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



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

AGENDA

1. **Why a cluster with zero crashed nodes goes fully read-only**
2. **The trigger**
   Pod status churn: kubelet, scheduler, controllers, and the annotation anti-pattern
3. **The failure mode**
   Quota breach → NOSPACE alarm → cluster-wide write lock
4. **The blast radius**
   What still works, what silently dies, and why it's worse than an outage
5. **The mechanism**
   etcd's MVCC model → why every PATCH is a full-object write, not a diff
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

Lets Break the etcd!

A healthy 3-node kind cluster. Under two minutes.

```
$ ./demo.sh break
[DEMO] === SCENARIO: Status Patch Storm ===
[DEMO] Replicas: 80 | payload: 96KiB/pod | parallelism: 20
[DEMO] Patching status on 80 pods as fast as the API server accepts it...
  Iteration N | elapsed <90s
  etcd db size: >100 MB / 100 MB
  Error from server: etcdserver: mvcc: database space exceeded
  WRITES BLOCKED: cluster is read-only!
$ kubectl create namespace test
Error: etcdserver: mvcc: database space exceeded
$ kubectl scale deployment status-victim --replicas=0 -n mvcc-demo
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, scheduling updates, annotations, leases); the demo uses pod status subresource patches.

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

THE TRIGGER

Where does the write volume actually come from?

kube-scheduler retries

Pod can't be scheduled (no capacity) → status patching conditions PodScheduled = False

Crash-loop status updates

kubelet writes container status + restart count + current state, and the lastTerminationState

Node lease renewals

Every node renews its Lease object every ~10s by default, baseline load that scales with fleet size (kube-node-lease)

Broken / runaway controllers

A reconcile loop that patches status unconditionally, even with no actual change (reflect.DeepEqual) → client.Status().Update()

