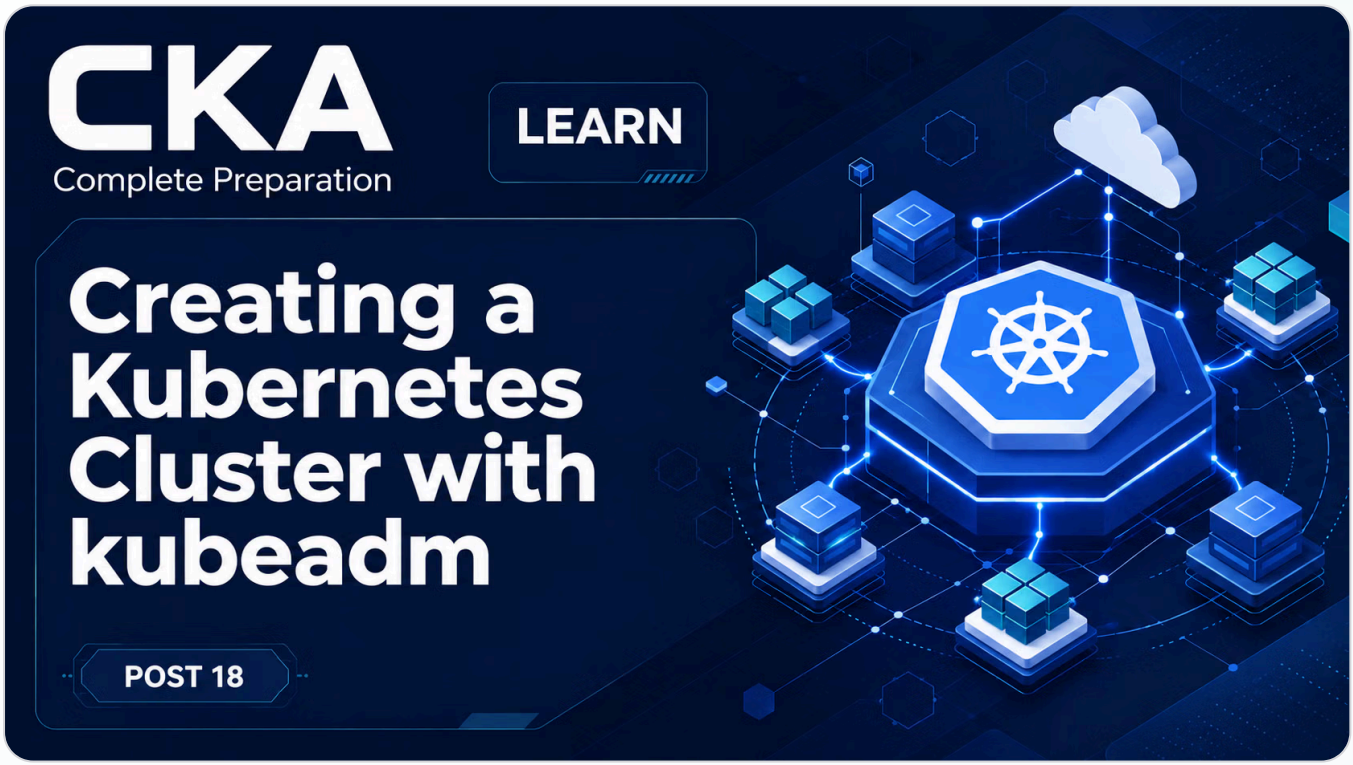
Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 28, 2026

Creating a Kubernetes Cluster with kubeadm

Build a single-control-plane Kubernetes cluster with kubeadm, understand each bootstrap phase, configure kubectl, install Pod networking, and verify the result.



This is Learn post 18 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. The previous lesson prepared Linux, cgroups, packet forwarding, and the container runtime. This lesson uses that ready host to create a working single-control-plane Kubernetes cluster.

