KubeCon CloudNativeCon
North America 2026

How Anthropic Almost Melted GKE (And the PRs That Saved It)

Salt Lake City, Utah, United States



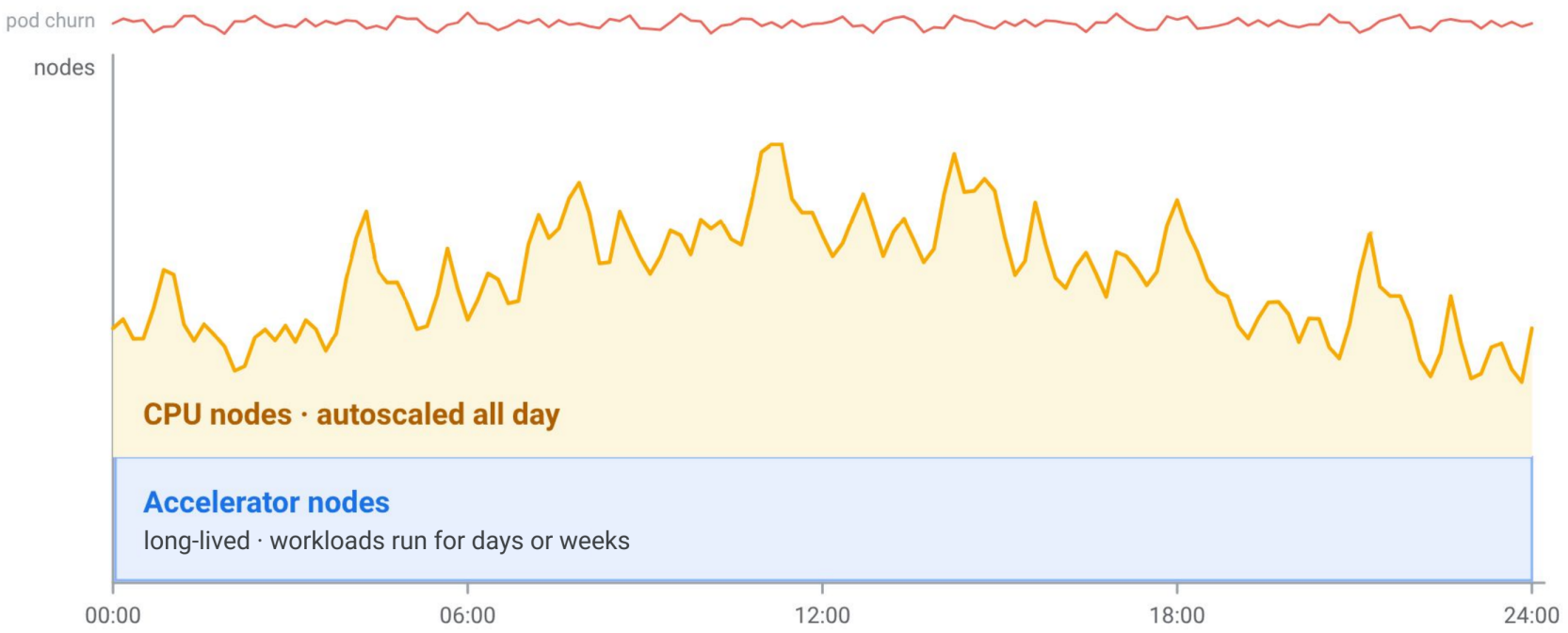


Maciek Różacki Jian Cheung

AWS is Anthropic's primary cloud provider

Myth 1: “AI clusters are static.”

Long-lived accelerators share one control plane with a CPU fleet that churns all day.



illustrative shape, not measured data

Myth 2: “etcd melts first.”

A Spanner-based store kept up, so the pressure moved up the stack into the recovery loops.

What we expected

Node agents
kubelet · CNI · device plugins

kube-controller-manager
one process, many controllers

kube-apiserver
APF admission · watch cache

etcd melts?
the usual suspect

What we saw

Node agents
re-sync storms after restarts

kube-controller-manager
locks, hot loops, queues

kube-apiserver
admission + recovery loops

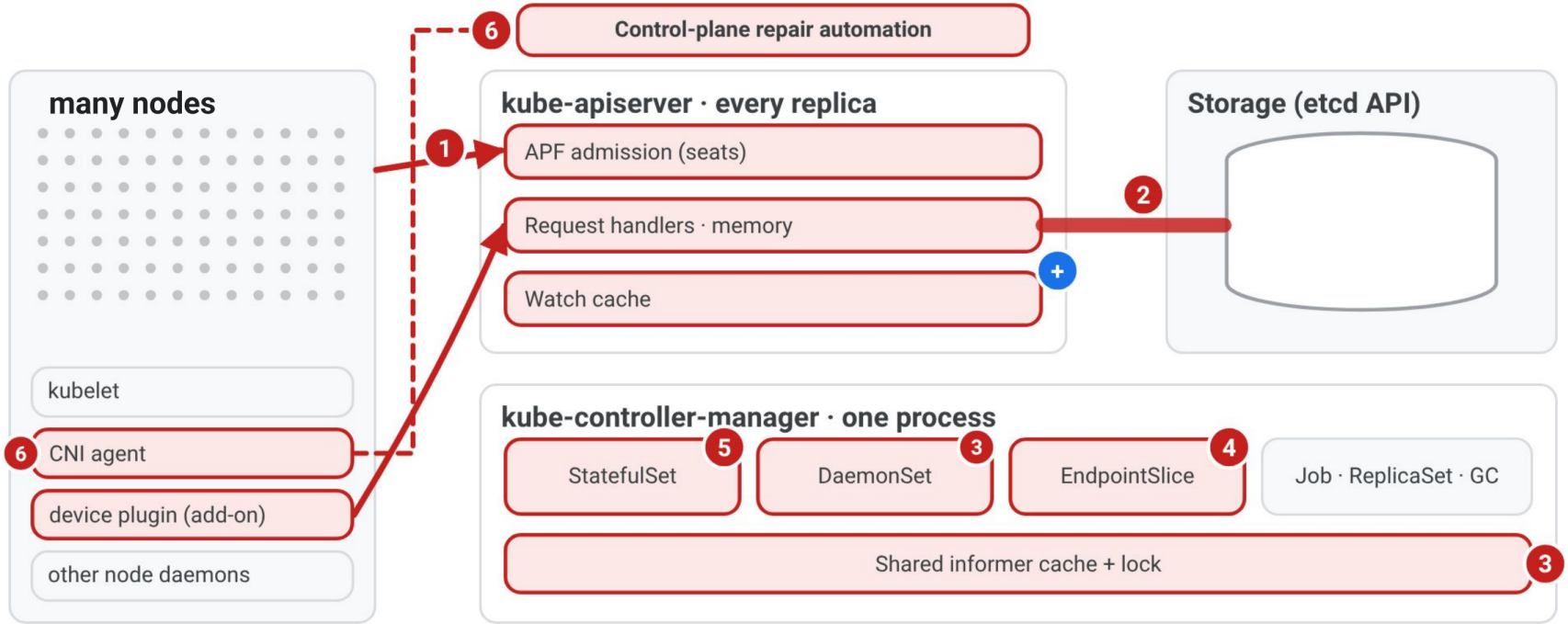
Spanner-based store (etcd API)
kept up: 13,000 lease updates/s

