Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 27, 2026

Preparing Linux and the Container Runtime for a kubeadm Cluster

Prepare a Linux host for kubeadm by verifying identity and connectivity, disabling swap, enabling packet forwarding, aligning cgroups, and validating a CRI-compatible runtime.

This is Learn post 17 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. The previous lesson separated kubelet, the container runtime, and kube-proxy. Now you will prepare the Linux foundation those node components depend on before kubeadm creates a cluster.

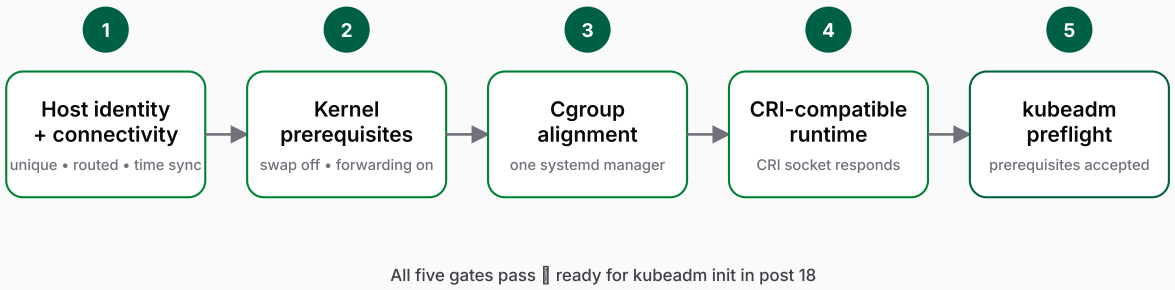
kubeadm automates Kubernetes bootstrap; it does not turn an arbitrary machine into a healthy node. A unique host identity, working routes, suitable kernel settings, one consistent cgroup model, and a reachable Container Runtime Interface (CRI) endpoint must already exist. Preparing those contracts first makes the bootstrap in the next lesson predictable.

What you'll learn

- Verify host identity, resources, name resolution, routes, and time synchronization.
- Prepare swap and IPv4 forwarding for the normal kubeadm path.
- Explain why kubelet and the container runtime must use a compatible cgroup configuration.
- Configure and verify containerd as a CRI-compatible runtime.
- Run kubeadm's preflight phase without creating a control plane.

The mental model: pass five readiness gates

> A kubeadm host is ready when Linux can identify and connect the machine, the kernel permits the required behavior, resource ownership is consistent, the CRI runtime answers, and kubeadm's preflight checks accept the result.



Five readiness gates before kubeadm init

Host readiness is a chain: each Linux and runtime contract must hold before cluster bootstrap begins.

Each gate protects the next one. For example, a running containerd service is not useful to kubelet if its CRI plugin is disabled, and a healthy runtime cannot compensate for duplicate node identity. Readiness is the agreement between layers, not the presence of one installed package.

Gate 1: establish a trustworthy Linux host

Start with facts rather than assumptions. kubeadm expects a compatible Linux host; a control-plane node should have at least two CPUs and a machine should have at least 2 GiB of RAM. Every node also needs a unique hostname, MAC address, and product UUID so Kubernetes does not confuse two machines.

```bash
uname -r
. /etc/os-release && printf '%s %s\n' "$NAME" "$VERSION_ID"
hostnamectl --static
nproc
free -h
ip -br link
sudo cat /sys/class/dmi/id/product_uuid
```

```plaintext
6.8.0-79-generic
Ubuntu 24.04
cp1
2
               total        used        free
Mem:           3.8Gi       620Mi       2.7Gi
lo             UNKNOWN        00:00:00:00:00:00
ens3           UP             52:54:00:12:34:56
4d3f7e9a-8c60-4ab7-93a0-81a6634a908e
```

Repeat the identity checks on every host. The exact kernel and distribution versions may differ, but the names, non-loopback MAC addresses, and product UUIDs must not collide. If a virtual-machine template copied an identity, correct it before Kubernetes records the node.

Confirm name resolution, routes, and time

Cluster nodes need full network connectivity. Verify that the names and addresses you intend to use resolve consistently, that Linux chooses the expected interface and gateway, and that the system clock is synchronized. Replace the example address with the planned address of your control-plane node.

```bash
getent hosts cp1 worker1
ip route
ip route get 192.0.2.10
timedatectl show --property=NTPSynchronized --value
```

```plaintext
192.0.2.10      cp1
192.0.2.21      worker1
default via 192.0.2.1 dev ens3 proto dhcp src 192.0.2.10
192.0.2.10 dev ens3 src 192.0.2.10 uid 1000
    cache local
yes
```

The route result shows which source address Kubernetes traffic will use. Also choose Pod and Service CIDR ranges that do not overlap the host network. Their detailed behavior belongs to the later networking lessons; at this stage, reserve non-overlapping ranges and preserve node-to-node reachability.

