inventory/inventory.ini cluster.yml -b -v \
  --private-key=~/.ssh/private_key
```

Large deployments (100+ nodes) may require specific adjustments for best results.

(5/5) Verify the deployment

Kubespray provides a way to verify inter-pod connectivity and DNS resolve with Netchecker. Netchecker ensures the netchecker-agents pods can resolve DNS requests and ping each over within the default namespace. Those pods mimic similar behavior as the rest of the workloads and serve as cluster health indicators.

Cluster operations

Kubespray provides additional playbooks to manage your cluster: *scale* and *upgrade*.

Scale your cluster

You can add worker nodes from your cluster by running the scale playbook. For more information, see "Adding nodes". You can remove worker nodes from your cluster by running the remove-node playbook. For more information, see "Remove nodes".

Upgrade your cluster

You can upgrade your cluster by running the upgrade-cluster playbook. For more information, see "Upgrades".

Cleanup

You can reset your nodes and wipe out all components installed with Kubespray via the reset playbook.

> **Caution:** When running the reset playbook, be sure not to accidentally target your production cluster!

Feedback

- Slack Channel: #kubespray (You can get your invite here)
- GitHub Issues

What's next

Check out planned work on Kubespray's roadmap.

2.3 - Turnkey Cloud Solutions

This page provides a list of Kubernetes certified solution providers. From each provider page, you can learn how to install and setup production ready clusters.

Platform - Certified Kubernetes - Hosted (47)

Alibaba Cloud Container Service for Kubernetes MCap: $289.7B
Alibaba Cloud

Amazon Elastic Container Service for Kubernetes (EKS) MCap: $1.5T
Amazon Web Services

Azure Kubernetes Service (AKS) MCap: $2.2T
Microsoft

BoCloud BeyondContainer Funding: $29.8M
Bocloud

Catalyst Kubernetes Service ★2
Catalyst Cloud

China Mobile KCS MCap: $144.8B
China Mobile

DigitalOcean Kubernetes MCap: $6.5B
DigitalOcean

eBaoCloud
eBaoTech

ELASTX Private Kubernetes
Elastx

German Edge Cloud Kubernetes Services (GKS)
German Edge Cloud

Google Kubernetes Engine (GKE) MCap: $1.8T
Google

Huawei Cloud Container Engine (CCE)
Huawei Technologies

KubeSphere® (QKE) ★27
QingCloud Funding: $280.8M

Linode Kubernetes Engine
Linode

Mail.Ru Cloud Containers MCap: $14.2B
Mail.Ru Group

NIFCLOUD Kubernetes Service Hatoba MCap: $28.5B
Fujitsu

Nirmata Managed Kubernetes Funding: $4M
Nirmata

Nutanix Karbon MCap: $5.8B
Nutanix

OVH Managed Kubernetes Service MCap: $4.5B
OVHcloud

Rafay Funding: $33M
Rafay Systems

Red Hat OpenShift Dedicated MCap: $108.8B
Red Hat

Samsung Cloud Platform - SCP Kubernetes Engine (SKE)
Samsung SDS

SAP Certified Gardener MCap: $129.9B
SAP

Scaleway Kubernetes Kapsule
Scaleway

Taikun
Itera Technologies a.s.

Tencent Kubernetes Engine (TKE) MCap: $526.7B
Tencent

UCloud Kubernetes Service（UK8S） MCap: $1.7B
UCloud Information Technology

Vultr Kubernetes Engine
Vultr Holdings Corporation

ZTE TECS OpenPalette MCap: $20.1B
ZTE

View the full interactive la

2.4 - Windows in Kubernetes

2.4.1 - Windows containers in Kubernetes

Windows applications constitute a large portion of the services and applications that run in many organizations. Windows containers provide a way to encapsulate processes and package dependencies, making it easier to use DevOps practices and follow cloud native patterns for Windows applications.

Organizations with investments in Windows-based applications and Linux-based applications don't have to look for separate orchestrators to manage their workloads, leading to increased operational efficiencies across their deployments, regardless of operating system.

Windows nodes in Kubernetes

To enable the orchestration of Windows containers in Kubernetes, include Windows nodes in your existing Linux cluster. Scheduling Windows containers in Pods on Kubernetes is similar to scheduling Linux-based containers.

In order to run Windows containers, your Kubernetes cluster must include multiple operating systems. While you can only run the control plane on Linux, you can deploy worker nodes running either Windows or Linux depending on your workload needs.

Windows nodes are supported provided that the operating system is Windows Server 2019.

This document uses the term *Windows containers* to mean Windows containers with process isolation. Kubernetes does not support running Windows containers with Hyper-V isolation.

Resource management

On Linux nodes, cgroups are used as a pod boundary for resource control. Containers are created within that boundary for network, process and file system isolation. The Linux cgroup APIs can be used to gather CPU, I/O, and memory use statistics.

In contrast, Windows uses a *job object* per container with a system namespace filter to contain all processes in a container and provide logical isolation from the host. (Job objects are a Windows process isolation mechanism and are different from what Kubernetes refers to as a Job).

There is no way to run a Windows container without the namespace filtering in place. This means that system privileges cannot be asserted in the context of the host, and thus privileged containers are not available on Windows. Containers cannot assume an identity from the host because the Security Account Manager (SAM) is separate.

Memory reservations

Windows does not have an out-of-memory process killer as Linux does. Windows always treats all user-mode memory allocations as virtual, and pagefiles are mandatory (on Linux, the kubelet will by default not start with swap space enabled).

Windows nodes do not overcommit memory for processes running in containers. The net effect is that Windows won't reach out of memory conditions the same way Linux does, and processes page to disk instead of being subject to out of memory (OOM) termination. If memory is over-provisioned and all physical memory is exhausted, then paging can slow down performance.

You can place bounds on memory use for workloads using the kubelet parameters `--kubelet-reserve` and/or `--system-reserve` ; these account for memory usage on the node (outside of containers), and reduce NodeAllocatable. As you deploy workloads, set resource limits on

containers. This also subtracts from `NodeAllocatable` and prevents the scheduler from adding more pods once a node is full.

> **Note:** When you set memory resource limits for Windows containers, you should either set a limit and leave the memory request unspecified, or set the request equal to the limit.

On Windows, good practice to avoid over-provisioning is to configure the kubelet with a system reserved memory of at least 2GiB to account for Windows, Kubernetes and container runtime overheads.

CPU reservations

To account for CPU use by the operating system, the container runtime, and by Kubernetes host processes such as the kubelet, you can (and should) reserve a percentage of total CPU. You should determine this CPU reservation taking account of to the number of CPU cores available on the node. To decide on the CPU percentage to reserve, identify the maximum pod density for each node and monitor the CPU usage of the system services running there, then choose a value that meets your workload needs.

You can place bounds on CPU usage for workloads using the kubelet parameters `--kubelet-reserve` and/or `--system-reserve` to account for CPU usage on the node (outside of containers). This reduces `NodeAllocatable` . The cluster-wide scheduler then takes this reservation into account when determining pod placement.

On Windows, the kubelet supports a command-line flag to set the priority of the kubelet process: `--windows-priorityclass` . This flag allows the kubelet process to get more CPU time slices when compared to other processes running on the Windows host. More information on the allowable values and their meaning is available at Windows Priority Classes. To ensure that running Pods do not starve the kubelet of CPU cycles, set this flag to `ABOVE_NORMAL_PRIORITY_CLASS` or above.

Compatibility and limitations

Some node features are only available if you use a specific container runtime; others are not available on Windows nodes, including:

- HugePages: not supported for Windows containers
- Privileged containers: not supported for Windows containers
- TerminationGracePeriod: requires containerD

Not all features of shared namespaces are supported. See API compatibility for more details.

See Windows OS version compatibility for details on the Windows versions that Kubernetes is tested against.

From an API and kubectl perspective, Windows containers behave in much the same way as Linux-based containers. However, there are some notable differences in key functionality which are outlined in this section.

Comparison with Linux

Key Kubernetes elements work the same way in Windows as they do in Linux. This section refers to several key workload enablers and how they map to Windows.

- Pods

  A Pod is the basic building block of Kubernetes–the smallest and simplest unit in the Kubernetes object model that you create or deploy. You may not deploy Windows and Linux containers in the same Pod. All containers in a Pod are scheduled onto a single Node where each Node represents a specific platform and architecture. The following Pod capabilities, properties and events are supported with Windows containers:

- Single or multiple containers per Pod with process isolation and volume sharing
- Pod `status` fields
- Readiness and Liveness probes
- postStart & preStop container lifecycle events
- ConfigMap, Secrets: as environment variables or volumes
- `emptyDir` volumes
- Named pipe host mounts
- Resource limits
- OS field:

  FEATURE STATE: `Kubernetes v1.23 [alpha]`

  `.spec.os.name` should be set to `windows` to indicate that the current Pod uses Windows containers. `IdentifyPodOS` feature gate needs to be enabled for this field to be recognized and used by control plane components and kubelet.

  > **Note:**
  > If the `IdentifyPodOS` feature gate is enabled and you set the `.spec.os.name` field to `windows`, you must not set the following fields in the `.spec` of that Pod: * `spec.hostPID` * `spec.hostIPC` * `spec.securityContext.seLinuxOptions` * `spec.securityContext.seccompProfile` * `spec.securityContext.fsGroup` * `spec.securityContext.fsGroupChangePolicy` * `spec.securityContext.sysctls` * `spec.shareProcessNamespace` * `spec.securityContext.runAsUser` * `spec.securityContext.runAsGroup` * `spec.securityContext.supplementalGroups` * `spec.containers[*].securityContext.seLinuxOptions` * `spec.containers[*].securityContext.seccompProfile` * `spec.containers[*].securityContext.capabilities` * `spec.containers[*].securityContext.readOnlyRootFilesystem` * `spec.containers[*].securityContext.privileged` * `spec.containers[*].securityContext.allowPrivilegeEscalation` * `spec.containers[*].securityContext.procMount` * `spec.containers[*].securityContext.runAsUser` * `spec.containers[*].securityContext.runAsGroup`
  >
  > ```
  > Note: In this table, wildcards (*) indicate all elements in a list
  > ```

- Workload resources including:
  - ReplicaSet
  - Deployments
  - StatefulSets
  - DaemonSet
  - Job
  - CronJob
  - ReplicationController
- Services See Load balancing and Services for more details.

Pods, workload resources, and Services are critical elements to managing Windows workloads on Kubernetes. However, on their own they are not enough to enable the proper lifecycle management of Windows workloads in a dynamic cloud native environment. Kubernetes also supports:

- `kubectl exec`
- Pod and container metrics
- Horizontal pod autoscaling
- Resource quotas
- Scheduler preemption

Networking on Windows nodes

Networking for Windows containers is exposed through CNI plugins. Windows containers function similarly to virtual machines in regards to networking. Each container has a virtual network adapter (vNIC) which is connected to a Hyper-V virtual switch (vSwitch). The Host Networking Service (HNS) and the Host Compute Service (HCS) work together to create containers and attach container vNICs to networks. HCS is responsible for the management of containers whereas HNS is responsible for the management of networking resources such as:

- Virtual networks (including creation of vSwitches)
- Endpoints / vNICs
- Namespaces
- Policies including packet encapsulations, load-balancing rules, ACLs, and NAT rules.

Container networking

The Windows HNS and vSwitch implement namespacing and can create virtual NICs as needed for a pod or container. However, many configurations such as DNS, routes, and metrics are stored in the Windows registry database rather than as files inside `/etc`, which is how Linux stores those configurations. The Windows registry for the container is separate from that of the host, so concepts like mapping `/etc/resolv.conf` from the host into a container don't have the same effect they would on Linux. These must be configured using Windows APIs run in the context of that container. Therefore CNI implementations need to call the HNS instead of relying on file mappings to pass network details into the pod or container.