pressure moved up

13,000 lease updates/s: GKE's public 130,000-node experiment

One control plane. One very large cluster. Shared fate.

Six stories on one map: each is a recovery loop that lost its boundary.



1 The underpriced LIST · 2 The 8 GB GET · 3 The hot loop · 4 Services × pods · 5 One 404 · 6 The restart storm · + What comes next

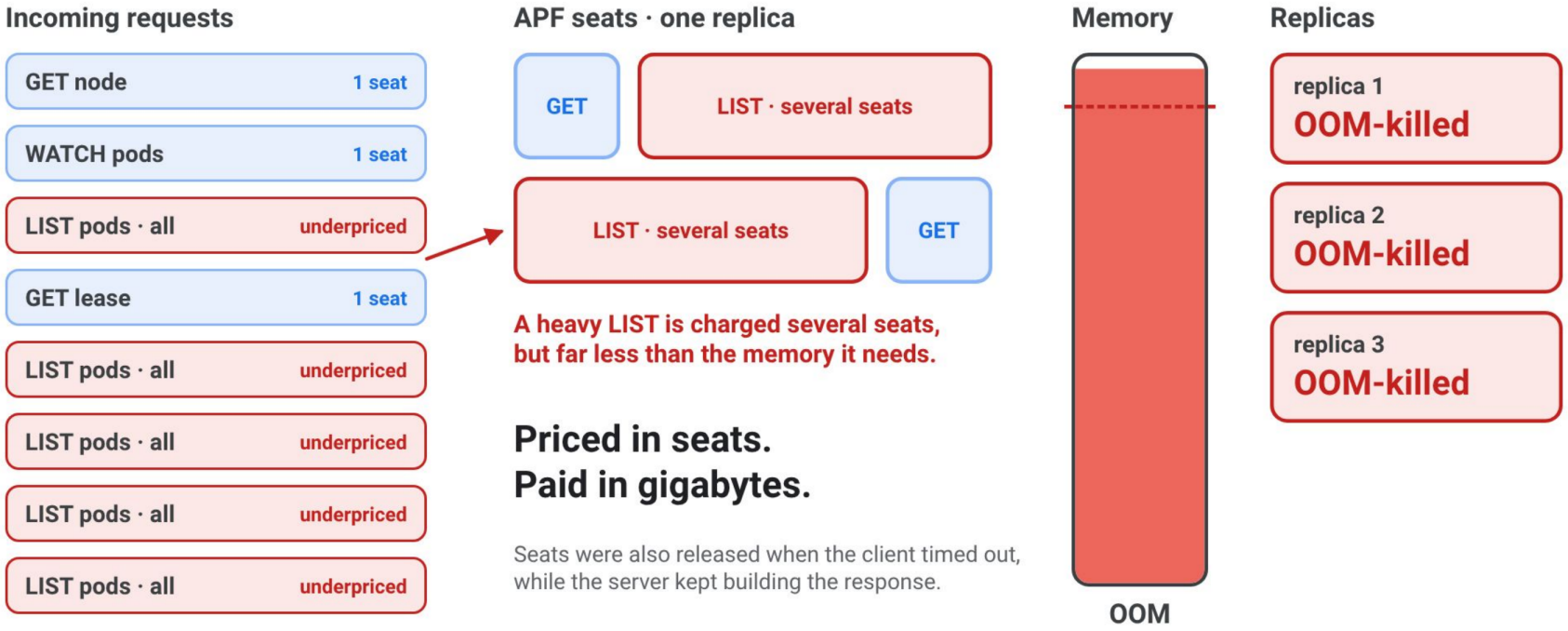
STORY 1

The underpriced LIST

Overload protection that mis-priced its heaviest request.

APF prices LISTs. We hit its corner cases.

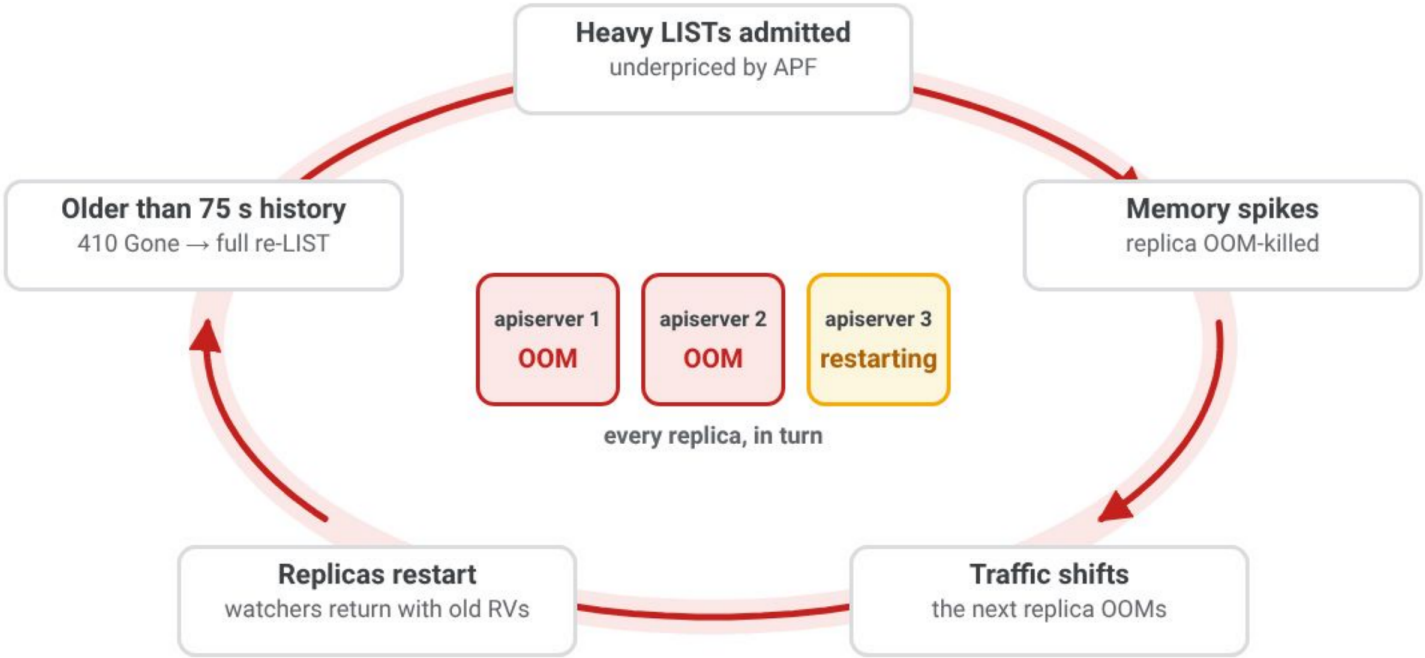
APF has priced LISTs by work since 2021; corner cases still made multi-GB LISTs look cheap.



