Surviving the Churn

Read Replicas and Controllers for Busy Kubernetes Clusters

Ryan Brewster, Anthropic

KubeCon + CloudNativeCon North America 2026

WHY WE'RE HERE

Kubernetes scales if you stay on the watch

Optimizations	A bunch of them allow larger and larger clusters.
Standard practice	Watch streams and informer caches have become the norm.
The catch	**You can scale to extremely large clusters, but it becomes more and more critical to stay on the watch stream.**

This talk is about how to do exactly that, even when you have very large, very active clusters with a wide mix of clients.

STABILITY

Scalability versus stability

Scalable

Looks fine under the load you planned for.

Stable

Diverse workloads keep finding new failure modes

- New controllers missing an index
- Large-scale misbehaving workloads saturating apiservers with allocations and GC pressure

Stability is avoiding the degradation spiral. The goal is a system robust to misbehavior.

STABILITY

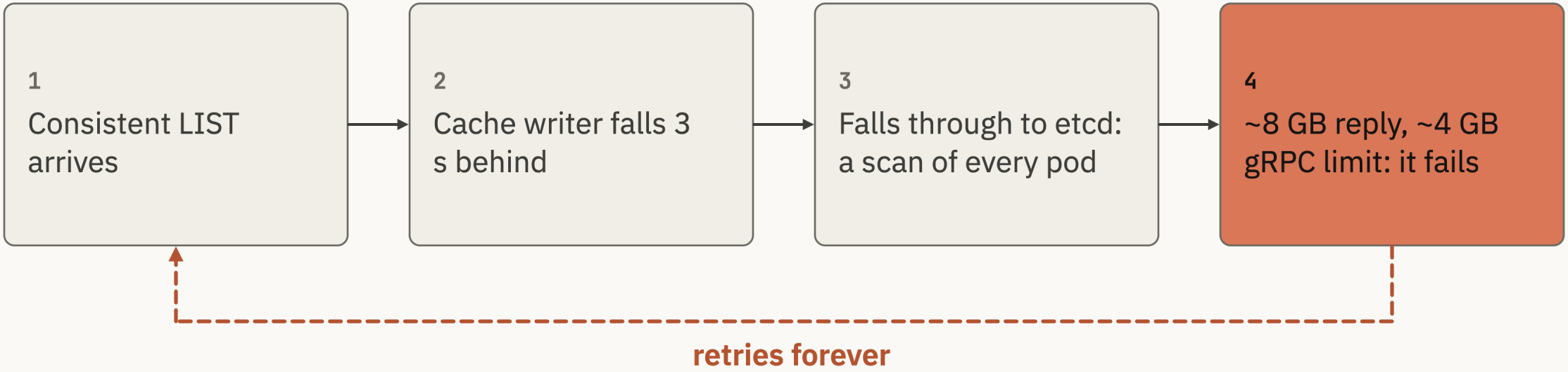
“Which pods are on my node?”

```
GET /api/v1/pods
  ?fieldSelector=spec.nodeName=$NODE
```

No resourceVersion: a consistent read.

STABILITY

Three seconds behind, it becomes an etcd scan



The 3 s is a wait for the cache to catch up. The selector doesn’t trigger the fallback. It makes the scan far worse, because etcd can’t index it.

STABILITY

Eight days later, a new client

Day 0	The agent triggers the scan, and the cluster is degraded for about 16 hours.
The fix	We set rv=0, and reads come from the cache.
Day 8	A new client tries to do pagination, which silently drops the rv=0, and the issue recurs.

Client fixes are whack-a-mole.

Optimizations make it scale. Structural isolation keeps it stable.

Protect the writer, and make catching up cheaper than starting over.

The watch cache is the cliff edge.

- As long as the cache keeps up and you can resume your watch, performance is okay.
- If you fall off, you are going to suffer. The best you can hope for is to recover without inducing a degradation spiral.

Why it’s a cliff

WHY IT'S A CLIFF

Busy has three dimensions

writes per second

Mutation rate

How fast history runs out.

bytes per event

Payload size

Fan-out bandwidth and GC pressure.

bytes across the cluster

Total size

What you re-read after falling off: the price of the cliff.

WHY IT'S A CLIFF

Aging out takes a minute. A kick takes seconds.

≈75 s of watch history, up to ~1,400 events/s on a resource

History resizes to hold 75 s of events, capped at 102,400 per resource.

A few seconds of disruption

Reconnect jitter, dropped packets, live-migrated VMs. You won't age out, but the apiserver can kick you. You resume.

