Kubernetes OOMKilled Runbook

A senior engineer's incident runbook for containers killed with reason OOMKilled (exit code 137). DevOps AI ToolKit · devopsaitoolkit.com

OOMKilled means the Linux kernel's cgroup OOM killer sent SIGKILL to a process inside your container because it tried to use more memory than it was allowed. Kubernetes surfaces this as `Reason: OOMKilled` and `Exit Code: 137` (that's `128 + 9`, where `9` is SIGKILL). There are two flavors, and they are *not* the same incident: a container hitting **its own memory limit** (per-container cgroup OOM), or the **node running out of memory** (which normally triggers *evictions*, `Reason: Evicted`, not OOMKills). Your first job is to tell those apart, then decide whether the fix is a bigger limit, a leaner app, or a less crowded node.

> Read-only checks first. `kubectl top` requires metrics-server; if it isn't installed, fall back to Prometheus (§5). Examples use namespace `prod`, pod `api-7d9f8`, node `node-1` — substitute your own.

1. Confirm It Was Actually OOMKilled

```
# The definitive signal: Last State terminated with reason OOMKilled
kubectl -n prod describe pod api-7d9f8 | grep -A6 "Last State"

# Same thing, machine-readable, per container
kubectl -n prod get pod api-7d9f8 \
  -o jsonpath='{range .status.containerStatuses[*]}{.name}{"  lastState="}{.lastState}{"\n"}{end}'

# Restart count climbing = repeated kills
kubectl -n prod get pod api-7d9f8 -o wide
```

- Look for `Last State: Terminated`, `Reason: OOMKilled`, `Exit Code: 137`. That is a per-container cgroup kill.
- Exit `137` alone is *SIGKILL* — usually OOM, but a liveness-probe failure or manual `kill -9` also produces 137. The `Reason: OOMKilled` string is what confirms memory.
- A high `RESTARTS` count with OOMKilled in the *last* state means it's crash-looping on memory, not a one-off spike.

2. Container Limit vs. Actual Usage

```
# What is this container actually allowed to use?
kubectl -n prod get pod api-7d9f8 -o jsonpath='{range .spec.containers[*]}{.name}{"  req="}{.resources.requests.memory}{"  lim="}
{.resources.limits.memory}{"\n"}{end}'

# What is it using right now? (needs metrics-server)
kubectl -n prod top pod api-7d9f8 --containers
```

- The OOM killer fires at `resources.limits.memory`, not at `requests`. If there is **no limit**, the container is only bounded by the node — so an OOMKill with no limit set means you were killed by node pressure, go to §4.
- `top` shows a point-in-time value and won't catch the spike that killed you. A container sitting at 90% of its limit is one request away from a kill.

3. Is It the LIMIT or the NODE?

This is the fork in the road. Same symptom, different fixes.

```
# Container hit its own limit → OOMKilled on the pod (see §1)
kubectl -n prod get pod api-7d9f8 \
  -o jsonpath='{.status.containerStatuses[0].lastState.terminated.reason}{"\n"}'

# Node ran out → look for Evicted pods and the kubelet's verdict
kubectl -n prod get pods --field-selector=status.phase=Failed
kubectl get events -A --field-selector reason=Evicted --sort-by=.lastTimestamp | tail
kubectl get events -A | grep -Ei "oom|evict|memorypressure" | tail
```

- `Reason: OOMKilled` on the container → the *cgroup* limit was exceeded. Fix lives with the workload (§6).
- `Reason: Evicted`, message "The node was low on resource: memory" → the *node* was under pressure and the kubelet evicted pods to reclaim memory. That's a node/scheduling problem (§4), even though the underlying trigger is also memory.
- You can have both: node pressure makes the kernel more aggressive, and a leaky pod can be the one that pushes the node over.

4. Node Memory Pressure

```
# MemoryPressure condition + how much is actually committed
kubectl describe node node-1 | grep -A5 "Conditions"
kubectl describe node node-1 | grep -A8 "Allocated resources"

# Allocatable vs live usage
kubectl top node node-1
```

- `MemoryPressure: True` → the kubelet is actively reclaiming and evicting; expect `Evicted` pods.
- Compare **Allocated** memory *requests* against **Allocatable**. If requests are near 100%, the node is oversubscribed by scheduling — even before real usage spikes.
- Remember the gap: pods without limits (Burstable/BestEffort) can burst past their requests and overrun the node, taking well-behaved neighbors down with them.

---

5. Reading Actual Usage Over Time

Point-in-time `top` lies about spikes. Trend it.

```
# Quick sampling loop if you have no dashboard
for i in $(seq 1 10); do kubectl -n prod top pod api-7d9f8 --containers --no-headers; sleep 30; done
```

```
# The number the OOM killer actually watches:
container_memory_working_set_bytes{namespace="prod", pod=~"api-7d9f8.*"}

# Overlay against the enforced limit to see headroom burn down:
container_memory_working_set_bytes{namespace="prod", pod=~"api-7d9f8.*"}
  / on(pod) kube_pod_container_resource_limits{namespace="prod", resource="memory"}
```

- `container_memory_working_set_bytes` — not RSS, not cache — is what the kernel compares to the cgroup limit. Alert on it crossing ~85% of limit.
- A **sawtooth** that resets on restart = a leak (§7). A **flat line that rises with traffic** and plateaus = genuine need; the limit is just too low (§6).

