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.

Several Kubernetes components such as kube-apiserver or kube-proxy can also be deployed as container images within the cluster.

It is **recommended** to run Kubernetes components as container images wherever that is possible, and to have Kubernetes manage those components. Components that run containers - notably, the kubelet - can't be included in this category.

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

Install Tools

Set up Kubernetes tools on your computer.

kubectl

The Kubernetes command-line tool, kubectl, allows you to run commands against Kubernetes clusters. You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs. For more information including a complete list of kubectl operations, see the `kubectl` reference documentation.

kubectl is installable on a variety of Linux platforms, macOS and Windows. Find your preferred operating system below.

- Install kubectl on Linux
- Install kubectl on macOS
- Install kubectl on Windows

kind

`kind` lets you run Kubernetes on your local computer. This tool requires that you have either Docker or Podman installed.

The kind Quick Start page shows you what you need to do to get up and running with kind.

View kind Quick Start Guide

minikube

Like `kind`, `minikube` is a tool that lets you run Kubernetes locally. `minikube` runs an all-in-one or a multi-node local Kubernetes cluster on your personal computer (including Windows, macOS and Linux PCs) so that you can try out Kubernetes, or for daily development work.

You can follow the official Get Started! guide if your focus is on getting the tool installed.

View minikube Get Started! Guide

Once you have `minikube` working, you can use it to run a sample application.

kubeadm

You can use the kubeadm tool to create and manage Kubernetes clusters. It performs the actions necessary to get a minimum viable, secure cluster up and running in a user friendly way.

Installing kubeadm shows you how to install kubeadm. Once installed, you can use it to create a cluster.

View kubeadm Install Guide

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.
- *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*: Read Node Autoscaling to learn about the tools available to automatically manage your nodes and the capacity they provide.
- *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, supported authorization modes must be set using an *--authorization-config* file or 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.
- *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.

Container Runtimes

Note: Dockershim has been removed from the Kubernetes project as of release 1.24. Read the Dockershim Removal FAQ for further details.

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

See CRI version support for more information.

This page provides an outline of how to use several common container runtimes with Kubernetes.

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

Note:

Kubernetes releases before v1.24 included a direct integration with Docker Engine, using a component named *dockershim*. That special direct integration is no longer part of Kubernetes (this removal was announced as part of the v1.20 release). You can read Check whether Dockershim removal affects you to understand how this removal might affect you. To learn about migrating from using dockershim, see Migrating from dockershim.

If you are running a version of Kubernetes other than v1.34, check the documentation for that version.

Install and configure prerequisites

Network configuration

By default, the Linux kernel does not allow IPv4 packets to be routed between interfaces. Most Kubernetes cluster networking implementations will change this setting (if needed), but some might expect the administrator to do it for them. (Some might also expect other sysctl parameters to be set, kernel modules to be loaded, etc; consult the documentation for your specific network implementation.)

Enable IPv4 packet forwarding

To manually enable IPv4 packet forwarding:

```
# sysctl params required by setup, params persist across reboots
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.ipv4.ip_forward = 1
EOF

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

Verify that `net.ipv4.ip_forward` is set to 1 with:

```
sysctl net.ipv4.ip_forward
```

cgroup drivers

On Linux, control groups are used to constrain resources that are allocated to processes.

Both the kubelet and the underlying container runtime need to interface with control groups to enforce resource management for pods and containers and set resources such as cpu/memory requests and limits. To interface with control groups, the kubelet and the container runtime need to use a *cgroup driver*. It's critical that the kubelet and the container runtime use the same cgroup driver and are configured the same.

There are two cgroup drivers available:

- `cgroupfs`
- `systemd`

cgroupfs driver

The `cgroupfs` driver is the default cgroup driver in the kubelet. When the `cgroupfs` driver is used, the kubelet and the container runtime directly interface with the cgroup filesystem to configure cgroups.

The `cgroupfs` driver is **not** recommended when systemd is the init system because systemd expects a single cgroup manager on the system. Additionally, if you use cgroup v2, use the `systemd` cgroup driver instead of `cgroupfs`.

systemd cgroup driver

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. As a result, if you use `systemd` as the init system with the `cgroupfs` driver, the system gets two different cgroup managers.

Two cgroup managers result in two views of the available and in-use resources in the system. In some cases, nodes that are configured to use `cgroupfs` for the kubelet and container runtime, but use `systemd` for the rest of the processes become unstable under resource pressure.

The approach to mitigate this instability is to use `systemd` as the cgroup driver for the kubelet and the container runtime when systemd is the selected init system.

To set `systemd` as the cgroup driver, edit the [`KubeletConfiguration`]() option of `cgroupDriver` and set it to `systemd`. For example:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
...
cgroupDriver: systemd
```

Note:

Starting with v1.22 and later, when creating a cluster with kubeadm, if the user does not set the `cgroupDriver` field under `KubeletConfiguration`, kubeadm defaults it to `systemd`.

If you configure `systemd` as the cgroup driver for the kubelet, you must also configure `systemd` as the cgroup driver for the container runtime. Refer to the documentation for your container runtime for instructions. For example:

- [containerd]()
- [CRI-O]()

In Kubernetes 1.34, with the `KubeletCgroupDriverFromCRI` [feature gate]() enabled and a container runtime that supports the `RuntimeConfig` CRI RPC, the kubelet automatically detects the appropriate cgroup driver from the runtime, and ignores the `cgroupDriver` setting within the kubelet configuration.

However, older versions of container runtimes (specifically, containerd 1.y and below) do not support the `RuntimeConfig` CRI RPC, and may not respond correctly to this query, and thus the Kubelet falls back to using the value in its own `--cgroup-driver` flag.

In Kubernetes 1.36, this fallback behavior will be dropped, and older versions of containerd will fail with newer kubelets.

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.

Migrating to the `systemd` driver in kubeadm managed clusters

If you wish to migrate to the `systemd` cgroup driver in existing kubeadm managed clusters, follow [configuring a cgroup driver]().

CRI version support

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

Kubernetes starting v1.26 *only works* with v1 of the CRI API. Earlier versions default to v1 version, however 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 outlines the necessary steps to use containerd as CRI runtime.

To install containerd on your system, follow the instructions on getting started with containerd. Return to this step once you've created a valid `config.toml` configuration file.

- Linux
- Windows

You can find this file under the path `/etc/containerd/config.toml`.

