

Virtualization & Containers

KVM virtualization, Kubernetes and container tooling.

- [Linux - Install KVM](#)
- [RedHat - Install a Kubernetes Cluster on RHEL 9.x | Rocky 9.x: A Step-by-Step Guide](#)
- [RedHat - Kubernetes Cluster on RHEL 9.x | Rocky 9.x \(Part 2\): Control Plane, Workers & Test](#)
- [VMWare - Redhat 9 Install NVidia Pass through and Container Setup](#)

Linux - Install KVM

Last verified: October 2026 for RHEL 9 against the Red Hat virtualization, networking and web console guides.

Prerequisites

- RHEL 9 installed (minimal or with a desktop; the web console used in step 5 does not need a desktop)
- Sudo user with admin rights
- Local repository or Red Hat subscription
- Internet connectivity (for Red Hat subscription)
- Minimum host resources from Red Hat: 6 GB free disk plus 6 GB per VM, 2 GB RAM plus 2 GB per VM, and 4 CPUs

Once the prerequisites are met then jump into installation steps of KVM.

1) Check Whether Virtualization is Enabled or not

To get off the ground, you need to verify if your system supports Virtualization. This is usually enabled in the BIOS/UEFI. To verify, run one of the following commands; any output means the CPU flag is present.

For Intel CPUs

```
grep -e 'vmx' /proc/cpuinfo
```

For AMD CPUs

```
grep -e 'svm' /proc/cpuinfo
```

Alternatively, you can run the following command. VT-x is Intel's virtualization technology and AMD-V is AMD's.

```
lscpu | grep Virtualization
```

Also, you might want to check if KVM modules are loaded.

2) Install Virtualization Packages

First refresh the repositories, install all available updates and reboot once.

```
sudo dnf update -y  
sudo reboot
```

Next, install the virtualization packages listed in the Red Hat guide.

```
sudo dnf install qemu-kvm libvirt virt-install virt-viewer
```

virt-install is a command-line tool for creating virtual machines. **virt-viewer** is a lightweight UI that lets you interact with a VM using the VNC or SPICE protocol.

Optionally, install additional virtualization tools.

```
sudo dnf install -y virt-top guestfs-tools
```

3) Start the Virtualization Services

RHEL 9 uses modular libvirt daemons (virtqemud, virtnetworkd and so on) instead of the old monolithic libvirtd, which is deprecated. Start their sockets as shown in the Red Hat guide:

```
for drv in qemu network nodedev nwfilter secret storage interface; do sudo systemctl start virt${drv}d{,-ro,-admin}.socket; done
```

Then verify that the host is ready for virtualization. If all checks return PASS, you can create VMs.

```
sudo virt-host-validate
```

4) Configure Network Bridge for KVM

If you want to access your KVM virtual machines from outside the KVM hypervisor, configure a network bridge (kvmbr0) and attach the physical interface to it.

Note: The virtual bridge `virbr0` is created automatically by libvirt. VMs on it get a NATed address, so it is mainly useful for testing.

To create the bridge `kvmb0` on RHEL 9.4 or later, run the following commands. Run them from the local console rather than over SSH on the same interface, because the existing connection on that interface is removed and the network drops until the bridge comes up.

```
nmcli connection show
sudo nmcli connection add type bridge con-name kvmb0 ifname kvmb0
sudo nmcli connection modify kvmb0 ipv4.addresses '192.168.1.179/24' ipv4.gateway '192.168.1.1' ipv4.dns '192.168.1.1' ipv4.method manual
sudo nmcli connection delete enp0s3
sudo nmcli connection add type ethernet port-type bridge con-name kvmb0-port1 ifname enp0s3 controller kvmb0
sudo nmcli connection up kvmb0
```

Note: Replace the interface name and IP address details as per your setup. Releases before RHEL 9.4 use `slave-type` and `master` instead of `port-type` and `controller`.

Check the bridge status and its ports with the `ip` command.

```
ip addr show kvmb0
ip link show master kvmb0
```

5) Create a Virtual Machine using the Web Console

On RHEL 9 the Virtual Machine Manager (`virt-manager`) GUI is deprecated; Red Hat recommends the RHEL web console (Cockpit) for managing VMs graphically. Install and enable the web console and its virtual machines plug-in:

```
sudo dnf install cockpit cockpit-machines
sudo systemctl enable --now cockpit.socket
sudo firewall-cmd --add-service=cockpit --permanent
sudo firewall-cmd --reload
```

Open `https://<server-ip>:9090` in a browser (or `https://localhost:9090` on the host itself) and log in with your sudo user.

