KUBECON 2026 · ETCD / CONTROL PLANE RELIABILITY

Death by a Thousand Patches

How Pod Status Updates Exhaust etcd — and What to Do About It

MVCC internals · failure mechanics · blast radius · recovery · prevention at scale

Reference: github.com/hstiwana/kubecon-2026-etcd-mvcc

Audience: platform / SRE / Kubernetes control-plane engineers



Sharik Pahwa (Pr. SA AWS)
Hardeep Singh Tiwana (Sr. TAM AWS)

AGENDA

1. **Why a cluster with zero crashed nodes goes fully read-only**
2. **The mechanism**
   etcd's MVCC model → why every PATCH is a full-object write, not a diff
3. **The trigger**
   Pod status churn: kubelet, scheduler, controllers, and the annotation anti-pattern
4. **The failure mode**
   Quota breach → NOSPACE alarm → cluster-wide write lock
5. **The blast radius**
   What still works, what silently dies, and why it's worse than an outage
6. **Recovery**
   compact → defrag → disarm — and the step that is easy to miss
7. **Prevention**
   Alert, admit, architect — what to do before the alarm fires

COLD OPEN

A healthy 3-node cluster. Under two minutes.

```
$ ./demo.sh break
[DEMO] Writing 100KB configmaps until
       the 50 MB test quota is hit...
  Created 50 configmaps.  etcd: 47.1 / 50 MB
  Created 85 configmaps.  etcd: 50.7 / 50 MB
  error: etcdserver: mvcc: database space exceeded
  error: etcdserver: mvcc: database space exceeded
  WRITES BLOCKED — cluster is read-only!

$ kubectl create namespace test
Error: etcdserver: mvcc: database space exceeded

$ kubectl delete namespace mvcc-demo
Error (InternalError): etcdserver: mvcc:
database space exceeded

$ kubectl run test --image=busybox
Error: etcdserver: mvcc: database space exceeded
```

- **Create. Blocked.**
- **Delete. Blocked.**
- **Scale. Blocked.**
- **Reschedule a crashed pod. Blocked.**

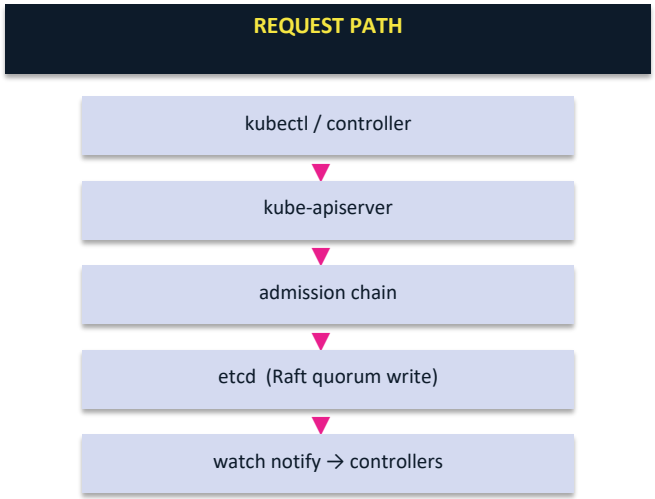
No node failed. No image pull error. No OOMKill.
Production gets here through revision churn (status patches, annotations, leases); the demo uses large objects.

Validated against github.com/hstiwana/kubecon-2026-etcd-mvcc

FOUNDATIONS

etcd is the single source of truth for the entire cluster

- Every object lives here. Pods, Deployments, Services, Secrets, ConfigMaps, Leases, Events (all of it), as key-value pairs.
- The API server is a thin proxy. kube-apiserver validates and routes; etcd is the durable state and the linearizable read/write path (Raft consensus).
- Default quota: 2 GB upstream, 8 GB suggested maximum (managed services such as Amazon EKS use 8 GB). Sized for the median cluster, not for pathological write patterns.
- It is a distributed system in the critical path of literally every mutating operation in Kubernetes.

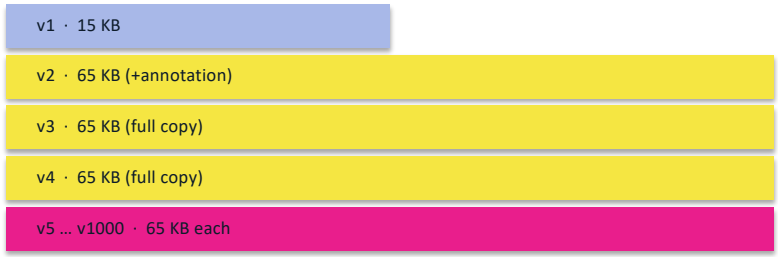


THE MECHANISM · MVCC

Every write is a new, full revision → never a diff. One global counter; every write takes the next number.

etcd uses Multi-Version Concurrency Control so reads never block writes. The cost: old versions of every key accumulate until something removes them.

