Bootstrapping clusters with kubeadm

1 - Installing kubeadm

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

Before you begin

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

Verify the MAC address and product_uuid are unique for every node

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

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

Check network adapters

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

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
```

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.

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

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`

Installing kubeadm, kubelet and kubectl

You will install these packages on all of your machines:

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

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

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

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

For more information on version skews, see:

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

Debian-based distributions | Red Hat-based distributions

Without a package manager

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

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

2. Download the Google Cloud public signing key:

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

3. Add the Kubernetes `apt` repository:

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

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

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

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

Configuring a cgroup driver

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

> **Warning:**
> Matching the container runtime and kubelet cgroup drivers is required or otherwise the kubelet process will fail.
>
> See Configuring a cgroup driver for more details.

Troubleshooting

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

What's next

- Using kubeadm to Create a Cluster

2 - 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
```

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

```
unset KUBECONFIG
```

Or set it to the default `KUBECONFIG` location:

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

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

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

Kubelet client certificate rotation fails

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

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

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

6. Restart the kubelet.
7. Make sure the node becomes `Ready`.

Default NIC When using flannel as the pod network in Vagrant

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

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

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

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

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

Non-public IP used for containers

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

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

- This may be due to Kubernetes using an IP that can not communicate with other IPs on the seemingly same subnet, possibly by policy of the machine provider.
- DigitalOcean assigns a public IP to `eth0` as well as a private one to be used internally as anchor for their floating IP feature, yet `kubelet` will pick the latter as the node's `InternalIP` instead of the public one.

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

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

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

  Then restart `kubelet`:

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

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

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

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

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

```
kubectl apply -f -
```

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

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

etcd pods restart continually

If you encounter the following error:

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

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

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

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

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

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

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

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

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

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

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

A known workaround is to use the kubeadm configuration file.

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

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

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

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

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

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

The tracking issue for this problem is here.

`/usr` is mounted read-only on nodes

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

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

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

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

On joining Nodes:

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

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

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

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

This issue is fixed as of version 1.19.

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

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

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

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

Cannot use the metrics-server securely in a kubeadm cluster

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

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

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

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

Also see How to run the metrics-server securely.

3 - Creating a cluster with kubeadm

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

The `kubeadm` tool is good if you need:

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

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

Before you begin

To follow this guide, you need:

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

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

Kubernetes' version and version skew support policy applies to `kubeadm` as well as to Kubernetes overall. Check that policy to learn about what versions of Kubernetes and `kubeadm` are supported. This page is written for Kubernetes v1.25.

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

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, run `apt-get update && apt-get upgrade` or `yum update` to get the latest version of kubeadm.

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

Preparing the required container images

This step is optional and only applies in case you wish `kubeadm init` and `kubeadm join` to not download the default container images which are hosted at `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.
4. (Optional) Unless otherwise specified, `kubeadm` uses the network interface associated with the default gateway to set the advertise address for this particular control-plane node's API server. To use a different network interface, specify the `--apiserver-advertise-address=<ip-address>` argument to `kubeadm init`. To deploy an IPv6 Kubernetes cluster using IPv6 addressing, you must specify an IPv6 address, for example `--apiserver-advertise-address=2001:db8::101`

To initialize the control-plane node run:

```
kubeadm init <args>
```

Considerations about apiserver-advertise-address and ControlPlaneEndpoint

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

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

Here is an example mapping:

```
192.168.0.102 cluster-endpoint
```

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

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

More information

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

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

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

To 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-toke
```

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

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

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

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

Warning: Kubeadm signs the certificate in the `admin.conf` to have `Subject: O = system:masters, CN = kubernetes-admin`. `system:masters` is a break-glass, super user group that bypasses the authorization layer (e.g. RBAC). Do not share the `admin.conf` file with anyone and instead grant users custom permissions by generating them a kubeconfig file using the `kubeadm kubeconfig user` command. For more details see Generating kubeconfig files for additional users.

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

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

Installing a Pod network add-on

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

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

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

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

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

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

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

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

You can install only one Pod network per cluster.

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

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

Managed node labels

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

Control plane node isolation

By default, your cluster will not schedule Pods on the control plane 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.

Joining your nodes

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

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

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

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

```
kubeadm token list
```

The output is similar to this:

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

```
                                                'kubeadm init'.
```

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

```
kubeadm token create
```

The output is similar to this:

```
5didvk.d09sbcov8ph2amjw
```

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

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

The output is similar to:

```
8cb2de97839780a412b93877f8507ad6c94f73add17d5d7058e91741c9d5ec78
```

> **Note:** To specify an IPv6 tuple for `<control-plane-host>:<control-plane-port>`, IPv6 address must be enclosed in square brackets, for example: `[2001:db8::101]:2073`.

The output should look something like:

```
[preflight] Running pre-flight checks

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

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

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

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

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

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

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

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

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

(Optional) Proxying API Server to localhost

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

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

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

Clean up

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

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

Remove the node

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

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

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

```
kubeadm reset
```

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

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

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

```
ipvsadm -C
```

Now remove the node:

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

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

Clean up the control plane

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

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

What's next

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

Feedback

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

Version skew policy

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 1.25

- `kubernetesVersion` must be at 1.25 or 1.24

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 one version older.

