
Production environment

Create a production-quality Kubernetes cluster

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

Production considerations

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

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

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

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

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

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

Production cluster setup

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

Production control plane

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

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

- *Choose deployment tools:* You can deploy a control plane using tools such as kubeadm, kops, and kubespray. See [Installing Kubernetes with deployment tools](#) to learn tips for production-quality deployments using each of those deployment methods. Different [Container Runtimes](#) are available to use with your deployments.
- *Manage certificates:* Secure communications between control plane services are implemented using certificates. Certificates are automatically generated during deployment or you can generate them using your own certificate authority. See [PKI certificates and requirements](#) for details.
- *Configure load balancer for apiserver:* Configure a load balancer to distribute external API requests to the apiserver service instances running on different nodes. See [Create an External Load Balancer](#) for details.
- *Separate and backup etcd service:* The etcd services can either run on the same machines as other control plane services or run on separate machines, for extra security and availability. Because etcd stores cluster configuration data, backing up the etcd database should be done regularly to ensure that you can repair that database if needed. See the [etcd FAQ](#) for details on configuring and using etcd. See [Operating etcd clusters for Kubernetes](#) and [Set up a High Availability etcd cluster with kubeadm](#) for details.
- *Create multiple control plane systems:* For high availability, the control plane should not be limited to a single machine. If the control plane services are run by an init service (such as systemd), each service should run on at least three machines. However, running control plane services as pods in Kubernetes ensures that the replicated number of services that you request will always be available. The scheduler should be fault tolerant, but not

highly available. Some deployment tools set up [Raft](#) consensus algorithm to do leader election of Kubernetes services. If the primary goes away, another service elects itself and take over.

- **Span multiple zones:** If keeping your cluster available at all times is critical, consider creating a cluster that runs across multiple data centers, referred to as zones in cloud environments. Groups of zones are referred to as regions. By spreading a cluster across multiple zones in the same region, it can improve the chances that your cluster will continue to function even if one zone becomes unavailable. See [Running in multiple zones](#) for details.
- **Manage on-going features:** If you plan to keep your cluster over time, there are tasks you need to do to maintain its health and security. For example, if you installed with kubeadm, there are instructions to help you with [Certificate Management](#) and [Upgrading kubeadm clusters](#). See [Administer a Cluster](#) for a longer list of Kubernetes administrative tasks.

To learn about available options when you run control plane services, see [kube-apiserver](#), [kube-controller-manager](#), and [kube-scheduler](#) component pages. For highly available control plane examples, see [Options for Highly Available topology](#), [Creating Highly Available clusters with kubeadm](#), and [Operating etcd clusters for Kubernetes](#). See [Backing up an etcd cluster](#) for information on making an etcd backup plan.

Production worker nodes

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

- **Configure nodes:** Nodes can be physical or virtual machines. If you want to create and manage your own nodes, you can install a supported operating system, then add and run the appropriate [Node services](#). Consider:
 - The demands of your workloads when you set up nodes by having appropriate memory, CPU, and disk speed and storage capacity available.
 - Whether generic computer systems will do or you have workloads that need GPU processors, Windows nodes, or VM isolation.
- **Validate nodes:** See [Valid node setup](#) for information on how to ensure that a node meets the requirements to join a Kubernetes cluster.
- **Add nodes to the cluster:** If you are managing your own cluster you can add nodes by setting up your own machines and either adding them manually or having them register themselves to the cluster's apiserver. See the [Nodes](#) section for information on how to set up Kubernetes to add nodes in these ways.
- **Scale nodes:** Have a plan for expanding the capacity your cluster will eventually need. See [Considerations for large clusters](#) to help determine how many nodes you need, based on the number of pods and containers you need to run. If you are managing nodes yourself, this can mean purchasing and installing your own physical equipment.
- **Autoscale nodes:** Read [Node Autoscaling](#) to learn about the tools available to automatically manage your nodes and the capacity they provide.
- **Set up node health checks:** For important workloads, you want to make sure that the nodes and pods running on those nodes are healthy. Using the [Node Problem Detector](#) daemon, you can ensure your nodes are healthy.