→

Kicked enough, or a degraded apiserver

Your watch goes silent. After 60 s you disconnect and resume on a healthy one, if you're still in its history.

→

Fall off: 410 Gone, relist

Way more expensive than anything before it. A whole-cluster pod relist: ~14 GB of JSON, ~77 s even as protobuf.

WHY IT'S A CLIFF

Every component must tolerate your churn

Storage layer

Every write serializes through one ordered log.

10–16k

Apiserver

It maintains the watch cache and fans events out to every watcher.

Schedulers and controllers on client-go

Their informer caches have to keep up with the stream.

WHY IT'S A CLIFF

Load tests show scalability, not stability

Scalability: what it shows

Where you survive extremely stable load.

Stability: what it misses

Bursts

A single degraded apiserver

Reconnects after partial failure

A misbehaving client

Lock contention and controllers

LOCK CONTENTION AND CONTROLLERS

Informers guard their state with an RWMutex

```
func (c *threadSafeMap) List() []interface{} {
    c.lock.RLock()
    defer c.lock.RUnlock()
    …
    for _, item := range c.items {
        list = append(list, item)
    } …
```

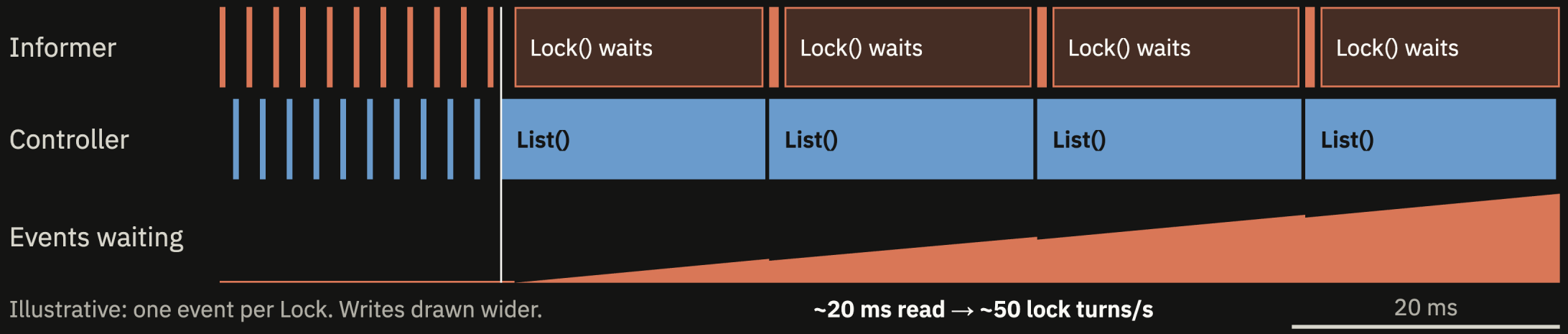
A pending writer blocks new readers.

Unlock admits the waiting readers first.

client-go/tools/cache/thread_safe_store.go, threadSafeMap.List

LOCK CONTENTION AND CONTROLLERS

Slow readers set the writer's pace



Informer falls behind → **Cache goes stale** → **Controllers act on a world that no longer exists**

LOCK CONTENTION AND CONTROLLERS

One stale cache, two failure modes

StatefulSet

Fixed names

A create hits 409 AlreadyExists, and the controller stalls: it stops reconciling and retries with backoff.

DaemonSet, ReplicaSet

Generated names

Expectations expire after 5 minutes, and duplicate pods add load: it thrashes.

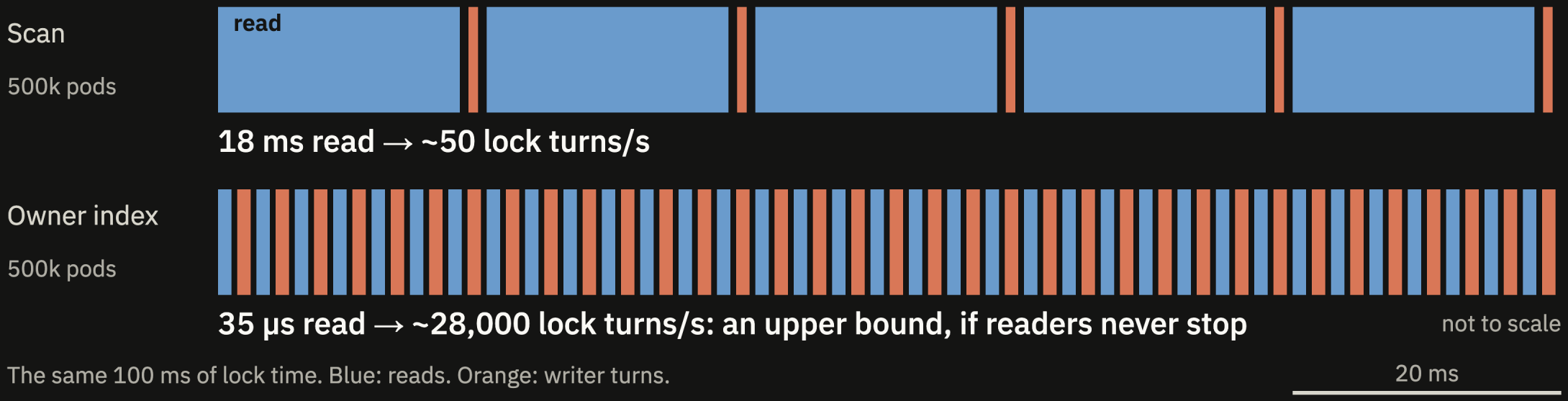
A fixed name makes etcd your backstop.

