KubeCon CloudNativeCon North America 2026

#KubeCon #CloudNativeCon

Why Quotas Are Hard in Kubernetes

Lessons from Building a Production-Grade Quota System

Xing Yang & Deepak Kinni, Broadcom



Agenda

KubeCon CloudNativeCon North America 2026

INTRO Why do we need resource quota?

FOCUS: STORAGE QUOTA PVCs · VolumeSnapshots · VMs

PART I
ResourceQuota Mechanics & Distributed Systems Reality

1. What is ResourceQuota?
2. Life cycle of a ResourceQuota
3. Architectural choices
4. Race conditions in quota
5. Quota through the CAP lens
6. Why storage quota is harder

PART II
A Custom Storage Quota System for Complex Limits



7. Why ResourceQuota wasn't enough
8. Architecture and data model
9. Quotas that span namespaces
10. Life of a request: reserve, then commit
11. Pitfalls at scale and drift correction
12. Our system through the CAP lens

WRAP-UP Evaluating quota architectures on the CAP triangle · Key takeaways · Demo · Q&A

Why Do We Need Resource Quota?

KubeCon CloudNativeCon North America 2026

Clusters are shared. CPU, memory, storage and etcd are not infinite.

WITHOUT QUOTA



one shared pool: CPU · memory · storage · etcd



- ✕ **Noisy neighbors:** one team's burst starves the rest
- ✕ **Runaway provisioning:** a bug creates 1000s of PVCs and Snapshots
- ✕ **Control-plane pressure:** unbounded objects bloat etcd

WITH RESOURCEQUOTA



one shared pool: CPU · memory · storage · etcd




```yaml
kind: ResourceQuota
metadata:
  namespace: team-a
spec:
  hard:
    requests.cpu: "20"
    requests.storage: 2Ti
    persistentvolumeclaims: "50"
    count/volumesnapshots.snapshot.storage.k8s.io: "100"
```

Per-namespace hard limits, enforced at creation. Simple in theory: track, limit, reject.

KubeCon | CloudNativeCon
North America 2026

PART I

ResourceQuota Mechanics & Distributed Systems Reality

Simple in theory: Track, Limit, Reject. Complex in practice across distributed state.



What is a ResourceQuota?

A namespaced API object that caps the **aggregate** resources a namespace can consume.

Checked by an admission plugin when objects are created; usage kept accurate by a controller.



Compute Quotas

Bounds total CPU and memory requests and limits.



Storage Quotas

Caps total requested storage and PVC count, overall or per StorageClass.



Object Counts

Restricts how many pods, services, secrets, snapshots and more can exist.

Scoped to a namespace: a request must fit every quota in that namespace.

The ResourceQuota API

KubeCon CloudNativeCon North America 2026

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: team-a-storage
  namespace: team-a

spec:
  hard:
    requests.storage: 2Ti
    persistentvolumeclaims: "50"
    gold.storageclass.storage.k8s.io/requests.storage: 500Gi
    count/volumesnapshots.snapshot.storage.k8s.io: "100"

  # optional: scopes / scopeSelector
  #   narrow which objects this quota counts
  #   e.g. VolumeAttributesClass for PVCs