One Pod object but revisions accumulate in etcd



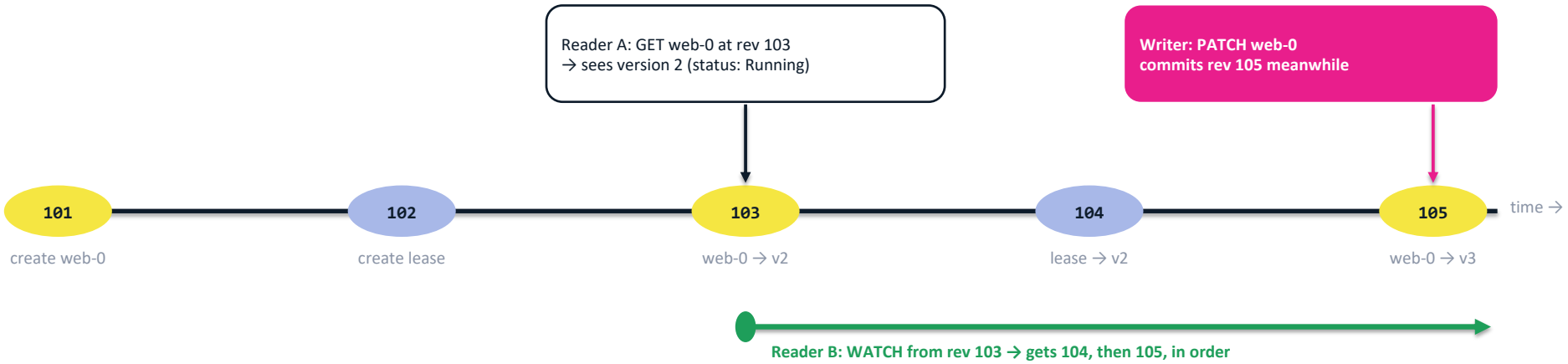
After 1,000 patches: 1000 × 65 KB ≈ 65 MB stored — for ONE pod object, before compaction.

WHY THIS MATTERS

- Reads never block → MVCC lets etcd serve consistent snapshots while writes continue.
- But nothing is free. Every PUT/PATCH keeps the FULL prior object alive as history until compacted.
- Object count can stay flat while revision count explodes → small clusters are not immune!

MVCC IN ACTION · 2

Readers ask for a revision. Writers never block them.



- Reader A's answer cannot change. Rev 103 is fixed, no matter what lands after it.
- In Kubernetes, resourceVersion is backed by the etcd revision (clients should treat it as opaque).
- This works only because the old revisions are still stored. That is the storage cost from the last slide.

MVCC IN ACTION · 3

Delete adds a row. A key can live more than one life.

TWO MORE WRITES

Rev	Operation	Version
106	DELETE web-0 → writes a tombstone	0 (reset)
107	CREATE web-0 again (StatefulSet, same name)	1

KEY INDEX FOR …/pods/default/web-0

Generation 1 `101 · v1` `103 · v2` `105 · v3` `106 · tombstone` ← generation ends

Generation 2 `107 · v1` ← new life, version starts at 1 again

- The tombstone is one more entry in the b+tree (marked type "t"). Nothing is removed yet.
- So a delete, on its own, frees no space. Revisions 101 to 106 are all still on disk.
- Versions only count up within a generation. Revisions count up forever, for the whole cluster.

COMPACTION vs DEFRAGMENTATION

Compaction removes old revisions (logical). Defrag rewrites the file (physical). Both are required.

BEFORE: every revision

```
101  web-0 v1
102  lease v1
103  web-0 v2
104  lease v2
105  web-0 v3
106  web-0 tombstone
107  web-0 v1 (gen 2)
```

AFTER: compact at rev 107

```
~~101  web-0 v1~~
~~102  lease v1~~
~~103  web-0 v2~~
104  lease v2  ✓ latest
~~105  web-0 v3~~
~~106  web-0 tombstone~~
107  web-0 v1 (gen 2)  ✓ latest
```

Kept: the newest value of each live key. Dropped: superseded values, and generation 1 of web-0, which ended before rev 107.

