Diagnostyka: objaw, przyczyna, pierwsze kroki

Typ: referencja **Dotyczy: Kubernetes 1.30+** **Czas: ~25 min**

Jedenaście sytuacji, z którymi administrator Kubernetesa styka się najczęściej — dla każdej: co widzimy, jaką komendą to potwierdzamy, co robimy i czego świadomie nie robimy.

Skróty użyte na tej stronie (2)

Skrót	Co znaczy
PV	Physical Volume: dysk lub partycja oddana pod LVM; w Kubernetesie PersistentVolume, trwały wolumen
PVC	PersistentVolumeClaim: zamówienie trwałego dysku przez aplikację w Kubernetesie

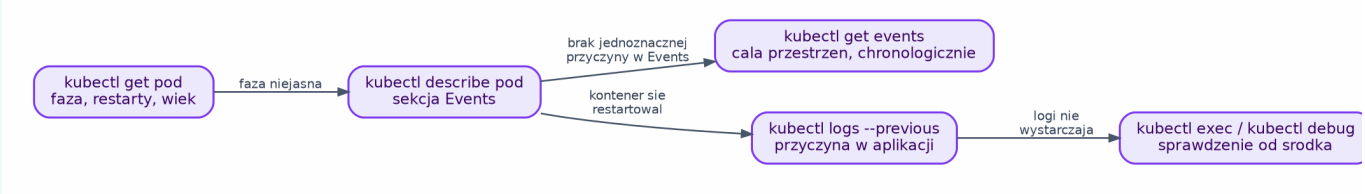
Fiszka

TYLKO ODCZYT

Kolejność diagnostyczna: get, describe, events, logs, exec

```
kubectl get pod NazwaPoda -n NazwaPrzestrzeni
```

Pierwszy rzut oka — status, liczba restartów, wiek poda. Mówi, w jakiej fazie utknął problem, zanim zajrzymy głębiej.



Diagnostykę zaczynamy od kubectl get, przechodzimy do describe i dalej do events albo logs, a do exec czy debug sięgamy dopiero, gdy logi nie wystarczają (kliknięcie otwiera pełny rozmiar)

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Szczegóły obiektu i historia zdarzeń: kubectl describe

```
kubectl describe pod NazwaPoda -n NazwaPrzestrzeni
```

Pełny opis poda razem z sekcją `Events` na samym końcu — w większości przypadków z tej strony to właśnie tam widać pierwszą, surową przyczynę problemu, zanim zajrzymy do logów aplikacji.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Zdarzenia całej przestrzeni nazw posortowane w czasie: kubectl get events

```
kubectl get events -n NazwaPrzestrzeni --sort-by=.lastTimestamp
```

Sprawdzamy zdarzenia całej przestrzeni nazw w kolejności chronologicznej — przydatne, gdy nie wiemy jeszcze, który obiekt jest źródłem problemu.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Logi kontenera: kubectl logs

```
kubectl logs NazwaPoda -n NazwaPrzestrzeni --previous
```

Sprawdzamy logi z poprzedniej instancji kontenera (flaga `--previous`) — kluczowe przy `CrashLoopBackOff`, gdzie bieżący kontener jeszcze nie zdążył nic zalogować, a to poprzednia instancja się wywróciła.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Wejście do działającego kontenera: kubectl exec

```
kubectl exec -it NazwaPoda -n NazwaPrzestrzeni -- sh
```

Wchodzimy do wnętrza działającego kontenera, żeby sprawdzić coś, czego logi nie pokazują — dostępność pliku, zmienną środowiskową, łączność sieciową z tego konkretnego miejsca.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Podłączenie efemerycznego kontenera diagnostycznego

```
kubectl debug -it NazwaPoda -n NazwaPrzestrzeni --image=busybox:1.36 --
target=NazwaKontenera
```

Dołączamy tymczasowy kontener z narzędziami diagnostycznymi do już działającego poda, przydatne zwłaszcza przy obrazach typu distroless, które nie mają wewnątrz żadnej powłoki do `exec`.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

Z ŻYCIA WZIĘTE

O 6:45 Tomek dostaje alert, że pod aplikacji raportowej na klastrze produkcyjnym jest w stanie `CrashLoopBackOff`, i od razu sięga po `kubectl logs` bez flagi `--previous`. Widzi pustą odpowiedź, bo świeżo zrestartowany kontener jeszcze nic nie zdążył zalogować, a to poprzednia instancja się wywróciła. Po dodaniu `--previous` w logach widzi dokładny błąd połączenia z bazą danych, sprawdza `kubectl describe pod` dla exit code i w kilka minut ustala, że baza była chwilowo niedostępna podczas własnej aktualizacji.