PATTERNS

Deterministic names	409 means you're behind	Per-object expectations with a TTL	Preconditions only on decisions that can flip

LOCK CONTENTION AND CONTROLLERS

An owner index keeps the writer alive



The same 100 ms of lock time. Blue: reads. Orange: writer turns.

10k pods	320 µs vs 30 µs
500k pods	18 ms vs 35 µs

EndpointSlice still scans: its label index was rejected as too risky.

kubernetes/kubernetes#130859

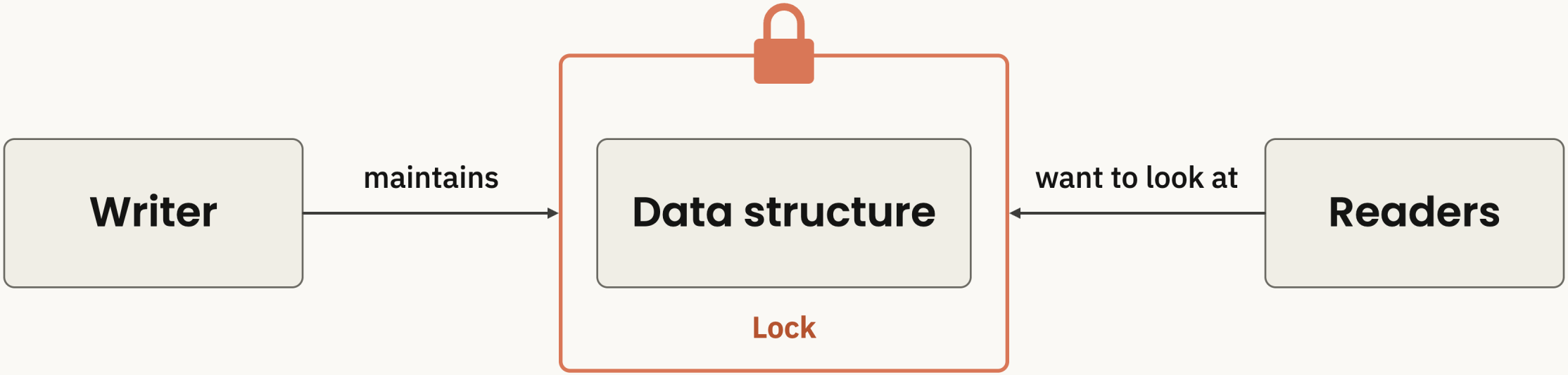
LOCK CONTENTION AND CONTROLLERS

This is whack-a-mole again.

Every new expensive read means weighing which indexes to add to avoid starving the writer. That’s inherent to the RWMutex approach.

