Autoskalowanie: HPA i requests, limits na kolorowym nginx

◫◫ Poziom: średnie

◫◫ Dotyczy: Kubernetes 1.36 (k3s 1.36 z metrics-server), HorizontalPodAutoscaler autoscaling/v2, wykres kolorowy-nginx 0.2.0

▯ Czas: ~50 min

Włączamy w wykresie kolorowego nginksa HorizontalPodAutoscaler i patrzymy, jak pod obciążeniem przybywa podów, a razem z nimi kolorów na stronie. Do tego pokazujemy, skąd autoskaler wie, że pod jest obciążony, i co się dzieje, gdy kontener przekroczy swoje limity.

◫◫ Skróty użyte na tej stronie (5)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
CD	Continuous Delivery/Deployment: automatyczne wdrażanie zbudowanego kodu
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
HPA	HorizontalPodAutoscaler: obiekt Kubernetesa, który sam zwiększa albo zmniejsza liczbę podów Deploymentu, gdy zużycie procesora lub pamięci odbiega od celu
QoS	Quality of Service: klasa ważności, np. który pod zostanie zabity pierwszy przy braku pamięci

Fiszka

Komendy wykonujemy z CLI bash na stacji administratora albo na `k3s01`. Zakładamy wydanie `kolorowy-nginx` z poprzedniej strony, Helm: kolorowy nginx jako pakiet, w przestrzeni nazw `kolory`, z wykresem zaktualizowanym do wersji 0.2.0 z sekcji o szablonie HPA.

◫◫ TYLKO ODCZYT

Sprawdzenie metrics-server w k3s: kubectl get apiservice

```
kubectl get apiservice v1beta1.metrics.k8s.io
```

Przykład:

```
NAME                     SERVICE                      AVAILABLE   AGE
v1beta1.metrics.k8s.io   kube-system/metrics-server   True        12d
```

Wartość `True` w kolumnie `AVAILABLE` znaczy, że API metryk odpowiada. k3s instaluje metrics-server razem z całą resztą, więc nie dodajemy go ręcznie.

API = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Podejrzenie zużycia węzła: kubectl top nodes

```
kubectl top nodes
```

Przykład:

```
NAME    CPU(cores)   CPU(%)   MEMORY(bytes)   MEMORY(%)
k3s01   212m         5%       1894Mi          24%
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Podejrzenie zużycia podów razem z kontenerami: kubectl top pods --containers

```
kubectl top pods -n kolory --containers
```

Przykład:

```
POD                              NAME    CPU(cores)   MEMORY(bytes)
kolorowy-nginx-6c8f9d7b5b-2xk4t  nginx   1m           6Mi
kolorowy-nginx-6c8f9d7b5b-7mqzv  nginx   1m           6Mi
kolorowy-nginx-6c8f9d7b5b-p9r2s  nginx   1m           6Mi
```

Metryki są migawką z ostatnich kilkudziesięciu sekund, więc tuż po starcie poda polecenie może jeszcze odpowiadać błędem `metrics not available yet`.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Sprawdzenie klasy QoS podów: kubectl get pod -o custom-columns

```
kubectl get pods -n kolory -o custom-columns=NAZWA:.metadata.name,QOS:.status.qosClass
```

Przykład:

```
NAZWA                           QOS
kolorowy-nginx-6c8f9d7b5b-2xk4t   Burstable
kolorowy-nginx-6c8f9d7b5b-7mqzv   Burstable
kolorowy-nginx-6c8f9d7b5b-p9r2s   Burstable
```

QoS = Quality of Service: klasa ważności, np. który pod zostanie zabity pierwszy przy braku pamięci · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Sprawdzenie, ile zasobów węzła jest już zarezerwowane: kubectl describe node

```
kubectl describe node k3s01 | grep -A 8 "Allocated resources"
```

Przykład:

```
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests     Limits
  --------           --------     ------
  cpu                450m (11%)   1210m (30%)
  memory             332Mi (4%)   786Mi (10%)
  ephemeral-storage  0 (0%)       0 (0%)
  hugepages-2Mi      0 (0%)       0 (0%)
```

Scheduler dopasowuje pody do węzła po sumie requests, a nie po faktycznym zużyciu. Ta suma ogranicza, ile podów autoskaler może jeszcze dołożyć.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Włączenie autoskalowania w wykresie: helm upgrade --set autoscaling.enabled

