# Utrzymanie klastra

**Typ: procedura** **Dotyczy: Kubernetes 1.30+**

**Czas: ~40 min na jedną turę aktualizacji**

*Efekt: klaster zaktualizowany o jedną wersję pomniejszą, z kopią zapasową etcd wykonaną przed zmianą i z pełnym zestawem węzłów z powrotem w rotacji.*

**Skróty użyte na tej stronie (5)**

| Skrót | Co znaczy |
|---|---|
| **API** | Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą |
| **CPU** | Central Processing Unit: procesor |
| **etcd** | etc distributed: rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały stan klastra, a zmiany uzgadnia protokołem Raft |
| **GitOps** | Git Operations: sposób prowadzenia klastra, w którym pożądany stan jest zapisany w repozytorium Git, a agent w klastrze sam doprowadza rzeczywistość do tego stanu |
| **REST** | sposób budowania API przez zwykłe żądania HTTP (GET, POST) zamiast osobnego protokołu |

# Zanim zaczniemy

**Z ŻYCIA WZIĘTE**

O 9:00, w zaplanowanym oknie serwisowym, Tomek zaczyna aktualizację klastra produkcyjnego od razu od `kubeadm upgrade apply`, pomijając wcześniejszą kopię zapasową etcd, bo „przecież nic się nie stanie na jednej wersji pomniejszej". W trakcie aktualizacji jeden z kontrolerów zawiesza się w połowie zapisu i klaster zaczyna zgłaszać niespójne obiekty w kilku przestrzeniach nazw naraz. Po tym incydencie Tomek wraca do procedury i zaczyna trzymać się jej dosłownie, zaczynając każdą kolejną aktualizację od świeżej migawki etcd, zanim dotknie czegokolwiek innego.

Procedura opisuje klaster zbudowany przez `kubeadm`, czyli ogólny sposób budowy klastra. Laboratorium Zielonej to k3s (węzły `k3s01`, `k3s02`, `k3s03`), które aktualizujemy własnym skryptem instalacyjnym, więc nazwy węzłów w wynikach są z laboratorium, a komendy `kubeadm` dotyczą klastrów postawionych tym narzędziem.

| Wymaganie | Jak sprawdzamy |
| --- | --- |
| aktualna wersja płaszczyzny sterowania i węzłów | `kubectl version`, `kubectl get nodes` |
| świeża kopia zapasowa etcd sprzed zmiany | procedura niżej, sekcja „Kopia zapasowa etcd" |
| różnica wersji między płaszczyzną sterowania a kubeletem w dopuszczalnym zakresie | tabela niżej |
| wystarczająca pojemność pozostałych węzłów, żeby przyjąć obciążenie z węzła wyłączanego z rotacji | `kubectl top nodes`, `kubectl describe nodes` |
| brak toczącego się wdrożenia aplikacji w trakcie okna serwisowego | `kubectl get deployments -A -o wide`, kontakt z zespołami aplikacyjnymi |

**Uważaj:** aktualizacja płaszczyzny sterowania przed węzłami jest jedynym wspieranym kierunkiem: kubelet może być o jedną do trzech wersji pomniejszych starszy niż `kube-apiserver`, ale nigdy nowszy. Aktualizacja w złej kolejności potrafi zostawić klaster w stanie, z którego oficjalna ścieżka powrotu nie istnieje.

| Komponent | Dopuszczalna różnica względem kube-apiserver |
| --- | --- |
| kube-controller-manager, kube-scheduler | tyle samo albo o jedną wersję pomniejszą starsze |
| kubelet, kube-proxy | tyle samo albo do trzech wersji pomniejszych starsze |
| kubectl | jedna wersja pomniejsza w dowolną stronę |

# Przebieg

## 1. Kopia zapasowa etcd

```
ETCDCTL_API=3 etcdctl snapshot save /var/backups/etcd/snapshot-$(date +%Y%m%d-%H%M).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
```

```
Snapshot saved at /var/backups/etcd/snapshot-20260925-0912.db
```

Wykonujemy kopię zapasową całej bazy etcd, czyli jedynego miejsca, w którym Kubernetes trzyma prawdziwy stan całego klastra: każdy obiekt, każdy Secret, każdą definicję. Bez tej kopii nie ma odtworzenia klastra po nieudanej aktualizacji, przypadkowym usunięciu przestrzeni nazw czy awarii samego etcd, a reszta procedury zakłada, że ten krok się powiódł, zanim przejdziemy dalej.