1. Open **Virtual Machines** and click **Create VM**.
2. Enter the basic configuration: **Name**, **Connection** (privilege level), **Installation type** (for example a local installation medium / ISO), **Operating system**, **Storage**, **Storage Limit** and **Memory**.

3. Click **Create and run** to start the installation straight away, or **Create and edit** to change the VM (for example its network interface) before the operating system is installed.

From the command line, `virt-install` can place a new VM directly on the bridge with `--network bridge=kvmbro`.

Conclusion

And there you have it. In this guide, we have illustrated how to install KVM on RHEL 9.

Sources

- https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/configuring_and_managing_virtualization/assembly_enabling-virtualization-in-rhel-9_configuring-and-managing-virtualization
- https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/configuring_and_managing_networking/configuring-a-network-bridge_configuring-and-managing-networking
- https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/managing_systems_using_the_rhel_9_web_console/getting-started-with-the-rhel-9-web-console_system-management-using-the-rhel-9-web-console
- https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/configuring_and_managing_virtualization/assembly_creating-virtual-machines_configuring-and-managing-virtualization
- https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/9.0_release_notes/deprecated_functionality
- <https://github.com/virt-manager/virt-manager/blob/main/man/virt-install.rst>

RedHat - Install a Kubernetes Cluster on RHEL 9.x | Rocky 9.x: A Step-by-Step Guide

Last verified: October 2026 against the official Kubernetes, containerd, Docker, Calico and EPEL documentation (see **Sources** at the bottom). Versions used in this guide: Kubernetes **v1.37** (latest patch v1.37.1; supported minors are 1.37, 1.36 and 1.35), containerd **2.x** from Docker's `containerd.io` package, Calico **v3.33.0**, RHEL **9.8** / Rocky Linux **9.8**.

Original third-party references (these may be out of date; the official sources are listed at the bottom of the page):

- <https://infotechys.com/install-a-kubernetes-cluster-on-rhel-9>
- https://www.youtube.com/watch?v=_ELvCuXO6y4
- <https://medium.com/weeklycloud/kubernetes-installation-on-rhel-9-d5629f2fa4f9>
- <https://www.youtube.com/watch?v=vX2n05t0AQg&t=1782s>

Prerequisites

On **each node** (control plane and workers), as root:

- RHEL 9.x or Rocky Linux 9.x (9.8 is the current minor release of both).
- At least 2 GB of RAM per machine and 2 or more CPUs on control plane machines.
- Full network connectivity between all machines in the cluster.
- A unique hostname, MAC address and `product_uuid` on every node (watch out for cloned VMs). Check with `ip link` and `cat /sys/class/dmi/id/product_uuid`.

Update the System

You can choose to disable the firewall, or keep it and open the required ports. SELinux is set to **permissive** in both cases.

Set SELinux to permissive (this is the method in the official kubeadm guide, so it applies to both options):

```
setenforce 0
sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/' /etc/selinux/config
```

Permissive mode effectively disables SELinux enforcement. Kubernetes requires this so containers can access the host filesystem, which some network plugins need. You can leave SELinux enforcing only if you know how to configure it yourself; kubeadm does not support that setup.

Option A: disable the firewall

```
systemctl disable --now firewalld.service
```

Calico's system requirements recommend disabling firewalld (or any other iptables manager) because it can interfere with the rules Calico adds.

Option B: keep firewalld and open the required ports

For Kubernetes components to communicate across nodes, open these ports. The Kubernetes ports come from the official *Ports and Protocols* reference. The Calico ports come from Calico's system requirements; the default Calico install in this guide uses BGP plus VXLAN.

- **6443/tcp**: Kubernetes API server
- **2379-2380/tcp**: etcd server client API
- **10250/tcp**: kubelet API
- **10257/tcp**: kube-controller-manager
- **10259/tcp**: kube-scheduler
- **179/tcp**: Calico BGP
- **5473/tcp**: Calico Typha (operator installs always deploy Typha)
- **4789/udp**: Calico VXLAN

Commands to open ports on the **Control Plane Node**:

```
firewall-cmd --permanent --add-port={6443,2379-2380,10250,10257,10259,179,5473}/tcp
firewall-cmd --permanent --add-port=4789/udp
firewall-cmd --reload
```

These ports handle node-to-node communication and pod access on the workers:

- **10250/tcp**: kubelet API
- **10256/tcp**: kube-proxy
- **30000-32767/tcp and udp**: NodePort Services
- **179/tcp**: Calico BGP
- **5473/tcp**: Calico Typha (operator installs always deploy Typha)
- **4789/udp**: Calico VXLAN

Commands to open ports on **Worker Nodes**:

```
firewall-cmd --permanent --add-port={10250,10256,30000-32767,179,5473}/tcp
firewall-cmd --permanent --add-port={30000-32767,4789}/udp
firewall-cmd --reload
```

End of firewall options

Stop NetworkManager from managing Calico's interfaces. Calico's troubleshooting guide says to do this before you install Calico. Run this on every node; the reboot in Step 4 applies it.

```
cat > /etc/NetworkManager/conf.d/calico.conf << EOF
[keyfile]
unmanaged-devices=interface-name:cali*;interface-name:tunl*;interface-name:vxlan.calico;interface-
name:vxlan-v6.calico;interface-name:wireguard.cali;interface-name:wg-v6.cali
EOF
```

EPEL Release (optional; Kubernetes itself does not need EPEL)

RHEL 9:

```
subscription-manager repos --enable codeready-builder-for-rhel-9-$(arch)-rpms
dnf install https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
```

Rocky Linux 9:

```
dnf config-manager --set-enabled crb
dnf install https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
```

After installing EPEL, rerun the upgrade to apply any available updates:

```
dnf upgrade -y
```

If you are running on a VMware virtual machine, run the following:

```
dnf install open-vm-tools -y
```

Install vim colors for scripting (optional):

```
dnf install git -y
git clone https://github.com/flazz/vim-colorschemes ~/.vim/
cp ~/.vim/colors/desert.vim /etc/vimrc.local
```

Step 1: Add Kernel Modules

Load the kernel modules used by the container runtime and pod networking (**on each node**):

```
modprobe br_netfilter
modprobe overlay
```

Next, create a configuration file (**as the root user on each node**) so these modules load at boot:

```
cat > /etc/modules-load.d/k8s.conf << EOF
br_netfilter
overlay
EOF
```

Step 2: Configure Sysctl

Set the kernel parameters that Kubernetes networking relies on (**on each node**):

```
cat > /etc/sysctl.d/k8s.conf << EOF
net.ipv4.ip_forward = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
```

These commands adjust the following kernel parameters:

Kernel Parameter	Description
net.ipv4.ip_forward	Enables IPv4 packet forwarding. This is the only sysctl the official Kubernetes container runtime guide requires.
net.bridge.bridge-nf-call-iptables	Enables iptables to process bridged IPv4 traffic. Not required by the official guide, but some network plugins expect it. Needs the <code>br_netfilter</code> module.

Kernel Parameter	Description
net.bridge.bridge-nf-call-ip6tables	Enables ip6tables to process bridged IPv6 traffic. Not required by the official guide, but some network plugins expect it. Needs the <code>br_netfilter</code> module.

Apply the changes without rebooting and verify:

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

Step 3: Disable Swap

By default the kubelet fails to start if swap is detected. To disable swap on each server in your cluster:

```
swapoff -a
```

This command turns off all swap devices. To keep swap off after a reboot, comment out the swap entry in `/etc/fstab`:

```
sed -e '/swap/s/^/#/' -i /etc/fstab
```

The `sed` command finds the line that contains the swap entry and comments it out by adding a `#` at the start, for example:

```
#/dev/mapper/vg00-swap none swap defaults 0 0
```

Alternatively, newer Kubernetes releases can run with swap enabled. Set `failSwapOn: false` and a `swapBehavior` in the kubelet configuration; see the official *Swap memory management* documentation. This guide disables swap.

Step 4: Install Containerd

In this step, we'll install containerd on each node as the container runtime. Kubernetes talks to containerd through the CRI. Docker Engine itself is not a CRI runtime: dockershim was removed from the kubelet in v1.24.

Add the Docker CE Repository

The `containerd.io` package comes from Docker's RHEL repository. First remove any conflicting packages; Docker's install guide lists these, and `dnf` may report that none are installed. Then add

the repository:

```
dnf remove docker docker-client docker-client-latest docker-common docker-latest docker-latest-logrotate
docker-logrotate docker-engine podman runc
dnf -y install dnf-plugins-core
dnf config-manager --add-repo https://download.docker.com/linux/rhel/docker-ce.repo
```

Update the package cache, then install the `containerd.io` package. As of October 2026 it installs containerd 2.3.x.

```
dnf makecache
dnf -y install containerd.io
```

If prompted to accept the GPG key, verify that the fingerprint is `060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35`.