```
helm upgrade kolorowy-nginx ./kolorowy-nginx --namespace kolory --set
autoscaling.enabled=true --wait
```

Przykład:

```
Release "kolorowy-nginx" has been upgraded. Happy Helming!
NAME: kolorowy-nginx
STATUS: deployed
REVISION: 2
```

Od tej chwili szablon Deploymentu pomija pole `replicas`, a liczbą podów zarządza HorizontalPodAutoscaler z wartościami z `values.yaml`: od 2 do 8 podów, cel 50 procent żądanego procesora.

helm = menedżer paczek aplikacji dla Kubernetesa

TYLKO ODCZYT

Podgląd autoskalera: kubectl get hpa

```
kubectl get hpa -n kolory
```

Przykład:

```
NAME             REFERENCE                   TARGETS       MINPODS   MAXPODS
REPLICAS   AGE
kolorowy-nginx   Deployment/kolorowy-nginx   cpu: 2%/50%   2         8
3          4m
```

Kolumna `TARGETS` pokazuje bieżące zużycie jako procent żądania, a po ukośniku cel. Zapis `<unknown>` oznacza, że autoskaler nie umie jeszcze policzyć zużycia, i opisujemy to w

częstych błędach.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Opis autoskalera razem z jego decyzjami: kubectl describe hpa

```
kubectl describe hpa kolorowy-nginx -n kolory
```

Przykład:

```
Name:                                                  kolorowy-nginx
Namespace:                                             kolory
Reference:                                             Deployment/kolorowy-nginx
Metrics:                                               ( current / target )
  resource cpu on pods  (as a percentage of request):  2% (1m) / 50%
Min replicas:                                          2
Max replicas:                                          8
Deployment pods:                                       3 current / 3 desired
Conditions:
  Type            Status  Reason              Message
  ----            ------  ------              -------
  AbleToScale     True    ReadyForNewScale    recommended size matches current size
  ScalingActive   True    ValidMetricFound    the HPA was able to successfully calculate
a replica count from cpu resource utilization (percentage of request)
  ScalingLimited  False   DesiredWithinRange  the desired count is within the acceptable
range
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Uruchomienie generatora obciążenia: kubectl create deployment

```
kubectl create deployment generator-obciazenia -n kolory --image=busybox:1.37.0 -- sh -c
'while true; do wget -q -O- http://kolorowy-nginx > /dev/null; done'
kubectl scale deployment/generator-obciazenia -n kolory --replicas=3
```

Przykład:

```
deployment.apps/generator-obciazenia created
deployment.apps/generator-obciazenia scaled
```

Trzy pody w nieskończonej pętli odpytują Service `kolorowy-nginx` po nazwie, wewnątrz klastra. Dla obciążenia z zewnątrz możemy użyć programu `hey`, który omawiamy w sekcji o testach.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Obserwacja autoskalowania na żywo: kubectl get hpa -w

```
kubectl get hpa kolorowy-nginx -n kolory -w
```

Przykład:

```
NAME             REFERENCE                   TARGETS         MINPODS   MAXPODS
REPLICAS   AGE
kolorowy-nginx   Deployment/kolorowy-nginx   cpu: 2%/50%     2         8
3          9m
kolorowy-nginx   Deployment/kolorowy-nginx   cpu: 118%/50%   2         8
3          9m30s
kolorowy-nginx   Deployment/kolorowy-nginx   cpu: 118%/50%   2         8
7          9m45s
kolorowy-nginx   Deployment/kolorowy-nginx   cpu: 71%/50%    2         8
8          10m
```

Autoskaler liczy: 3 pody razy 118 podzielone przez 50 daje 7,08, czyli 8 po zaokrągleniu w górę. Zmiana z 3 na 7 w pierwszym kroku wynika z ograniczenia tempa zwiększania: domyślnie najwyżej dwukrotnie albo o cztery pody na 15 sekund, zależnie od tego, co daje więcej.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Zliczenie kolorów w odpowiedziach: curl w pętli

```
for i in $(seq 1 60); do
  curl -s --resolve kolory.zielona.example:80:10.20.10.44 http://kolory.zielona.example/
| grep -o "Kolor: [a-z]*"
done | sort | uniq -c | sort -rn
```