status:
  hard:  { ...copied from spec.hard... }
  used:
    requests.storage: 1200Gi
    persistentvolumeclaims: "31"
    gold.storageclass.storage.k8s.io/requests.storage: 380Gi
    count/volumesnapshots.snapshot.storage.k8s.io: "42"
```



1

1 spec.hard: what you can cap

Compute `requests.cpu, requests.memory`
Storage `requests.storage, persistentvolumeclaims`
Per class `<class>.storageclass.storage.k8s.io/…`
Objects `count/<resource>.<group>`

2

2 scopes / scopeSelector: what counts

Filters which objects a quota counts.
Pods `BestEffort, Terminating, PriorityClass, …`
PVCs `VolumeAttributesClass`
StorageClass splitting is separate: it lives in `spec.hard` keys.

3

3 status: hard vs used

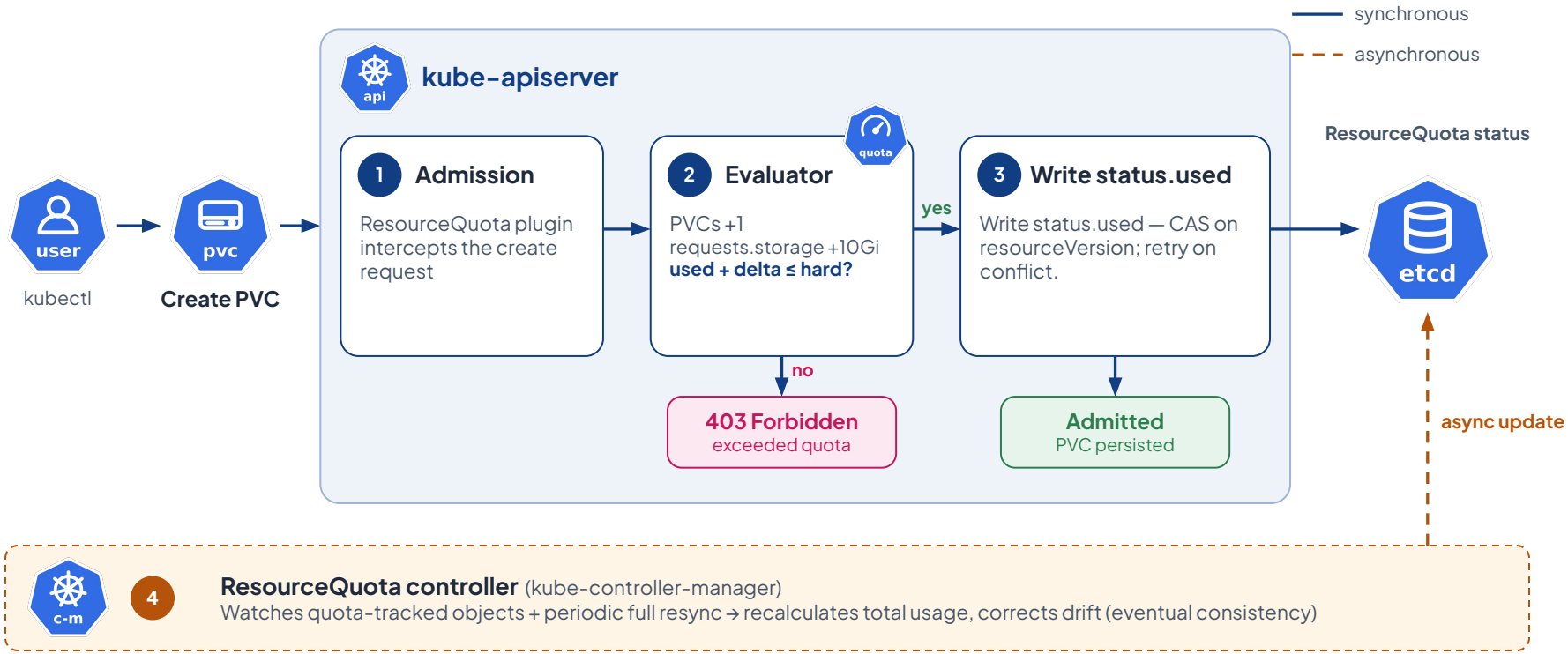
`hard` copied from spec by the controller; admission enforces this copy

`used` incremented by admission on create/update (sync); recalculated by the controller (async)