Configure Containerd

The `/etc/containerd/config.toml` shipped with the package contains `disabled_plugins = ["cri"]`, which disables the CRI plugin that Kubernetes needs. Replace it with the full default configuration:

```
containerd config default > /etc/containerd/config.toml
```

Next, set the `SystemdCgroup` option to `true`. The kubelet and the container runtime must use the same cgroup driver. The official docs recommend `systemd` on systemd-based hosts such as RHEL/Rocky and with cgroup v2. With containerd 2.x, the kubelet automatically detects the cgroup driver from containerd.

```
sed -i 's/SystemdCgroup = false/SystemdCgroup = true/g' /etc/containerd/config.toml
grep -n 'SystemdCgroup' /etc/containerd/config.toml
```

containerd 2.x uses a newer config format, so the setting is now under a different section. The `sed` command still works because it only changes the value:

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

(On containerd 1.x the section was `[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]`.)

Enable and start containerd so it also starts on boot:

```
systemctl enable --now containerd.service
```

Reboot the machine to apply the SELinux, module and NetworkManager changes:

```
systemctl reboot
```

After the reboot, check that containerd is up and running:

```
systemctl status containerd.service
```

Alternative runtime: CRI-O can be used instead of containerd. It is installed from the CRI-O project's own packages (see the [cri-o/packaging README](#)), and its socket is `unix:///var/run/crio/crio.sock`. Install only one container runtime per node. If kubeadm detects more than one, specify which to use with `--cri-socket`.

Step 5: Install Kubernetes Components

Install kubelet, kubeadm and kubectl on **every node** from the official community-owned repository at `pkgs.k8s.io`. Each Kubernetes minor version has its own repository; this one only has packages for **v1.37**. For another supported minor (1.36 or 1.35), change `v1.37` in both URLs.

Add Kubernetes Repository

The `exclude` line stops a normal `dnf update` from upgrading the Kubernetes packages. Kubernetes upgrades must follow the `kubeadm upgrade` procedure.

```
# 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:/v1.37/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/v1.37/rpm/repodata/repomd.xml.key
exclude=kubelet kubeadm kubectl cri-tools kubernetes-cni
EOF
```

Install Kubernetes Packages

```
dnf makecache; dnf install -y kubelet kubeadm kubectl --disableexcludes=kubernetes
```

The `--disableexcludes=kubernetes` flag lets this install use the Kubernetes packages that the repository's `exclude` line otherwise hides. This is the DNF4 syntax used on RHEL/Rocky 9.

Enable the kubelet Service

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

To verify the installation so far, run:

```
kubeadm version  
kubelet --version  
kubectl version --client
```

Don't worry about kubelet errors at this point. The kubelet restarts in a loop until `kubeadm init` (control plane) or `kubeadm join` (workers) gives it a configuration.

Continued in [RedHat - Kubernetes Cluster on RHEL 9.x | Rocky 9.x \(Part 2\): Control Plane, Workers & Test.](#)

RedHat - Kubernetes Cluster on RHEL 9.x | Rocky 9.x (Part 2): Control Plane, Workers & Test

This page continues [RedHat - Install a Kubernetes Cluster on RHEL 9.x | Rocky 9.x: A Step-by-Step Guide](#).

Install a Kubernetes Cluster on RHEL 9.x | Rocky 9.x: Control Plane (Master) Node Configuration

NOTE: Up to this point, we've installed and configured the Kubernetes components on all nodes. From here on, we will focus on the control plane (master) node, called `k8s-master` in the examples.

Step 6: Initialize the Kubernetes Control Plane

Optionally, pre-pull the control plane images:

```
kubeadm config images pull
```

This downloads the control plane container images (API server, controller manager, scheduler, etcd, CoreDNS, kube-proxy, pause) from `registry.k8s.io`, so `kubeadm init` doesn't have to download

them. `kubeadm init` pulls them automatically if you skip this step.