Przykład:

```
     14 Kolor: fioletowy
     12 Kolor: zielony
     11 Kolor: niebieski
      9 Kolor: czerwony
      8 Kolor: zolty
      6 Kolor: pomaranczowy
```

Sześć różnych kolorów przy ośmiu podach oznacza, że pewne kolory się powtarzają. Pod trzema podami zwykle widzimy mniej niż sześć kolorów.

curl = client URL: pobieranie i wysyłanie przez HTTP z terminala

OSTROŻNIE · NIEODWRACALNE

Zatrzymanie generatora obciążenia: kubectl delete deployment

```
kubectl delete deployment generator-obciazenia -n kolory
```

Po zatrzymaniu generatora autoskaler czeka 60 sekund okna stabilizacji z wykresu i zmniejsza liczbę podów, aż wróci do dwóch.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Odczyt licznika dławienia procesora: cat cpu.stat

```
kubectl exec -n kolory deploy/kolorowy-nginx -c nginx -- cat /sys/fs/cgroup/cpu.stat
```

Przykład:

```
usage_usec 18423311
user_usec 9211002
```

```
system_usec 9212309
nr_periods 4521
nr_throttled 3187
throttled_usec 95338120
nr_bursts 0
burst_usec 0
```

Pole `nr_throttled` to liczba okresów, w których kontener dostał mniej procesora, niż chciał, bo uderzył w limit. Rosnąca wartość przy niskim zużyciu średnim oznacza zbyt ciasny `limits.cpu`.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Zawężenie limitu procesora na próbę: helm upgrade --set resources.limits.cpu

```
helm upgrade kolorowy-nginx ./kolorowy-nginx --namespace kolory --reuse-values --set
resources.limits.cpu=60m --wait
```

Po tej zmianie kolejne pody mają limit 60m procesora. Przy generatorze obciążenia odczyt `cpu.stat` z poprzedniej fiszki pokaże rosnące `nr_throttled`.

helm = menedżer paczek aplikacji dla Kubernetesa

ZMIENIA SYSTEM

Uruchomienie poda, który przekracza limit pamięci: kubectl apply

```
kubectl apply -f oom-test.yaml
kubectl get pod oom-test -n kolory
```

Przykład:

```
pod/oom-test created
NAME       READY   STATUS      RESTARTS   AGE
oom-test   0/1     OOMKilled   0          8s
```

Plik `oom-test.yaml` pokazujemy w sekcji o limitach pamięci. Pod ma limit 20Mi i próbuje zająć 50Mi, więc jądro zabija jego proces.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Odczyt przyczyny zabicia kontenera: kubectl describe pod

```
kubectl describe pod oom-test -n kolory | grep -A 5 "State:"
```