illustrative: not exact seat counts

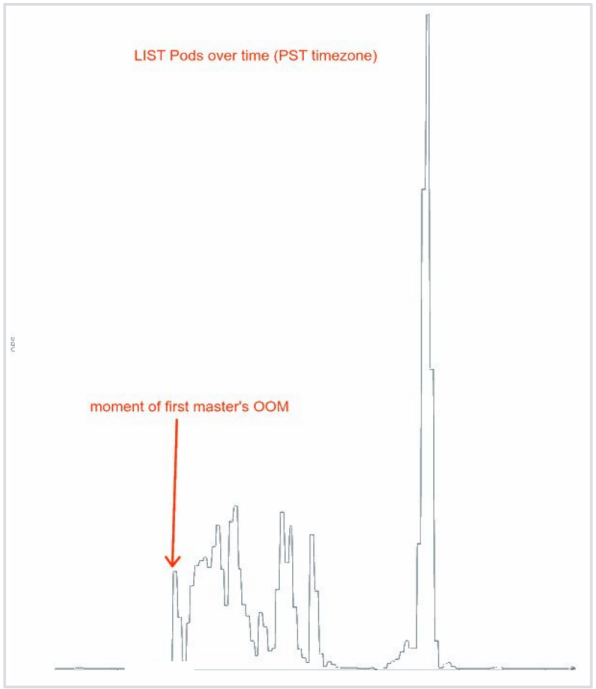
Then recovery made it worse

Replicas OOMed in turn; returning watchers fell outside the 75-second history and re-LISTed.



The relist storm, as Google saw it

LIST pods requests per second, late 2024. Google's chart, Google's annotations.



THE FIRST OOM

LIST pods traffic appears, and in the same minute the first control-plane replica runs out of memory. Before this, almost nobody was listing pods.

THE WAVES

Each time a replica comes back, its clients start their informers again and list every pod. Each of those LISTs was priced at one seat, so admission control let them all in.

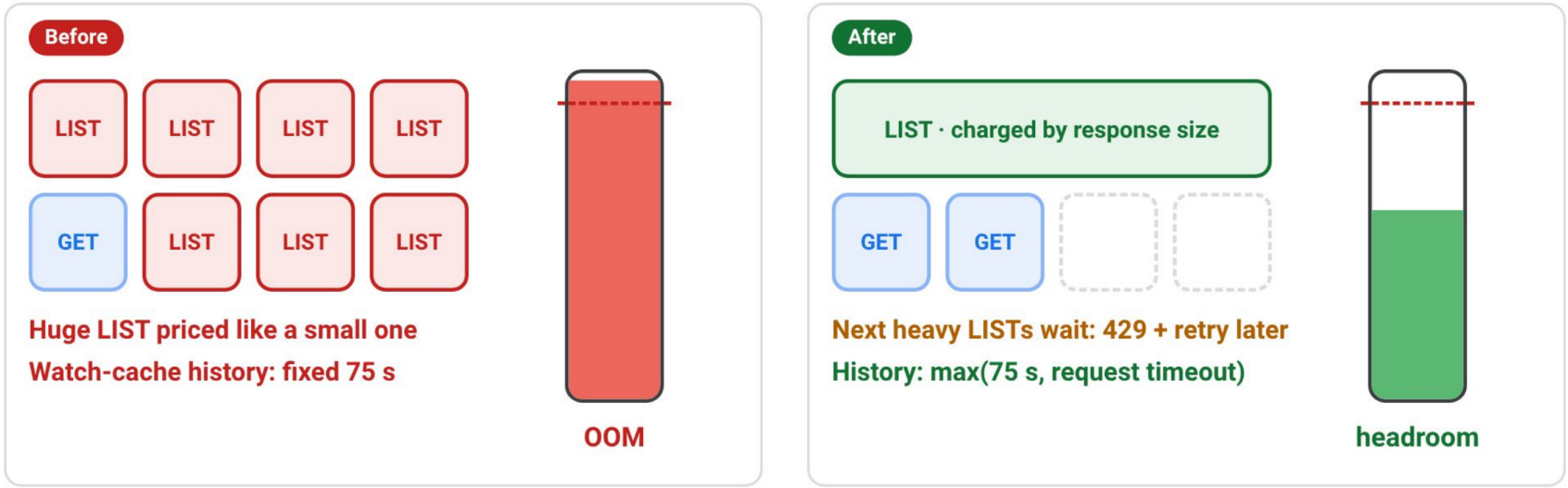
THE BIGGEST WAVE

An hour in, the biggest wave, several times the size of the first. Each LIST is multi-gigabyte work for the server.

Every restart sent the clients back for a full LIST.

Price by real cost. Size history to the timeout.

Two boundaries: honest admission, and a watch-cache window that clients can recover into.



k/k#132932 + #133034 · LISTs priced by size · Google · v1.34

k/k#129205 · history sized to the timeout · Google · v1.33

Admission control is only as good as its cost estimate.

STORY 2

The 8 GB GET

One missing query parameter, on every node.

You asked for one node's pods. It read all of them.