Now initialize the control plane:

```
kubeadm init --pod-network-cidr 10.244.0.0/16 --control-plane-endpoint "<control-plane-ip-or-dns>:6443" --upload-certs --v=5
```

- `--pod-network-cidr` sets the pod IP range. It must not overlap with your host networks (for example 192.168.x.x), and it must match the CIDR configured for Calico below.
- `--control-plane-endpoint` sets the shared endpoint for the control plane: an IP address or DNS name such as `k8s-master`, or a load balancer address. Setting it now allows you to make the cluster highly available later.
- `--upload-certs` uploads the control plane certificates to the cluster so additional control plane nodes can join.
- `--v=5` enables verbose logging.

When it finishes, `kubeadm init` prints a `kubeadm join` command. Save it. The token in that command is secret: anyone who has it can add nodes to your cluster.

Set Up kubeconfig File

Set up the kubeconfig file so `kubectl` can talk to the cluster. Run these commands as the user who will run `kubectl`:

```
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:

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

`admin.conf` grants cluster-admin rights, and `super-admin.conf` bypasses RBAC entirely. Do not share either file; the official docs recommend moving `super-admin.conf` to a safe location.

Deploy Pod Network (Calico)

A CNI network add-on is required. CoreDNS does not start until one is installed. The steps below follow the current official Calico operator install for **Calico v3.33.0**, which is tested against Kubernetes 1.35, 1.36 and 1.37.

Install the Calico custom resource definitions (this manifest is for Kubernetes 1.36 or later):

```
kubectl create -f
https://raw.githubusercontent.com/projectcalico/calico/v3.33.0/manifests/v3_projectcalico_org.yaml
```

Install the Tigera Operator:

```
kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.33.0/manifests/tigera-operator.yaml
```

Download the custom resources manifest (iptables data plane):

```
curl -O https://raw.githubusercontent.com/projectcalico/calico/v3.33.0/manifests/custom-resources.yaml
```

Or using `wget`:

```
wget https://raw.githubusercontent.com/projectcalico/calico/v3.33.0/manifests/custom-resources.yaml
```

The manifest defaults to a `192.168.0.0/16` pod pool. Change the `cidr` setting to match the `--pod-network-cidr` you passed to `kubeadm init`:

```
sed -i 's/cidr: 192\168\0\0\16/cidr: 10.244.0.0\16/g' custom-resources.yaml
grep -n 'cidr' custom-resources.yaml
```

Finally, create the Calico custom resources:

```
kubectl create -f custom-resources.yaml
```

Watch the rollout until every component shows `True` in the `AVAILABLE` column (this takes a few minutes):

```
watch kubectl get tigerastatus
```

Step 7: Join Worker Nodes

After the control plane is initialized, join the worker nodes to the cluster. Kubernetes provides a join command that includes a token and the control plane's address.

Get Join Command on the Control Plane Node

If you didn't save the join command from `kubeadm init`, or its token has expired (the token created by `kubeadm init` is valid for 24 hours), generate a new one on the control plane node:

```
kubeadm token create --print-join-command
```

Run Join Command on Worker Nodes

Run the join command as root on each worker node. It looks like this:

```
kubeadm join <control-plane-host>:6443 --token <token> --discovery-token-ca-cert-hash sha256:<hash>
```

Verify Worker Node Join

Back on the control plane node, check that the workers have joined:

```
kubectl get nodes
```

This should list all nodes in the cluster: the control plane node and the newly joined workers. When Calico is running on a node, its status becomes **Ready**, which means it can accept workloads.

NGINX Test Deployment

To test your cluster, deploy a simple application such as an NGINX web server. Here's a sample YAML manifest that deploys NGINX as a test deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
```

```
ports:
- containerPort: 80
```

Deploy NGINX

Save the YAML above to a file named `nginx-deployment.yaml`, then apply it with `kubectl apply`:

```
kubectl apply -f nginx-deployment.yaml
```

```
deployment.apps/nginx-deployment created
```

This creates three NGINX pod replicas, each running a container that exposes port 80. To check the status of the deployment:

```
kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3/3	3	3	2m40s

To verify that the NGINX pods are running:

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-7c79c4bf97-gnbfn	1/1	Running	0	6m6s
nginx-deployment-7c79c4bf97-tmbpg	1/1	Running	0	6m6s
nginx-deployment-7c79c4bf97-vgh42	1/1	Running	0	6m6s

Expose NGINX to the external network

Once the pods are running, expose NGINX outside the cluster with a Kubernetes Service:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
```

```
ports:
- protocol: TCP
  port: 80
  targetPort: 80
type: LoadBalancer
```

Save the YAML above to a file named `nginx-service.yaml`, then apply it:

```
kubectl apply -f nginx-service.yaml
```

```
service/nginx-service created
```

This creates a Service of type LoadBalancer. To see its external IP address and ports:

```
kubectl get service nginx-service
```

Once you have the external IP address, open it in a web browser. You should see the default NGINX welcome page, which shows the cluster is serving web traffic.