Tabela: objaw → prawdopodobna przyczyna → pierwsze kroki

Objaw	Prawdopodobna przyczyna	Pierwsze kroki
Pod w stanie `Pending`	brak zasobów na węzłach, brak pasującego węzła, PVC nie może się zbindować	`kubectl describe pod`, sekcja `Events`
`ImagePullBackOff` / `ErrImagePull`	zła nazwa obrazu, brak `imagePullSecrets`, rejestr niedostępny	`kubectl describe pod`, sprawdzenie nazwy obrazu i sekretu
`CrashLoopBackOff`	aplikacja kończy działanie zaraz po starcie	`kubectl logs --previous`, `kubectl describe pod`
`OOMKilled`	proces przekroczył `limits.memory`	`kubectl describe pod`, `kubectl top pod`
`Init:Error` / `Init:CrashLoopBackOff`	kontener inicjujący kończy się błędem przed startem właściwego kontenera	`kubectl logs -c NazwaKontenera --previous` na kontenerze inicjującym
pod działa, ale Service nie odpowiada	selektor nie pasuje, zły `targetPort`, brak endpointów	`kubectl get endpoints`, `kubectl describe svc`
Ingress zwraca 502 albo 404	backend niedostępny (502) albo zła reguła hosta/ścieżki (404)	logi kontrolera Ingress, `kubectl describe ingress`
węzeł `NotReady`	kubelet nie zgłasza się do API, awaria sieci węzła, awaria CNI na węźle	`kubectl describe node`, `journalctl -u kubelet` na węźle
brak miejsca na węźle, eksmisja podów	zapełniony dysk węzła (`ephemeral-storage`)	`kubectl describe node`, `df -h` na węźle
pod nie może zamontować wolumenu	zły tryb dostępu, PVC nie zbindowany, awaria sterownika CSI	`kubectl describe pod`, `kubectl get pvc`
DNS nie rozwiązuje nazw	awaria CoreDNS, `NetworkPolicy` blokująca port 53	`kubectl get pods -n kube-system -l k8s-app=kube-dns`

Pod w stanie Pending

Co widzimy:

```
NAME              READY   STATUS    RESTARTS   AGE
sklep-api-7f9d4   0/1     Pending   0          4m
```

Jaką komendą to potwierdzamy:

```
kubectl describe pod sklep-api-7f9d4 -n produkcja
```

```
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  4m    default-scheduler   0/3 nodes are available: 3 Insufficient
```

```
memory.
```

Co robimy: sprawdzamy zapotrzebowanie zadeklarowane w `resources.requests` poda względem realnie dostępnych zasobów na węzłach (`kubectl describe nodes` albo `kubectl top nodes`) — najczęstszą przyczyną jest zbyt wysoki `requests` względem faktycznej pojemności klastra, albo zwykły brak wolnych zasobów po wdrożeniu kolejnej aplikacji. Sprawdzamy też `taints` węzłów i `tolerations` poda, jeśli komunikat mówi o niepasujących węzłach, a nie o braku zasobów.

Czego nie robimy: nie podnosimy limitów zasobów całego klastra odruchowo, zanim sprawdzimy, czy zadeklarowane `requests` w ogóle odpowiadają rzeczywistym potrzebom aplikacji — zawyżone `requests` „na zapas" jest częstszą przyczyną tego objawu niż faktyczny brak sprzętu.