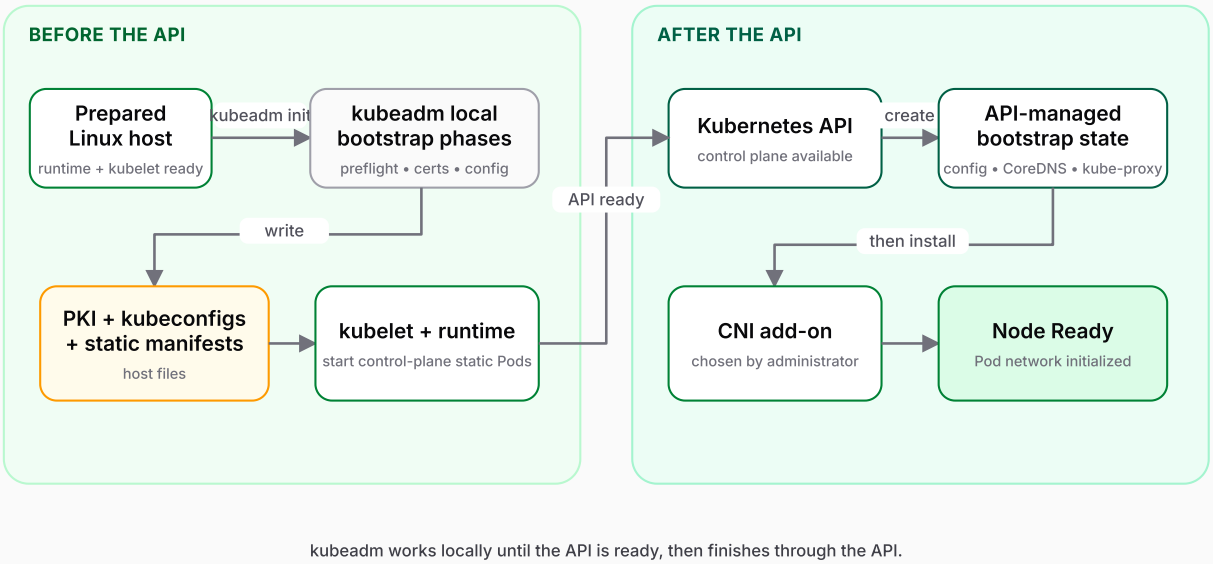
kubeadm is a bootstrap orchestrator. It generates trust material and configuration, asks kubelet to start the control plane as static Pods, and then uses the new API to finish cluster-level setup. It deliberately does not choose a Container Network Interface (CNI) implementation for you.

What you'll learn

- Turn cluster design choices into a small, reviewable kubeadm configuration file.
- Explain the important phases that run behind kubeadm init.
- Configure kubectl with the generated administrator kubeconfig.
- Recognize the expected NotReady and Pending state before Pod networking exists.
- Install a compatible CNI add-on and verify the finished control plane.

The mental model: bootstrap crosses an API boundary

> Before the API server exists, kubeadm must work through files and local host services. After kubelet starts the static control-plane Pods, kubeadm crosses into the new API and creates cluster-level state such as bootstrap configuration and add-ons.



kubeadm works locally until the API is ready, then finishes through the API.

kubeadm bootstrap crosses from the host into the API

kubeadm works through local files first, then uses the new API to finish cluster bootstrap.

This sequence explains why kubeadm can start a control plane before Pod networking exists. The control-plane static Pods use the host network, so kubelet can run them locally. CoreDNS is an ordinary Deployment and needs Pod networking before its Pods can run normally.

Choose the cluster contract before running init

A successful bootstrap begins with a few explicit choices. For this demonstration, cp1 has address 192.0.2.10, the stable API name cp1.example.net resolves to that address, the Pod CIDR is 10.244.0.0/16, and the Service CIDR is 10.96.0.0/12. Replace all example values with routes and names valid in your environment.

- The advertise address identifies this API server instance on cp1.
- The control-plane endpoint is the stable address clients and future nodes use for the cluster API.
- The Pod CIDR must match the selected CNI add-on and must not overlap host or Service networks.
- The CRI socket identifies the container runtime kubeadm and kubelet will use on this host.

A stable control-plane endpoint is useful even in a small cluster because it separates the cluster identity from one node's local API address. High-availability load balancing and additional control-plane nodes belong to post 21; here the name resolves directly to cp1.

```bash
getent hosts cp1.example.net
ip route get 192.0.2.10
sudo crictl info >/dev/null && echo 'CRI reachable'
kubeadm version -o short
```

Resolve and route the endpoint before embedding it into certificates and kubeconfigs. The final two checks reconnect this lesson to post 17: the runtime must answer, and the installed kubeadm version determines which Kubernetes and configuration API versions are supported.