A consistent read normally comes from the watch cache. The cache had fallen behind, so the read fell through to storage.

```
GET /api/v1/pods?fieldSelector=spec.nodeName=<node>
```

no resourceVersion

Node add-on
Runs on every node. Asks only for the pods on its own node.

→

API server
No resourceVersion means a consistent read. Normally answered from the watch cache.

→

The cache was behind
The read waited 3 s for the cache to catch up, then fell through to storage.

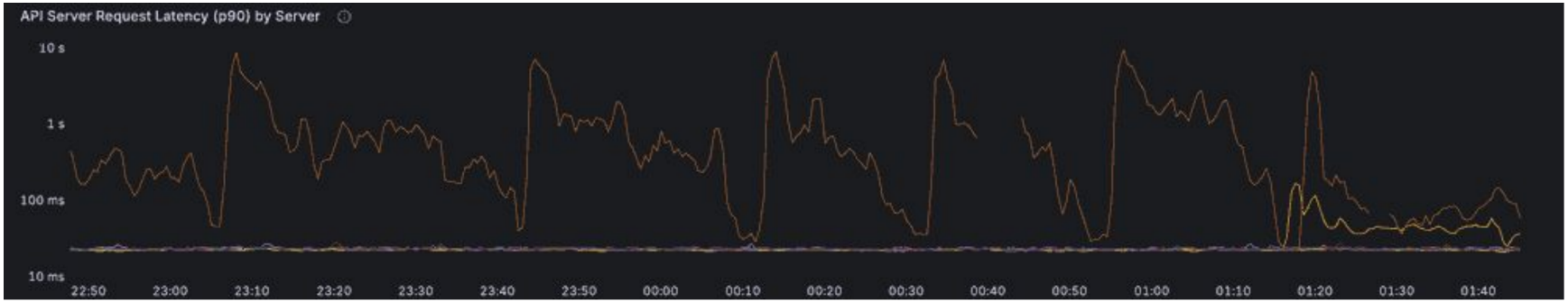
→

Storage returns every pod
≈ 8 GB to return a handful of pods. Over the gRPC limit, so it fails. The client retries.

Pods read	Data moved	Pods wanted
all of them	**≈ 8 GB per read**	**a few**

One API server. The rest never noticed.

p90 latency for each API server replica over three hours, log scale. Real dashboard, early 2026.



THE ORANGE LINE

One replica swings between 100 ms and 10 s, again and again. Its watch cache was behind, so every consistent read it took became a full scan of storage.

THE FLAT LINE

Every other replica, at about 25 ms the whole time. All of the errors came from the one bad replica too.

THE RECOVERY

A controller-manager failover takes load off that replica and it starts to recover. We found the root cause after that, not before.

Graph every replica, not the average.

Fix the client. Then make the server say no.

resourceVersion=0 fixed that add-on. Eight days later a different client fell off the same cliff.

1 · FIX THE CLIENT

Per-node agents read with resourceVersion=0, then watch. Served from the watch cache's node index: kilobytes, not gigabytes.

No upstream PR: it was one line in the add-on.

2 · GUARD THE SERVER

When the cache is behind, answer with a retryable 429. Never fall through to a full scan of storage.

k/k#138701 · alpha in v1.37, off by default.

We run with this gate on.

3 · THE PR THIS STORY PRODUCED

A paged LIST from the cache no longer copies the whole key range.

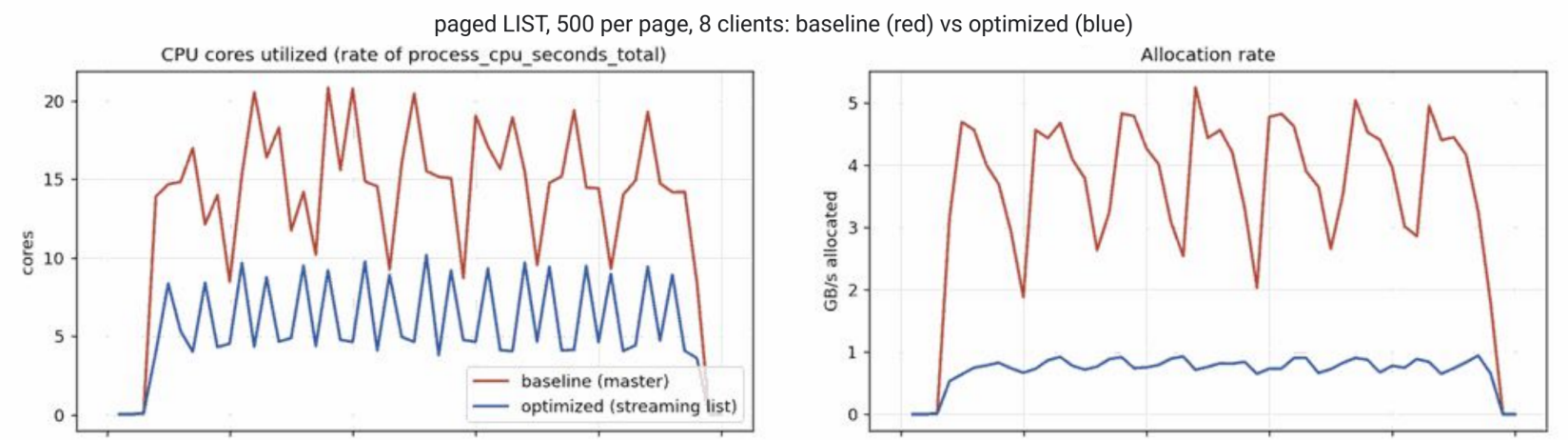
k/k#140896 · Anthropic PR, approved by Google · v1.38.

Found while reading that code. Deep dive: Ryan Brewster's talk.

Client fixes are whack-a-mole. Put the boundary in the server.

The PR this story produced: k/k#140896

Same paged-LIST load, before and after. Synthetic test rig: 8 clients paging through pods 500 at a time.



CPU
About 15 to 20 cores before. About 4 to 10 after, for the same request rate.

ALLOCATION
About 4 to 5 GB/s before. Under 1 GB/s after.

THE LIMIT
Paged LISTs only. An unpaged LIST costs the same as before, and paging is a rare pattern.

Read only what you return.