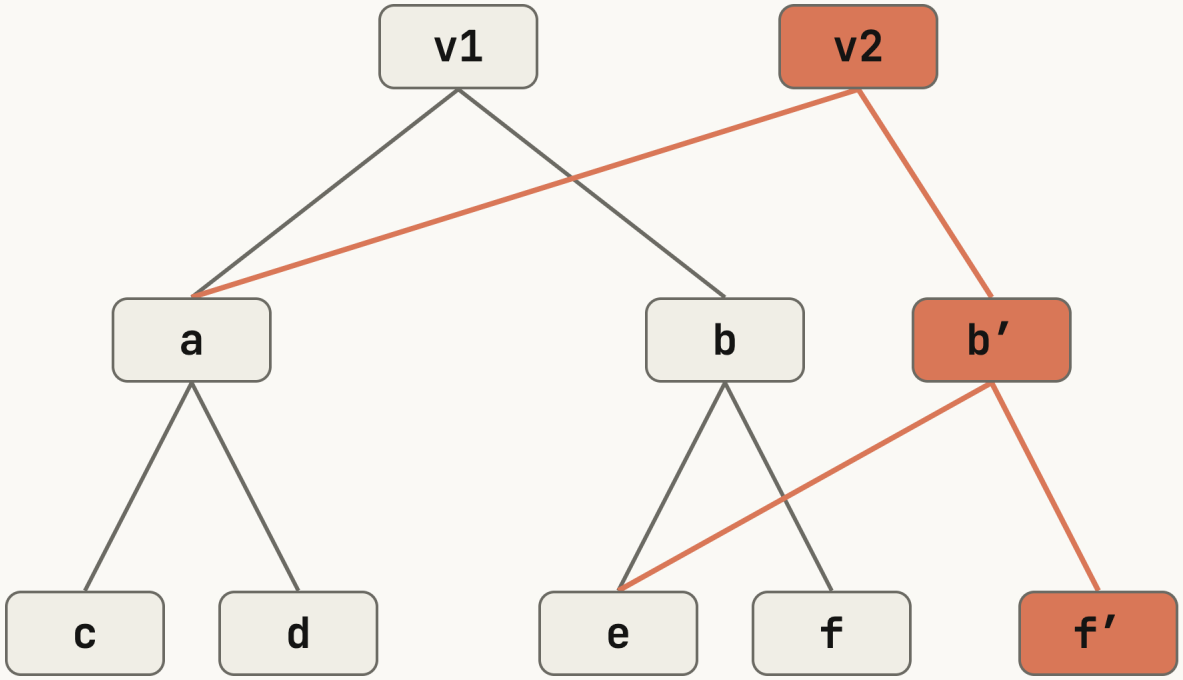
Indexes buy scalability. Stability means a new expensive read can’t starve the writer.

Why they have to synchronize



The writer maintains it. Readers want to look at it. So they synchronize.

Path copying: copy one path, share the rest



Clone is O(1). Publish swaps a pointer.

No synchronization. Readers get a ton of flexibility.

A slow or misbehaving reader costs CPU but can’t hold up the writer.

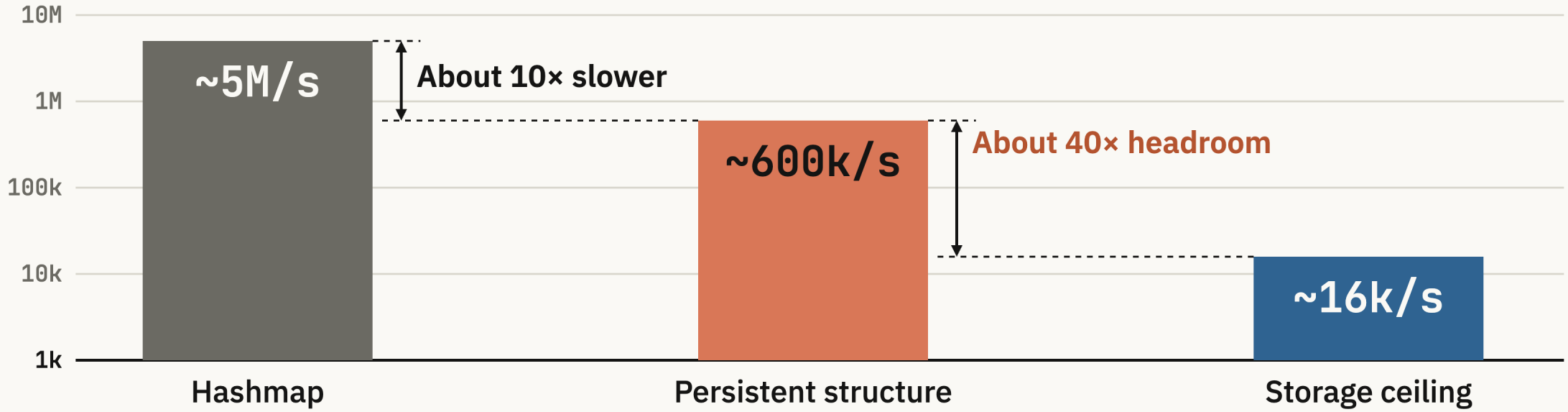
The writer side is slower.

Orange nodes are copied. The rest are shared.

LOCK CONTENTION AND CONTROLLERS

Much slower than a hashmap. It doesn't matter.

