Move2Cloud

• WHITE PAPER — 2027 EDITION

CLOUD & KUBERNETES

Upgrading Kubernetes without service interruption

An operational guide to absorbing the pace of three minor versions a year without a maintenance window: compatibility, cutover strategies, controlled draining and automation.

11 pages · Compatibility matrix, strategy comparison, runbook and pre-upgrade checklist included

3	14 months	n-3
minor versions released upstream per year	of upstream support per minor version	kubelet / API server skew tolerated

PREPARED BY
Move2Cloud — Cloud & Infrastructure Consulting, Paris

PUBLICATION
September 2026

CONTENTS

What you will find in this document

This document starts from the real constraints of the Kubernetes lifecycle and leads to a repeatable, tested and automated upgrade roadmap.

01 **Executive summary — the essentials on one page** p.3

02 **The tempo set by upstream** p.4

03 **Version skew and deprecated APIs** p.5

04 **Upgrade strategies compared** p.6

05 **Draining, PDBs and awkward workloads** p.7

06 **Automate, test, validate continuously** p.8

07 **Self-assessment grid** p.9

08 **A 5-step roadmap** p.10

09 **Conclusion & about Move2Cloud** p.11

Who is this document for? CTOs, platform leads and SREs running one or more Kubernetes clusters in production (managed EKS, AKS, GKE or self-hosted) who must keep up with the release cadence without degrading availability or freezing application delivery.

EXECUTIVE SUMMARY

The essentials on one page

Keeping up with Kubernetes' upstream cadence without ever interrupting service has become an operational skill, not an exceptional project.

Kubernetes has released **three minor versions per year** since 2021, and each version is maintained for only **14 months** (12 months of active support, 2 months of maintenance). A platform that upgrades just once a year therefore lives structurally at end of support, without upstream security fixes and often outside the guarantees of its managed provider.

The constraint is hardening on the provider side: AWS charges EKS extended support at **$0.60 per cluster per hour**, roughly six times the standard rate, before an automatic upgrade at the end of the line; AKS and GKE apply equivalent mechanisms. Add to this API debt — the removal of Ingress extensions/v1beta1 in 1.22 and of PodSecurityPolicy in 1.25 — and the independent schedules of CNIs, CSIs and operators, which turn a deferred version jump into a remediation project.

The good news is that Kubernetes' compatibility policy makes the operation predictable: the control plane moves up one minor version at a time and kubelets can stay **up to 3 minor versions** behind the API server, which opens a genuine window for progressive migration. This document describes how to exploit that margin: detect obsolete APIs before the switch, choose between in-place and blue/green node pools, calibrate PodDisruptionBudgets and draining, then automate and verify the whole thing.

3
minor versions released each year by upstream

14 months
of support per minor version (12 active months + 2 of maintenance)

n-3
tolerated gap between kubelet and API server (version skew)

The question this document answers is simple to state and hard to handle: how do you absorb three version upgrades a year on production clusters, with no maintenance window and no delivery freeze? The sections that follow set out a sequenced method — compatibility audit, migration strategy, controlled draining, automation and verification — together with a directly usable checklist.

Copyright Move2Cloud — move2cloud.fr

SECTION 02

The tempo set by upstream

The question is no longer whether to upgrade, but how often and under what conditions. Three dynamics are converging today to turn the Kubernetes upgrade from a one-off project into a permanent operational routine.

1. Three releases a year, fourteen months of patches

Since 2021, Kubernetes has shipped **three minor releases per year** and provides patches for only 14 months (12 months of active support + 2 months of maintenance). In practice, a cluster that skips two cycles finds itself out of support in less than a year.

2. Managed services are closing the escape hatch

Managed offerings no longer let you drift: AWS charges EKS extended support at $0.60 per cluster per hour, six times the standard rate, before an automatic upgrade at the end of the road. AKS and GKE apply the same forced-migration logic.