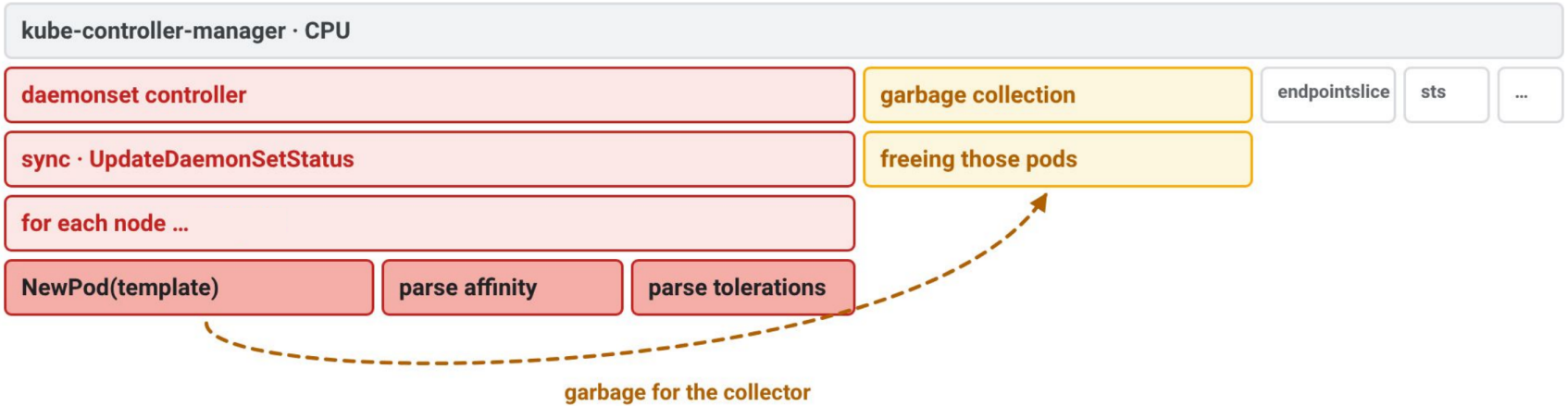
STORY 3

The hot loop

One loop in one controller, burning more than ten cores.

The DaemonSet controller rebuilt a pod for every node

Live control-plane CPU profiles showed one loop burning more than ten cores.



DAEMONSET CONTROLLER, LARGE CLUSTERS

>10 cores

the same pod rebuilt for every node, for every DaemonSet, on every sync

HOW IT WAS FOUND

Live CPU profiles from a production control plane

one loop stood out; the fix was a few lines

illustrative flame graph; widths are not measured

Hoist it out of the loop

Parse once per DaemonSet, check once per node: a few lines, upstream for everyone.

Before

```
for _, node := range nodes {
  pod := NewPod(ds, node.Name)
  tol := tolerations(pod)
  aff := nodeAffinity(pod)
  if shouldRun(node, tol, aff) {…}
}
```

After · simplified

```
tol := ds.Spec.Template.Spec.Tolerations
aff := nodeAffinity(ds.Spec.Template)
for _, node := range nodes {
  if shouldRun(node, tol, aff) {…}
}
```

−97%
CPU · benchmark, 10k nodes

−99.99%
allocations · same benchmark

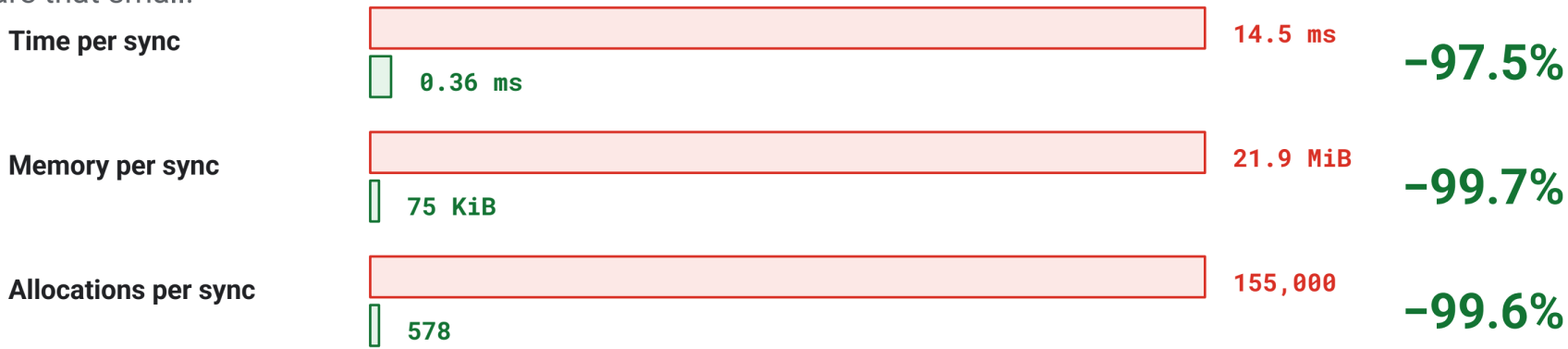
40–60% less total controller-manager CPU expected in large clusters
author's estimate from live profiles, not a measurement

k/k#140167 · Anthropic · approved by SIG Apps · v1.37

At large scale, every per-node allocation is a fleet-wide cost.

The benchmark says 97%. Real clusters: about half.

k/k#140167, Go benchmark of one DaemonSet sync, before and after. Bars are to scale: the green ones really are that small.



THE HONEST NUMBER

The benchmark isolates this one loop, so it overstates the effect. From live profiles of large clusters, the author estimates 40 to 60% less controller-manager CPU. We have no production before-and-after measurement, and we say so.

Benchmarks find the loop. Profiles tell you what it is worth.