Przykład:

```
    State:          Terminated
      Reason:       OOMKilled
      Exit Code:    137
      Started:      Mon, 05 Oct 2026 10:41:12 +0200
      Finished:     Mon, 05 Oct 2026 10:41:13 +0200
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

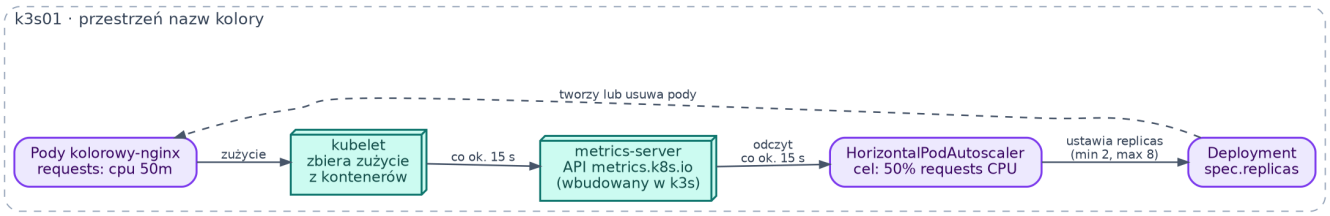
Z ŻYCIA WZIĘTE

Poniedziałek, 9:40. Ola ogłasza w firmie testy nowej strony na `kolory.zielona.example`, a kilkadziesiąt osób otwiera ją naraz. Tomek, który wiedział o tym dzień wcześniej, włączył w wykresie autoskalowanie, więc w `kubectl get hpa -w` ogląda, jak pody rosną z trzech do ośmiu, a na stronie przybywa kolorów. Wcześniej, bez HPA, musiałby wpisać `kubectl scale` ręcznie i trafić z liczbą na oko.

Jak to działa

Rozwiń: jak to działa pod spodem

Autoskaler nie patrzy na ruch ani na liczbę zapytań, tylko na zużycie procesora albo pamięci zmierzone w podach. Ktoś musi to zużycie zebrać, ktoś je porównać z celem, a ktoś zmienić liczbę podów. Te trzy role odgrywają kubelet, metrics-server i kontroler HPA.



Kubelet zbiera zużycie z kontenerów, metrics-server udostępnia je w API metrics.k8s.io, HPA porównuje je z celem i ustawia replicas w Deploymencie, a Deployment tworzy lub usuwa pody (kliknięcie otwiera pełny rozmiar)

> **Pojęcie: metrics-server.** Mały komponent klastra, który co kilkanaście sekund pyta kubelety o bieżące zużycie procesora i pamięci i udostępnia je w API `metrics.k8s.io`. Pamięta tylko ostatnią migawkę, bez historii, więc nie zastępuje Prometheusa z następnej strony. Z jego danych korzystają `kubectl top` i HPA.

> **Pojęcie: HorizontalPodAutoscaler.** Obiekt, który co 15 sekund porównuje średnie zużycie podów Deploymentu z celem i zmienia pole `spec.replicas` tak, żeby średnia zbliżyła się do celu. Słowo „poziomy” oznacza dokładanie kolejnych podów, w odróżnieniu od powiększania jednego poda. Podstawy opisaliśmy na stronie Deployment, ReplicaSet i aktualizacje bez przerwy, tu patrzymy na jego liczby.

Wzór, którym HPA liczy liczbę podów

Autoskaler stosuje jeden wzór: pożądana liczba podów to obecna liczba podów razy iloraz bieżącej wartości metryki i wartości docelowej, zaokrąglony w górę. Przy trzech podach z zużyciem 118 procent żądania i celem 50 procent wychodzi 3 razy 2,36, czyli 7,08, a po zaokrągleniu 8 podów.

Procent liczymy względem `requests`, nie względem `limits`. Pod z `requests.cpu` równym 50m i zużyciem 59m ma więc 118 procent, nawet jeśli limit wynosi 200m. To dlatego pierwszą rzeczą, którą sprawdzamy przy `<unknown>` w kolumnie `TARGETS`, jest brak `requests`.

Mechanizm	Wartość domyślna	Znaczenie
Okres obliczeń	15 sekund	jak często HPA odczytuje metryki i decyduje

Mechanizm	Wartość domyślna	Znaczenie
Tolerancja	10 procent	w zakresie plus minus 10 procent od celu HPA nic nie zmienia, żeby nie huśtać liczby podów
Zwiększanie	bez okna stabilizacji	najwyżej dwukrotnie albo o 4 pody na 15 sekund, zależnie od tego, co daje więcej
Zmniejszanie	okno stabilizacji 300 sekund	HPA bierze najwyższą wartość z ostatnich pięciu minut, dlatego pody znikają powoli

W naszym wykresie skracamy okno zmniejszania do 60 sekund przez `scaleDownStabilizationSeconds`, żeby ćwiczenie nie trwało pięć minut. Na produkcji zostawiamy domyślne 300 sekund, żeby krótki spadek ruchu nie zdejmował podów tuż przed kolejnym szczytem.

Obiekt HPA i jego szablon w wykresie

Czysty manifest HPA, jaki Helm generuje z naszego szablonu, wygląda tak:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: kolorowy-nginx
  namespace: kolory
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: kolorowy-nginx
  minReplicas: 2
  maxReplicas: 8
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

```
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 60
```

Pole `scaleTargetRef` wskazuje, czym HPA steruje, a `metrics` mówi, co mierzy i do jakiej wartości dąży. Typ `Utilization` oznacza procent żądania, a typ `AverageValue` pozwala podać średnią wartość bezwzględną, na przykład `100m`. Sekcja `behavior` jest opcjonalna i służy do strojenia tempa.

Do wykresu z poprzedniej strony dopisujemy trzy rzeczy. Pierwsza to blok `autoscaling` w `values.yaml`, druga to nowy plik szablonu, a trzecia to warunek w Deploymencie, który pomija `replicas`, gdy autoskalowanie jest włączone. Wersję wykresu podnosimy do 0.2.0.