Writes per second, log scale. Baseline is 1k, not zero.

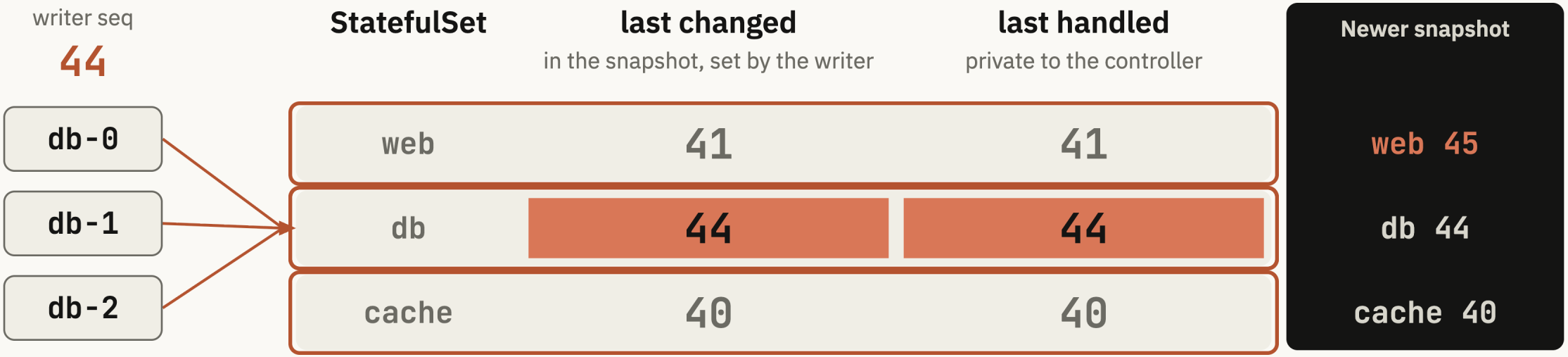


In exchange: structural isolation between writers and readers.

Hashmap and persistent structure: the structure alone, not the full pipeline, at about 1M keys. Storage: the ceiling.

LOCK CONTENTION AND CONTROLLERS

A slow controller is late, but never in the way



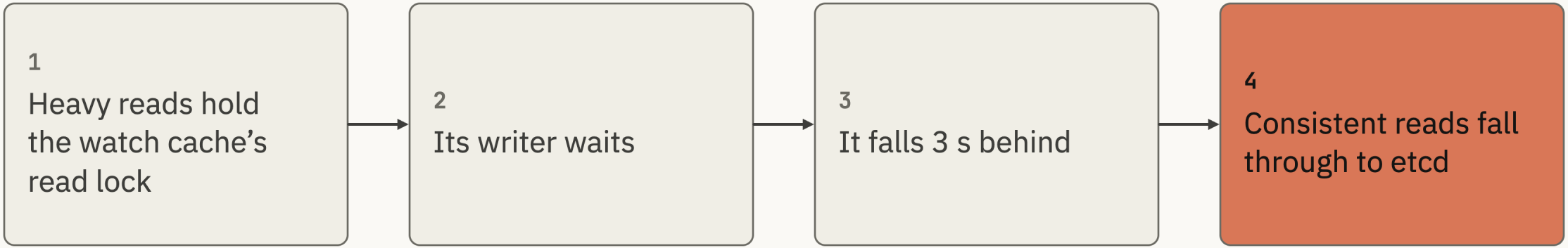
The writer never waited. Caught next pass.

In-tree precedent: kube-scheduler’s UpdateSnapshot skips nodes by generation.

The apiserver's own watch cache

THE APISERVER'S OWN WATCH CACHE

Same lock, one level up



That's what took us down.

1.37's cure: hold the read lock only to grab a copy-on-write snapshot.

Heading for 1.38: a lock-free pointer to the latest snapshot.

THE APISERVER'S OWN WATCH CACHE

Upstream keeps moving the edge

1.31	Consistent reads from cache, beta
1.33	Btree store GA
1.34	Snapshots serve paginated and exact-RV LIST. Consistent reads GA
1.37	LIST and WatchList stop holding the read lock for O(N)
1.38	Lock-free latest snapshot, in alpha. Stall-resume, targeted

As upstream improvements land, you can push an apiserver harder, and that lets you run larger clusters.

But large clusters increase at least two of the three threats: total size and mutation rate.

Unless you can tolerate a full reboot, plan for surviving your worst spike rather than steady-state load.

rv=0 has come from cache for years.

