Move2Cloud

• WHITE PAPER — ÉDITION 2027

CLOUD & KUBERNETES

Mettre à jour Kubernetes sans interruption de service

Un guide opérationnel pour absorber le rythme de trois versions mineures par an sans fenêtre de maintenance : compatibilités, stratégies de bascule, drainage maîtrisé et automatisation.

11 pages · Matrice de compatibilité, comparatif de stratégies, runbook et checklist pré-upgrade inclus

3
versions mineures publiées par an en amont

14 mois
de support upstream par version mineure

n-3
d'écart kubelet / API server toléré

PRÉPARÉ PAR
Move2Cloud — Conseil Cloud & Infrastructure, Paris

PUBLICATION
Septembre 2026

SOMMAIRE

Ce que vous allez trouver dans ce document

Ce document part des contraintes réelles du cycle de vie Kubernetes pour aboutir à une feuille de route d'upgrade reproductible, testée et automatisée.

À qui s'adresse ce document ? Les CTO, responsables plateforme et SRE qui exploitent un ou plusieurs clusters Kubernetes en production (managés EKS, AKS, GKE ou auto-hébergés) et doivent tenir la cadence des versions sans dégrader la disponibilité ni geler les livraisons applicatives.

SYNTHÈSE

L'essentiel en une page

Tenir la cadence upstream de Kubernetes sans jamais interrompre le service est devenu une compétence d'exploitation, pas un projet exceptionnel.

Kubernetes publie **trois versions mineures par an** depuis 2021, et chaque version n'est maintenue que **14 mois** (12 mois de support actif, 2 mois de maintenance). Une plateforme qui n'upgrade qu'une fois par an vit donc structurellement en fin de support, sans correctif de sécurité upstream et souvent hors des garanties de son fournisseur managé.

La contrainte se durcit côté fournisseurs : AWS facture le support étendu EKS **0,60 $ par cluster et par heure**, soit environ six fois le tarif standard, avant une mise à niveau automatique en fin de course ; AKS et GKE appliquent des mécanismes équivalents. À cela s'ajoute la dette d'API — retrait d'Ingress extensions/v1beta1 en 1.22, de PodSecurityPolicy en 1.25 — et les calendriers indépendants des CNI, CSI et opérateurs, qui transforment un saut de version différé en chantier de remédiation.

La bonne nouvelle est que la politique de compatibilité de Kubernetes rend l'opération prévisible : le control plane monte d'une mineure à la fois et les kubelets peuvent rester **jusqu'à 3 versions mineures** en retrait de l'API server, ce qui ouvre une vraie fenêtre de bascule progressive. Ce document décrit comment exploiter cette marge : détecter les API obsolètes avant la bascule, choisir entre in-place et node pools bleu/vert, calibrer PodDisruptionBudgets et drainage, puis automatiser et vérifier le tout.

3
versions mineures publiées chaque année par l'upstream

14 mois
de support par version mineure (12 mois actifs + 2 de maintenance)

n-3
d'écart toléré entre kubelet et API server (version skew)

La question à laquelle répond ce document est simple à formuler et difficile à traiter : comment absorber trois montées de version par an sur des clusters de production, sans fenêtre de maintenance ni gel des livraisons ? Les sections qui suivent proposent une méthode séquencée — audit de compatibilité, stratégie de bascule, drainage maîtrisé, automatisation et vérification — assortie d'une checklist directement exploitable.

SECTION 02

Le tempo imposé par l'upstream

La question n'est plus de savoir s'il faut mettre à jour, mais à quelle fréquence et dans quelles conditions. Trois dynamiques convergent aujourd'hui pour transformer l'upgrade Kubernetes d'un projet ponctuel en routine d'exploitation permanente.

1. Trois releases par an, quatorze mois de patchs

Depuis 2021, Kubernetes publie **trois versions mineures par an** et n'assure les correctifs que pendant 14 mois (12 mois de support actif + 2 mois de maintenance). Concrètement, un cluster qui saute deux cycles se retrouve hors support en moins d'un an.