On bare-metal or VM clusters (like this one), nothing provisions LoadBalancer services out of the box, so `EXTERNAL-IP` stays `<pending>` unless you install a load-balancer implementation. A LoadBalancer Service still gets a NodePort by default, though: take the port after `80:` in the `PORT(S)` column and browse to `http://<any-node-ip>:<node-port>`.

Sources

- Kubernetes: Installing kubeadm: <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/>
- Kubernetes: Container Runtimes: <https://kubernetes.io/docs/setup/production-environment/container-runtimes/>
- Kubernetes: Creating a cluster with kubeadm: <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/>
- Kubernetes: Ports and Protocols: <https://kubernetes.io/docs/reference/networking/ports-and-protocols/>
- Kubernetes: Releases and support periods: <https://kubernetes.io/releases/> (current stable: <https://dl.k8s.io/release/stable.txt>)

- Kubernetes: Swap memory management: <https://kubernetes.io/docs/concepts/cluster-administration/swap-memory-management/>
- Kubernetes: kubeadm token: <https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-token/>
- Kubernetes: Service type LoadBalancer: <https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer>
- containerd: Getting started: <https://github.com/containerd/containerd/blob/main/docs/getting-started.md>
- Docker: Install Docker Engine on RHEL (repository and containerd.io): <https://docs.docker.com/engine/install/rhel/>
- CRI-O packaging: <https://github.com/cri-o/packaging/blob/main/README.md#usage>
- Calico: Install on on-premises deployments: <https://docs.tigera.io/calico/latest/getting-started/kubernetes/self-managed-onprem/onpremises>
- Calico: System requirements: <https://docs.tigera.io/calico/latest/getting-started/kubernetes/requirements>
- Calico: Configure NetworkManager: <https://docs.tigera.io/calico/latest/operations/troubleshoot/troubleshooting#configure-networkmanager>
- Calico: Typha overview: <https://docs.tigera.io/calico/latest/reference/typha/overview>
- Calico releases: <https://github.com/projectcalico/calico/releases>
- Fedora EPEL: Getting started: <https://docs.fedoraproject.org/en-US/epel/getting-started/>
- Red Hat Enterprise Linux release dates: <https://access.redhat.com/articles/3078>
- Rocky Linux news (9.8 release): <https://rockylinux.org/news>

VMWare - Redhat 9 Install NVidia Pass through and Container Setup

Installing an NVIDIA driver in a Red Hat 9 virtual machine (VM) with GPU passthrough involves several steps, including configuring both the VMware ESXi host and the RHEL guest OS. GPU passthrough dedicates the entire physical GPU to a single VM, bypassing the hypervisor.

Reviewed October 2026 against RHEL 9.8 and NVIDIA data center driver branch 615. This guide uses an NVIDIA **Tesla M10 (Maxwell)**. NVIDIA driver branch **R580** is the last branch that supports Maxwell, Pascal and Volta GPUs. It is a Long Term Support Branch with end of life in June 2028, so a Maxwell card must stay on the **580** proprietary DKMS stream. Newer branches drop Maxwell, and from branch 615 NVIDIA ships only the open kernel modules, which need Turing or newer.

Step 1: Enable GPU passthrough on the ESXi host

1. In the host BIOS/UEFI, make sure **Intel VT-d** (AMD: **IOMMU**) is enabled and PCI mapping above 4 GB is allowed.
2. In the vSphere Client select the host, then **Configure > Hardware > PCI Devices** and open **All PCI Devices**. (ESXi Host Client: **Manage > Hardware > PCI Devices**.)
3. Select the NVIDIA GPU and click **Toggle Passthrough**.
4. If the device shows an orange icon, put the host in maintenance mode and reboot it before using the device. The passthrough status should then show as Enabled/Active.

Step 2: Configure the RHEL 9 virtual machine

1. Power off the virtual machine. **This is required before assigning PCI devices.**
2. Edit the settings of the RHEL 9 VM in the vSphere Client.
3. Under **VM Options > Boot Options** set **Firmware** to **EFI**, because GPU passthrough VMs will not boot correctly on legacy BIOS.
4. **Turn off Secure Boot (required):** with the VM still powered off, go to **Edit Settings > VM Options > Boot Options** and uncheck **Secure Boot**. With Secure Boot on, the NVIDIA kernel modules built in Step 4 are unsigned and will not load unless you sign them yourself and enroll the key (MOK). This guide does not use module signing, so Secure Boot must be off.

