

OpenShift Troubleshooting Knowledge Base

This space serves as the central knowledge base for diagnosing and resolving common issues within our OpenShift environment. It is built from post-mortems and incident reports to create actionable, topic-based playbooks.

Categories:

- **OpenShift Application Troubleshooting** (Problems with workloads, e.g., Deployments, Pods, Services)
- **OpenShift Platform Troubleshooting** (Problems with the control plane or cluster infrastructure, e.g., Nodes, etcd)

OpenShift Application Troubleshooting

Guides for issues related to services running on the cluster.

1. Playbook: Troubleshooting Pod CrashLoopBackOff

Summary This state indicates that a pod's container is starting, crashing, and then being restarted by OpenShift, in a continuous loop. The pod's status will show **CrashLoopBackOff**.

Common Root Causes

- **Missing Configuration:** The application crashes because a required environment variable, ConfigMap, or Secret is missing or incorrect.
- **Application Error:** The application itself is failing on startup (e.g., code panic, invalid configuration file, failed database connection).
- **Failed Liveness/Readiness Probes:** The probes are misconfigured (e.g., wrong port, bad health check endpoint, timeout too low), causing OpenShift to kill the pod.
- **Incorrect Container **command** or **args**:** The entrypoint for the container is invalid or pointing to a non-existent file.

Diagnostic Playbook

1. **Check Pod Status:** Identify the failing pod.

None

```
oc get pods -n <namespace>
```

2. **Describe the Pod:** This shows the reason for the current state and any events.

None

```
oc describe pod <pod-name> -n <namespace>
```

- Look at the **Events** section for clues.
 - Look at the **Last State** section. It will often show **Reason: Error** and an **Exit Code**.
3. **Check Logs (Previous Container):** This is the most critical step. A pod in **CrashLoopBackOff** is constantly restarting, so its current logs are often empty. You must check the logs from the *previous* terminated container.

None

```
oc logs <pod-name> -n <namespace> --previous
```

- This log will almost always contain the fatal error that caused the crash (e.g., "Error: PAYMENT_API_KEY environment variable not set").

Resolution & Prevention

- **Resolution:**
 - Fix the underlying issue found in the logs (e.g., add the missing environment variable, correct the app code, fix the probe).
 - Apply the fix to your **DeploymentConfig** (DC) or **Deployment**.

None

```
oc apply -f <your-manifest.yaml>
```

- If using a **Deployment**, trigger a rollout to pick up the new config:

None

```
oc rollout restart deployment/<deployment-name> -n <namespace>
```

- (A change to a **DeploymentConfig**'s pod template will typically trigger a new rollout automatically).
- **Prevention:**
 - **Validate Configuration:** Use CI/CD linting tools to ensure all required ConfigMaps, Secrets, and env vars are present in manifests before deployment.
 - **Application Fallbacks:** Code applications to handle missing non-critical configuration with sane defaults, rather than crashing.

Key Alert (Prometheus) Alert when any container is in a **CrashLoopBackOff** state.

None

```
kube_pod_container_status_waiting_reason{reason="CrashLoopBackOff"} > 0
```

Incident Example (APP-01)

- **Summary:** **payment-service** pods entered **CrashLoopBackOff**.
- **Root Cause:** `oc logs --previous` showed "Fatal error: Database connection failed". A **ConfigMap** update had applied an incorrect database URL.
- **Resolution:** The **ConfigMap** was corrected, and the **DeploymentConfig** was re-deployed with `oc rollout latest dc/payment-service`.

2. Playbook: Troubleshooting ImagePullBackOff

Summary This state indicates that the kubelet (on the worker node) is unable to pull the container image specified in the pod definition. The pod will be stuck in a **Pending** state with the **ImagePullBackOff** or **ErrImagePull** reason.

Common Root Causes

- **Incorrect Image Name/Tag:** The image name or tag in the **DeploymentConfig/Deployment** is misspelled or does not exist in the registry (e.g., `my-app:latest` instead of `my-app:latest`).

- **Authentication Failure:** The cluster does not have the necessary credentials (`imagePullSecrets`) to access a private registry.
- **Internal Registry Issue:** In OpenShift, the `ImageStreamTag` does not exist or the internal registry is unreachable.
- **Network Issues:** The node cannot resolve or route traffic to the external registry (DNS, proxy, or firewall issue).
- **Registry Rate Limiting:** The image registry (e.g., Docker Hub) is rate-limiting your cluster's IP.

Diagnostic Playbook

1. Identify the Failing Pod:

None

```
oc get pods -n <namespace>
```

- ### 2. Describe the Pod:
- This is the most important step. Check the **Events** section at the bottom.