You can find this file under the path `C:\Program Files\containerd\config.toml`.

On Linux the default CRI socket for containerd is `/run/containerd/containerd.sock`. On Windows the default CRI endpoint is `npipe://./pipe/containerd-containerd`.

Configuring the `systemd` cgroup driver

To use the `systemd` cgroup driver in `/etc/containerd/config.toml` with `runc`, set the following config based on your Containerd version

Containerd versions 1.x:

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
  ...

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

Containerd versions 2.x:

```
[plugins.'io.containerd.cri.v1.runtime'.containerd.runtimes.runc]
  ...

[plugins.'io.containerd.cri.v1.runtime'.containerd.runtimes.runc.options]
    SystemdCgroup = true
```

The `systemd` cgroup driver is recommended if you use cgroup v2.

Note:

If you installed containerd from a package (for example, RPM or `.deb`), you may find that the CRI integration plugin is disabled by default.

You need CRI support enabled to use containerd with Kubernetes. Make sure that `cri` is not included in the`disabled_plugins` list within `/etc/containerd/config.toml`; if you made changes to that file, also restart `containerd`.

If you experience container crash loops after the initial cluster installation or after installing a CNI, the containerd configuration provided with the package might contain incompatible configuration parameters. Consider resetting the containerd configuration with `containerd config default > /etc/containerd/config.toml` as specified in getting-started.md and then set the configuration parameters specified above accordingly.

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

```
sudo systemctl restart containerd
```

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

In Kubernetes v1.28, you can enable automatic detection of the cgroup driver as an alpha feature. See systemd cgroup driver for more details.

Overriding the sandbox (pause) image

In your containerd config you can overwrite the sandbox image by setting the following config:

```
[plugins."io.containerd.grpc.v1.cri"]
  sandbox_image = "registry.k8s.io/pause:3.10"
```

You might need to restart `containerd` as well once you've updated the config file: `systemctl restart containerd`.

CRI-O

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

To install CRI-O, follow CRI-O Install Instructions.

cgroup driver

CRI-O uses the systemd cgroup driver per default, which is likely to work fine for you. 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"
```

You should 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.

In Kubernetes v1.28, you can enable automatic detection of the cgroup driver as an alpha feature. See systemd cgroup driver for more details.

For CRI-O, the CRI socket is `/var/run/crio/crio.sock` by default.

Overriding the sandbox (pause) image

In your CRI-O config you can set the following config value:

```
[crio.image]
pause_image="registry.k8s.io/pause:3.10"
```

This config option supports live configuration reload to apply this change: `systemctl reload crio` or by sending `SIGHUP` to the `crio` process.

Docker Engine

Note:

These instructions assume that you are using the `cri-dockerd` adapter to integrate Docker Engine with Kubernetes.

1. On each of your nodes, install Docker for your Linux distribution as per Install Docker Engine.

2. Install `cri-dockerd`, following the directions in the install section of the documentation.

For `cri-dockerd`, the CRI socket is `/run/cri-dockerd.sock` by default.

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.

To learn more about how to install Mirantis Container Runtime, visit MCR Deployment Guide.

Check the systemd unit named `cri-docker.socket` to find out the path to the CRI socket.

Overriding the sandbox (pause) image

The `cri-dockerd` adapter accepts a command line argument for specifying which container image to use as the Pod infrastructure container ("pause image"). The command line argument to use is `--pod-infra-container-image`.

What's next

As well as a container runtime, your cluster will need a working network plugin.

Installing Kubernetes with deployment tools

There are many methods and tools for setting up your own production Kubernetes cluster. For example:

- kubeadm
- Cluster API: A Kubernetes sub-project focused on providing declarative APIs and tooling to simplify provisioning, upgrading, and operating multiple Kubernetes clusters.
- kops: An automated cluster provisioning tool. For tutorials, best practices, configuration options and information on reaching out to the community, please check the `kOps` website for details.
- kubespray: A composition of Ansible playbooks, inventory, provisioning tools, and domain knowledge for generic OS/Kubernetes clusters configuration management tasks. You can reach out to the community on Slack channel #kubespray.

Bootstrapping clusters with kubeadm

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 Creating a cluster with kubeadm page.

This installation guide is for Kubernetes v1.34. If you want to use a different Kubernetes version, please refer to the following pages instead:

- Installing kubeadm (Kubernetes v1.33)
- Installing kubeadm (Kubernetes v1.32)

- Installing kubeadm (Kubernetes v1.31)
- Installing kubeadm (Kubernetes v1.30)

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 for control plane machines.
- 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.

Note:

The `kubeadm` installation is done via binaries that use dynamic linking and assumes that your target system provides `glibc`. This is a reasonable assumption on many Linux distributions (including Debian, Ubuntu, Fedora, CentOS, etc.) but it is not always the case with custom and lightweight distributions which don't include `glibc` by default, such as Alpine Linux. The expectation is that the distribution either includes `glibc` or a compatibility layer that provides the expected symbols.

Check your OS version

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.

- Linux
- Windows

- The kubeadm project supports LTS kernels. See List of LTS kernels.
- You can get the kernel version using the command `uname -r`

For more information, see Linux Kernel Requirements.

- The kubeadm project supports recent kernel versions. For a list of recent kernels, see Windows Server Release Information.
- You can get the kernel version (also called the OS version) using the command `systeminfo`

For more information, see Windows OS version compatibility.

A Kubernetes cluster created by kubeadm depends on software that use kernel features. This software includes, but is not limited to the container runtime, the kubelet, and a Container Network Interface plugin.

To help you avoid unexpected errors as a result of an unsupported kernel version, kubeadm runs the `SystemVerification` pre-flight check. This check fails if the kernel version is not supported.

You may choose to skip the check, if you know that your kernel provides the required features, even though kubeadm does not support its version.

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.

Check required ports

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