Express the plan as kubeadm configuration

Simple flags are useful for quick labs, but a configuration file is easier to review and reproduce. kubeadm's current configuration API uses separate documents for node-local init settings and cluster-wide settings. Generate a reference first, then keep only the fields you understand.

```bash
kubeadm config print init-defaults > kubeadm-defaults.yaml
less kubeadm-defaults.yaml
```

The generated file is a reference, not a ready-made production configuration. Sensitive token values are placeholders, and defaults can change by release. The focused file below states only the decisions needed for this cluster.

kubeadm-config.yaml · yaml

```yaml
apiVersion: kubeadm.k8s.io/v1beta4
kind: InitConfiguration
localAPIEndpoint:
  advertiseAddress: 192.0.2.10
  bindPort: 6443
nodeRegistration:
  criSocket: unix:///run/containerd/containerd.sock
---
apiVersion: kubeadm.k8s.io/v1beta4
kind: ClusterConfiguration
controlPlaneEndpoint: cp1.example.net:6443
kubernetesVersion: v1.37.0
networking:
  podSubnet: 10.244.0.0/16
  serviceSubnet: 10.96.0.0/12
```

InitConfiguration applies to this first control-plane node: its local API address and runtime socket. ClusterConfiguration describes durable cluster-wide choices. The Kubernetes version is intentionally pinned for reproducibility; set it to a release supported by the kubeadm binary you actually installed.

The example uses kubeadm.k8s.io/v1beta4, which is the supported configuration API in current Kubernetes documentation. For a different installed release, print its defaults and migrate older configuration rather than changing the apiVersion by guesswork.

bash

```bash
kubeadm config validate --config kubeadm-config.yaml
sudo kubeadm config images list --config kubeadm-config.yaml
sudo kubeadm config images pull --config kubeadm-config.yaml
```

Validation catches unknown fields, unsupported API versions, and invalid values before the host is changed. Listing images makes the target visible; pre-pulling them separates registry access from the later bootstrap sequence and makes failures easier to locate.

Initialize the control plane

Run init once after the configuration and host checks pass. The command needs root access because it writes system configuration, creates certificates, and coordinates with kubelet and the runtime.

```bash
sudo kubeadm init --config kubeadm-config.yaml
```

Do not treat the scrolling phase names as noise. They are a progress map. If init stops, the last completed phase narrows the layer to inspect: preflight, image pulling, certificate generation, kubelet startup, control-plane health, or API-level configuration.

```plaintext
[init] Using Kubernetes version: v1.37.0
[preflight] Running pre-flight checks
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[control-plane] Creating static Pod manifest ...
[kubelet-start] Starting the kubelet
[wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods
...
Your Kubernetes control-plane has initialized successfully!
```

This abbreviated output shows phase structure, not every line from a particular release. Successful init also prints instructions for configuring kubectl, installing a Pod network, and joining nodes. Treat the generated join token as a secret; it normally expires after 24 hours. Joining and token management are the subject of post 19.

What kubeadm created

kubeadm first establishes a local trust and configuration layer under /etc/kubernetes. It creates the cluster public key infrastructure (PKI), component kubeconfigs, and static Pod manifests for the API server, controller manager, scheduler, and local etcd.

```bash
sudo ls -1 /etc/kubernetes/manifests
sudo ls -1 /etc/kubernetes/*.conf
```

```plaintext
etcd.yaml
kube-apiserver.yaml
kube-controller-manager.yaml
kube-scheduler.yaml

/etc/kubernetes/admin.conf
/etc/kubernetes/controller-manager.conf
/etc/kubernetes/kubelet.conf
/etc/kubernetes/scheduler.conf
/etc/kubernetes/super-admin.conf
```

Because kubelet watches /etc/kubernetes/manifests, writing those files causes the control-plane containers to start. Once the API answers, kubeadm uploads the reusable cluster and kubelet configuration, labels and taints the control-plane node, establishes bootstrap-token rules, and creates the CoreDNS and kube-proxy add-ons.

The files do not all have the same audience. Component kubeconfigs give each control-plane component its own identity. admin.conf is a powerful administrator credential. super-admin.conf bypasses normal RBAC authorization and should be reserved for emergency recovery, not routine administration.

