

# Kubernetes

- [Talos Linux](#)
  - [Provision nodes](#)
  - [How to scale down a Talos cluster](#)
  - [Node setup with Nvidia GPU](#)
  - [Switch CNI](#)
- [Cluster Setup](#)
  - [Metal LB](#)
  - [nginx Ingress Controller](#)
- [Maintenance](#)
  - [Convert PV from ReadWriteOnce to ReadWriteMany](#)
  - [Restore Longhorn PV from backup](#)

# Talos Linux

# Provision nodes

After creating the nodes in Proxmox run the following commands. In this example, I have 3 API controllers and 3 worker nodes. Each controller will have an etcd database.

```
talosctl gen config koryscluster https://<haproxy IP>:6443
```

Edit the worker.yaml that gets created by the last command. We will need to add longhorn support.

```
kubelet:
  extraMounts:
    - destination: /var/lib/longhorn
      type: bind
      source: /var/lib/longhorn
    options:
      - bind
      - rshared
      - rw
```

Apply the config to the controlplanes.

```
talosctl apply-config --insecure -n <IP ctl-1> --file controlplane.yaml
talosctl apply-config --insecure -n <IP ctl-2> --file controlplane.yaml
talosctl apply-config --insecure -n <IP ctl-3> --file controlplane.yaml
```

Bootstrap the cluster. This will initialize etcd.

```
talosctl bootstrap -n <IP ctl-1> -e <IP ctl-1> --talosconfig=./talosconfig
```

Generate the k8s config file

```
talosctl kubeconfig ./kube -n <IP ctl-1> -e <IP ctl-1> --talosconfig=./talosconfig
```

Either you can copy the config to ~/.kube/config or export as an environment variable. Lets confirm the cluster is alive and we can talk to the API servers via HA proxy.

```
kubectl get nodes
```

After this is successful, we can get the workers setup. These will hold our longhorn data. Let add more space to these and we will need to have talos upgrade the config to add iscsi support.

```
talosctl apply-config --insecure -n <IP worker-1> --file worker.yaml  
talosctl apply-config --insecure -n <IP worker-2> --file worker.yaml  
talosctl apply-config --insecure -n <IP worker-3> --file worker.yaml
```

Lets start the upgrade process. We can use image factory to setup a new config. The hash needed for longhorn support is already setup with the following. Note the version number must be greater than or equal to the current talos version installed on the node.

```
talosctl upgrade --talosconfig=./talosconfig --nodes <IP worker-1> -e <IP worker-1> --image  
factory.talos.dev/installer/613e1592b2da41ae5e265e8789429f22e121aab91cb4deb6bc3c0b6262961245:v1.9.1  
talosctl upgrade --talosconfig=./talosconfig --nodes <IP worker-2> -e <IP worker-2> --image  
factory.talos.dev/installer/613e1592b2da41ae5e265e8789429f22e121aab91cb4deb6bc3c0b6262961245:v1.9.1  
talosctl upgrade --talosconfig=./talosconfig --nodes <IP worker-3> -e <IP worker-3> --image  
factory.talos.dev/installer/613e1592b2da41ae5e265e8789429f22e121aab91cb4deb6bc3c0b6262961245:v1.9.1
```

We now have a working cluster. Nothing besides the core k8s system is installed. We can refer to the Kubernetes section to continue setup.

# How to scale down a Talos cluster

To remove nodes from a Talos Linux cluster:

```
talosctl -n <IP.of.node.to.remove> reset
```

```
kubecttl delete node <nodename>
```

# Node setup with Nvidia GPU

## Introduction

Refer to the previous page for the general concept of Talos worker.yaml. This will build on the same concept but add required extensions and load modules needed for pods to see a GPU and use it for AI workloads. This guide will reference this [documentation](#).

## Image Factory

First, we need to create the [new image](#) that will be loaded on the node. We have the option to choose from open-source Nvidia drivers to the proprietary version. The following are the extensions you need to search for, depending on your choice. You will notice they have a lts and production version. I have been using LTS. The main item to note is that they should match. Meaning if you choose LTS chose this for both options.

