

**Now lets go ahead with writing article,**

please include:

1. Introduction to topic like most commonly encountered error. -- Make it sound like i faced it recently and this article shows how i troubleshooted and resolved.
2. List out following commands from attached file. Make sure you explain each command in one line. present this as table.
3. Explain process of resolving temporarily like increase replicas, how to use auto-scaling
4. Most important is going through the output and understanding the root cause and if required go for code modification.

Make it informative, include all Linkedin emojis (wherever applicable)

Here’s a ready-to-post article draft that narrates a recent incident, lists every troubleshooting command from the attached file in a clear table with one-line explanations, shows quick mitigations (replica scale-up and HPA), and emphasizes reading outputs to find and fix the root cause. 🚀

**Introduction**

Last week, a production-facing workload started flapping with intermittent 5xx errors and frequent restarts — classic symptoms that pointed to memory pressure and OOMKilled events in Kubernetes. After a rapid triage, a structured runbook helped confirm the OutOfMemory pattern, stabilize traffic via replicas and limits, and finally trace the leakage pattern back to faulty caching in the app code. This post walks through the exact commands used, the quick wins to keep the service alive, and how to reason through outputs to reach a code-level fix. 💡[[1]](#fn1)

**Troubleshooting commands**

Below are the exact commands used during incident response, with one-line explanations for each.

|  |  |
| --- | --- |
| Command | What it does |
| kubectl get events --field-selector reason=Killing,type=Warning | Lists recent kill/OOM warnings to confirm OOMKilled patterns quickly. [[1]](#fn1) |
| kubectl describe pod <pod> | Shows last-state, restart reasons, exit codes, and probe results for the container. [[1]](#fn1) |
| kubectl describe nodes | Prints node conditions (e.g., MemoryPressure) indicating cluster-level stress. [[1]](#fn1) |
| kubectl top pod --sort-by=memory | Ranks pods by current memory use to spot the biggest consumers. [[1]](#fn1) |
| kubectl get pod <pod> -o custom-columns="POD:.metadata.name,REQ:.spec.containers[].resources.requests.memory,LIM:.spec.containers[].resources.limits.memory" | Compares configured requests/limits vs observed usage to detect under-provisioning. [[1]](#fn1) |
| kubectl get pod <pod> -o jsonpath='{.status.qosClass}'; echo | Reveals QoS class (Guaranteed/Burstable/BestEffort) impacting eviction priority. [[1]](#fn1) |
| kubectl exec <pod> -- cat /sys/fs/cgroup/memory/memory.limit\_in\_bytes | Reads the container’s memory limit from cgroups to validate enforcement. [[1]](#fn1) |
| kubectl exec <pod> -- cat /sys/fs/cgroup/memory/memory.usage\_in\_bytes | Reads the container’s actual memory usage from cgroups for precise measurement. [[1]](#fn1) |
| kubectl scale deploy/memory-leak --replicas=4 | Temporarily spreads load by increasing replicas to reduce per-pod pressure. [[1]](#fn1) |
| kubectl set resources deploy/memory-leak --limits=memory=512Mi --requests=memory=256Mi | Raises memory limits/requests to align capacity with observed needs. [[1]](#fn1) |
| kubectl rollout restart deploy/memory-leak | Restarts pods to immediately reclaim leaked memory and apply new resources. [[1]](#fn1) |
| kubectl get hpa | Lists HPAs to verify autoscaling policies exist in the namespace. [[1]](#fn1) |
| kubectl describe hpa memory-leak-hpa | Shows HPA target metrics, events, and current/desired replicas for scaling health. [[1]](#fn1) |
| kubectl get hpa memory-leak-hpa -o yaml | Prints HPA status (desiredReplicas, currentReplicas, conditions) to see scaling decisions. [[1]](#fn1) |

**Temporary stabilization**

When OOMs hit, the goal is to stop the bleeding while investigating.

* Scale out to dilute memory pressure per pod:
  + kubectl scale deploy/memory-leak --replicas=4[[1]](#fn1)
* Lift memory headroom (paired with a restart to reclaim leaked RAM):
  + kubectl set resources deploy/memory-leak --limits=memory=512Mi --requests=memory=256Mi[[1]](#fn1)
  + kubectl rollout restart deploy/memory-leak[[1]](#fn1)
* Confirm improvement by rechecking:
  + kubectl top pod --sort-by=memory and kubectl get events for fewer OOMKilled entries.[[1]](#fn1)

Pro tip: ensure memory requests are set, because HPA’s Resource-based autoscaling uses utilization = usage/request — missing requests break memory-based scaling logic. ✅[[1]](#fn1)

**Autoscaling for resilience**

Autoscaling helps handle bursts while pursuing a permanent fix.

* Verify HPA exists and is healthy:
  + kubectl get hpa and kubectl describe hpa memory-leak-hpa[[1]](#fn1)
* Watch HPA decisions:
  + kubectl get hpa memory-leak-hpa -o yaml | sed -n '/status:/,$p' to see desired vs current replicas, and conditions like AbleToScale/ScalingActive. [[1]](#fn1)
* If missing, add an autoscaling/v2 HPA on memory utilization (remember to define memory requests in the Deployment). Then watch it with kubectl get hpa -w to confirm scale-out when utilization crosses thresholds. 🔁[[1]](#fn1)

**Reading outputs to find the root cause**

The difference between “stabilized” and “fixed” is careful interpretation of outputs:

* Events + describe pod: OOMKilled with ExitCode 137 confirms the kernel killed the container, often due to limit breaches or leaks. Next, correlate timestamps with traffic or batch jobs to spot patterns. 🕵️[[1]](#fn1)
* top + cgroup files: If usage trends toward limit irrespective of traffic, the app likely leaks or retains data excessively (cache, buffers, unbounded queues). If usage maps to traffic spikes, right-size or scale-out may be sufficient. 📈[[1]](#fn1)
* Requests/limits + QoS: Burstable pods with low requests are evicted earlier; elevating requests to realistic baselines improves stability under node pressure. 🧭[[1]](#fn1)
* HPA status: If desiredReplicas stays flat while utilization is high, either metrics aren’t available, requests are missing, or targets are misconfigured. Fix those before expecting autoscaling to save the day. ⚙️[[1]](#fn1)

**When to modify code**

After stabilizing, confirm whether the application actually leaks or simply needs more memory for expected workloads:

* Persistent upward memory slope regardless of traffic → audit caches (TTL/eviction), object retention, large response buffers, image/byte array accumulation, or missing close/free calls. Implement bounded caches and streaming where possible. 🧠[[1]](#fn1)
* Spiky but bounded memory matching traffic → keep HPA and raise limits/requests to realistic values; also consider Vertical Pod Autoscaler or periodic heap profiling in non-prod. 📊[[1]](#fn1)
* Add observability hooks: expose process\_resident\_memory\_bytes and a custom gauge for in-app allocations to correlate code paths with memory growth. This shortens MTTR on future incidents and turns guesswork into data. 🔬[[1]](#fn1)

**Closing thoughts**

This incident reinforced a practical playbook: confirm OOMs fast, stabilize with replicas and resource headroom, validate autoscaling, and let evidence from events, cgroups, and HPA status guide a code-level fix. Sharing the exact commands and reasoning helps future “it’s happening again” moments become routine, not panic. If this helped, drop a comment or share your war story — someone’s 3 a.m. self will thank you. 🙌[[1]](#fn1)