2. Les managés ferment la porte de sortie

Les offres managées ne laissent plus dériver : AWS facture le support étendu EKS 0,60 $ par cluster et par heure, soit six fois le tarif standard, avant mise à niveau automatique en fin de parcours. AKS et GKE appliquent la même logique de bascule forcée.

3. La dette d'API se paie comptant

Chaque version retire des API : Ingress extensions/v1beta1 en 1.22, PodSecurityPolicy en 1.25, des ressources flowcontrol ensuite. S'y ajoutent les calendriers propres des CNI, CSI, opérateurs et contrôleurs d'ingress, rarement alignés sur celui de l'upstream.

« Un cluster Kubernetes n'est jamais à jour : il est seulement plus ou moins en retard sur un tempo qu'il ne fixe pas. »

Reste à traduire cette contrainte de calendrier en pratique d'ingénierie : compatibilités à vérifier, stratégie de bascule, drainage et automatisation.

SECTION 03

Skew de versions et API dépréciées

Kubernetes autorise un décalage borné entre ses composants, et cette fenêtre — pas le calendrier de l'équipe — détermine l'ordre et le rythme des bascules.

COMPOSANT	DÉCALAGE TOLÉRÉ VIS-À-VIS DU KUBE-APISERVER
kubelet et kube-proxy	Jusqu'à trois versions mineures de retard sur l'apiserver depuis Kubernetes 1.28 (deux auparavant), jamais en avance ; kube-proxy suit la version de son kubelet.
controller-manager, scheduler, cloud-controller-manager	Au plus une version mineure de retard sur l'apiserver, et jamais en avance : ces composants se mettent à jour avec le control plane, pas après.
kubectl, clients et CI	Supportés à ±1 version mineure de l'apiserver : une image de build figée depuis deux ans casse silencieusement les pipelines le jour de la bascule.

À retenir : Ne validez pas seulement les manifestes de votre dépôt Git : activez les logs d'audit et la métrique `apiserver_requested_deprecated_apis` pendant au moins un cycle complet (batch mensuel inclus) pour capter les appels réels des opérateurs, charts Helm et scripts CI avant de figer la date d'upgrade.

Les API dépréciées se détectent dans le trafic, pas dans le dépôt

Le piège le plus coûteux n'est pas le skew, qui est mécanique et vérifiable, mais la disparition d'API : la politique upstream garantit un préavis (trois releases minimum pour les API beta, douze mois pour les GA), après quoi l'appel renvoie simplement une erreur. Les suppressions historiques rappellent l'ampleur du sujet : `extensions/v1beta1` Ingress et `apiextensions/v1beta1` CRD en 1.22, PodSecurityPolicy et `batch/v1beta1` CronJob en 1.25, `autoscaling/v2beta2` en 1.26. Les coupables sont rarement vos Deployments : ce sont les charts Helm anciens, les opérateurs tiers embarquant un client-go obsolète, les webhooks de conversion de CRD et les jobs de sauvegarde. Un scan statique du dépôt (kubent, Pluto) couvre le déclaratif ; seuls les logs d'audit couvrent l'impératif.

SECTION 04

Stratégies d'upgrade comparées

Trois familles de stratégies coexistent en production, et elles ne se distinguent pas par leur élégance mais par le temps qu'il faut pour revenir en arrière.