### Open source extensions

- `nvidia-open-gpu-kernel-modules`
- `nvidia-container-toolkit`

### Proprietary extensions

- `nonfree-kmod-nvidia`
- `nvidia-container-toolkit`

## Provision node

Now that we have the image file, we can add this to our worker.yaml, as seen on the previous page. This should look similar to the following under install.

```
machine:  
  install:
```

```
disk: /dev/nvme0n1 # The disk used for installations.
image:
factory.talos.dev/installer/53b5a3efb4cca0300d7947f45577df156effe1be2f373daf3ffd8d7ba08ea899:v1.9.5
wipe: false
```

Depending on the node type (virtual/physical), we need to boot the new node from the iso or image that came out of our image factory process. After the node is booted and ready to accept configuration, let's tell it to install these extensions and join our cluster.

```
talosctl apply-config --insecure -n <new node IP> --file worker.yaml
```

After the node reboots, we can verify the extensions are installed by running the following

```
talosctl get extensions
```

The output should look similar to the following

NODE	NAMESPACE	TYPE	ID	VERSION	NAME	VERSION
192.168.249.11	runtime	ExtensionStatus	0	1	iscsi-tools	v0.1.6
192.168.249.11	runtime	ExtensionStatus	1	1	nonfree-kmod-nvidia-lts	535.216.03-v1.9.1
192.168.249.11	runtime	ExtensionStatus	2	1	nvidia-container-toolkit-lts	535.216.03-v1.17.2
192.168.249.11	runtime	ExtensionStatus	3	1	schematic	
4ba64c429e0aa252d716a668cf66b056b6ee3805f0ee0d7258a3a71e81df8e50						
192.168.249.11	runtime	ExtensionStatus	modules.dep	1	modules.dep	6.12.6-talos

## Patch node

Now we need to patch the node to load the nvidia modules. Create a patch-gpu.yaml.

```
machine:
  kernel:
    modules:
      - name: nvidia
      - name: nvidia_uvm
      - name: nvidia_drm
      - name: nvidia_modeset
  sysctls:
    net.core.bpf_jit_harden: 1
```

Apply the patch with the following.

```
talosctl patch mc --patch @patch-gpu.yaml
```

Confirm the patch with the following commands.

```
talosctl read /proc/driver/nvidia/version
```

```
NVRM version: NVIDIA UNIX x86_64 Kernel Module 535.216.03 Fri Oct 25 22:43:06 UTC 2024
```

```
GCC version: gcc version 14.2.0 (GCC)
```

```
talosctl read /proc/modules
```

```
nvidia_uvm 1908736 0 - Live 0xffffffffc4175000 (PO)
```

```
nvidia_drm 94208 0 - Live 0xffffffffc0575000 (PO)
```

```
nvidia_modeset 1531904 2 nvidia_drm, Live 0xffffffffc4356000 (PO)
```

```
nvidia 62771200 19 nvidia_uvm,nvidia_modeset, Live 0xffffffffc0596000 (PO)
```

## Patch runtime

We need to set Nvidia as the default runtime. This can be done with a YAML inside of k8s, but I found the best way is to apply this to the node. I don't know why this is required, but pods would not see the GPU or even run nvidia-smi without this

create a file runtime-patch.yaml

```
- op: add
  path: /machine/files
  value:
    - content: |
        [plugins]
        [plugins."io.containerd.cri.v1.runtime"]
        [plugins."io.containerd.cri.v1.runtime".containerd]
        default_runtime_name = "nvidia"
      path: /etc/cri/conf.d/20-customization.part
      op: create
```

```
talosctl patch mc --patch @runtime-patch.yaml
```

# Nvidia device plugin



We can use Helm to install the device plugin. This runs a daemonset looking for nodes with a GPU. It will only run on those nodes.

Time slicing is not enabled by default. This means one GPU can only be used by one pod. The following configuration will enable one GPU to share time with many pods at once.

Create the config file time-slicing-config.yaml

```
version: v1
flags:
  migStrategy: none
sharing:
  timeSlicing:
    resources:
      - name: nvidia.com/gpu
        replicas: 5
```

Now, we can pass the config file to helm and install the device plugin.

```
helm repo add nvdp https://nvidia.github.io/k8s-device-plugin
helm repo update
helm install nvidia-device-plugin nvdp/nvidia-device-plugin --version=0.13.0 --set=runtimeClassName=nvidia \
--set-file config.map.config=./time-slicing-config.yaml \
--namespace nvidia-system
```

# Testing

Let us run this pod to test if the GPU is picked up correctly.

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod
spec:
  restartPolicy: Never
  containers:
    - name: cuda-container
      image: nvcr.io/nvidia/k8s/cuda-sample:devicequery-cuda11.7.1-ubuntu20.04
      resources:
```

limits:

nvidia.com/gpu: 1 # requesting 1 GPU

tolerations:

- key: nvidia.com/gpu

operator: Exists

effect: NoSchedule

# Switch CNI

## Flannel to Calico

### Introduction

When I first provided my Talos cluster, I didn't realize it comes with Flannel as the default CNI. At first, this wasn't a problem, but as I mature using Kubernetes, I quickly found I need more advanced network policies. These are the steps I took to convert a live cluster from Flannel to Calico with no downtime.

### Preperation

Since this is a big move and a lot can go wrong, I went through several steps to ensure uptime.

1. Set up a test cluster to mirror your current setup. All the actions we are going to take can first be done there to ensure nothing pops up during conversion.
2. Snapshot all controllers (if possible) or do a backup of etcd

### Convert the cluster

The steps are straightforward, and thankfully, Kubernetes is very good at extensibility, so adding dual CNI providers does not break networking.

First, let's create our namespace and set the proper permissions needed by Calico

```
kubectrl create namespace tigera-operator  
kubectrl label namespace tigera-operator pod-security.kubernetes.io/enforce=privileged
```

Next, we will use Helm to install the provider. This can be done with regular manifests, but it will be easier to maintain and upgrade in the future with Helm. You may need to update the version.