Production user management

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

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

- **Authentication:** The apiserver can authenticate users using client certificates, bearer tokens, an authenticating proxy, or HTTP basic auth. You can choose which authentication methods you want to use. Using plugins, the apiserver can leverage your organization's existing authentication methods, such as LDAP or Kerberos. See [Authentication](#) for a description of these different methods of authenticating Kubernetes users.
- **Authorization:** When you set out to authorize your regular users, you will probably choose between RBAC and ABAC authorization. See [Authorization Overview](#) to review different modes for authorizing user accounts (as well as service account access to your cluster):
 - **Role-based access control (RBAC):** Lets you assign access to your cluster by allowing specific sets of permissions to authenticated users. Permissions can be assigned for a specific namespace (Role) or across the entire cluster (ClusterRole). Then using RoleBindings and ClusterRoleBindings, those permissions can be attached to particular users.
 - **Attribute-based access control (ABAC):** Lets you create policies based on resource attributes in the cluster and will allow or deny access based on those attributes. Each line of a policy file identifies versioning properties (apiVersion and kind) and a map of spec properties to match the subject (user or group), resource property, non-resource property (/version or /apis), and readonly. See [Examples](#) for details.

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

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

Set limits on workload resources

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

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

What's next

- Decide if you want to build your own production Kubernetes or obtain one from available [Turnkey Cloud Solutions](#) or [Kubernetes Partners](#).
- If you choose to build your own cluster, plan how you want to handle [certificates](#) and set up high availability for features such as [etcd](#) and the [API server](#).
- Choose from [kubeadm](#), [kops](#) or [Kubespray](#) deployment methods.
- Configure user management by determining your [Authentication](#) and [Authorization](#) methods.
- Prepare for application workloads by setting up [resource limits](#), [DNS autoscaling](#) and [service accounts](#).

Dual-stack support with kubeadm

FEATURE STATE: Kubernetes v1.23 [stable]

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

Before you begin

You need to have installed the [kubeadm](#) tool, following the steps from [Installing kubeadm](#).

For each server that you want to use as a [node](#), make sure it allows IPv6 forwarding.

Enable IPv6 packet forwarding

To check if IPv6 packet forwarding is enabled:

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

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

To manually enable IPv6 packet forwarding:

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

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

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

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

Note:

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

Create a dual-stack cluster

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

```
# These address ranges are examples
kubeadm init --pod-network-cidr=10.244.0.0/16,2001:db8:42:0::/56 --service-cidr=10.96.0.0/16,2001:db8:42:1::/112
```

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

```
---
apiVersion: kubeadm.k8s.io/v1beta4kind: ClusterConfigurationnetworking: podSubnet: 10.244.0.0/16,2001:db8:42:0::/56 serviceSubnet: 10.96.0.0/16,2001:db8:42:1::/112
advertiseAddress in InitConfiguration specifies the IP address that the API Server will advertise it is listening on. The value of advertiseAddress equals the --apiserver-advertise-address flag of kubeadm init.
```

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

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

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

Note:

The `--apiserver-advertise-address` flag does not support dual-stack.

Join a node to dual-stack cluster

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

Here is an example kubeadm [configuration file](#) kubeadm-config.yaml for joining a worker node to the cluster.

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: JoinConfigurationdiscovery: bootstrapToken: apiServerEndpoint: 10.100.0.1:6443 token: "c1vldh.vjjwg16ucnbp94qr" ca:
```

Also, here is an example kubeadm [configuration file](#) kubeadm-config.yaml for joining another control plane node to the cluster.

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: JoinConfigurationcontrolPlane: localAPIEndpoint: advertiseAddress: "10.100.0.2" bindPort: 6443discovery: bootstrapTo:
```

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

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

Create a single-stack cluster

Note:

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

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

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: ClusterConfigurationnetworking: 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](#)

Creating a cluster with kubeadm

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

The kubeadm tool is good if you need:

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

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

Before you begin

To follow this guide, you need:

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

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

[Kubernetes' version and version skew support policy](#), applies to kubeadm as well as to Kubernetes overall. Check that policy to learn about what versions of Kubernetes and kubeadm are supported. This page is written for Kubernetes v1.34.