```
nc 127.0.0.1 6443 -zv -w 2
```

The pod network plugin you use 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.

Swap configuration

The default behavior of a kubelet is to fail to start if swap memory is detected on a node. This means that swap should either be disabled or tolerated by kubelet.

- To tolerate swap, add `failSwapOn: false` to kubelet configuration or as a command line argument. Note: even if `failSwapOn: false` is provided, workloads wouldn't have swap access by default. This can be changed by setting a `swapBehavior`, again in the kubelet configuration file. To use swap, set a `swapBehavior` other than the default `NoSwap` setting. See Swap memory management for more details.
- To disable swap, `sudo swapoff -a` can be used to disable swapping temporarily. To make this change persistent across reboots, make sure swap is disabled in config files like `/etc/fstab`, `systemd.swap`, depending how it was configured on your system.

Installing a container runtime

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

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 known endpoints.

If multiple or no container runtimes are detected kubeadm will throw an error and will request that you specify which one you want to use.

See container runtimes for more information.

Note:

Docker Engine does not implement the CRI which is a requirement for a container runtime to work with Kubernetes. For that reason, an additional service cri-dockerd has to be installed. cri-dockerd is a project based on the legacy built-in Docker Engine support that was removed from the kubelet in version 1.24.

The tables below include the known endpoints for supported operating systems:

- Linux
- Windows

Linux container runtimes

Runtime	Path to Unix domain socket
containerd	`unix:///var/run/containerd/containerd.sock`
CRI-O	`unix:///var/run/crio/crio.sock`
Docker Engine (using cri-dockerd)	`unix:///var/run/cri-dockerd.sock`

Windows container runtimes

Runtime	Path to Windows named pipe
containerd	`npipe:////./pipe/containerd-containerd`
Docker Engine (using cri-dockerd)	`npipe:////./pipe/cri-dockerd`

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

Note: The legacy package repositories (`apt.kubernetes.io` and `yum.kubernetes.io`) have been deprecated and frozen starting from September 13, 2023. **Using the new package repositories hosted at `pkgs.k8s.io` is strongly recommended and required in order to install Kubernetes versions released after September 13, 2023.** The deprecated legacy repositories, and their contents, might be removed at any time in the future and without a further notice period. The new package repositories provide downloads for Kubernetes versions starting with v1.24.0.

Note:

There's a dedicated package repository for each Kubernetes minor version. If you want to install a minor version other than v1.34, please see the installation guide for your desired minor version.

- Debian-based distributions
- Red Hat-based distributions
- Without a package manager

These instructions are for Kubernetes v1.34.

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

```
sudo apt-get update
# apt-transport-https may be a dummy package; if so, you can
skip that package
sudo apt-get install -y apt-transport-https ca-certificates
curl gpg
```

2. Download the public signing key for the Kubernetes package repositories. The same signing key is used for all repositories so you can disregard the version in the URL:

```
# If the directory `/etc/apt/keyrings` does not exist, it
should be created before the curl command, read the note
below.
# sudo mkdir -p -m 755 /etc/apt/keyrings
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.34/deb/
Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/
kubernetes-apt-keyring.gpg
```

Note:

In releases older than Debian 12 and Ubuntu 22.04, directory `/etc/apt/keyrings` does not exist by default, and it should be created before the curl command.

1. Add the appropriate Kubernetes `apt` repository. Please note that this repository have packages only for Kubernetes 1.34; for other Kubernetes minor versions, you need to change

the Kubernetes minor version in the URL to match your desired minor version (you should also check that you are reading the documentation for the version of Kubernetes that you plan to install).

```
# This overwrites any existing configuration in /etc/apt/
sources.list.d/kubernetes.list
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-
keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.34/deb/ /'
| sudo tee /etc/apt/sources.list.d/kubernetes.list
```

2. Update the `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
```

3. (Optional) Enable the kubelet service before running kubeadm:

```
sudo systemctl enable --now kubelet
```

1. Set SELinux to `permissive` mode:

   These instructions are for Kubernetes 1.34.

```
# Set SELinux in permissive mode (effectively disabling it)
sudo setenforce 0
sudo sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/' /etc/
selinux/config
```

Caution:

- Setting SELinux in permissive mode by running `setenforce 0` and `sed ...` effectively disables it. This is required to allow containers to access the host filesystem; for example, some cluster network plugins require that. You have to do this until SELinux support is improved in the kubelet.
- You can leave SELinux enabled if you know how to configure it but it may require settings that are not supported by kubeadm.

1. Add the Kubernetes `yum` repository. The `exclude` parameter in the repository definition ensures that the packages related to Kubernetes are not upgraded upon running `yum update` as there's a special procedure that must be followed for upgrading Kubernetes. Please note that this repository have packages only for Kubernetes 1.34; for other Kubernetes minor versions, you need to change the Kubernetes minor version in the URL to match your desired minor version (you should also check that you are reading the documentation for the version of Kubernetes that you plan to install).

```
# This overwrites any existing configuration in /etc/
yum.repos.d/kubernetes.repo
cat <<EOF | sudo tee /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/v1.34/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/v1.34/rpm/repodata/
repomd.xml.key
exclude=kubelet kubeadm kubectl cri-tools kubernetes-cni
EOF
```

2. Install kubelet, kubeadm and kubectl:

   For systems with DNF:

   ```
   sudo yum install -y kubelet kubeadm kubectl --
   disableexcludes=kubernetes
   ```

   For systems with DNF5:

   ```
   sudo yum install -y kubelet kubeadm kubectl --setopt=disable_
   excludes=kubernetes
   ```

3. (Optional) Enable the kubelet service before running kubeadm:

   ```
   sudo systemctl enable --now kubelet
   ```

Install CNI plugins (required for most pod network):

```
CNI_PLUGINS_VERSION="v1.3.0"
ARCH="amd64"
DEST="/opt/cni/bin"
sudo mkdir -p "$DEST"
curl -L "https://github.com/containernetworking/plugins/releases/
download/${CNI_PLUGINS_VERSION}/cni-plugins-linux-${ARCH}-${CNI_P
LUGINS_VERSION}.tgz" | sudo tar -C "$DEST" -xz
```

Define the directory to download command files:

Note:

The `DOWNLOAD_DIR` variable must be set to a writable directory. If you are running Flatcar Container Linux, set `DOWNLOAD_DIR="/opt/bin"`.

```
DOWNLOAD_DIR="/usr/local/bin"
sudo mkdir -p "$DOWNLOAD_DIR"
```

Optionally install crictl (required for interaction with the Container Runtime Interface (CRI), optional for kubeadm):

```
CRICTL_VERSION="v1.31.0"
ARCH="amd64"
curl -L "https://github.com/kubernetes-sigs/cri-tools/releases/
download/${CRICTL_VERSION}/crictl-${CRICTL_VERSION}-linux-$
{ARCH}.tar.gz" | sudo tar -C $DOWNLOAD_DIR -xz
```