```
$ kubectl describe resourcequota team-a-storage -n team-a      # Used vs Hard, per resource
```

Life cycle of a ResourceQuota

KubeCon CloudNativeCon North America 2026



Architectural Choices



Optimistic Locking

Relies on etcd Compare-And-Swap (ResourceVersion). Concurrent resource creations race to update the single ResourceQuota object status.



Lookaside Cache

Admission reads quotas from the API server's informer cache, not etcd. A small lookaside cache holds quotas it just wrote, so back-to-back requests see the latest version and don't conflict.

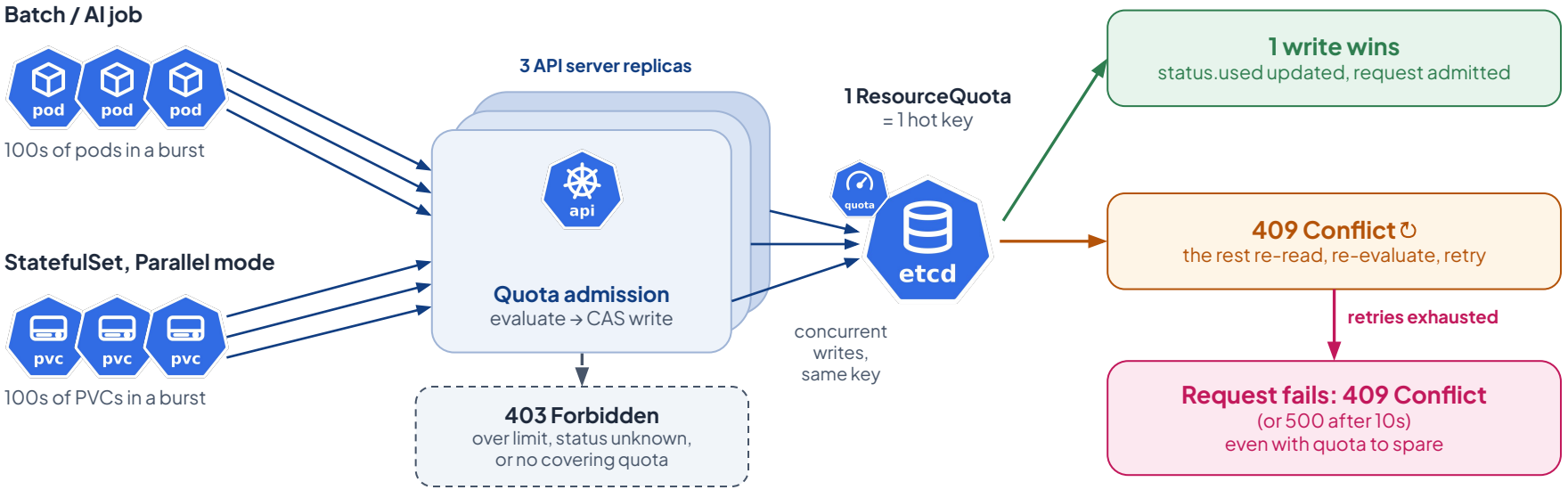


Per-Namespace Batch

Requests are queued per namespace and checked one by one, then a single status write covers the batch. A hot quota is still a single object.

Race Conditions in Quota

KubeCon CloudNativeCon North America 2026

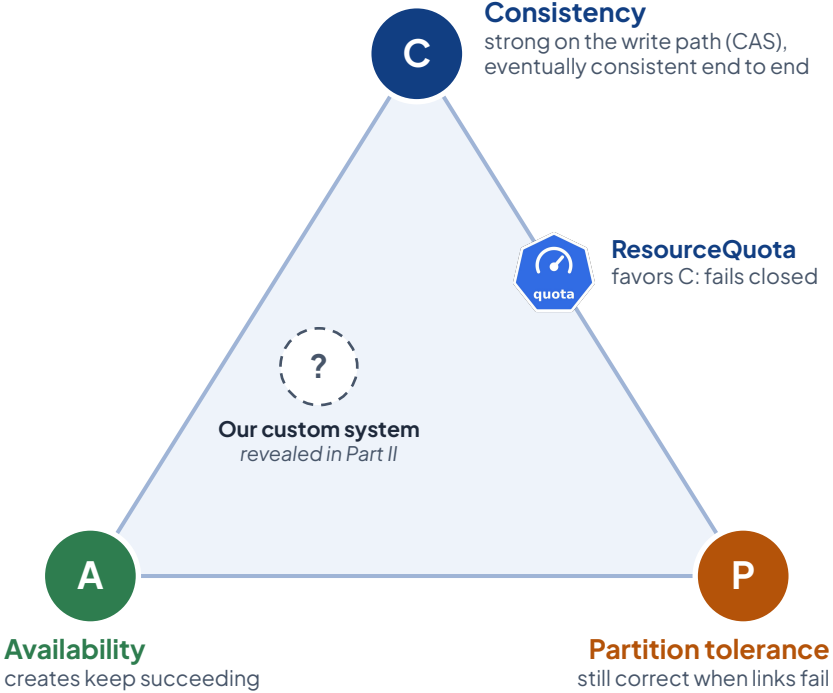


Every distributed quota system has to handle concurrent requests. A lost race returns a 409 or a timeout.

Quota Through the CAP Lens

KubeCon CloudNativeCon North America 2026

CAP theorem: during a network partition, a system must choose consistency (latest data) or availability (always responds).