```
helm install calico projectcalico/tigera-operator --version v3.29.3 --namespace tigera-operator
```

After some time, we can check to make sure that all pods are running and ready! This took a bit to finish so make sure its done before moving on.

```
kubectl get pods -n calico-system
```

Once that is complete, we can edit the default IP pool used by Calico. This is optional, just note the default is 192.168.0.0/16, which was not ideal for my network. Change the cidr field to what you require.

```
apiVersion: crd.projectcalico.org/v1
kind: IPPool
metadata:
  name: default-ipv4-ippool
spec:
  allowedUses:
    - Workload
    - Tunnel
  blockSize: 26
  cidr: 10.225.0.0/16
  ipipMode: Always
  natOutgoing: true
  nodeSelector: all()
  vxlanMode: Never
```

```
kubectl delete ippools.crd.projectcalico.org default-ipv4-ippool
kubectl apply -f ippool.yaml
```

The cluster should be in a stable state at this point. You will begin to see new pods with the updated IP range. The nodes may also have a secondary IP. This is ok. Now we can patch the controller nodes to remove the Flannel manifest.

```
cluster:
  network:
    cni:
      name: none
  proxy:
    disabled: true
```

```
talosctl patch mc --patch @cnipatch.yaml
```

Perform this step on all controllers. Then we can verify it worked by querying Talos controllers for their built-in manifest files. We do not want to see Flannel list anywhere.

```
talosctl get manifests -n <controller-ip>
```

Sample output:

NODE	NAMESPACE	TYPE	ID	VERSION
192.168.249.4	controlplane	Manifest	00-kubelet-bootstrapping-token	1
192.168.249.4	controlplane	Manifest	01-csr-approver-role-binding	1
192.168.249.4	controlplane	Manifest	01-csr-node-bootstrap	1
192.168.249.4	controlplane	Manifest	01-csr-renewal-role-binding	1
192.168.249.4	controlplane	Manifest	11-core-dns	1
192.168.249.4	controlplane	Manifest	11-core-dns-svc	1
192.168.249.4	controlplane	Manifest	11-kube-config-in-cluster	1

The final step is to remove the flannel daemonset and configmap

```
kubectrl delete daemonset -n kube-system kube-flannel  
kubectrl delete cm kube-flannel-cfg -n kube-system
```

## Conclusion

You may notice that some pods still have the old cidr range. This is mostly okay. I found that networking for the most part still works. Its best to go through the namespaces and do a rollout restart on the various resources and they will pick up the new networking rules.

# Cluster Setup

# Metal LB

This section is to get metallb setup and working for a bare metal setup.

Let create the namespace

```
kubectl create namespace metallb-system  
kubectl label namespace metallb-system pod-security.kubernetes.io/enforce=privileged
```

We will use helm for ease of upgrades and the initial install. First, we need to add the repo.

```
helm repo add metallb https://metallb.github.io/metallb
```

Now, install metallb with helm.

```
helm install metallb metallb/metallb -n metallb-system
```

We need to choose a pool of IP addresses that metal lb can hand out for the type LoadBalancer. In my case, I really just want this for nginx ingress. We need to create the following yaml file to apply to the API.

```
apiVersion: metallb.io/v1beta1  
kind: IPAddressPool  
metadata:  
  name: main-pool  
  namespace: metallb-system  
spec:  
  addresses:  
    - 192.168.249.10-192.168.249.11  
---  
apiVersion: metallb.io/v1beta1  
kind: L2Advertisement  
metadata:  
  name: l2-lb  
  namespace: metallb-system
```

Lets verify everything is up and running.

```
kubectrl get pods -n metallb-system
```



# nginx Ingress Controller

After installing metallb we can move on to the ingress part of setup. My preferred choice is nginx. We will also get this ready for monitoring with Prometheus.

Create the namespace.

```
kubectl create namespace ingress-nginx  
kubectl label namespace ingress-nginx pod-security.kubernetes.io/enforce=privileged
```

We will use helm to install nginx. This will help with upgrades in the future. Note the extra values for Prometheus.