STRATÉGIE	RETOUR ARRIÈRE	FONCTIONNEMENT	À RÉSERVER À
Rolling in-place	Impossible sur le control plane, lent sur les nœuds	Le fournisseur met à jour l'apiserver, puis les nœuds sont remplacés par vagues avec surge et drainage	Clusters applicatifs standards, workloads stateless bien couverts par des PDB
Node pools bleu/vert	Quelques minutes, par re-scheduling	Un nouveau pool est créé en version n+1, les anciens nœuds sont cordonnés puis vidés progressivement	Workloads sensibles, présence de StatefulSets ou de nœuds GPU/spot hétérogènes
Cluster bleu/vert	Immédiat, par bascule du trafic	Un cluster neuf est reconstruit par GitOps, validé, puis le trafic est déplacé via DNS ou load balancer global	Sauts de plusieurs versions, refonte de CNI/CSI, plateformes critiques multi-tenant

> **Piège fréquent :** Un downgrade de version mineure du control plane n'est pas supporté : les schémas etcd migrent en avant seulement. Toute stratégie in-place doit donc s'appuyer sur un plan de roll-forward, un snapshot etcd vérifié et des sauvegardes applicatives testées — pas sur l'illusion d'un bouton retour.

Le control plane d'abord, les nœuds au rythme que vous choisissez

Quelle que soit l'option retenue, l'ordre est non négociable : le control plane d'abord, les nœuds ensuite, une version mineure à la fois, sans saut. Les offres managées matérialisent cette séparation — GKE avec ses release channels (rapid, regular, stable) et ses fenêtres de maintenance, EKS avec quatorze mois de support standard prolongeables par un support étendu payant, AKS avec ses versions LTS. Exploitez-la : laissez le fournisseur gérer le control plane dans un canal stable avec un blackout window sur vos périodes de gel, et gardez la main sur la rotation des nœuds, qui est la seule phase où vos utilisateurs peuvent réellement voir quelque chose. Sur les nœuds, un surge de 30 % réduit la durée totale sans créer de pression de capacité, à condition que le cluster autoscaler ait le quota pour provisionner les instances supplémentaires.

SECTION 05

Drainage, PDB et workloads capricieux

Le drainage est le seul moment de l'upgrade où l'interruption devient visible : c'est là que se concentrent les 502 et les pods bloqués pendant des heures.

Le PDB qui bloque au lieu de protéger

Un PDB avec `maxUnavailable: 0`, ou un Deployment à une seule réplique couvert par `minAvailable: 1`, bloque l'éviction indéfiniment : le drain boucle sans jamais réussir. Visez deux répliques minimum et `maxUnavailable: 1`.

Les connexions coupées trop tôt

À la suppression du pod, kube-proxy et l'ingress controller n'ont pas encore retiré l'endpoint. Un `preStop` avec une pause d'une dizaine de secondes, suivi d'un arrêt propre dans le `terminationGracePeriodSeconds`, supprime l'essentiel des erreurs 502.

StatefulSets, stockage local et batchs

Quorums (etcd, Kafka, Zookeeper), volumes locaux et jobs longs exigent un traitement dédié : `topologySpreadConstraints` pour éviter la co-localisation, et l'annotation `cluster-autoscaler.kubernetes.io/safe-to-evict: "false"` pour les batchs non interruptibles.

RÉGLAGE	EFFET SUR LE DRAINAGE
Grâce et timeout alignés	Le `terminationGracePeriodSeconds` doit rester inférieur au timeout de drain, sinon le nœud est supprimé pod encore vivant.
DaemonSets et emptyDir	`kubectl drain` ignore les DaemonSets mais refuse les pods à emptyDir sans `--delete-emptydir-data` : à décider avant, pas pendant.

SECTION 06

Automatiser, tester, valider en continu

Trois versions mineures par an signifient un upgrade par trimestre : à ce rythme, seule l'automatisation empêche l'exercice de redevenir un projet.

Un cluster jetable en n+1 dans chaque pipeline

La version n+1 doit entrer dans la CI avant d'entrer en production. Un cluster éphémère (kind, k3d) provisionné à chaque merge, sur lequel l'ensemble des charts et opérateurs sont déployés via Argo CD ou Flux, détecte en quelques minutes une API supprimée, un webhook incompatible ou un CSI driver trop ancien. Ajoutez-y une matrice qui teste explicitement le skew maximal — kubelet trois mineures en retard — car c'est la configuration réelle du cluster pendant plusieurs jours de rotation. Le même pipeline doit rejouer les scans kubent/Pluto et échouer sur toute utilisation d'API supprimée dans la version cible.