ImagePullBackOff

Co widzimy:

```
NAME              READY   STATUS             RESTARTS   AGE
sklep-api-7f9d4    0/1     ImagePullBackOff   0          2m
```

Jaką komendą to potwierdzamy:

```
kubectl describe pod sklep-api-7f9d4 -n produkcja
```

```
Events:
  Type     Reason   Age   From     Message
  ----     ------   ----  ----     -------
  Warning  Failed   90s   kubelet  Failed to pull image "git.zielona.example/zielona/sklep-
api:1.9": unauthorized
```

Co robimy: sprawdzamy dokładną nazwę i tag obrazu pod kątem literówki, sprawdzamy istnienie i poprawność `imagePullSecrets` opisanego na stronie ConfigMap i Secret, i próbujemy ręcznie pobrać ten sam obraz poleceniem `docker pull` albo `crictl pull` z tego samego węzła, żeby wykluczyć problem z samym rejestrem.

Czego nie robimy: nie usuwamy i nie tworzymy poda od nowa w nadziei, że problem sam zniknie — bez poprawki nazwy obrazu albo sekretu nowy pod trafi w dokładnie ten sam błąd.

CrashLoopBackOff

Co widzimy:

```
NAME              READY   STATUS             RESTARTS   AGE
sklep-api-7f9d4    0/1     CrashLoopBackOff   7          6m
```

Jaką komendą to potwierdzamy:

```
kubectl logs sklep-api-7f9d4 -n produkcja --previous
```

```
Error: connect ECONNREFUSED 10.96.142.7:5432
    at TCPConnectWrap.afterConnect
```

Co robimy: czytamy logi poprzedniej instancji kontenera (`--previous`), bo bieżąca instancja po restarcie zwykle nie zdążyła jeszcze nic zalogować — samo `CrashLoopBackOff` mówi tylko, że proces kończy się z błędem, konkretną przyczynę pokazują dopiero logi. Sprawdzamy też `exit code` w `kubectl describe pod`, bo różni się w zależności od przyczyny (128+sygnał dla zabicia przez system, kod aplikacji dla świadomego zakończenia).

Czego nie robimy: nie zwiększamy `restartPolicy` ani liczby prób w nadziei, że kontener w końcu wystartuje — rosnący odstęp między kolejnymi próbami restartu jest mechanizmem wbudowanym (backoff), a nie usterką do obejścia.

OOMKilled

Co widzimy:

```
NAME              READY   STATUS      RESTARTS   AGE
sklep-api-7f9d4    0/1     OOMKilled   3          10m
```

Jaką komendą to potwierdzamy:

```
kubectl describe pod sklep-api-7f9d4 -n produkcja
```

```
Last State:     Terminated
  Reason:       OOMKilled
  Exit Code:    137
```

Co robimy: sprawdzamy realne zużycie pamięci przez aplikację w normalnych warunkach (`kubectl top pod`) i porównujemy je z ustawionym `limits.memory` — albo podnosimy limit, jeśli aplikacja rzeczywiście go potrzebuje, albo szukamy przyczyny wycieku pamięci po stronie samej aplikacji, jeśli zużycie rośnie bez końca.

Czego nie robimy: nie usuwamy limitu pamięci całkowicie, żeby problem „zniknął” — pod bez `limits.memory` może w chwili szczytu zająć całą dostępną pamięć węzła i doprowadzić do eksmisji sąsiadujących podów, zamiast tylko siebie.

Init:Error

Co widzimy:

```
NAME              READY   STATUS       RESTARTS   AGE
sklep-api-7f9d4    0/1     Init:Error   2          3m
```

Jaką komendą to potwierdzamy:

```
kubectl logs sklep-api-7f9d4 -n produkcja -c czekaj-na-baze --previous
```

```
psql: error: connection to server at "baza.produkcja.svc.cluster.local", port 5432 failed:
Connection refused
```

Co robimy: sprawdzamy logi konkretnego kontenera inicjującego po nazwie (flaga `-c`), bo pod może mieć kilka kontenerów inicjujących uruchamianych po kolei, i tylko ten, który akurat zawiódł, ma znaczenie. Sprawdzamy, czy zasób, na który kontener inicjujący czeka (tu: baza danych), w ogóle istnieje i odpowiada.

