**Lab Guide: ETCD Backup and Restore in Kubernetes**

ETCD is the brain of Kubernetes. Every state change — Pods, ConfigMaps, Deployments, Secrets — is persisted inside ETCD. That’s why ETCD backup/restore is the single most important recovery procedure for cluster administrators.

In this lab, you will:

1. Deploy some test workloads.
2. Back up the ETCD datastore.
3. Simulate failure by deleting resources.
4. Restore ETCD from the snapshot.
5. Verify that workloads are back.

**Lab Setup**

* Kubernetes cluster (kubeadm based, single control plane).
* Access to master/control plane node with root privileges.
* etcdctl client installed.

**Step 1: Create Sample Workloads**

Let’s create some random pods to see the effect of ETCD restore.

kubectl create namespace raman

kubectl run test-app --image=nginx -n raman --replicas=3

kubectl create deployment test-app2 --image=nginx -n raman --replicas=3

kubectl create deployment raman-nfs --image=busybox -n raman -- sleep 3600

Check pods:

kubectl get pods -n raman

You should see pods similar to:

NAME READY STATUS RESTARTS AGE

raman-nfs-xxxxx 0/1 Completed 0 22h

test-app-xxxxx 1/1 Running 0 19h

test-app2-xxxxx 1/1 Running 0 19h

This is your **prime state**.

**Step 2: Install etcdctl Client**

On the control plane node:

sudo apt update

sudo apt install etcd-client -y

Enable API v3:

export ETCDCTL\_API=3

**Step 3: Locate Certificates and Configs**

ETCD is secured with TLS, so we need the right certs.

Find Kubelet config:

ps -ef | grep kubelet

cd /var/lib/kubelet

cat config.yaml

Find ETCD process and confirm its data dir:

ps -ef | grep etcd

You’ll see something like:

--data-dir=/var/lib/etcd

So the original ETCD data directory is /var/lib/etcd.

Certificates are usually in /etc/kubernetes/pki/etcd/.

**Step 4: Take an ETCD Snapshot (Backup)**

Run:

etcdctl --endpoints=https://127.0.0.1:2379 \

--cacert=/etc/kubernetes/pki/etcd/ca.crt \

--cert=/etc/kubernetes/pki/etcd/server.crt \

--key=/etc/kubernetes/pki/etcd/server.key \

snapshot save /root/myclust.db

Check the snapshot metadata:

etcdctl snapshot status /root/myclust.db -w table

**Step 5: Simulate a Failure**

Delete all pods in raman namespace:

kubectl delete pods --all -n raman

If you list pods now, everything’s gone. This is your **failure state**.

**Step 6: Restore ETCD Snapshot**

**6.1 Restore snapshot into a new directory**

Do not manually create the directory; etcdctl will handle it.

etcdctl --data-dir /var/lib/etcd-new snapshot restore /root/myclust.db

This creates a new folder /var/lib/etcd-new containing restored ETCD data.

**6.2 Update ETCD Manifest**

Kubeadm clusters run ETCD as a **static pod**, defined in /etc/kubernetes/manifests/etcd.yaml.  
Edit this file:

vi /etc/kubernetes/manifests/etcd.yaml

Find the **volumes** section, and update the hostPath for ETCD data:

volumes:

- hostPath:

path: /etc/kubernetes/pki/etcd

type: DirectoryOrCreate

name: etcd-certs

- hostPath:

path: /var/lib/etcd-new

type: DirectoryOrCreate

name: etcd-data

Save and exit.

Since /etc/kubernetes/manifests/ is watched by kubelet, the ETCD static pod will restart automatically with the new data directory.

**Step 7: Verification**

Give it a few seconds, then check ETCD pod health:

kubectl get pods -n kube-system | grep etcd

Once ETCD is running, check your workloads:

kubectl get pods -n raman

Expected result: All pods/deployments you had before deletion are back — restored from snapshot.

**Step 8: Cleanup (Optional)**

If needed, revert to original data directory:

* Change back /var/lib/etcd in the ETCD manifest.
* Remove /var/lib/etcd-new if not required.

**Key Takeaways**

1. **ETCD snapshot = entire cluster state.**  
   Restoring snapshot is not about restoring “a pod” but reverting the **entire cluster state** at snapshot time.
2. **Static Pod Manifests** are the lever.  
   Kubelet reads /etc/kubernetes/manifests. Any changes here directly restart system pods (apiserver, controller, scheduler, etcd).
3. Always restore to a **new data directory**. Don’t overwrite /var/lib/etcd.
4. Pod recreation after restore is proof that cluster state is back.

This lab not only covers the mechanics but also shows **why** we do things: you see the cluster forget its workloads and then resurrect them from the ETCD snapshot.

**Lab Guide: Kubernetes Cluster Upgrade with kubeadm (v1.29 → v1.30)**

This lab shows how to perform a safe, rolling upgrade of a kubeadm-based Kubernetes cluster from **v1.29.x** to **v1.30.0**.

