Kubernetes Pod Failure Runbook Pack

A senior engineer's incident runbook for every way a Pod refuses to run — Crash loops, image pulls, OOM kills, scheduling, probes, and dead nodes. DevOps AI ToolKit · devopsaitoolkit.com

A Pod can fail in a dozen distinct ways, and the fix for each is different — but the *diagnosis* almost always starts the same way. Kubernetes tells you exactly what is wrong if you read the right fields: the container `State`/`Reason`, the last-terminated `Exit Code`, and the Pod's `Events`. This pack walks the common failure modes top-to-bottom. Read the Pod's own signals first, form a hypothesis, then confirm it before you delete, scale, or restart anything.

> Assumes a vanilla Kubernetes cluster with `kubectl` access to the affected namespace. Examples use namespace `prod`, pod `api-7d9f8c5b4-x2ktp`, node `node-1`, image `registry.example.com/myapp:1.4.2`. All checks here are read-only unless noted — never `delete` a Pod before you have captured its describe output and logs.

1. Triage Order (read the Pod before you touch it)

Work these four commands in order — 90% of Pod failures are identified here without changing anything.

```
NS=prod
POD=api-7d9f8c5b4-x2ktp

# 1) High-level state + which node it landed on
kubectl -n $NS get pod $POD -o wide

# 2) The single most useful command: State/Reason, Exit Code, and Events
kubectl -n $NS describe pod $POD

# 3) Cluster events for this pod, oldest→newest
kubectl -n $NS get events --field-selector involvedObject.name=$POD \
  --sort-by=.lastTimestamp

# 4) Current logs, and the logs from the crashed instance
kubectl -n $NS logs $POD --tail=100
kubectl -n $NS logs $POD --previous --tail=100        # the container that just died
```

- The `STATUS` column (`CrashLoopBackOff`, `ImagePullBackOff`, `Pending`, `Init:0/1`, `OOMKilled`, `CreateContainerConfigError`) tells you which section below to jump to.
- In `describe`, read **Last State → Terminated → Reason / Exit Code** and the **Events** at the bottom — that pair localizes almost everything.
- `--previous` is essential for a crash loop: the *current* container is often too young to have logged the real error; the *previous* one holds the stack trace.
- Multi-container Pods: add `-c <container>` to `logs`. `kubectl -n $NS get pod $POD -o jsonpath='{.spec.containers[*].name}'` lists them.

2. CrashLoopBackOff — the container starts, then dies, repeatedly

`CrashLoopBackOff` is not the error; it is Kubernetes backing off (10s, 20s, 40s… capped at 5m) because the container keeps exiting. The real cause is the **exit code** and what the process logged before it died.

```
# Exit code + reason of the instance that just died
kubectl -n $NS get pod $POD \
  -o jsonpath='{.status.containerStatuses[0].lastState.terminated.exitCode}{"  "}{.status.containerStatuses[0].lastState.terminated.reason}
{"\n"}'

# The actual crash output
kubectl -n $NS logs $POD --previous --tail=200

# How many times and how recently
kubectl -n $NS get pod $POD \
  -o jsonpath='{.status.containerStatuses[0].restartCount}{"\n"}'
```

- **Exit 1 / non-zero app code** → the application itself threw (bad config, missing env var, failed DB connect, unhandled exception). Read `--previous` logs.
- **Exit 137** → SIGKILL (128+9). Usually OOMKilled — jump to §4. Can also be a `kubectl delete`/liveness kill.
- **Exit 143** → SIGTERM (128+15): the process was asked to stop and exited cleanly-ish; often a liveness probe restart (§8) or shutdown that races startup.
- **Exit 0 that still loops** → the entrypoint runs to completion and exits; a long-running service needs a foreground process. Check the `command`/`args` and the image's `ENTRYPOINT`.

```
# Is the entrypoint what you think it is?
kubectl -n $NS get pod $POD -o jsonpath='{.spec.containers[0].command}{"\n"}{.spec.containers[0].args}{"\n"}'
```

3. ImagePullBackOff / ErrImagePull — the image never loaded

These two are the same story at different stages: `ErrImagePull` is the *first* failed pull; `ImagePullBackOff` is Kubernetes backing off after repeated failures. The Events line names the cause.

```
kubectl -n $NS describe pod $POD | grep -A5 -i "events\|failed to pull\|error"

# Exactly which image reference is it trying to pull?
kubectl -n $NS get pod $POD -o jsonpath='{.spec.containers[*].image}{"\n"}'
```

Read the Event message and match it:

- **`not found`** / **`manifest unknown`** → wrong repo, tag, or a typo. Confirm the tag exists in the registry; beware `:latest` pointing at a deleted digest.
- **`unauthorized`** / **`denied`** → registry auth. The Pod needs an `imagePullSecret`, or the node/service account lacks pull rights.
- **`toomanyrequests`** / **Docker Hub `pull rate limit`** → you are being rate-limited. Authenticate pulls or mirror the image.
- **`dial tcp … i/o timeout`** → the node can't reach the registry (network policy, egress firewall, DNS).

```
# Is a pull secret attached, and is it a valid dockerconfig?
kubectl -n $NS get pod $POD -o jsonpath='{.spec.imagePullSecrets[*].name}{"\n"}'
kubectl -n $NS get secret <pull-secret> -o jsonpath='{.type}{"\n"}'   # expect kubernetes.io/dockerconfigjson
kubectl -n $NS get serviceaccount default -o jsonpath='{.imagePullSecrets[*].name}{"\n"}'
```

- A missing/mistyped secret name in `imagePullSecrets` silently produces `unauthorized`. The secret must live in the **same namespace** as the Pod.

---

4. OOMKilled — exit 137 at the container's memory limit

`OOMKilled` means the kernel OOM killer terminated the container because it exceeded its cgroup memory **limit** (or the node ran out of memory). This is per-container, enforced at `resources.limits.memory` — it is *not* the same as a node-level MemoryPressure eviction (§10).

```
kubectl -n $NS get pod $POD \
  -o jsonpath='{.status.containerStatuses[0].lastState.terminated.reason}{" exit "}{.status.containerStatuses[0].lastState.terminated.exitCode}
{"\n"}'
# Reason: OOMKilled   exit 137

# What limit was it killed at?
kubectl -n $NS get pod $POD -o jsonpath='{range .spec.containers[*]}{.name}: req={.resources.requests.memory} lim={.resources.limits.memory}
{"\n"}{end}'

# Actual usage right now (needs metrics-server)
kubectl -n $NS top pod $POD --containers
```

- **Container hit its own limit** → the app needs more memory or has a leak. Raise `limits.memory`, or fix the leak. A limit set equal to a modest request with a spiky workload OOMs under load.
- **Node under memory pressure** → different failure: see the *node* conditions and eviction path in §10. OOMKilled shows in the *container* terminated reason; eviction shows as a Pod `Evicted` status with a node-level message.
- Check `dmesg`/node logs only if the container limit looks generous — a JVM/Go process ignoring `GOMEMLIMIT`/`-Xmx` will exceed the cgroup and get killed regardless of headroom.

---

5. Pending / FailedScheduling — the scheduler can't place it

A `Pending` Pod has no node. The scheduler records exactly why in an Event. Always read that message before assuming "the cluster is full."

```
kubectl -n $NS describe pod $POD | grep -A10 -i "events"
# Look for: 0/5 nodes are available: <reasons>
```

Decode the `FailedScheduling` reasons:

- **`Insufficient cpu`** / **`Insufficient memory`** → no node has room for the Pod's `requests`. Check node allocatable vs requests.
- **`node(s) had untolerated taint`** → the Pod lacks a toleration for a node taint (e.g. a dedicated/GPU pool, or a `NotReady`/`unreachable` taint).
- **`node(s) didn't match Pod's node affinity/selector`** → `nodeSelector`/`affinity` excludes every node.
- **`pod has unbound immediate PersistentVolumeClaims`** → the PVC isn't bound; a `WaitForFirstConsumer` StorageClass or no available PV.

```
# Node capacity and current pressure
kubectl get nodes -o wide
kubectl describe node node-1 | grep -A6 "Allocated resources"
kubectl get nodes -o json | jq -r '.items[] | .metadata.name + " " + (.spec.taints // [] | tostring)'

# PVC bound?
kubectl -n $NS get pvc
kubectl -n $NS describe pvc <claim>   # Events show provisioning failures
```

- If `requests` are far larger than reality, right-size them — oversized requests strand capacity and cause false "cluster full."

---

6. Init Container Failures — stuck before the app even starts

If `STATUS` shows `Init:0/1`, `Init:CrashLoopBackOff`, or `Init:Error`, an init container is failing and the main containers will never start. Init containers run sequentially and must each exit 0.

```
# Which init container, and its state/exit code
kubectl -n $NS get pod $POD -o jsonpath='{range .status.initContainerStatuses[*]}{.name}: {.state}{" exit="}{.lastState.terminated.exitCode}
{"\n"}{end}'

# Logs from the failing init container (name it explicitly)
kubectl -n $NS logs $POD -c wait-for-db --previous
```

- Common init patterns: `wait-for-db`, `run-migrations`, `fetch-config`. A hang here usually means the dependency (DB, config service, another Pod) isn't reachable yet — cross-check §9 connectivity.
- An init container that exits non-zero blocks the whole Pod. Fix the dependency or the init logic; the main container is fine.
- `describe` Events will show `Init:Error` with the failing container name — read its logs, not the app's.