# 2. Kopiowanie migawki poza węzeł płaszczyzny sterowania

```
export RESTIC_REST_USERNAME=orkiestracja
read -r -s -p "Hasło konta REST: " RESTIC_REST_PASSWORD; echo; export RESTIC_REST_PASSWORD
restic -r rest:https://bkp01.ad.zielona.example:8000/orkiestracja --password-file
/etc/restic/haslo-orkiestracja backup --tag kubeadm /var/backups/etcd/snapshot-20260925-
0912.db
```

Przenosimy migawkę poza węzeł, na którym powstała, bo kopia leżąca na tym samym dysku, który właśnie aktualizujemy, nie chroni przed awarią tego dysku. Komendy wykonujemy jako root i można je wykonać jako blok. `bkp01` przyjmuje kopie tylko przez rest-server w trybie tylko do dopisywania, z kontem REST nazwanym tak jak repozytorium, według strony Serwery kopii bkp01 i bkp02: restic, kopia krzyżowa i niezmienność, więc migawka trafia do repozytorium `orkiestracja` (poziom 90 dni) kontem `orkiestracja`, a nie przez `scp`. Skrypt z harmonogramem migawek co 6 godzin opisuje strona Kubernetes z kubeadm w wysokiej dostępności: trzy węzły płaszczyzny sterowania, etcd, certyfikaty i aktualizacja.

# 3. Aktualizacja płaszczyzny sterowania: pakiet kubeadm

```
sudo apt-mark unhold kubeadm
sudo apt-get update && sudo apt-get install -y kubeadm=1.31.2-1.1
sudo apt-mark hold kubeadm
```

Aktualizujemy narzędzie `kubeadm` jako pierwsze, przed samym klastrem: to ono orkiestruje resztę procesu aktualizacji węzła płaszczyzny sterowania.

# 4. Sprawdzenie planu aktualizacji

```
sudo kubeadm upgrade plan
```

```
Components that must be upgraded manually after you have upgraded the control plane with
'kubeadm upgrade apply':
COMPONENT   CURRENT       TARGET
```

```
kubelet     3 x v1.30.4   v1.31.2

Upgrade to the latest stable version:
COMPONENT                 CURRENT   TARGET
kube-apiserver             v1.30.4   v1.31.2
kube-controller-manager    v1.30.4   v1.31.2
kube-scheduler             v1.30.4   v1.31.2
kube-proxy                 v1.30.4   v1.31.2

You can now apply the upgrade by executing the following command:

kubeadm upgrade apply v1.31.2
```

Sprawdzamy, co dokładnie `kubeadm` zamierza zmienić, zanim cokolwiek się wykona: plan pokazuje też wprost, że kubelet trzeba będzie zaktualizować osobno, ręcznie, na każdym węźle.

# 5. Zastosowanie aktualizacji na pierwszym węźle płaszczyzny sterowania

```
sudo kubeadm upgrade apply v1.31.2
```

```
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.31.2". Enjoy!
```

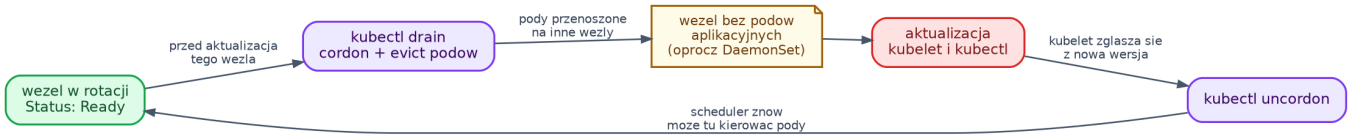
Wykonujemy właściwą aktualizację komponentów płaszczyzny sterowania na pierwszym z jej węzłów: w klastrze z kilkoma węzłami płaszczyzny sterowania pozostałe aktualizujemy komendą `kubeadm upgrade node`, jeden po drugim, nigdy równolegle.

