Conquer The CKA Exam

Practical Hands-On Exercises with Explanations

Author: DevOpsDynamo

December 2025

Contents

1 Introduction **4**

2 Kubernetes Fundamentals **5**
2.1 What Kubernetes Is and Why It Matters . . . 5

3 Why Kubernetes Changes the Game **6**

4 Practical Scenarios **7**
4.1 Scenario 1: Upgrading a Kubernetes Cluster from v1.34.0 to v1.34.1 . . . 8
4.1.1 Lab vs. Exam Requirements . . . 8
4.1.2 Version Discovery . . . 9
4.1.3 Control Plane Upgrade: `controlplane` . . . 9
4.1.4 Worker Node Upgrade: `node01` . . . 11
4.1.5 Verification . . . 13
4.1.6 Success Criteria . . . 13
4.2 Scenario 2: Creating and Using a Persistent Volume (PV) with a Persistent Volume Claim (PVC) . . . 14
4.2.1 Lab vs. Exam Requirements . . . 14
4.2.2 Create PV, PVC, and Pod . . . 15
4.2.3 Verification . . . 16
4.2.4 Explanation of Key Concepts . . . 16
4.2.5 Success Criteria . . . 17
4.3 Scenario 3: Draining a Node for Maintenance and Rescheduling Pods . . . 18
4.3.1 Lab vs. Exam Requirements . . . 18
4.3.2 Step-by-step Instructions . . . 18
4.3.3 Explanation of Key Commands . . . 21
4.3.4 Success Criteria . . . 21
4.4 Scenario 4: Configuring an Ingress with TLS for a Production Service . . . 22
4.5 Scenario 5: Deploying a StatefulSet with Persistent Storage for a Database 24
4.6 Scenario 6: Backing Up and Restoring etcd with TLS Certificates . . . 27
4.6.1 Exam Strategy Overview . . . 27
4.6.2 Backup Process . . . 28
4.6.3 Restore Process . . . 28
4.6.4 Validation . . . 29
4.6.5 Simulation Notes (Real Lab Behavior) . . . 30
4.6.6 Exam Day Checklist . . . 30
4.7 Scenario 7: Creating a Custom ClusterRole and Binding It to a Namespace-Scoped ServiceAccount . . . 31
4.7.1 Exam Strategy Overview . . . 31
4.7.2 ClusterRole Creation . . . 32
4.7.3 ServiceAccount Creation . . . 32
4.7.4 Binding the Role to the ServiceAccount . . . 33
4.7.5 Verification of Objects . . . 33
4.7.6 Testing Permissions (Recommended in Lab and Exam) . . . 34
4.7.7 Success Criteria . . . 35
4.8 Scenario 8: Enforcing Pod Placement with Node Affinity and Taints . . . 36

5 Reinforcement Exercises: Network Policies **38**
5.1 Exercise 1: Namespace-Specific Policies 38
5.2 Exercise 2: Cross-Namespace Traffic 40

6 StorageClass and PersistentVolumeClaim Exercises **42**
6.1 Exercise 1: Creating a Custom StorageClass with PersistentVolumeClaim 42
6.2 Exercise 2: Deploying an Application with PVC 44
6.3 Exercise 3: Expanding a PersistentVolumeClaim 45

7 Additional Scenarios for Certification and Troubleshooting **47**
7.1 Certificate Management Exercises 47
7.1.1 Scenario 1: Using Self-Signed Certificates in a Pod 47
7.1.2 Scenario 2: Automating Certificate Issuance with cert-manager . . 49
7.2 Troubleshooting Exercises 51
7.2.1 Scenario 1: Pod in `CrashLoopBackOff` State 51
7.2.2 Scenario 2: Node in `NotReady` State 51
7.2.3 Scenario 3: Node Reporting `DiskPressure` 52

8 Other Advanced Kubernetes Exercises for CKA® Preparation **53**
8.1 Exercise 1: Implementing Resource Quotas in a Namespace 53
8.2 Exercise 2: Configuring Node Affinity for Pod Placement 55

9 Advanced Kubernetes Scheduling and Configuration Exercises **56**
9.1 Exercise 1: Implementing Horizontal Pod Autoscaler (HPA) 56
9.2 Exercise 2: Taints and Tolerations 57

10 Conclusion **59**

11 References **60**

Disclaimer

1 Introduction

Why This Book Exists

When I prepared for the CKA, I wasted weeks bouncing between outdated blog posts, random YouTube videos, and forum debates. Everyone had tips, but almost nobody had realistic, command-driven tasks that matched the exam. So I built my own practice system. It worked: I passed with confidence. This book is that same system, refined for you.

The CKA exam is hands-on, time-pressured, and unforgiving of hesitation. This is not a “read and memorize” kind of preparation. Here, you’ll train with **real-world labs** that match both the exam and production Kubernetes challenges.

Who This Book Is For

- DevOps engineers validating Kubernetes skills
- Sysadmins and cloud architects managing container workloads
- Developers moving into deployment and infrastructure work
- Self-learners tired of passive learning
- IT pros aiming to pass the CKA and apply it on the job

If you want speed, accuracy, and confidence under pressure: you’re in the right place.

2 Kubernetes Fundamentals

2.1 What Kubernetes Is and Why It Matters

Kubernetes is the control system for modern containerized workloads. It handles deployment, scaling, failover, service discovery, and traffic routing automatically. You declare how things should run, Kubernetes keeps them that way.

Core Strengths:

- **Scalability:** Grow or shrink workloads based on demand
- **Self-Healing:** Restart failed containers automatically
- **Load Balancing:** Distribute requests to avoid bottlenecks
- **Zero-Downtime Updates:** Roll forward or back without service impact
- **Smart Scheduling:** Place workloads where resources are available

In Practice

Kubernetes frees developers to focus on code and gives operations teams predictable, repeatable deployments. It is the common ground between speed and stability.

3 Why Kubernetes Changes the Game

1. **Run Anywhere:** On-prem, AWS, GCP, Azure, or all at once, Kubernetes keeps the workflow consistent.
2. **Scale Without Pain:** Add pods for horizontal growth or boost CPU/RAM for vertical scaling.
3. **Use Every Bit of Hardware:** Kubernetes schedules workloads efficiently, cutting waste.
4. **Security From the Start:** RBAC, secrets management, and network policies come built in.

4 Practical Scenarios

The Certified Kubernetes Administrator (CKA) exam tests not just what you know, but how fast and accurately you can apply it. Every scenario in this section is mapped to one of the **five official exam domains**:

- **Cluster Architecture, Installation Configuration** : 25%
- **Workloads Scheduling** : 15%
- **Services Networking** : 20%
- **Storage** : 10%
- **Troubleshooting** : 30%

Some tasks will push your speed, others will challenge your accuracy under pressure. All are modeled on real-world Kubernetes operations you'll face both in the exam and in production.

If you can, set up a practice environment with **Minikube** or **kubeadm** and work through these scenarios step-by-step. Hands-on repetition will burn these patterns into your memory. In the exam, the official Kubernetes documentation will be available, all solutions here are built on top of it.

Tip

Do not just read these scenarios. Visualize the cluster: the pods, services, volumes, and the control-plane components that keep it running. Mental simulation now saves precious seconds in the exam.

By the end, you will not only know the commands, but also:

- When to run them
- How to verify they worked
- How to fix them when they don't

4.1 Scenario 1: Upgrading a Kubernetes Cluster from v1.34.0 to v1.34.1

Domain: Cluster Architecture, Installation & Configuration (25%)

Objective:
Upgrade the Kubernetes cluster from version `v1.34.0` to `v1.34.1` using `kubeadm` while maintaining cluster availability. Perform upgrades in the correct order:

1. Upgrade **kubeadm**
2. Upgrade **control plane** using `kubeadm upgrade apply` (**no drain**)
3. Drain control plane only when upgrading **kubelet/kubectl**
4. Upgrade workers one by one using **drain** → **upgrade** → **uncordon**

Cluster Topology (Lab):

- Control plane: `controlplane`
- Worker: `node01`
- OS: Ubuntu 22.04
- Runtime: containerd
- Initial versions: `v1.34.0`

Always Remember

- **Do not drain before `kubeadm upgrade apply`.**
- **You must drain before upgrading kubelet/kubectl.**
- Workers are upgraded **one at a time**.
- Use `apt-mark hold/unhold` to pin component versions.

4.1.1 Lab vs. Exam Requirements

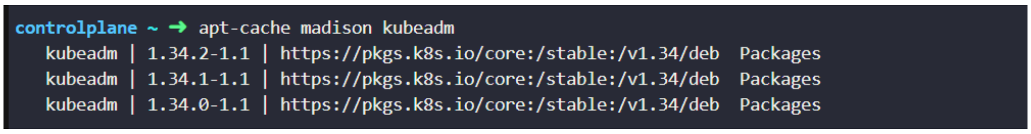
In a lab environment (local VMs, Minikube, Kubeadm, KodeKloud), repository setup and package cache refresh may be required. In the CKA exam, these are already configured.

Task	Lab	Exam
Add Kubernetes APT repo	✓	×
Run `apt update`	✓	×
Check versions with `apt-cache madison`	✓	✓
Upgrade `kubeadm`	✓	✓
Run `kubeadm upgrade apply`	✓	✓
Drain only before kubelet	✓	✓
Worker: drain → upgrade → uncordon	✓	✓

4.1.2 Version Discovery

Check available versions:

```
apt-cache madison kubeadm
```



```
controlplane ~ ➜ apt-cache madison kubeadm
   kubeadm | 1.34.2-1.1 | https://pkgs.k8s.io/core:/stable:/v1.34/deb  Packages
   kubeadm | 1.34.1-1.1 | https://pkgs.k8s.io/core:/stable:/v1.34/deb  Packages
   kubeadm | 1.34.0-1.1 | https://pkgs.k8s.io/core:/stable:/v1.34/deb  Packages
```

Expected:

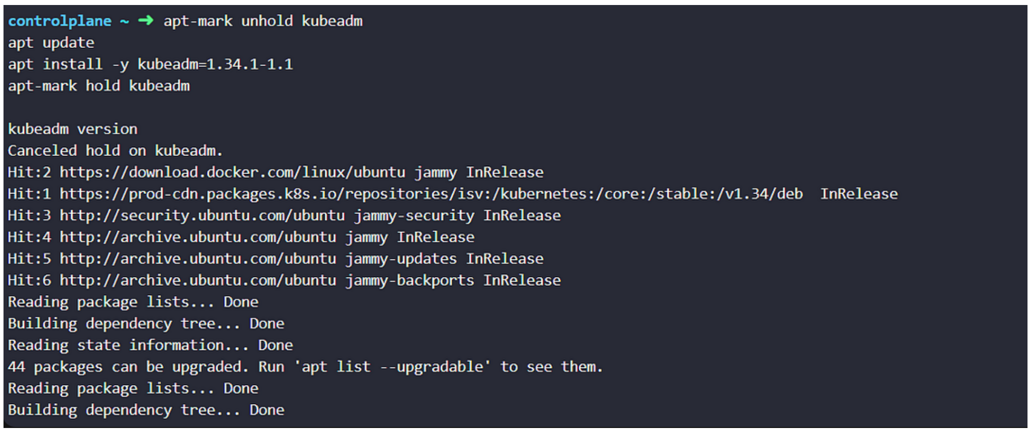
```
kubeadm | 1.34.1-1.1
kubeadm | 1.34.0-1.1
```

Target upgrade version: `1.34.1`.