```
# Dopisek do values.yaml (przed nameOverride)
autoscaling:
  enabled: false
  minReplicas: 2
  maxReplicas: 8
  targetCPUUtilizationPercentage: 50
  # Jak długo czekamy z zmniejszaniem liczby podów po spadku obciążenia
  scaleDownStabilizationSeconds: 60
```

```
{{- if .Values.autoscaling.enabled }}
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: {{ include "kolorowy-nginx.fullname" . }}
  labels:
    {{- include "kolorowy-nginx.labels" . | nindent 4 }}
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: {{ include "kolorowy-nginx.fullname" . }}
  minReplicas: {{ .Values.autoscaling.minReplicas }}
  maxReplicas: {{ .Values.autoscaling.maxReplicas }}
  metrics:
    - type: Resource
      resource:
```

```
        name: cpu
        target:
          type: Utilization
          averageUtilization: {{ .Values.autoscaling.targetCPUUtilizationPercentage }}
  behavior:
    scaleDown:
      stabilizationWindowSeconds: {{ .Values.autoscaling.scaleDownStabilizationSeconds }}
{{- end }}
```

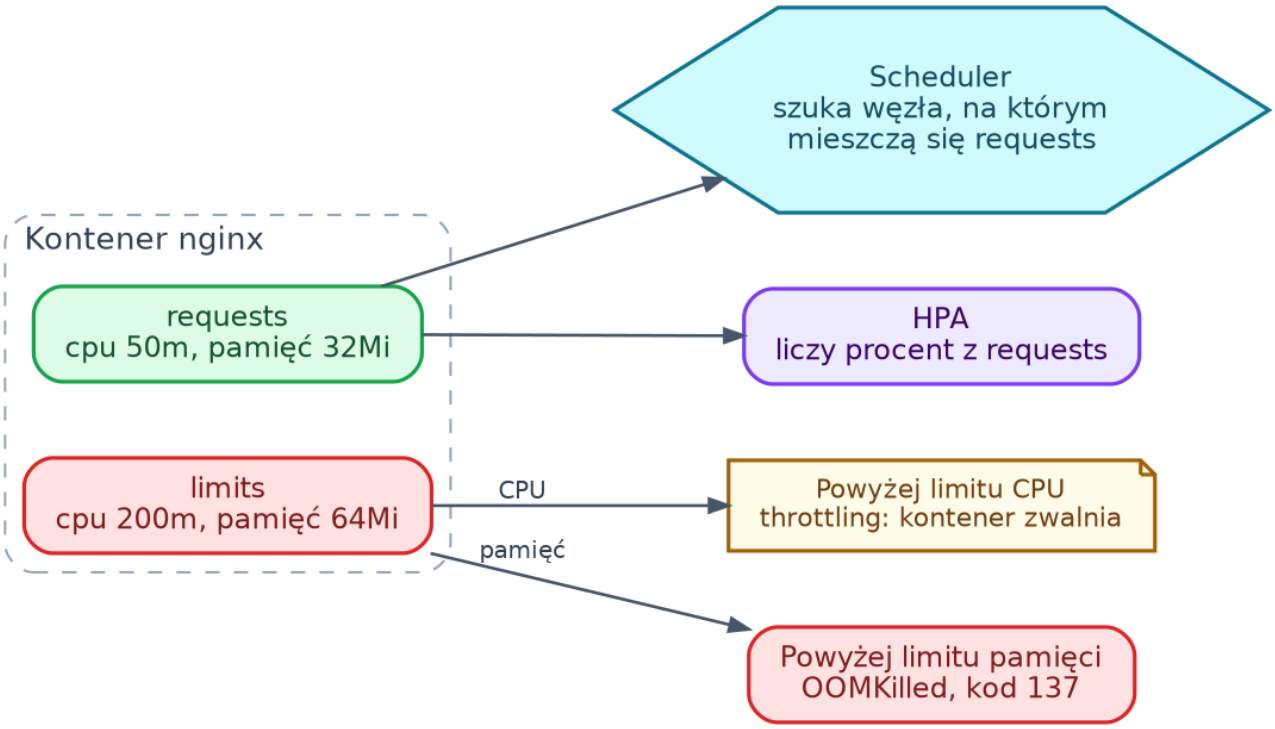
W szablonie Deploymentu zamieniamy jedną linię na warunek:

```
  {{- if not .Values.autoscaling.enabled }}
  replicas: {{ .Values.replicaCount }}
  {{- end }}
```

> ⚠ Gdy HPA działa, a Deployment dalej ma w manifeście stałe `replicas: 3`, każdy `helm upgrade` albo synchronizacja z Gita przywraca trzy pody i kasuje decyzję autoskalera. Dlatego pole `replicas` w szablonie pomijamy, gdy `autoscaling.enabled` ma wartość `true`. W Argo CD ten sam problem opisujemy na stronie GitOps z Argo CD: klaster sam pobiera zmiany z Gitea.

Requests, limits i klasy QoS na kolorowym nginksie

Znaczenie `requests` i `limits` oraz trzy klasy QoS opisuje strona Pod: najmniejsza jednostka. Tu sprawdzamy, co z nich wynika dla naszego wykresu. Kontener nginx ma `requests` 50m procesora i 32Mi pamięci oraz `limits` 200m i 64Mi, więc pod trafia do klasy Burstable.



Requests trafiają do schedulera i do HPA, limit procesora kończy się dławieniem, a limit pamięci zabiciem kontenera z kodem 137 (kliknięcie otwiera pełny rozmiar)

Do klasy Guaranteed pod trafia dopiero wtedy, gdy każdy jego kontener ma `requests` równe `limits`, zarówno dla procesora, jak i pamięci. Dotyczy to także kontenerów inicjalizujących, dlatego w `values.yaml` nadaliśmy kontenerowi `generator-koloru` własne, równe `requests` i `limits`. Bez tego pojedynczy kontener bez zasobów zepsułby klasę całego poda.

Klasę Guaranteed uzyskujemy przez podwyższenie `requests` do wartości limitów:

```
helm upgrade kolorowy-nginx ./kolorowy-nginx --namespace kolory --reuse-values \
  --set resources.requests.cpu=200m --set resources.requests.memory=64Mi --wait
```

Po tej zmianie `kubectl get pods -o custom-columns=...` pokazuje `Guaranteed`. Zmienia się też cel HPA: 50 procent z 200m to 100m, więc pody są dokładane przy czterokrotnie większym zużyciu bezwzględnym niż wcześniej. Zmiana `requests` zmienia więc zachowanie autoskalera, nawet jeśli jego własne ustawienia zostają bez zmian.

> Węzeł ma skończoną pojemność. Osiem podów po 50m i 32Mi to zaledwie 400m i 256Mi, ale na `k3s01` stoi też Traefik, CoreDNS i reszta laboratorium. Gdy suma `requests` przekroczy wolne zasoby, nowe pody zostają w stanie

> `Pending`, a HPA nie ma jak dołożyć węzła, bo klaster ma jeden węzeł.

Dławienie procesora

> **Pojęcie: dławienie (throttling).** Mechanizm jądra Linuksa, który po wykorzystaniu przez kontener limitu procesora w danym okresie wstrzymuje jego procesy do końca tego okresu. Okres trwa domyślnie 100 milisekund. Kontener o limicie 60m dostaje więc 6 milisekund czasu procesora na każde 100 milisekund i zostaje zatrzymany na resztę.

Dławienie nie zabija procesu, tylko go spowalnia, więc łatwo je przeoczyć. Strona odpowiada wolniej, sonda gotowości może się spóźnić, a `kubectl top` pokazuje zużycie równe limitowi, czyli „zupełnie normalne” 60m. Prawdziwy objaw to rosnące `nr_throttled` w pliku `cpu.stat`, które odczytujemy fiszką wyżej.

Częsta reakcja na dławienie to podniesienie albo usunięcie `limits.cpu`. Usunięcie jest zwykle bezpieczne, bo procesor, w przeciwieństwie do pamięci, da się dzielić między kontenery po cichu. Zostawiamy za to `requests.cpu`, bo bez niego scheduler i HPA nie wiedzą, ile procesora aplikacja potrzebuje.

Przekroczenie limitu pamięci: OOMKilled

Pamięci nie da się spowolnić ani odzyskać, więc przekroczenie `limits.memory` kończy się natychmiastowym zabiciem procesu przez jądro. Kubelet oznacza kontener jako `OOMKilled` z kodem wyjścia 137 i, zależnie od `restartPolicy`, uruchamia go od nowa. Diagnostykę opisuje strona Diagnostyka: objaw, przyczyna, pierwsze kroki, tu pokazujemy sam mechanizm.

Pod doświadczalny zajmuje 50 megabajtów pamięci przy limicie 20Mi. Zapisujemy go jako `oom-test.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: oom-test
  namespace: kolory
spec:
  restartPolicy: Never
  containers:
```

```
    - name: pamieciozerca
      image: busybox:1.37.0
      command:
        - sh
        - -c
        - |
          x=$(head -c 52428800 /dev/zero | tr '\000' a)
          echo ${#x}
      resources:
        requests:
          cpu: 10m
          memory: 16Mi
        limits:
          cpu: 100m
          memory: 20Mi
```