Plan firewall access deliberately

A kubeadm cluster uses known control-plane and node ports. Permit them only between the systems that require them; do not expose the full range indiscriminately to the internet. A chosen network add-on can require additional ports later.

```plaintext
Control-plane inbound TCP
  6443          Kubernetes API server
  2379-2380     etcd client and peer traffic
  10250         kubelet API
  10257         kube-controller-manager
  10259         kube-scheduler

Worker inbound
  10250/TCP     kubelet API
  10256/TCP     kube-proxy health endpoint
  30000-32767   default NodePort range (TCP and UDP)
```

Use sudo ss -lntup to inspect listeners on a host. Before the API server exists, a connection refusal on port 6443 can merely mean that no process is listening yet; it does not by itself prove a firewall problem. Verify paths again after the relevant component starts.

Gate 2: prepare kernel behavior

Disable swap for the standard kubeadm path

By default, kubelet fails to start when it detects swap. Kubernetes can be configured to tolerate swap, but that is a deliberate resource-management choice. For a conventional kubeadm cluster, disable active swap and remove its persistent activation source.

bash

```
swapon --show
grep -nE '\sswap\s' /etc/fstab
systemctl list-units --type=swap --all

sudo swapoff -a
swapon --show
```

The final command should print no active swap devices. swapoff changes the running system only, so edit the actual persistent source—commonly an /etc/fstab entry or a systemd swap unit—then verify again after a reboot. Do not delete an unfamiliar storage entry without first identifying how the host provisions swap.

Enable IPv4 packet forwarding

Kubernetes networking sends packets between interfaces and network namespaces. Linux must therefore be allowed to forward IPv4 packets. Store the setting under /etc/sysctl.d so it survives a reboot, load it, and read the effective value.

/etc/sysctl.d/k8s.conf · bash

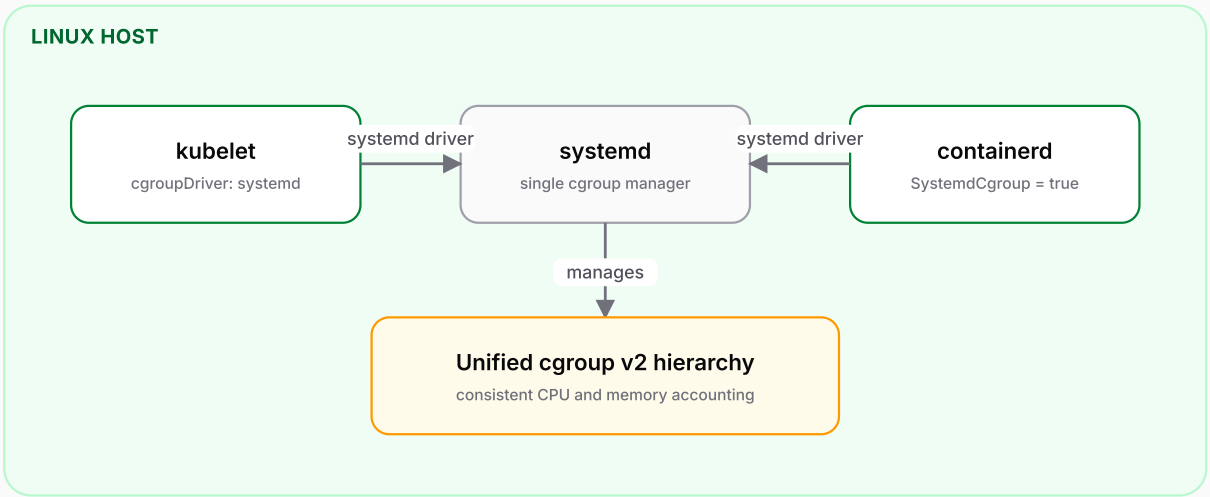
```
cat <<'EOF' | sudo tee /etc/sysctl.d/k8s.conf
net.ipv4.ip_forward = 1
EOF

sudo sysctl --system
sysctl net.ipv4.ip_forward
```

```plaintext
net.ipv4.ip_forward = 1
```

Some network add-ons also require the br_netfilter module and bridge-related sysctls. Those are add-on-specific requirements, not universal commands to copy onto every host. Apply the documented prerequisites for the network implementation chosen during cluster creation.