Des portes de sortie fondées sur les SLO, pas sur l'intuition

La bascule doit être pilotée par des signaux, pas par une checklist manuelle. Instrumentez au minimum la latence et le taux d'erreur de l'apiserver, la santé d'etcd, le nombre de pods non planifiés et le compteur d'API dépréciées appelées ; puis conditionnez la poursuite de la rotation des nœuds au budget d'erreur des SLO applicatifs. Concrètement : la vague suivante ne démarre que si les indicateurs sont verts pendant une durée d'observation définie, sinon le pipeline se met en pause avec le pool partiellement migré — un état parfaitement viable grâce au skew toléré. Les outils de rotation continue (Karpenter avec ses disruption budgets, Kured pour les redémarrages) appliquent la même logique au quotidien.

> **À retenir :** Un upgrade répété quatre fois par an est un incident ; une rotation de nœuds exécutée chaque semaine est une routine. Automatisez la rotation avant d'automatiser l'upgrade : c'est elle qui révèle les PDB mal réglés et les arrêts sales, à froid, hors fenêtre critique.

L'objectif final n'est pas de réussir un upgrade, mais de rendre la question sans intérêt : quand la version n+1 est validée en CI sur cluster éphémère, que le control plane suit un canal stable dans une fenêtre de maintenance et que les nœuds tournent en continu sous contrainte de PDB et de budget d'erreur, le passage d'une mineure devient un événement d'exploitation ordinaire. C'est à ce stade que l'équipe cesse d'accumuler du retard technique — et récupère le droit d'adopter les fonctionnalités récentes plutôt que de courir après les suppressions d'API.

SECTION 07

Grille d'auto-évaluation

Passez ces six points en revue avant de planifier votre prochaine montée de version mineure, cluster par cluster.

- [ ] **Connaissez-vous la date de fin de support (End of Life) de la version mineure actuellement en production sur chacun de vos clusters ?** Sans cette échéance, l'upgrade reste une intention permanente que rien ne déclenche, jusqu'à ce que le cluster sorte du support et des correctifs de sécurité.
- [ ] **Avez-vous instrumenté `apiserver_requested_deprecated_apis` et les logs d'audit sur un cycle complet, batchs mensuels compris, plutôt que de vous fier au seul contenu de vos dépôts Git ?** Les appels aux API dépréciées viennent le plus souvent d'opérateurs, de charts Helm ou de scripts CI invisibles dans les manifestes versionnés.
- [ ] **Chaque application exposée dispose-t-elle d'un PodDisruptionBudget cohérent avec son nombre de réplicas, et d'un `terminationGracePeriodSeconds` aligné sur la durée réelle de ses requêtes ?** C'est au drainage que l'interruption devient visible : un PDB absent provoque des 502, un PDB mal calibré bloque le nœud pendant des heures.
- [ ] **Avez-vous restauré, et pas seulement produit, un snapshot etcd au cours des six derniers mois ?** Le downgrade d'une mineure du control plane n'étant pas supporté, une sauvegarde jamais testée n'est pas un plan de secours.
- [ ] **Êtes-vous capable de créer un cluster éphémère en version n+1 depuis votre CI et d'y déployer la totalité de votre stack sans intervention manuelle ?** C'est le seul moyen de découvrir les régressions d'opérateurs, de CRD et de webhooks avant la production plutôt que pendant.
- [ ] **Vos nœuds sont-ils immuables et remplacés régulièrement, plutôt que mis à jour en place et conservés plusieurs mois ?** Un parc de nœuds renouvelé en continu transforme l'upgrade du data plane en routine déjà éprouvée au lieu d'une opération exceptionnelle.