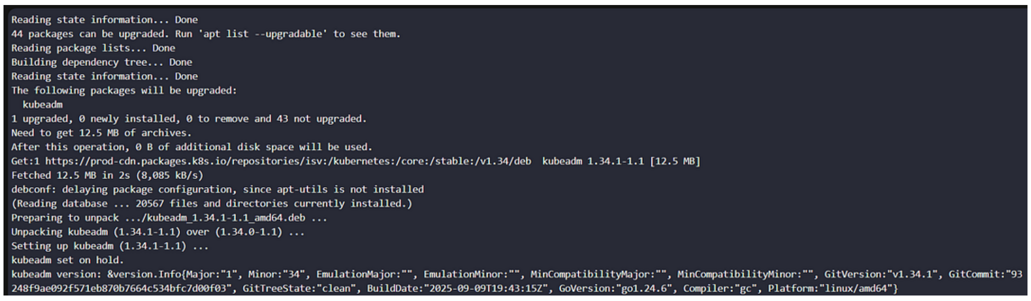
4.1.3 Control Plane Upgrade: `controlplane`

Step 1: Upgrade kubeadm

```
apt-mark unhold kubeadm
apt install -y kubeadm=1.34.1-1.1
apt-mark hold kubeadm

kubeadm version
```



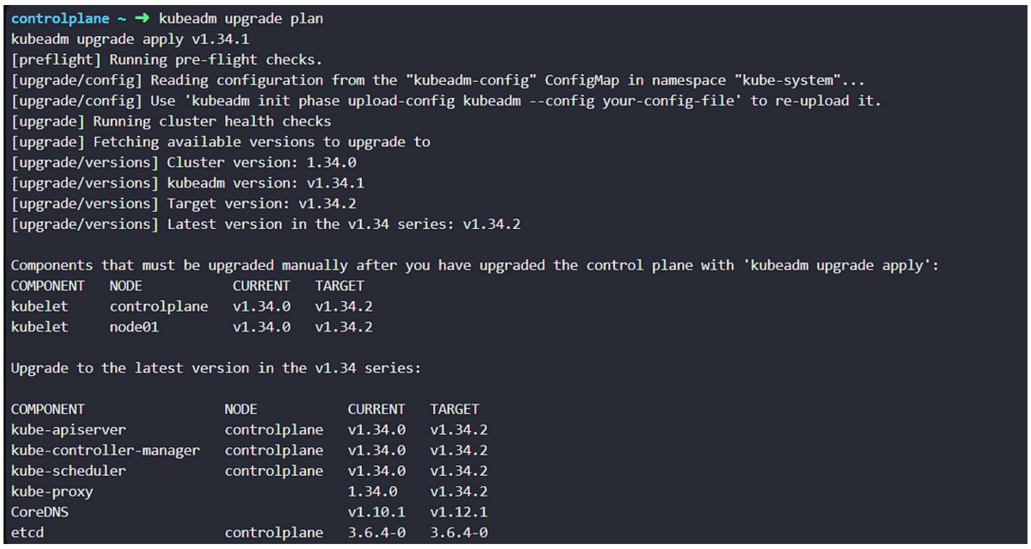
```
controlplane ~ ➜ apt-mark unhold kubeadm
apt update
apt install -y kubeadm=1.34.1-1.1
apt-mark hold kubeadm

kubeadm version
Canceled hold on kubeadm.
Hit:2 https://download.docker.com/linux/ubuntu jammy InRelease
Hit:1 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.34/deb  InRelease
Hit:3 http://security.ubuntu.com/ubuntu jammy-security InRelease
Hit:4 http://archive.ubuntu.com/ubuntu jammy InRelease
Hit:5 http://archive.ubuntu.com/ubuntu jammy-updates InRelease
Hit:6 http://archive.ubuntu.com/ubuntu jammy-backports InRelease
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
44 packages can be upgraded. Run 'apt list --upgradable' to see them.
Reading package lists... Done
Building dependency tree... Done
```



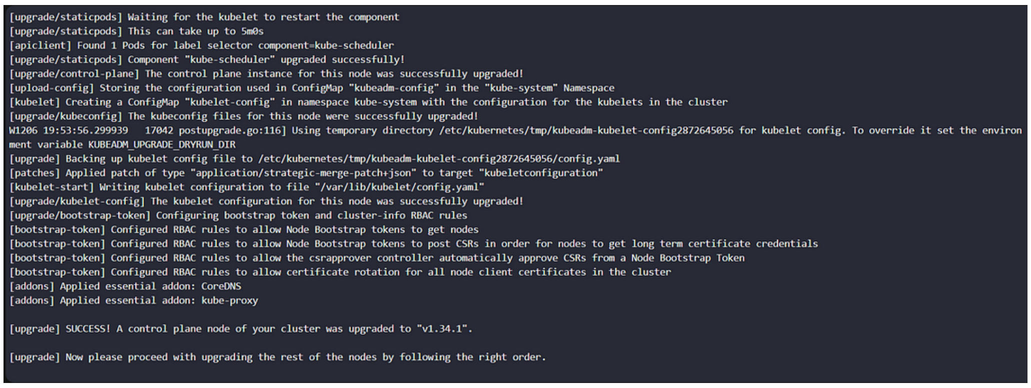
```
Reading state information... Done
44 packages can be upgraded. Run 'apt list --upgradable' to see them.
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following packages will be upgraded:
  kubeadm
1 upgraded, 0 newly installed, 0 to remove and 43 not upgraded.
Need to get 12.5 MB of archives.
After this operation, 0 B of additional disk space will be used.
Get:1 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.34/deb  kubeadm 1.34.1-1.1 [12.5 MB]
Fetched 12.5 MB in 2s (8,085 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
(Reading database ... 20567 files and directories currently installed.)
Preparing to unpack .../kubeadm_1.34.1-1.1_amd64.deb ...
Unpacking kubeadm (1.34.1-1.1) over (1.34.0-1.1) ...
Setting up kubeadm (1.34.1-1.1) ...
kubeadm set on hold.
kubeadm version: &version.Info{Major:"1", Minor:"34", EmulationMajor:"", EmulationMinor:"", MinCompatibilityMajor:"", MinCompatibilityMinor:"", GitVersion:"v1.34.1", GitCommit:"93
248f9ae092f571eb870b7664c534bfc7d00f03", GitTreeState:"clean", BuildDate:"2025-09-09T19:43:15Z", GoVersion:"go1.24.6", Compiler:"gc", Platform:"linux/amd64"}
```

Step 2: Plan and Apply Control Plane Upgrade (No Drain)

```
kubeadm upgrade plan
kubeadm upgrade apply v1.34.1
```



```
controlplane ~ ➜ kubeadm upgrade plan
kubeadm upgrade apply v1.34.1
[preflight] Running pre-flight checks.
[upgrade/config] Reading configuration from the "kubeadm-config" ConfigMap in namespace "kube-system"...
[upgrade/config] Use 'kubeadm init phase upload-config kubeadm --config your-config-file' to re-upload it.
[upgrade] Running cluster health checks
[upgrade] Fetching available versions to upgrade to
[upgrade/versions] Cluster version: 1.34.0
[upgrade/versions] kubeadm version: v1.34.1
[upgrade/versions] Target version: v1.34.2
[upgrade/versions] Latest version in the v1.34 series: v1.34.2

Components that must be upgraded manually after you have upgraded the control plane with 'kubeadm upgrade apply':
COMPONENT   NODE           CURRENT   TARGET
kubelet     controlplane   v1.34.0   v1.34.2
kubelet     node01         v1.34.0   v1.34.2

Upgrade to the latest version in the v1.34 series:

COMPONENT                 NODE           CURRENT   TARGET
kube-apiserver            controlplane   v1.34.0   v1.34.2
kube-controller-manager   controlplane   v1.34.0   v1.34.2
kube-scheduler            controlplane   v1.34.0   v1.34.2
kube-proxy                               1.34.0    v1.34.2
CoreDNS                                  v1.10.1   v1.12.1
etcd                      controlplane   3.6.4-0   3.6.4-0
```

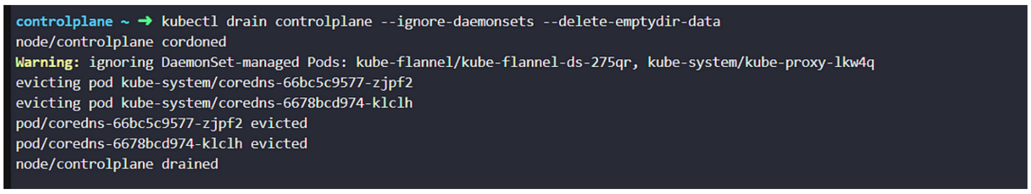


```
[upgrade/staticpods] Waiting for the kubelet to restart the component
[upgrade/staticpods] This can take up to 5m0s
[apiclient] Found 1 Pods for label selector component=kube-scheduler
[upgrade/staticpods] Component "kube-scheduler" upgraded successfully!
[upgrade/control-plane] The control plane instance for this node was successfully upgraded!
[upload-config] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-system" Namespace
[kubelet] Creating a ConfigMap "kubelet-config" in namespace kube-system with the configuration for the kubelets in the cluster
[upgrade/kubeconfig] The kubeconfig files for this node were successfully upgraded!
W1206 19:53:56.299939   17042 postupgrade.go:116] Using temporary directory /etc/kubernetes/tmp/kubeadm-kubelet-config2872645056 for kubelet config. To override it set the environ
ment variable KUBEADM_UPGRADE_DRYRUN_DIR
[upgrade] Backing up kubelet config file to /etc/kubernetes/tmp/kubeadm-kubelet-config2872645056/config.yaml
[patches] Applied patch of type "application/strategic-merge-patch+json" to target "kubeletconfiguration"
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[upgrade/kubelet-config] The kubelet configuration for this node was successfully upgraded!
[upgrade/bootstrap-token] Configuring bootstrap token and cluster-info RBAC rules
[bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to get nodes
[bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order for nodes to get long term certificate credentials
[bootstrap-token] Configured RBAC rules to allow the csrapprover controller automatically approve CSRs from a Node Bootstrap Token
[bootstrap-token] Configured RBAC rules to allow certificate rotation for all node client certificates in the cluster
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

[upgrade] SUCCESS! A control plane node of your cluster was upgraded to "v1.34.1".

[upgrade] Now please proceed with upgrading the rest of the nodes by following the right order.
```

Important: The control plane is **not drained yet**. Static pod upgrades are safe.

Step 3: Drain Control Plane Before Kubelet Upgrade

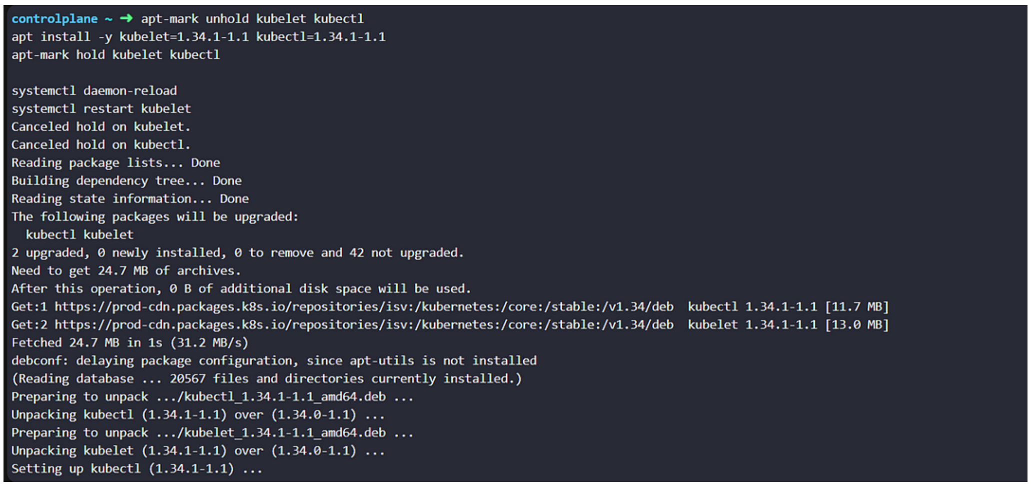
```
kubectl drain controlplane --ignore-daemonsets --delete-emptydir-data
```



```
controlplane ~ ➜ kubectl drain controlplane --ignore-daemonsets --delete-emptydir-data
node/controlplane cordoned
Warning: ignoring DaemonSet-managed Pods: kube-flannel/kube-flannel-ds-275qr, kube-system/kube-proxy-lkw4q
evicting pod kube-system/coredns-66bc5c9577-zjpf2
evicting pod kube-system/coredns-6678bcd974-klclh
pod/coredns-66bc5c9577-zjpf2 evicted
pod/coredns-6678bcd974-klclh evicted
node/controlplane drained
```

Step 4: Upgrade kubelet and kubectl

```
apt-mark unhold kubelet kubectl
apt install -y kubelet=1.34.1-1.1 kubectl=1.34.1-1.1
apt-mark hold kubelet kubectl

systemctl daemon-reload
systemctl restart kubelet
```



```
controlplane ~ ➜ apt-mark unhold kubelet kubectl
apt install -y kubelet=1.34.1-1.1 kubectl=1.34.1-1.1
apt-mark hold kubelet kubectl

systemctl daemon-reload
systemctl restart kubelet
Canceled hold on kubelet.
Canceled hold on kubectl.
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following packages will be upgraded:
  kubectl kubelet
2 upgraded, 0 newly installed, 0 to remove and 42 not upgraded.
Need to get 24.7 MB of archives.
After this operation, 0 B of additional disk space will be used.
Get:1 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.34/deb  kubectl 1.34.1-1.1 [11.7 MB]
Get:2 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.34/deb  kubelet 1.34.1-1.1 [13.0 MB]
Fetched 24.7 MB in 1s (31.2 MB/s)
debconf: delaying package configuration, since apt-utils is not installed
(Reading database ... 20567 files and directories currently installed.)
Preparing to unpack .../kubectl_1.34.1-1.1_amd64.deb ...
Unpacking kubectl (1.34.1-1.1) over (1.34.0-1.1) ...
Preparing to unpack .../kubelet_1.34.1-1.1_amd64.deb ...
Unpacking kubelet (1.34.1-1.1) over (1.34.0-1.1) ...
Setting up kubectl (1.34.1-1.1) ...
```

