Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 29, 2026

Kubernetes Cluster Lifecycle and Version Upgrades

Plan and perform safe kubeadm cluster upgrades by reading version skew, upgrading the control plane before nodes, draining deliberately, and verifying each transition.

This is Learn post 20 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. The previous lessons built a kubeadm control plane and joined workers. This lesson keeps that cluster supportable as Kubernetes releases move forward.

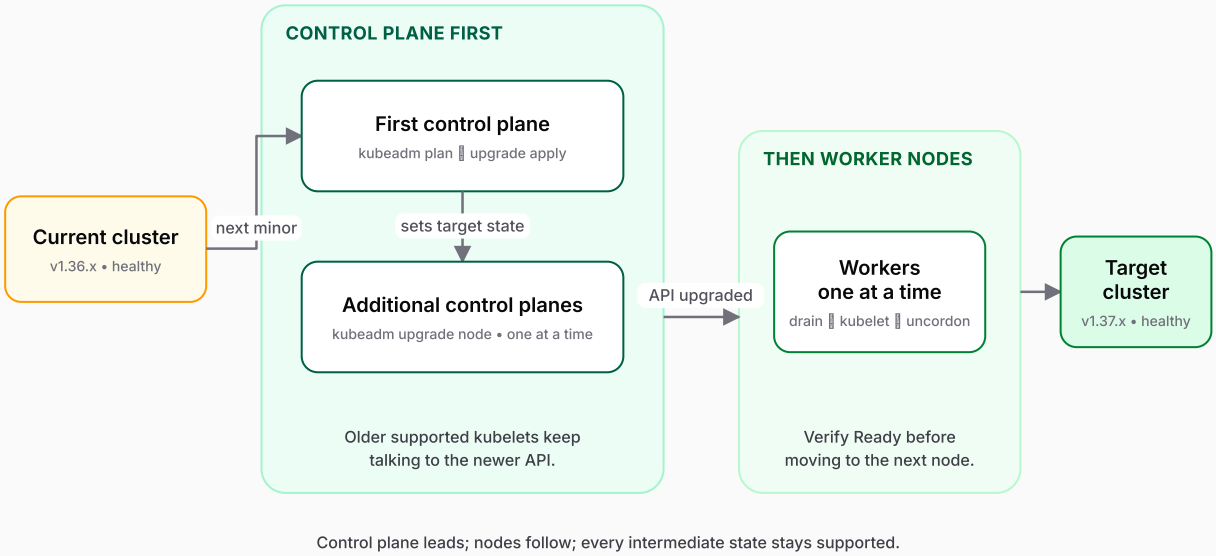
A cluster upgrade is not one package update. The API server, other control plane components, kubelets, kube-proxy, kubectl, and add-ons are separate moving parts. The administrator's job is to move them through a supported sequence while preserving a working API and enough workload capacity.

What you'll learn

- Read cluster component versions and distinguish kubeadm, kubelet, kubectl, and control plane versions.
- Use version skew rules to explain why the control plane moves before worker kubelets.
- Prepare, execute, and verify a one-minor-version kubeadm upgrade on control plane and worker nodes.
- Recognize what kubeadm changes, what remains a host package operation, and what must be managed separately.

The mental model: the API moves first, nodes follow

> Move one minor release at a time, move the control plane before kubelets, and return each node to health before moving the next one.



A safe Kubernetes upgrade moves the API before node agents

The first control plane establishes the target version. Additional control planes and then workers follow one at a time, with a health checkpoint after every node.

The ordering follows compatibility, not ceremony. Kubelets talk to the API server, so an older supported kubelet can continue talking to a newly upgraded API server. A newer kubelet must not talk to an older API server. Because the API server moves first, every intermediate state stays inside the supported version-skew window.

The second safety boundary is a node. Cordon prevents new scheduling, drain evicts suitable workload Pods, the host software changes, and uncordon admits new scheduling again. Repeating that cycle preserves capacity and gives you a checkpoint after every node.

Know which version you are looking at

Kubernetes versions use major.minor.patch, such as v1.37.2. A patch upgrade stays within one minor release and normally delivers fixes. A minor upgrade, such as v1.36.x to v1.37.x, can add features, deprecate behavior, and change component configuration. The supported kubeadm path does not skip minor versions.