None

```
oc describe pod <pod-name> -n <namespace>
```

- Look for events like `Failed to pull image ...: rpc error: code = Unknown desc = repository ... not found.`
- Or: `Failed to pull image ...: unauthorized: authentication required.`

- ### 3. Check Service Account:
- Verify that the pod's `ServiceAccount` has the correct `imagePullSecrets`.

None

```
# Get the pod's service account (e.g., 'default')
oc get pod <pod-name> -o jsonpath='{.spec.serviceAccountName}'

# Describe the service account
oc describe sa <service-account-name> -n <namespace>
```

- Look for the **Image pull secrets** section.
4. Check ImageStream (If using internal):

None

```
oc get is <imagestream-name> -n <namespace>
```

Resolution & Prevention

- **Resolution:**
 - **Wrong Tag:** Correct the image tag in the **DeploymentConfig** or **Deployment** and re-deploy.
 - **Auth Failure:**
 1. Create the pull secret: `oc create secret docker-registry <secret-name> --docker-server=... --docker-username=...`
 2. Link the secret to the service account: `oc secrets link <service-account-name> <secret-name> --for=pull -n <namespace>`
 - **ImageStream Issue:** Import the image: `oc import-image <is-name> --from=<image-url> --confirm -n <namespace>`
- **Prevention:**
 - **Use Specific Tags:** Avoid using the `:latest` tag, which is ambiguous. Use semantic versioning or Git SHA tags.
 - **CI Validation:** Have your CI pipeline verify that an image exists in the registry *before* updating the deployment manifests.
 - **Use Internal Registry:** Leverage the internal OpenShift registry and **ImageStreams** to manage image promotion and access.

Key Alert (Prometheus)

None

```
kube_pod_container_status_waiting_reason{reason=~"ImagePullBackOff|ErrImagePull"} > 0
```

Incident Example (APP-02)

- **Summary:** `checkout-service` pods were stuck in `ImagePullBackOff`.

- **Root Cause:** `oc describe pod` showed "unauthorized". The team had moved their image to a new private Quay.io repository, but the `imagePullSecret` was not linked to the `default` service account in that namespace.
- **Resolution:** The secret was linked to the service account using `oc secrets link`, and the pods started successfully.

3. Playbook: Troubleshooting OOMKilled (Out of Memory)

Summary This indicates the container's process was "Out of Memory Killed" (OOMKilled) by the Linux kernel. This happens because the container tried to use more memory than its `resources.limits.memory` setting allowed. The pod will restart, and its `RESTARTS` count will increase.

Common Root Causes

- **Memory Leak:** The application has a bug causing its memory usage to grow over time until it hits the limit.
- **Under-provisioned Limit:** The memory limit is set too low for the application's normal operation (e.g., a traffic spike, or a JVM `-Xmx` setting that is too close to or larger than the container limit).
- **Incorrect `requests` vs. `limits`:** The memory `request` might be set low, allowing the pod to be scheduled, but the `limit` is too low for the actual workload.

Diagnostic Playbook

1. Identify Restarting Pod:

None

```
oc get pods -n <namespace>
```

- Look for a pod with a high `RESTARTS` count.

2. Describe the Pod: This confirms the reason for the restart.

None

```
oc describe pod <pod-name> -n <namespace>
```

- Look at the **Last State** section for one of the containers.
- You will see **Reason: OOMKilled**.

3. Check Metrics (OpenShift Monitoring):

- Go to the OpenShift Console > Observe > Metrics.
- Run a query for `container_memory_usage_bytes{pod="<pod-name>", namespace="<namespace>"}`.
- You will see a "sawtooth" pattern: the memory climbs, hits the limit, and then drops to zero as the container is killed and restarts.

4. Check Logs: Check the logs from the *previous* container.

None

```
oc logs <pod-name> -n <namespace> --previous
```

- You often won't see a "crash" error, just an abrupt end to the log stream. For Java apps, you might see an `java.lang.OutOfMemoryError` if it was the JVM heap that was filled, but an OOMKill can happen before the JVM heap is exhausted.

Resolution & Prevention

● Resolution:

- **Short-Term:** Increase the memory limit to stabilize the service.

None

```
oc edit dc/<dc-name> # or oc edit deployment/<deployment-name>
# Find spec.template.spec.containers[...].resources.limits.memory
```

- **Long-Term (Leak):** Use application-specific tools (e.g., Java heap dumps, Go pprof) to identify and fix the memory leak in the code.
- **Long-Term (Tuning):** Perform load testing to determine the application's true memory high-water mark and set `limits` and `requests` appropriately. (e.g., `limit = 1.25 * high-water-mark`).