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

Note:

Any commands under kubeadm alpha are, by definition, supported on an alpha level.

Objectives

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

Instructions

Preparing the hosts

Component installation

Install a [container runtime](#) and kubeadm on all the hosts. For detailed instructions and other prerequisites, see [Installing kubeadm](#).

Note:

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

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

Network setup

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

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

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

Note:

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

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

Note:

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

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

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

For dual-stack see [Dual-stack support with kubeadm](#).

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

Warning:

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

Preparing the required container images

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

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

Kubeadm allows you to use a custom image repository for the required images. See [Using custom images](#) for more details.

Initializing your control-plane node

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

1. (Recommended) If you have plans to upgrade this single control-plane kubeadm cluster to [high availability](#) you should specify the `--control-plane-endpoint` to set the shared endpoint for all control-plane nodes. Such an endpoint can be either a DNS name or an IP address of a load-balancer.
2. Choose a Pod network add-on, and verify whether it requires any arguments to be passed to `kubeadm init`. Depending on which third-party provider you choose, you might need to set the `--pod-network-cidr` to a provider-specific value. See [Installing a Pod network add-on](#).
3. (Optional) kubeadm tries to detect the container runtime by using a list of well known endpoints. To use different container runtime or if there are more than one installed on the provisioned node, specify the `--cri-socket` argument to kubeadm. See [Installing a runtime](#).

To initialize the control-plane node run:

```
kubeadm init <args>
```

Considerations about apiserver-advertise-address and ControlPlaneEndpoint

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

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

Here is an example mapping:

```
192.168.0.102 cluster-endpoint
```

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

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

More information

For more information about `kubeadm init` arguments, see the [kubeadm reference guide](#).

To configure `kubeadm init` with a configuration file see [Using kubeadm init with a configuration file](#).

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

To reconfigure a cluster that has already been created see [Reconfiguring a kubeadm cluster](#).

To run `kubeadm init` again, you must first [tear down the cluster](#).

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

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

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

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

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

You should now deploy a Pod network to the cluster.

Run `"kubectl apply -f [podnetwork].yaml"` with one of the options listed at: [/docs/concepts/cluster-administration/addons/](#)

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

```
kubeadm join <control-plane-host>:<control-plane-port> --token <token> --discovery-token-ca-cert-hash sha256:<hash>
```

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

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

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

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

Warning:

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

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

See [Generating kubeconfig files for additional users](#) on how to use `kubeadm kubeconfig user` to generate kubeconfig files for additional users.

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

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

Installing a Pod network add-on

Caution:

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

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

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

Note:

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

Several external projects provide Kubernetes Pod networks using CNI, some of which also support [Network Policy](#).

See a list of add-ons that implement the [Kubernetes networking model](#).

Please refer to the [Installing Addons](#) page for a non-exhaustive list of networking addons supported by Kubernetes. You can install a Pod network add-on with the following command on the control-plane node or a node that has the kubeconfig credentials:

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

Note:

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

You can install only one Pod network per cluster.

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

If your network is not working or CoreDNS is not in the Running state, check out the [troubleshooting guide](#) for kubeadm.

Managed node labels

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

Control plane node isolation

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

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

The output will look something like:

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

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

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

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

Adding more control plane nodes

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

Adding worker nodes

The worker nodes are where your workloads run.

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

- [Adding Linux worker nodes](#)
- [Adding Windows worker nodes](#)

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

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

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

Note:

The example above assumes SSH access is enabled for root. If that is not the case, you can copy the `admin.conf` file to be accessible by some other user and `scp` using that other user instead.

The `admin.conf` file gives the user *superuser* privileges over the cluster. This file should be used sparingly. For normal users, it's recommended to generate an unique credential to which you grant privileges. You can do this with the `kubeadm kubeconfig user --client-name <CN>` command. That command will print out a KubeConfig file to STDOUT which you should save to a file and distribute to your user. After that, grant privileges by using `kubectl create (cluster)rolebinding`.