The names are easy to blur, but their responsibilities are different:

- kubeadm is the lifecycle tool. Its version determines which upgrade workflow and target versions it understands.
- kube-apiserver represents the cluster's control plane version and leads the upgrade order.
- kubelet is the node agent. It is an operating-system package and is restarted separately after kubeadm updates the node's configuration.
- kubectl is a client. Upgrading it does not upgrade the cluster.
- The container runtime and third-party add-ons have their own release and compatibility policies; kubeadm does not upgrade them for you.

Start an inventory from a machine with cluster access, then check the local lifecycle tool on the node where you will run it:

Version inventory · bash

```
kubectl version
kubectl get nodes -o wide
kubectl get nodes \
  -o custom-
columns='NODE:.metadata.name,KUBELET:.status.nodeInfo.kubeletVersion,RUNTIME:.status.nodeInfo.containerRuntimeVersio
n'

# Run on the node whose kubeadm package you are inspecting.
kubeadm version
```

kubectl version shows the client and API server versions. The Node table reports each kubelet version, so mixed versions during a rolling node upgrade are visible rather than surprising. The runtime column is useful context, but a Kubernetes upgrade does not

imply a container runtime upgrade.

Version skew is a compatibility budget

Version skew is the supported difference between communicating components. It allows rolling upgrades, but it is not a reason to leave the cluster mixed indefinitely. For current supported Kubernetes releases, remember these administrator-facing rules:

- kubelet must not be newer than kube-apiserver and may be up to three minor versions older.
- kube-proxy must not be newer than kube-apiserver and may be up to three minor versions older.
- kube-controller-manager and kube-scheduler should match kube-apiserver, but may be one minor version older during a live upgrade.
- kubectl is supported within one minor version older or newer than kube-apiserver.

In a highly available control plane, mixed API server versions narrow the safe ranges because clients and components may reach either version. The next lesson teaches high-availability architecture in depth; for now, treat every intermediate version combination as something that must satisfy the policy.

Prepare the change before touching packages

A safe lifecycle starts before the maintenance window. Choose the latest patch of the next minor release, read its release notes and deprecation guidance, and verify that your operating system, container runtime, Container Network Interface (CNI) plugin, Container Storage Interface (CSI) driver, admission webhooks, and other add-ons support the target.

Also confirm that the cluster is healthy and has enough spare capacity for a drained node. A drain moves disruption elsewhere; it does not create CPU or memory. Check the API, nodes, and system Pods before creating a baseline:

Pre-upgrade health baseline · bash

```
kubectl get --raw='/readyz?verbose'
kubectl get nodes
kubectl -n kube-system get pods -o wide
kubectl get pods -A --field-selector=status.phase!=Running,status.phase!=Succeeded
```

The verbose ready endpoint checks API server dependencies. Every Node should be Ready, kube-system Pods should have their expected replicas, and the final command should not reveal unexplained failed or pending workloads. Fix an unhealthy starting state before adding upgrade variables.

Back up the state that matters

For a stacked-etcd kubeadm control plane, etcd contains Kubernetes API state. The following command uses the certificates mounted for the local etcd member to create and inspect a snapshot:

Control plane host · bash

```
sudo mkdir -p /var/backups/kubernetes

sudo ETCDCTL_API=3 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 /var/backups/kubernetes/etcd-before-v1.37.db

sudo etcdutl --write-out=table snapshot status \
  /var/backups/kubernetes/etcd-before-v1.37.db
```

A valid snapshot gives you a recovery point for API objects. It does not contain application data stored in external databases or persistent volumes, so those systems need their own backups. Copy the snapshot and relevant kubeadm configuration or PKI material to protected storage outside the node. Restoring etcd is a separate recovery operation, not part of a normal upgrade.

Worked demonstration: upgrade v1.36 to v1.37

This demonstration uses a Debian-based kubeadm cluster with cp-1, worker-1, and worker-2. It moves from the latest v1.36 patch to the latest v1.37 patch. Replace every x below with the patch package version shown by your package manager. The repository at pkgs.k8s.io is minor-version-specific, so enable the v1.37 repository before looking for v1.37 packages.

> Do not paste a placeholder package version into a real upgrade. Discover the available patch, select it deliberately, and use the same target patch across the cluster.

1. Upgrade kubeadm on the first control plane

Upgrade kubeadm before asking it to plan the target release. Holding Kubernetes packages prevents unattended package upgrades from changing component versions outside this controlled sequence.

cp-1 · bash

```
sudo apt update
sudo apt-cache madison kubeadm

# Replace x with the selected v1.37 patch.
sudo apt-mark unhold kubeadm
sudo apt-get install -y kubeadm='1.37.x-*'
sudo apt-mark hold kubeadm

kubeadm version
```

Only the kubeadm binary changed. The API server and kubelet are still on the old version, which is expected at this checkpoint.