Step 5: Uncordon Control Plane

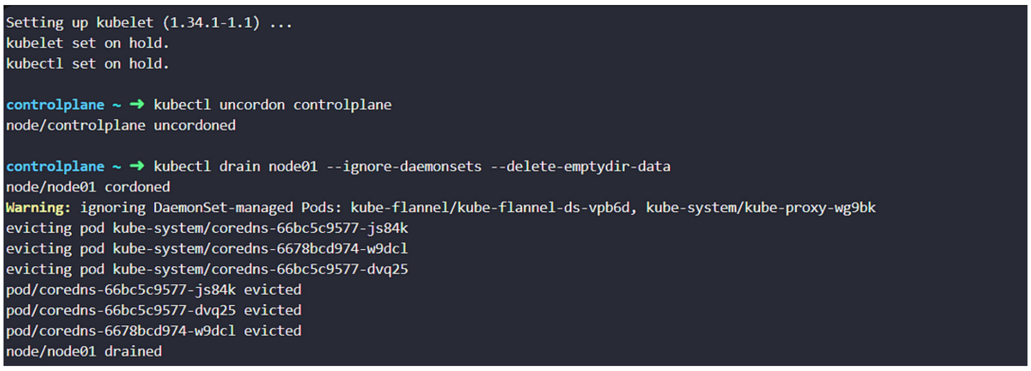
```
kubectl uncordon controlplane
```

4.1.4 Worker Node Upgrade: `node01`

Workers are upgraded one at a time.

Step 6: Drain Worker

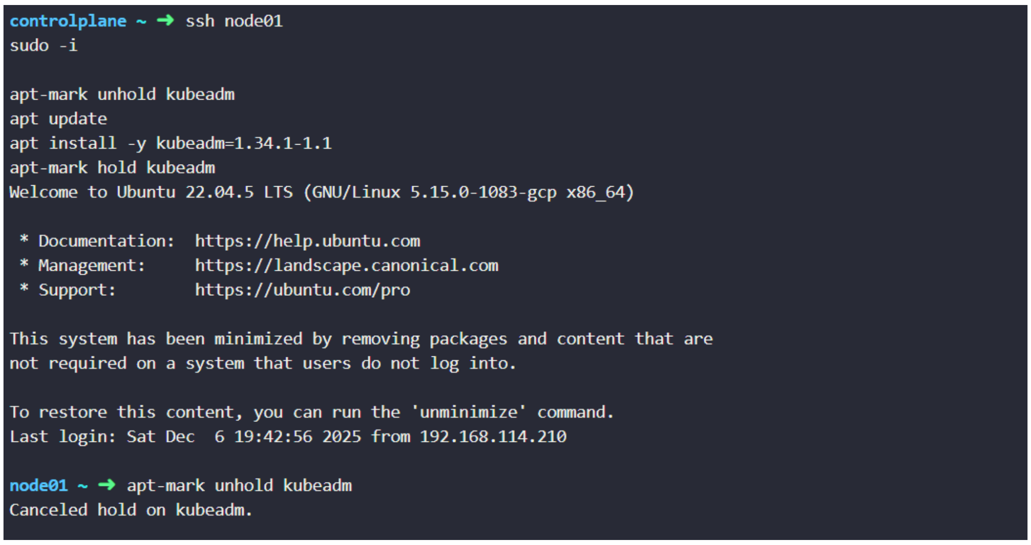
```
kubectl drain node01 --ignore-daemonsets --delete-emptydir-data
```



```
Setting up kubelet (1.34.1-1.1) ...
kubelet set on hold.
kubectl set on hold.

controlplane ~ ➜ kubectl uncordon controlplane
node/controlplane uncordoned

controlplane ~ ➜ kubectl drain node01 --ignore-daemonsets --delete-emptydir-data
node/node01 cordoned
Warning: ignoring DaemonSet-managed Pods: kube-flannel/kube-flannel-ds-vpb6d, kube-system/kube-proxy-wg9bk
evicting pod kube-system/coredns-66bc5c9577-js84k
evicting pod kube-system/coredns-6678bcd974-w9dcl
evicting pod kube-system/coredns-66bc5c9577-dvq25
pod/coredns-66bc5c9577-js84k evicted
pod/coredns-66bc5c9577-dvq25 evicted
pod/coredns-6678bcd974-w9dcl evicted
node/node01 drained
```

Step 7: Upgrade kubeadm

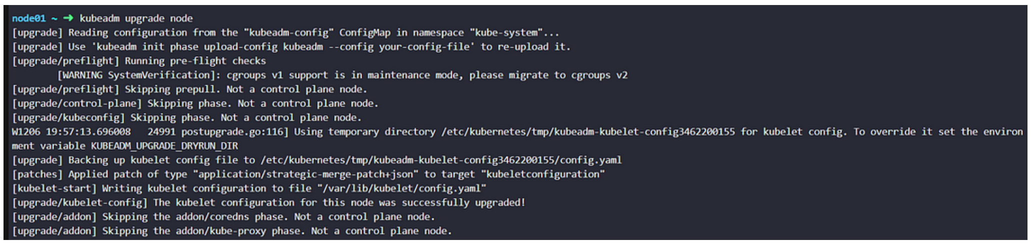
```
ssh node01
sudo -i

apt-mark unhold kubeadm
apt install -y kubeadm=1.34.1-1.1
apt-mark hold kubeadm
```



```
controlplane ~ ➜ ssh node01
sudo -i

apt-mark unhold kubeadm
apt update
apt install -y kubeadm=1.34.1-1.1
apt-mark hold kubeadm
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 5.15.0-1083-gcp x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Sat Dec  6 19:42:56 2025 from 192.168.114.210

node01 ~ ➜ apt-mark unhold kubeadm
Canceled hold on kubeadm.
```

Step 8: Apply Worker Node Upgrade

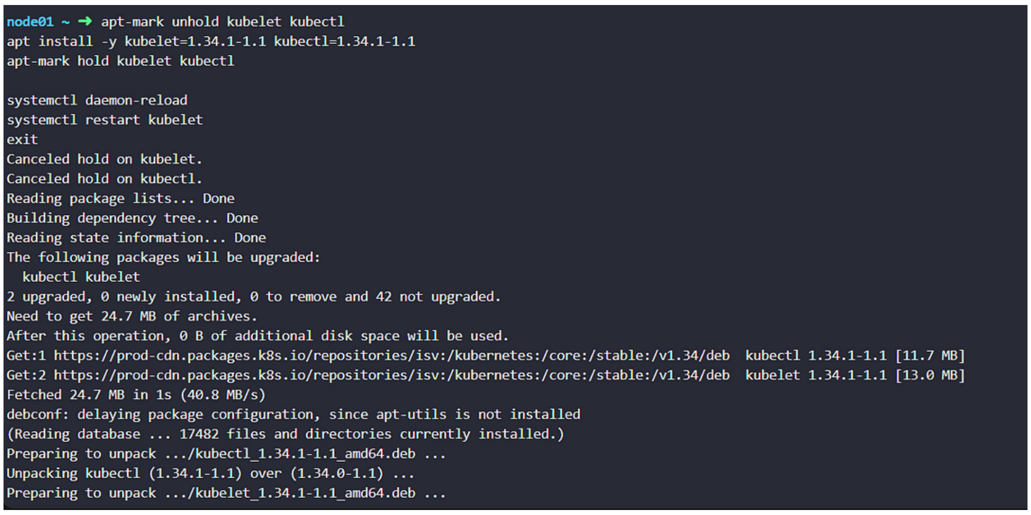
```
kubeadm upgrade node
```



```
node01 ~ ➜ kubeadm upgrade node
[upgrade] Reading configuration from the "kubeadm-config" ConfigMap in namespace "kube-system"...
[upgrade] Use 'kubeadm init phase upload-config kubeadm --config your-config-file' to re-upload it.
[upgrade/preflight] Running pre-flight checks
        [WARNING SystemVerification]: cgroups v1 support is in maintenance mode, please migrate to cgroups v2
[upgrade/preflight] Skipping prepull. Not a control plane node.
[upgrade/control-plane] Skipping phase. Not a control plane node.
[upgrade/kubeconfig] Skipping phase. Not a control plane node.
W1206 19:57:13.696008   24991 postupgrade.go:116] Using temporary directory /etc/kubernetes/tmp/kubeadm-kubelet-config3462200155 for kubelet config. To override it set the environ
ment variable KUBEADM_UPGRADE_DRYRUN_DIR
[upgrade] Backing up kubelet config file to /etc/kubernetes/tmp/kubeadm-kubelet-config3462200155/config.yaml
[patches] Applied patch of type "application/strategic-merge-patch+json" to target "kubeletconfiguration"
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[upgrade/kubelet-config] The kubelet configuration for this node was successfully upgraded!
[upgrade/addon] Skipping the addon/coredns phase. Not a control plane node.
[upgrade/addon] Skipping the addon/kube-proxy phase. Not a control plane node.
```

Step 9: Upgrade kubelet and kubectl

```
apt-mark unhold kubelet kubectl
apt install -y kubelet=1.34.1-1.1 kubectl=1.34.1-1.1
apt-mark hold kubelet kubectl

systemctl daemon-reload
systemctl restart kubelet
exit
```



```
node01 ~ ➜ apt-mark unhold kubelet kubectl
apt install -y kubelet=1.34.1-1.1 kubectl=1.34.1-1.1
apt-mark hold kubelet kubectl

systemctl daemon-reload
systemctl restart kubelet
exit
Canceled hold on kubelet.
Canceled hold on kubectl.
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following packages will be upgraded:
  kubectl kubelet
2 upgraded, 0 newly installed, 0 to remove and 42 not upgraded.
Need to get 24.7 MB of archives.
After this operation, 0 B of additional disk space will be used.
Get:1 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.34/deb  kubectl 1.34.1-1.1 [11.7 MB]
Get:2 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.34/deb  kubelet 1.34.1-1.1 [13.0 MB]
Fetched 24.7 MB in 1s (40.8 MB/s)
debconf: delaying package configuration, since apt-utils is not installed
(Reading database ... 17482 files and directories currently installed.)
Preparing to unpack .../kubectl_1.34.1-1.1_amd64.deb ...
Unpacking kubectl (1.34.1-1.1) over (1.34.0-1.1) ...
Preparing to unpack .../kubelet_1.34.1-1.1_amd64.deb ...
```

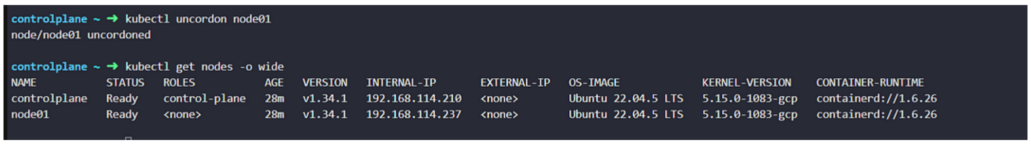
Step 10: Uncordon Worker

```
kubectl uncordon node01
```

4.1.5 Verification

Check Node Versions

```
kubectl get nodes -o wide
```



```
controlplane ~ ➜ kubectl uncordon node01
node/node01 uncordoned

controlplane ~ ➜ kubectl get nodes -o wide
NAME           STATUS   ROLES           AGE   VERSION   INTERNAL-IP       EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION     CONTAINER-RUNTIME
controlplane   Ready    control-plane   28m   v1.34.1   192.168.114.210   <none>        Ubuntu 22.04.5 LTS   5.15.0-1083-gcp    containerd://1.6.26
node01         Ready    <none>          28m   v1.34.1   192.168.114.237   <none>        Ubuntu 22.04.5 LTS   5.15.0-1083-gcp    containerd://1.6.26
```

Expected:

```
NAME           STATUS   ROLES           VERSION
controlplane   Ready    control-plane   v1.34.1
node01         Ready    <none>          v1.34.1
```

Check kubelet Versions

```
kubectl describe node controlplane | grep -i kubelet
kubectl describe node node01       | grep -i kubelet
```

Both should show `v1.34.1`.