```
helm upgrade --install ingress-nginx ingress-nginx --repo https://kubernetes.github.io/ingress-nginx --namespace  
ingress-nginx --set controller.metrics.enabled=true --set-string  
controller.podAnnotations."prometheus.io/scrape"="true" --set-string  
controller.podAnnotations."prometheus.io/port"="10254"
```

After a little bit of time, we can check the status of the namespace. The important part to note is the service/ingress-nginx-controller. It should have an External-IP provided by metallb.

```
kubectl get all -n ingress-nginx
```

## Example Output:

NAME	READY	STATUS	RESTARTS	AGE
pod/ingress-nginx-controller-8b8b9f598-jqxcr	1/1	Running	0	23m

  

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/ingress-nginx-controller	LoadBalancer	10.109.19.205	192.168.249.10	80:32588/TCP,443:30617/TCP	23m
service/ingress-nginx-controller-admission	ClusterIP	10.100.21.15	<none>	443/TCP	23m
service/ingress-nginx-controller-metrics	ClusterIP	10.107.12.193	<none>	10254/TCP	23m
service/prometheus-server	NodePort	10.97.78.93	<none>	9090:32631/TCP	5s

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/ingress-nginx-controller	1/1	1	1	23m

# Maintenance

# Convert PV from ReadWriteOnce to ReadWriteMany

## Change reclaim policy of the persistent volume to **Retain**

**This is required!!!** Before you delete the persistent volume claim to avoid a surprise that your data got wiped. The default reclaim policy is **Delete** and we do not want that to happen.

```
kubectrl patch pv grafana-pv -n grafana -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```

## Scale down any workloads (deployments, stateful sets etc etc) using the volume

In this step you are removing binds to the persistent volume. Check what is using it and scale it down to 0. No need to delete deployments, just scale them down to no pods running.

```
kubectrl scale --replicas=0 -n grafana deployment grafana
```

When all workloads using the persistent volume have been scaled down to zero you should see the status of it change from `Bound` to `Released`. We are not done yet though.

# Delete existing persistent volume claim to be able to free persistent volume

NOTE: **Did you back up its manifest just in case?**

```
kubectl delete pvc grafana-pvc -n grafana
```

# Free persistent volume status to become `Available`

Here we just remove the reference to the deleted persistent volume claim from the persistent volume so that a new persistent volume claim can be set.

```
kubectl patch pv grafana-pv -n grafana -p '{"spec":{"claimRef":{"uid":""}}}'
```

You should see that status of the persistent volume changed to `Available`

# Change the persistent volume access mode to `ReadWriteMany`

Just run this command:

```
kubectrl patch pv grafana-pv -n grafana -p '{"spec":{"accessModes":["ReadWriteMany"]}}'
```

# Recreate persistent volume claim with access mode ReadWriteMany

Here you either use your existing code to create this kubernetes resource with access mode `ReadWriteMany` or just use the backup we did in first step, edit its `spec.accessModes` and apply that.

I am just showing what the part of `spec` that is interesting to us should be, if you are editing existing manifest or backup, you will easily find the place to change it.

```
spec:
  accessModes:
    - ReadWriteMany
```

Apply the new persistent volume claim manifest (assuming we saved it to a file called `persistent_volume_claim_read_write_many.yaml`):

```
kubectrl -n grafana apply -f pvc.yaml
```

## Scale the workloads back up to start using the volume with new access mode.

Adjust number of replicas to what your workloads manifests actually specify.

```
kubectrl scale --replicas=1 -n grafana deployment grafana
```

The status of your persistent volume should become `Bound` now.

# Restore Longhorn PV from backup

Unfortunately, the longhorn UIs restore function does not work for in place backups. This means the PV needs to be deleted for the new volume to be created! As of this writing, there is a [Github issue](#) open from 2023 regarding this problem.

## Steps to restore from backup

1. Ensure the backup exists in the UI. Navigate to Volumes > <broken PV> > Backups. I open this in a new window. We need to take note of the PV and PVC names.
2. Scale down the deployment/statefulset using the PV.

```
kubectl scale statefulset <name> --replicas 0
```

3. Delete the PV/PVC from the longhorn UI
4. In the backup windows in the longhorn UI. Select Restore. Check the box "Use Previous Name"
5. Navigate back to volumes and find the restored PV. On the right dropdown click Create PV/PVC. It should ask to use the old information, if not we need to fill in the information from step 1.
6. Verify the PV/PVC exist and we can edit the PV and look for "volumeName" to match step 1.

```
spec:  
  storageClassName: longhorn  
  volumeMode: Filesystem  
  volumeName: pvc-<ID from step 1>
```

7. Scale the deployment back up

```
kubectl scale statefulset <name> --replicas 1
```