# 6. Odizolowanie węzła przed aktualizacją kubeleta: kubectl drain

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

```
node/NazwaWezla cordoned
evicting pod produkcja/sklep-api-7f9d4
node/NazwaWezla drained
```

Przenosimy wszystkie pody z węzła na pozostałe węzły klastra, zanim zaczniemy aktualizować sam kubelet: `--ignore-daemonsets` pomija pody DaemonSetów, które i tak muszą działać na każdym węźle i nie da się ich stamtąd usunąć, a `--delete-emptydir-data` zgadza się na utratę danych w wolumenach `emptyDir`, opisanych na stronie Wolumeny, PV, PVC i StorageClass, bo z definicji nie przeżywają one przeniesienia poda.




*Węzeł przechodzi przez drain, aktualizację i uncordon, wracając do rotacji dopiero gdy scheduler znowu może na niego kierować pody (kliknięcie otwiera pełny rozmiar)*

# 7. Aktualizacja kubeleta i kubectl na odizolowanym węźle

```
sudo apt-mark unhold kubelet kubectl
sudo apt-get update && sudo apt-get install -y kubelet=1.31.2-1.1 kubectl=1.31.2-1.1
sudo apt-mark hold kubelet kubectl
sudo systemctl daemon-reload
sudo systemctl restart kubelet
```

Aktualizujemy same pakiety i restartujemy usługę kubeleta: węzeł zgłasza się z powrotem do API z nową wersją.

# 8. Przywrócenie węzła do rotacji: kubectl uncordon

```
kubectl uncordon NazwaWezla
```

```
node/NazwaWezla uncordoned
```

Zdejmujemy oznaczenie `cordoned`: scheduler znowu może kierować nowe pody na ten węzeł. Kroki od 6 do 8 powtarzamy dla każdego kolejnego węzła roboczego, jeden po drugim, nigdy dla kilku naraz, żeby w każdej chwili klaster miał wystarczającą pojemność na przyjęcie przenoszonego

obciążenia.

# Weryfikacja

```
kubectl get nodes
```

```
NAME    STATUS   ROLES           AGE    VERSION
k3s01   Ready    control-plane   180d   v1.31.2
k3s02   Ready    <none>          180d   v1.31.2
k3s03   Ready    <none>          180d   v1.31.2
```

Sprawdzamy, czy wszystkie węzły zgłaszają nową wersję i status `Ready`: różne wersje na różnych węzłach w trakcie samej procedury są normalne, po jej zakończeniu wszystkie powinny się zgadzać.

```
kubectl get pods -A --field-selector=status.phase!=Running,status.phase!=Succeeded
```

Sprawdzamy, czy po całej turze aktualizacji nie zostały żadne pody utknięte poza stanem `Running` albo `Succeeded`: pusta lista jest tu wynikiem poprawnym.

# Wycofanie zmian

Kubernetes nie ma jednej komendy cofającej aktualizację wersji: `kubeadm` aktualizuje w jedną stronę. Cofnięcie oznacza w praktyce dwie ścieżki, zależnie od tego, jak daleko zaszła awaria:

- jeśli tylko część węzłów roboczych zdążyła się zaktualizować, a płaszczyzna sterowania działa poprawnie, po prostu instalujemy z powrotem poprzednią wersję pakietów `kubelet` i `kubectl` na dotkniętych węzłach, tą samą procedurą co kroki od 6 do 8, tylko ze starszym numerem wersji
- jeśli awaria dotyczy samej płaszczyzny sterowania i nie da się jej ustabilizować, jedynym pewnym sposobem powrotu jest odtworzenie etcd z migawki wykonanej w kroku 1, zgodnie z dokumentacją odtwarzania klastra z kopii zapasowej

# Gdy coś nie zadziała

| Objaw | Przyczyna | Naprawa |
|---|---|---|

| | | |
|---|---|---|
| `kubeadm upgrade apply` kończy się błędem walidacji wersji | zbyt duża różnica wersji między obecną a docelową (przeskok więcej niż jednej wersji pomniejszej) | aktualizacja krok po kroku, jedna wersja pomniejsza na raz |
| węzeł zostaje `NotReady` po restarcie kubeleta | niezgodna wersja kubeleta z `kube-apiserver`, uszkodzona konfiguracja po aktualizacji pakietu | `journalctl -u kubelet -n 100`, porównanie wersji z tabelą zgodności wyżej |
| `kubectl drain` wisi bez końca | `PodDisruptionBudget` nie pozwala usunąć ostatniej repliki, pod bez kontrolera nadrzędnego | `kubectl get pdb -A`, dodanie `--force` tylko dla podów bez kontrolera, świadomie |
| pody utykają w `Pending` po drenażu jednego węzła | za mało pojemności na pozostałych węzłach na przyjęcie przeniesionego obciążenia | wstrzymanie aktualizacji kolejnych węzłów do czasu zwolnienia zasobów |