Check Cluster Health

```
kubectl get pods -A
```

All pods should be `Running` or `Completed`.

4.1.6 Success Criteria

- Both nodes (`controlplane` and `node01`) report version `v1.34.1`.
- `kubeadm`, control plane components, kubelet, and `kubectl` were upgraded in the correct order.
- Control plane upgrade occurred **without draining first**.
- Control plane was drained **only** for the kubelet upgrade.
- Worker node was upgraded using the `drain` → `upgrade` → `uncordon` sequence.
- Cluster remained healthy and schedulable throughout.

Exam Tip

- The only time you drain the control plane is when upgrading `kubelet`.
- **Never drain before `kubeadm upgrade apply`.**
- Workers must be upgraded sequentially with `drain` → `upgrade` → `uncordon`.
- Use `apt-mark` to control versions and avoid accidental upgrades.

4.2 Scenario 2: Creating and Using a Persistent Volume (PV) with a Persistent Volume Claim (PVC)

Domain: Storage (10%)

Objective:
Create a **10Gi PersistentVolume (PV)** using a **hostPath** at `/mnt/data`, with a **Retain** reclaim policy and a **storageClassName** of `manual`. Then create a matching **PersistentVolumeClaim (PVC)** requesting exactly **10Gi**, and mount it inside an `nginx` pod at `/usr/share/nginx/html`.

Context:
PV/PVC tasks in the CKA check your ability to:

- Provide matching storage size between PV and PVC.
- Use correct **storageClassName**.
- Confirm binding before scheduling the pod.

Always Remember

- Capacity must match: **PV = 10Gi**, **PVC = 10Gi**.
- **storageClassName** must match: `manual`.
- hostPath is acceptable for single-node or exam scenarios.

4.2.1 Lab vs. Exam Requirements

In a lab, you may need to verify the node path `/mnt/data` exists. In the CKA exam, resources are pre-configured and focus is on YAML correctness.

Task	**Lab**	**Exam**
Create /mnt/data on node	✓	×
Write full manifest file	✓	✓
Apply YAML	✓	✓
Check binding with `kubectl get pvc`	✓	✓
Troubleshoot missing storage path	✓	×

Note: In the exam environment, hostPath directories already exist; no path creation is required.

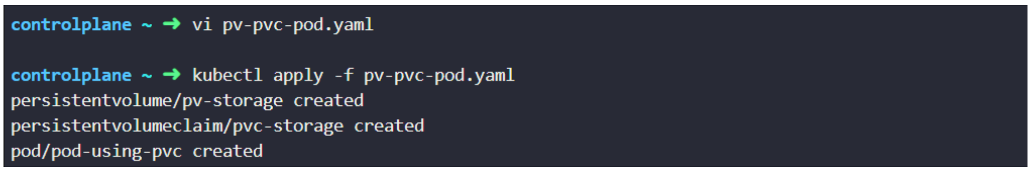
4.2.2 Create PV, PVC, and Pod

Step 1: Create the manifest

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-storage
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: manual
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-storage
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: manual
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-pvc
spec:
  containers:
  - name: app-container
    image: nginx
    volumeMounts:
    - mountPath: "/usr/share/nginx/html"
      name: storage-volume
  volumes:
  - name: storage-volume
    persistentVolumeClaim:
      claimName: pvc-storage
```

Step 2: Apply the manifest

```
kubectl apply -f pv-pvc-pod.yaml
```

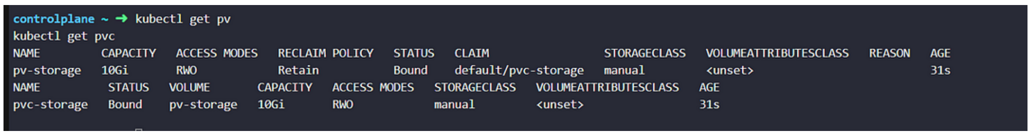


```
controlplane ~ ➜ vi pv-pvc-pod.yaml

controlplane ~ ➜ kubectl apply -f pv-pvc-pod.yaml
persistentvolume/pv-storage created
persistentvolumeclaim/pvc-storage created
pod/pod-using-pvc created
```

4.2.3 Verification

Step 3: Confirm PV and PVC state

```
kubectl get pv
kubectl get pvc
```



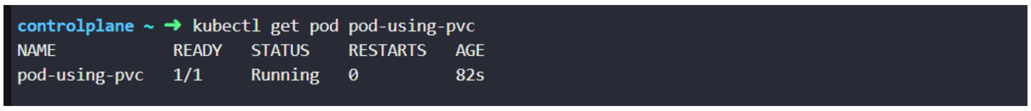
```
controlplane ~ ➜ kubectl get pv
kubectl get pvc
NAME         CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                 STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
pv-storage   10Gi       RWO            Retain           Bound    default/pvc-storage   manual         <unset>                          31s
NAME          STATUS   VOLUME       CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
pvc-storage   Bound    pv-storage   10Gi       RWO            manual         <unset>                 31s
```

Expected:

```
NAME         CAPACITY   STATUS   STORAGECLASS
pv-storage   10Gi       Bound    manual

NAME          STATUS   VOLUME       CAPACITY
pvc-storage   Bound    pv-storage   10Gi
```

Step 4: Confirm pod is running

```
kubectl get pod pod-using-pvc
```



```
controlplane ~ ➜ kubectl get pod pod-using-pvc
NAME            READY   STATUS    RESTARTS   AGE
pod-using-pvc   1/1     Running   0          82s
```

4.2.4 Explanation of Key Concepts

- **PV** defines storage capacity, path, and reclaim policy.
- **PVC** requests storage; binding occurs when:
  - capacity matches, and
  - storageClassName matches.
- **volumeMounts** attach the claim inside the container.

4.2.5 Success Criteria

- PV created with:
  - capacity: **10Gi**
  - reclaim policy: **Retain**
  - storageClassName: `manual`
- PVC requests **10Gi** with same storage class.
- PV and PVC reach **Bound** state.
- Pod runs and mounts volume at: `/usr/share/nginx/html`

Exam Tip

- Most binding errors are caused by **storageClassName mismatch**.
- If PVC is `Pending`, check:
  - `kubectl get pv`
  - `kubectl describe pvc`
- Do not debug the pod before PVC binding is correct.

4.3 Scenario 3: Draining a Node for Maintenance and Rescheduling Pods

Domain: Workloads & Scheduling (15%)

Objective:
Simulate node maintenance by draining the worker node `node01`, evicting all non-DaemonSet pods, and restoring the node back to service once maintenance is completed.

Context:
In the CKA exam, node maintenance tasks verify:

- Selecting and draining the correct node
- Evicting pods safely using `-ignore-daemonsets`
- Returning the node to scheduling with `uncordon`

Always Remember

- **Use `-ignore-daemonsets`** when draining a node.
- Draining changes the node state to `SchedulingDisabled`.
- **Do not use destructive flags** unless required.
- Verify that workloads are rescheduled before uncordoning.

4.3.1 Lab vs. Exam Requirements

A drain operation behaves the same in lab and exam, except lab requires no real maintenance.

Task	Lab	Exam
Identify node	✓	✓
Drain node with `-ignore-daemonsets`	✓	✓
Use `-delete-emptydir-data` if requested	✓	✓
Simulate maintenance	simulated	×
Uncordon after maintenance	✓	✓
Verify pods moved	✓	✓

4.3.2 Step-by-step Instructions

Step 1: Verify Node State

```
kubectl get nodes
```

Expected:

```
NAME           STATUS   ROLES           VERSION
controlplane   Ready    control-plane   v1.34.0
node01         Ready    <none>          v1.34.0
```

Step 2: Drain the Worker Node

Evict non-DaemonSet pods:

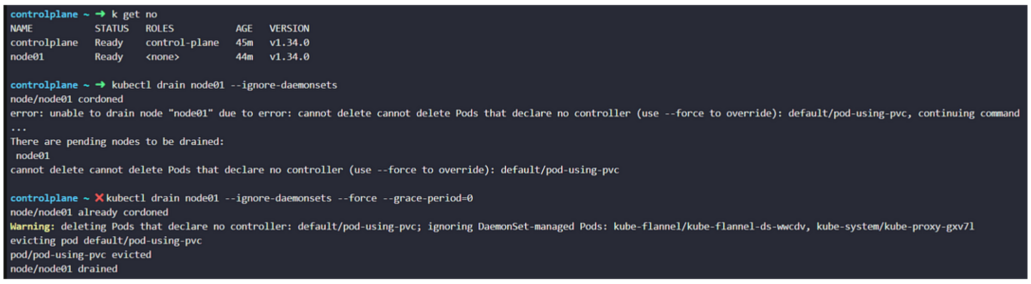
```
kubectl drain node01 --ignore-daemonsets
```

If a task allows removal of local storage:

```
kubectl drain node01 --ignore-daemonsets --delete-emptydir-data
```

If unmanaged pods block drain:

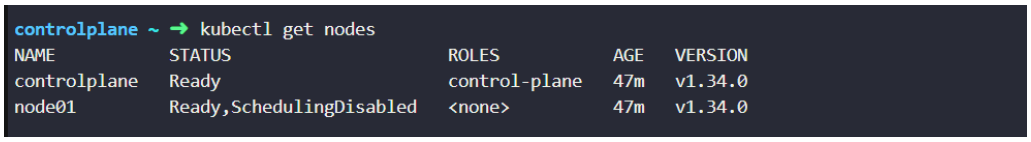
```
kubectl drain node01 --ignore-daemonsets --force --grace-period=0
```



```
controlplane ~ ➜ k get no
NAME           STATUS   ROLES           AGE   VERSION
controlplane   Ready    control-plane   45m   v1.34.0
node01         Ready    <none>          44m   v1.34.0

controlplane ~ ➜ kubectl drain node01 --ignore-daemonsets
node/node01 cordoned
error: unable to drain node "node01" due to error: cannot delete cannot delete Pods that declare no controller (use --force to override): default/pod-using-pvc, continuing command
...
There are pending nodes to be drained:
 node01
cannot delete cannot delete Pods that declare no controller (use --force to override): default/pod-using-pvc

controlplane ~ ✗ kubectl drain node01 --ignore-daemonsets --force --grace-period=0
node/node01 already cordoned
Warning: deleting Pods that declare no controller: default/pod-using-pvc; ignoring DaemonSet-managed Pods: kube-flannel/kube-flannel-ds-wwcdv, kube-system/kube-proxy-gxv7l
evicting pod default/pod-using-pvc
pod/pod-using-pvc evicted
node/node01 drained
```

Node state becomes:

```
kubectl get nodes
```



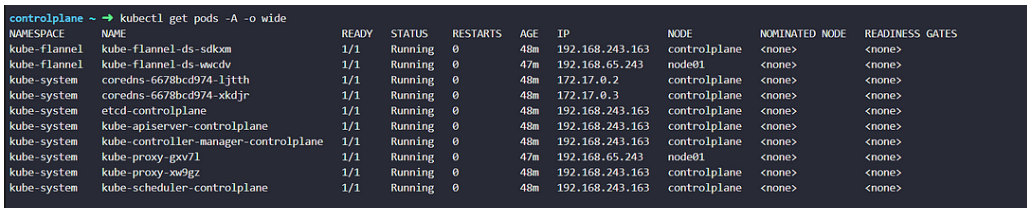
```
controlplane ~ ➜ kubectl get nodes
NAME           STATUS                     ROLES           AGE   VERSION
controlplane   Ready                      control-plane   47m   v1.34.0
node01         Ready,SchedulingDisabled   <none>          47m   v1.34.0
```

Expected:

```
node01   Ready,SchedulingDisabled
```

Step 3: Verify Pod Rescheduling

```
kubectl get pods -A -o wide
```



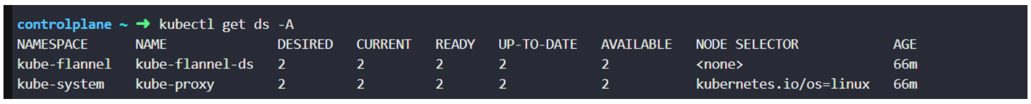
```
controlplane ~ ➜ kubectl get pods -A -o wide
NAMESPACE      NAME                                   READY   STATUS    RESTARTS   AGE   IP                NODE           NOMINATED NODE   READINESS GATES
kube-flannel   kube-flannel-ds-sdkxm                  1/1     Running   0          48m   192.168.243.163   controlplane   <none>           <none>
kube-flannel   kube-flannel-ds-wwcdv                  1/1     Running   0          47m   192.168.65.243    node01         <none>           <none>
kube-system    coredns-6678bcd974-ljtth               1/1     Running   0          48m   172.17.0.2        controlplane   <none>           <none>
kube-system    coredns-6678bcd974-xkdjr               1/1     Running   0          48m   172.17.0.3        controlplane   <none>           <none>
kube-system    etcd-controlplane                      1/1     Running   0          48m   192.168.243.163   controlplane   <none>           <none>
kube-system    kube-apiserver-controlplane            1/1     Running   0          48m   192.168.243.163   controlplane   <none>           <none>
kube-system    kube-controller-manager-controlplane   1/1     Running   0          48m   192.168.243.163   controlplane   <none>           <none>
kube-system    kube-proxy-gxv7l                       1/1     Running   0          47m   192.168.65.243    node01         <none>           <none>
kube-system    kube-proxy-xw9gz                       1/1     Running   0          48m   192.168.243.163   controlplane   <none>           <none>
kube-system    kube-scheduler-controlplane            1/1     Running   0          48m   192.168.243.163   controlplane   <none>           <none>
```

Non-DaemonSet workloads should move to `controlplane`.

Why Pods May Stay Visible After Drain

In many lab clusters, including the one used for this scenario, the only pods running on the worker node are:

- `kube-flannel-ds-`
- `kube-proxy-`



```
controlplane ~ ➜ kubectl get ds -A
NAMESPACE     NAME              DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
kube-flannel  kube-flannel-ds   2         2         2       2            2           <none>                   66m
kube-system   kube-proxy        2         2         2       2            2           kubernetes.io/os=linux   66m
```

These are **DaemonSets**. DaemonSet pods are designed to run on every node and are **never evicted during a drain** when using `-ignore-daemonsets`.

This means:

- These pods will remain on `node01` during the drain.
- This **does not indicate a problem**.
- New workloads will still avoid the node since it is `SchedulingDisabled`.

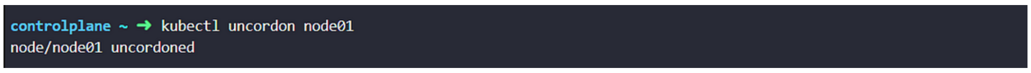
This is the correct and expected behavior.

Step 4: Perform Maintenance (Simulated)

No real commands are required.

Step 5: Restore Scheduling

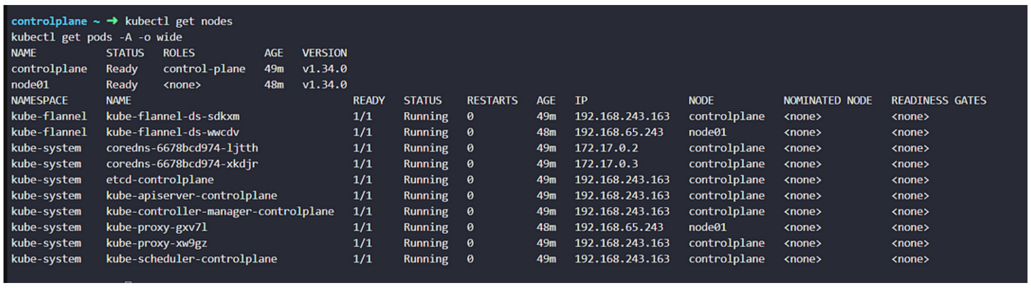
```
kubectl uncordon node01
```



```
controlplane ~ ➜ kubectl uncordon node01
node/node01 uncordoned
```

Step 6: Verify

```
kubectl get nodes
kubectl get pods -A -o wide
```



```
controlplane ~ ➜ kubectl get nodes
kubectl get pods -A -o wide
NAME           STATUS   ROLES           AGE   VERSION
controlplane   Ready    control-plane   49m   v1.34.0
node01         Ready    <none>          48m   v1.34.0
NAMESPACE      NAME                                   READY   STATUS    RESTARTS   AGE   IP                NODE           NOMINATED NODE   READINESS GATES
kube-flannel   kube-flannel-ds-sdkxm                  1/1     Running   0          49m   192.168.243.163   controlplane   <none>           <none>
kube-flannel   kube-flannel-ds-wwcdv                  1/1     Running   0          48m   192.168.65.243    node01         <none>           <none>
kube-system    coredns-6678bcd974-ljtth               1/1     Running   0          49m   172.17.0.2        controlplane   <none>           <none>
kube-system    coredns-6678bcd974-xkdjr               1/1     Running   0          49m   172.17.0.3        controlplane   <none>           <none>
kube-system    etcd-controlplane                      1/1     Running   0          49m   192.168.243.163   controlplane   <none>           <none>
kube-system    kube-apiserver-controlplane            1/1     Running   0          49m   192.168.243.163   controlplane   <none>           <none>
kube-system    kube-controller-manager-controlplane   1/1     Running   0          49m   192.168.243.163   controlplane   <none>           <none>
kube-system    kube-proxy-gxv7l                       1/1     Running   0          48m   192.168.65.243    node01         <none>           <none>
kube-system    kube-proxy-xw9gz                       1/1     Running   0          49m   192.168.243.163   controlplane   <none>           <none>
kube-system    kube-scheduler-controlplane            1/1     Running   0          49m   192.168.243.163   controlplane   <none>           <none>
```

Expected:

```
NAME      STATUS   ROLES    VERSION
node01    Ready    <none>   v1.34.0
```

4.3.3 Explanation of Key Commands

- `kubectl drain`: marks node unschedulable and evicts pods
- `-ignore-daemonsets`: prevents drain from hanging
- `kubectl uncordon`: re-enables scheduling

4.3.4 Success Criteria

- Node `node01` transitions to `Ready,SchedulingDisabled` after drain
- Non-DaemonSet workloads are rescheduled
- Node returns to `Ready` after `uncordon`

Exam Tip

- Always include `-ignore-daemonsets` when draining a node.
- Use destructive flags only when explicitly instructed.
- Confirm you are operating on the correct node before draining.

4.4 Scenario 4: Configuring an Ingress with TLS for a Production Service

Domain: Services & Networking (20%)

Objective:
Expose the `web-service` in the `production` namespace over HTTPS using an **Ingress** resource secured with a TLS certificate stored in the secret `tls-secret`. Requests to `https://prod.example.com` should be routed to the service on port 80.

Context:
An Ingress can terminate HTTPS traffic using a TLS certificate stored in a Kubernetes Secret. In the CKA exam, you may be required to configure Ingress TLS using a pre-provided certificate and key. The exam environment provides an Ingress controller when Ingress-related tasks are required; you do *not* need to install one.

References for further reading:

- Ingress Overview
- TLS in Ingress
- Secrets

Always Remember

- The TLS secret must be in the **same namespace** as the Ingress.
- The secret type must be `kubernetes.io/tls` with `tls.crt` and `tls.key`.
- An Ingress controller must already exist in the cluster; the exam provides one when needed.
- If the environment requires it, set `spec.ingressClassName` appropriately.

Step-by-step instructions:

Step 1: Verify Context and Namespace

```
kubectl config use-context cluster-prod
kubectl get ns
```

Step 2: Confirm the Service Exists

```
kubectl get svc web-service -n production
```

Step 3: Create TLS Secret from Provided Certificate and Key

```
kubectl create secret tls tls-secret \
  --cert=/opt/certs/prod.crt \
  --key=/opt/certs/prod.key \
  -n production
```

Step 4: Create the Ingress Manifest with TLS Enabled

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: prod-ingress
  namespace: production
spec:
  # Uncomment if the cluster requires specifying the ingress class:
  # ingressClassName: nginx
  tls:
  - hosts:
      - prod.example.com
    secretName: tls-secret
  rules:
  - host: prod.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: web-service
            port:
              number: 80
```

Step 5: Apply the Ingress and Verify

```
kubectl apply -f prod-ingress.yaml
kubectl describe ingress prod-ingress -n production
```

Step 6: Test HTTPS Access Ensure `prod.example.com` resolves to the Ingress controller IP, then test:

```
curl -k https://prod.example.com
```

Explanation for critical commands:

- `kubectl create secret tls` : Generates a TLS secret from certificate and key files.
- `tls:` block in Ingress : Enables HTTPS termination using the specified secret.
- `curl -k` : Tests HTTPS without verifying the certificate chain.

Expected Outcome:

- The secret `tls-secret` exists in the `production` namespace.
- The `prod-ingress` resource terminates TLS and routes traffic to `web-service`.
- Accessing `https://prod.example.com` returns the application successfully over HTTPS.

Exam Tip

- TLS secrets may already exist in exam tasks, always check before creating new ones.
- If Ingress routing fails, verify the `host`, `secretName`, and `service port` for typos.

4.5 Scenario 5: Deploying a StatefulSet with Persistent Storage for a Database

Domain: Workloads & Scheduling (15%)

Objective:
Deploy a StatefulSet running `mysql:8.0` in the `database` namespace with:

- 3 replicas,
- A headless service for stable DNS names,
- A PersistentVolumeClaim template (size: 5Gi) for each replica.

Context:
StatefulSets manage stateful applications that require stable network identities and persistent storage across pod restarts. In the CKA exam, you may be asked to create both the `Service` and the `StatefulSet`, ensuring that storage is automatically provisioned using a volume claim template.

References for further reading:

- StatefulSet Overview
- Persistent Volume Claims
- Running Stateful Applications

Always Remember

- A headless service (`clusterIP: None`) is required for stable DNS in StatefulSets.
- Each replica receives its own PVC created from the volume claim template.
- PVCs created by StatefulSets are **not deleted automatically** when the StatefulSet is removed.

Step-by-step instructions:

Step 1: Verify Context and Namespace

```
kubectl config use-context cluster-prod
kubectl create namespace database
```

Step 2: Create the Headless Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql
  namespace: database
spec:
  clusterIP: None
  ports:
    - port: 3306
      name: mysql
  selector:
    app: mysql
```

Step 3: Create the StatefulSet Manifest

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
  namespace: database
spec:
  serviceName: mysql
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:8.0
        ports:
        - containerPort: 3306
          name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: rootpassword
        volumeMounts:
        - name: mysql-data
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 5Gi
      # Uncomment if the cluster requires specifying a StorageClass:
      # storageClassName: standard
```

Step 4: Apply the Manifests

```
kubectl apply -f mysql-service.yaml
kubectl apply -f mysql-statefulset.yaml
```

Step 5: Verify Deployment

```
kubectl get pods -n database -o wide
kubectl get pvc -n database
```

Explanation for critical commands:

- `clusterIP: None` : Creates a headless service for stable DNS names.
- `volumeClaimTemplates` : Automatically provisions a unique PVC per replica.
- `kubectl get pvc` : Confirms PVCs were created for each StatefulSet pod.

Expected Outcome:

- Three MySQL pods (`mysql-0`, `mysql-1`, `mysql-2`) running in the `database` namespace.
- Three PVCs bound to persistent volumes (`mysql-data-mysql-0`, `mysql-data-mysql-1`, etc.).
- Each pod reachable via stable DNS:
  - `mysql-0.mysql.database.svc.cluster.local`
  - `mysql-1.mysql.database.svc.cluster.local`
  - `mysql-2.mysql.database.svc.cluster.local`

Exam Tip

- StatefulSet pods start **sequentially**: `mysql-0` must become Ready before `mysql-1` starts.
- Match the `storageClassName` to the exam environment if one is required.

4.6 Scenario 6: Backing Up and Restoring etcd with TLS Certificates

Domain: Cluster Maintenance (11%)

Objective:
Perform a complete etcd snapshot backup and restore in a kubeadm-managed cluster using the correct TLS certificates, ensuring the control plane returns to a healthy state.

Context:
This scenario mirrors a frequent exam pattern:

- etcd runs as a **static pod** under `/etc/kubernetes/manifests/`
- Backup and restore must be executed using **etcdctl v3**
- Certificate paths must be taken from `etcd.yaml`
- Restore must target a **clean data directory**

Always Remember

Stopping **etcd** directly does nothing. Static pods are controlled by the **kubelet**. To stop etcd: **stop the kubelet**.

4.6.1 Exam Strategy Overview

When this appears in the exam, the fastest approach is:

1. Export API version
2. Take snapshot
3. Stop kubelet
4. Move old data
5. Restore into clean directory
6. Update manifest
7. Start kubelet
8. Validate health

This sequence is linear and score-efficient.