● Prevention:

- **Tune JVMs:** If running Java, set the `-Xmx` (max heap size) to be ~75-80% of the container's memory limit. This allows headroom for other processes and non-heap memory.
- **Application Monitoring (APM):** Use APM tools to spot memory leak trends *before* they cause OOMKills.

- **Set Requests = Limits:** For critical applications, setting memory **requests** equal to **limits** provides a "Guaranteed" QoS class, making the pod less likely to be killed.

Key Alert (Prometheus)

None

```
rate(kube_pod_container_status_terminated_reason{reason="OOMKilled"}[5m]) > 0
```

Incident Example (APP-03)

- **Summary:** **user-profile-api** (a Java app) was restarting every 30 minutes.
- **Root Cause:** **oc describe pod** showed **Last State: Reason: OOMKilled**. The container limit was **1Gi**. The JVM was configured with **-Xmx1g**, leaving no room for the OS, other processes, or Java's own non-heap memory.
- **Resolution:** The container limit was raised to **1.5Gi** and the **-Xmx** was explicitly set to **1G** (a safer 75% of the new limit would be **1152m**).

4. Playbook: Troubleshooting PVC Not Bound / Volume Attachment Failure

Summary This issue occurs when a pod fails to start and gets stuck in a **Pending** state. The pod's events show errors like **FailedAttachVolume**, **FailedMount**, or **PersistentVolumeClaim is not bound**.

Common Root Causes

- **StorageClass Issues:** The **StorageClass** specified in the **PersistentVolumeClaim** (PVC) does not exist, is misconfigured, or the underlying cloud provisioner is failing.
- **Zone Mismatch:** The **PersistentVolume** (PV) exists in a different availability zone (e.g., **us-east-1a**) than the node the pod is scheduled on (e.g., **us-east-1b**). Cloud disks (like EBS, GCP PD) generally cannot be attached across zones.
- **Volume in Use:** The volume is already mounted by another node. This is common with **ReadWriteOnce** (RWO) volumes. If a pod on **node-A** fails without detaching, **node-B** cannot attach the volume.
- **Cloud Provider Quotas:** You have hit a limit on the number of volumes that can be created in your account/region or attached to a single node.

Diagnostic Playbook

1. Identify Pending Pod:

None

```
oc get pods -n <namespace>
```

2. **Describe the Pod:** This is the most important command. Check the **Events** section at the bottom.

None

```
oc describe pod <pod-name> -n <namespace>
```

- Look for events like:
 - `Warning FailedAttachVolume ... AttachVolume.Attach failed for volume "pvc-..." : ... cloud provider error: volume limit exceeded`
 - `Warning FailedMount ... MountVolume.NewVolume provisioner ... failed to provision volume ...`
 - `Warning FailedScheduling ... 0/3 nodes are available: 1 node(s) had a volume node affinity conflict.`

3. **Check the PVC:** See if the PVC is stuck in **Pending** or is **Bound**.

None

```
oc get pvc <pvc-name> -n <namespace>
```

4. **Describe the PVC:** If it's **Pending**, describing it can reveal the problem.

None

```
oc describe pvc <pvc-name> -n <namespace>
```

- Look for events related to provisioning or **StorageClass** errors.

Resolution & Prevention

- **Resolution:**

- **Quota Issues:** Request an increase in your cloud provider volume limits.

- **Zone Mismatch:**
 - Set the `StorageClass`'s `volumeBindingMode` to `WaitForFirstConsumer`. This tells OpenShift to *wait* until a pod is scheduled to a node *before* it provisions the volume, ensuring the volume is created in the correct zone.
 - Delete the pod to force a reschedule: `oc delete pod <pod-name> -n <namespace>`.
- **Volume in Use:** Manually detach the volume from the old node via the cloud provider console. This is a last resort. More often, deleting the "stuck" pod and waiting (up to 5-10 min) will allow the cluster to reconcile and detach the volume properly.
- **Prevention:**
 - **Use `WaitForFirstConsumer`:** This is the most effective fix for zone-mismatch issues. Make this the default `volumeBindingMode` in your `StorageClasses`.
 - **Monitor Quotas:** Implement alerts in your cloud provider to warn when you approach 80% of volume or attachment quotas.
 - **Use `ReadWriteMany` (RWX):** For services that need to be accessible from multiple nodes, use an RWX-capable `StorageClass` (like OpenShift Data Foundation, NFS, etc.) instead of RWO.

Key Alert (Prometheus/Events) Alerting on `FailedAttachVolume` events is the most direct method. You can also track the phase of PVCs.