wins: **consistency** pays: **latency**	**Synchronous CAS write**	Usage recorded before admit; 409 retries on hot namespaces
wins: **consistency** pays: **availability**	**Fails closed**	Can't evaluate quota, or no quota covers a limited resource? Rejected
wins: **fewer retries** pays: **complexity**	**Batching + lookaside cache**	Per-namespace batching claws back throughput; consistency intact
wins: **self-healing** pays: **drift window**	**Async reconciliation controller**	Recomputes usage from ground truth; drift fixed eventually

No partition? The trade-off doesn't go away: every synchronous check still trades latency for consistency (PACELC).

KubeCon CloudNativeCon North America 2026

PART II

Building a Custom Storage Quota System for Complex Limits



Why ResourceQuota Wasn't Enough for Us

KubeCon CloudNativeCon North America 2026

1 Many object kinds, one pool

PVCs, VolumeSnapshots, VMs and VM snapshots all draw from one storage budget. ResourceQuota does not support this.

The storage consumption may not be obvious for a resource

2 Limits at several granularities

A namespace total and per-storage-policy limits, enforced together. One policy can back several StorageClasses.

We charge by policy ID, not by class name.

3 Cost is known late, charged early

The real size settles only when the CSI driver provisions. We charge at admission and reconcile later.

The gap between the two is where drift lives.

4 Quotas may span namespaces

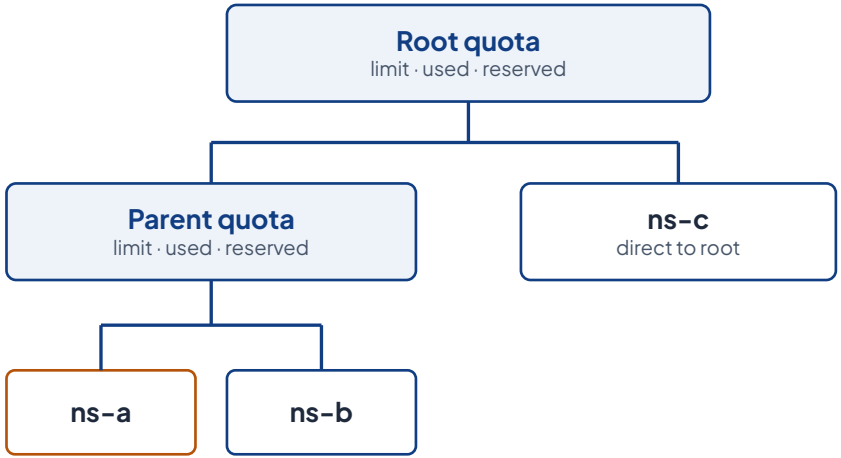
Optionally, one limit shared by many namespaces, possibly nested in several levels. ResourceQuota stops at the namespace boundary.

Namespaces come and go; the pool stays.

Same goal as ResourceQuota: track, limit, reject. Different shape: many kinds, many limits, late truth, shared pools.

Quotas That Span Namespaces

KubeCon CloudNativeCon North America 2026



Shared quotas have deterministic names, so lookup is O(1): no LIST on the hot path.

The validation ladder

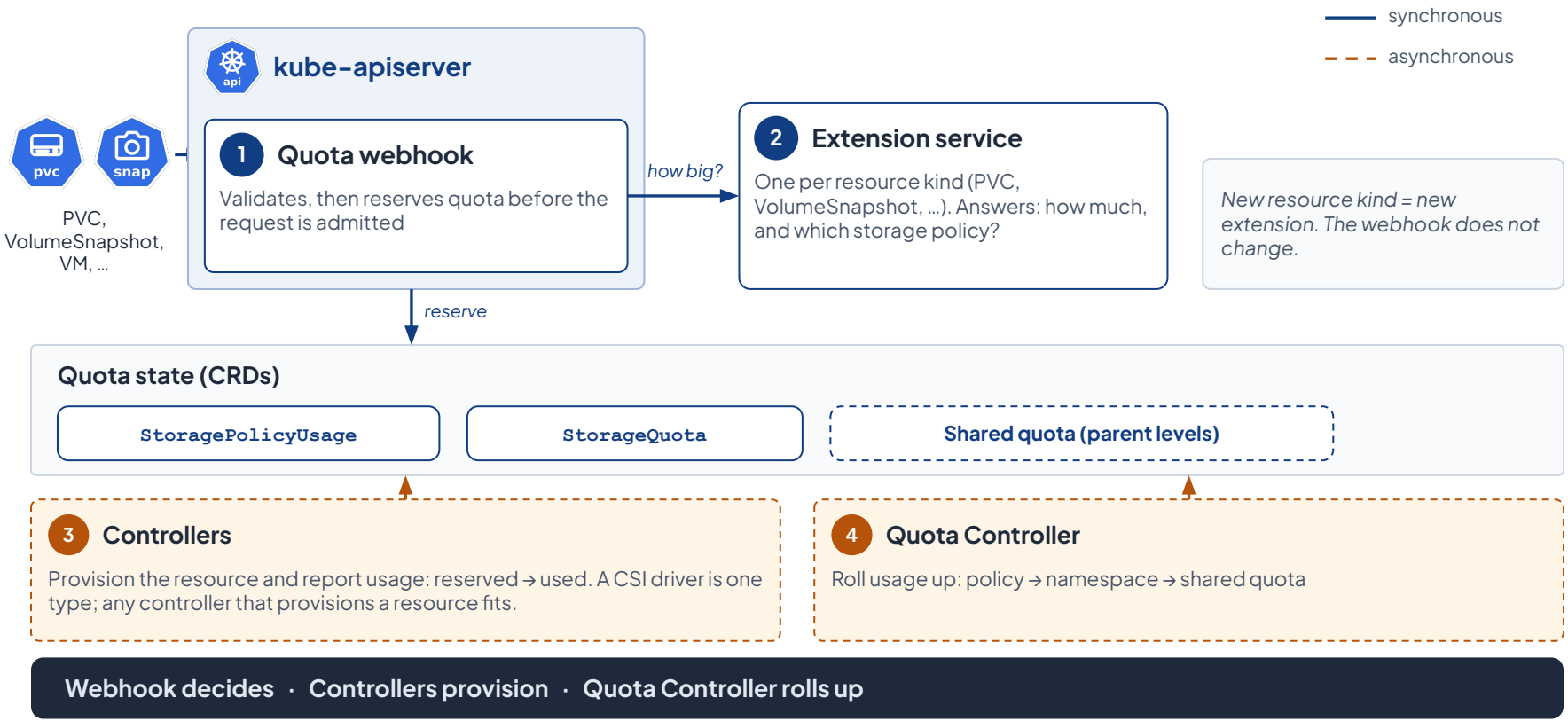
1. **Per-policy limit** in the namespace
2. **Namespace total** StorageQuota
3. **Parent quota** if the namespace joined one
4. **Root quota** if the parent joined one

Every level is checked before anything is reserved: a denial charges nothing.

Four paths, one webhook