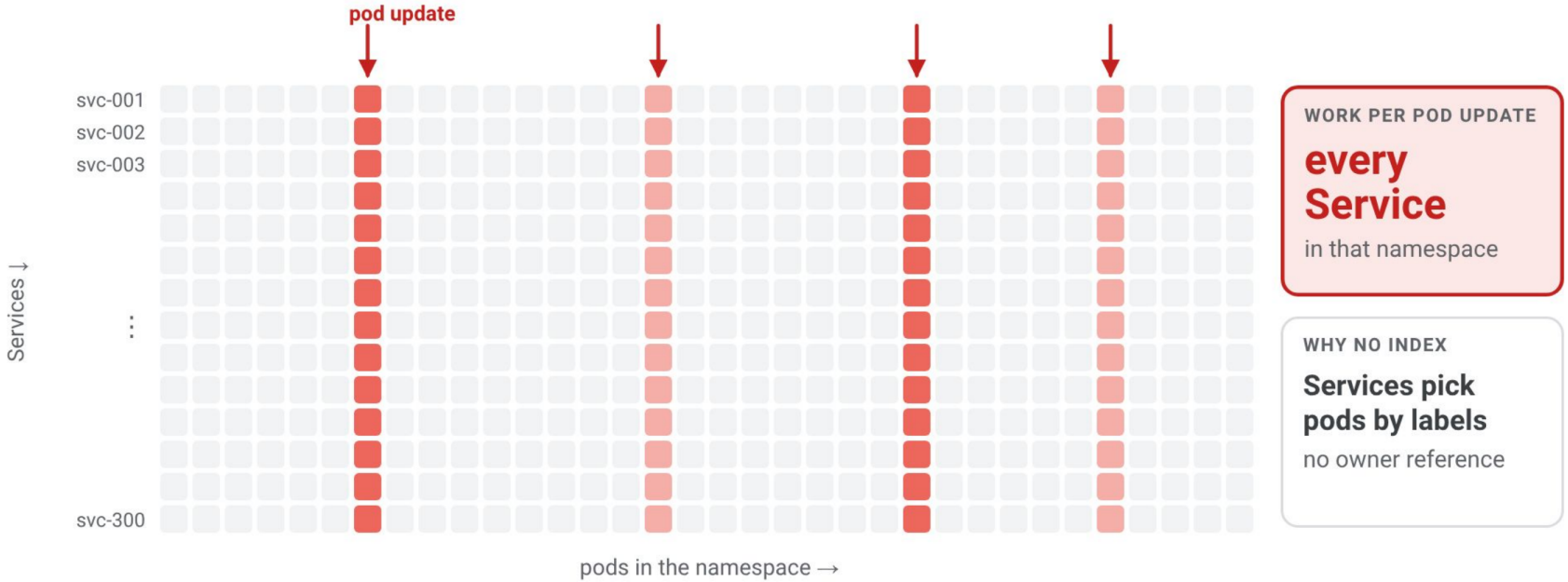
STORY 4

Services × pods

Work that grows with every Service and every pod.

Every pod update was checked against every Service

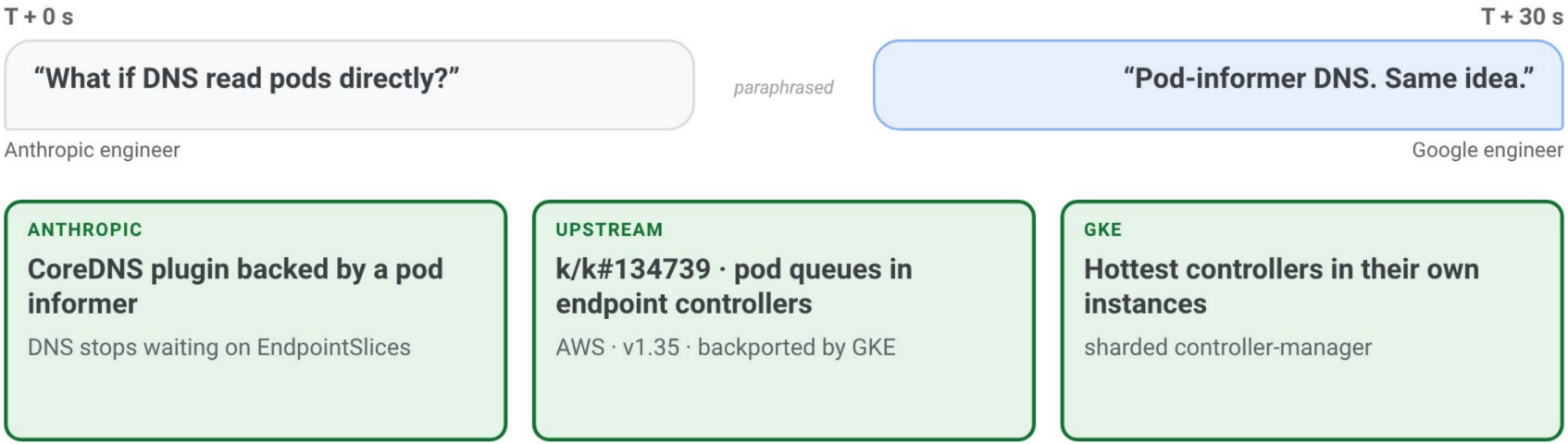
Services select pods by labels, so the EndpointSlice controller had nothing to index on.



Hundreds of Services in one namespace (one per replica) × constant pod churn: the EndpointSlice handler falls behind and other controllers wait in the same process.

Two teams, one idea, thirty seconds apart

In the shared channel, both sides proposed the same workaround within half a minute.



None is the architectural fix. All three shrink the blast radius tonight.

When the real fix is hard, isolate the blast radius first.

STORY 5

One 404

A pod that was already gone stopped a whole scale-down.

One 404 stopped the line

A pod that was already gone made the StatefulSet controller abort its whole scale-down.




```
controller: DELETE web-9  → 200 OK
controller: DELETE web-8  → 200 OK
controller: DELETE web-7  → 200 OK
controller: DELETE web-6  → 404 NotFound → sync aborted, requeue (backoff 1)
controller: DELETE web-6  → 404 NotFound → sync aborted, requeue (backoff 2)
controller: DELETE web-6  → 404 NotFound → sync aborted, requeue (backoff 3)
```

podManagementPolicy: Parallel → one phantom pod can block thousands

What a stalled scale-down looks like

Replica skew: how far every StatefulSet is from its requested size. Real dashboard, early 2026.



The controller was not slow. It was stuck.

Make delete idempotent

Anthropic's upstream fix treats a 404 on delete as done, and the scale-down keeps going.

simplified

```
- if err := deletePod(pod); err != nil {
-     return err            // one 404 aborts the whole sync
- }
+ if err := deletePod(pod); err != nil && !apierrors.IsNotFound(err) {
+     return err            // 404 = already gone = success
+ }
```



scale-down completes; the phantom pod is simply skipped

k/k#136812 · Anthropic · v1.36 · merged Feb 2026

the DaemonSet controller already handled NotFound this way

“Not found” during a delete is success.