Configure kubectl for the administrator

kubectl needs a server address, cluster CA, and client identity. kubeadm placed those values in admin.conf. Copy that file into the current non-root user's standard kubeconfig location and transfer ownership to that user.

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

The resulting context normally points to the new cluster using an administrator identity. Protect the file: copying admin.conf to another machine grants powerful access there as well. This is different from the short-lived bootstrap token printed for node joining.

Observe the cluster before installing networking

The API can now answer even though the cluster is not finished. Inspect this intermediate state before changing it; the contrast makes the CNI dependency memorable.

```bash
kubectl get nodes -o wide
kubectl get pods -n kube-system -o wide
```

```plaintext
NAME   STATUS     ROLES         AGE   VERSION
cp1    NotReady   control-plane   2m    v1.37.0

NAME                          READY   STATUS    NODE
coredns-...                   0/1     Pending   <none>
etcd-cp1                      1/1     Running   cp1
kube-apiserver-cp1            1/1     Running   cp1
kube-controller-manager-cp1   1/1     Running   cp1
kube-proxy-...                1/1     Running   cp1
kube-scheduler-cp1            1/1     Running   cp1
```

Names and ages vary, but the relationship is stable. The host-networked static Pods can run and expose the API. The node remains NotReady because its Pod network is not configured, and CoreDNS remains Pending because it is a normal Pod that needs that network. This is expected bootstrap state, not evidence that init failed.

kube-proxy may already be Running, but it does not provide Pod networking. It manages Service forwarding rules. The CNI add-on is responsible for giving Pods network connectivity; the extension interfaces and Kubernetes network model are taught in posts 25 and 27.

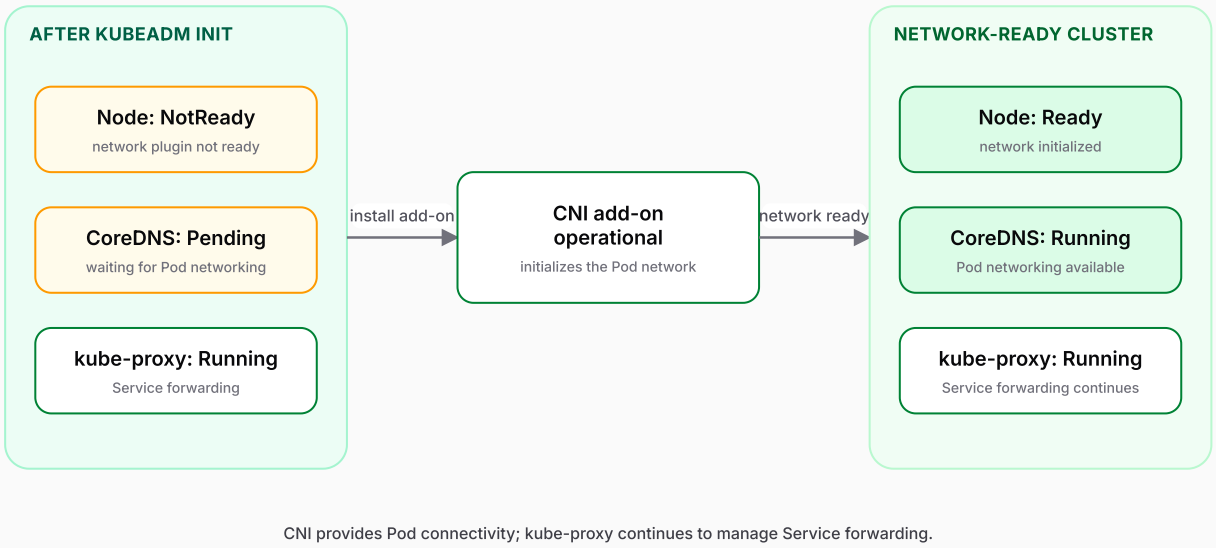
Install one compatible Pod network

Select one CNI-based network add-on that supports your Kubernetes release, architecture, address family, and Pod CIDR. Use the provider's maintained installation instructions; manifest URLs and supported versions change independently of Kubernetes.

```bash
CNI_MANIFEST_URL='<provider-maintained-manifest-url>'
kubectl apply -f "$CNI_MANIFEST_URL"

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

Replace the placeholder with the exact official URL or local manifest for the chosen provider. Do not install two primary CNI add-ons. Watch the provider's node agents and CoreDNS reach Running; then stop the watch with Ctrl-C.



CNI installation completes cluster network readiness

CNI provides Pod connectivity and unlocks Node and CoreDNS readiness; it does not replace kube-proxy.

Verify the finished cluster

Verification should cover four layers: API reachability, API readiness, node readiness, and system workloads. One green command is not a substitute for the others.