Czego nie robimy: nie pomijamy kontenera inicjującego przez usunięcie go z manifestu tylko po to, żeby główny kontener wystartował — kontenery inicjujące zwykle zabezpieczają jakiś warunek wstępny, a jego pominięcie przenosi ten sam problem głębiej, w mniej czytelne miejsce.

Pod działa, ale Service nie odpowiada

Co widzimy: `kubectl get pod` pokazuje `Running` i `1/1 Ready`, ale `curl` do adresu Service kończy się `Connection refused` albo przekroczeniem czasu.

Jaką komendą to potwierdzamy:

```
kubectl get endpoints NazwaUslugi -n produkcja
```

```
NAME          ENDPOINTS   AGE
sklep-api     <none>      15m
```

Co robimy: sprawdzamy, czy selektor usługi (`kubectl get svc -o yaml`) faktycznie pasuje do etykiet poda (`kubectl get pod --show-labels`) — pusta lista endpointów przy działającym podzie niemal zawsze oznacza niedopasowanie selektora albo pod, który nie przeszedł jeszcze próby gotowości (`readinessProbe`). Sprawdzamy też zgodność `targetPort` z portem, na którym faktycznie nasłuchuje aplikacja wewnątrz kontenera, mechanizm opisany na stronie Service: jak ruch trafia do poda.

Czego nie robimy: nie zmieniamy typu usługi (np. z `ClusterIP` na `NodePort`) w nadziei, że to naprawi brak endpointów — zmiana typu nie ma wpływu na dobór podów przez selektor, problem zostaje dokładnie ten sam.

Ingress zwraca 502 albo 404

Co widzimy: klient dostaje odpowiedź od kontrolera Ingress, więc sam kontroler działa, ale odpowiedź jest błędem.

Jaką komendą to potwierdzamy:

```
kubectl logs -n ingress-nginx -l app.kubernetes.io/component=controller --tail=50
```

```
2026/09/24 10:15:02 [error] connect() failed (111: Connection refused) while connecting to
upstream, client: 203.0.113.10, upstream: "http://10.244.1.12:8080/"
```

Co robimy: przy 502 sprawdzamy, czy usługa wskazana w regule Ingressa ma działające endpointy (dokładnie ten sam krok co w poprzednim przypadku) — kontroler dostał adres backendu, ale nikt tam nie odpowiada. Przy 404 sprawdzamy dopasowanie `host` i `pathType` w regule względem żądania klienta, mechanizm opisany na stronie Ingress i Gateway API: wejście z zewnątrz.

Czego nie robimy: nie zakładamy, że błąd 502 czy 404 pochodzi od aplikacji za Ingressem, zanim sprawdzimy logi samego kontrolera — te dwa kody błędów generuje najczęściej sam kontroler, zanim ruch w ogóle dotrze do backendu.

Węzeł NotReady

Co widzimy:

```
NAME    STATUS     ROLES    AGE   VERSION
k3s03   NotReady   <none>   90d   v1.30.4
```

Jaką komendą to potwierdzamy:

```
kubectl describe node k3s03
```

```
Conditions:
  Type             Status   Reason
  ----             ------   ------
  Ready            Unknown   NodeStatusUnknown
```

Co robimy: logujemy się na sam węzeł i sprawdzamy stan usługi kubelet (`systemctl status kubelet`, `journalctl -u kubelet -n 100`) — `NodeStatusUnknown` oznacza, że płaszczyzna sterowania od jakiegoś czasu nie dostaje żadnego sygnału (heartbeat) od tego węzła, co wskazuje na awarię kubeleta, awarię sieci między węzłem a płaszczyzną sterowania, albo pełne przeciążenie węzła uniemożliwiające jakąkolwiek odpowiedź.

Czego nie robimy: nie usuwamy węzła z klastra pochopnie, dopóki nie potwierdzimy, że jest trwale niedostępny — pochopne usunięcie węzła, który za chwilę wróci, tworzy niepotrzebny ruch przenoszenia obciążenia w obie strony.