The following networking functionality is *not* supported on Windows nodes:

- Host networking mode
- Local NodePort access from the node itself (works for other nodes or external clients)
- More than 64 backend pods (or unique destination addresses) for a single Service
- IPv6 communication between Windows pods connected to overlay networks
- Local Traffic Policy in non-DSR mode
- Outbound communication using the ICMP protocol via the `win-overlay`, `win-bridge`, or using the Azure-CNI plugin.
  Specifically, the Windows data plane (VFP) doesn't support ICMP packet transpositions, and this means:
  - ICMP packets directed to destinations within the same network (such as pod to pod communication via ping) work as expected and without any limitations;
  - TCP/UDP packets work as expected and without any limitations;
  - ICMP packets directed to pass through a remote network (e.g. pod to external internet communication via ping) cannot be transposed and thus will not be routed back to their source;
  - Since TCP/UDP packets can still be transposed, you can substitute `ping <destination>` with `curl <destination>` to get some debugging insight into connectivity with the outside world.

Overlay networking support in kube-proxy is a beta feature. In addition, it requires KB4482887 to be installed on Windows Server 2019.

Network modes

Windows supports five different networking drivers/modes: L2bridge, L2tunnel, Overlay (beta), Transparent, and NAT. In a heterogeneous cluster with Windows and Linux worker nodes, you need to select a networking solution that is compatible on both Windows and Linux. The following out-of-tree plugins are supported on Windows, with recommendations on when to use each CNI:

Network Driver	Description	Container Packet Modifications	Network Plugins	Network Plugin Characteristics
L2bridge	Containers are attached to an external vSwitch. Containers are attached to the underlay network, although the physical network doesn't need to learn the container MACs because they are rewritten on ingress/egress.	MAC is rewritten to host MAC, IP may be rewritten to host IP using HNS OutboundNAT policy.	win-bridge, Azure-CNI, Flannel host-gateway uses win-bridge	win-bridge uses L2bridge network mode, connects containers to the underlay of hosts, offering best performance. Requires user-defined routes (UDR) for inter-node connectivity.
L2Tunnel	This is a special case of l2bridge, but only used on Azure. All packets are sent to the virtualization host where SDN policy is applied.	MAC rewritten, IP visible on the underlay network	Azure-CNI	Azure-CNI allows integration of containers with Azure vNET, and allows them to leverage the set of capabilities that Azure Virtual Network provides. For example, securely connect to Azure services or use Azure NSGs. See azure-cni for some examples
Overlay (Overlay networking for Windows in Kubernetes is in *alpha* stage)	Containers are given a vNIC connected to an external vSwitch. Each overlay network gets its own IP subnet, defined by a custom IP prefix.The overlay network driver uses VXLAN encapsulation.	Encapsulated with an outer header.	win-overlay, Flannel VXLAN (uses win-overlay)	win-overlay should be used when virtual container networks are desired to be isolated from underlay of hosts (e.g. for security reasons). Allows for IPs to be re-used for different overlay networks (which have different VNID tags) if you are restricted on IPs in your datacenter. This option requires KB4489899 on Windows Server 2019.

Network Driver	Description	Container Packet Modifications	Network Plugins	Network Plugin Characteristics
Transparent (special use case for ovn-kubernetes)	Requires an external vSwitch. Containers are attached to an external vSwitch which enables intra-pod communication via logical networks (logical switches and routers).	Packet is encapsulated either via GENEVE or STT tunneling to reach pods which are not on the same host. Packets are forwarded or dropped via the tunnel metadata information supplied by the ovn network controller. NAT is done for north-south communication.	ovn-kubernetes	Deploy via ansible. Distributed ACLs can be applied via Kubernetes policies. IPAM support. Load-balancing can be achieved without kube-proxy. NATing is done without using iptables/netsh.
NAT (*not used in Kubernetes*)	Containers are given a vNIC connected to an internal vSwitch. DNS/DHCP is provided using an internal component called WinNAT	MAC and IP is rewritten to host MAC/IP.	nat	Included here for completeness

As outlined above, the Flannel CNI meta plugin is also supported on Windows via the VXLAN network backend (**alpha support** ; delegates to win-overlay) and host-gateway network backend (stable support; delegates to win-bridge).

This plugin supports delegating to one of the reference CNI plugins (win-overlay, win-bridge), to work in conjunction with Flannel daemon on Windows (Flanneld) for automatic node subnet lease assignment and HNS network creation. This plugin reads in its own configuration file (cni.conf), and aggregates it with the environment variables from the FlannelD generated subnet.env file. It then delegates to one of the reference CNI plugins for network plumbing, and sends the correct configuration containing the node-assigned subnet to the IPAM plugin (for example: `host-local`).

For Node, Pod, and Service objects, the following network flows are supported for TCP/UDP traffic:

- Pod → Pod (IP)
- Pod → Pod (Name)
- Pod → Service (Cluster IP)
- Pod → Service (PQDN, but only if there are no ".")
- Pod → Service (FQDN)
- Pod → external (IP)
- Pod → external (DNS)
- Node → Pod
- Pod → Node

CNI plugin limitations

- Windows reference network plugins win-bridge and win-overlay do not implement CNI spec v0.4.0, due to a missing `CHECK` implementation.
- The Flannel VXLAN CNI plugin has the following limitations on Windows:

1. Node-pod connectivity isn't possible by design. It's only possible for local pods with Flannel v0.12.0 (or higher).
2. Flannel is restricted to using VNI 4096 and UDP port 4789. See the official Flannel VXLAN backend docs for more details on these parameters.

IP address management (IPAM)

The following IPAM options are supported on Windows:

- host-local
- HNS IPAM (Inbox platform IPAM, this is a fallback when no IPAM is set)
- azure-vnet-ipam (for azure-cni only)

Load balancing and Services

A Kubernetes Service is an abstraction that defines a logical set of Pods and a means to access them over a network. In a cluster that includes Windows nodes, you can use the following types of Service:

- `NodePort`
- `ClusterIP`
- `LoadBalancer`
- `ExternalName`

Warning:
There are known issue with NodePort services on overlay networking, if the target destination node is running Windows Server 2022. To avoid the issue entirely, you can configure the service with `externalTrafficPolicy: Local`.

There are known issues with pod to pod connectivity on l2bridge network on Windows Server 2022 with KB5005619 or higher installed. To workaround the issue and restore pod-pod connectivity, you can disable the WinDSR feature in kube-proxy.

These issues require OS fixes. Please follow https://github.com/microsoft/Windows-Containers/issues/204 for updates.

Windows container networking differs in some important ways from Linux networking. The Microsoft documentation for Windows Container Networking provides additional details and background.

On Windows, you can use the following settings to configure Services and load balancing behavior:

Feature	Description	Supported Kubernetes version	Supported Windows OS build	How to enable

Feature	Description	Supported Kubernetes version	Supported Windows OS build	How to enable
Session affinity	Ensures that connections from a particular client are passed to the same Pod each time.	v1.20+	Windows Server vNext Insider Preview Build 19551 (or higher)	Set `service.spec.sessionAffinity` to "ClientIP"
Direct Server Return (DSR)	Load balancing mode where the IP address fixups and the LBNAT occurs at the container vSwitch port directly; service traffic arrives with the source IP set as the originating pod IP.	v1.20+	Windows Server 2019	Set the following flags in kube-proxy: `--feature-gates="WinDSR=true" --enable-dsr=true`
Preserve-Destination	Skips DNAT of service traffic, thereby preserving the virtual IP of the target service in packets reaching the backend Pod. Also disables node-node forwarding.	v1.20+	Windows Server, version 1903 (or higher)	Set `"preserve-destination": "true"` in service annotations and enable DSR in kube-proxy.
IPv4/IPv6 dual-stack networking	Native IPv4-to-IPv4 in parallel with IPv6-to-IPv6 communications to, from, and within a cluster	v1.19+	Windows Server, version 2019	See IPv4/IPv6 dual-stack
Client IP preservation	Ensures that source IP of incoming ingress traffic gets preserved. Also disables node-node forwarding.	v1.20+	Windows Server, version 2019	Set `service.spec.externalTrafficPolicy` to "Local" and enable DSR in kube-proxy

Session affinity

Setting the maximum session sticky time for Windows services using `service.spec.sessionAffinityConfig.clientIP.timeoutSeconds` is not supported.

DNS

- ClusterFirstWithHostNet is not supported for DNS. Windows treats all names with a `.` as a FQDN and skips FQDN resolution
- On Linux, you have a DNS suffix list, which is used when trying to resolve PQDNs. On Windows, you can only have 1 DNS suffix, which is the DNS suffix associated with that pod's namespace (mydns.svc.cluster.local for example). Windows can resolve FQDNs and services or names resolvable with just that suffix. For example, a pod spawned in the default namespace, will have the DNS suffix **default.svc.cluster.local**. Inside a Windows pod, you can resolve both **kubernetes.default.svc.cluster.local** and **kubernetes**, but not the in-betweens, like **kubernetes.default** or **kubernetes.default.svc**.
- On Windows, there are multiple DNS resolvers that can be used. As these come with slightly different behaviors, using the `Resolve-DNSName` utility for name query resolutions is recommended.

IPv6 networking

Kubernetes on Windows does not support single-stack "IPv6-only" networking. However, dual-stack IPv4/IPv6 networking for pods and nodes with single-family services is supported.

You can use IPv4/IPv6 dual-stack networking with `l2bridge` networks. See configure IPv4/IPv6 dual stack for more details.

> **Note:** Overlay (VXLAN) networks on Windows do not support dual-stack networking.

Persistent storage

Windows has a layered filesystem driver to mount container layers and create a copy filesystem based on NTFS. All file paths in the container are resolved only within the context of that container.

- With Docker, volume mounts can only target a directory in the container, and not an individual file. This limitation does not exist with CRI-containerD runtime.
- Volume mounts cannot project files or directories back to the host filesystem.
- Read-only filesystems are not supported because write access is always required for the Windows registry and SAM database. However, read-only volumes are supported.
- Volume user-masks and permissions are not available. Because the SAM is not shared between the host & container, there's no mapping between them. All permissions are resolved within the context of the container.

As a result, the following storage functionality is not supported on Windows nodes:

- Volume subpath mounts: only the entire volume can be mounted in a Windows container
- Subpath volume mounting for Secrets
- Host mount projection
- Read-only root filesystem (mapped volumes still support `readOnly`)
- Block device mapping
- Memory as the storage medium (for example, `emptyDir.medium` set to `Memory`)
- File system features like uid/gid; per-user Linux filesystem permissions
- DefaultMode (due to UID/GID dependency)
- NFS based storage/volume support
- Expanding the mounted volume (resizefs)

Kubernetes volumes enable complex applications, with data persistence and Pod volume sharing requirements, to be deployed on Kubernetes. Management of persistent volumes associated with a specific storage back-end or protocol includes actions such as provisioning/de-provisioning/resizing of volumes, attaching/detaching a volume to/from a Kubernetes node and mounting/dismounting a volume to/from individual containers in a pod that needs to persist data.

The code implementing these volume management actions for a specific storage back-end or protocol is shipped in the form of a Kubernetes volume plugin. The following broad classes of Kubernetes volume plugins are supported on Windows:

In-tree volume plugins

Code associated with in-tree volume plugins ship as part of the core Kubernetes code base. Deployment of in-tree volume plugins do not require installation of additional scripts or deployment of separate containerized plugin components. These plugins can handle provisioning/de-provisioning and resizing of volumes in the storage backend, attaching/detaching of volumes to/from a Kubernetes node and mounting/dismounting a volume to/from individual containers in a pod. The following in-tree plugins support persistent storage on Windows nodes:

- `awsElasticBlockStore`
- `azureDisk`
- `azureFile`
- `gcePersistentDisk`
- `vsphereVolume`

FlexVolume plugins

Code associated with FlexVolume plugins ship as out-of-tree scripts or binaries that need to be deployed directly on the host. FlexVolume plugins handle attaching/detaching of volumes to/from a Kubernetes node and mounting/dismounting a volume to/from individual containers in a pod. Provisioning/De-provisioning of persistent volumes associated with FlexVolume plugins may be handled through an external provisioner that is typically separate from the FlexVolume plugins. The following FlexVolume plugins, deployed as PowerShell scripts on the host, support Windows nodes:

- SMB
- iSCSI

CSI plugins

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

Code associated with CSI plugins ship as out-of-tree scripts and binaries that are typically distributed as container images and deployed using standard Kubernetes constructs like DaemonSets and StatefulSets. CSI plugins handle a wide range of volume management actions in Kubernetes: provisioning/de-provisioning/resizing of volumes, attaching/detaching of volumes to/from a Kubernetes node and mounting/dismounting a volume to/from individual containers in a pod, backup/restore of persistent data using snapshots and cloning. CSI plugins typically consist of node plugins (that run on each node as a DaemonSet) and controller plugins.

CSI node plugins (especially those associated with persistent volumes exposed as either block devices or over a shared file-system) need to perform various privileged operations like scanning of disk devices, mounting of file systems, etc. These operations differ for each host operating system. For Linux worker nodes, containerized CSI node plugins are typically deployed as privileged containers. For Windows worker nodes, privileged operations for containerized CSI node plugins is supported using csi-proxy, a community-managed, stand-alone binary that needs to be pre-installed on each Windows node.

For more details, refer to the deployment guide of the CSI plugin you wish to deploy.

Command line options for the kubelet

The behavior of some kubelet command line options behave differently on Windows, as described below:

- The `--windows-priorityclass` lets you set the scheduling priority of the kubelet process (see CPU resource management)
- The `--kubelet-reserve`, `--system-reserve` , and `--eviction-hard` flags update NodeAllocatable
- Eviction by using `--enforce-node-allocable` is not implemented
- Eviction by using `--eviction-hard` and `--eviction-soft` are not implemented
- A kubelet running on a Windows node does not have memory restrictions. `--kubelet-reserve` and `--system-reserve` do not set limits on kubelet or processes running on the host. This means kubelet or a process on the host could cause memory resource starvation outside the node-allocatable and scheduler.
- The `MemoryPressure` Condition is not implemented
- The kubelet does not take OOM eviction actions

API compatibility

There are no differences in how most of the Kubernetes APIs work for Windows. The subtleties around what's different come down to differences in the OS and container runtime. In certain situations, some properties on workload resources were designed under the assumption that they would be implemented on Linux, and fail to run on Windows.

At a high level, these OS concepts are different:

- Identity - Linux uses userID (UID) and groupID (GID) which are represented as integer types. User and group names are not canonical - they are just an alias in `/etc/groups` or `/etc/passwd` back to UID+GID. Windows uses a larger binary security identifier (SID) which is stored in the Windows Security Access Manager (SAM) database. This database is not shared between the host and containers, or between containers.
- File permissions - Windows uses an access control list based on (SIDs), whereas POSIX systems such as Linux use a bitmask based on object permissions and UID+GID, plus *optional* access control lists.
- File paths - the convention on Windows is to use `\` instead of `/`. The Go IO libraries typically accept both and just make it work, but when you're setting a path or command line that's interpreted inside a container, `\` may be needed.
- Signals - Windows interactive apps handle termination differently, and can implement one or more of these:
  - A UI thread handles well-defined messages including `WM_CLOSE`.
  - Console apps handle Ctrl-C or Ctrl-break using a Control Handler.
  - Services register a Service Control Handler function that can accept `SERVICE_CONTROL_STOP` control codes.

Container exit codes follow the same convention where 0 is success, and nonzero is failure. The specific error codes may differ across Windows and Linux. However, exit codes passed from the Kubernetes components (kubelet, kube-proxy) are unchanged.

Field compatibility for container specifications

The following list documents differences between how Pod container specifications work between Windows and Linux:

- Huge pages are not implemented in the Windows container runtime, and are not available. They require asserting a user privilege that's not configurable for containers.
- `requests.cpu` and `requests.memory` - requests are subtracted from node available resources, so they can be used to avoid overprovisioning a node. However, they cannot be used to guarantee resources in an overprovisioned node. They should be applied to all containers as a best practice if the operator wants to avoid overprovisioning entirely.
- `securityContext.allowPrivilegeEscalation` - not possible on Windows; none of the capabilities are hooked up

- `securityContext.capabilities` - POSIX capabilities are not implemented on Windows
- `securityContext.privileged` - Windows doesn't support privileged containers
- `securityContext.procMount` - Windows doesn't have a `/proc` filesystem
- `securityContext.readOnlyRootFilesystem` - not possible on Windows; write access is required for registry & system processes to run inside the container
- `securityContext.runAsGroup` - not possible on Windows as there is no GID support
- `securityContext.runAsNonRoot` - this setting will prevent containers from running as `ContainerAdministrator` which is the closest equivalent to a root user on Windows.
- `securityContext.runAsUser` - use `runAsUserName` instead
- `securityContext.seLinuxOptions` - not possible on Windows as SELinux is Linux-specific
- `terminationMessagePath` - this has some limitations in that Windows doesn't support mapping single files. The default value is `/dev/termination-log`, which does work because it does not exist on Windows by default.

Field compatibility for Pod specifications

The following list documents differences between how Pod specifications work between Windows and Linux:

- `hostIPC` and `hostpid` - host namespace sharing is not possible on Windows
- `hostNetwork` - There is no Windows OS support to share the host network
- `dnsPolicy` - setting the Pod `dnsPolicy` to `ClusterFirstWithHostNet` is not supported on Windows because host networking is not provided. Pods always run with a container network.
- `podSecurityContext` (see below)
- `shareProcessNamespace` - this is a beta feature, and depends on Linux namespaces which are not implemented on Windows. Windows cannot share process namespaces or the container's root filesystem. Only the network can be shared.
- `terminationGracePeriodSeconds` - this is not fully implemented in Docker on Windows, see the GitHub issue. The behavior today is that the ENTRYPOINT process is sent CTRL_SHUTDOWN_EVENT, then Windows waits 5 seconds by default, and finally shuts down all processes using the normal Windows shutdown behavior. The 5 second default is actually in the Windows registry inside the container, so it can be overridden when the container is built.
- `volumeDevices` - this is a beta feature, and is not implemented on Windows. Windows cannot attach raw block devices to pods.
- `volumes`
  - If you define an `emptyDir` volume, you cannot set its volume source to `memory`.
- You cannot enable `mountPropagation` for volume mounts as this is not supported on Windows.

Field compatibility for Pod security context

None of the Pod `securityContext` fields work on Windows.

Node problem detector

The node problem detector (see Monitor Node Health) is not compatible with Windows.

Pause container

In a Kubernetes Pod, an infrastructure or “pause” container is first created to host the container. In Linux, the cgroups and namespaces that make up a pod need a process to maintain their continued existence; the pause process provides this. Containers that belong to the same pod, including infrastructure and worker containers, share a common network endpoint (same IPv4 and / or IPv6 address, same network port spaces). Kubernetes uses pause containers to allow for worker containers crashing or restarting without losing any of the networking configuration.

Kubernetes maintains a multi-architecture image that includes support for Windows. For Kubernetes v1.23 the recommended pause image is `k8s.gcr.io/pause:3.6`. The source code is available on GitHub.

Microsoft maintains a different multi-architecture image, with Linux and Windows amd64 support, that you can find as `mcr.microsoft.com/oss/kubernetes/pause:3.6`. This image is built from the same source as the Kubernetes maintained image but all of the Windows binaries are authenticode signed by Microsoft. The Kubernetes project recommends using the Microsoft maintained image if you are deploying to a production or production-like environment that requires signed binaries.

Container runtimes

You need to install a container runtime into each node in the cluster so that Pods can run there.

The following container runtimes work with Windows:

> **Note:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

cri-containerd

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

You can use ContainerD 1.4.0+ as the container runtime for Kubernetes nodes that run Windows.

Learn how to install ContainerD on a Windows node.

> **Note:** There is a known limitation when using GMSA with containerd to access Windows network shares, which requires a kernel patch.

Mirantis Container Runtime

Mirantis Container Runtime (MCR) is available as a container runtime for all Windows Server 2019 and later versions.

See Install MCR on Windows Servers for more information.

Windows OS version compatibility

On Windows nodes, strict compatibility rules apply where the host OS version must match the container base image OS version. Only Windows containers with a container operating system of Windows Server 2019 are fully supported.

For Kubernetes v1.23, operating system compatibility for Windows nodes (and Pods) is as follows:

Windows Server LTSC release

Windows Server 2019

Windows Server 2022

Windows Server SAC release

Windows Server version 20H2

The Kubernetes version-skew policy also applies.

Security for Windows nodes

On Windows, data from Secrets are written out in clear text onto the node's local storage (as compared to using tmpfs / in-memory filesystems on Linux). As a cluster operator, you should take both of the following additional measures:

1. Use file ACLs to secure the Secrets' file location.
2. Apply volume-level encryption using BitLocker.

RunAsUsername can be specified for Windows Pods or containers to execute the container processes as a node-default user. This is roughly equivalent to RunAsUser.

Linux-specific pod security context privileges such as SELinux, AppArmor, Seccomp, or capabilities (POSIX capabilities), and others are not supported.

Privileged containers are not supported on Windows.

Getting help and troubleshooting

Your main source of help for troubleshooting your Kubernetes cluster should start with the Troubleshooting page.

Some additional, Windows-specific troubleshooting help is included in this section. Logs are an important element of troubleshooting issues in Kubernetes. Make sure to include them any time you seek troubleshooting assistance from other contributors. Follow the instructions in the SIG Windows contributing guide on gathering logs.

Node-level troubleshooting

1. How do I know `start.ps1` completed successfully?

   You should see kubelet, kube-proxy, and (if you chose Flannel as your networking solution) flanneld host-agent processes running on your node, with running logs being displayed in separate PowerShell windows. In addition to this, your Windows node should be listed as "Ready" in your Kubernetes cluster.

2. Can I configure the Kubernetes node processes to run in the background as services?

   The kubelet and kube-proxy are already configured to run as native Windows Services, offering resiliency by re-starting the services automatically in the event of failure (for example a process crash). You have two options for configuring these node components as services.

   1. As native Windows Services

      You can run the kubelet and kube-proxy as native Windows Services using `sc.exe`.

