PURETENSOR
INTELLIGENCE DISTILLED

Surviving the Loss of Six of Nine Nodes

What actually blocks stateful recovery in Kubernetes

Paper: PT-TN-2026-008 v1.0

Author: PureTensor Ltd

Date: 16 March 2026

Status: Published

Abstract

We powered down five of nine nodes in a K3s cluster, removing the entire Ceph storage backend and the GPU compute node, to learn what would actually block stateful recovery. The answer is simple and slightly embarrassing: stateless failover worked perfectly. Every stateless workload (roughly 35 pods across 44 deployments) rescheduled to the four surviving nodes without intervention. The blockers were all stateful, and both of them are Kubernetes behaving exactly as specified.

The first blocker is StatefulSet at-most-one semantics. A pod on an unreachable node stays in Terminating indefinitely because the kubelet that would confirm deletion is gone; the controller will not create a replacement while the old pod exists. The second is RBD VolumeAttachments stuck on dead nodes: the attachment object persists, so a replacement pod scheduled elsewhere fails with Multi-Attach. Clearing either requires force-deletion or finalizer removal, operations that trade the at-most-one guarantee for liveness.

We built a CronJob to automate the mechanical case. It runs every two minutes, waits for a node to be NotReady for ten minutes, then force-deletes Terminating pods and clears stale VolumeAttachments. A live test against the five-node-down state cleared 15 Terminating pods and roughly 12 VolumeAttachments in 16 seconds; both StatefulSets immediately started fresh pods.

The automation would have done nothing for the most instructive failure we found. A zombie Ceph client session, with no kernel-mapped devices, held stale watchers on 10 RBD images and blocked 27 VolumeAttachments for 19 hours. Every node was Ready and Ceph reported HEALTH_OK. One blocklist command released all 10 watchers instantly. Automation and diagnosis are different jobs.

A separate pass on scheduling constraints revealed that soft affinity is a hint, not placement. On a heterogeneous fleet the scheduler's LeastRequestedPriority scoring reliably beats a preference weight; we applied 39 patches converting to required affinity before misplaced pods dropped from 15–24 to zero. Those patches were runtime-only and revert on manifest reapply, an unresolved persistence gap.

1. The induced failure

The cluster under test ran K3s across nine nodes: one control-plane compute node (also the K3s server), one GPU compute node, three small monitoring nodes (one of them an arm64 single-board machine), and four storage nodes carrying the Ceph OSDs and all three Ceph monitors. Persistent storage was Ceph RBD for databases and application state, CephFS for bulk paths, plus a small number of workloads still pinned to local-path volumes. At the time of the audit the cluster held 44 deployments and 3 StatefulSets, later counted as 105 pods across the nine nodes.

The test was blunt: power down the GPU compute node and all four storage nodes. That removed Ceph entirely and left four nodes Ready (the control-plane compute node and the three monitoring nodes) against five NotReady. We measured what survived, what rescheduled, and what deadlocked.

Immediately before the test we hardened three critical services. The ingress controller scaled from one replica to three with topology spread constraints, landing on two storage nodes and the GPU compute node. A DNS and ad-filtering service moved to a hostNetwork pod on the GPU compute node, hard-excluding the control-plane node because port 53 was already taken there. The Git and registry service migrated off Docker into the cluster (1.2 GB of data) with a LoadBalancer service and svclb on all nine nodes. That migration surfaced an unrelated failure worth mentioning: the Git service started but was silently blocked by AppArmor on the host kernel and produced no logs at all until we added an unconfined annotation.

2. Stateless failover passed cleanly

Every stateless workload rescheduled to the four surviving nodes without intervention. The list is long enough to be boring, which is the point: report bot, dictation, voice-memo, publisher API and researcher and worker, dashboards, task and diary services, object store, webhook receiver, uptime monitors, credential vault, translator, alertmanager, log store, metrics stack, ingress replicas, two of three CoreDNS replicas, metrics-server, local-path provisioner. Roughly 35 pods running, all on Ready nodes.

Ceph-dependent pods failed as designed. A database StatefulSet, a cache StatefulSet, a vector store, a search engine, a file-sync service, a long-term Grafana and Prometheus pair, the DNS filter, an automation engine, a document archive, and a voice service all backed by RBD or CephFS sat waiting for storage that was not there. We do not count these as defects.

Metric	Value
Nodes Ready	4
Nodes NotReady	5
Running pods	~35
Stuck pods	~10
Pods stuck in Terminating at first observation	9+

3. The two structural blockers

Neither of these is a misconfiguration. Both are Kubernetes behaving exactly as specified, and both stall stateful recovery indefinitely without operator action.

StatefulSet at-most-one semantics versus a Terminating pod on a dead node. The StatefulSet controller guarantees that at most one pod with a given ordinal exists at any time. When a node becomes unreachable, pods on that node enter Terminating but never complete deletion because the kubelet that would confirm it is gone. The controller sees an existing pod and will

not create a replacement. The database and cache StatefulSets sat blocked until we force-deleted the old pods with --force --grace-period=0.