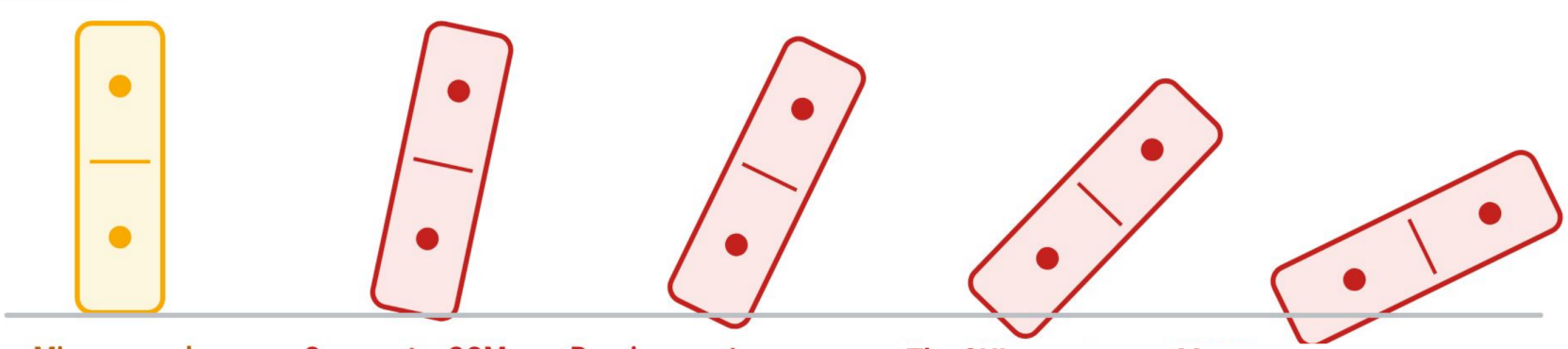
STORY 6

The restart storm

When recovery itself was the problem — and how we fixed it.

The restart storm: how it started

A routine upgrade, a control plane near its memory limit, and repair built for smaller clusters.

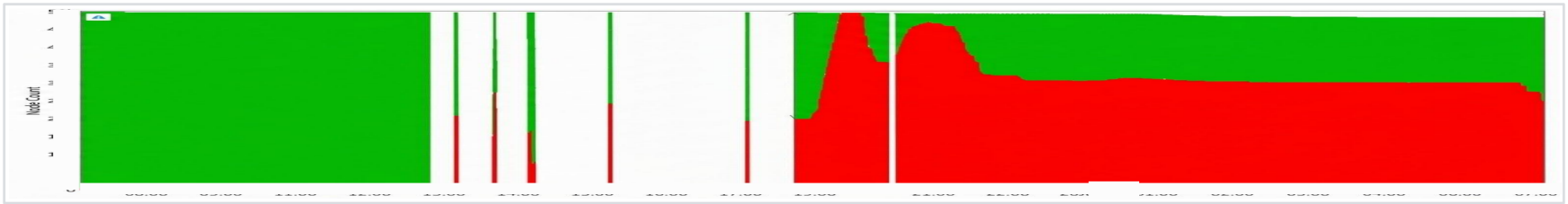


Plus a compounding bug: a VPA recommender without working leader election, so every replica watched every pod.

Healthy, no data, stuck: Google's view

Node count by health, early 2026. Google's dashboard, because ours had no data either.

GREEN = healthy RED = not healthy WHITE = no data



ALL HEALTHY
Every node healthy, until the control plane overloads.

NO DATA
White is not good news. The control plane is too overloaded to report anything. The thin bars are the few moments a scrape got through.

BACK, THEN LOOPING
Metrics return and most nodes show as not ready. They start to recover, then restart again.

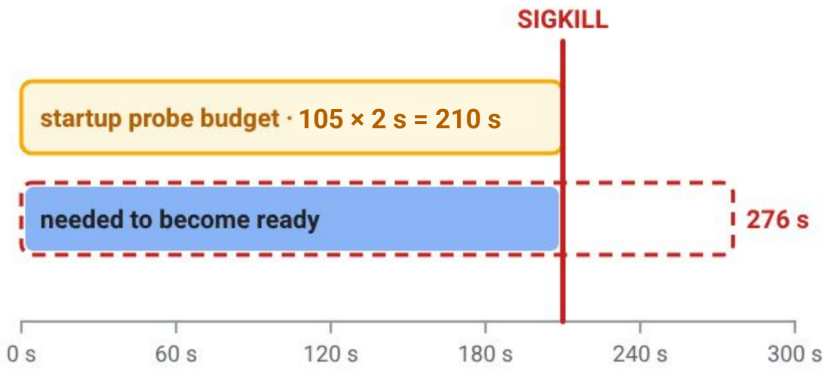
STUCK
Node health stops changing: agents keep restarting and none get further. The dip at the far right is the first fix starting to work.

A flat line means something is looping, not recovering.

210 seconds allowed. 276 seconds needed.

Most CNI agents were killed by their own startup probe and started over, all in lockstep.

One node · CNI agent start-up

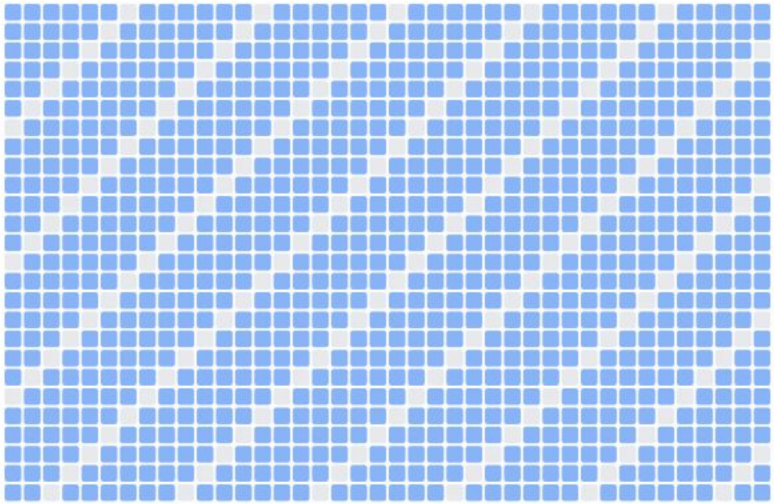


attempt #3

76% ready → killed, start over from 0

The probe budget was sized for a smaller cluster.
Start-up work grew with the cluster.

Most nodes · in lockstep



**every agent restarts at the same moment,
so the load on the control plane never drops**

What would you do?

Show of hands. The control plane is overloaded and the node agents keep restarting.