Example:

- kubeadm is at 1.25
- kubelet on the host must be at 1.25 or 1.24

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 1.25 was used to create a cluster with `kubeadm init`
- Joining nodes must use a kubeadm binary that is at version 1.25

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 1.24 was used to create or upgrade the node
- The version of kubeadm used for upgrading the node must be at 1.24 or 1.25

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.

4 - Customizing components with the kubeadm API

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

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

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

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 `key: value` pairs. To override a flag for a control plane component:

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

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

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

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

APIServer flags

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

Example usage:

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
kubernetesVersion: v1.16.0
apiServer:
  extraArgs:
    anonymous-auth: "false"
    enable-admission-plugins: AlwaysPullImages,DefaultStorageClass
    audit-log-path: /home/johndoe/audit.log
```

ControllerManager flags

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

Example usage:

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

Scheduler flags

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

Example usage:

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

Etcd flags

For details, see the etcd server documentation.

Example usage:

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
etcd:
  local:
    extraArgs:
      election-timeout: 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/v1beta3
kind: InitConfiguration
patches:
  directory: /home/user/somedir
```

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

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

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

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

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

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

Customizing the kubelet

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

5 - Options for Highly Available Topology

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

You can set up an HA cluster:

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

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

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

Stacked etcd topology

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

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

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

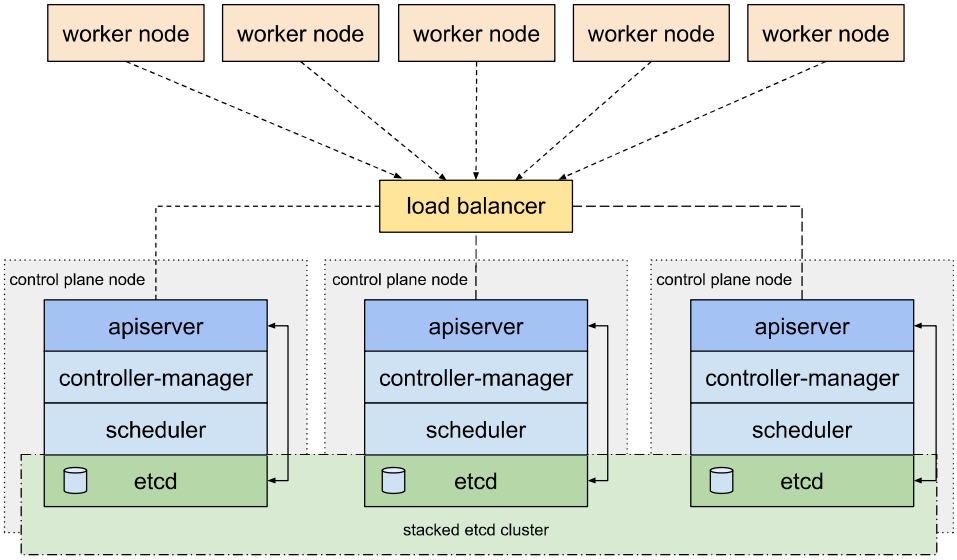
This topology couples the control planes and etcd members on the same nodes. It is simpler to set up than a cluster with external etcd nodes, and simpler to manage for replication.

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

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

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

kubeadm HA topology - stacked etcd



External etcd topology

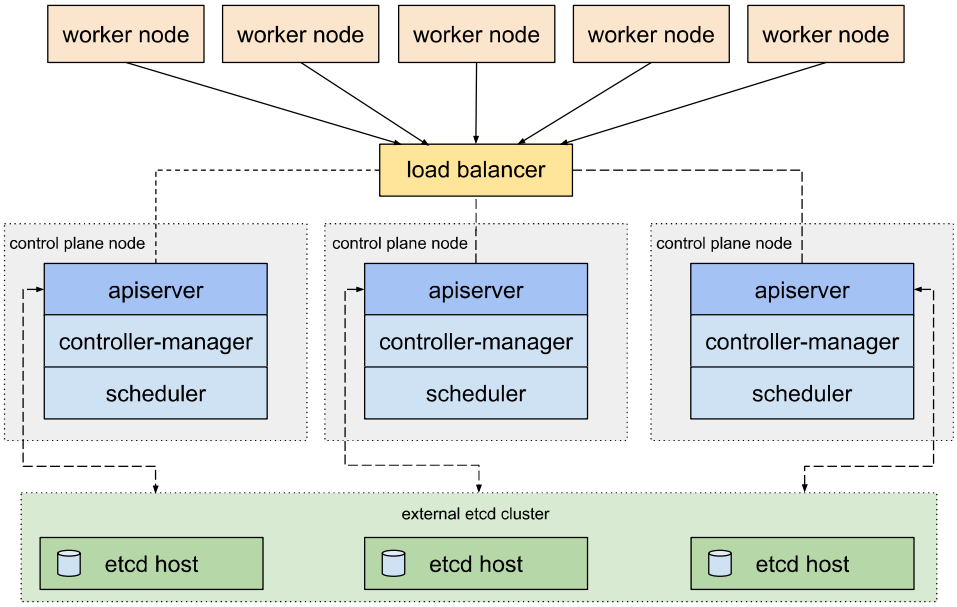
An HA cluster with external etcd is a topology where the distributed data storage cluster provided by etcd is external to the cluster formed by the nodes that run control plane components.