Brak miejsca na węźle i eksmisja podów

Co widzimy:

```
NAME              READY   STATUS    RESTARTS   AGE
sklep-api-7f9d4   0/1     Evicted   0          2m
```

Jaką komendą to potwierdzamy:

```
kubectl describe node k3s03
```

```
Conditions:
  Type             Status
  ----             ------
  DiskPressure     True
```

Co robimy: sprawdzamy zajętość dysku bezpośrednio na węźle (`df -h`), identyfikujemy, co go zapełnia — najczęściej nagromadzone, nieużywane obrazy kontenerów albo logi kontenerów bez rotacji — i czyścimy zgodnie z procedurą opisaną na stronie Utrzymanie klastra. Sam pod w stanie `Evicted` można bezpiecznie usunąć po ustąpieniu przyczyny — kontroler wystawi go ponownie.

Czego nie robimy: nie usuwamy ręcznie plików wewnątrz katalogów danych innych, działających kontenerów w poszukiwaniu wolnego miejsca — ryzyko uszkodzenia danych działającej aplikacji jest większe niż korzyść z doraźnie odzyskanych megabajtów.

Pod nie może zamontować wolumenu

Co widzimy:

```
NAME              READY   STATUS              RESTARTS   AGE
sklep-baza-0       0/1     ContainerCreating   0          5m
```

Jaką komendą to potwierdzamy:

```
kubectl describe pod sklep-baza-0 -n produkcja
```

```
Events:
  Type     Reason       Age   From                     Message
  ----     ------       ----  ----                     -------
  Warning  FailedMount  90s   kubelet                   Unable to attach or mount volumes:
unmounted volumes=[dane]: timed out waiting for the condition
```

Co robimy: sprawdzamy stan samego PVC (`kubectl get pvc`) — jeśli jest w stanie `Pending`, problem leży w StorageClass albo w braku pasującego PV, opisanym na stronie Wolumeny, PV, PVC i StorageClass. Jeśli PVC jest `Bound`, sprawdzamy, czy wolumen w trybie `ReadWriteOnce` nie jest już zamontowany do poda na innym węźle — pod nie przeniesie się na nowy węzeł, dopóki poprzedni nie zwolni wolumenu.

Czego nie robimy: nie usuwamy PVC w nadziei na szybszą naprawę, jeśli polityka odzyskiwania to `Delete` — razem z PVC znika też fizyczny wolumen i wszystkie dane na nim.

DNS nie rozwiązuje nazw

Co widzimy: aplikacja zgłasza błędy typu `getaddrinfo ENOTFOUND` albo `Temporary failure in name resolution` dla nazw usług, które istnieją.

Jaką komendą to potwierdzamy:

```
kubectl get pods -n kube-system -l k8s-app=kube-dns
```

```
NAME                       READY   STATUS             RESTARTS   AGE
coredns-6f9c4b8d5f-2xk9m   0/1     CrashLoopBackOff   5          8m
coredns-6f9c4b8d5f-p7qwl   1/1     Running            0          21d
```

Co robimy: sprawdzamy stan podów CoreDNS i ich logi, sprawdzamy istnienie `NetworkPolicy` blokującej ruch do portu 53, i sprawdzamy `resolv.conf` w problematycznym podzie, dokładnie tak jak na stronie DNS w klastrze i wykrywanie usług.

Czego nie robimy: nie restartujemy poda aplikacji jako pierwszy odruch — jeśli przyczyna leży w CoreDNS albo w polityce sieciowej, nowy pod aplikacji trafi na dokładnie ten sam problem.

Powiązane: Service: jak ruch trafia do poda · DNS w klastrze i wykrywanie usług · Ingress i Gateway API: wejście z zewnątrz · Wolumeny, PV, PVC i StorageClass · ConfigMap i Secret · Utrzymanie klastra

Źródła: dokumentacja kubernetes.io, rozdziały „Debug Running Pods", „Debug Services", „Troubleshoot Clusters" i „Determine the Reason for Pod Failure"

Revision #34
Created 2026-09-25 10:33:11 UTC by Tomasz Konieczny
Updated 2026-10-26 14:17:13 UTC by Tomasz Konieczny