# Kopia zapasowa etcd: dlaczego jest warunkiem, nie opcją

Etcd jest jedynym źródłem prawdy o stanie całego klastra: każdy Deployment, każdy Secret, każdy ConfigMap, każdy PersistentVolumeClaim istnieje naprawdę tylko jako wpis w etcd, a wszystko inne (pody, kontenery, pliki na dyskach węzłów) da się odtworzyć na nowo z tego jednego źródła. Utrata etcd bez aktualnej kopii zapasowej oznacza utratę całej konfiguracji klastra bezpowrotnie: same dane aplikacji na wolumenach `PersistentVolume` mogą przetrwać, ale nikt nie będzie wiedział, jak je z powrotem podłączyć, bez definicji obiektów, które to opisywały.

Kopię zapasową wykonujemy przed każdą aktualizacją klastra i dodatkowo w regularnym cyklu niezależnym od aktualizacji: częstotliwość dobieramy do tego, ile zmian w konfiguracji klastra jesteśmy w stanie odtworzyć ręcznie z repozytorium GitOps, jeśli takie prowadzimy, a ile istnieje wyłącznie jako stan w klastrze.

# Co realnie monitorujemy w klastrze

- dostępność `kube-apiserver` i czas odpowiedzi na zapytania: bez działającego API cała reszta obserwowalności też przestaje działać
- liczbę węzłów w stanie innym niż `Ready`
- zużycie CPU i pamięci względem `requests` i `limits` na poziomie węzłów i najważniejszych aplikacji

- zajętość dysku na każdym węźle, zwłaszcza katalogu danych kubeleta i środowiska uruchomieniowego kontenerów
- liczbę podów w stanach `Pending`, `CrashLoopBackOff` i `Evicted` utrzymujących się dłużej niż kilka minut
- wiek i rozmiar ostatniej kopii zapasowej etcd

# Zajętość węzłów: kubectl top

```
kubectl top nodes
```

```
NAME    CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
k3s01   420m         21%    3120Mi          39%
k3s02   1850m        92%    7200Mi          88%
k3s03   610m         30%    4100Mi          51%
```

Sprawdzamy bieżące zużycie zasobów na poziomie węzłów: wymaga zainstalowanego `metrics-server` w klastrze, bez którego ta komenda kończy się błędem braku dostępnych metryk. Węzeł z zużyciem bliskim 100% jest pierwszym kandydatem do przeniesienia części obciążenia, zanim dojdzie do eksmisji podów opisanej na stronie Diagnostyka: objaw, przyczyna, pierwsze kroki.

# Czyszczenie nieużywanych obrazów na węzłach

```
sudo crictl rmi --prune
```

```
Deleted: sha256:8f3e02c1b4d9...
Deleted: sha256:1a9c7e0f22ab...
```

Usuwamy z węzła obrazy kontenerów, do których żaden aktualnie istniejący kontener już się nie odwołuje: kubelet robi to też sam automatycznie w ramach mechanizmu odśmiecania (garbage collection) po przekroczeniu progów zajętości dysku, ale ręczne czyszczenie bywa potrzebne od razu, w chwili aktywnego problemu z miejscem na dysku, opisanego na stronie Diagnostyka: objaw, przyczyna, pierwsze kroki.

**Powiązane:** Diagnostyka: objaw, przyczyna, pierwsze kroki · Wolumeny, PV, PVC i StorageClass

**Źródła:** dokumentacja kubernetes.io, rozdziały „Upgrading kubeadm clusters", „Safely Drain a Node", „Operating etcd clusters for Kubernetes" i „Resource Metrics Pipeline"

---

Revision #34
Created 2026-09-25 10:33:12 UTC by Tomasz Konieczny
Updated 2026-10-26 14:17:15 UTC by Tomasz Konieczny