(Optional) Proxying API Server to localhost

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

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

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

Clean up

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

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

Remove the node

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

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

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

```
kubeadm reset
```

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

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

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

```
ipvsadm -C
```

Now remove the node:

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

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

Clean up the control plane

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

See the [kubeadm reset](#) reference documentation for more information about this subcommand and its options.

Version skew policy

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

kubeadm's skew against the Kubernetes version

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

Example:

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

kubeadm's skew against the kubelet

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

Example:

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

kubeadm's skew against kubeadm

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

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

Example for `kubeadm join`:

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

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

Example for `kubeadm upgrade`:

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

To learn more about the version skew between the different Kubernetes component see the [Version Skew Policy](#).

Limitations

Cluster resilience

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

Workarounds:

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

Platform compatibility

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

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

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

Troubleshooting

If you are running into difficulties with kubeadm, please consult our [troubleshooting docs](#).

What's next

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

Feedback

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

Customizing components with the kubeadm API

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

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

Note:

To reconfigure a cluster that has already been created see [Reconfiguring a kubeadm cluster](#).

Customizing the control plane with flags in `ClusterConfiguration`

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

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

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

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

Note:

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

Note:

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

Note:

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

APIServer flags

For details, see the [reference documentation for kube-apiserver](#).

Example usage:

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: ClusterConfigurationkubernetesVersion: v1.16.0apiServer:  extraArgs:  - name: "enable-admission-plugins"    value: "AlwaysPu:
```

ControllerManager flags

For details, see the [reference documentation for kube-controller-manager](#).

Example usage:

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: ClusterConfigurationkubernetesVersion: v1.16.0controllerManager:  extraArgs:  - name: "cluster-signing-key-file"    value: ",
```

Scheduler flags

For details, see the [reference documentation for kube-scheduler](#).

Example usage:

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: ClusterConfigurationkubernetesVersion: v1.16.0scheduler:  extraArgs:  - name: "config"    value: "/etc/kubernetes/scheduler-
```

Etd flags

For details, see the [etcd server documentation](#).

Example usage:

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: ClusterConfigurationetcd:  local:    extraArgs:  - name: "election-timeout"    value: 1000
```

Customizing with patches

FEATURE STATE: Kubernetes v1.22 [beta]

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

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

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: InitConfigurationpatches: directory: /home/user/somedir
```

Note:

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

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

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: JoinConfigurationpatches: directory: /home/user/somedir
```

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

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

Note:

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

Customizing the kubelet

To customize the kubelet you can add a [KubeletConfiguration](#) next to the `ClusterConfiguration` or `InitConfiguration` separated by `---` within the same configuration file. This file can then be passed to `kubeadm init` and `kubeadm` will apply the same base `KubeletConfiguration` to all nodes in the cluster.

For applying instance-specific configuration over the base `KubeletConfiguration` you can use the [kubeletconfiguration patch target](#).

Alternatively, you can use kubelet flags as overrides by passing them in the `nodeRegistration.kubeletExtraArgs` field supported by both `InitConfiguration` and `JoinConfiguration`. Some kubelet flags are deprecated, so check their status in the [kubelet reference documentation](#) before using them.

For additional details see [Configuring each kubelet in your cluster using kubeadm](#)

Customizing kube-proxy

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

For more details you can navigate to our [API reference pages](#).

Note:

`kubeadm` deploys kube-proxy as a [DaemonSet](#), which means that the `KubeProxyConfiguration` would apply to all instances of kube-proxy in the cluster.

Customizing CoreDNS

`kubeadm` allows you to customize the CoreDNS Deployment with patches against the [corednsdeployment patch target](#).