```
# Create the services for kubelet and kube-proxy in two separate commands
sc.exe create <component_name> binPath= "<path_to_binary> --service <other_arg

# Please note that if the arguments contain spaces, they must be escaped.
sc.exe create kubelet binPath= "C:\kubelet.exe --service --hostname-override '

# Start the services
Start-Service kubelet
Start-Service kube-proxy

# Stop the service
Stop-Service kubelet (-Force)
Stop-Service kube-proxy (-Force)

# Query the service status
Get-Service kubelet
Get-Service kube-proxy
```

2. Using `nssm.exe`

   You can also always use alternative service managers like nssm.exe to run these processes (flanneld, kubelet & kube-proxy) in the background for you. You can use this sample script, leveraging nssm.exe to register kubelet, kube-proxy, and flanneld.exe to run as Windows services in the background.

```
register-svc.ps1 -NetworkMode <Network mode> -ManagementIP <Windows Node IP> -

# NetworkMode      = The network mode l2bridge (flannel host-gw, also the defa
# ManagementIP     = The IP address assigned to the Windows node. You can use
# ClusterCIDR      = The cluster subnet range. (Default value 10.244.0.0/16)
# KubeDnsServiceIP = The Kubernetes DNS service IP (Default value 10.96.0.10)
# LogDir           = The directory where kubelet and kube-proxy logs are redir
```

   If the above referenced script is not suitable, you can manually configure `nssm.exe` using the following examples.

```
# Register flanneld.exe
nssm install flanneld C:\flannel\flanneld.exe
nssm set flanneld AppParameters --kubeconfig-file=c:\k\config --iface=<Managem
nssm set flanneld AppEnvironmentExtra NODE_NAME=<hostname>
nssm set flanneld AppDirectory C:\flannel
nssm start flanneld

# Register kubelet.exe
# Microsoft releases the pause infrastructure container at mcr.microsoft.com/o
nssm install kubelet C:\k\kubelet.exe
nssm set kubelet AppParameters --hostname-override=<hostname> --v=6 --pod-infr
nssm set kubelet AppDirectory C:\k
nssm start kubelet

# Register kube-proxy.exe (l2bridge / host-gw)
nssm install kube-proxy C:\k\kube-proxy.exe
nssm set kube-proxy AppDirectory c:\k
nssm set kube-proxy AppParameters --v=4 --proxy-mode=kernelspace --hostname-ov
nssm.exe set kube-proxy AppEnvironmentExtra KUBE_NETWORK=cbr0
nssm set kube-proxy DependOnService kubelet
nssm start kube-proxy

# Register kube-proxy.exe (overlay / vxlan)
nssm install kube-proxy C:\k\kube-proxy.exe
nssm set kube-proxy AppDirectory c:\k
nssm set kube-proxy AppParameters --v=4 --proxy-mode=kernelspace --feature-gat
nssm set kube-proxy DependOnService kubelet
nssm start kube-proxy
```

   For initial troubleshooting, you can use the following flags in nssm.exe to redirect stdout and stderr to a output file:

```
nssm set <Service Name> AppStdout C:\k\mysvc.log
nssm set <Service Name> AppStderr C:\k\mysvc.log
```

   For additional details, see NSSM - the Non-Sucking Service Manager.

3. My Pods are stuck at "Container Creating" or restarting over and over

Check that your pause image is compatible with your OS version. The instructions assume that both the OS and the containers are version 1803. If you have a later version of Windows, such as an Insider build, you need to adjust the images accordingly. See Pause container for more details.

Network troubleshooting

1. My Windows Pods do not have network connectivity

   If you are using virtual machines, ensure that MAC spoofing is **enabled** on all the VM network adapter(s).

2. My Windows Pods cannot ping external resources

   Windows Pods do not have outbound rules programmed for the ICMP protocol. However, TCP/UDP is supported. When trying to demonstrate connectivity to resources outside of the cluster, substitute `ping <IP>` with corresponding `curl <IP>` commands.

   If you are still facing problems, most likely your network configuration in cni.conf deserves some extra attention. You can always edit this static file. The configuration update will apply to any new Kubernetes resources.

   One of the Kubernetes networking requirements (see Kubernetes model) is for cluster communication to occur without NAT internally. To honor this requirement, there is an ExceptionList for all the communication where you do not want outbound NAT to occur. However, this also means that you need to exclude the external IP you are trying to query from the `ExceptionList` . Only then will the traffic originating from your Windows pods be SNAT'ed correctly to receive a response from the outside world. In this regard, your `ExceptionList` in `cni.conf` should look as follows:

   ```
   "ExceptionList": [
                   "10.244.0.0/16",  # Cluster subnet
                   "10.96.0.0/12",   # Service subnet
                   "10.127.130.0/24" # Management (host) subnet
               ]
   ```

3. My Windows node cannot access `NodePort` type Services

   Local NodePort access from the node itself fails. This is a known limitation. NodePort access works from other nodes or external clients.

4. vNICs and HNS endpoints of containers are being deleted

   This issue can be caused when the `hostname-override` parameter is not passed to kube-proxy. To resolve it, users need to pass the hostname to kube-proxy as follows:

   ```
   C:\k\kube-proxy.exe --hostname-override=$(hostname)
   ```

5. With flannel, my nodes are having issues after rejoining a cluster

   Whenever a previously deleted node is being re-joined to the cluster, flannelD tries to assign a new pod subnet to the node. Users should remove the old pod subnet configuration files in the following paths:

   ```
   Remove-Item C:\k\SourceVip.json
   Remove-Item C:\k\SourceVipRequest.json
   ```

6. After launching `start.ps1` , flanneld is stuck in "Waiting for the Network to be created"

   There are numerous reports of this issue; most likely it is a timing issue for when the management IP of the flannel network is set. A workaround is to relaunch `start.ps1` or relaunch it manually as follows:

```
[Environment]::SetEnvironmentVariable("NODE_NAME", "<Windows_Worker_Hostname>")
C:\flannel\flanneld.exe --kubeconfig-file=c:\k\config --iface=<Windows_Worker_Node_
```

7. My Windows Pods cannot launch because of missing `/run/flannel/subnet.env`

   This indicates that Flannel didn't launch correctly. You can either try to restart `flanneld.exe` or you can copy the files over manually from `/run/flannel/subnet.env` on the Kubernetes master to `C:\run\flannel\subnet.env` on the Windows worker node and modify the `FLANNEL_SUBNET` row to a different number. For example, if node subnet 10.244.4.1/24 is desired:

```
FLANNEL_NETWORK=10.244.0.0/16
FLANNEL_SUBNET=10.244.4.1/24
FLANNEL_MTU=1500
FLANNEL_IPMASQ=true
```

8. My Windows node cannot access my services using the service IP

   This is a known limitation of the networking stack on Windows. However, Windows Pods can access the Service IP.

9. No network adapter is found when starting the kubelet

   The Windows networking stack needs a virtual adapter for Kubernetes networking to work. If the following commands return no results (in an admin shell), virtual network creation — a necessary prerequisite for the kubelet to work — has failed:

```
Get-HnsNetwork | ? Name -ieq "cbr0"
Get-NetAdapter | ? Name -Like "vEthernet (Ethernet*"
```

   Often it is worthwhile to modify the InterfaceName parameter of the start.ps1 script, in cases where the host's network adapter isn't "Ethernet". Otherwise, consult the output of the `start-kubelet.ps1` script to see if there are errors during virtual network creation.

10. DNS resolution is not properly working

    Check the DNS limitations for Windows in this section.

11. `kubectl port-forward` fails with "unable to do port forwarding: wincat not found"

    This was implemented in Kubernetes 1.15 by including `wincat.exe` in the pause infrastructure container `mcr.microsoft.com/oss/kubernetes/pause:3.6`. Be sure to use a supported version of Kubernetes. If you would like to build your own pause infrastructure container be sure to include wincat.

12. My Kubernetes installation is failing because my Windows Server node is behind a proxy

    If you are behind a proxy, the following PowerShell environment variables must be defined:

```
[Environment]::SetEnvironmentVariable("HTTP_PROXY", "http://proxy.example.com:80/",
[Environment]::SetEnvironmentVariable("HTTPS_PROXY", "http://proxy.example.com:443/
```

Further investigation

If these steps don't resolve your problem, you can get help running Windows containers on Windows nodes in Kubernetes through:

- StackOverflow Windows Server Container topic
- Kubernetes Official Forum discuss.kubernetes.io
- Kubernetes Slack #SIG-Windows Channel

Reporting issues and feature requests

If you have what looks like a bug, or you would like to make a feature request, please use the GitHub issue tracking system. You can open issues on GitHub and assign them to SIG-Windows. You should first search the list of issues in case it was reported previously and comment with your experience on the issue and add additional logs. SIG-Windows Slack is also a great avenue to get some initial support and troubleshooting ideas prior to creating a ticket.

If filing a bug, please include detailed information about how to reproduce the problem, such as:

- Kubernetes version: output from `kubectl version`
- Environment details: Cloud provider, OS distro, networking choice and configuration, and Docker version
- Detailed steps to reproduce the problem
- Relevant logs

It helps if you tag the issue as **sig/windows**, by commenting on the issue with `/sig windows`. This helps to bring the issue to a SIG Windows member's attention

What's next

Deployment tools

The kubeadm tool helps you to deploy a Kubernetes cluster, providing the control plane to manage the cluster it, and nodes to run your workloads. Adding Windows nodes explains how to deploy Windows nodes to your cluster using kubeadm.

The Kubernetes cluster API project also provides means to automate deployment of Windows nodes.

Windows distribution channels

For a detailed explanation of Windows distribution channels see the Microsoft documentation.

Information on the different Windows Server servicing channels including their support models can be found at Windows Server servicing channels.

2.4.2 - Guide for scheduling Windows containers in Kubernetes

Windows applications constitute a large portion of the services and applications that run in many organizations. This guide walks you through the steps to configure and deploy a Windows container in Kubernetes.

Objectives

- Configure an example deployment to run Windows containers on the Windows node
- (Optional) Configure an Active Directory Identity for your Pod using Group Managed Service Accounts (GMSA)

Before you begin

- Create a Kubernetes cluster that includes a control plane and a worker node running Windows Server
- It is important to note that creating and deploying services and workloads on Kubernetes behaves in much the same way for Linux and Windows containers. Kubectl commands to interface with the cluster are identical. The example in the section below is provided to jumpstart your experience with Windows containers.

Getting Started: Deploying a Windows container

To deploy a Windows container on Kubernetes, you must first create an example application. The example YAML file below creates a simple webserver application. Create a service spec named `win-webserver.yaml` with the contents below:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: win-webserver
  labels:
    app: win-webserver
spec:
  ports:
    # the port that this service should serve on
    - port: 80
      targetPort: 80
  selector:
    app: win-webserver
  type: NodePort
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: win-webserver
  name: win-webserver
spec:
  replicas: 2
  selector:
    matchLabels:
      app: win-webserver
  template:
    metadata:
      labels:
        app: win-webserver
```

```
    name: win-webserver
  spec:
   containers:
    - name: windowswebserver
      image: mcr.microsoft.com/windows/servercore:ltsc2019
      command:
      - powershell.exe
      - -command
      - "<#code used from https://gist.github.com/19WAS85/5424431#> ; $$listener = New
   nodeSelector:
    kubernetes.io/os: windows
```

Note: Port mapping is also supported, but for simplicity in this example the container port 80 is exposed directly to the service.

1. Check that all nodes are healthy:

```
kubectl get nodes
```

2. Deploy the service and watch for pod updates:

```
kubectl apply -f win-webserver.yaml
kubectl get pods -o wide -w
```

When the service is deployed correctly both Pods are marked as Ready. To exit the watch command, press Ctrl+C.

3. Check that the deployment succeeded. To verify:

   - Two containers per pod on the Windows node, use `docker ps`
   - Two pods listed from the Linux control plane node, use `kubectl get pods`
   - Node-to-pod communication across the network, `curl` port 80 of your pod IPs from the Linux control plane node to check for a web server response
   - Pod-to-pod communication, ping between pods (and across hosts, if you have more than one Windows node) using docker exec or kubectl exec
   - Service-to-pod communication, `curl` the virtual service IP (seen under `kubectl get services`) from the Linux control plane node and from individual pods
   - Service discovery, `curl` the service name with the Kubernetes default DNS suffix
   - Inbound connectivity, `curl` the NodePort from the Linux control plane node or machines outside of the cluster
   - Outbound connectivity, `curl` external IPs from inside the pod using kubectl exec

Note: Windows container hosts are not able to access the IP of services scheduled on them due to current platform limitations of the Windows networking stack. Only Windows pods are able to access service IPs.

Observability

Capturing logs from workloads

Logs are an important element of observability; they enable users to gain insights into the operational aspect of workloads and are a key ingredient to troubleshooting issues. Because Windows containers and workloads inside Windows containers behave differently from Linux containers, users had a hard time collecting logs, limiting operational visibility. Windows workloads for example are usually configured to log to ETW (Event Tracing for Windows) or

push entries to the application event log. LogMonitor, an open source tool by Microsoft, is the recommended way to monitor configured log sources inside a Windows container. LogMonitor supports monitoring event logs, ETW providers, and custom application logs, piping them to STDOUT for consumption by `kubectl logs <pod>`.

