K8s Troubleshooting Playbook

Bavithran

bavicnative®

Bavithran
@bavicnative

1. The "Ghost" Image (ImagePullBackOff)

The Problem:

Your deployment is stuck in **ImagePullBackOff**. You've verified the image name is correct and it exists in your private registry (like AWS ECR or Docker Hub).

The Investigation:

Run **kubectl describe pod <pod-name>**. Look at the Events section. If you see "Failed to pull image... unauthorized," K8s cannot authenticate with your registry.

The Real-World Solution:

1. Create a Docker-registry secret: **kubectl create secret docker-registry my-registry-key ...**
2. Crucial Step: Ensure the secret is in the same namespace as your deployment.
3. Update your **YAML to include: yaml spec: imagePullSecrets: - name: my-registry-key**

2. The Silent Killer (OOMKilled)

The Problem:

Your Java or Node.js pod starts fine, but after 10 minutes of heavy traffic, it restarts. **kubectl get pods** shows a status of **CrashLoopBackOff**, but the reason is **OOMKilled**.

The Investigation:

Check **kubectl describe pod**. Under **Last State**, you'll see **Reason: OOMKilled**. This means the container exceeded its defined memory limit.

The Real-World Solution:

1. For Java: Ensure your **-Xmx** (Heap size) is roughly 75-80% of your container limit. If they are equal, the JVM will be killed by K8s before it can trigger its own Garbage Collection.
2. Increase the **limits.memory** in your manifest.
3. Implement a Vertical Pod Autoscaler (VPA) to observe and recommend the right memory sizes based on real usage.

3.The "It Works on My Machine" (Config Errors)

The Problem:

The pod status is **CrashLoopBackOff**. When you check the logs (**kubectl logs**), it says: **java.io.FileNotFoundException: /app/config/settings.yaml (No such file or directory)**.

The Investigation:

You realize you created the **ConfigMap**, but the application can't see it.

The Real-World Solution:

1. Verify the **volumeMounts** path in your container matches the exact path your application code is looking at.
2. Check if the **ConfigMap** name in the **volumes** section is spelled correctly.
3. Pro-Tip: Remember that mounting a ConfigMap to a directory that already contains files in the image will hide those existing files. Use **subPath** if you only want to add one specific file.

4.The 503 Service Unavailable (Selector Mismatch)

The Problem:

Your Ingress and Service are set up. You hit the URL, and you get a **503 Service Unavailable.**

The Investigation:

Run **kubectl get endpoints <service-name>**. If the **ENDPOINTS column is <none>**, the Service is "homeless"—it can't find any pods to send traffic to.

The Real-World Solution:

1. Check the **spec.selector** in your Service YAML.
2. Check the **metadata.labels** in your Deployment's pod template (not the deployment labels).
3. They must match exactly. Even a small typo or a missing "app: my-app" label will cause the Service to fail.

5.The "Zombie" Node (NodeNotReady)

The Problem:
Your application is down because several pods are in **Unknown** state. **kubectl get nodes** shows a node as **NotReady**.

The Investigation:
SSH into the node and run **journalctl -u kubelet**. You might see "disk pressure" or "out of memory" at the OS level.

The Real-World Solution:
1. Immediate Fix: Clear space. Run **docker image prune -a** (if using Docker shim) or check **/var/log** for massive unrotated log files.
2. Permanent Fix: Set up **Eviction Thresholds** in your Kubelet configuration so K8s gracefully moves pods before the node completely freezes.
3. Ensure you have an **Image Garbage Collector policy** set up.


Bavithran
@bavicnative

6.The DNS Identity Crisis (CoreDNS Issues)

The Problem:

Microservice A tries to call Microservice B using its service name (**http://orders-svc**), but the connection fails with "Temporary failure in name resolution."

The Investigation:

Run a temporary pod and try to nslookup the service: **kubectl run curl --image=radial/busyboxplus:curl -i --tty**. If it fails, the issue is cluster-wide DNS.

The Real-World Solution:

1. Check CoreDNS health: **kubectl get pods -n kube-system -l k8s-app=kube-dns**.
2. Check the logs: **kubectl logs -n kube-system -l k8s-app=kube-dns**. Often, CoreDNS is failing because it hit its own memory limit.
3. The Fix: Increase the memory limits for the CoreDNS deployment or scale it to more replicas to handle the request load.

7.The "Stuck" Finalizer (Namespace Deletion)

The Problem:

You deleted a namespace (**kubectl delete ns dev**), but it has been stuck in **Terminating** for 30 minutes. You can't recreate it.

The Investigation:

Run **kubectl get namespace dev -o yaml**. Look for the **finalizers** section. K8s is waiting for a resource inside the namespace to be cleaned up, but that resource is orphaned.

The Real-World Solution:

1. Identify what's left: **kubectl get all -n dev**.
2. If it's a CRD or a PVC that won't die, edit the resource directly: **kubectl edit <resource-type> <name> -n dev**.
3. Manually delete the lines under **finalizers**:. Once saved, the namespace will vanish instantly.

8.The CPU Throttling Mystery

The Problem:
Your application is incredibly slow (high latency), but your monitoring tool shows CPU usage is only at 40% of the limit.

The Investigation:
Look at the "CPU Throttling" metric (if using Prometheus) or **top** inside the container. If the app tries to burst above its **limit**, K8s "throttles" the CPU cycles, even if the Node has 90% idle CPU.

The Real-World Solution:
1. Avoid setting CPU **limits** too strictly for bursty applications (like Java or Go).
2. Strategy: Set a high **request** to guarantee performance and a much higher (or no) **limit** to allow for spikes. This prevents the "Completely Fair Scheduler" (CFS) from killing your app's speed.

9.The Pending Pod (Resource Exhaustion)

The Problem:
You triggered a new deployment, but the new pods are stuck in **Pending** forever.

The Investigation:
Run **kubectl describe pod <pod-name>**. You will see an event: **0/5 nodes are available: 5 Insufficient cpu**.

The Real-World Solution:
1. Scale Up: Add more nodes to your worker group or use a Cluster Autoscaler.
2. Optimization: Review your **requests**. Often, developers set **requests: cpu: 1** when the app only uses **0.1.** This "squats" on resources that aren't being used, preventing other pods from scheduled.

10.The DatabaseTimeout (NetworkPolicy)

The Problem:

The app is running, but it can’t reach the managed Database (like AWS RDS). You get "Connection Timeout."

The Investigation:

Check if a **NetworkPolicy** was recently applied to the namespace. By default, K8s is "Allow All," but once you add one policy, it becomes "Deny All" for anything not explicitly allowed.

The Real-World Solution:

1. Ensure your **NetworkPolicy** has an **egress** rule that allows traffic to the DB's IP range and port (e.g., 5432 for Postgres).
2. Check the Cloud Security Group. Ensure the K8s Node's IP is whitelisted in the Database's inbound rules.

Found this useful?

Follow

Let's connect

💬 Found it useful? Drop your thoughts below and share it with your fellow DevOps engineers!