THE bbolt FILE ON DISK

After compaction: same size



grey = free pages still inside the file

defrag

After defrag: file rewritten



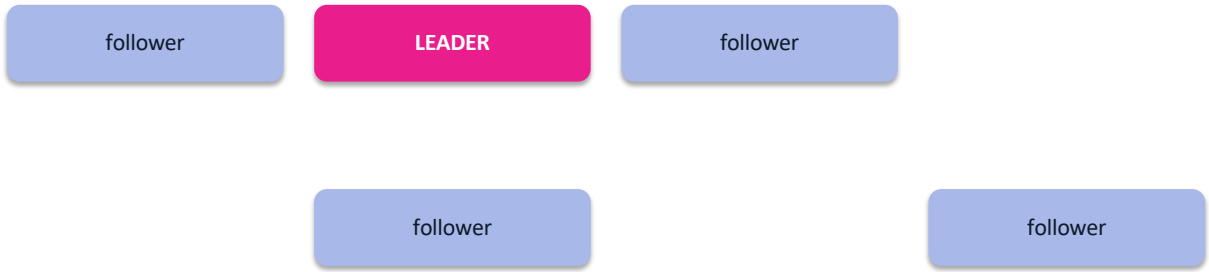
space returned to the filesystem

After compaction, a read or watch at rev 103 fails: "required revision has been compacted". Kubernetes clients get 410 Gone and re-list.

FOUNDATIONS · ARCHITECTURE

Example: a 5-member Raft cluster, one leader, replicated writes

etcd uses Raft consensus. The leader accepts writes and replicates to followers; serializable reads can be served by any member. Adding nodes improves durability, not throughput.



WHAT THIS MEANS OPERATIONALLY

- Odd member count: five members tolerate 2 failures, three tolerate 1
- More nodes ≠ more throughput. Every write needs majority replication — more members means MORE Raft overhead, not less
- Kubernetes docs: run a static five-member etcd in production, don't autoscale it; going from 3 to 5 is the reasonable scale-up
- A replaced member re-syncs the FULL keyspace from the leader — fast disks matter more than more members

etcd.io FAQ · kubernetes.io: Operating etcd clusters for Kubernetes

THE TRIGGER

Where does the write volume actually come from?

kube-scheduler retries

Pod can't be scheduled (no capacity) → status re-patched on every reconcile pass until it lands

Crash-loop status updates

kubelet writes container status + restart count + condition transitions on every restart cycle

Node lease renewals

Every node renews its Lease object every ~10s by default — baseline load that scales with fleet size

Broken / runaway controllers

A reconcile loop that patches status unconditionally, even with no actual change (non-idempotent controller)

Annotations as an IPC channel

Two controllers 'talking' via pod annotations — each message is a full-object revision, not a message queue

Unbounded Job history

Jobs without ttlSecondsAfterFinished, or CronJobs with raised history limits, pile up as objects

Scale math: 50-node cluster ≈ 5 writes/sec baseline; 10k nodes ≈ 1,000/sec from leases alone. Alert at 3× YOUR baseline.

THE FAILURE MODE

Quota breach → NOSPACE alarm → cluster-wide write lock



THE QUOTA MECHANICS

- Upstream etcd: the alarm fires when a member's backend exceeds its quota (2 GB default). etcd_mvcc_db_total_size_in_use_in_bytes is live data; the physical file (etcd_mvcc_db_total_size_in_bytes) can be 2–3× larger due to fragmentation.
- The quota checks the physical file size, not logical. This means fragmentation alone — without any growth in live data — can trip the alarm.
- Compaction must run first (mark old revisions dead), then defrag (reclaim the space). Both steps, every member.

BLAST RADIUS

What breaks, what keeps running — and why it's worse than an outage

Action / Component	Result	Why
Deploy new pods	BLOCKED	Create = write
Scale deployments	BLOCKED	Spec update = write
Delete a namespace	BLOCKED	Finalizer cascade requires writes
HPA / VPA autoscaling	DEAD	Can't write new replica counts
Node lease renewal	FAILS	Lease and status updates can't be written; node health goes stale
Pod eviction / reschedule	BLOCKED	Eviction record requires a write
Service endpoint updates	FROZEN	Restarted pod's new IP never registers
Cert / secret rotation	BLOCKED	Can't write the new Secret object
Existing running pods	OK	No new write needed — until they crash
Some object deletes	MIXED	Small tombstones may succeed; cascades won't

It looks fine until the first pod restarts. One crash cascades into a hole in the service. Blast radius: the entire cluster, every namespace.

THE TRAP → RECOVERY

"Just delete some objects" doesn't work — but the recovery runbook takes ~30 seconds when known

- Many deletes need an update first. Running pods and objects with finalizers must be updated before removal, and updates are blocked.
- Finalizers make it worse. Namespace deletion triggers a cascade of dependent-object updates — all blocked.
- Confusing partial success. You may delete a ConfigMap fine, but the same delete on a namespace fails — inconsistent behavior, not a bug.
- Running pods can't be touched cleanly. Anything requiring an update-before-delete (running pods, finalized objects) is stuck.