Gate 3: make cgroup ownership consistent

Control groups (cgroups) are the Linux mechanism behind CPU and memory accounting and isolation. kubelet and the container runtime both participate in the same workload hierarchy. If they use different cgroup drivers, each layer can see and manage resources differently, producing unstable behavior under pressure.



kubelet and containerd delegate cgroup ownership to the same systemd manager.

One cgroup manager for kubelet and containerd

Matching systemd drivers prevent kubelet and containerd from competing over cgroup ownership.

On a systemd-based host, Kubernetes recommends the systemd cgroup driver, especially with cgroup v2. First identify the init system and cgroup filesystem in use.

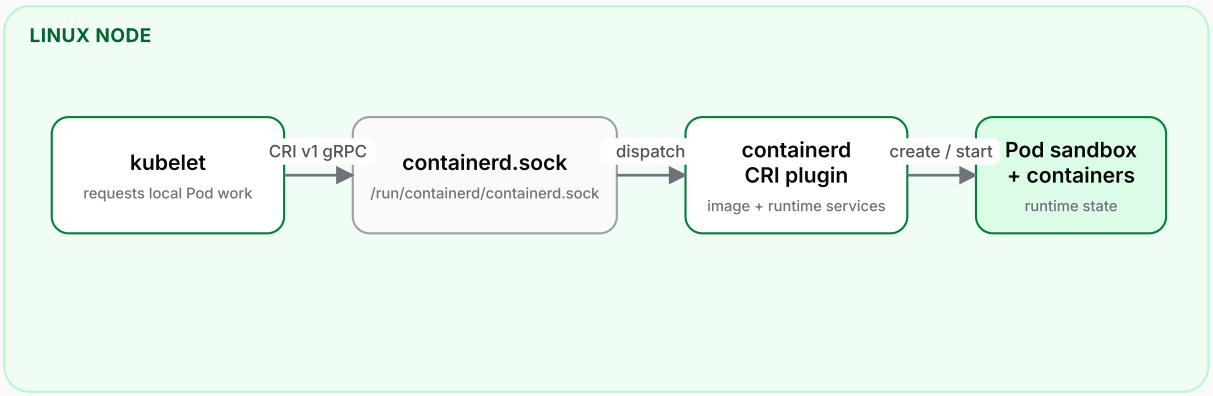
```bash
ps -p 1 -o comm=
stat -fc %T /sys/fs/cgroup/
```

```plaintext
systemd
cgroup2fs
```

This host uses systemd with cgroup v2, so configure the runtime for SystemdCgroup and allow kubeadm to create a matching kubelet configuration during bootstrap. The durable idea is alignment: both components must use the same driver.

Gate 4: provide a working CRI runtime

kubelet does not manage containers through a vendor-specific command line. It calls the CRI v1 API over a local Unix socket. containerd and CRI-O implement that boundary. Docker Engine alone does not: direct dockershim support was removed from Kubernetes, so Docker Engine requires a separate CRI adapter such as cri-dockerd.



kubelet reaches containerd through the local CRI socket; the API server does not.

kubelet reaches containerd through the local CRI socket

The local CRI socket is the contract between kubelet and containerd; the API server does not call the runtime directly.

The following normal path uses containerd. Package installation differs by Linux distribution, so use the current vendor or Kubernetes installation instructions, then verify the installed service and socket rather than assuming the package succeeded.

bash

```
containerd --version
sudo systemctl enable --now containerd
sudo systemctl is-active containerd
sudo test -S /run/containerd/containerd.sock && echo 'containerd socket present'
```

plaintext

```
containerd containerd.io 2.0.6 ...
active
containerd socket present
```

Enable CRI and select the systemd cgroup driver

Inspect /etc/containerd/config.toml. The CRI plugin must not appear in disabled_plugins, and the runc runtime must set SystemdCgroup to true. The plugin path changed between containerd 1.x and 2.x, so edit the section that matches the installed major version.

```bash
sudo grep -nE 'disabled_plugins|SystemdCgroup' /etc/containerd/config.toml
```

containerd 1.x configuration path · toml

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
  runtime_type = "io.containerd.runc.v2"

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

containerd 2.x configuration path · toml

```
[plugins.'io.containerd.cri.v1.runtime'.containerd.runtimes.runc]
  runtime_type = "io.containerd.runc.v2"

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

These are focused fragments, not complete replacement files. Preserve the rest of the generated configuration, make the change in the existing runtime section, then restart containerd and inspect the service if it does not return to active state.