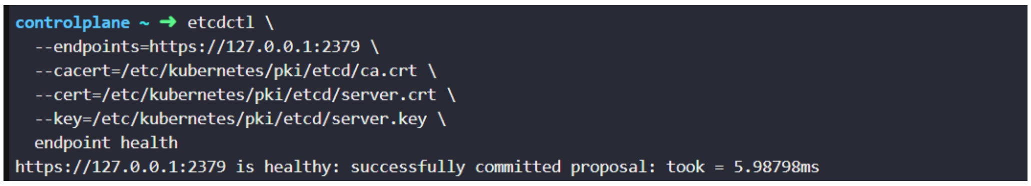
4.6.2 Backup Process

Step 1: Export API Version

```
export ETCDCTL_API=3
```

Step 2: Verify Health (Optional)

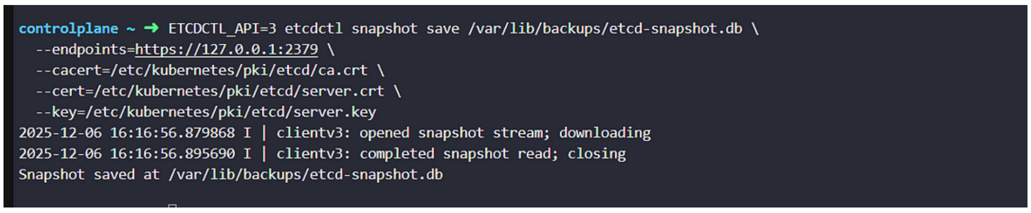
```
etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  endpoint health
```



```
controlplane ~ ➜ 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 \
  endpoint health
https://127.0.0.1:2379 is healthy: successfully committed proposal: took = 5.98798ms
```

Step 3: Create Snapshot

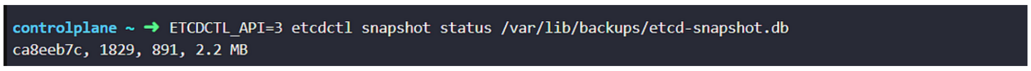
```
etcdctl snapshot save /var/lib/backups/etcd-snapshot.db \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key
```



```
controlplane ~ ➜ ETCDCTL_API=3 etcdctl snapshot save /var/lib/backups/etcd-snapshot.db \
  --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
2025-12-06 16:16:56.879868 I | clientv3: opened snapshot stream; downloading
2025-12-06 16:16:56.895690 I | clientv3: completed snapshot read; closing
Snapshot saved at /var/lib/backups/etcd-snapshot.db
```

Step 4: Verify

```
etcdctl snapshot status /var/lib/backups/etcd-snapshot.db
```



```
controlplane ~ ➜ ETCDCTL_API=3 etcdctl snapshot status /var/lib/backups/etcd-snapshot.db
ca8eeb7c, 1829, 891, 2.2 MB
```

4.6.3 Restore Process

Step 1: Stop kubelet

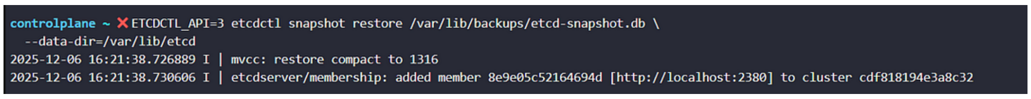
```
systemctl stop kubelet
```

Step 2: Archive Old Data

```
mv /var/lib/etcd /var/lib/etcd-old
```

Step 3: Restore Snapshot to a Clean Directory

```
etcdctl snapshot restore /var/lib/backups/etcd-snapshot.db \
  --data-dir=/var/lib/etcd
```



```
controlplane ~ ✗ ETCDCTL_API=3 etcdctl snapshot restore /var/lib/backups/etcd-snapshot.db \
  --data-dir=/var/lib/etcd
2025-12-06 16:21:38.726889 I | mvcc: restore compact to 1316
2025-12-06 16:21:38.730606 I | etcdserver/membership: added member 8e9e05c52164694d [http://localhost:2380] to cluster cdf818194e3a8c32
```

Step 4: Update Static Pod Manifest

Open:

```
vi /etc/kubernetes/manifests/etcd.yaml
```

Ensure:

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

Step 5: Start kubelet

```
systemctl start kubelet
```

Wait for the pod to reappear.

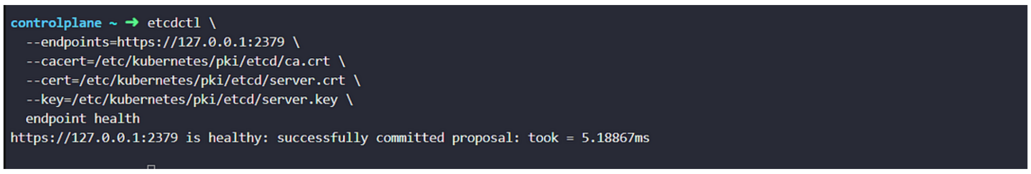
4.6.4 Validation

Step 1: Check Pod Status

```
crictl ps | grep etcd
# or:
docker ps | grep etcd
```

Step 2: Verify Health

```
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 \
  endpoint health
```



```
controlplane ~ ➜ 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 \
  endpoint health
https://127.0.0.1:2379 is healthy: successfully committed proposal: took = 5.18867ms
```

Expected output:

```
https://127.0.0.1:2379 is healthy
```

Success condition:

$$etcd endpoint = healthy$$

4.6.5 Simulation Notes (Real Lab Behavior)

During the lab:

- Installing `etcdctl` may be necessary:

  ```
  apt-get update && apt-get install -y etcd-client
  ```

- Stopping kubelet does not instantly kill the etcd container, this is expected.
- The snapshot restore prints messages like:

  ```
  added member 8e9e05c52164694d [http://localhost:2380]
  ```

Important

This means the restore succeeded. After updating the manifest and starting kubelet, the cluster returns to a healthy state.

4.6.6 Exam Day Checklist

- Use **ETCDCTL_API=3**
- Backup path and restore path must be exact
- Stop kubelet before restoring
- Use a **new data directory**
- Update **only** the `-data-dir` flag
- Validate with `endpoint health`

Exam Tip

This task is fast to complete and highly reliable, **always do it when it appears.**

4.7 Scenario 7: Creating a Custom ClusterRole and Binding It to a Namespace-Scoped ServiceAccount

Domain: Cluster Architecture, Installation & Configuration (25%)

Objective:
Create a ClusterRole named `deployment-clusterrole` that grants permission to create:

- Deployments
- StatefulSets
- DaemonSets

Then create a ServiceAccount named `cicd-token` in the namespace `app-team1` and bind the ClusterRole so that its permissions are valid only in that namespace.

Context:
RBAC tasks in the CKA often test:

- Knowing the difference between ClusterRole and Role
- How to restrict ClusterRole permissions to a single namespace
- Writing RBAC YAML or generating resources quickly with kubectl

Always Remember

A ClusterRole is cluster-wide, but you can restrict it to a namespace by using a RoleBinding. Never use ClusterRoleBinding unless the question asks for cluster-wide access.

4.7.1 Exam Strategy Overview

Typical exam pattern:

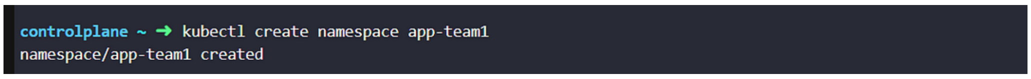
1. Create the namespace
2. Create the ClusterRole
3. Create the ServiceAccount in the correct namespace
4. Bind the role using RoleBinding
5. Test that permissions work only where they should

This scenario is fast and very score efficient.

4.7.2 ClusterRole Creation

Step 1: Verify Namespace

```
kubectl create namespace app-team1
```

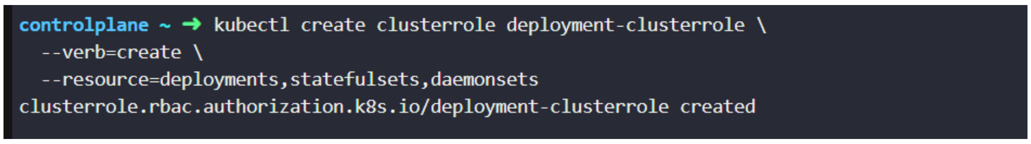


```
controlplane ~ ➜ kubectl create namespace app-team1
namespace/app-team1 created
```

Step 2: Create the ClusterRole

Fast creation using CLI:

```
kubectl create clusterrole deployment-clusterrole \
  --verb=create \
  --resource=deployments,statefulsets,daemonsets
```



```
controlplane ~ ➜ kubectl create clusterrole deployment-clusterrole \
  --verb=create \
  --resource=deployments,statefulsets,daemonsets
clusterrole.rbac.authorization.k8s.io/deployment-clusterrole created
```

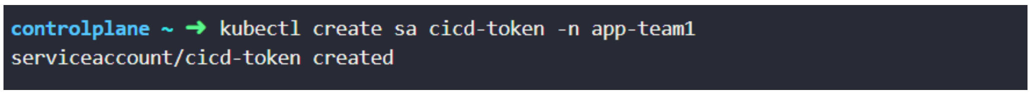
YAML version if needed:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: deployment-clusterrole
rules:
- apiGroups: ["apps"]
  resources:
    - deployments
    - statefulsets
    - daemonsets
  verbs:
    - create
```

4.7.3 ServiceAccount Creation

Step 3: Create ServiceAccount in Namespace

```
kubectl create sa cicd-token -n app-team1
```

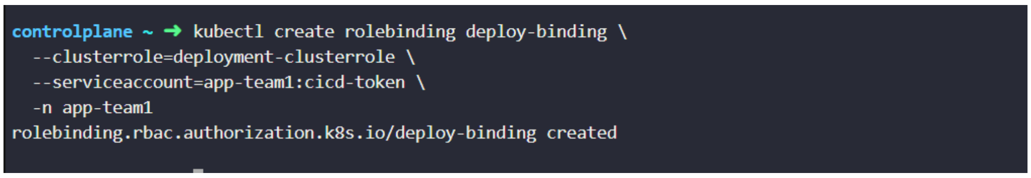


```
controlplane ~ ➜ kubectl create sa cicd-token -n app-team1
serviceaccount/cicd-token created
```

4.7.4 Binding the Role to the ServiceAccount

Step 4: Apply RoleBinding

```
kubectl create rolebinding deploy-binding \
  --clusterrole=deployment-clusterrole \
  --serviceaccount=app-team1:cicd-token \
  -n app-team1
```



```
controlplane ~ ➜ kubectl create rolebinding deploy-binding \
  --clusterrole=deployment-clusterrole \
  --serviceaccount=app-team1:cicd-token \
  -n app-team1
rolebinding.rbac.authorization.k8s.io/deploy-binding created
```

This restricts permissions to the namespace `app-team1`. Using a ClusterRoleBinding here would apply access cluster-wide, which is incorrect.

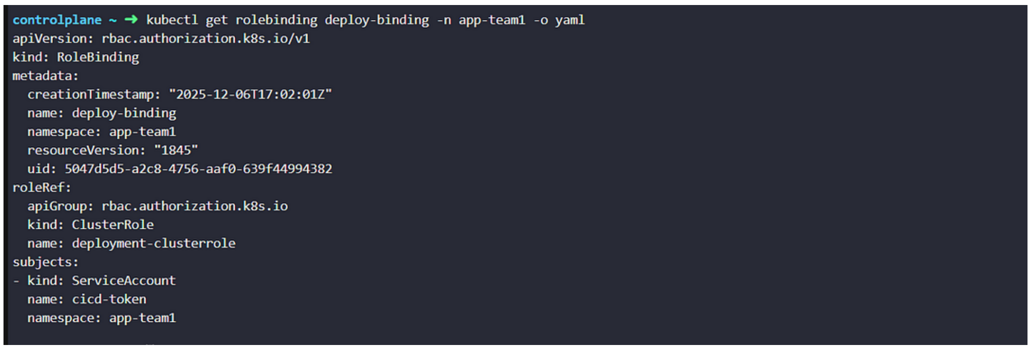
4.7.5 Verification of Objects

Step 5: Verify RoleBinding Configuration