THE APISERVER'S OWN WATCH CACHE

The next cause is unknown

Proxy absorbs: reads, fan-out

Apiserver OOM

Heavy reads and fan-out can tip one over and cascade.

Needs throttling: write-driven

Crashlooping daemonset

The primary load is kubelets updating pod status.

Self-annotating pods

PATCHes cause a lot of GC pressure.

Fix the problem structurally.

Caching proxies

CACHING PROXIES

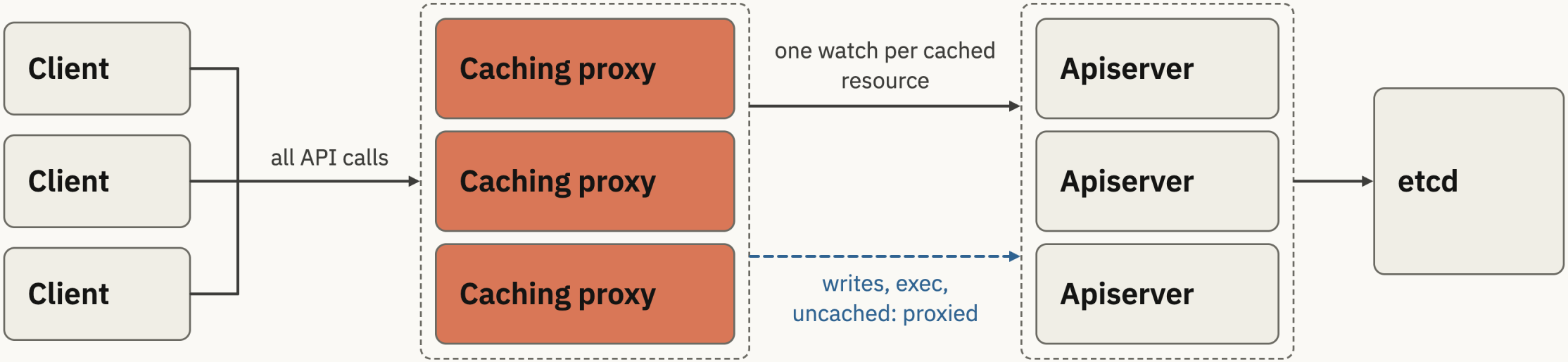
Normally, apiservers take every read and write



Your most expensive reads and your writes land on the same machines.

CACHING PROXIES

A caching proxy takes the reads that hurt



Apiservers keep headroom for writes.

Costs a few ms on reads.

Offloads the expensive shapes.

CACHING PROXIES

The war story, with three fixes

Stop holding the lock for O(N)

The upstream fix, 1.37

Move the heavy reads off the apiserver

Caching proxies

Etcd fallback: off

Next time the cause is different, and the writer is still protected.

CACHING PROXIES

Opt-in gets you started; opt-out gets the wins

Opt-in

Easy to roll out.

Incrementally shift clients over.

Opt-out

The big wins: every client uses the caching proxy by default, even new clients.

How: repoint default/kubernetes.

The cost: once every client depends on the proxy by default, it stops being sacrificial. Losing all the proxies took kubelets down with them.

CACHING PROXIES

Bursts need a plan on one side

Steady state is what you can survive. Events inevitably cause bursts of mutations.