```bash
sudo systemctl restart containerd
sudo systemctl status containerd --no-pager
sudo journalctl -u containerd -n 50 --no-pager
```

Verify the CRI boundary with crictl

crictl is the administrator's client for a CRI runtime. Point it at the same socket kubelet will use. An explicit endpoint avoids slow probing and makes the ownership boundary visible.

/etc/crictl.yaml · yaml

```yaml
runtime-endpoint: unix:///run/containerd/containerd.sock
image-endpoint: unix:///run/containerd/containerd.sock
timeout: 10
debug: false
```

bash

```bash
sudo crictl info
sudo crictl images
sudo crictl pods
```

A structured response from crictl info proves that the CRI plugin answered through the socket—not merely that the containerd process exists. Before cluster bootstrap, empty image and Pod lists are normal.

Install and inspect the Kubernetes tools

kubeadm bootstraps and upgrades the cluster, kubelet runs node workloads, and kubectl talks to the API. kubeadm does not install or manage these packages. Install the release line intended for the cluster using the current Kubernetes package instructions, then keep the components within the supported version-skew policy.

```bash
kubeadm version -o short
kubelet --version
kubectl version --client
sudo systemctl enable kubelet
systemctl status kubelet --no-pager
```

Before kubeadm writes kubelet configuration, the kubelet service can restart repeatedly. That is expected at this point. Verify the binary versions and runtime separately; kubelet becoming healthy is an outcome of cluster bootstrap in the next lesson.

Gate 5: run a normal readiness check

A compact verification pass should move from Linux state to the runtime and finally to kubeadm. This ordering helps locate a failure at the layer that owns it.

```bash
hostnamectl --static
swapon --show
sysctl net.ipv4.ip_forward
ps -p 1 -o comm=
stat -fc %T /sys/fs/cgroup/
systemctl is-active containerd
sudo crictl info >/dev/null && echo 'CRI reachable'
kubeadm version -o short
```

```plaintext
cp1

net.ipv4.ip_forward = 1
systemd
cgroup2fs
active
CRI reachable
v1.x.y
```

The blank line after the hostname represents an empty swapon --show result. Together, the output says that swap is inactive, forwarding is enabled, systemd owns a cgroup v2 hierarchy, containerd is active, and its CRI endpoint responds.

Now run only kubeadm's preflight phase. Supplying the socket removes ambiguity if a machine has multiple runtimes or a nonstandard endpoint.

```bash
sudo kubeadm init phase preflight \
  --cri-socket unix:///run/containerd/containerd.sock
```

```plaintext
[preflight] Running pre-flight checks
```

This command validates prerequisites but does not create the control plane. The remaining output varies with the host; a successful run completes without fatal preflight errors. Resolve reported errors at their owning layer instead of routinely bypassing them with --ignore-preflight-errors.

Important distinctions

- An installed Docker command-line client is not a CRI runtime. kubelet needs a CRI v1 endpoint.
- An active containerd service does not prove CRI readiness. crictl info tests the actual interface kubelet uses.
- A restarting kubelet before bootstrap can be expected. kubeadm has not yet written the node's working configuration.
- Disabling swap now and removing its persistent source are separate actions; both are required for the state to survive a reboot.

- Generic host preparation does not replace the prerequisites of the network add-on selected during cluster creation.

What to remember

- kubeadm consumes a prepared host; it is not a general-purpose Linux remediation tool.
- Unique identity, correct routing, synchronized time, and deliberate firewall access form the machine-level foundation.
- The standard path disables swap, enables IPv4 forwarding, and aligns kubelet and the runtime on the systemd cgroup driver.
- Verify containerd through its CRI socket with crictl, not only through systemctl.
- kubeadm init phase preflight is the last readiness gate; the next lesson uses the prepared host to create the cluster.

Official references

Installing kubeadm documents host, swap, network, and package prerequisites.

Container Runtimes covers CRI endpoints, packet forwarding, containerd configuration, and cgroup drivers.

Ports and Protocols lists the default control-plane and worker-node ports.

Debugging Kubernetes nodes with crictl explains endpoint configuration and CRI inspection.

About cgroup v2 explains the cgroup hierarchy and systemd driver recommendation.

kubeadm init phase documents the standalone preflight phase and CRI socket option.