3. API debt is payable in cash

Every release removes APIs: Ingress extensions/v1beta1 in 1.22, PodSecurityPolicy in 1.25, flowcontrol resources thereafter. On top of that come the independent schedules of CNIs, CSIs, operators and ingress controllers, rarely aligned with upstream's.

“ A Kubernetes cluster is never up to date: it is only more or less behind a tempo it does not set. ”

What remains is to translate this calendar constraint into engineering practice: compatibilities to verify, migration strategy, draining and automation.

Copyright Move2Cloud — move2cloud.fr

SECTION 03

Version skew and deprecated APIs

Kubernetes allows a bounded skew between its components, and that window — not the team's calendar — determines the order and pace of your upgrades.

COMPONENT	TOLERATED SKEW RELATIVE TO THE KUBE-APISERVER
kubelet and kube-proxy	Up to three minor versions behind the apiserver since Kubernetes 1.28 (two previously), never ahead; kube-proxy follows the version of its kubelet.
controller-manager, scheduler, cloud-controller-manager	At most one minor version behind the apiserver, and never ahead: these components are updated with the control plane, not after it.
kubectl, clients and CI	Supported within ±1 minor version of the apiserver: a build image frozen two years ago silently breaks pipelines on upgrade day.

> **Key takeaway:** Don't just validate the manifests in your Git repository: enable audit logs and the `apiserver_requested_deprecated_apis` metric for at least one full cycle (including monthly batch jobs) to capture the actual calls made by operators, Helm charts and CI scripts before you lock in the upgrade date.

Deprecated APIs are detected in traffic, not in the repository

The costliest trap is not skew, which is mechanical and verifiable, but API removal: the upstream policy guarantees advance notice (a minimum of three releases for beta APIs, twelve months for GA), after which the call simply returns an error. Past removals are a reminder of how broad the issue is: `extensions/v1beta1` Ingress and `apiextensions/v1beta1` CRD in 1.22, PodSecurityPolicy and `batch/v1beta1` CronJob in 1.25, `autoscaling/v2beta2` in 1.26. The culprits are rarely your Deployments: they are old Helm charts, third-party operators shipping an outdated client-go, CRD conversion webhooks and backup jobs. A static scan of the repository (kubent, Pluto) covers the declarative side; only audit logs cover the imperative one.

Copyright Move2Cloud — move2cloud.fr

SECTION 04

Upgrade strategies compared

Three families of strategies coexist in production, and what sets them apart is not their elegance but how long it takes to roll back.

STRATEGY	ROLLBACK	HOW IT WORKS	BEST SUITED TO
Rolling in-place	Impossible on the control plane, slow on the nodes	The provider upgrades the apiserver, then nodes are replaced in waves with surge and draining	Standard application clusters, stateless workloads well covered by PDBs
Blue/green node pools	A few minutes, through re-scheduling	A new pool is created on version n+1, the old nodes are cordoned then drained progressively	Sensitive workloads, presence of StatefulSets or heterogeneous GPU/spot nodes
Blue/green cluster	Immediate, by switching traffic	A brand-new cluster is rebuilt through GitOps, validated, then traffic is moved via DNS or a global load balancer	Jumps of several versions, CNI/CSI overhauls, critical multi-tenant platforms

Common pitfall: Downgrading a control plane minor version is not supported: etcd schemas only migrate forward. Any in-place strategy must therefore rely on a roll-forward plan, a verified etcd snapshot and tested application backups — not on the illusion of a rollback button.

Control plane first, nodes at the pace you choose

Whichever option you choose, the order is non-negotiable: control plane first, nodes next, one minor version at a time, no skipping. Managed offerings make this separation explicit — GKE with its release channels (rapid, regular, stable) and maintenance windows, EKS with fourteen months of standard support extendable through paid extended support, AKS with its LTS versions. Use it: let the provider manage the control plane in a stable channel with a blackout window over your freeze periods, and keep control of node rotation, which is the only phase where your users can actually notice anything. On the nodes, a 30% surge shortens the total duration without creating capacity pressure, provided the cluster autoscaler has the quota to provision the extra instances.