---

7. CreateContainerConfigError — a referenced ConfigMap or Secret is missing

`CreateContainerConfigError` (sometimes `CreateContainerError`) means the kubelet couldn't assemble the container because something it references doesn't exist — almost always a ConfigMap or Secret named in `env`, `envFrom`, or a volume.

```
kubectl -n $NS describe pod $POD | grep -A3 -i "createcontainer\|not found"
# e.g. Error: configmap "app-config" not found

# What does the Pod reference?
kubectl -n $NS get pod $POD -o jsonpath='{range .spec.containers[*].envFrom[*]}{.configMapRef.name}{.secretRef.name}{"\n"}{end}'

# Do those objects actually exist in THIS namespace?
kubectl -n $NS get configmap
kubectl -n $NS get secret
```

- The referenced object must exist in the **same namespace** as the Pod. A ConfigMap in `default` won't satisfy a Pod in `prod`.
- A single missing key (e.g. `secretKeyRef.key` not present in the Secret) triggers the same error — check the exact key, not just the object name.
- Create/fix the ConfigMap/Secret, then the kubelet retries automatically; you don't need to delete the Pod.

---

8. Readiness / Liveness Probe Failures — running but never Ready, or killed on a loop

Probes are a common cause of "the Pod is Running but traffic 503s" (readiness) or "the Pod keeps restarting on its own" (liveness → SIGTERM, exit 143, then a fresh start). The Events name the probe and the HTTP/TCP result.

```
kubectl -n $NS describe pod $POD | grep -A2 -i "readiness\|liveness\|unhealthy"
# Warning  Unhealthy  Readiness probe failed: HTTP probe failed with statuscode: 503

# What are the probes actually configured to hit?
kubectl -n $NS get pod $POD -o jsonpath='{.spec.containers[0].readinessProbe}{"\n"}{.spec.containers[0].livenessProbe}{"\n"}'

# Ready condition per container
kubectl -n $NS get pod $POD -o jsonpath='{range .status.containerStatuses[*]}{.name} ready={.ready}{"\n"}{end}'
```

- **Readiness failing** → Pod stays out of the Service endpoints (no traffic) but is *not* restarted. The app isn't serving the probe path/port yet, or a dependency is down.
- **Liveness failing** → kubelet restarts the container (you'll see `restartCount` climb with exit 143). A too-aggressive `initialDelaySeconds` / `timeoutSeconds` on a slow-starting app causes a *false* liveness kill loop — raise the delay or add a `startupProbe`.
- Confirm the probe target from *inside* the Pod before blaming the app: `kubectl -n $NS exec $POD -- wget -qO- localhost:8080/healthz` (or `curl`).

---

9. Service / DNS / In-Cluster Connectivity — the Pod runs but can't reach its peers

When the Pod is healthy but the app can't talk to a database or another service, the failure is DNS or Service routing, not the Pod. Test from inside the affected Pod.

```
# Does in-cluster DNS resolve? (uses kube-dns/CoreDNS)
kubectl -n $NS exec $POD -- nslookup my-svc.prod.svc.cluster.local
kubectl -n $NS exec $POD -- nslookup kubernetes.default

# Does the Service have healthy endpoints? (empty = no ready backing Pods)
kubectl -n $NS get endpoints my-svc
kubectl -n $NS get endpointslices -l kubernetes.io/service-name=my-svc

# Can the Pod reach the Service IP:port?
kubectl -n $NS exec $POD -- wget -qO- --timeout=3 my-svc:8080/healthz
```

- **Empty `endpoints`** → no Pod matches the Service selector *and* is Ready. Fix the selector or the backing Pods' readiness (§8) — the Service itself is fine.
- **DNS fails for everything** → CoreDNS problem: `kubectl -n kube-system get pods -l k8s-app=kube-dns` and check its logs.
- **DNS resolves but connection refused/times out** → the target isn't listening, or a `NetworkPolicy` is blocking the path. List policies: `kubectl -n $NS get networkpolicy`.

---

10. Node NotReady & Evictions — the failure is the node, not the Pod

When many Pods on one node degrade at once, suspect the node. A `NotReady` node stops reporting; after the eviction timeout its Pods are rescheduled (or stuck `Terminating` if the node is truly gone).

```
kubectl get nodes
kubectl describe node node-1 | grep -A8 "Conditions"
# Look for MemoryPressure / DiskPressure / PIDPressure = True, or Ready = Unknown

# Which pods are evicted, and why?
kubectl -n $NS get pods --field-selector status.phase=Failed
kubectl -n $NS get pod $POD -o jsonpath='{.status.reason}: {.status.message}{"\n"}'
# Evicted: The node was low on resource: memory. ...
```