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

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

```
dns:
  disabled: true
```

Also, by executing the following command:

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

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

Installing kubeadm

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

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

- [Installing kubeadm \(Kubernetes v1.33\)](#)
- [Installing kubeadm \(Kubernetes v1.32\)](#)
- [Installing kubeadm \(Kubernetes v1.31\)](#)
- [Installing kubeadm \(Kubernetes v1.30\)](#)



Before you begin

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

Note:

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

Check your OS version

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

- [Linux](#)
- [Windows](#)
- The kubeadm project supports LTS kernels. See [List of LTS kernels](#).
- You can get the kernel version using the command `uname -r`

For more information, see [Linux Kernel Requirements](#).

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

For more information, see [Windows OS version compatibility](#).

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

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

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

Verify the MAC address and product_uuid are unique for every node

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

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

Check network adapters

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

Check required ports

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

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

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

Swap configuration

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

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

Installing a container runtime

To run containers in Pods, Kubernetes uses a [container runtime](#).

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

If you don't specify a runtime, kubeadm automatically tries to detect an installed container runtime by scanning through a list of known endpoints.

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

See [container runtimes](#) for more information.

Note:

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

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

- [Linux](#)
- [Windows](#)

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

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

Installing kubeadm, kubelet and kubect1

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.
- `kubect1`: the command line util to talk to your cluster.

kubeadm **will not** install or manage kubelet or kubect1 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 kubect1, see [Install and set up kubect1](#).

Warning:

These instructions exclude all Kubernetes packages from any system upgrades. This is because kubeadm and Kubernetes require [special attention to upgrade](#).

For more information on version skews, see:

- Kubernetes [version and version-skew policy](#).
- Kubeadm-specific [version skew policy](#).

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

Note:

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

- [Debian-based distributions](#)
- [Red Hat-based distributions](#)
- [Without a package manager](#)

These instructions are for Kubernetes v1.34.

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

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

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

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

Note:

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

3. Add the appropriate Kubernetes apt repository. Please note that this repository have packages only for Kubernetes 1.34; for other Kubernetes minor versions, you need to change the Kubernetes minor version in the URL to match your desired minor version (you should also check that you are reading the documentation for the version of Kubernetes that you plan to install).

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

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

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

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

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

1. Set SELinux to permissive mode:

These instructions are for Kubernetes 1.34.

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

Caution:

- Setting SELinux in permissive mode by running `setenforce 0` and `sed ...` effectively disables it. This is required to allow containers to access the host filesystem; for example, some cluster network plugins require that. You have to do this until SELinux support is improved in the kubelet.
 - You can leave SELinux enabled if you know how to configure it but it may require settings that are not supported by kubeadm.
2. Add the Kubernetes yum repository. The `exclude` parameter in the repository definition ensures that the packages related to Kubernetes are not upgraded upon running `yum update` as there's a special procedure that must be followed for upgrading Kubernetes. Please note that this repository have packages only for Kubernetes 1.34; for other Kubernetes minor versions, you need to change the Kubernetes minor version in the URL to match your desired minor version (you should also check that you are reading the documentation for the version of Kubernetes that you plan to install).

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

3. Install kubelet, kubeadm and kubectl:

For systems with DNF:

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

For systems with DNF5:

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

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

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

Install CNI plugins (required for most pod network):

```
CNI_PLUGINS_VERSION="v1.3.0"
```

```
ARCH="amd64"
DEST="/opt/cni/bin"
sudo mkdir -p "$DEST"
curl -L "https://github.com/containernetworking/plugins/releases/download/${CNI_PLUGINS_VERSION}/cni-plugins-linux-${ARCH}-${CNI_PLUGINS_VERSION}.tar.gz"
```

Define the directory to download command files:

Note:

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

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

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

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

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

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

RELEASE_VERSION="v0.16.2"
curl -sSL "https://raw.githubusercontent.com/kubernetes/release/${RELEASE_VERSION}/cmd/krel/templates/latest/kubelet/kubelet.service"
sudo mkdir -p /usr/lib/systemd/system/kubelet.service.d
curl -sSL "https://raw.githubusercontent.com/kubernetes/release/${RELEASE_VERSION}/cmd/krel/templates/latest/kubeadm/10-kubeadm.conf"
```

Note:

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

Install `kubect1` by following the instructions on [Install Tools page](#).

Optionally, enable the `kubelet` service before running `kubeadm`:

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

Note:

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

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

Configuring a cgroup driver

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

Warning:

Matching the container runtime and `kubelet` cgroup drivers is required or otherwise the `kubelet` process will fail.

See [Configuring a cgroup driver](#) for more details.

Troubleshooting

If you are running into difficulties with `kubeadm`, please consult our [troubleshooting docs](#).

What's next

- [Using kubeadm to Create a Cluster](#)

Creating Highly Available Clusters with kubeadm

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

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

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

If you encounter issues with setting up the HA cluster, please report these in the `kubeadm` [issue tracker](#).

See also the [upgrade documentation](#).

Caution:

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

Before you begin

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

- [Stacked etcd](#)
- [External etcd](#)

You need:

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

See [Stacked etcd topology](#) for context.

You need:

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

And you also need:

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

See [External etcd topology](#) for context.

Container images

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

Command line interface

To manage Kubernetes once your cluster is set up, you should [install kubectl](#) on your PC. It is also useful to install the `kubect1` tool on each control plane node, as this can be helpful for troubleshooting.

First steps for both methods

Create load balancer for kube-apiserver

Note:

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

1. Create a kube-apiserver load balancer with a name that resolves to DNS.
 - In a cloud environment you should place your control plane nodes behind a TCP forwarding load balancer. This load balancer distributes traffic to all healthy control plane nodes in its target list. The health check for an apiserver is a TCP check on the port the kube-apiserver listens on (default value `:6443`).
 - It is not recommended to use an IP address directly in a cloud environment.

- The load balancer must be able to communicate with all control plane nodes on the apiserver port. It must also allow incoming traffic on its listening port.
- Make sure the address of the load balancer always matches the address of kubeadm's ControlPlaneEndpoint.
- Read the [Options for Software Load Balancing](#) guide for more details.

2. Add the first control plane node to the load balancer, and test the connection:

```
nc -zv -w 2 <LOAD_BALANCER_IP> <PORT>
```

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

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

Stacked control plane and etcd nodes

Steps for the first control plane node

1. Initialize the control plane:

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

- You can use the `--kubernetes-version` flag to set the Kubernetes version to use. It is recommended that the versions of kubeadm, kubelet, kubect and Kubernetes match.
- The `--control-plane-endpoint` flag should be set to the address or DNS and port of the load balancer.
- The `--upload-certs` flag is used to upload the certificates that should be shared across all the control-plane instances to the cluster. If instead, you prefer to copy certs across control-plane nodes manually or using automation tools, please remove this flag and refer to [Manual certificate distribution](#) section below.

Note:

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

Note:

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

The output looks similar to:

```
...
You can now join any number of control-plane node by running the following command on each as a root:
  kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discovery-token-ca-cert-hash sha256:7c2e69131a36ae2a042
```

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

Then you can join any number of worker nodes by running the following on each as root:
 kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discovery-token-ca-cert-hash sha256:7c2e69131a36ae2a042

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

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

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

```
kubeadm certs certificate-key
```

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

Note:

The `kubeadm-certs Secret` and the decryption key expire after two hours.

Caution:

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

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

Note:

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

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

```
kubect1 get pod -n kube-system -w
```

Steps for the rest of the control plane nodes

For each additional control plane node you should:

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

```
sudo kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discovery-token-ca-cert-hash sha256:7c2e69131a36ae2a04
```

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

Note:

As the cluster nodes are usually initialized sequentially, the CoreDNS Pods are likely to all run on the first control plane node. To provide higher availability, please rebalance the CoreDNS Pods with `kubect1 -n kube-system rollout restart deployment coredns` after at least one new node is joined.

External etcd nodes

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

Set up the etcd cluster

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

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

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

Set up the first control plane node

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

```
---
apiVersion: kubeadm.k8s.io/v1beta4kind: ClusterConfigurationkubernetesVersion: stablecontrolPlaneEndpoint: "LOAD_BALANCER_DNS
```

Note:

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

- Replace the following variables in the config template with the appropriate values for your cluster:

- `LOAD_BALANCER_DNS`
- `LOAD_BALANCER_PORT`
- `ETCD_0_IP`
- `ETCD_1_IP`
- `ETCD_2_IP`

The following steps are similar to the stacked etcd setup:

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

Note:

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

Steps for the rest of the control plane nodes

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

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

Common tasks after bootstrapping control plane

Install workers

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