We’ll upgrade:

* The **control plane node** (master).
* The **worker nodes** (w1, w2).

**Pre-Requisites**

* A working kubeadm cluster:
  + 1 control plane node (master)
  + 2 worker nodes (w1, w2)
* Current versions:
  + master: v1.29.x
  + w1, w2: v1.29.x

Check current versions:

kubectl get nodes -o wide

**Step 1: Prepare the APT Repository**

Kubernetes now uses **pkgs.k8s.io** for package distribution. On **all nodes** (master + workers), configure the v1.30 repo:

sudo mkdir -p /etc/apt/keyrings

curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.30/deb/Release.key | \

sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg

sudo tee /etc/apt/sources.list.d/kubernetes.list <<EOF

deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.30/deb/ /

EOF

**Step 2: Upgrade the Control Plane**

**2.1 Upgrade kubeadm on master**

sudo apt-mark unhold kubeadm && \

sudo apt-get update && sudo apt-get install -y kubeadm=1.30.0-1.1 && \

sudo apt-mark hold kubeadm

Verify:

kubeadm version

**2.2 Plan the upgrade**

This checks versions and compatibility.

kubeadm upgrade plan --ignore-preflight-errors=CoreDNSUnsupportedPlugins,CoreDNSMigration

**2.3 Apply the upgrade**

Perform the real control plane upgrade:

sudo kubeadm upgrade apply v1.30.0 --ignore-preflight-errors=CoreDNSUnsupportedPlugins,CoreDNSMigration

This updates:

* API server, controller-manager, scheduler, etcd (if managed by kubeadm).
* CoreDNS, kube-proxy (if applicable).

**2.4 Upgrade kubelet and kubectl on master**

sudo apt-mark unhold kubelet kubectl && \

sudo apt-get update && sudo apt-get install -y kubelet=1.30.0-1.1 kubectl=1.30.0-1.1 && \

sudo apt-mark hold kubelet kubectl

sudo systemctl daemon-reload

sudo systemctl restart kubelet

Verify:

kubectl get nodes

You should see the master at v1.30.0, workers still at v1.29.x.

NAME STATUS ROLES AGE VERSION

master Ready control-plane 3d23h v1.30.0

w1 Ready <none> 3d22h v1.29.15

w2 Ready <none> 3d22h v1.29.15

**Step 3: Upgrade Worker Nodes**

We’ll do this one node at a time to avoid downtime.

**3.1 Upgrade kubeadm on worker (w1)**

sudo apt-mark unhold kubeadm && \

sudo apt-get update && sudo apt-get install -y kubeadm=1.30.0-1.1 && \

sudo apt-mark hold kubeadm

Verify:

kubeadm version

**3.2 Drain the node from master**

Before upgrading, drain the workloads to other nodes:

kubectl drain w1 --ignore-daemonsets

**3.3 Perform node upgrade**

On w1:

sudo kubeadm upgrade node

**3.4 Upgrade kubelet and kubectl on w1**

sudo apt-mark unhold kubelet kubectl && \

sudo apt-get update && sudo apt-get install -y kubelet=1.30.0-1.1 kubectl=1.30.0-1.1 && \

sudo apt-mark hold kubelet kubectl

sudo systemctl daemon-reload

sudo systemctl restart kubelet

**3.5 Uncordon the node**

Bring it back into service:

kubectl uncordon w1

**3.6 Verify**

kubectl get nodes

Expected:

NAME STATUS ROLES AGE VERSION

master Ready control-plane 3d23h v1.30.0

w1 Ready <none> 3d22h v1.30.0

w2 Ready <none> 3d22h v1.29.15

**3.7 Repeat for other workers**

Perform steps **3.1 – 3.6** for w2 and any other worker nodes.

**Step 4: Post-Upgrade Validation**

Check cluster components:

kubectl get nodes

kubectl get pods -A

kubectl get cs # componentstatus

Check versions:

kubectl version --short

All nodes should now be v1.30.0.

**Key Notes & Best Practices**

* Always **back up ETCD** before an upgrade (link back to your ETCD lab).
* Upgrade is always **sequential** (master → workers).
* Do **not** upgrade all workers at once; drain and upgrade one by one.
* Keep kubeadm, kubectl, and kubelet versions in sync with cluster version.
* If you skip a version, kubeadm will block you. Upgrades must be **minor version by minor version** (e.g., 1.28 → 1.29 → 1.30).

This lab gives you the **hands-on rolling upgrade workflow** exactly as it’s done in production clusters.

**Lab Guide: Kubernetes Monitoring with Prometheus & Grafana (kube-prometheus-stack)**

**Objective**

By the end of this lab you will:

* Deploy the Prometheus + Grafana monitoring stack using Helm.
* Expose Prometheus and Grafana via NodePort services.
* Run PromQL queries in Prometheus.
* Log into Grafana and add pre-built Kubernetes dashboards.