```
kubectl get rolebinding deploy-binding -n app-team1 -o yaml
```

Check:

- `roleRef.kind` is `ClusterRole`
- `roleRef.name` is `deployment-clusterrole`
- `subjects[0].kind` is `ServiceAccount`
- `subjects[0].name` is `cicd-token`
- `subjects[0].namespace` is `app-team1`



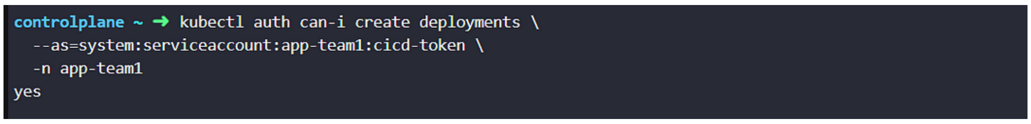
```
controlplane ~ ➜ kubectl get rolebinding deploy-binding -n app-team1 -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  creationTimestamp: "2025-12-06T17:02:01Z"
  name: deploy-binding
  namespace: app-team1
  resourceVersion: "1845"
  uid: 5047d5d5-a2c8-4756-aaf0-639f44994382
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: deployment-clusterrole
subjects:
- kind: ServiceAccount
  name: cicd-token
  namespace: app-team1
```

4.7.6 Testing Permissions (Recommended in Lab and Exam)

To be sure you created exactly what the task asked for, test it with `kubectl auth can-i` using impersonation.

Test 1: Allowed action in allowed namespace

```
kubectl auth can-i create deployments \
  --as=system:serviceaccount:app-team1:cicd-token \
  -n app-team1
```



```
controlplane ~ ➜ kubectl auth can-i create deployments \
  --as=system:serviceaccount:app-team1:cicd-token \
  -n app-team1
yes
```

Expected:

```
yes
```

Test 2: Other resources should be denied

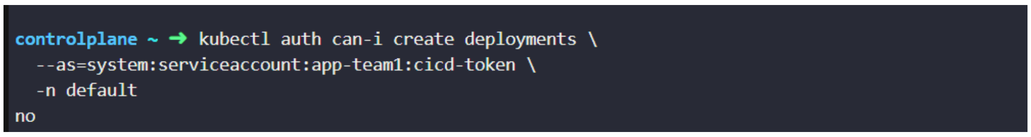
```
kubectl auth can-i create services \
  --as=system:serviceaccount:app-team1:cicd-token \
  -n app-team1
```

Expected:

```
no
```

Test 3: Same resources in another namespace should be denied

```
kubectl auth can-i create deployments \
  --as=system:serviceaccount:app-team1:cicd-token \
  -n default
```



```
controlplane ~ ➜ kubectl auth can-i create deployments \
  --as=system:serviceaccount:app-team1:cicd-token \
  -n default
no
```

Expected:

```
no
```

Why This Is Important

These three checks confirm:

- The ClusterRole grants the right verbs on the right resources
- The binding is limited to the `app-team1` namespace
- No extra permissions leaked to other namespaces

This is exactly the kind of validation the exam expects you to perform quietly before moving on.

4.7.7 Success Criteria

- ClusterRole `deployment-clusterrole` exists and targets deployments, statefulsets, and daemonsets
- ServiceAccount `cicd-token` exists in namespace `app-team1`
- RoleBinding `deploy-binding` ties them together in `app-team1`
- `kubectl auth can-i` reports:
  - `yes` for creating workloads in `app-team1`
  - `no` for other resources or other namespaces

Exam Tip

Use `kubectl auth can-i` with `-as` to test RBAC quickly without switching contexts or messing with kubeconfigs. It is one of the fastest ways to confirm that your Role, ClusterRole, and RoleBinding are wired correctly.

4.8 Scenario 8: Enforcing Pod Placement with Node Affinity and Taints

Domain: Workloads & Scheduling (15%)

Objective:
Ensure that a workload is scheduled only on nodes labeled `env=prod` and tolerates the taint `dedicated=frontend:NoSchedule` applied to those nodes.

Context:
Scheduling tasks in the CKA often combine:

- Node labels used by nodeAffinity
- Taints that repel pods from a node
- Tolerations that allow placement on a tainted node

Premium Scenarios Unlocked

You've completed 7 scenarios. There are 33 more exam-style scenarios waiting.

What's inside the premium version:

- **Total scenarios: 40**
- **Fully simulated on real Kubernetes clusters**
- **Every scenario includes:**
  - Detailed step-by-step commands
  - Verification outputs (`kubectl get`, `describe`, events)
  - Common failure cases and fixes
  - Time-saving shortcuts used during the real exam
- **Covers 100% of the CKA domains:**
  - Cluster Setup & Maintenance
  - Workloads & Scheduling
  - Networking & Services
  - Storage
  - Troubleshooting

Why it matters:

- These scenarios are **not theory**. They are **real commands** executed on **real clusters**.
- Every scenario has been **validated in a lab**, so you can reproduce it instantly.
- You don't have to guess, you see **exact outputs** and know if you're right.

Unlock all 32 remaining scenarios + lab simulations:

- Gumroad: https://devopsdynamo.gumroad.com/l/Conquer-cka-exam
- Payhip: https://payhip.com/b/3iAsH

Guarantee: *If you can run the commands in these scenarios, you can pass the CKA.*

5 Reinforcement Exercises: Network Policies

While preparing for the CKA, I encountered certain Kubernetes concepts that were particularly challenging to grasp. NetworkPolicies, in particular, took extra time and practice to fully understand. Defining policies that control traffic between pods, especially across namespaces, or specifying fine-grained egress and ingress rules, required more than just theoretical knowledge, it needed hands-on scenarios to bring the concepts to life.

These exercises are designed to reinforce some of the trickiest Kubernetes concepts, helping you simulate the exam environment and apply the right thinking process.

5.1 Exercise 1: Namespace-Specific Policies

Domain: Services & Networking (20%)

Objective: Create a `NetworkPolicy` that allows traffic only from pods with the label `app: nginx` **within the same namespace**, blocking identical labels from other namespaces.

Always Remember

- A `NetworkPolicy` is namespace-scoped; without a `namespaceSelector`, it applies only inside the namespace where it is created.
- Labels must match exactly; any mismatch results in denial.
- NetworkPolicies only affect pods that match their `podSelector`.

Policy Manifest:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-nginx-pod-2
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: nginx
  policyTypes:
  - Ingress
```

Explanation: This NetworkPolicy allows ingress traffic **to** pods labeled `app: nginx` only if the **source** pod also has label `app: nginx` in the **same namespace**. Traffic from other namespaces is blocked, even if the source pods have the same label.

Expected Outcome:

- Pods with label `app: nginx` in the same namespace can communicate with each other.

- Pods with the same label in other namespaces are denied.

Exam Tip

- Use:

  ```
  kubectl get pods --show-labels -n <namespace>
  ```

  to verify label matching before applying the policy.
- If a question mentions "within the same namespace", you typically do *not* need a `namespaceSelector`.

5.2 Exercise 2: Cross-Namespace Traffic

Domain: Services & Networking (20%)

Objective: Allow ingress traffic to pods labeled `app: nginx` from pods in a **different** namespace that has a specific label, `ns-label: allowed-namespace`.

Always Remember

- Use `namespaceSelector` to match namespaces by label.
- Combining `namespaceSelector` and `podSelector` in the same `from` entry creates an **AND** condition.
- In multi-namespace questions, always check namespace labels first.

Policy Manifest:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-specific-namespace
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          ns-label: allowed-namespace
      podSelector:
        matchLabels:
          app: nginx
  policyTypes:
  - Ingress
```

Explanation: This policy allows traffic from pods labeled `app: nginx` that are located in namespaces labeled `ns-label: allowed-namespace`. Traffic from other namespaces or pods without the correct label combination is blocked.

Expected Outcome:

- Only pods in namespaces labeled `ns-label: allowed-namespace` and with label `app: nginx` can connect.
- All other namespaces, even with matching pod labels, are denied.

Exam Tip

- Quickly list namespaces and their labels with:

  ```
  kubectl get ns --show-labels
  ```

- Remember: both the namespace label and the pod label must match for the rule to apply.

6 StorageClass and PersistentVolumeClaim Exercises

In the CKA exam, you'll almost certainly be asked to work with Kubernetes storage. These exercises are designed to reinforce the key concepts of `StorageClass`, `PersistentVolume` (PV), and `PersistentVolumeClaim` (PVC), from defining custom storage to binding volumes for applications. The goal is to simulate both the creation and troubleshooting steps you may need in a real-world or exam scenario.

6.1 Exercise 1: Creating a Custom StorageClass with PersistentVolumeClaim

Domain: Storage (10%)

Objective: Create a custom `StorageClass` with static provisioning and a `hostPath`-backed `PersistentVolume`, then bind it via a PVC for application use.

Exam Thinking

When asked to use a custom `StorageClass` with a static `PersistentVolume`:

- `kubernetes.io/no-provisioner` disables dynamic provisioning; you must create the PV manually.
- `hostPath` stores data on a single node; pods must keep running on that node to see the same data.
- `volumeBindingMode: WaitForFirstConsumer` delays binding until a pod is scheduled, which helps avoid scheduling conflicts.

Step 1: Define the StorageClass, PV, and PVC

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: custom-storageclass
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: custom-pv
spec:
  capacity:
    storage: 1Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: custom-storageclass
  hostPath:
    path: /mnt/data
---
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: custom-pvc
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: custom-storageclass
  resources:
    requests:
      storage: 1Gi
```

Always Remember

For static provisioning:

- `storageClassName` must match between PV and PVC.
- The requested PVC size must be less than or equal to the PV capacity.

6.2 Exercise 2: Deploying an Application with PVC

Domain: Storage (10%)

Objective: Deploy an application (NGINX) that uses the PVC from Exercise 1, and ensure data persists across pod restarts.

Deployment Manifest

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
        volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: storage
      volumes:
      - name: storage
        persistentVolumeClaim:
          claimName: custom-pvc
```

Example commands to verify persistence:

```
kubectl apply -f nginx-deployment.yaml
kubectl exec -it <nginx-pod> -- sh -c \
  "echo 'Persistent Test' > /usr/share/nginx/html/index.html"
kubectl delete pod <nginx-pod>
kubectl exec -it <new-nginx-pod> -- cat /usr/share/nginx/html/index.html
```

Always Remember

PVCs are namespace-scoped. The Deployment and the PVC must be in the **same** namespace to bind correctly.

Exam Tip

If the pod is stuck in `ContainerCreating`:

- Verify that `/mnt/data` exists on the node and has correct permissions.
- Confirm that the PV's and PVC's `storageClassName` values match exactly.

6.3 Exercise 3: Expanding a PersistentVolumeClaim

Domain: Storage (10%)

Objective: Increase the storage capacity of an existing PVC and verify that Kubernetes reflects the new size.

Exam Thinking

When you see "increase PVC size":

- Ensure the `StorageClass` has `allowVolumeExpansion: true`.
- PVs and their underlying storage can grow but not shrink.
- Some backends support online expansion; others may require pod restart or filesystem resize inside the container.

Step 1: Edit the PVC to Request More Storage

```
kubectl edit pvc custom-pvc
```

Modify:

```
spec:
  resources:
    requests:
      storage: 2Gi
```

Step 2: Verify the PVC Has Been Resized

```
kubectl get pvc custom-pvc
```

Step 3: Check the PV Reflects the New Size

```
kubectl get pv custom-pv
```

Always Remember

- Expansion support depends on the storage backend; `hostPath` is often used only for conceptual or lab examples.
- If expansion appears stuck, check `kubectl describe pvc` for events and errors.

Exam Tip

If the PVC shows the new size but the application still sees the old filesystem size, restart the pod or consult backend-specific resize steps.

Note: This section includes **3 StorageClass & PVC exercises**. You can unlock **4 additional lab-tested exercises** (7 total) in the premium version:
Gumroad: Conquer CKA Exam (Premium)
Payhip: Conquer CKA Exam (Premium)

You've already completed 3. Only 4 more to master the Storage section.

7 Additional Scenarios for Certification and Troubleshooting

Working with Kubernetes often involves handling certificates to secure communications and troubleshooting complex issues in live clusters. The following exercises provide practical examples to help you master these skills, both are highly relevant for the **CKA® exam**.

7.1 Certificate Management Exercises

7.1.1 Scenario 1: Using Self-Signed Certificates in a Pod

Domain: Workloads & Scheduling (15%) / Services & Networking (20%)

Objective: Prepare a pod to use a self-signed TLS certificate by storing the certificate and key in a Kubernetes TLS `Secret` and mounting it into the container.

Exam Thinking

Maps to:

- **Workloads & Scheduling:** Use `Secrets` to configure applications.
- **Services & Networking:** Enable TLS for in-cluster HTTP services.

In the exam, when you see "store TLS certs in a Secret and mount them into a pod":

- Use `kubectl create secret tls` for speed.
- Make sure the Secret type is `kubernetes.io/tls`.
- Mount the Secret via a volume, then point the app to the mounted path.

Step 1: Generate TLS Certificate and Key (Self-Signed)

```
openssl req -x509 -newkey rsa:4096 \
  -keyout tls.key -out tls.crt \
  -days 365 -nodes \
  -subj "/CN=my-service"
```

Step 2: Create Kubernetes TLS Secret

Fast CLI method (recommended in exam):

```
kubectl create secret tls tls-secret \
  --cert=tls.crt \
  --key=tls.key
```

Equivalent YAML (if required):

```
apiVersion: v1
kind: Secret
metadata:
  name: tls-secret
type: kubernetes.io/tls
data:
  tls.crt: <base64_encoded_certificate>
  tls.key: <base64_encoded_key>
```

Step 3: Mount Secret in a Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - mountPath: "/etc/tls"
      name: tls-volume
      readOnly: true
  volumes:
  - name: tls-volume
    secret:
      secretName: tls-secret
```

Always Remember

- Secrets are **base64-encoded**, not encrypted by default.
- The application (e.g. nginx) must be configured to read `tls.crt` and `tls.key` from the mounted directory.
- `type: kubernetes.io/tls` expects keys named `tls.crt` and `tls.key`.