2. Let kubeadm validate the route

```
cp-1 · bash

sudo kubeadm upgrade plan
```

The plan checks whether the cluster is upgradeable, applies skew rules, shows available targets, and reports component configuration versions. Read the result. A preflight failure is information about an unsafe starting condition, not a prompt to add an ignore flag automatically.

You can also preview changes to static Pod manifests before applying the target:

```
cp-1 · bash

sudo kubeadm upgrade diff v1.37.x
```

3. Apply the control plane upgrade

```
cp-1 · bash

# Replace x with the exact patch selected from the plan.
sudo kubeadm upgrade apply v1.37.x
```

kubeadm runs preflight checks, enforces skew, downloads or verifies the target images, and rewrites the static Pod manifests under /etc/kubernetes/manifests. The kubelet already watches that directory, so it replaces the API server, controller manager, scheduler, and local etcd static Pods from the new manifests. kubeadm also updates managed configuration and the kubeadm-managed CoreDNS and kube-proxy resources when the upgrade sequence permits it.

This connects the upgrade to the static Pod model from Learn post 15: kubeadm changes desired state in local manifest files, and kubelet reconciles the running control plane containers. kubeadm also writes temporary manifest and local-etcd backups under /etc/kubernetes/tmp, but those operational backups do not replace your independent backup plan.

After apply succeeds, confirm the API and control plane Pods before continuing:

cp-1 · bash

```
kubectl get --raw='/readyz?verbose'
kubectl -n kube-system get pods -l tier=control-plane -o wide
kubectl get nodes -o wide
```

At this moment, cp-1 can report a v1.37 API server while its kubelet still reports v1.36. That temporary state is valid because the kubelet is older than the API server and within the supported skew.

4. Drain cp-1, then upgrade its kubelet and kubectl

A minor kubelet upgrade requires draining the node first. Drain is an API operation, so run it from any healthy administrative shell. The ignore-daemonsets flag acknowledges that DaemonSet Pods are node-scoped and are not evicted by drain.

Administrative shell, then cp-1 · bash

```
kubectl drain cp-1 --ignore-daemonsets

# On cp-1; replace x with the selected patch.
sudo apt-mark unhold kubelet kubectl
sudo apt-get install -y kubelet='1.37.x-*' kubectl='1.37.x-*'
sudo apt-mark hold kubelet kubectl
sudo systemctl daemon-reload
sudo systemctl restart kubelet
```

Static control plane Pods are not ordinary scheduler-managed workloads, so draining does not upgrade them. kubeadm already replaced their manifests. This step protects other Pods while the node-local kubelet package and service change.

Verify and return cp-1 to service · bash

```
sudo systemctl --no-pager --full status kubelet
kubectl get node cp-1 -o wide
kubectl uncordon cp-1
kubectl get node cp-1
```

Do not uncordon by habit. First confirm that cp-1 is Ready and reports the target kubelet version. Uncordon only changes scheduling eligibility; it does not prove the node is healthy.

5. Upgrade each worker as a complete unit

On a worker, kubeadm upgrade node fetches the cluster configuration and upgrades this node's kubelet configuration. It does not replace the kubelet binary; the package manager still does that. Complete worker-1 and verify it before starting worker-2.

worker-1 · bash

```
sudo apt update
sudo apt-mark unhold kubeadm kubectl
sudo apt-get install -y kubeadm='1.37.x-*' kubectl='1.37.x-*'
sudo apt-mark hold kubeadm kubectl

sudo kubeadm upgrade node
```

Now move workloads away before restarting the node agent:

Administrative shell, then worker-1 · bash

```
kubectl drain worker-1 --ignore-daemonsets

# On worker-1; replace x with the selected patch.
sudo apt-mark unhold kubelet
sudo apt-get install -y kubelet='1.37.x-*'
sudo apt-mark hold kubelet
sudo systemctl daemon-reload
sudo systemctl restart kubelet
```

Verify and return worker-1 · bash

```
kubectl get node worker-1 -o wide
kubectl get pods -A -o wide --field-selector spec.nodeName=worker-1
kubectl uncordon worker-1
kubectl get node worker-1
```

The first command must show Ready and the target kubelet version. The Pod listing lets you see node-local DaemonSet Pods and newly scheduled workloads. Only after those signals look normal should you repeat the same sequence for worker-2.