Install `kubeadm`, `kubelet` and add a `kubelet` systemd service:

```
RELEASE="$(curl -sSL https://dl.k8s.io/release/stable.txt)"
ARCH="amd64"
cd $DOWNLOAD_DIR
sudo curl -L --remote-name-all https://dl.k8s.io/release/${RELEAS
E}/bin/linux/${ARCH}/{kubeadm,kubelet}
sudo chmod +x {kubeadm,kubelet}

RELEASE_VERSION="v0.16.2"
curl -sSL "https://raw.githubusercontent.com/kubernetes/release/$
{RELEASE_VERSION}/cmd/krel/templates/latest/kubelet/
kubelet.service" | sed "s:/usr/bin:${DOWNLOAD_DIR}:g" | sudo
```

```
tee /usr/lib/systemd/system/kubelet.service
sudo mkdir -p /usr/lib/systemd/system/kubelet.service.d
curl -sSL "https://raw.githubusercontent.com/kubernetes/release/${RELEASE_VERSION}/cmd/krel/templates/latest/kubeadm/10-kubeadm.conf" | sed "s:/usr/bin:${DOWNLOAD_DIR}:g" | sudo tee /usr/lib/systemd/system/kubelet.service.d/10-kubeadm.conf
```

Note:

Please refer to the note in the Before you begin section for Linux distributions that do not include `glibc` by default.

Install `kubectl` by following the instructions on Install Tools page.

Optionally, enable the kubelet service before running kubeadm:

```
sudo systemctl enable --now kubelet
```

Note:

The Flatcar Container Linux distribution mounts the `/usr` directory as a read-only filesystem. Before bootstrapping your cluster, you need to take additional steps to configure a writable directory. See the Kubeadm Troubleshooting guide to learn how to set up a writable directory.

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

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 ...`:

```
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 the container runtime 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 container runtime and then re-run `kubeadm reset`. You can also use `crictl` to debug the state of the container runtime. See Debugging Kubernetes nodes with crictl.

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.

`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 because of "crypto/rsa: verification
error" while trying to verify candidate authority certificate
"kubernetes")
```

- 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` (see Generating kubeconfig files for additional users). 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-glc5m/mysql: dial tcp 10.19.0.41:10250:
getsockopt: no route to host
```

- 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: starting container process caused
"process_linux.go:110: decoding init error from pipe caused
\"read parent: connection reset by peer\""
```

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.git8633870.el7.centos.x86_64 docker-
  common-1.13.1-75.git8633870.el7.centos.x86_64
  ```

- 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.repo
  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": {
        "tolerations": [
          {
            "key": "CriticalAddonsOnly",
            "operator": "Exists"
          },
          {
            "effect": "NoSchedule",
            "key": "node-role.kubernetes.io/control-plane"
          }
        ]
      }
    }
  }
}'
```

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`:

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: InitConfiguration
nodeRegistration:
  kubeletExtraArgs:
  - name: "volume-plugin-dir"
    value: "/opt/libexec/kubernetes/kubelet-plugins/volume/exec/"
---
apiVersion: kubeadm.k8s.io/v1beta4
kind: ClusterConfiguration
controllerManager:
  extraArgs:
  - name: "flex-volume-plugin-dir"
    value: "/opt/libexec/kubernetes/kubelet-plugins/volume/exec/"
```

On joining Nodes:

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: JoinConfiguration
nodeRegistration:
  kubeletExtraArgs:
  - name: "volume-plugin-dir"
    value: "/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.

Upgrade fails due to etcd hash not changing

Only applicable to upgrading a control plane node with a kubeadm binary v1.28.3 or later, where the node is currently managed by kubeadm versions v1.28.0, v1.28.1 or v1.28.2.

Here is the error message you may encounter:

```
[upgrade/etcd] Failed to upgrade etcd: couldn't upgrade control
plane. kubeadm has tried to recover everything into the earlier
state. Errors faced: static Pod hash for component etcd on Node
kinder-upgrade-control-plane-1 did not change after 5m0s: timed
out waiting for the condition
[upgrade/etcd] Waiting for previous etcd to become available
I0907 10:10:09.109104    3704 etcd.go:588] [etcd] attempting to
see if all cluster endpoints ([https://172.17.0.6:2379/ https://
172.17.0.4:2379/ https://172.17.0.3:2379/]) are available 1/10
[upgrade/etcd] Etcd was rolled back and is now available
static Pod hash for component etcd on Node kinder-upgrade-
control-plane-1 did not change after 5m0s: timed out waiting for
the condition
couldn't upgrade control plane. kubeadm has tried to recover
everything into the earlier state. Errors faced
k8s.io/kubernetes/cmd/kubeadm/app/phases/
```

```
upgrade.rollbackOldManifests
        cmd/kubeadm/app/phases/upgrade/staticpods.go:525
k8s.io/kubernetes/cmd/kubeadm/app/phases/upgrade.upgradeComponent
        cmd/kubeadm/app/phases/upgrade/staticpods.go:254
k8s.io/kubernetes/cmd/kubeadm/app/phases/
upgrade.performEtcdStaticPodUpgrade
        cmd/kubeadm/app/phases/upgrade/staticpods.go:338
...
```

The reason for this failure is that the affected versions generate an etcd manifest file with unwanted defaults in the PodSpec. This will result in a diff from the manifest comparison, and kubeadm will expect a change in the Pod hash, but the kubelet will never update the hash.

There are two way to workaround this issue if you see it in your cluster:

- The etcd upgrade can be skipped between the affected versions and v1.28.3 (or later) by using:

  ```
  kubeadm upgrade {apply|node} [version] --etcd-upgrade=false
  ```

  This is not recommended in case a new etcd version was introduced by a later v1.28 patch version.

- Before upgrade, patch the manifest for the etcd static pod, to remove the problematic defaulted attributes:

  ```
  diff --git a/etc/kubernetes/manifests/etcd_defaults.yaml b/
  etc/kubernetes/manifests/etcd_origin.yaml
  index d807ccbe0aa..46b35f00e15 100644
  --- a/etc/kubernetes/manifests/etcd_defaults.yaml
  +++ b/etc/kubernetes/manifests/etcd_origin.yaml
  @@ -43,7 +43,6 @@ spec:
          scheme: HTTP
        initialDelaySeconds: 10
        periodSeconds: 10
  -      successThreshold: 1
        timeoutSeconds: 15
      name: etcd
      resources:
  @@ -59,26 +58,18 @@ spec:
          scheme: HTTP
        initialDelaySeconds: 10
        periodSeconds: 10
  -      successThreshold: 1
        timeoutSeconds: 15
  -    terminationMessagePath: /dev/termination-log
  -    terminationMessagePolicy: File
      volumeMounts:
      - mountPath: /var/lib/etcd
        name: etcd-data
      - mountPath: /etc/kubernetes/pki/etcd
        name: etcd-certs
  -  dnsPolicy: ClusterFirst
  -  enableServiceLinks: true
    hostNetwork: true
    priority: 2000001000
    priorityClassName: system-node-critical
  -  restartPolicy: Always
  -  schedulerName: default-scheduler
  ```

```
  securityContext:
    seccompProfile:
      type: RuntimeDefault
-  terminationGracePeriodSeconds: 30
  volumes:
  - hostPath:
      path: /etc/kubernetes/pki/etcd
```

More information can be found in the tracking issue for this bug.

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 v1.34.

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

Preparing the hosts

Component installation

Install a container runtime and kubeadm on all the hosts. For detailed instructions and other prerequisites, see Installing kubeadm.

Note:

If you have already installed kubeadm, see the first two steps of the Upgrading Linux nodes document for instructions on how to upgrade 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.

Network setup

kubeadm similarly to other Kubernetes components tries to find a usable IP on the network interfaces associated with a default gateway on a host. Such an IP is then used for the advertising and/or listening performed by a component.

To find out what this IP is on a Linux host you can use:

```
ip route show # Look for a line starting with "default via"
```

Note:

If two or more default gateways are present on the host, a Kubernetes component will try to use the first one it encounters that has a suitable global unicast IP address. While making this choice, the exact ordering of gateways might vary between different operating systems and kernel versions.

Kubernetes components do not accept custom network interface as an option, therefore a custom IP address must be passed as a flag to all components instances that need such a custom configuration.

Note:

If the host does not have a default gateway and if a custom IP address is not passed to a Kubernetes component, the component may exit with an error.

To configure the API server advertise address for control plane nodes created with both `init` and `join`, the flag `--apiserver-advertise-address` can be used. Preferably, this option can be set in the kubeadm API as `InitConfiguration.localAPIEndpoint` and `JoinConfiguration.controlPlane.localAPIEndpoint`.

For kubelets on all nodes, the `--node-ip` option can be passed in `.nodeRegistration.kubeletExtraArgs` inside a kubeadm configuration file (`InitConfiguration` or `JoinConfiguration`).

For dual-stack see Dual-stack support with kubeadm.

The IP addresses that you assign to control plane components become part of their X.509 certificates' subject alternative name fields. Changing these IP addresses would require signing new certificates and restarting the affected components, so that the change in certificate files is reflected. See Manual certificate renewal for more details on this topic.

Warning:

The Kubernetes project recommends against this approach (configuring all component instances with custom IP addresses). Instead, the Kubernetes maintainers recommend to setup the host network, so that the default gateway IP is the one that Kubernetes components auto-detect and use. On Linux nodes, you can use commands such as `ip route` to configure networking; your operating system might also provide higher level network management tools. If your node's default gateway is a public IP address, you should configure packet filtering or other security measures that protect the nodes and your cluster.

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 `registry.k8s.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) `kubeadm` tries to detect the container runtime by using a list of well known endpoints. To use different container runtime or if there are more than one installed on the provisioned node, specify the `--cri-socket` argument to `kubeadm`. See Installing a runtime.

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 advertised 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 a 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 reconfigure a cluster that has already been created see Reconfiguring a kubeadm cluster.

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-token-ca-cert-hash sha256:<hash>
```

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:

The kubeconfig file `admin.conf` that `kubeadm init` generates contains a certificate with `Subject: O = kubeadm:cluster-admins, CN = kubernetes-admin`. The group `kubeadm:cluster-admins` is bound to the built-in `cluster-admin` ClusterRole. Do not share the `admin.conf` file with anyone.

`kubeadm init` generates another kubeconfig file `super-admin.conf` that contains a certificate with `Subject: O = system:masters, CN = kubernetes-super-admin`. `system:masters` is a break-glass, super user group that bypasses the authorization layer (for example RBAC). Do not share the `super-admin.conf` file with anyone. It is recommended to move the file to a safe location.

See Generating kubeconfig files for additional users on how to use `kubeadm kubeconfig user` to generate 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.

Please refer to the Installing Addons page for a non-exhaustive list of networking addons supported by Kubernetes. 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>
```

Note:

Only a few CNI plugins support Windows. More details and setup instructions can be found in Adding Windows worker nodes.

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 nodes for security reasons. If you want to be able to schedule Pods on the control plane nodes, for example for a single machine Kubernetes cluster, run:

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

The output will look something like:

```
node "test-01" untainted
...
```

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

Additionally, you can execute the following command to remove the `node.kubernetes.io/exclude-from-external-load-balancers` label from the control plane node, which excludes it from the list of backend servers:

```
kubectl label nodes --all node.kubernetes.io/exclude-from-
external-load-balancers-
```

Adding more control plane nodes

See Creating Highly Available Clusters with kubeadm for steps on creating a high availability kubeadm cluster by adding more control plane nodes.

Adding worker nodes

The worker nodes are where your workloads run.

The following pages show how to add Linux and Windows worker nodes to the cluster by using the `kubeadm join` command:

- Adding Linux worker nodes
- Adding Windows worker nodes

(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 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.

Version skew policy

While kubeadm allows version skew against some components that it manages, it is recommended that you match the kubeadm version with the versions of the control plane components, kube-proxy and kubelet.

kubeadm's skew against the Kubernetes version

kubeadm can be used with Kubernetes components that are the same version as kubeadm or one version older. The Kubernetes version can be specified to kubeadm by using the `--kubernetes-version` flag of `kubeadm init` or the [`ClusterConfiguration.kubernetesVersion`] field when using `--config`. This option will control the versions of kube-apiserver, kube-controller-manager, kube-scheduler and kube-proxy.

Example:

- kubeadm is at 1.34
- `kubernetesVersion` must be at 1.34 or 1.33

kubeadm's skew against the kubelet

Similarly to the Kubernetes version, kubeadm can be used with a kubelet version that is the same version as kubeadm or three versions older.

Example:

- kubeadm is at 1.34
- kubelet on the host must be at 1.34, 1.33, 1.32 or 1.31

kubeadm's skew against kubeadm

There are certain limitations on how kubeadm commands can operate on existing nodes or whole clusters managed by kubeadm.

If new nodes are joined to the cluster, the kubeadm binary used for `kubeadm join` must match the last version of kubeadm used to either create the cluster with `kubeadm init` or to upgrade the same node with `kubeadm upgrade`. Similar rules apply to the rest of the kubeadm commands with the exception of `kubeadm upgrade`.

Example for `kubeadm join`:

- kubeadm version 1.34 was used to create a cluster with `kubeadm init`
- Joining nodes must use a kubeadm binary that is at version 1.34

Nodes that are being upgraded must use a version of kubeadm that is the same MINOR version or one MINOR version newer than the version of kubeadm used for managing the node.

Example for `kubeadm upgrade`:

- kubeadm version 1.33 was used to create or upgrade the node
- The version of kubeadm used for upgrading the node must be at 1.33 or 1.34

To learn more about the version skew between the different Kubernetes component see the Version Skew Policy.

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.

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

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:

To reconfigure a cluster that has already been created see Reconfiguring a kubeadm cluster.

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 `name` / `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/v1beta4
kind: ClusterConfiguration
kubernetesVersion: v1.16.0
apiServer:
  extraArgs:
  - name: "enable-admission-plugins"
    value: "AlwaysPullImages,DefaultStorageClass"
  - name: "audit-log-path"
    value: "/home/johndoe/audit.log"
```

ControllerManager flags

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

Example usage:

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: ClusterConfiguration
kubernetesVersion: v1.16.0
controllerManager:
  extraArgs:
  - name: "cluster-signing-key-file"
    value: "/home/johndoe/keys/ca.key"
  - name: "deployment-controller-sync-period"
    value: "50"
```

Scheduler flags

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

Example usage:

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: ClusterConfiguration
kubernetesVersion: v1.16.0
scheduler:
  extraArgs:
  - name: "config"
    value: "/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/v1beta4
kind: ClusterConfiguration
etcd:
  local:
    extraArgs:
    - name: "election-timeout"
      value: 1000
```

Customizing 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 component configuration is written to disk.

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

```
apiVersion: kubeadm.k8s.io/v1beta4
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/v1beta4
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`, `etcd`, `kubeletconfiguration` and `corednsdeployment`.
- `suffix` is an optional string that can be used to determine which patches are applied first alpha-numerically.

- `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`.

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` and kubeadm will apply the same base `KubeletConfiguration` to all nodes in the cluster.

For applying instance-specific configuration over the base `KubeletConfiguration` you can use the `kubeletconfiguration` patch target.

Alternatively, 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 additional 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.

Customizing CoreDNS

kubeadm allows you to customize the CoreDNS Deployment with patches against the `corednsdeployment` patch target.

Patches for other CoreDNS related API objects like the `kube-system/coredns` ConfigMap are currently not supported. You must manually patch any of these objects using kubectl and recreate the CoreDNS Pods after that.

Alternatively, you can disable the kubeadm CoreDNS deployment by including the following option in your `ClusterConfiguration`:

```
dns:
  disabled: true
```

Also, by executing the following command:

```
kubeadm init phase addon coredns --print-manifest --config my-
config.yaml`
```

you can obtain the manifest file kubeadm would create for CoreDNS on your setup.

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`.

Stacked etcd topology

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.

External etcd topology

What's next

- Set up a highly available cluster with kubeadm

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.

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.

And you also need:

- Three or more additional machines, that will become etcd cluster members. Having an odd number of members in the etcd cluster is a requirement for achieving optimal voting quorum.
    - These machines again need to have `kubeadm` and `kubelet` installed.
    - These machines also require a container runtime, that is already set up and working.

See External etcd topology for context.

Container images

Each host should have access read and fetch images from the Kubernetes container image registry, `registry.k8s.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 -zv -w 2 <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" --upload-certs
    ```

    - 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 on each as a root:
    kubeadm join 192.168.0.200:6443 --token
9vr73a.a8uxyaju799qwdjv --discovery-token-ca-cert-hash
sha256:7c2e69131a36ae2a042a339b33381c6d0d43887e2de83720eff535
9e26aec866 --control-plane --certificate-key
f8902e114ef118304e561c3ecd4d0b543adc226b7a07f675f56564185ffe0
c07

Please note that the certificate-key gives access to cluster
sensitive data, keep it secret!
As a safeguard, uploaded-certs will be deleted in two hours;
If necessary, you can use kubeadm init phase upload-certs to
reload certs afterward.

Then you can join any number of worker nodes by running the
following on each as root:
    kubeadm join 192.168.0.200:6443 --token
9vr73a.a8uxyaju799qwdjv --discovery-token-ca-cert-hash
sha256:7c2e69131a36ae2a042a339b33381c6d0d43887e2de83720eff535
9e26aec866
```

- 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
  ```

The certificate key is a hex encoded string that is an AES key of size 32 bytes.

Note:

The `kubeadm-certs` Secret and the 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-token-ca-cert-hash
   sha256:7c2e69131a36ae2a042a339b33381c6d0d43887e2de83720eff535
   9e26aec866 --control-plane --certificate-key
   f8902e114ef118304e561c3ecd4d0b543adc226b7a07f675f56564185ffe0
   c07
   ```

   - 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.

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.