Follow the instructions in the LogMonitor GitHub page to copy its binaries and configuration files to all your containers and add the necessary entrypoints for LogMonitor to push your logs to STDOUT.

Using configurable Container usernames

Starting with Kubernetes v1.16, Windows containers can be configured to run their entrypoints and processes with different usernames than the image defaults. The way this is achieved is a bit different from the way it is done for Linux containers. Learn more about it here.

Managing Workload Identity with Group Managed Service Accounts

Starting with Kubernetes v1.14, Windows container workloads can be configured to use Group Managed Service Accounts (GMSA). Group Managed Service Accounts are a specific type of Active Directory account that provides automatic password management, simplified service principal name (SPN) management, and the ability to delegate the management to other administrators across multiple servers. Containers configured with a GMSA can access external Active Directory Domain resources while carrying the identity configured with the GMSA. Learn more about configuring and using GMSA for Windows containers here.

Taints and Tolerations

Users today need to use some combination of taints and node selectors in order to keep Linux and Windows workloads on their respective OS-specific nodes. This likely imposes a burden only on Windows users. The recommended approach is outlined below, with one of its main goals being that this approach should not break compatibility for existing Linux workloads.

> **Note:**
> If the `IdentifyPodOS` feature gate is enabled, you can (and should) set `.spec.os.name` for a Pod to indicate the operating system that the containers in that Pod are designed for. For Pods that run Linux containers, set `.spec.os.name` to `linux`. For Pods that run Windows containers, set `.spec.os.name` to Windows.
>
> The scheduler does not use the value of `.spec.os.name` when assigning Pods to nodes. You should use normal Kubernetes mechanisms for assigning pods to nodes to ensure that the control plane for your cluster places pods onto nodes that are running the appropriate operating system. no effect on the scheduling of the Windows pods, so taints and tolerations and node selectors are still required to ensure that the Windows pods land onto appropriate Windows nodes.

Ensuring OS-specific workloads land on the appropriate container host

Users can ensure Windows containers can be scheduled on the appropriate host using Taints and Tolerations. All Kubernetes nodes today have the following default labels:

- kubernetes.io/os = [windows|linux]
- kubernetes.io/arch = [amd64|arm64|...]

If a Pod specification does not specify a nodeSelector like `"kubernetes.io/os": windows`, it is possible the Pod can be scheduled on any host, Windows or Linux. This can be problematic since a Windows container can only run on Windows and a Linux container can only run on Linux. The best practice is to use a nodeSelector.

However, we understand that in many cases users have a pre-existing large number of deployments for Linux containers, as well as an ecosystem of off-the-shelf configurations, such as community Helm charts, and programmatic Pod generation cases, such as with Operators. In those situations, you may be hesitant to make the configuration change to add nodeSelectors. The alternative is to use Taints. Because the kubelet can set Taints during registration, it could easily be modified to automatically add a taint when running on Windows only.

For example: `--register-with-taints='os=windows:NoSchedule'`

By adding a taint to all Windows nodes, nothing will be scheduled on them (that includes existing Linux Pods). In order for a Windows Pod to be scheduled on a Windows node, it would need both the nodeSelector and the appropriate matching toleration to choose Windows.

```
nodeSelector:
    kubernetes.io/os: windows
    node.kubernetes.io/windows-build: '10.0.17763'
tolerations:
    - key: "os"
      operator: "Equal"
      value: "windows"
      effect: "NoSchedule"
```

Handling multiple Windows versions in the same cluster

The Windows Server version used by each pod must match that of the node. If you want to use multiple Windows Server versions in the same cluster, then you should set additional node labels and nodeSelectors.

Kubernetes 1.17 automatically adds a new label `node.kubernetes.io/windows-build` to simplify this. If you're running an older version, then it's recommended to add this label manually to Windows nodes.

This label reflects the Windows major, minor, and build number that need to match for compatibility. Here are values used today for each Windows Server version.

Product Name	Build Number(s)
Windows Server 2019	10.0.17763
Windows Server version 1809	10.0.17763
Windows Server version 1903	10.0.18362

Simplifying with RuntimeClass

RuntimeClass can be used to simplify the process of using taints and tolerations. A cluster administrator can create a `RuntimeClass` object which is used to encapsulate these taints and tolerations.

1. Save this file to `runtimeClasses.yml`. It includes the appropriate `nodeSelector` for the Windows OS, architecture, and version.

```
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
```

```
    name: windows-2019
  handler: 'docker'
  scheduling:
    nodeSelector:
      kubernetes.io/os: 'windows'
      kubernetes.io/arch: 'amd64'
      node.kubernetes.io/windows-build: '10.0.17763'
    tolerations:
    - effect: NoSchedule
      key: os
      operator: Equal
      value: "windows"
```

1. Run `kubectl create -f runtimeClasses.yml` using as a cluster administrator
2. Add `runtimeClassName: windows-2019` as appropriate to Pod specs

For example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iis-2019
  labels:
    app: iis-2019
spec:
  replicas: 1
  template:
    metadata:
      name: iis-2019
      labels:
        app: iis-2019
    spec:
      runtimeClassName: windows-2019
      containers:
      - name: iis
        image: mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2019
        resources:
          limits:
            cpu: 1
            memory: 800Mi
          requests:
            cpu: .1
            memory: 300Mi
        ports:
          - containerPort: 80
 selector:
    matchLabels:
      app: iis-2019
---
apiVersion: v1
kind: Service
metadata:
  name: iis
spec:
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 80
  selector:
    app: iis-2019
```

3 - Best practices

3.1 - Considerations for large clusters

A cluster is a set of nodes (physical or virtual machines) running Kubernetes agents, managed by the control plane. Kubernetes v1.23 supports clusters with up to 5000 nodes. More specifically, Kubernetes is designed to accommodate configurations that meet *all* of the following criteria:

- No more than 110 pods per node
- No more than 5000 nodes
- No more than 150000 total pods
- No more than 300000 total containers

You can scale your cluster by adding or removing nodes. The way you do this depends on how your cluster is deployed.

Cloud provider resource quotas

To avoid running into cloud provider quota issues, when creating a cluster with many nodes, consider:

- Requesting a quota increase for cloud resources such as:
  - Computer instances
  - CPUs
  - Storage volumes
  - In-use IP addresses
  - Packet filtering rule sets
  - Number of load balancers
  - Network subnets
  - Log streams
- Gating the cluster scaling actions to bring up new nodes in batches, with a pause between batches, because some cloud providers rate limit the creation of new instances.

Control plane components

For a large cluster, you need a control plane with sufficient compute and other resources.

Typically you would run one or two control plane instances per failure zone, scaling those instances vertically first and then scaling horizontally after reaching the point of falling returns to (vertical) scale.

You should run at least one instance per failure zone to provide fault-tolerance. Kubernetes nodes do not automatically steer traffic towards control-plane endpoints that are in the same failure zone; however, your cloud provider might have its own mechanisms to do this.

For example, using a managed load balancer, you configure the load balancer to send traffic that originates from the kubelet and Pods in failure zone *A*, and direct that traffic only to the control plane hosts that are also in zone *A*. If a single control-plane host or endpoint failure zone *A* goes offline, that means that all the control-plane traffic for nodes in zone *A* is now being sent between zones. Running multiple control plane hosts in each zone makes that outcome less likely.

etcd storage

To improve performance of large clusters, you can store Event objects in a separate dedicated etcd instance.

When creating a cluster, you can (using custom tooling):

- start and configure additional etcd instance
- configure the API server to use it for storing events

See Operating etcd clusters for Kubernetes and Set up a High Availability etcd cluster with kubeadm for details on configuring and managing etcd for a large cluster.

Addon resources

Kubernetes resource limits help to minimize the impact of memory leaks and other ways that pods and containers can impact on other components. These resource limits apply to addon resources just as they apply to application workloads.

For example, you can set CPU and memory limits for a logging component:

```
...
containers:
- name: fluentd-cloud-logging
  image: fluent/fluentd-kubernetes-daemonset:v1
  resources:
    limits:
      cpu: 100m
      memory: 200Mi
```

Addons' default limits are typically based on data collected from experience running each addon on small or medium Kubernetes clusters. When running on large clusters, addons often consume more of some resources than their default limits. If a large cluster is deployed without adjusting these values, the addon(s) may continuously get killed because they keep hitting the memory limit. Alternatively, the addon may run but with poor performance due to CPU time slice restrictions.

To avoid running into cluster addon resource issues, when creating a cluster with many nodes, consider the following:

- Some addons scale vertically - there is one replica of the addon for the cluster or serving a whole failure zone. For these addons, increase requests and limits as you scale out your cluster.
- Many addons scale horizontally - you add capacity by running more pods - but with a very large cluster you may also need to raise CPU or memory limits slightly. The VerticalPodAutoscaler can run in *recommender* mode to provide suggested figures for requests and limits.
- Some addons run as one copy per node, controlled by a DaemonSet: for example, a node-level log aggregator. Similar to the case with horizontally-scaled addons, you may also need to raise CPU or memory limits slightly.

What's next

`VerticalPodAutoscaler` is a custom resource that you can deploy into your cluster to help you manage resource requests and limits for pods.
Visit Vertical Pod Autoscaler to learn more about `VerticalPodAutoscaler` and how you can use it to scale cluster components, including cluster-critical addons.

The cluster autoscaler integrates with a number of cloud providers to help you run the right number of nodes for the level of resource demand in your cluster.

The addon resizer helps you in resizing the addons automatically as your cluster's scale changes.

3.2 - Running in multiple zones

This page describes running Kubernetes across multiple zones.

Background

Kubernetes is designed so that a single Kubernetes cluster can run across multiple failure zones, typically where these zones fit within a logical grouping called a *region*. Major cloud providers define a region as a set of failure zones (also called *availability zones*) that provide a consistent set of features: within a region, each zone offers the same APIs and services.

Typical cloud architectures aim to minimize the chance that a failure in one zone also impairs services in another zone.

Control plane behavior

All control plane components support running as a pool of interchangeable resources, replicated per component.

When you deploy a cluster control plane, place replicas of control plane components across multiple failure zones. If availability is an important concern, select at least three failure zones and replicate each individual control plane component (API server, scheduler, etcd, cluster controller manager) across at least three failure zones. If you are running a cloud controller manager then you should also replicate this across all the failure zones you selected.

> **Note:** Kubernetes does not provide cross-zone resilience for the API server endpoints. You can use various techniques to improve availability for the cluster API server, including DNS round-robin, SRV records, or a third-party load balancing solution with health checking.

Node behavior

Kubernetes automatically spreads the Pods for workload resources (such as Deployment or StatefulSet) across different nodes in a cluster. This spreading helps reduce the impact of failures.

When nodes start up, the kubelet on each node automatically adds labels to the Node object that represents that specific kubelet in the Kubernetes API. These labels can include zone information.

If your cluster spans multiple zones or regions, you can use node labels in conjunction with Pod topology spread constraints to control how Pods are spread across your cluster among fault domains: regions, zones, and even specific nodes. These hints enable the scheduler to place Pods for better expected availability, reducing the risk that a correlated failure affects your whole workload.

For example, you can set a constraint to make sure that the 3 replicas of a StatefulSet are all running in different zones to each other, whenever that is feasible. You can define this declaratively without explicitly defining which availability zones are in use for each workload.

Distributing nodes across zones

Kubernetes' core does not create nodes for you; you need to do that yourself, or use a tool such as the Cluster API to manage nodes on your behalf.

Using tools such as the Cluster API you can define sets of machines to run as worker nodes for your cluster across multiple failure domains, and rules to automatically heal the cluster in case of whole-zone service disruption.

Manual zone assignment for Pods

You can apply node selector constraints to Pods that you create, as well as to Pod templates in workload resources such as Deployment, StatefulSet, or Job.

Storage access for zones

When persistent volumes are created, the `PersistentVolumeLabel` admission controller automatically adds zone labels to any PersistentVolumes that are linked to a specific zone. The scheduler then ensures, through its `NoVolumeZoneConflict` predicate, that pods which claim a given PersistentVolume are only placed into the same zone as that volume.

You can specify a StorageClass for PersistentVolumeClaims that specifies the failure domains (zones) that the storage in that class may use. To learn about configuring a StorageClass that is aware of failure domains or zones, see Allowed topologies.

Networking

By itself, Kubernetes does not include zone-aware networking. You can use a network plugin to configure cluster networking, and that network solution might have zone-specific elements. For example, if your cloud provider supports Services with `type=LoadBalancer`, the load balancer might only send traffic to Pods running in the same zone as the load balancer element processing a given connection. Check your cloud provider's documentation for details.

For custom or on-premises deployments, similar considerations apply. Service and Ingress behavior, including handling of different failure zones, does vary depending on exactly how your cluster is set up.

Fault recovery

When you set up your cluster, you might also need to consider whether and how your setup can restore service if all the failure zones in a region go off-line at the same time. For example, do you rely on there being at least one node able to run Pods in a zone?
Make sure that any cluster-critical repair work does not rely on there being at least one healthy node in your cluster. For example: if all nodes are unhealthy, you might need to run a repair Job with a special toleration so that the repair can complete enough to bring at least one node into service.

Kubernetes doesn't come with an answer for this challenge; however, it's something to consider.

What's next

To learn how the scheduler places Pods in a cluster, honoring the configured constraints, visit Scheduling and Eviction.

3.3 - Validate node setup

Node Conformance Test

Node conformance test is a containerized test framework that provides a system verification and functionality test for a node. The test validates whether the node meets the minimum requirements for Kubernetes; a node that passes the test is qualified to join a Kubernetes cluster.

Node Prerequisite

To run node conformance test, a node must satisfy the same prerequisites as a standard Kubernetes node. At a minimum, the node should have the following daemons installed:

- Container Runtime (Docker)
- Kubelet

Running Node Conformance Test

To run the node conformance test, perform the following steps:

1. Work out the value of the `--kubeconfig` option for the kubelet; for example: `--kubeconfig=/var/lib/kubelet/config.yaml`. Because the test framework starts a local control plane to test the kubelet, use `http://localhost:8080` as the URL of the API server. There are some other kubelet command line parameters you may want to use:

- `--pod-cidr`: If you are using `kubenet`, you should specify an arbitrary CIDR to Kubelet, for example `--pod-cidr=10.180.0.0/24`.
- `--cloud-provider`: If you are using `--cloud-provider=gce`, you should remove the flag to run the test.

2. Run the node conformance test with command:

```
# $CONFIG_DIR is the pod manifest path of your Kubelet.
# $LOG_DIR is the test output path.
sudo docker run -it --rm --privileged --net=host \
  -v /:/rootfs -v $CONFIG_DIR:$CONFIG_DIR -v $LOG_DIR:/var/result \
  k8s.gcr.io/node-test:0.2
```

Running Node Conformance Test for Other Architectures

Kubernetes also provides node conformance test docker images for other architectures:

Arch	Image
amd64	node-test-amd64
arm	node-test-arm
arm64	node-test-arm64

Running Selected Test

To run specific tests, overwrite the environment variable `FOCUS` with the regular expression of tests you want to run.

```
sudo docker run -it --rm --privileged --net=host \
  -v /:/rootfs:ro -v $CONFIG_DIR:$CONFIG_DIR -v $LOG_DIR:/var/result \
  -e FOCUS=MirrorPod \ # Only run MirrorPod test
  k8s.gcr.io/node-test:0.2
```

To skip specific tests, overwrite the environment variable `SKIP` with the regular expression of tests you want to skip.

```
sudo docker run -it --rm --privileged --net=host \
  -v /:/rootfs:ro -v $CONFIG_DIR:$CONFIG_DIR -v $LOG_DIR:/var/result \
  -e SKIP=MirrorPod \ # Run all conformance tests but skip MirrorPod test
  k8s.gcr.io/node-test:0.2
```

Node conformance test is a containerized version of node e2e test. By default, it runs all conformance tests.

Theoretically, you can run any node e2e test if you configure the container and mount required volumes properly. But **it is strongly recommended to only run conformance test**, because it requires much more complex configuration to run non-conformance test.

Caveats

- The test leaves some docker images on the node, including the node conformance test image and images of containers used in the functionality test.
- The test leaves dead containers on the node. These containers are created during the functionality test.

3.4 - Enforcing Pod Security Standards

This page provides an overview of best practices when it comes to enforcing Pod Security Standards.

Using the built-in Pod Security Admission Controller

FEATURE STATE: `Kubernetes v1.23 [beta]`

The Pod Security Admission Controller intends to replace the deprecated PodSecurityPolicies.

Configure all cluster namespaces

Namespaces that lack any configuration at all should be considered significant gaps in your cluster security model. We recommend taking the time to analyze the types of workloads occurring in each namespace, and by referencing the Pod Security Standards, decide on an appropriate level for each of them. Unlabeled namespaces should only indicate that they've yet to be evaluated.

In the scenario that all workloads in all namespaces have the same security requirements, we provide an example that illustrates how the PodSecurity labels can be applied in bulk.

Embrace the principle of least privilege

In an ideal world, every pod in every namespace would meet the requirements of the `restricted` policy. However, this is not possible nor practical, as some workloads will require elevated privileges for legitimate reasons.

- Namespaces allowing `privileged` workloads should establish and enforce appropriate access controls.
- For workloads running in those permissive namespaces, maintain documentation about their unique security requirements. If at all possible, consider how those requirements could be further constrained.

Adopt a multi-mode strategy

The `audit` and `warn` modes of the Pod Security Standards admission controller make it easy to collect important security insights about your pods without breaking existing workloads.

It is good practice to enable these modes for all namespaces, setting them to the *desired* level and version you would eventually like to `enforce`. The warnings and audit annotations generated in this phase can guide you toward that state. If you expect workload authors to make changes to fit within the desired level, enable the `warn` mode. If you expect to use audit logs to monitor/drive changes to fit within the desired level, enable the `audit` mode.

When you have the `enforce` mode set to your desired value, these modes can still be useful in a few different ways:

- By setting `warn` to the same level as `enforce`, clients will receive warnings when attempting to create Pods (or resources that have Pod templates) that do not pass validation. This will help them update those resources to become compliant.
- In Namespaces that pin `enforce` to a specific non-latest version, setting the `audit` and `warn` modes to the same level as `enforce`, but to the `latest` version, gives visibility into settings that were allowed by previous versions but are not allowed per current best practices.

Third-party alternatives

> **Note:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

Other alternatives for enforcing security profiles are being developed in the Kubernetes ecosystem:

- Kubewarden.
- Kyverno.
- OPA Gatekeeper.

The decision to go with a *built-in* solution (e.g. PodSecurity admission controller) versus a third-party tool is entirely dependent on your own situation. When evaluating any solution, trust of your supply chain is crucial. Ultimately, using *any* of the aforementioned approaches will be better than doing nothing.

3.5 - PKI certificates and requirements

Kubernetes requires PKI certificates for authentication over TLS. If you install Kubernetes with kubeadm, the certificates that your cluster requires are automatically generated. You can also generate your own certificates -- for example, to keep your private keys more secure by not storing them on the API server. This page explains the certificates that your cluster requires.

How certificates are used by your cluster

Kubernetes requires PKI for the following operations:

- Client certificates for the kubelet to authenticate to the API server
- Server certificate for the API server endpoint
- Client certificates for administrators of the cluster to authenticate to the API server
- Client certificates for the API server to talk to the kubelets
- Client certificate for the API server to talk to etcd
- Client certificate/kubeconfig for the controller manager to talk to the API server
- Client certificate/kubeconfig for the scheduler to talk to the API server.
- Client and server certificates for the front-proxy

> **Note:** `front-proxy` certificates are required only if you run kube-proxy to support an extension API server.

etcd also implements mutual TLS to authenticate clients and peers.

Where certificates are stored

If you install Kubernetes with kubeadm, most certificates are stored in `/etc/kubernetes/pki`. All paths in this documentation are relative to that directory, with the exception of user account certificates which kubeadm places in `/etc/kubernetes`.

Configure certificates manually

If you don't want kubeadm to generate the required certificates, you can create them using a single root CA or by providing all certificates. See Certificates for details on creating your own certificate authority. See Certificate Management with kubeadm for more on managing certificates.

Single root CA

You can create a single root CA, controlled by an administrator. This root CA can then create multiple intermediate CAs, and delegate all further creation to Kubernetes itself.

Required CAs:

path	Default CN	description
ca.crt,key	kubernetes-ca	Kubernetes general CA
etcd/ca.crt,key	etcd-ca	For all etcd-related functions
front-proxy-ca.crt,key	kubernetes-front-proxy-ca	For the front-end proxy

On top of the above CAs, it is also necessary to get a public/private key pair for service account management, `sa.key` and `sa.pub`. The following example illustrates the CA key and certificate files shown in the previous table:

```
/etc/kubernetes/pki/ca.crt
/etc/kubernetes/pki/ca.key
/etc/kubernetes/pki/etcd/ca.crt
/etc/kubernetes/pki/etcd/ca.key
/etc/kubernetes/pki/front-proxy-ca.crt
/etc/kubernetes/pki/front-proxy-ca.key
```

All certificates

If you don't wish to copy the CA private keys to your cluster, you can generate all certificates yourself.

Required certificates:

Default CN	Parent CA	O (in Subject)	kind	hosts (SAN)
kube-etcd	etcd-ca		server, client	`<hostname>`, `<Host_IP>`, `localhost`, `127.0.0.1`
kube-etcd-peer	etcd-ca		server, client	`<hostname>`, `<Host_IP>`, `localhost`, `127.0.0.1`
kube-etcd-healthcheck-client	etcd-ca		client	
kube-apiserver-etcd-client	etcd-ca	system:masters	client	
kube-apiserver	kubernetes-ca		server	`<hostname>`, `<Host_IP>`, `<advertise_IP>`, [1]
kube-apiserver-kubelet-client	kubernetes-ca	system:masters	client	
front-proxy-client	kubernetes-front-proxy-ca		client	

[1]: any other IP or DNS name you contact your cluster on (as used by kubeadm the load balancer stable IP and/or DNS name, `kubernetes`, `kubernetes.default`, `kubernetes.default.svc`, `kubernetes.default.svc.cluster`, `kubernetes.default.svc.cluster.local`)

where `kind` maps to one or more of the x509 key usage types:

kind	Key usage
server	digital signature, key encipherment, server auth
client	digital signature, key encipherment, client auth

> **Note:** Hosts/SAN listed above are the recommended ones for getting a working cluster; if required by a specific setup, it is possible to add additional SANs on all the server

certificates.

Note:
For kubeadm users only:

- The scenario where you are copying to your cluster CA certificates without private keys is referred as external CA in the kubeadm documentation.
- If you are comparing the above list with a kubeadm generated PKI, please be aware that `kube-etcd`, `kube-etcd-peer` and `kube-etcd-healthcheck-client` certificates are not generated in case of external etcd.

Certificate paths

Certificates should be placed in a recommended path (as used by kubeadm). Paths should be specified using the given argument regardless of location.

Default CN	recommended key path	recommended cert path	command	key argument	cert argument
etcd-ca	etcd/ca.key	etcd/ca.crt	kube-apiserver		--etcd-cafile
kube-apiserver-etcd-client	apiserver-etcd-client.key	apiserver-etcd-client.crt	kube-apiserver	--etcd-keyfile	--etcd-certfile
kubernetes-ca	ca.key	ca.crt	kube-apiserver		--client-ca-file
kubernetes-ca	ca.key	ca.crt	kube-controller-manager	--cluster-signing-key-file	--client-ca-file, --root-ca-file, --cluster-signing-cert-file
kube-apiserver	apiserver.key	apiserver.crt	kube-apiserver	--tls-private-key-file	--tls-cert-file
kube-apiserver-kubelet-client	apiserver-kubelet-client.key	apiserver-kubelet-client.crt	kube-apiserver	--kubelet-client-key	--kubelet-client-certificate
front-proxy-ca	front-proxy-ca.key	front-proxy-ca.crt	kube-apiserver		--requestheader-client-ca-file
front-proxy-ca	front-proxy-ca.key	front-proxy-ca.crt	kube-controller-manager		--requestheader-client-ca-file
front-proxy-client	front-proxy-client.key	front-proxy-client.crt	kube-apiserver	--proxy-client-key-file	--proxy-client-cert-file

Default CN	recommended key path	recommended cert path	command	key argument	cert argument
etcd-ca	etcd/ca.key	etcd/ca.crt	etcd		--trusted-ca-file, --peer-trusted-ca-file
kube-etcd	etcd/server.key	etcd/server.crt	etcd	--key-file	--cert-file
kube-etcd-peer	etcd/peer.key	etcd/peer.crt	etcd	--peer-key-file	--peer-cert-file
etcd-ca		etcd/ca.crt	etcdctl		--cacert
kube-etcd-healthcheck-client	etcd/healthcheck-client.key	etcd/healthcheck-client.crt	etcdctl	--key	--cert

Same considerations apply for the service account key pair:

private key path	public key path	command	argument
sa.key		kube-controller-manager	--service-account-private-key-file
	sa.pub	kube-apiserver	--service-account-key-file

The following example illustrates the file paths from the previous tables you need to provide if you are generating all of your own keys and certificates:

```
/etc/kubernetes/pki/etcd/ca.key
/etc/kubernetes/pki/etcd/ca.crt
/etc/kubernetes/pki/apiserver-etcd-client.key
/etc/kubernetes/pki/apiserver-etcd-client.crt
/etc/kubernetes/pki/ca.key
/etc/kubernetes/pki/ca.crt
/etc/kubernetes/pki/apiserver.key
/etc/kubernetes/pki/apiserver.crt
/etc/kubernetes/pki/apiserver-kubelet-client.key
/etc/kubernetes/pki/apiserver-kubelet-client.crt
/etc/kubernetes/pki/front-proxy-ca.key
/etc/kubernetes/pki/front-proxy-ca.crt
/etc/kubernetes/pki/front-proxy-client.key
/etc/kubernetes/pki/front-proxy-client.crt
/etc/kubernetes/pki/etcd/server.key
/etc/kubernetes/pki/etcd/server.crt
/etc/kubernetes/pki/etcd/peer.key
/etc/kubernetes/pki/etcd/peer.crt
/etc/kubernetes/pki/etcd/healthcheck-client.key
/etc/kubernetes/pki/etcd/healthcheck-client.crt
/etc/kubernetes/pki/sa.key
/etc/kubernetes/pki/sa.pub
```

Configure certificates for user accounts

You must manually configure these administrator account and service accounts:

filename	credential name	Default CN	O (in Subject)
admin.conf	default-admin	kubernetes-admin	system:masters
kubelet.conf	default-auth	system:node: `<nodeName>` (see note)	system:nodes
controller-manager.conf	default-controller-manager	system:kube-controller-manager	
scheduler.conf	default-scheduler	system:kube-scheduler	

> **Note:** The value of `<nodeName>` for `kubelet.conf` **must** match precisely the value of the node name provided by the kubelet as it registers with the apiserver. For further details, read the Node Authorization.

1. For each config, generate an x509 cert/key pair with the given CN and O.
2. Run `kubectl` as follows for each config:

```
KUBECONFIG=<filename> kubectl config set-cluster default-cluster --server=https://<host i
KUBECONFIG=<filename> kubectl config set-credentials <credential-name> --client-key <path
KUBECONFIG=<filename> kubectl config set-context default-system --cluster default-cluster
KUBECONFIG=<filename> kubectl config use-context default-system
```

These files are used as follows:

filename	command	comment
admin.conf	kubectl	Configures administrator user for the cluster
kubelet.conf	kubelet	One required for each node in the cluster.
controller-manager.conf	kube-controller-manager	Must be added to manifest in `manifests/kube-controller-manager.yaml`
scheduler.conf	kube-scheduler	Must be added to manifest in `manifests/kube-scheduler.yaml`

The following files illustrate full paths to the files listed in the previous table:

```
/etc/kubernetes/admin.conf
/etc/kubernetes/kubelet.conf
/etc/kubernetes/controller-manager.conf
/etc/kubernetes/scheduler.conf
```