What changes for additional control plane nodes

In a multi-control-plane cluster, upgrade one control plane node at a time. Upgrade kubeadm on the first node and run kubeadm upgrade apply once. On each additional control plane node, upgrade kubeadm and run kubeadm upgrade node instead; then drain that node, upgrade kubelet and kubectl, restart kubelet, verify, and uncordon. Wait for one node to become healthy before touching the next.

The availability mechanics depend on the control plane and etcd topology. That architecture belongs to Learn post 21; the lifecycle rule here is that kubeadm apply establishes the cluster-wide target, while kubeadm node updates each additional node's local control plane and kubelet configuration.

Verify the cluster, not just the command exit code

A successful package installation proves that files changed. A successful kubeadm command proves that its phases completed. Neither alone proves that the whole cluster is serving workloads correctly. Compare the final state with the pre-upgrade baseline:

Post-upgrade verification · bash

```
kubectl version
kubectl get --raw='/readyz?verbose'
kubectl get nodes -o wide
kubectl -n kube-system get pods -o wide
kubectl get pods -A --field-selector=status.phase!=Running,status.phase!=Succeeded
kubectl get events -A --sort-by=.metadata.creationTimestamp | tail -30
```

Look for the target server and kubelet versions, Ready nodes, healthy control plane and add-on Pods, no unexplained non-running workloads, and no repeating warning events. Then perform an application-level check appropriate to the cluster; Kubernetes component health cannot prove that an application dependency is working.

Understand common stopping points

The target version is absent from apt

The pkgs.k8s.io repository is scoped to a Kubernetes minor release. If apt-cache does not list v1.37 packages, verify that the configured repository points to the v1.37 channel and refresh package metadata. Do not install an arbitrary version from a mismatched repository.

kubeadm upgrade plan rejects the transition

Confirm the current and target minor versions, node readiness, control plane health, and the installed kubeadm version. Skipped minor releases and unhealthy prerequisites should change the plan. Avoid treating --ignore-preflight-errors as a generic fix because it removes a safety check without correcting the condition.

Drain is blocked

Read the drain error. A PodDisruptionBudget may be protecting availability, or an unmanaged Pod may require an explicit decision. Flags such as --delete-emptydir-data and --force have data-loss or ownership implications. The command explains the blocker so the administrator can resolve it deliberately rather than bypass it blindly.

The kubelet does not return after restart

Affected node · bash

```
sudo systemctl --no-pager --full status kubelet
sudo journalctl -u kubelet -n 100 --no-pager
sudo kubeadm upgrade node
```

Inspect the service error before repeating actions. kubeadm upgrade operations are designed to be rerunnable, so after correcting an interrupted or local configuration problem, rerunning the appropriate upgrade command can reconcile the intended state. Do not uncordon the node until kubelet is healthy and the Node is Ready.

What kubeadm does not own

kubeadm manages the kubeadm-defined control plane, kubelet configuration, and the kubeadm-installed CoreDNS and kube-proxy resources. It does not upgrade your operating system, container runtime, CNI provider, CSI drivers, ingress controller,

monitoring stack, or application workloads. Those dependencies belong in the same change plan because they interact with Kubernetes, but each follows its own documentation and compatibility matrix.

This boundary is a useful lifecycle habit: inventory ownership before changing anything. If kubeadm owns the configuration, use kubeadm's supported workflow. If an add-on or platform owns it, follow that owner's upgrade workflow. Avoid turning a Kubernetes version upgrade into an unreviewed full-stack upgrade.

What to remember

- An upgrade is an ordered transition among components, not one package transaction.
- Do not skip minor versions; select the latest patch of the next minor and read its release guidance.
- Upgrade the first control plane with kubeadm upgrade apply, additional control planes with kubeadm upgrade node, then workers.
- Drain before a minor kubelet upgrade, verify Ready and the expected version, then uncordon.
- Backups, preflight health, add-on compatibility, spare capacity, and post-change verification are part of the upgrade—not optional decoration around it.

Official documentation

Upgrading kubeadm clusters is the version-specific source for the supported control plane workflow and package commands.

Upgrading Linux nodes provides the current worker-node sequence.

Version Skew Policy defines supported component combinations and the resulting upgrade order.

Operating etcd clusters for Kubernetes documents snapshots, verification, restoration, and etcd upgrade considerations.

Safely Drain a Node explains eviction behavior and the flags that require deliberate data or availability decisions.