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 (Prometheus metric 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

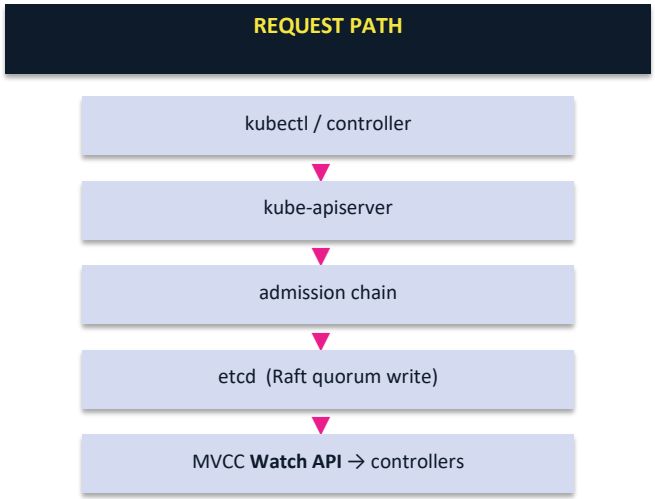
Action / Component	Result	Why
Deploy new pods	FAILS	Create = write
Scale deployments	FAILS	Spec update = write
Delete a namespace	FAILS	Finalizer cascade requires writes
HPA / VPA autoscaling	FAILS	Can't write new replica counts
Node lease renewal	FAILS	Lease and status updates can't be written; node health goes stale
Pod eviction / reschedule	FAILS	Eviction record requires a write
Service endpoint updates	FAILS	Restarted pod's new IP never registers
Cert / secret rotation	FAILS	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.

FOUNDATIONS

etcd is the single source of truth for the entire cluster

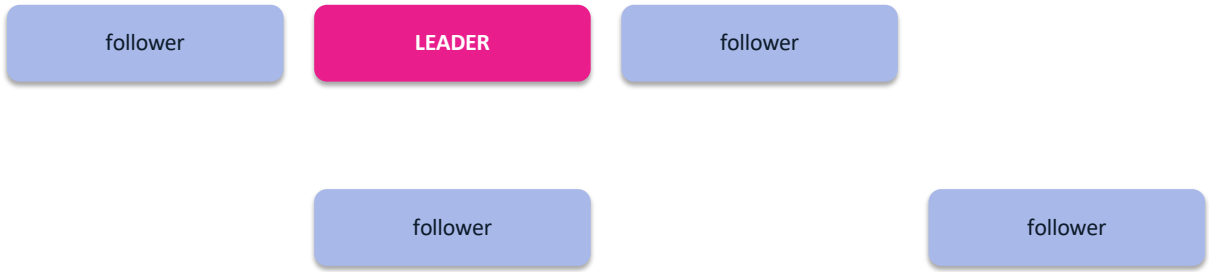
- Every object lives here. Pods, Deployments, Services, Secrets, ConfigMaps, Leases, Events (all of it), as key-value pairs.
- The API server is a thin proxy. kube-apiserver validates and routes; etcd is the durable state and the linearizable read/write path (Raft consensus).
- Default quota: 2 GB upstream, 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.



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

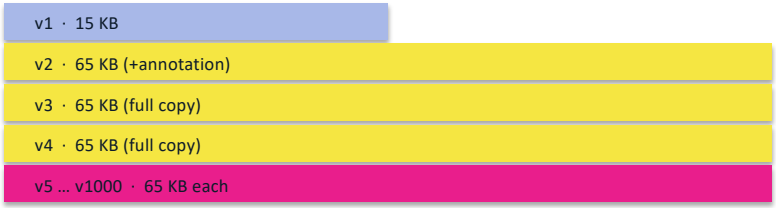
etcd.io FAQ · kubernetes.io: Operating etcd clusters for Kubernetes

THE 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

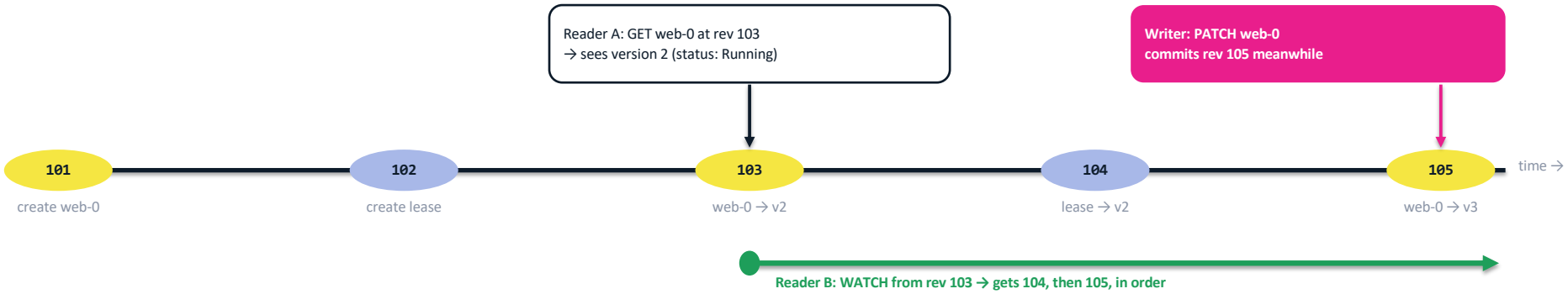


After 1,000 patches: 1000 × 65 KB ≈ 65 MB stored for ONE pod object, before compaction.

WHY THIS MATTERS

- Reads never block: Reader A at rev 103 always sees v2, even while rev 105 commits.
- Nothing is free: every PUT/PATCH keeps the FULL prior object until compacted.
- Object count stays flat while revision count explodes; small clusters are not immune!

Global Revision Counter: one counter for the whole cluster. Different keys, same counter.



COMPACTION

Delete adds a tombstone. Compaction removes old revisions (logical). Next: defrag reclaims the file (physical).