Copyright Move2Cloud — move2cloud.fr

SECTION 05

Draining, PDBs and awkward workloads

Draining is the only moment in an upgrade when disruption becomes visible: that is where the 502s and the pods stuck for hours concentrate.

The PDB that blocks instead of protecting

A PDB with `maxUnavailable: 0`, or a single-replica Deployment covered by `minAvailable: 1`, blocks eviction indefinitely: the drain loops without ever succeeding. Aim for at least two replicas and `maxUnavailable: 1`.

Connections cut too early

When the pod is deleted, kube-proxy and the ingress controller have not yet removed the endpoint. A `preStop` with a pause of about ten seconds, followed by a clean shutdown within the `terminationGracePeriodSeconds`, removes most 502 errors.

StatefulSets, local storage and batch jobs

Quorums (etcd, Kafka, Zookeeper), local volumes and long-running jobs require dedicated handling: `topologySpreadConstraints` to avoid co-location, and the `cluster-autoscaler.kubernetes.io/safe-to-evict: "false"` annotation for non-interruptible batch jobs.

SETTING	EFFECT ON DRAINING
Grace period and timeout aligned	The `terminationGracePeriodSeconds` must stay below the drain timeout, otherwise the node is deleted while the pod is still alive.
DaemonSets and emptyDir	`kubectl drain` ignores DaemonSets but refuses pods with emptyDir without `--delete-emptydir-data`: decide beforehand, not during.

Copyright Move2Cloud — move2cloud.fr

SECTION 06

Automate, test, validate continuously

Three minor versions a year means one upgrade per quarter: at that pace, only automation keeps the exercise from turning back into a project.

A disposable n+1 cluster in every pipeline

Version n+1 must enter CI before it enters production. An ephemeral cluster (kind, k3d) provisioned on every merge, onto which all charts and operators are deployed via Argo CD or Flux, detects a removed API, an incompatible webhook or an outdated CSI driver within minutes. Add a matrix that explicitly tests maximum skew — kubelet three minors behind — because that is the cluster's real configuration during several days of rotation. The same pipeline must re-run the kubent/Pluto scans and fail on any use of an API removed in the target version.

Exit gates based on SLOs, not on intuition

The switchover must be driven by signals, not by a manual checklist. At a minimum, instrument apiserver latency and error rate, etcd health, the number of unschedulable pods and the counter of deprecated APIs called; then make the continuation of node rotation conditional on the error budget of your application SLOs. Concretely: the next wave only starts if the indicators stay green for a defined observation period, otherwise the pipeline pauses with the pool partially migrated — a perfectly viable state thanks to the tolerated skew. Continuous rotation tools (Karpenter with its disruption budgets, Kured for reboots) apply the same logic day to day.

> **Key takeaway:** An upgrade repeated four times a year is an incident; node rotation performed every week is a routine. Automate rotation before automating the upgrade: it is rotation that reveals badly tuned PDBs and unclean shutdowns, calmly, outside the critical window.

The ultimate goal is not to succeed at an upgrade, but to make the question irrelevant: when version n+1 is validated in CI on an ephemeral cluster, when the control plane follows a stable channel within a maintenance window, and when nodes rotate continuously under PDB and error-budget constraints, moving up a minor version becomes an ordinary operational event. That is the point at which the team stops accumulating technical lag — and regains the right to adopt recent features rather than chasing API removals.

Copyright Move2Cloud — move2cloud.fr

SECTION 07

Self-assessment grid

Review these six points before planning your next minor version upgrade, cluster by cluster.