> **Comment lire vos réponses :** Un ou deux « non » signalent des angles morts à combler avant la fenêtre ; trois « non » ou plus indiquent qu'il faut d'abord industrialiser le processus avant de tenter l'upgrade.

SECTION 08

Feuille de route en 5 étapes

Cette trajectoire se déroule cluster par cluster, sans gel des livraisons applicatives : chaque étape produit un livrable exploitable même si la suivante est repoussée.

1

Cartographier versions et échéances

Recensez pour chaque cluster la version du control plane, celle des kubelets, la date de fin de support et l'écart de skew réel. Confrontez ce tableau au calendrier des trois versions mineures annuelles pour identifier les clusters déjà hors fenêtre.

2

Mesurer les usages dépréciés

Activez les logs d'audit et exposez `apiserver_requested_deprecated_apis` sur au moins un cycle complet, batchs mensuels inclus. Remontez chaque appel à son émetteur — opérateur, chart Helm, script CI — et corrigez à la source avant de figer une date.

3

Fiabiliser le drainage

Publiez ou révisez les PodDisruptionBudget de chaque workload exposé, alignez les `terminationGracePeriodSeconds` sur la durée réelle des requêtes et identifiez les singletons et StatefulSets qui exigent un traitement manuel au drainage.

4

Valider en cluster éphémère

Automatisez la création d'un cluster éphémère en version n+1 depuis la CI, déployez-y la stack complète — CNI, ingress, opérateurs, CRD, applications — et faites de ce test un bloquant de merge sur le dépôt de la plateforme.

5

Basculer puis cadencer

Exécutez l'upgrade du control plane dans une fenêtre courte, snapshot etcd vérifié et plan de roll-forward en main, puis laissez le renouvellement des nœuds se faire en continu sous contrainte de PDB et de budget d'erreur. Répétez au rythme des versions amont.

CONCLUSION

Faire de l'upgrade un non-événement

Les organisations qui traitent l'upgrade comme un processus continu plutôt que comme un projet annuel gagnent trois choses à la fois : elles restent dans la fenêtre de support et donc de correctifs de sécurité, elles suppriment les fenêtres de maintenance nocturnes qui usent les équipes, et elles récupèrent la capacité d'adopter les fonctionnalités récentes au lieu de subir les suppressions d'API. Le rythme de trois versions mineures par an n'est pas une contrainte subie : c'est un cadencement qui, une fois absorbé par l'outillage, garantit qu'aucun écart technique ne s'accumule silencieusement.

À l'échelle de plusieurs clusters et de plusieurs équipes produit, la vraie question n'est plus technique mais organisationnelle : qui décide de la date, qui porte le budget d'erreur, et comment les équipes applicatives sont-elles prévenues des dépréciations qui les concernent. Les plateformes qui tiennent le rythme sont celles où l'upgrade est un contrat explicite entre l'équipe plateforme et ses consommateurs internes, documenté, outillé et répété — pas une négociation reprise à zéro à chaque version.

Move2Cloud	est un cabinet de conseil spécialisé en infrastructure, cloud et transformation IT. Nous accompagnons les organisations dans l'audit, la modernisation et l'exploitation continue de leur environnement cloud.
Contact	Move2Cloud — Paris · move2cloud.fr
Prochaine étape	Un diagnostic de deux à trois semaines — inventaire des versions et échéances, mesure réelle des API dépréciées, revue des PDB et test de restauration etcd — suffit à transformer les « non » de la grille précédente en un plan d'upgrade daté et chiffré.

Sources

Kubernetes — Version Skew Policy, documentation officielle — Kubernetes — Deprecated API Migration Guide — Kubernetes SIG Release — Release Cycle et Patch Releases — Kubernetes — Safely Drain a Node & Disruptions (PodDisruptionBudget) — CNCF — Annual Survey, éditions récentes