```
sudo kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discovery-token-ca-cert-hash sha256:7c2e69131a36ae2a042a3391
```

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
```

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: KubeletConfigurationclusterDNS:- 10.96.0.10
```

For more details on the `KubeletConfiguration` have a look at [this section](#).

Providing instance-specific configuration details

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

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

The recommended way of applying such instance-specific configuration is by using [KubeletConfiguration patches](#).

Configure kubelets using kubeadm

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

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

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

Workflow when using kubeadm init

When you call `kubeadm init`, the kubelet configuration is marshalled to disk at `/var/lib/kubelet/config.yaml`, and also uploaded to a `kubelet-config ConfigMap` in the `kube-system` namespace of the cluster. Additionally, the kubeadm tool detects the CRI socket on the node and writes its details (including the socket path) into a local configuration, `/var/lib/kubelet/instance-config.yaml`. A kubelet configuration file is also written to `/etc/kubernetes/kubelet.conf` with the baseline cluster-wide configuration for all kubelets in the cluster. This configuration file points to the client certificates that allow the kubelet to communicate with the API server. This addresses the need to [propagate cluster-level configuration to each kubelet](#).

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

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

In addition to the flags used when starting the kubelet, the file also contains dynamic parameters such as the cgroup driver.

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

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

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

Workflow when using `kubeadm join`

When you run `kubeadm join`, `kubeadm` uses the Bootstrap Token credential to perform a TLS bootstrap, which fetches the credential needed to download the `kubelet-config` ConfigMap and writes it to `/var/lib/kubelet/config.yaml`. Additionally, the `kubeadm` tool detects the CRI socket on the node and writes its details (including the socket path) into a local configuration, `/var/lib/kubelet/instance-config.yaml`. The dynamic environment file is generated in exactly the same way as `kubeadm init`.

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

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

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

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

The kubelet drop-in file for `systemd`

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

This configuration file installed by the `kubeadm` [package](#) is written to `/usr/lib/systemd/system/kubelet.service.d/10-kubeadm.conf` and is used by `systemd`. It augments the basic [kubelet.service](#).

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

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

Note:

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

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

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

- The KubeConfig file to use for the TLS Bootstrap is `/etc/kubernetes/bootstrap-kubelet.conf`, but it is only used if `/etc/kubernetes/kubelet.conf` does not exist.
- The KubeConfig file with the unique kubelet identity is `/etc/kubernetes/kubelet.conf`.
- The file containing the kubelet's ComponentConfig is `/var/lib/kubelet/config.yaml`.
- The dynamic environment file that contains `KUBELET_KUBEADM_ARGS` is sourced from `/var/lib/kubelet/kubeadm-flags.env`.
- The file that can contain user-specified flag overrides with `KUBELET_EXTRA_ARGS` is sourced from `/etc/default/kubelet` (for DEBs), or `/etc/sysconfig/kubelet` (for RPMs). `KUBELET_EXTRA_ARGS` is last in the flag chain and has the highest priority in the event of conflicting settings.

Kubernetes binaries and package contents

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

Package name	Description
<code>kubeadm</code>	Installs the <code>/usr/bin/kubeadm</code> CLI tool and the kubelet drop-in file for the kubelet.
<code>kubelet</code>	Installs the <code>/usr/bin/kubelet</code> binary.
<code>kubect1</code>	Installs the <code>/usr/bin/kubect1</code> binary.
<code>cri-tools</code>	Installs the <code>/usr/bin/crictl</code> binary from the cri-tools git repository .
<code>kubernetes-cni</code>	Installs the <code>/opt/cni/bin</code> binaries from the plugins git repository .

Installing Kubernetes with deployment tools

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

- [kubeadm](#)

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

Turnkey Cloud Solutions

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