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. Set up 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/v1beta4
kind: ClusterConfiguration
kubernetesVersion: stable
controlPlaneEndpoint: "LOAD_BALANCER_DNS:LOAD_BALANCER_PORT"
# change this (see below)
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-cert-hash
sha256:7c2e69131a36ae2a042a339b33381c6d0d43887e2de83720eff5359e26
aec866
```

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}"@$hos
   t:
       scp /etc/kubernetes/pki/front-proxy-ca.key "${USER}"@$hos
   t:
       scp /etc/kubernetes/pki/etcd/ca.crt "${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
```

Set up a High Availability etcd Cluster with kubeadm

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 (`registry.k8s.io`) or list/pull the required etcd image using `kubeadm config images list/pull`. This guide will set up 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 cryptographic 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/
   kubelet.conf
   # Replace "systemd" with the cgroup driver of your container
   runtime. The default value in the kubelet is "cgroupfs".
   # Replace the value of "containerRuntimeEndpoint" for a
   different container runtime if needed.
   #
   apiVersion: kubelet.config.k8s.io/v1beta1
   kind: KubeletConfiguration
   authentication:
     anonymous:
       enabled: false
     webhook:
       enabled: false
   authorization:
     mode: AlwaysAllow
   cgroupDriver: systemd
   address: 127.0.0.1
   containerRuntimeEndpoint: unix:///var/run/containerd/
   containerd.sock
   staticPodPath: /etc/kubernetes/manifests
   EOF

   cat << EOF > /etc/systemd/system/kubelet.service.d/20-etcd-
   service-manager.conf
   [Service]
   ExecStart=
   ExecStart=/usr/bin/kubelet --config=/etc/systemd/system/
   kubelet.service.d/kubelet.conf
   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/v1beta4"
kind: InitConfiguration
nodeRegistration:
    name: ${NAME}
localAPIEndpoint:
    advertiseAddress: ${HOST}
---
apiVersion: "kubeadm.k8s.io/v1beta4"
kind: ClusterConfiguration
etcd:
    local:
        serverCertSANs:
        - "${HOST}"
        peerCertSANs:
        - "${HOST}"
        extraArgs:
        - name: initial-cluster
          value: ${NAMES[0]}=https://${HOSTS[0]}:2380,$
{NAMES[1]}=https://${HOSTS[1]}:2380,${NAMES[2]}=https://$
{HOSTS[2]}:2380
        - name: initial-cluster-state
          value: new
        - name: name
          value: ${NAME}
        - name: listen-peer-urls
          value: https://${HOST}:2380
        - name: listen-client-urls
          value: https://${HOST}:2379
        - name: advertise-client-urls
          value: https://${HOST}:2379
        - name: initial-advertise-peer-urls
          value: 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.yaml
   kubeadm init phase certs apiserver-etcd-client --config=/tmp/
   ${HOST2}/kubeadmcfg.yaml
   cp -R /etc/kubernetes/pki /tmp/${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.yaml
   kubeadm init phase certs apiserver-etcd-client --config=/tmp/
   ${HOST1}/kubeadmcfg.yaml
   cp -R /etc/kubernetes/pki /tmp/${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.yaml
   kubeadm init phase certs apiserver-etcd-client --config=/tmp/
   ${HOST0}/kubeadmcfg.yaml
   # 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/${HO
   ST0}/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.

   If `etcdctl` isn't available, you can run this tool inside a container image. You would do that directly with your container runtime using a tool such as `crictl run` and not through Kubernetes

   ```
   ETCDCTL_API=3 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
   ...
   https://[HOST0 IP]:2379 is healthy: successfully committed
   proposal: took = 16.283339ms
   https://[HOST1 IP]:2379 is healthy: successfully committed
   proposal: took = 19.44402ms
   https://[HOST2 IP]:2379 is healthy: successfully committed
   proposal: took = 35.926451ms
   ```

   - Set `${HOST0}`to the IP address of the host you are testing.

What's next

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

Configuring each kubelet in your cluster using kubeadm

Note: Dockershim has been removed from the Kubernetes project as of release 1.24. Read the Dockershim Removal FAQ for further details.
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 container 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 container runtime, but the value of `--cgroup-driver` must match the cgroup driver used by the container runtime to ensure the health of the kubelet.

- To specify the container runtime you must set its endpoint with the `--container-runtime-endpoint=<path>` flag.

The recommended way of applying such instance-specific configuration is by using `KubeletConfiguration` patches.

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.

It is also possible to apply instance-specific patches over the base `KubeletConfiguration`. Have a look at Customizing the kubelet for more details.

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 `kubelet-config` ConfigMap in the `kube-system` namespace of the cluster. Additionally, the kubeadm tool detects the CRI socket on the node and writes its details (including the socket path) into a local configuration, `/var/lib/kubelet/instance-config.yaml`. 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.

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` ConfigMap and writes it to `/var/lib/kubelet/config.yaml`. Additionally, the kubeadm tool detects the CRI socket on the node and writes its details (including the socket path) into a local configuration, `/var/lib/kubelet/instance-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` package is written to `/usr/lib/systemd/system/kubelet.service.d/10-kubeadm.conf` and is used by systemd. It augments the basic `kubelet.service`.

If you want to override that further, you can make a directory `/etc/systemd/system/kubelet.service.d/` (not `/usr/lib/systemd/system/kubelet.service.d/`) and put your own customizations into a file there. For example, you might add a new local file `/etc/systemd/system/kubelet.service.d/local-overrides.conf` to override the unit settings configured by `kubeadm`.

Here is what you are likely to find in `/usr/lib/systemd/system/kubelet.service.d/10-kubeadm.conf`:

Note:

The contents below are just an example. If you don't want to use a package manager follow the guide outlined in the (Without a package manager) section.

```
[Service]
Environment="KUBELET_KUBECONFIG_ARGS=--bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --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. Preferably,
# the user should use the .NodeRegistration.KubeletExtraArgs
object in the configuration files instead.
# 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_ARGS $KUBELET_EXTRA_ARGS
```

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 `/usr/bin/kubelet` binary.
`kubectl`	Installs the `/usr/bin/kubectl` binary.
`cri-tools`	Installs the `/usr/bin/crictl` binary from the cri-tools git repository.
`kubernetes-cni`	Installs the `/opt/cni/bin` binaries from the plugins git repository.

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.

Enable IPv6 packet forwarding

To check if IPv6 packet forwarding is enabled:

```
sysctl net.ipv6.conf.all.forwarding
```

If the output is `net.ipv6.conf.all.forwarding = 1` it is already enabled. Otherwise it is not enabled yet.