Exam Tip

If the task says "secure a service with TLS using a Secret":

1. Generate or use the given cert and key.
2. `kubectl create secret tls <name> -cert=<crt> -key=<key>`
3. Mount the Secret into the pod or reference it from an Ingress.

7.1.2 Scenario 2: Automating Certificate Issuance with cert-manager

Domain: Cluster Architecture, Installation & Configuration (25%) / Services & Networking (20%)

Objective: Automate TLS certificate management using `cert-manager` and an ACME issuer (e.g. Let's Encrypt), so that certificates are issued and renewed without manual OpenSSL commands.

Exam Thinking

Maps to:

- **Cluster Architecture, Installation & Configuration:**
  - Install cluster add-ons via Helm.
  - Work with CRDs and operators (cert-manager).
- **Services & Networking:**
  - Terminate HTTPS for Ingress or services.

When you see "automated TLS" / "Let's Encrypt" / "ACME":

- Think: `Issuer`/`ClusterIssuer` + `Certificate`.
- Certificates are written into a `Secret` named in `spec.secretName`.

Step 1: Install cert-manager (Helm Example)

```
helm repo add jetstack https://charts.jetstack.io
helm repo update

helm install cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --create-namespace \
  --version v1.7.1 \
  --set installCRDs=true
```

Step 2: Create a ClusterIssuer (ACME / Let's Encrypt)

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: your-email@example.com
    privateKeySecretRef:
      name: letsencrypt-prod
    solvers:
    - http01:
        ingress:
          class: nginx
```

```
kubectl apply -f clusterissuer.yaml
```

Step 3: Request a Certificate (Namespace-Scoped)

```yaml
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-service-cert
  namespace: default
spec:
  secretName: my-service-tls
  dnsNames:
  - my-service.example.com
  issuerRef:
    name: letsencrypt-prod
    kind: ClusterIssuer
```

```
kubectl apply -f certificate.yaml
```

Always Remember

- `Certificate` is namespace-scoped and writes the keypair into `spec.secretName` in the **same** namespace.
- ACME HTTP-01 challenges require:
  - Public DNS pointing to your Ingress,
  - A working Ingress controller for the specified `class`.
- `ClusterIssuer` is cluster-wide; `Issuer` is namespace-scoped.

Exam Tip

If something is wrong:

- Check `kubectl describe certificate my-service-cert` for events.
- Check the `cert-manager` pod logs in the `cert-manager` namespace.
- Confirm the `secretName` from `Certificate` is the same Secret used in your Ingress or application.

7.2 Troubleshooting Exercises

Objective: Practice diagnosing and resolving common Kubernetes cluster issues through hands-on troubleshooting, focusing on quick triage and targeted fixes.

7.2.1 Scenario 1: Pod in `CrashLoopBackOff` State

Domain: Workloads & Troubleshooting (15%)

Problem: A Pod repeatedly crashes and restarts.

Step 1: Identify the Crashing Pod

```
kubectl get pods
```

Step 2: Check Pod Logs

```
kubectl logs <pod-name>
kubectl logs <pod-name> -p   # previous container
```

Step 3: Inspect Events and Config

```
kubectl describe pod <pod-name>
```

Step 4: Fix and Reapply

```
kubectl apply -f <manifest>.yaml
```

Remember

Common causes: wrong entrypoint, missing env vars, failing probes, missing dependencies.

7.2.2 Scenario 2: Node in `NotReady` State

Domain: Cluster Maintenance & Troubleshooting (15%)

Problem: One or more nodes are unreachable.

Step 1: Confirm Node Status

```
kubectl get nodes
```

Step 2: SSH and Check Kubelet Logs

```
sudo journalctl -u kubelet -f
```

Step 3: Restart Critical Services

```
sudo systemctl daemon-reload
sudo systemctl restart kubelet
sudo systemctl restart containerd   # or docker
```

Step 4: Verify Recovery

```
kubectl get nodes
```

7.2.3 Scenario 3: Node Reporting `DiskPressure`

Domain: Cluster Maintenance & Troubleshooting (15%)

Step 1: Confirm Condition

```
kubectl describe node <node-name>
```

Step 2: Check Disk Usage

```
sudo du -sh /* | sort -h
df -h
```

Step 3: Clean Up Space

```
sudo docker system prune -af       # or crictl
sudo journalctl --vacuum-size=100M
sudo rm -rf /var/logcontainers/*.log
```

Step 4: Verify

```
df -h
```

Note: This section includes **3 Troubleshooting exercises**. You can unlock **4 additional lab-tested exercises** (7 total) in the premium version:
Gumroad: Conquer CKA Exam (Premium)
Payhip: Conquer CKA Exam (Premium)

8 Other Advanced Kubernetes Exercises for CKA® Preparation

This section covers advanced Kubernetes topics that often appear in the CKA® exam; resource quotas, node affinity, and anti-affinity. Mastering these ensures you can handle production-grade constraints and optimize pod placement under real-world scheduling rules.

8.1 Exercise 1: Implementing Resource Quotas in a Namespace

Domain: Workloads & Scheduling (10%) / Cluster Administration (10%)

Objective: Create a namespace and set a `ResourceQuota` to enforce CPU and memory limits for all workloads inside it.

Exam Thinking

When you see "limit resources per namespace" in the exam:

- You must create or edit a `ResourceQuota` object in that namespace.
- Quotas apply to aggregate resource requests/limits for all pods in that namespace.
- Always check if `LimitRange` is also required for per-pod defaults.

Step 1: Create Namespace

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: development
```

```
kubectl apply -f namespace.yaml
```

Step 2: Apply ResourceQuota

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-quota
  namespace: development
spec:
  hard:
    requests.cpu: "2"
    requests.memory: 4Gi
    limits.cpu: "4"
    limits.memory: 8Gi
```

```
kubectl apply -f resourcequota.yaml
```

Step 3: Verify Quota

```
kubectl get resourcequota dev-quota -n development
```

Always Remember

`ResourceQuota` sums the requests/limits of all pods in a namespace. If new pods exceed these totals, they will not be scheduled.

Exam Tip

If a pod fails to schedule with “exceeded quota” errors, check both the namespace’s `ResourceQuota` and `LimitRange` objects.

8.2 Exercise 2: Configuring Node Affinity for Pod Placement

Domain: Workloads & Scheduling (15%)

Objective: Deploy a pod that is restricted to run only on nodes with a specific label.

Exam Thinking

When given a requirement like "run only on SSD nodes":

- You must label the target node first.
- Use `nodeAffinity` in pod spec to ensure scheduling rules.
- If you see "avoid certain nodes," that's `anti-affinity`.

Step 1: Label the Node

```
kubectl label nodes <node-name> disktype=ssd
```

Step 2: Create Pod with Required Node Affinity

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
```

```
kubectl apply -f pod-affinity.yaml
```

Step 3: Verify Placement

```
kubectl get pod nginx-pod -o wide
```

Always Remember

`nodeAffinity` only affects where a pod can be scheduled, not where it can be moved later unless it is re-scheduled.

Exam Tip

If a pod stays in `Pending`, check that at least one node matches the affinity labels, missing labels are the most common cause.

9 Advanced Kubernetes Scheduling and Configuration Exercises

This section focuses on practical, exam-relevant Kubernetes concepts: autoscaling workloads, controlling scheduling with taints and tolerations, maintaining availability with Pod Disruption Budgets, and managing configuration data securely with ConfigMaps and Secrets.

9.1 Exercise 1: Implementing Horizontal Pod Autoscaler (HPA)

Domain: Workloads & Scheduling (15%)

Objective: Configure a Horizontal Pod Autoscaler to dynamically adjust pod replicas based on CPU utilization.

Exam Thinking

- HPA requires `metrics-server`; without it, scaling will not occur.
- Containers must define CPU/memory `requests`, otherwise metrics are unavailable.
- Always verify scaling using a load generator.

Step 1: Deploy Application

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  replicas: 1
  selector:
    matchLabels:
      run: php-apache
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
      - name: php-apache
        image: k8s.gcr.io/hpa-example
        resources:
          requests:
            cpu: 200m
          limits:
            cpu: 500m
```

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

Step 2: Create HPA

```bash
kubectl autoscale deployment php-apache \
  --cpu-percent=50 --min=1 --max=10
```

Step 3: Test Scaling

```
kubectl get hpa
```

Always Remember

HPA will not scale without active CPU load. Metrics must show utilization above the threshold for scaling.

Exam Tip

If scaling does not occur, check:

```
kubectl get apiservices | grep metrics
```

9.2 Exercise 2: Taints and Tolerations

Domain: Workloads & Scheduling (15%)

Objective: Control which pods run on specific nodes using taints and tolerations.

Exam Thinking

- Taints repel pods from a node.
- Tolerations allow specific pods to bypass the restriction.
- Tolerations do **not** force scheduling to that node; use `nodeAffinity` if required.

Step 1: Taint the Node

```
kubectl taint nodes <node-name> key=value:NoSchedule
```

Step 2: Create Pod with Matching Toleration

```
apiVersion: v1
kind: Pod
metadata:
  name: tolerant-pod
spec:
  tolerations:
  - key: "key"
    operator: "Equal"
    value: "value"
    effect: "NoSchedule"
  containers:
  - name: nginx
    image: nginx
```

```
kubectl apply -f tolerant-pod.yaml
```

Step 3: Verify Scheduling

```
kubectl get pod tolerant-pod -o wide
```

10 Conclusion

Passing the Certified Kubernetes Administrator (CKA®) exam is about more than theory, it's about being able to perform under time pressure while making accurate decisions. The scenarios in this book were designed to push you into thinking like you're already in the exam: read the requirements, choose the fastest correct approach, and validate your work without hesitation.

If you have worked through every scenario here, you've already touched most of the practical patterns that appear in the exam. However, doing them once is not enough, repeat them in your own lab until the commands and workflows become second nature.

Exam-Day Strategy:

- **Read the question carefully.** Understand the namespace, context, and constraints before typing anything.
- **Use the docs smartly.** The official Kubernetes documentation is available during the exam, use the search bar, not endless navigation.
- **Validate constantly.** Run `kubectl get` and `kubectl describe` after changes to confirm correctness.
- **Don't over-engineer.** The exam rewards correct, working solutions, not overly complex YAML.
- **Skip and return.** If a task is dragging past 5 minutes, move on and come back later with fresh eyes.
- **Use imperative commands to save time.** Then switch to YAML editing only when necessary.
- **Practice under timed conditions.** Simulate the 2-hour limit in your own environment to build speed and confidence.

The work you've done here mirrors real-world Kubernetes administration. Once certified, keep applying these skills, not just for exam performance, but for building and running production-grade clusters efficiently.

11 References

These are the official and external resources actually referenced within the scenarios in this book:

- Kubernetes® Official Documentation
- Kubernetes Network Policies Documentation
- Kubernetes Persistent Volumes and PersistentVolumeClaims
- Kubernetes StorageClass Documentation
- Kubernetes Resource Quotas
- Kubernetes Taints and Tolerations
- Kubernetes Horizontal Pod Autoscaler
- etcd Official Documentation
- Kubernetes GitHub Repository
- Certified Kubernetes Administrator (CKA) Exam Overview

Stay Connected

If you enjoyed this book and want more practical Kubernetes content, you can follow my writing on Medium:

medium.com/@DynamoDevOps

I regularly publish new scenarios, troubleshooting deep-dives, and tips for real-world cluster operations and certification success.