Powłoka trzyma w zmiennej `x` całe 50 megabajtów, więc przekracza limit 20Mi i zostaje zabita. Po obejrzeniu wyniku usuwamy pod poleceniem `kubectl delete pod oom-test -n kolory`.

Dla nginksa limit 64Mi jest zapasem, a zużycie wynosi kilka megabajtów. Zwykle OOMKilled dotyka aplikacji z pamięcią zależną od danych, a nie statycznego serwera. Ten sam mechanizm obejmuje jednak wszystkie kontenery w podzie, także sidecar z następnej strony, któremu damy własne, niewielkie limity.

Test obciążenia z zewnątrz: program hey

Pętla `wget` w podach pokazuje autoskalowanie z wnętrza klastra. Test z zewnątrz, przez Traefika, robimy ze stacji administratora programem `hey`, który wysyła zadaną liczbę równoległych zapytań przez zadany czas.

```
hey -z 2m -c 50 -host kolory.zielona.example http://10.20.10.44/
```

Flaga `-z` ustawia czas testu, `-c` liczbę równoległych połączeń, a `-host` nagłówek `Host`, po którym Traefik wybiera regułę Ingress. Zapytania idą przez Traefika, więc obciążają także jego pod, a nie tylko nginksa. Program instalujemy według jego strony projektu, a dokładną wersję przypinamy tak jak każde inne narzędzie.

Częste błędy

Objaw	Przyczyna	Co robimy
`kubectl top` kończy się `error: Metrics API not available`	metrics-server nie działa albo k3s uruchomiono z `--disable metrics-server`	`kubectl get apiservice v1beta1.metrics.k8s.io`, `kubectl get pods -n kube-system`
`TARGETS` pokazuje `<unknown>/50%` zaraz po utworzeniu HPA	metrics-server nie zebrał jeszcze pierwszej próbki	czekamy do minuty i powtarzamy `kubectl get hpa`
`TARGETS` pokazuje `<unknown>/50%` na stałe, a w `describe` stoi `missing request for cpu`	któryś kontener poda nie ma `requests.cpu`, na przykład dopisany sidecar	dopisujemy `requests` każdemu kontenerowi aplikacyjnemu, a pojedynczy kontener można mierzyć typem `ContainerResource`
Pody się nie dokładają mimo obciążenia	zużycie mieści się w tolerancji 10 procent albo nginx jest zbyt lekki dla generatora	`kubectl top pods`, więcej replik generatora przez `kubectl scale`, albo niższy cel w `values.yaml`
Liczba podów skacze do trzech po każdym `helm upgrade`	szablon Deploymentu ma stałe `replicas` mimo włączonego HPA	warunek `if not .Values.autoscaling.enabled` wokół `replicas`
Nowe pody mają `Pending` i zdarzenie `Insufficient cpu`	suma `requests` przekracza wolne zasoby jedynego węzła	`kubectl describe node k3s01`, zmniejszamy `requests` albo `maxReplicas`
Po ustaniu obciążenia pody znikają dopiero po kilku minutach	okno stabilizacji zmniejszania wynosi domyślnie 300 sekund	zmieniamy `behavior.scaleDown.stabilizationWindowSeconds` albo czekamy
Strona zwalnia, a `kubectl top` pokazuje zużycie równe limitowi	dławienie procesora przez zbyt niski `limits.cpu`	odczyt `cpu.stat`, podnosimy limit albo go usuwamy
`OOMKilled` i `Exit Code: 137`	zużycie pamięci przekroczyło `limits.memory`	`kubectl top pod`, podnosimy limit, gdy zużycie jest realne, albo szukamy wycieku w aplikacji