To manually enable IPv6 packet forwarding:

```
# sysctl params required by setup, params persist across reboots
cat <<EOF | sudo tee -a /etc/sysctl.d/k8s.conf
net.ipv6.conf.all.forwarding = 1
EOF

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

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=10.96.0.0/16,2001:db8:42:1::/112
```

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/v1beta4
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/v1beta4
kind: InitConfiguration
localAPIEndpoint:
  advertiseAddress: "10.100.0.1"
  bindPort: 6443
```

```
nodeRegistration:
  kubeletExtraArgs:
  - name: "node-ip"
    value: "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/v1beta4
kind: JoinConfiguration
discovery:
  bootstrapToken:
    apiServerEndpoint: 10.100.0.1:6443
    token: "clvldh.vjjwg16ucnhp94qr"
    caCertHashes:
    - "sha256:a4863cde706cfc580a439f842cc65d5ef112b7b2be31628513a
9881cf0d9fe0e"
    # change auth info above to match the actual token and CA
certificate hash for your cluster
nodeRegistration:
  kubeletExtraArgs:
  - name: "node-ip"
    value: "10.100.0.2,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/v1beta4
kind: JoinConfiguration
controlPlane:
  localAPIEndpoint:
    advertiseAddress: "10.100.0.2"
    bindPort: 6443
discovery:
  bootstrapToken:
    apiServerEndpoint: 10.100.0.1:6443
    token: "clvldh.vjjwg16ucnhp94qr"
    caCertHashes:
```

```
    - "sha256:a4863cde706cfc580a439f842cc65d5ef112b7b2be31628513a
9881cf0d9fe0e"
    # change auth info above to match the actual token and CA
certificate hash for your cluster
nodeRegistration:
  kubeletExtraArgs:
  - name: "node-ip"
    value: "10.100.0.2,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.

```
apiVersion: kubeadm.k8s.io/v1beta4
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

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.

Best practices

Considerations for large clusters

Running in multiple zones

Validate node setup

Enforcing Pod Security Standards

[PKI certificates and requirements](#)

Considerations for large clusters

A cluster is a set of nodes (physical or virtual machines) running Kubernetes agents, managed by the control plane. Kubernetes v1.34 supports clusters with up to 5,000 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 5,000 nodes
- No more than 150,000 total pods
- No more than 300,000 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 Vertical Pod Autoscaler 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.

Learn more about Vertical Pod Autoscaler and how you can use it to scale cluster components, including cluster-critical addons.

- Read about Node autoscaling
- The addon resizer helps you in resizing the addons automatically as your cluster's scale changes.

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, Kubernetes 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.

Please note that the method of adding zone labels can depend on your cloud provider and the storage provisioner you're using. Always refer to the specific documentation for your environment to ensure correct configuration.

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.

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:

- CRI-compatible container runtimes such as Docker, containerd and CRI-O
- 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:

    - `--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 \
  registry.k8s.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
  registry.k8s.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
  registry.k8s.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.

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 v1.25 [stable]`

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.

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:

Server certificates

- Server certificate for the API server endpoint
- Server certificate for the etcd server
- Server certificates for each kubelet (every node runs a kubelet)
- Optional server certificate for the front-proxy

Client certificates

- Client certificates for each kubelet, used to authenticate to the API server as a client of the Kubernetes API
- Client certificate for each API server, used to authenticate to etcd
- Client certificate for the controller manager to securely communicate with the API server
- Client certificate for the scheduler to securely communicate with the API server
- Client certificates, one for each node, for kube-proxy to authenticate to the API server
- Optional client certificates for administrators of the cluster to authenticate to the API server
- Optional client certificate for the front-proxy

Kubelet's server and client certificates

To establish a secure connection and authenticate itself to the kubelet, the API Server requires a client certificate and key pair.

In this scenario, there are two approaches for certificate usage:

- Shared Certificates: The kube-apiserver can utilize the same certificate and key pair it uses to authenticate its clients. This means that the existing certificates, such as `apiserver.crt` and `apiserver.key`, can be used for communicating with the kubelet servers.

- Separate Certificates: Alternatively, the kube-apiserver can generate a new client certificate and key pair to authenticate its communication with the kubelet servers. In this case, a distinct certificate named `kubelet-client.crt` and its corresponding private key, `kubelet-client.key` are created.

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`

Default CN	Parent CA	O (in Subject)	kind	hosts (SAN)
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		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	

Note:

Instead of using the super-user group `system:masters` for `kube-apiserver-kubelet-client` a less privileged group can be used. kubeadm uses the `kubeadm:cluster-admins` group for that purpose.

where `kind` maps to one or more of the x509 key usage, which is also documented in the `.spec.usages` of a CertificateSigningRequest type:

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.

DefaultCN	recommendedkeypath	recommendedcertpath	command	keyargument	certargument
etcd-ca	etcd/ca.key	etcd/ca.crt	kube-apiserver		--etcd-cafile

DefaultCN	recommendedkeypath	recommendedcertpath	command	keyargument	certargument
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
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 accounts and service accounts:

Filename	Credential name	Default CN	O (in Subject)
admin.conf	default-admin	kubernetes-admin	`<admin-group>`
super-admin.conf	default-super-admin	kubernetes-super-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.

Note:

In the above example `<admin-group>` is implementation specific. Some tools sign the certificate in the default `admin.conf` to be part of the `system:masters` group. `system:masters` is a break-glass, super user group can bypass the authorization layer of Kubernetes, such as RBAC. Also some tools do not generate a separate `super-admin.conf` with a certificate bound to this super user group.

kubeadm generates two separate administrator certificates in kubeconfig files. One is in `admin.conf` and has `Subject: O = kubeadm:cluster-admins, CN = kubernetes-admin`. `kubeadm:cluster-admins` is a custom group bound to the `cluster-admin` ClusterRole. This file is generated on all kubeadm managed control plane machines.

Another is in `super-admin.conf` that has `Subject: O = system:masters, CN = kubernetes-super-admin`. This file is generated only on the node where `kubeadm init` was called.

1. For each configuration, generate an x509 certificate/key pair with the given Common Name (CN) and Organization (O).

2. Run `kubectl` as follows for each configuration:

   ```
   KUBECONFIG=<filename> kubectl config set-cluster default-cluster --server=https://<host ip>:6443 --certificate-authority <path-to-kubernetes-ca> --embed-certs
   KUBECONFIG=<filename> kubectl config set-credentials <credential-name> --client-key <path-to-key>.pem --client-certificate <path-to-cert>.pem --embed-certs
   KUBECONFIG=<filename> kubectl config set-context default-system --cluster default-cluster --user <credential-name>
   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
super-admin.conf	kubectl	Configures super 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/super-admin.conf
/etc/kubernetes/kubelet.conf
/etc/kubernetes/controller-manager.conf
/etc/kubernetes/scheduler.conf
```

---

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`) ↩