RBD VolumeAttachments stuck on dead nodes. The VolumeAttachment object records that a volume is attached to a particular node. When the node is unreachable, the object persists because only the node's kubelet can confirm detachment. A replacement pod scheduled elsewhere fails with Multi-Attach: the volume appears to be attached in two places. Clearing this required deleting the VolumeAttachments, and where the node could not confirm detachment, removing their finalizers by hand.

Both fixes are genuinely unsafe operations. Force-deletion and finalizer removal trade the at-most-one guarantee for liveness. If the "dead" node is actually partitioned and its workload still running, you now have two writers. The 10-minute threshold we chose later is the only thing standing between that trade and a split-brain write.

4. Scheduling constraints that only fail when degraded

Two constraints were correct in the healthy nine-node state and became defects only when the topology shrank.

A hard nodeSelector pinned a network-diagnostic pod to the GPU compute node by hostname. With that node down, the pod was unschedulable anywhere. The fix is to remove the selector or express it as soft affinity.

A DoNotSchedule topology-spread constraint governed the third CoreDNS replica. With only two topology zones remaining among the four Ready nodes, the third replica had nowhere legal to go. Relaxing to ScheduleAnyway for degraded mode would resolve this.

A third entry was app-level rather than a cluster defect: a feed reader in CrashLoopBackOff because it rescheduled correctly but depended on the offline database. The scheduler did its job; the application had nowhere to go.

The lesson is that a scheduling constraint is only tested by the topology it will actually face. Degraded-mode scheduling deserves its own review pass.

5. The node-recovery CronJob

We designed and deployed a CronJob to handle the mechanical case: a node that is genuinely dead rather than merely partitioned, where the safe choice is to unblock the control plane's bookkeeping.

Property	Value
Schedule	every 2 minutes
Placement	pinned to the control-plane node via nodeSelector
NotReady threshold before acting	600 s (10 minutes)
Actions	force-delete Terminating pods on qualifying dead nodes; delete stale VolumeAttachments with --wait=false
Resources	ServiceAccount, ClusterRole, ClusterRoleBinding, ConfigMap, CronJob
activeDeadlineSeconds	120 (raised from 60 for image-pull headroom)
Scheduled run time	16 seconds

The 10-minute threshold sits safely above the 300-second default toleration. The ClusterRole needs get, list, delete, and watch on nodes, pods, and volumeattachments.

We tested it live against the five-node-down state. It cleared 15 Terminating pods across the GPU compute node and all four storage nodes, deleted roughly 12 stale VolumeAttachments, and the two StatefulSets immediately started fresh pods on the control-plane node.

Three defects emerged during deployment. The container image tags matching the cluster's Kubernetes minor version did not exist upstream; we fell back to a floating tag, which we have flagged as needing a digest pin. VolumeAttachment deletes hung because kubectl delete waits by default, and the wait needs the watch verb; we added the verb and also --wait=false. The deadline increase was the third.

One piece of residual noise: CSI nodeplugin DaemonSets re-create VolumeAttachments on dead nodes, so the job re-cleans them every cycle. This is idempotent and harmless, but it means the stale-VolumeAttachment count is never durably zero while a node is down.

6. Soft affinity is a hint, not placement

A separate finding emerged because pods that fled to the control-plane node during power cycles never came back. Before the work, 40 or more pods were crammed on the control-plane node while the GPU compute node sat idle. Roughly 14 deployments had no affinity rules at all.

The first pass added 22 soft (preferredDuringScheduling) affinity patches with tier weights, plus podAntiAffinity for the ingress replicas, an arch=amd64 requirement for the ad-filter (it had been landing on the arm64 monitoring node), and PVC migrations from local-path to the RBD SSD class. We cleared 11 stale VolumeAttachments during that migration, again requiring finalizer removal. A descheduler CronJob ran every 10 minutes with LowNodeUtilization (evict above 70% when a node sits below 30%), RemoveDuplicates, RemovePodsViolatingNodeAffinity, and RemovePodsHavingTooManyRestarts (threshold 50). Its test run evicted one pod with 124 restarts.

Pod distribution before and after that pass:

Node role	Before	After
GPU compute node	0 (powered off)	17
Control-plane compute node	40+	43
Monitoring node 1	0	9
Monitoring node 2	0	6
Monitoring node 3 (arm64)	0	6
Storage nodes 1–3	0 (powered off)	5–6 each
Storage node 4	0 (powered off)	7

The soft rules then failed in production. A later audit of all 105 pods found 15 misplaced by the dashboard's count, growing to roughly 24 as pods reshuffled mid-investigation. The root cause: every affinity was preferred, and preferences lose to the scheduler's LeastRequestedPriority scoring when one node has roughly 60 times the RAM of another. The descheduler was configured to act only on required violations.