Tolerate the spike

on the read side

Caching proxy

Long history

Stalling watchers

Smear it out

we throttle writes

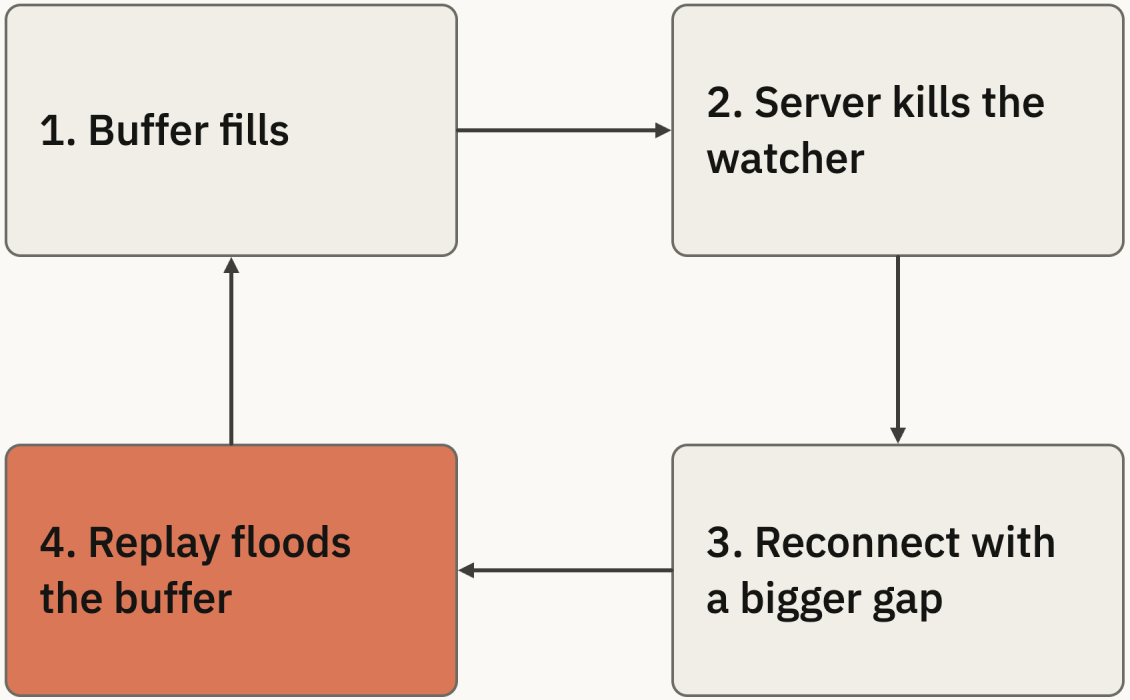
Rate limits

Concurrency bounds

Slow clients

SLOW CLIENTS

Kicking slow watchers feeds the spiral



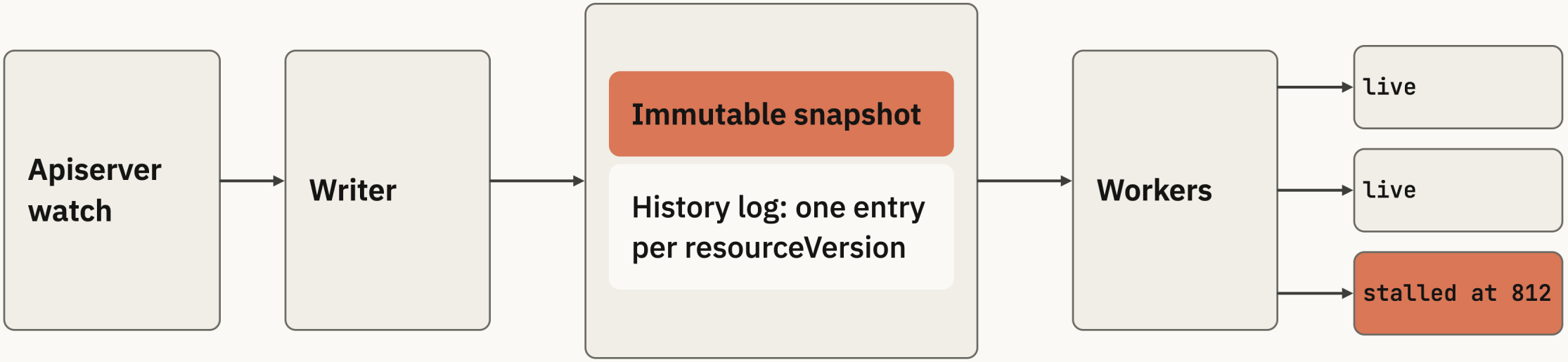
The natural design: a per-client buffer that kills the client when full.

The client reconnects and resumes, so the replay backlog refills the buffer that just got it killed.

Each kill makes the next gap bigger.

SLOW CLIENTS

The cache is the replay buffer



One list+watch per kind

One writer per resource; publish is a pointer swap

The same history serves resumes and LIST at an old resourceVersion

Read the cache once, dispatch by index

Each client is a position in that log

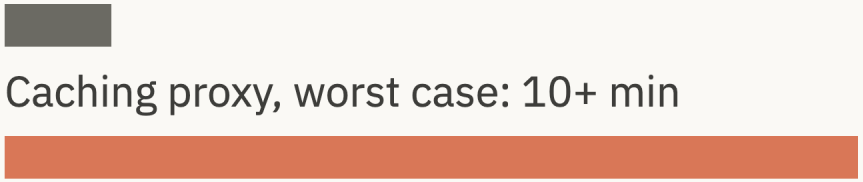
SLOW CLIENTS

History is what the extra RAM buys

75 s → 10+ min

Worst case

The default apiserver history target vs the proxy's worst-case window.



No GC multiplier

The heap can actually use the RAM.

Cache what hurts

Only resources that are read-heavy and churning.