- **`Ready: Unknown`** / **`NotReady`** → kubelet stopped heartbeating (node down, network, or kubelet crash). Kubernetes taints it `node.kubernetes.io/unreachable` and evicts after the toleration window (default 300s).
- **`MemoryPressure: True`** → **Pod `Evicted`** → this is the *node-level* eviction path, distinct from a per-container OOMKill (§4). The kubelet proactively evicts Pods to reclaim memory; the killed container's own limit may be fine.
- **`DiskPressure: True`** → image/log/ephemeral-storage exhaustion; the kubelet also garbage-collects images and may evict Pods exceeding `ephemeral-storage`.
- Evicted Pod objects linger for postmortem — clean them up only after capturing the reason: `kubectl -n $NS delete pod --field-selector status.phase=Failed`.

---

11. Pod Failure Decision Tree (STATUS + exit code → next move)

```
kubectl get pod → read STATUS, then describe → Exit Code
 ├─ ImagePullBackOff / ErrImagePull ─→ §3  image ref? auth/imagePullSecret? rate limit? node→registry net?
 ├─ Pending (never scheduled) ────────→ §5  read FailedScheduling: resources / taint / affinity / PVC
 ├─ Init:Error / Init:CrashLoop ──────→ §6  logs -c <init> --previous; fix the dependency it waits on
 ├─ CreateContainerConfigError ───────→ §7  missing ConfigMap/Secret/key in THIS namespace
 ├─ Running but not Ready / 503 ──────→ §8  readiness probe path/port; endpoints empty? (§9)
 ├─ Running but restarting itself ────→ §8  liveness kill (exit 143); raise delay / add startupProbe
 └─ CrashLoopBackOff / Terminated ─→ read Exit Code:
      ├─ 137 → OOMKilled? (§4 container limit)  or node MemoryPressure eviction (§10)
      ├─ 143 → SIGTERM: liveness restart or shutdown race (§2/§8)
      ├─ 0   → entrypoint exits; needs a foreground long-running process (§2)
      └─ 1/other → app error: logs --previous (§2); config / env / dependency (§9)

 Many pods on ONE node failing at once ─→ §10  describe node: NotReady / MemoryPressure / DiskPressure
```

---

12. Copy/Paste One-Shot Triage

```
NS=prod; POD=api-7d9f8c5b4-x2ktp
echo "== overview =="; kubectl -n $NS get pod $POD -o wide
echo "== state/exit =="; kubectl -n $NS get pod $POD -o jsonpath='{range .status.containerStatuses[*]}{.name} ready={.ready} restarts=
{.restartCount} last={.lastState.terminated.reason}/{.lastState.terminated.exitCode}{"\n"}{end}'
echo "== requests/limits =="; kubectl -n $NS get pod $POD -o jsonpath='{range .spec.containers[*]}{.name} cpu=
{.resources.requests.cpu}/{.resources.limits.cpu} mem={.resources.requests.memory}/{.resources.limits.memory}{"\n"}{end}'
echo "== events =="; kubectl -n $NS get events --field-selector involvedObject.name=$POD --sort-by=.lastTimestamp | tail -12
echo "== prev logs =="; kubectl -n $NS logs $POD --previous --tail=40 2>/dev/null || kubectl -n $NS logs $POD --tail=40
echo "== node =="; NODE=$(kubectl -n $NS get pod $POD -o jsonpath='{.spec.nodeName}'); kubectl describe node "$NODE" 2>/dev/null | grep -A6
"Conditions" | grep -E "Pressure|Ready"
```

---

13. Incident Notes Template

```
INCIDENT: Kubernetes Pod Failure
Started (UTC):        ____   Detected by: ____   Sev: ____
Namespace / Pod:      ____ / ____
Workload:             (Deployment/StatefulSet/DaemonSet) ____   Node: ____
STATUS observed:      [ ] CrashLoopBackOff [ ] ImagePull* [ ] OOMKilled [ ] Pending
                      [ ] Init:Error [ ] CreateContainerConfigError [ ] Probe fail [ ] Evicted
Exit code / reason:   ____ / ____   Restart count: ____
Key event message:    ____
Evidence:             [ ] logs --previous captured  [ ] describe saved  [ ] top pod
                      mem req/lim: ____   scheduler reason: ____
Dependency checked:   [ ] DNS  [ ] endpoints  [ ] ConfigMap/Secret  [ ] PVC bound
Node condition:       [ ] Ready [ ] MemoryPressure [ ] DiskPressure [ ] NotReady/unreachable
Action taken:         (config fix / limit bump / secret created / node cordoned + who + when)
Result:               [ ] recovered @ ____   [ ] escalated to ____
Root cause:           ____
Follow-ups:           (limits/requests, probes, image pinning, alerts, capacity) ____
```