5. Under **Virtual Hardware**, click **Add New Device**, choose **PCI Device** and select the NVIDIA GPU you enabled for passthrough.
6. Under **Memory**, reserve all guest memory (All locked). A passthrough device needs the full memory reservation.
7. Go to **VM Options > Advanced**, click **Edit Configuration** and add these parameters:
 - `pciPassthru.use64bitMMIO = TRUE`
 - `pciPassthru.64bitMMIOSizeGB = 64`: the value depends on GPU memory. Broadcom's rule is to add up the framebuffer memory of every GPU in the VM. If the total is a power of 2, use the next power of 2. If it falls between two powers of 2, round up twice. One Tesla M10 GPU (8 GB) needs at least **16**; **64** also works.
 - `hypervisor.cpuid.v0 = FALSE`: used in the tested setup. It is not part of Broadcom's GPU passthrough KBs.
8. Save the VM settings and power on the VM.

Step 3: Prepare the OS

VMware tools: Broadcom recommends the open-vm-tools package shipped by Red Hat.

```
sudo dnf install -y open-vm-tools
```

Enable CodeReady Builder and EPEL: the NVIDIA driver RPMs depend on packages that only exist in EPEL (for example DKMS).

```
sudo subscription-manager repos --enable=codeready-builder-for-rhel-9-x86_64-rpms
sudo dnf install https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
```

Install build dependencies: DKMS needs the kernel headers and development packages. On RHEL 9 NVIDIA uses `kernel-devel-matched`, which also installs the matching devel package for future kernel updates so DKMS can rebuild the module.

```
sudo dnf groupinstall "Development Tools"
sudo dnf install kernel-devel-matched kernel-headers
```

Disable the Nouveau driver (optional): the open-source Nouveau driver can conflict with the NVIDIA driver. The NVIDIA RPMs disable nouveau automatically during install (`nvidia-boot-update`). To do it by hand:

```
sudo bash -c "echo blacklist nouveau > /etc/modprobe.d/blacklist-nvidia-nouveau.conf"
sudo bash -c "echo options nouveau modeset=0 >> /etc/modprobe.d/blacklist-nvidia-nouveau.conf"
sudo dracut --force
sudo reboot
```

Step 4: Install the NVIDIA driver (580 branch, proprietary DKMS)

For Maxwell GPUs such as the Tesla M10, do **not** use `nvidia-driver:latest-dkms`, `nvidia-driver:open-dkms` or `nvidia-open`. In NVIDIA's RHEL 9 repository `latest-dkms` now resolves to the 610 branch, which no longer supports Maxwell. The open kernel modules only support Turing and newer GPUs.

Add the repository: NVIDIA's network repository for RHEL 9.

```
sudo dnf config-manager --add-repo
https://developer.download.nvidia.com/compute/cuda/repos/rhel9/x86_64/cuda-rhel9.repo
sudo dnf clean expire-cache
```

Install the driver: use the `dnf module` stream locked to the **580** branch with proprietary DKMS modules. Its default profile installs the full driver plus `kmod-nvidia-latest-dkms`.

```
sudo dnf module install nvidia-driver:580-dkms
```

Later updates stay on the 580 branch with a normal `sudo dnf update`.

Secure Boot: the DKMS-built proprietary modules are not signed by Red Hat. Red Hat-signed precompiled NVIDIA modules exist only for the open kernel modules (RHEL 10), which Maxwell GPUs cannot use. Secure Boot must be off (Step 2). Check it inside the VM before installing the driver:

```
mokutil --sb-state
```

The output should be `SecureBoot disabled`. If it says `SecureBoot enabled`, power off the VM and uncheck Secure Boot under **Edit Settings > VM Options > Boot Options**.