Share unchanged bytes

An unchanged spec is stored once.

SLOW CLIENTS

Stall first: a slow client is a cursor



A stalled client holds a position in history, and no buffer.

We still kill on falling off history, an upstream relist, and 60 s of zero TCP window.

One stalled client: five healthy watchers force-closed ~75 times in 30 s.

Synthetic benchmark, stock 1.37: 5k events/s, 4 KiB objects, not Pods.

SLOW CLIENTS

KEP-6268 brings stall-resume upstream

A healthy watcher’s p99 goes from 24 ms to 52 µs next to 1,000 stalled watchers.

KEP-6268

WatchCacheStallResume

k/k#141450

The PR, still open

Alpha

Off by default, targeting 1.38

What’s still broken

WHAT'S STILL BROKEN

Three problems we haven't solved

The fallback	By default, a LIST a lagging cache can't serve falls through to etcd. We run with the upstream gate on: a retryable 429 instead. Alpha in 1.37, off by default.
Giant informers	Too big for our proxy: ~25 MB/s JSON decode on one goroutine times out.
Recovery and chained writers	Partial failure is worse than steady state. The proxy's feed comes from the apiserver's watch cache: if that starves, so does the proxy.

kubernetes/kubernetes#138701

WRAP-UP

What to take home

Optimizations make it scale. Structural isolation keeps it stable.

1 **Alert on falling off**
410 and relist rate, etcd fallbacks, watch cache and informer lag.

2 **Run 1.37 and throttle write bursts**
Snapshot reads are on by default (ListFromCacheSnapshot). Rate-limit your bursty writers.

3 **Try the fallback gate; we run it. Watch the KEP.**
k/k#138701 and KEP-6268.

Optimizations make it scale. Structural isolation keeps it stable.

Code keeps changing faster, and the only way to tolerate that velocity is to ensure systems stay stable when things go wrong.

Questions?

KEP-6268 · k/k#138701

Thanks to sig-api-machinery.

Backup

BACKUP

Table stakes, in one slide

RAM and GOGC

To keep GOGC=400's savings, live heap must stay under ~20% of the memory limit.

API Priority and Fairness

Rules ready to change live, with a privileged lane for operators.

Split storage

Events, Leases, Nodes, Pods, any busy resource.

Client throttling

Rate limits and concurrency bounds. Controllers behave better self-limited than queued.

BACKUP

Who pays to free old snapshots

Publishing every event pays for path copies. Batching pays for one big free. Choose who gets the bill, and cap the batch.

`2.6 → 116 µs`	p99.9 at batch 256, in an older tree
`256`	our production batch cap
`GC`	in Go, that’s who gets the bill

BACKUP

Writer stamps, reader remembers: the loop

```
on_event(e):
  seq += 1
  modified[owner(e)] = seq
  publish(snapshot)

every tick:
  for k, m in load().modified:
    if m > done[k] or timer_due(k):
      reconcile(k)
```

BACKUP

Four questions before you build one

Which resources?	Read-heavy, churning, low-sensitivity. Secrets aren’t cached.
Authn and authz?	Scoped to low-sensitivity resources. Everything passed through stays the apiserver’s call.
Routing?	Stream-aware pool: LIST and long WATCH aren’t ordinary requests. Move off lagging apiserver replicas.
Which clients?	Opt in by URL, or repoint default/kubernetes.

BACKUP

Copy-on-write in client-go

Prototyped and dropped

Claimed: writes 2–14×,
memory 6–100×.

The cost is paid at every scale; the problem exists only at large scale.

Upstream went with indexes plus batching: ~30× writes.

My answer

Those benchmarks can’t show contention.

At this scale, the price is worth paying.

kubernetes/kubernetes#133241, #132240

BACKUP

Route around the controller



- Relist without clearing: bump a generation, sweep at the end.
- A slow client holds a cursor.
- Tens of ms to deliver at p99.

Benchmark the tail: one wide object took out a writer (mean ~1, tail ~55k).

BACKUP

DNS: remove a hop

Before



After: by pod name, ~3 GiB per replica



- Every replica decodes the whole pod stream.
- No readiness: any pod with an IP resolves.
- Staleness is yours, so alert on watch freshness.

BACKUP

Reclaiming snapshots without a GC

A reader’s pointer load and refcount increment are two steps, and the writer can free in between. So readers announce themselves in epoch-style counters.

x86	ours	arc-swap
Store, 16 readers	1.9 µs	5.0 µs
Store, 64 readers	6.3 µs	22.9 µs
Scoped read	11.6 ns	12.7 ns