**Lab Setup**

* Kubernetes cluster (created with kubeadm or managed).
* Nodes must be at least **t3.large** (2 vCPU, 8GB RAM) to handle Prometheus stack load.
* kubectl configured to access the cluster.
* helm installed (we’ll install it in Step 1).

**Step 1: Install Helm**

Helm is the package manager for Kubernetes, simplifying complex deployments.

curl -fsSL -o get\_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3

chmod 700 get\_helm.sh

./get\_helm.sh

Verify:

helm version

**Step 2: Add Prometheus Community Helm Repo**

helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm repo update

helm search repo prometheus-community/kube-prometheus-stack

**Step 3: Create Namespace for Monitoring**

kubectl create ns monitoring

**Step 4: Install kube-prometheus-stack**

We’ll install version **45.7.1**, disabling ETCD monitoring (since it requires special certs).

helm install prometheus prometheus-community/kube-prometheus-stack \

--version 45.7.1 \

--namespace monitoring \

--set kubeEtcd.enabled=false

Check deployments:

helm list -n monitoring

kubectl get pods -n monitoring

**Step 5: Expose Prometheus**

By default, Prometheus is a headless service (clusterIP: None). We’ll expose it via **NodePort**.

Edit the service:

kubectl edit svc -n monitoring prometheus-operated

Change:

type: NodePort

Remove these lines:

clusterIP: None

clusterIPs:

- None

Save and exit.

If your edit gets saved to a temporary file, apply it back:

kubectl replace --force -f /tmp/kubectl-edit-xxxx.yaml

Check service:

kubectl get svc -n monitoring

You’ll see a NodePort, e.g.:

prometheus-operated NodePort 10.96.2.15 <none> 9090:31585/TCP 5m

Now browse:  
http://<NodeIP>:31585

**Step 6: Try Prometheus Queries**

In the Prometheus UI → **Graph**:

1. **Check running pods**
2. kube\_pod\_container\_status\_running
3. **Check CPU usage of a node**  
   Replace instance with your node’s exporter IP.
4. 100 \* (1 - avg(rate(node\_cpu\_seconds\_total{mode="idle", instance="172.31.27.229:9100"}[5m])))

**Step 7: Expose Grafana Dashboard**

Edit Grafana service:

kubectl edit svc -n monitoring prometheus-grafana

Change type: ClusterIP → NodePort.

Check service:

kubectl get svc -n monitoring

You’ll see a NodePort (e.g., 32000).

Browse Grafana at:  
http://<NodeIP>:<NodePort>

**Step 8: Grafana Login**

Default credentials:

* **Username:** admin
* **Password:** retrieved from secret

kubectl get secret -n monitoring prometheus-grafana -o jsonpath="{.data.admin-password}" | base64 --decode; echo

Log in with these credentials.

**Step 9: Import Kubernetes Dashboards**

Grafana comes pre-loaded with dashboards, but you can install richer community dashboards.

Two good sources:

1. [dotdc Kubernetes Grafana Dashboards](https://github.com/dotdc/grafana-dashboards-kubernetes?tab=readme-ov-file#install-via-grafanacom)
2. [0xdc Modern Dashboards for Kubernetes](https://0xdc.me/blog/a-set-of-modern-grafana-dashboards-for-kubernetes/)

In Grafana → **Dashboards → Import**:

* Use dashboard IDs from Grafana.com, or
* Upload JSON from GitHub repos.

Examples:

* Node Exporter Full (ID: 1860)
* Kubernetes / Views / Namespaces (ID: 15758)
* Kubernetes / Cluster Monitoring (ID: 15759)

**Step 10: Validate Monitoring**

1. View Prometheus metrics:  
   http://<NodeIP>:31585
2. View Grafana dashboards:  
   http://<NodeIP>:<Grafana-NodePort>
   * Cluster health
   * Pod status
   * Node CPU, memory, disk
3. Scale a deployment:

kubectl scale deploy test-app --replicas=5 -n raman

Then check Grafana → the pod metrics should update in near real-time.

**Key Takeaways**

* **Prometheus** is the metrics collection + query engine (time-series database).
* **Grafana** is the visualization frontend.
* kube-prometheus-stack bundles Prometheus, Grafana, Alertmanager, and exporters.
* NodePorts make quick lab access easy, but in production you’d prefer **Ingress** or **LoadBalancer**.
* Community dashboards accelerate setup, showing cluster, node, pod, and namespace health.

This setup is the foundation of Kubernetes monitoring. From here you can layer in:

* **Alertmanager** (alerts on Slack, email, PagerDuty).
* **Persistent volumes** for Prometheus (so metrics survive pod restarts).
* **Ingress with TLS** for secure dashboard access.