None

```
kube_persistentvolumeclaim_status_phase{phase="Pending"} > 0
```

Incident Example (APP-04)

- **Summary:** `orders-db-0` (a `StatefulSet` pod) was stuck in `Pending`.
- **Root Cause:** `oc describe pod` revealed `FailedScheduling` due to "volume node affinity conflict". The PV was created in `us-west-2a` but the pod was trying to schedule to a node in `us-west-2b`. The `StorageClass` was using the default `volumeBindingMode: Immediate`.
- **Resolution:** The `StorageClass` was updated to `volumeBindingMode: WaitForFirstConsumer`. The old PV and PVC were deleted, and the new PVC was created, which remained `Pending` until the pod was scheduled, at which point a new PV was correctly provisioned in `us-west-2b`.

OpenShift Platform Troubleshooting

Guides for issues related to the cluster's core components (control plane, nodes).

1. Playbook: Troubleshooting Node NotReady

Summary A worker node stops reporting its status to the control plane. The node's status changes to **NotReady** or **SchedulingDisabled**. Pods on this node are *not* immediately evicted. After a timeout (default 5 min), the control plane will mark the pods for eviction and reschedule them on healthy nodes.

Common Root Causes

- **Node Service Failure:** The **kubelet** service on the worker node has crashed, is frozen, or is misconfigured. In OpenShift 4, the **crio** (container runtime) service may also be a factor.
- **Network Partition:** The node is running, but network issues (firewall, routing, OpenShift SDN issues) prevent it from communicating with the **kube-apiserver**.
- **Resource Starvation:** The node is out of memory (**MemoryPressure**), disk (**DiskPressure**), or PIDs (**PIDPressure**), which starves the **kubelet** and stops it from reporting.
- **Hardware/VM Failure:** The underlying physical machine or virtual machine has shut down, rebooted, or failed.

Diagnostic Playbook

1. Identify the Node:

None

```
oc get nodes
```

- Look for any node with **STATUS** other than **Ready**.

2. Describe the Node: This is the most important command.

None

```
oc describe node <node-name>
```

- Check the **Conditions** section. This is the source of truth.
 - **Ready: False** confirms the problem.
 - Look for other conditions: **MemoryPressure: True, DiskPressure: True, PIDPressure: True**. These point directly to the root cause.
 - Check the **Events** for any recent errors.
3. **SSH to the Node:** This is the primary diagnostic step. (Requires cluster-admin access).

None

```
oc debug node/<node-name>
# This chroots into the node's host filesystem
```

4. **Inside the Node Debug Shell:**

- **Check Services:**

None

```
systemctl status kubelet
systemctl status crio
```

- **Check Logs:**

None

```
journalctl -u kubelet -f --no-pager
```

- Look for errors like "Failed to post node status" or "unable to contact api-server".
- **Check Resources:**
 - **df -h** (Is `/var/lib/kubelet` or `/` full?)
 - **free -m** (Is memory exhausted?)
 - **ps aux | wc -l** (Are you out of PIDs?)

Resolution & Prevention

- **Resolution:**
 - **Service Crash:** `systemctl restart kubelet`

- **Disk Full:** Identify and clear large files (often old container logs or images). Run `crictl rmi --prune` or `podman image prune`.
- **Network Issue:** Troubleshoot routing, firewalls, or proxy settings between the node and the API server.
- **Unrecoverable:**
 1. Drain the node to safely evict pods: `oc adm drain <node-name> --force --ignore-daemonsets`
 2. Terminate and rebuild the node (via cloud console or `MachineSet` scaling).
- **Prevention:**
 - **Monitor Node Resources:** Have strong alerts on node-level disk, memory, and CPU usage.
 - **Log Rotation:** Ensure all node-level log rotation is configured correctly.
 - **Reserve Resources:** Use `kube-reserved` and `system-reserved` flags to reserve resources for system daemons, preventing workloads from starving the `kubelet`.

Key Alert (Prometheus)

None

```
kube_node_status_condition{condition="Ready", status="false"} > 0
```

Incident Example (PLAT-01)

- **Summary:** `worker-05.example.com` went `NotReady`.
- **Root Cause:** `oc describe node` showed `DiskPressure: True`. An `oc debug node` and `df -h` revealed `/var/lib/kubelet` was 100% full. A pod without log rotation had written 50Gi of logs to its `emptyDir` volume.
- **Resolution:** The node was drained. The debug shell was used to find and delete the large log files. The `kubelet` was restarted, and the node returned to `Ready`. The problematic pod's `Deployment` was updated to remove the `emptyDir` log dump.