Kubernetes CrashLoopBackOff Runbook

A senior engineer's incident runbook for a pod that keeps restarting (STATUS `CrashLoopBackOff`). DevOps AI ToolKit · devopsaitoolkit.com

`CrashLoopBackOff` is a *symptom, not a cause*. It means the container started, exited, and the kubelet is now backing off (10s, 20s, 40s... capped at 5m) before trying again. The real question is always **why did the process exit** — and the exit code plus the container's *Last State* almost always tell you. Work read-only first: read the pod state, read the exit code, read the logs of the crashed instance. Don't restart or redeploy until the exit code has told you whether this is your app, your config, or the platform (OOM).

> Examples use namespace `prod` and a pod name of `myapp-7d9f8c6b5-abcde`. Substitute your own. Run every read-only check before you delete a pod or roll back — the crashed container's logs and exit code disappear once you lose the previous instance.

1. Read the Pod State First

```
kubectl get pod myapp-7d9f8c6b5-abcde -n prod -o wide
kubectl get pods -n prod -l app=myapp        # is it one replica or all of them?
```

- `RESTARTS` with a rising count and a `(30s ago)` suffix confirms an active crash loop, not a one-off.
- One replica crashing while others are `Running` points at that node or a bad pod; *all* replicas crashing points at config, image, or a shared dependency.
- `-o wide` gives you the `NODE` — you'll want it if this turns out to be node-local (disk, image pull, kubelet).

2. Describe the Pod — Events and Last State

```
kubectl describe pod myapp-7d9f8c6b5-abcde -n prod
```

Read two regions of the output:

- **`Last State: Terminated`** — this is the crashed instance. Note **`Reason`** (`Error`, `OOMKilled`, `Completed`) and **`Exit Code`**. This single line usually decides your entire investigation.
- **`Events`** (bottom) — `Back-off restarting failed container`, `Liveness probe failed`, `Startup probe failed`, or `Created/Started` loops. A liveness message here changes the diagnosis (see §6).

Exit-code cheat sheet: **0** clean exit (a non-daemon that ran and stopped — maybe it shouldn't be a Deployment). **1** generic application error. **137** = 128 + 9 (SIGKILL) — usually **OOMKilled**, sometimes a failed liveness kill. **143** = 128 + 15 (SIGTERM) — graceful shutdown signal, often the probe or the node evicting it.

3. Read the Logs — Current and Previous

```
# Current instance (may be empty if it dies before logging)
kubectl logs myapp-7d9f8c6b5-abcde -n prod

# The instance that ACTUALLY crashed — this is the important one
kubectl logs myapp-7d9f8c6b5-abcde -n prod --previous

# Multi-container pod: name the container explicitly
kubectl logs myapp-7d9f8c6b5-abcde -n prod -c myapp --previous
```

- `--previous` (`-p`) reads the last terminated container's logs. In a fast crash loop the *current* logs are empty; the answer is in `--previous`.
- A stack trace, `panic:`, `FATAL`, or a config-parse error here is your root cause. Read the **last 20 lines** before the exit.
- Empty previous logs + exit 137 → the process never got to log; suspect OOM (§7) or a missing entrypoint (§4).

4. Verify the Entrypoint, Command, and Args

```
kubectl get pod myapp-7d9f8c6b5-abcde -n prod \
  -o jsonpath='{range .spec.containers[*]}{.name}: {.command} {.args}{"\n"}{end}'
```

- An overridden `command`/`args` that points at a binary or flag that doesn't exist in the image produces an instant exit with `Error` and often no logs (`exec: "serve": executable file not found`).
- If `command` is empty, the image's own `ENTRYPOINT`/`CMD` runs — compare against what the Dockerfile actually ships.
- A shell-form entrypoint that exits immediately (script runs to completion) yields **exit 0** and a crash loop — the container isn't a long-running process.

5. Check Env Vars, ConfigMap and Secret References

```
kubectl get pod myapp-7d9f8c6b5-abcde -n prod \
  -o jsonpath='{range .spec.containers[*].env[*]}{.name}={.value}{"\n"}{end}'

# Referenced ConfigMaps / Secrets must exist in the SAME namespace
kubectl get configmap,secret -n prod
kubectl describe pod myapp-7d9f8c6b5-abcde -n prod | grep -Ei "configmap|secret|not found"
```

- A missing ConfigMap/Secret named in `envFrom` or a volume blocks the pod at `CreateContainerConfigError` — you'll see it in Events, not as a crash. A missing *key* referenced by `configMapKeyRef` / `secretKeyRef` does the same.
- If the container *does* start but exits 1 with `config: required key DATABASE_URL missing`, the reference resolved but the value is wrong or empty. Print the env and confirm.

---

6. Rule Out Probes Killing It Too Early

```
kubectl get pod myapp-7d9f8c6b5-abcde -n prod \
  -o jsonpath='{range .spec.containers[*]}{.name}:{"\n"}  liveness={.livenessProbe}{"\n"}  readiness={.readinessProbe}{"\n"}  startup=
{.startupProbe}{"\n"}{end}'
kubectl describe pod myapp-7d9f8c6b5-abcde -n prod | grep -Ei "liveness|readiness|startup|probe failed|unhealthy"
```

- **`Liveness probe failed` in Events + exit 137/143** means the *app is fine but slow to start* — the liveness probe fired during boot and the kubelet killed it. This is a classic false CrashLoopBackOff.
- Fix is usually a **`startupProbe`** or a larger `initialDelaySeconds` / `failureThreshold`, not an app change.
- A readiness probe never crash-loops a pod (it only removes it from Service endpoints) — if you see restarts, it's liveness or the process itself.