```
Namespace → Parent → Root
Namespace → Root
Namespace → Parent
Namespace only  zero extra cost
```

Architecture at a Glance



Our CRDs: The Data Model

KubeCon CloudNativeCon North America 2026

Aggregate quota

any level of the hierarchy

A limit shared by many namespaces. Tracks used and reserved across all of them.

async roll-up

`StorageQuota`

per namespace

Namespace total plus per-policy limits. status.total[] holds used and reserved per policy.

`StoragePolicyQuota`

per policy, per namespace

One storage policy's limit in a namespace: limit and appliedLimit.

`StoragePolicyUsage`

per policy × resource kind

Used and reserved for PVCs, VolumeSnapshots, VMs… updated as the CSI driver provisions.

i Reading the stack

Up: each level sums the one below, rolled up by controllers.

Admission: the webhook reserves against the namespace and any shared quota above it.

ii Extensible by design

Each usage object names the extension that sizes its resource kind (`resourceExtensionName`). A new kind plugs in without touching the webhook.

The Hard Part: Webhooks Have No Order

KubeCon CloudNativeCon North America 2026

BIGGEST CHALLENGE



How a phantom reservation happens

1. Our webhook reserves 10Gi
2. Another webhook denies the request
3. Nothing is persisted, so no controller ever releases it
4. **10Gi stays reserved until we repair it: a phantom**

1 Webhook order is not guaranteed

There is no defined order between webhooks of the same kind.

2 We can't know we run last

Another webhook can deny after we reserve. The request never persists: a phantom reservation.

3 Mutating webhook: less, not zero

It narrows the window, but multiple mutating webhooks have no ordering guarantee either.

4 Built-in ResourceQuota is special

A fixed-position plugin that runs after all webhooks, so it only charges requests that pass. We can't get that slot.

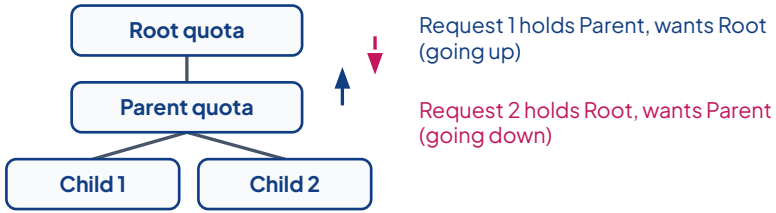
Challenges with Quotas Spanning Multiple Namespaces

KubeCon CloudNativeCon North America 2026

1 Circular lock wait (deadlock)

THE PROBLEM