```bash
kubectl cluster-info
kubectl get --raw='/readyz?verbose'
kubectl get nodes -o wide
kubectl get pods -n kube-system -o wide
```

plaintext

```
NAME   STATUS   ROLES         AGE   VERSION
cp1    Ready    control-plane   8m    v1.37.0

NAME                          READY   STATUS    NODE
coredns-...                    1/1     Running   cp1
etcd-cp1                       1/1     Running   cp1
kube-apiserver-cp1             1/1     Running   cp1
kube-controller-manager-cp1    1/1     Running   cp1
kube-proxy-...                 1/1     Running   cp1
kube-scheduler-cp1             1/1     Running   cp1
<network-add-on-pods>           ...     Running   cp1
```

readyz checks the API server's internal readiness gates; get nodes confirms kubelet and networking have produced a Ready Node condition; the kube-system listing confirms the control plane and cluster add-ons are running. Provider Pod names and counts vary, so interpret ownership and status rather than memorizing one table.

bash

```
kubectl get node cp1 --show-labels
kubectl describe node cp1 | sed -n '/Taints:/,/Unschedulable:/p'
kubectl -n kube-system get configmap kubeadm-config kubelet-config
```

kubeadm labels the node for its control-plane role and normally adds a NoSchedule taint so ordinary workloads do not land there. The configuration ConfigMaps are the API-side record later kubeadm operations use. A single-node learning cluster can remove the control-plane taint deliberately, but a multi-node cluster should keep control-plane capacity isolated.

optional: single-node lab only · bash

```
kubectl taint nodes cp1 node-role.kubernetes.io/control-plane-
```

Important distinctions

- kubeadm bootstraps Kubernetes; it does not install the container runtime or choose a CNI provider.
- The advertise address belongs to one API server; the control-plane endpoint is the stable cluster-facing address.
- admin.conf authenticates an administrator; a bootstrap token temporarily authenticates a joining node. They are not interchangeable.
- Static control-plane Pods can start through kubelet before CNI; CoreDNS cannot become operational until Pod networking exists.
- kube-proxy implements Service forwarding behavior; a CNI add-on provides Pod network connectivity.

If initialization must be repeated

Do not repeatedly run kubeadm init over a partially initialized host. Read the failure first; logs and generated files may contain the evidence you need. When you intentionally abandon that cluster attempt, kubeadm reset performs a best-effort reversal of kubeadm-managed state.

```bash
sudo kubeadm reset --cri-socket unix:///run/containerd/containerd.sock
```

Reset is not a complete machine scrub. It does not remove CNI configuration, kube-proxy traffic rules, or the user's $HOME/.kube configuration. Review the official reset documentation and remove only the leftovers appropriate for the host's next use. Cluster lifecycle and upgrades are covered in post 20.

What to remember

- kubeadm init moves from local files and host services to API-managed cluster state.

- A small validated configuration makes the API endpoint, runtime socket, version, and network ranges explicit.
- Static Pod manifests let kubelet start the control plane before the cluster network exists.
- NotReady and Pending are expected before CNI; Ready nodes and Running CoreDNS Pods prove the network transition completed.
- Protect admin.conf and bootstrap tokens according to the authority each grants.

Official references

Creating a cluster with kubeadm documents the supported bootstrap and Pod-network workflow.

kubeadm init lists the init phases, flags, generated files, and add-ons.

kubeadm implementation details explains PKI, kubeconfigs, static Pods, bootstrap tokens, and TLS bootstrap.

kubeadm config documents printing defaults, validating configuration, and listing or pulling images.

kubeadm Configuration v1beta4 defines InitConfiguration, ClusterConfiguration, and their fields.

Troubleshooting kubeadm confirms why CoreDNS remains Pending before a Pod network is installed.

kubeadm reset describes the best-effort reset and the state it intentionally leaves behind.