□□ Ćwiczenia

1. Trzy pody mają zużycie 90 procent żądanego procesora, a cel HPA wynosi 60 procent. Ile podów będzie chciał HPA, przy `maxReplicas` równym 8?

Odpowiedź

Wzór daje 3 razy 90 podzielone przez 60, czyli 4,5, a po zaokrągleniu w górę 5 podów. Wynik mieści się w limicie 8 i w tempie zwiększania (dwukrotnie albo o cztery pody na 15 sekund), więc HPA ustawi pięć replik jednym krokiem.

1. Dlaczego procent w kolumnie `TARGETS` zmienia się, gdy podnosimy `requests.cpu`, choć aplikacja zużywa tyle samo procesora?

Odpowiedź

HPA liczy zużycie jako procent żądania, a nie limitu ani mocy węzła. Dwukrotnie większe `requests` oznacza dwukrotnie mniejszy procent przy tym samym zużyciu bezwzględnym, więc autoskaler dokłada pody później.

1. Dodajemy do poda sidecar bez `requests`, a HPA przestaje działać. Jak to wyjaśnić i co poprawić?

Odpowiedź

HPA liczy procent względem sumy `requests` wszystkich kontenerów poda, a kontener bez `requests` uniemożliwia policzenie ilorazu. W `kubectl describe hpa` widzimy komunikat `missing request for cpu`. Poprawiamy to, dopisując sidecarowi `requests.cpu`, albo zmieniamy metrykę na `ContainerResource` ze wskazaniem kontenera nginx.

1. Pod ma kontener główny z `requests` równymi `limits` i kontener inicjalizujący bez zasobów. Jaką klasę QoS dostanie?

Odpowiedź

Burstable. Do klasy Guaranteed wszystkie kontenery poda, także inicjalizujące, muszą mieć `requests` równe `limits` dla procesora i pamięci. Jeden kontener bez zasobów wystarczy, żeby pod nie był Guaranteed.

1. Zwiększamy generator do sześciu podów. Co zobaczymy w `kubectl get hpa -w` i dlaczego liczba podów nie przekroczy 8?

Odpowiedź

`TARGETS` wzrośnie, a `REPLICAS` dojdzie do 8 i się zatrzyma, bo `maxReplicas` jest twardym sufitem. W `describe hpa` pojawi się warunek `ScalingLimited` ze statusem `True` i powodem `TooManyReplicas`. Alert o takim stanie dopisujemy na następnej stronie.

Powiązane: Helm: kolorowy nginx jako pakiet · Pod: najmniejsza jednostka · Deployment, ReplicaSet i aktualizacje bez przerwy · Diagnostyka: objaw, przyczyna, pierwsze kroki · Obserwowalność klastra: metryki, logi i alerty bez bólu

Źródła: kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (wzór, tolerancja 10 procent, okres 15 sekund, okno stabilizacji 300 sekund, typ ContainerResource) · kubernetes.io/docs/concepts/workloads/pods/pod-qos/ ·

kubernetes.io/docs/concepts/configuration/manage-resources-containers/ · docs.k3s.io/installation/packaged-components (metrics-server w k3s) · github.com/kubernetes-sigs/metrics-server · github.com/rakyll/hey

---

Revision #17
Created 2026-10-03 09:39:08 UTC by Tomasz Konieczny
Updated 2026-10-26 14:17:20 UTC by Tomasz Konieczny