---

7. Resource Limits — OOMKill vs App Crash

```
kubectl get pod myapp-7d9f8c6b5-abcde -n prod \
  -o jsonpath='{range .spec.containers[*]}{.name}: req={.resources.requests} lim={.resources.limits}{"\n"}{end}'
kubectl describe pod myapp-7d9f8c6b5-abcde -n prod | grep -A2 "Last State"
kubectl top pod myapp-7d9f8c6b5-abcde -n prod   # needs metrics-server
```

- **`Reason: OOMKilled`, `Exit Code: 137`** → the container exceeded its memory **limit** and the kernel OOM-killer reaped it. Raise the limit *or* fix the leak — don't just bump the number blindly.
- Exit 137 **without** `OOMKilled` → a SIGKILL from elsewhere (failed liveness, node pressure eviction). Check Events to disambiguate.
- Exit 1/2 with an application stack trace → this is **not** a resource problem; go back to §3.

---

8. Image / Config Mismatch and Unready Dependencies

```
# What image tag is actually running vs what you think you deployed?
kubectl get pod myapp-7d9f8c6b5-abcde -n prod \
  -o jsonpath='{range .spec.containers[*]}{.name}: {.image}{"\n"}{end}'

# Is the DB / upstream the app needs actually reachable?
kubectl get endpoints -n prod
kubectl run netcheck --rm -it --image=busybox:1.36 -n prod --restart=Never \
  -- sh -c 'nc -zv postgres.prod.svc.cluster.local 5432'
```

- A new image tag (e.g. `1.4.2` → `1.5.0`) that crash-loops while the old one was fine is a **bad build or a config the new version now requires**. Compare with the previous ReleaseSet and check §5.
- App exits 1 early with `connection refused` / `could not connect to database` → the crash loop is a **dependency-not-ready** problem. The pod is honest; fix the dependency or add a proper init/startup wait. `kubectl get endpoints` with an empty subset for the dependency Service confirms it.

---

9. Decision Tree — Keyed on Exit Code + Last State

```
CrashLoopBackOff on <pod>?  → kubectl describe → read Exit Code + Reason
 ├─ Reason: OOMKilled / Exit 137 ──── memory limit hit → raise limit or fix leak (§7)
 ├─ Exit 137, NOT OOMKilled ────────── SIGKILL: liveness probe (§6) or node eviction — check Events
 ├─ Exit 143 (SIGTERM) ─────────────── graceful kill: probe timing (§6) or eviction, not an app bug
 ├─ Exit 0 (Completed) ─────────────── process ran and finished → not a daemon; wrong workload type / entrypoint (§4)
 ├─ Exit 1/2 + stack trace in --previous logs
 │    ├─ "config/env key missing" ──── ConfigMap/Secret/env (§5)
 │    ├─ "connection refused" ──────── dependency not ready (§8)
 │    └─ app panic / uncaught error ─ code/build bug → roll back image (§8)
 └─ No logs at all + instant exit ──── bad command/args or missing binary (§4) or CreateContainerConfigError (§5)
```

---

10. Copy/Paste One-Shot Triage

```
NS=prod; POD=myapp-7d9f8c6b5-abcde
echo "== state ==";      kubectl get pod $POD -n $NS -o wide
echo "== last state =="; kubectl describe pod $POD -n $NS | grep -A4 "Last State"
echo "== events ==";     kubectl describe pod $POD -n $NS | sed -n '/Events:/,$p' | tail -15
echo "== prev logs ==";  kubectl logs $POD -n $NS --previous --tail=25 2>&1
echo "== limits ==";     kubectl get pod $POD -n $NS -o jsonpath='{range .spec.containers[*]}{.name}: lim={.resources.limits}{"\n"}{end}'
echo "== image ==";      kubectl get pod $POD -n $NS -o jsonpath='{range .spec.containers[*]}{.name}: {.image}{"\n"}{end}'
```

11. Incident Notes Template

```
INCIDENT: Kubernetes CrashLoopBackOff
Started (UTC):        ____   Detected by: ____   Sev: ____
Pod / namespace:      ____ / ____   Replicas affected: __ of __
Node:                 ____   Image + tag: ____
Exit code:            ____   Last State reason: [ ] Error [ ] OOMKilled [ ] Completed
Probe involvement:    [ ] liveness failed  [ ] startup  [ ] none
Log evidence (--previous last lines): ____
Category:             [ ] OOM/limit  [ ] config/env  [ ] bad command/args  [ ] dependency not ready
                      [ ] probe timing  [ ] bad build/image
Action taken:         (limit raised / config fixed / probe tuned / image rolled back + who + when)
Result:               [ ] recovered @ ____   [ ] escalated to ____
Root cause:           ____
Follow-ups:           (limits, probe defaults, readiness gates, alerts) ____
```