Reboot the VM:

```
sudo reboot
```

Step 5: Verify the installation

After the VM reboots, run `nvidia-smi`. If the install worked, it shows the GPU, which confirms that the RHEL guest is using it through passthrough:

```
$ nvidia-smi
+-----+

```

```

| NVIDIA-SMI 580.95.05      Driver Version: 580.95.05   CUDA Version: 13.0   |
+-----+-----+-----+
| GPU  Name           Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap |      Memory-Usage | GPU-Util  Compute M. |
|                                   |        MIG M.     |                        |
+=====+
|  0  Tesla M10             Off | 00000000:XX:00.0 Off |          N/A |
| N/A   36C   P8             8W /  53W |   3MiB /  8192MiB |      0%      Default |
|                                   |          N/A      |                        |
+-----+-----+-----+
+-----+
| Processes:                                |
| GPU  GI  CI           PID  Type  Process name          GPU Memory |
|   ID ID              |           |          Usage           |      |
+=====+
| No running processes found                |
+-----+

```

Step 6: Set up containers

Add NVIDIA's container toolkit repository (the CUDA repository above also carries the same packages):

```

curl -s -L https://nvidia.github.io/libnvidia-container/stable/rpm/nvidia-container-toolkit.repo | \
sudo tee /etc/yum.repos.d/nvidia-container-toolkit.repo

```

Install the NVIDIA Container Toolkit:

```

sudo dnf install -y nvidia-container-toolkit

```

The older `nvidia-docker2` / `nvidia-docker` and `nvidia-container-runtime` repositories and packages are not part of NVIDIA's current install guide; use the repository above.

Docker

Configure Docker to use the NVIDIA runtime and restart it:

```

sudo nvidia-ctl runtime configure --runtime=docker
sudo systemctl restart docker

```

Test it:

```
$ docker run --rm --runtime=nvidia --gpus all ubuntu nvidia-smi
```

+-----+											
NVIDIA-SMI 580.95.05			Driver Version: 580.95.05			CUDA Version: 13.0					
+-----+											
GPU Name		Persistence-M		Bus-Id		Disp.A		Volatile Uncorr. ECC			
Fan Temp Perf		Pwr:Usage/Cap				Memory-Usage		GPU-Util		Compute M.	
						MIG M.					
=====+											
=====											
0 Tesla M10		Off		00000000:XX:00.0		Off		N/A			
N/A 36C P8		8W / 53W		3MiB / 8192MiB		0%		Default			
						N/A					
+-----+											
+-----+											
Processes:											
GPU		GI		CI		PID		Type		Process name	
ID		ID								Usage	
=====											
=====											
No running processes found											
+-----+											

Podman (CDI)

For Podman, RHEL's built-in container engine, NVIDIA recommends CDI instead of `nvidia-ctk runtime configure`. Since toolkit v1.18 the `nvidia-cdi-refresh` service generates the CDI spec automatically at `/var/run/cdi/nvidia.yaml`. It runs when the toolkit or driver is installed or upgraded and at boot. List the devices:

```
nvidia-ctk cdi list
```

Regenerate the spec if needed (for example after removing a driver):

```
sudo systemctl restart nvidia-cdi-refresh.service
# or manually
sudo nvidia-ctk cdi generate --output=/var/run/cdi/nvidia.yaml
```

Test it:

```
podman run --rm --security-opt=label=disable --device=nvidia.com/gpu=all ubuntu nvidia-smi
```

Troubleshooting: driver missing after a kernel update

With `kernel-devel-matched` installed, DKMS rebuilds the module automatically after kernel updates. If `nvidia-smi` stops working after a kernel update, check which kernels the module is built for:

```
dkms status
```

Example output where the module is only built for an older kernel:

```
nvidia/580.95.05, 5.14.0-427.42.1.el9_4.x86_64, x86_64: installed
```

Make sure `kernel-devel` for the running kernel is installed, then build the module for the current kernel. Use the version shown by `dkms status`:

```
sudo dkms install nvidia/580.95.05
```

References

- [NVIDIA Container Toolkit install guide](#) and [CDI support](#)
- [Ollama Docker image](#) (container configuration)
- [NVIDIA blog: driver module streams on RHEL](#) (background)

Sources (checked October 2026)

- [Red Hat: RHEL release dates](#) (RHEL 9.8 GA 2026-05-19)
- [NVIDIA Driver Installation Guide: RHEL](#), [Recent Updates](#) (proprietary modules removed), [Advanced Options](#) (nvidia-boot-update)
- [NVIDIA supported drivers](#) and [NVIDIA blog: R580 last branch for Maxwell/Pascal/Volta](#)
- [NVIDIA Container Toolkit sample workload](#)
- [Broadcom KB 334594](#) (EFI + 64-bit MMIO sizing), [KB 312208](#) (DirectPath I/O requirements), [KB 309986](#) (Toggle Passthrough), [KB 313456](#) (open-vm-tools)