BEFORE: every revision



AFTER: compact at rev 107

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

WHY A DELETE ALONE DOESN'T HELP

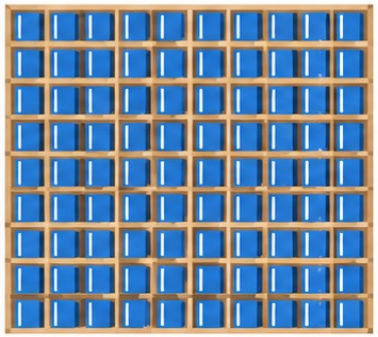
- DELETE writes a tombstone (rev 106): one more entry in the b+tree. Nothing is removed yet.
- Revisions 101 to 106 all stay on disk until compaction.
- Re-create the same name (StatefulSet) and a new generation starts at v1 (rev 107).
- Versions count up within a generation; revisions count up forever, cluster-wide.

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

etcd: Compaction vs. Defragmentation

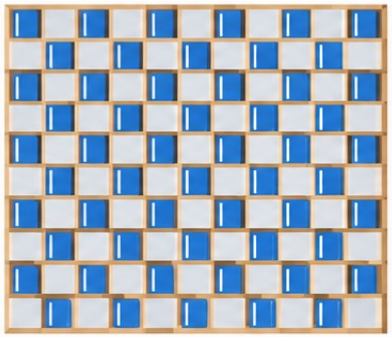
Removing old revisions is not the same as shrinking the database file

1. Full bookshelf



Backend file: 100 units
In use: 100 units
Unused space: 0%

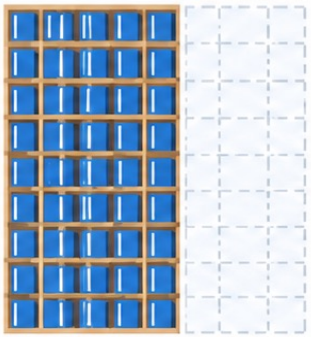
2. After compaction



Backend file: 100 units
In use: 50 units
Unused space: 50%

Old revisions removed; freed space can be reused inside etcd.

3. After defragmentation



Space returned to the OS

Backend file: ~50 units
In use: 50 units
Unused space: ~0%

Live data repacked; backend file shrinks.

Unused-space ratio = 1 − (in-use bytes ÷ backend file bytes)

After compaction: $1 - \frac{50}{100} = 50\%$

After defrag: $1 - \frac{50}{50} \approx 0\%$

Simplified illustration: actual files retain metadata and allocation overhead. Compaction removes historical revisions, not current keys. Empty slots are reusable; this is not a literal contiguous-book allocation constraint.

THE TRAP → RECOVERY

"Just delete some objects" doesn't work!

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

Watch the demo: backend file size drops only after defrag; exact MB varies by run.

PREVENTION · ALERTING

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

50%

quota consumed; you have days to investigate

75%

quota consumed; you should page someone now

Alert when backend file size hit 50% quota