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.

kubeadm HA topology - external etcd



What's next

- Set up a highly available cluster with kubeadm

6 - Creating Highly Available Clusters with kubeadm

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

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

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

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

See also the upgrade documentation.

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

Before you begin

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

Stacked etcd | External etcd

You need:

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

See Stacked etcd topology for context.

Container images

Each host should have access read and fetch images from the Kubernetes container image registry, `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 -v <LOAD_BALANCER_IP> <PORT>
```

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

3. Add the remaining control plane nodes to the load balancer target group.

Stacked control plane and etcd nodes

Steps for the first control plane node

1. Initialize the control plane:

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

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

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

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

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

The output looks similar to:

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

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

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

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

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

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

  ```
  kubeadm certs certificate-key
  ```

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

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

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

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

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

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

Steps for the rest of the control plane nodes

For each additional control plane node you should:

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

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

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

You can join multiple control-plane nodes in parallel.

External etcd nodes

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

Set up the etcd cluster

1. Follow these instructions to set up the etcd cluster.
2. 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/v1beta3
kind: ClusterConfiguration
kubernetesVersion: stable
controlPlaneEndpoint: "LOAD_BALANCER_DNS:LOAD_BALANCER_PORT" # change this (see b
etcd:
  external:
    endpoints:
```

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

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

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

The following steps are similar to the stacked etcd setup:

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

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

Steps for the rest of the control plane nodes

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

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

Common tasks after bootstrapping control plane

Install workers

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

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

Manual certificate distribution

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

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

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

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

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

2. Add your SSH identity to the session:

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

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

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

     ```
     ssh -A 10.0.0.7
     ```

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

     ```
     sudo -E -s
     ```

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

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

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

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

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

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

7 - Set up a High Availability etcd Cluster with kubeadm

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

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

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

Before you begin

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

Setting up the cluster

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

> **Note:** kubeadm contains all the necessary 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/20-etcd-service-manager.conf
[Service]
ExecStart=
# Replace "systemd" with the cgroup driver of your container runtime. The default
```

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

systemctl daemon-reload
systemctl restart kubelet
```

Check the kubelet status to ensure it is running.

```
systemctl status kubelet
```

2. Create configuration files for kubeadm.

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

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

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

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

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

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

3. Generate the certificate authority

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

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

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

   This creates two files

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

4. Create certificates for each member

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

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

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

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

5. Copy certificates and kubeadm configs

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

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

6. Ensure all expected files exist

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

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

On `$HOST1`:

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

On `$HOST2`

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

7. Create the static pod manifests

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

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

8. Optional: Check the cluster health

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

```
-v /etc/kubernetes:/etc/kubernetes registry.k8s.io/etcd:${ETCD_TAG} etcdctl \
--cert /etc/kubernetes/pki/etcd/peer.crt \
--key /etc/kubernetes/pki/etcd/peer.key \
--cacert /etc/kubernetes/pki/etcd/ca.crt \
--endpoints https://${HOST0}:2379 endpoint health --cluster
...
https://[HOST0 IP]:2379 is healthy: successfully committed proposal: took = 16.28
https://[HOST1 IP]:2379 is healthy: successfully committed proposal: took = 19.44
https://[HOST2 IP]:2379 is healthy: successfully committed proposal: took = 35.92
```

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

What's next

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

8 - 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. A kubelet configuration file is also written to `/etc/kubernetes/kubelet.conf` with the baseline cluster-wide configuration for all kubelets in the cluster. This configuration file points to the client certificates that allow the kubelet to communicate with the API server. This addresses the need to propagate cluster-level configuration to each kubelet.

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

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

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

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

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

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

Workflow when using `kubeadm join`

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

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

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

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

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

The kubelet drop-in file for systemd

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

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

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

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

- The KubeConfig file to use for the TLS Bootstrap is `/etc/kubernetes/bootstrap-kubelet.conf`, but it is only used if `/etc/kubernetes/kubelet.conf` does not exist.
- The KubeConfig file with the unique kubelet identity is `/etc/kubernetes/kubelet.conf`.
- The file containing the kubelet's ComponentConfig is `/var/lib/kubelet/config.yaml`.

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

Kubernetes binaries and package contents

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

Package name	Description
`kubeadm`	Installs the `/usr/bin/kubeadm` CLI tool and the kubelet drop-in file for the kubelet.
`kubelet`	Installs the `/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.

9 - Dual-stack support with kubeadm

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

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

Before you begin

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

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

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

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

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

Create a dual-stack cluster

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

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

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

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

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

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

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

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

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

Join a node to dual-stack cluster

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

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

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

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

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

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

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

Create a single-stack cluster

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

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

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

What's next

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