RECOVERY RUNBOOK

- 1. etcdctl alarm list → confirm NOSPACE
- 2. etcdctl endpoint status → get current revision
- 3. etcdctl compact <rev> → remove old revisions
- 4. etcdctl defrag → reclaim physical space (REQUIRED — compaction alone won't do it)
- 5. etcdctl alarm disarm → resume writes

Validated: 50.7 MB → 21.8 MB after compact + defrag (demo numbers)

PREVENTION · ALERTING

Catch it in days, not minutes — alert before the alarm

50%
days to investigate

75%
page someone now

quota consumed

quota consumed

50% quota

```
etcd_mvcc_db_total_size_in_use_in_bytes / etcd_server_quota_backend_bytes > 0.5
```

75% quota

```
etcd_mvcc_db_total_size_in_use_in_bytes / etcd_server_quota_backend_bytes > 0.75
```

Fragmentation (defrag needed)

```
1 - (etcd_mvcc_db_total_size_in_use_in_bytes / etcd_mvcc_db_total_size_in_bytes) > 0.5
```

Anomalous write rate

```
rate(etcd_mvcc_put_total[5m]) > 3 * avg_over_time(rate(...)[7d:1h])
```

Alert on the metric etcd actually enforces: physical file size vs quota. These queries are self-scaling — no fixed thresholds.

PREVENTION · GUARDRAILS & PRACTICES

Admission control + operating habits that prevent this from ever paging you

- Kubernetes ValidatingAdmissionPolicy (CEL, stable since v1.30), or Kyverno / OPA Gatekeeper, can reject objects whose annotations exceed a size threshold, e.g. 10 KB
- Enforced at admission, before the object reaches etcd. The API server already caps total annotations at 256 KiB; a tighter limit keeps each revision small
- Roll out with validationActions: [Audit] first to find the offending controller, then switch to Deny
- VAP runs in-process (no webhook to operate). Webhook engines add a network hop and a failurePolicy decision

Rejected requests never generate a write — never consume etcd quota.

GOTCHA: kubectl apply counts against your limit (tested on kind, v1.37)
Client-side kubectl apply adds kubectl.kubernetes.io/last-applied-configuration, a copy of the object you applied, and it is summed like any other annotation. A pod at exactly 10,240 B passed with kubectl create but was denied with kubectl apply. Leave headroom for it.

```
manifests/vap-annotation-limit.yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicy
metadata:
  name: limit-annotation-size
spec:
  matchConstraints:
    resourceRules:
    - apiGroups: [""]
      apiVersions: ["v1"]
      operations: ["CREATE", "UPDATE"]
      resources: ["pods"]
  validations:
  - expression: >-
      !has(object.metadata.annotations) ||
      object.metadata.annotations.map(k, size(k) +
        size(object.metadata.annotations[k])).sum() <= 10240
    message: "Annotations exceed 10KB"
# plus a ValidatingAdmissionPolicyBinding
# (validationActions: [Audit], then [Deny])
```

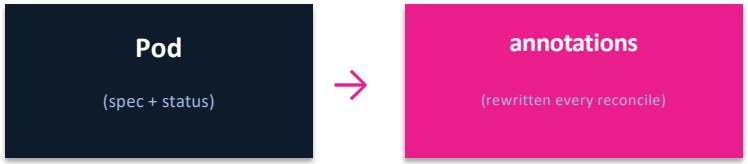


PREVENTION · ARCHITECTURAL FIX

Offload high-churn data out of core objects

Move frequently-updated metadata out of pod annotations into small, disposable, TTL'd CRDs — isolate the churn from your stable core objects.

BEFORE — churn lives inside the Pod



Every annotation patch = a full new revision of the ENTIRE pod object.

AFTER — churn isolated to a disposable CRD



Churn hits a small object with a bounded lifetime — Pod stays stable.

TAKEAWAYS

What to do Monday morning

- Alert at 50% / 75% on etcd_mvcc_db_total_size_in_use_in_bytes — not physical file size
- Deploy an admission guardrail capping annotation/object size before it reaches etcd
- Audit patch-heavy controllers for idempotency — don't write status if nothing changed
- Put defrag on a schedule compaction alone will not save you
- Know your recovery runbook cold compact → defrag → disarm, in that order, before you need it at 3am

Demo + manifests + alerting rules: github.com/hstiwana/kubecon-2026-etcd-mvcc

Further reading: etcd.io Space Quota docs · KEP-4222 (CBOR serialization) · AWS “Managing etcd database size on Amazon EKS clusters” · Datadog etcd storage guide



Thank You

Q & A

github.com/hstiwana/kubecon-2026-etcd-mvcc

Sharik Pahwa (Pr. SA AWS)
Hardeep Singh Tiwana (Sr. TAM AWS)