```
etcd_mvcc_db_total_size_in_bytes / etcd_server_quota_backend_bytes > 0.5
```

Alert when backend file size hit 75% quota

```
etcd_mvcc_db_total_size_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 backend file size vs quota.

Use in-use vs total to catch fragmentation. These queries self-scale; no fixed write thresholds.

PREVENTION · GUARDRAILS & PRACTICES

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

- Kubernetes **ValidatingAdmissionPolicy** (CEL, stable since v1.30) can block risky pod writes before they reach etcd
- Policy in this demo rejects oversized pod annotations, labels, and status condition messages on pods/status
- Roll out with Audit first; advanced option: **Kyverno** adds policy-as-code, reports, exceptions, background scans, and CLI tests

STAGE DEMO: ./demo.sh guardrails verifies two denials
1. 50KiB annotation → RequestEntityTooLarge
2. 96KiB status condition message → RequestEntityTooLarge

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

```
manifests/native-pod-mvcc-guardrails.yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicy
metadata:
  name: pod-mvcc-write-guardrails
spec:
  matchConstraints:
    resourceRules:
    - resources: ["pods"]
      operations: ["CREATE", "UPDATE"]
  validations:
  - expression: annotations entry <= 10KiB
  - expression: labels entry <= 2KiB
---
kind: ValidatingAdmissionPolicy
metadata:
  name: pod-status-mvcc-guardrails
spec:
  matchConstraints:
    resourceRules:
    - resources: ["pods/status"]
      operations: ["UPDATE"]
  validations:
  - expression: status.condition.message <= 10KiB
```

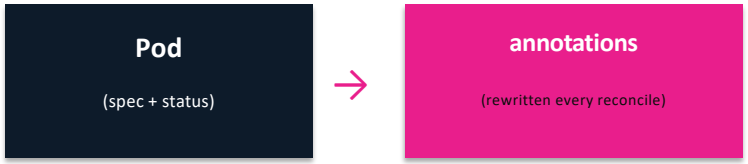


PREVENTION · ARCHITECTURAL FIX

Offload high-churn data out of core objects

Move frequently-updated metadata out of pod annotations into small, disposable, TTL'd CRDs to 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 bounded side object



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

TAKEAWAYS

What to do Monday morning

- Alert at 50% / 75% on etcd_mvcc_db_total_size_in_bytes / quota, then use in-use vs total for fragmentation
- Deploy native admission guardrails for pod metadata and status bloat 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: 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 Space Quota docs · Kubernetes ValidatingAdmissionPolicy · KEP-4222 · AWS EKS etcd size guide · Datadog etcd storage guide
https://etcd.io/docs/v3.7/op-guide/maintenance/

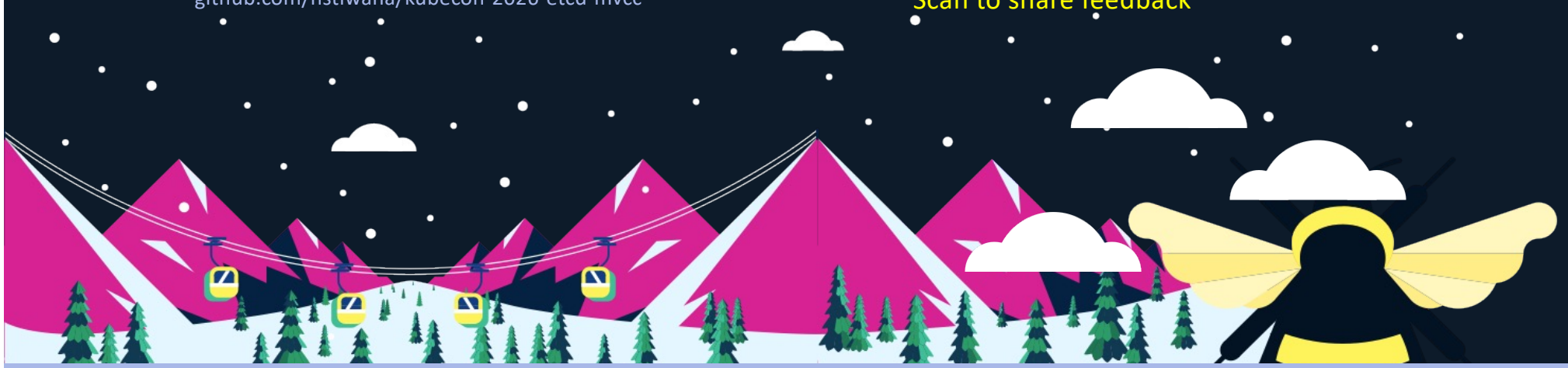
Thank You

We'd love to hear your feedback!

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

Death by a Thousand Patches: How Pod Status Updates Exhaust etcd and What to Do About It

Scan to share feedback



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