Requests A and B lock the same parent and root in opposite orders, so each waits on the other.



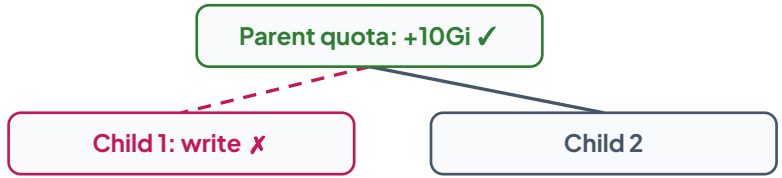
OUR ANSWER

One fixed order: lock child → parent → root, storage classes sorted. Writes go parent-first.

2 Partial failure across levels

THE PROBLEM

The parent write succeeds, the child write fails. Now the levels disagree.



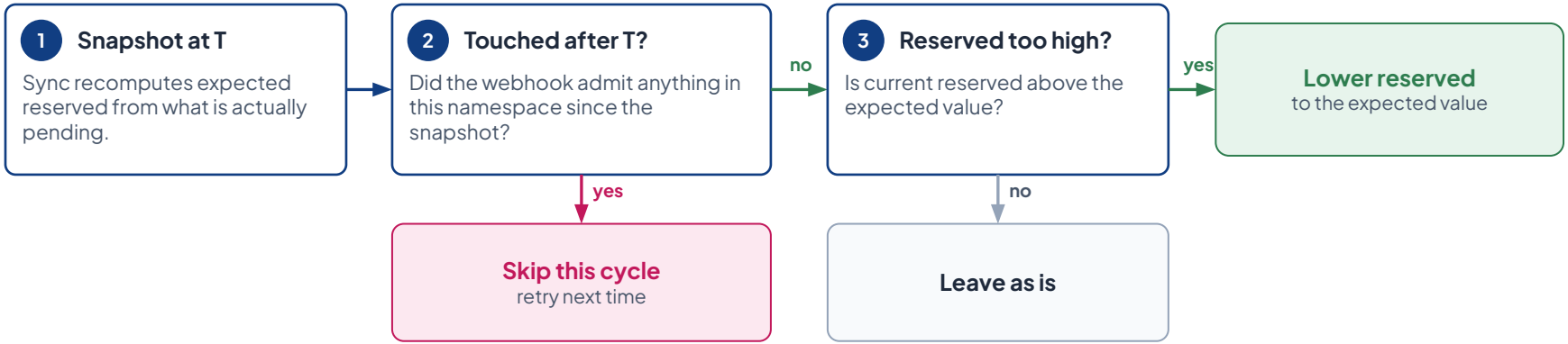
parent counts 10Gi, child counts 0: the levels disagree

OUR ANSWER

Fail toward over-reserving. It self-heals; under-reserving would let a limit be broken.

Drift Repair Can Clobber Live Reservations

KubeCon CloudNativeCon North America 2026



Only ever lowers

Raising from a stale snapshot could re-count something the CSI driver already released.

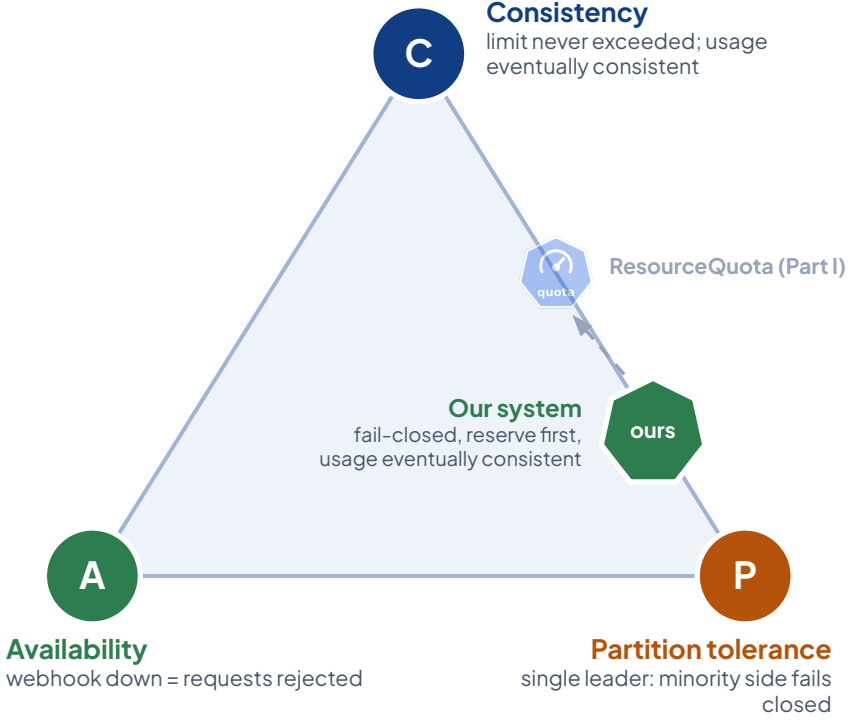
Skips busy namespaces

If the webhook admitted anything after the snapshot, wait for the next cycle. Late beats wrong.

Repair, not authority

The webhook stays the source of truth on the hot path. Sync clears leaked (phantom) reservations.

Our System Through the CAP Lens



Design choices mapped to CAP

Two-phase: Reserve, then Commit
Consistency: Reserve checks the limit before admitting
Eventual: async Commit means usage catches up later

Hierarchical lock ordering
Consistency: serialized writers keep parent totals exact
Availability: contended requests wait on locks

Tolerating unordered webhooks
Consistency: over-reserve rather than under-reserve
Availability: some requests rejected until repair

Timestamp-guarded drift correction
Consistency: usage converges, never raised from stale data
Eventual: repair waits for the next cycle

Why between C and P: if the leader or webhook is cut off, requests are rejected rather than guessed (we give up A). That keeps limits safe (C) and the system safe under partitions (P). Usage numbers lag, but the limit itself is never broken.

DEMO

Quota in Action

A PVC, then a VolumeSnapshot of it: charged, denied, and reserved becoming used.



KubeCon CloudNativeCon
North America 2026

Thank You!