We applied 39 patches converting deployments and StatefulSets to requiredDuringSchedulingIgnoredDuringExecution across tiers: 10 always-on, 4 always-on with compute fallback, 11 compute, 2 StatefulSets, 8 storage, 1 storage with a port-conflict exclusion. We also added a blanket kubernetes.io/arch: amd64 on always-on pods that had been crashlooping on the arm64 node. Misplaced pods dropped from 15–24 to zero. Nodes

Ready: 9 of 9. Crashlooping pods: 2 to 1.

Two side-findings from that rollout. All four storage nodes went NotReady mid-patch because of an MTU black hole: 9000 on the storage fabric, 1500 on the control-plane node's interface. TCP SYN/ACK passed; the roughly 1600-byte TLS Client Hello was silently dropped. We fixed it with a per-host MTU-1500 route persisted on the four storage nodes. This is a workaround, not a root-cause fix of broken path-MTU discovery.

Old ReplicaSets without the new required affinity kept recreating pods on the wrong node and holding the PVs, deadlocking the new ReplicaSet. We scaled down 7 old ReplicaSets by hand.

The 39 patches were applied at runtime only. Reapplying source manifests would silently revert them. This is an unresolved persistence gap.

7. The zombie storage client

The most instructive single failure, because none of the automation above could have cleared it.

On 15 March we found 21 or more blocked pods, 27 stale RBD VolumeAttachments held on the external-attacher finalizer, and 18 dual-attached PVs. The CSI provisioner had been stuck for 19 or more hours with blocked ControllerUnpublishVolume GRPC calls.

The root cause was a zombie Ceph client session on the control-plane node's secondary address. It held stale watchers on 10 RBD images. It had no kernel-mapped devices; the mappings were long gone. But the Ceph session persisted, so no node could take the exclusive lock.

Before touching anything, we did the preflight work. We mapped all 22 RBD PVCs to volume UUIDs, nodes, and namespaces. We verified all 8 kernel-mapped RBD devices (2 on the GPU compute node, 6 on the control-plane node) as ACTIVE and correctly placed, proving no stale kernel mappings. We distinguished the zombie session from the active session originating from the same address. Ceph confirmed HEALTH_OK.

The fix was one command: blocklist the zombie client's address and nonce. All 10 watchers released instantly. The active client was unaffected. Deleting the CSI provisioner pod cleared the 19-hour GRPC backlog, and all 27 stale VolumeAttachments auto-cleaned to zero. We removed the blocklist entry afterwards to restore clean state.

Follow-on repairs in the same pass: bounced 6 CrashLoopBackOff pods to reset exponential backoff; fixed two image pulls by adding registry auth to the node config on both compute nodes and creating imagePullSecrets in two namespaces; moved the DNS filter off a monitoring node where port 3000 collided with a Grafana container; capped a backup CronJob with activeDeadlineSeconds: 600; gave the log-shipper DaemonSet an emptyDir /tmp to fix positions-file write errors on all 9 pods.

End state: stale VolumeAttachments 27 to 0; dual-attached PVs 18 to 0; 95 pods Running, 9 Completed, 1 not running (an expired third-party session needing a manual QR scan, not automatable); Ceph HEALTH_OK. The database recovered its WAL, the cache reloaded its AOF, file-sync returned 200 OK.

The underlying reason the storage client session survived unmapping was never established. We mitigated the 19-hour stall; we did not prevent it.

8. What the evidence supports and what it does not

The storage backend was deliberately removed. "Recovery" here means unblocking the control plane's bookkeeping, not recovering data. No data loss was observed; the database's WAL and

the cache's AOF both replayed cleanly. But this was a test of orchestration, not of storage durability.

The node-recovery CronJob handles the mechanical case and only the mechanical case. It looks at NotReady nodes. The zombie-client failure had every node Ready and a healthy storage cluster; the job would have run, found nothing to do, and exited. Automation and diagnosis remain different jobs.

Several items remain unresolved. The 39 affinity patches are runtime-only and revert on manifest reapply. The MTU fix is a per-host route workaround. The CronJob's container image runs on a floating tag rather than a pinned digest. The CSI driver re-creates VolumeAttachments on dead nodes each cycle, so the cleanup is perpetual rather than terminal.

Force-deletion and finalizer removal are genuinely unsafe. The 10-minute threshold is a guess at "long enough to be sure the node is dead" rather than a proof. A partitioned node that recovers after 9 minutes could find its workloads duplicated.

Sample size is one cluster, one induced failure, one zombie-client incident. The structural blockers (StatefulSet semantics, VolumeAttachment persistence) are by-design behaviours and will reproduce anywhere. The specific failure modes around scheduling constraints and MTU black holes may or may not generalise.

PureTensor operates a sovereign AI infrastructure fleet (on-premises GPU compute, storage, and serving) run day-to-day by autonomous agents under human direction.

PT-TN-2026-008. Measurements taken 11–15 March 2026; placement-hardening pass closed early on 16 March 2026. Raw artefacts retained.