- [ ] **Do you know the End of Life date of the minor version currently running in production on each of your clusters?** Without that deadline, the upgrade remains a permanent intention that nothing ever triggers, until the cluster falls out of support and out of security fixes.
- [ ] **Have you instrumented `apiserver_requested_deprecated_apis` and audit logs over a full cycle, monthly batches included, rather than relying on the contents of your Git repositories alone?** Calls to deprecated APIs most often come from operators, Helm charts or CI scripts that are invisible in versioned manifests.
- [ ] **Does every exposed application have a PodDisruptionBudget consistent with its replica count, and a `terminationGracePeriodSeconds` aligned with the real duration of its requests?** Draining is where the disruption becomes visible: a missing PDB causes 502s, while a badly calibrated PDB blocks the node for hours.
- [ ] **Have you restored — not merely produced — an etcd snapshot within the last six months?** Since downgrading a control plane minor version is not supported, a backup that has never been tested is not a contingency plan.
- [ ] **Are you able to create an ephemeral n+1 cluster from your CI and deploy your entire stack onto it without any manual intervention?** This is the only way to discover operator, CRD and webhook regressions before production rather than during it.
- [ ] **Are your nodes immutable and replaced regularly, rather than updated in place and kept for months?** A continuously renewed node fleet turns the data plane upgrade into an already proven routine instead of an exceptional operation.

> **How to read your answers:** One or two "no" answers point to blind spots to close before the window; three "no" answers or more indicate that the process must be industrialised before attempting the upgrade.

Copyright Move2Cloud — move2cloud.fr

SECTION 08

A 5-step roadmap

This trajectory unfolds cluster by cluster, without freezing application releases: each step produces a usable deliverable even if the next one is postponed.

1

Map versions and deadlines

For each cluster, record the control plane version, the kubelet versions, the end-of-support date and the actual skew gap. Compare this table with the calendar of three minor versions a year to identify the clusters already outside the window.

2

Measure deprecated usage

Enable audit logs and expose `apiserver_requested_deprecated_apis` over at least one full cycle, monthly batches included. Trace every call back to its source — operator, Helm chart, CI script — and fix it at source before setting a date.

3

Make draining reliable

Publish or review the PodDisruptionBudget of every exposed workload, align `terminationGracePeriodSeconds` with the real duration of requests, and identify the singletons and StatefulSets that require manual handling during draining.

4

Validate on an ephemeral cluster

Automate the creation of an ephemeral n+1 cluster from CI, deploy the full stack onto it — CNI, ingress, operators, CRDs, applications — and make this test a merge blocker on the platform repository.

5

Switch over, then set the cadence

Run the control plane upgrade in a short window, with a verified etcd snapshot and a roll-forward plan in hand, then let node renewal run continuously under PDB and error-budget constraints. Repeat at the pace of upstream releases.

Copyright Move2Cloud — move2cloud.fr

CONCLUSION

Making the upgrade a non-event

Organisations that treat the upgrade as a continuous process rather than an annual project gain three things at once: they stay within the support window and therefore within the scope of security fixes, they eliminate the night-time maintenance windows that wear teams out, and they regain the ability to adopt recent features instead of suffering API removals. The pace of three minor versions a year is not a constraint to be endured: it is a cadence that, once absorbed by tooling, guarantees that no technical gap builds up silently.

Across several clusters and several product teams, the real question is no longer technical but organisational: who decides the date, who owns the error budget, and how are application teams informed of the deprecations that concern them. The platforms that keep up are those where the upgrade is an explicit contract between the platform team and its internal consumers — documented, tooled and repeated — not a negotiation restarted from scratch with every version.

Move2Cloud	is a consulting firm specialised in infrastructure, cloud and IT transformation. We support organisations in the audit, the modernisation and continuous operation of their cloud environment.
Contact	Move2Cloud — Paris · move2cloud.fr
Next step	A two- to three-week diagnostic — an inventory of versions and deadlines, real measurement of deprecated API usage, a PDB review and an etcd restore test — is enough to turn the "no" answers from the previous grid into a dated, costed upgrade plan.

Sources

Kubernetes — Version Skew Policy, official documentation — Kubernetes — Deprecated API Migration Guide — Kubernetes SIG Release — Release Cycle and Patch Releases — Kubernetes — Safely Drain a Node & Disruptions (PodDisruptionBudget) — CNCF — Annual Survey, recent editions

Document written by Move2Cloud as an example and illustration, for educational purposes. Skew policies, release cadences and end-of-support calendars change with every version: check the upstream documentation before any planning.