---

6. Right-Size Requests and Limits

```
# Patch a Deployment's resources (adjust to your measured steady-state + headroom)
kubectl -n prod set resources deployment/api \
  --requests=memory=512Mi --limits=memory=768Mi
```

- Set **requests** to observed steady-state usage — that's what the scheduler packs against.
- Set **limits** to steady-state plus real headroom for spikes (GC, bursts, warm caches). Too tight and you OOMKill on normal variance.
- `limit == request` puts the pod in **Guaranteed** QoS (good for protection, §8) but leaves *zero* burst room — a transient spike kills you instantly. Size it deliberately, don't do it by accident.
- The classic trap: a runtime that sizes its heap to the *host*, not the cgroup. A JVM with no `-Xmx`, or Node.js with no `--max-old-space-size`, sees node RAM, grows its heap past the container limit, and gets OOMKilled while "well under" what it thinks is available. Align the runtime to the limit (§7).

---

7. Application Leak vs. Genuine Need

```
# Is the runtime's heap ceiling aligned to the container limit?
kubectl -n prod exec api-7d9f8 -- printenv | grep -Ei "JAVA_OPTS|GOMEMLIMIT|NODE_OPTIONS|_XMX"
```

- **Leak signature:** working set climbs monotonically across hours, never plateaus, resets only on restart. No limit increase saves a leak — it just delays the next kill. Fix the app (unbounded cache, retained references, connection pools).
- **Genuine need:** usage tracks load and plateaus. Give it a bigger, correctly measured limit (§6).
- Align the runtime to the cgroup so it manages memory *before* the kernel does:
  - JVM: `-Xmx` (or `-XX:MaxRAMPercentage=75`) well under the limit.
  - Go: `GOMEMLIMIT=700MiB` (a soft limit that makes the GC work harder near the ceiling).
  - Node.js: `--max-old-space-size=640` (in MB) under the limit.
- Leaving ~20-25% between the runtime ceiling and the cgroup limit lets the app GC or shed load instead of getting SIGKILLed mid-request.

---

8. QoS Class and Eviction Order

```
kubectl -n prod get pod api-7d9f8 -o jsonpath='{.status.qosClass}{"\n"}'
```

- **Guaranteed** (requests == limits on every container): last to be evicted under node pressure; strongest protection.
- **Burstable** (requests set, limits higher or unset): evicted after BestEffort, and among Burstable pods those *most over their requests* go first.
- **BestEffort** (no requests or limits): first out the door when the node reclaims memory.
- Two takeaways: node pressure evicts by QoS, but a **per-container OOMKill (§1) ignores QoS entirely** — it fires the moment *that* cgroup exceeds *its* limit, no matter how healthy the node is. Protect critical workloads by giving them limits *and* by making them Guaranteed.

---

9. Decision Tree: Which Fix?

```
Container shows Reason: OOMKilled (exit 137)?
 ├─ Working set flat/rises-with-load, plateaus near limit? ── LIMIT TOO LOW
 │     → raise limits (§6), align runtime heap to it (§7)
 ├─ Working set climbs forever, resets only on restart?    ── APP LEAK
 │     → fix the app; bigger limit only buys time (§7)
 └─ No limit set, or Reason: Evicted + MemoryPressure?     ── NODE OVERSUBSCRIBED
       → set requests/limits so the scheduler packs honestly (§6),
         add capacity or reschedule, protect critical pods via QoS (§8)
```

---

10. Copy/Paste One-Shot Triage

```
NS=prod; POD=api-7d9f8; NODE=node-1
echo "== last state ==";   kubectl -n $NS get pod $POD -o jsonpath='{range .status.containerStatuses[*]}{.name}{" reason="}
{.lastState.terminated.reason}{" exit="}{.lastState.terminated.exitCode}{" restarts="}{.restartCount}{"\n"}{end}'
echo "== limits ==";       kubectl -n $NS get pod $POD -o jsonpath='{range .spec.containers[*]}{.name}{" req="}{.resources.requests.memory}{"
lim="}{.resources.limits.memory}{"\n"}{end}'
echo "== usage ==";        kubectl -n $NS top pod $POD --containers 2>/dev/null || echo "metrics-server missing"
echo "== qos ==";          kubectl -n $NS get pod $POD -o jsonpath='{.status.qosClass}{"\n"}'
echo "== node pressure ==";kubectl describe node $NODE | grep -A5 Conditions | grep -i memory
echo "== evictions ==";    kubectl get events -A --field-selector reason=Evicted --sort-by=.lastTimestamp | tail -5
```

---

11. Incident Notes Template

```
INCIDENT: Kubernetes OOMKilled (exit 137)
Started (UTC):          ____    Detected by: ____    Sev: ____
Impact:                 (which service / % of pods / requests failing)
Pod/Container:          ns=____  pod=____  container=____
Confirmed signal:       [ ] Reason: OOMKilled  exit 137  restarts: ____
LIMIT or NODE:          [ ] container hit its limit   [ ] node MemoryPressure / Evicted
Limit vs usage:         limit=____  request=____  peak working_set=____
QoS class:              [ ] Guaranteed [ ] Burstable [ ] BestEffort
Usage shape:            [ ] plateaus w/ load (right-size)  [ ] climbs forever (leak)
Runtime heap aligned:   [ ] -Xmx/GOMEMLIMIT/--max-old-space-size vs limit: ____
Action taken:           (limit change / app fix / reschedule + who + when)
Result:                 [ ] recovered @ ____   [ ] escalated to ____
Root cause:             ____
Follow-ups:             (limits, HPA/VPA, node capacity, memory alerts) ____
```