A Raise the probe timeout and roll it out

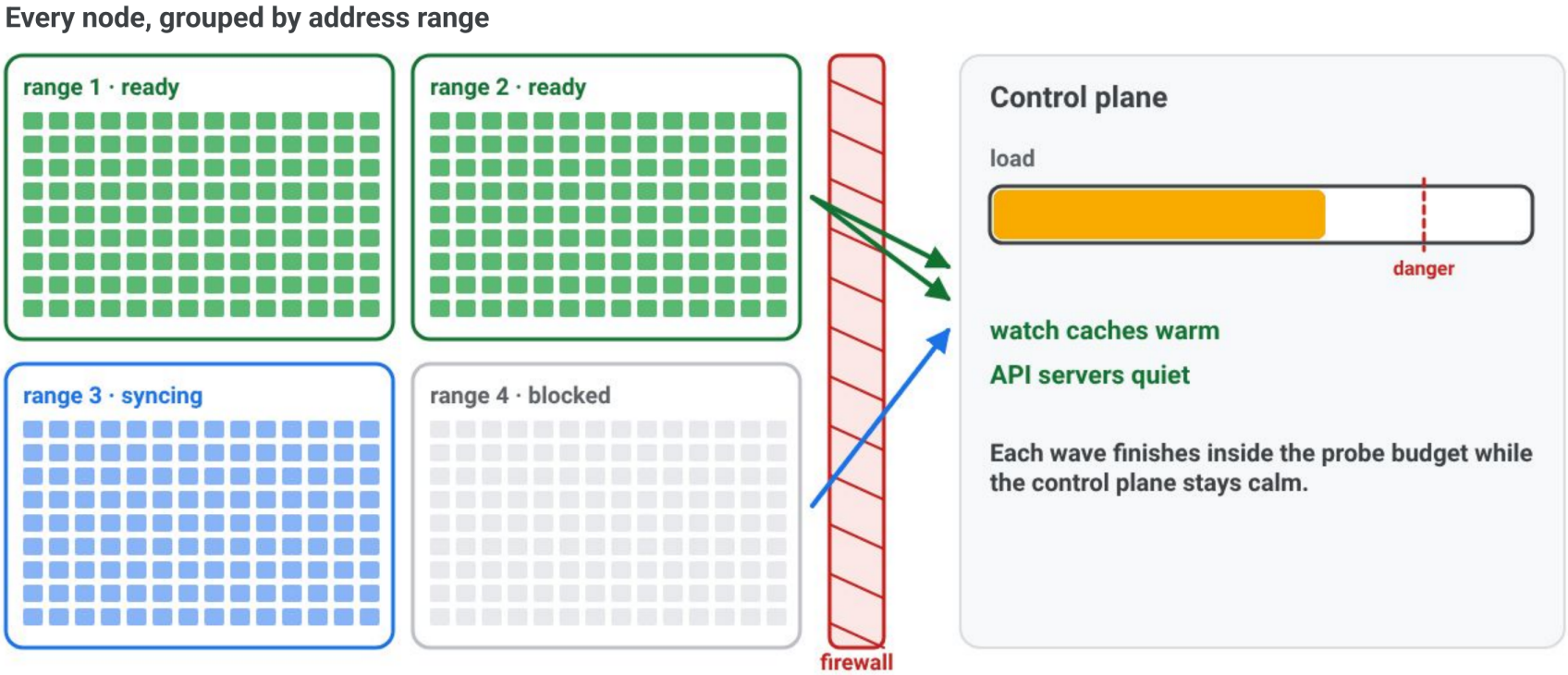
B Add control-plane capacity

C Roll back the upgrade

D Cut the nodes off. Let them back in waves.

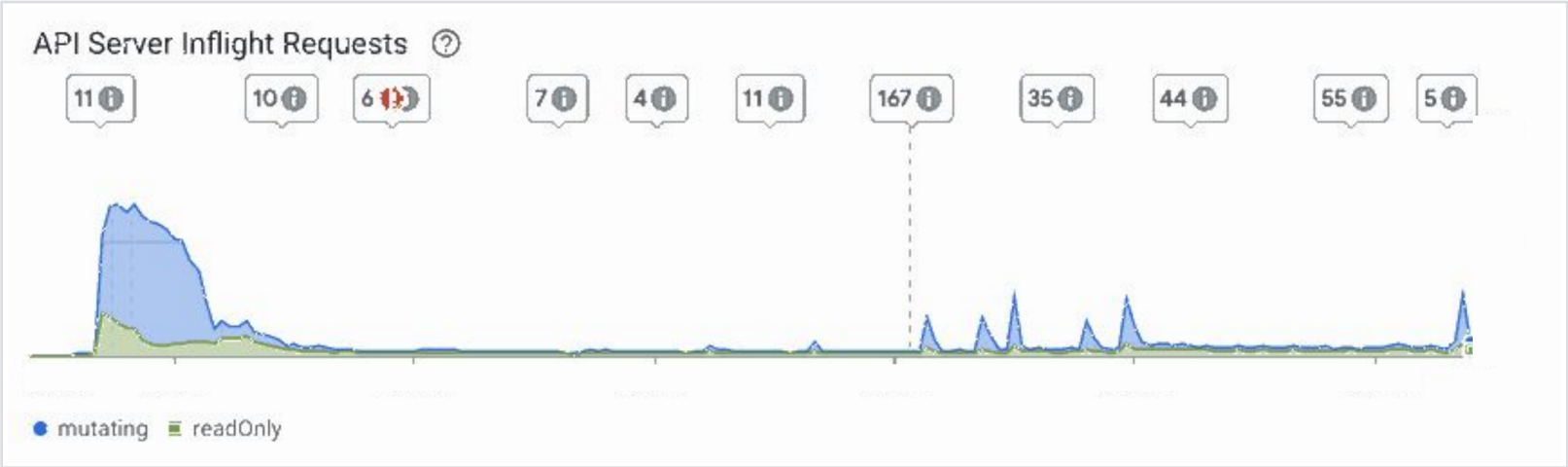
Firewall-and-quiesce: let them back in waves

Block node traffic, let the control plane go quiet, then admit one address range at a time.



What a wave looks like from the API server

Requests in flight at the API server as the firewall was lifted. Google's chart.



TOO MANY AT ONCE
A wall of requests in flight at the same moment, most of them writes.

THE WAVES
Each small peak is one batch of nodes let back through the firewall. Each settles before the next.

SAME NODES, SAME SERVERS
Let in a batch at a time, the peaks stay small and the floor stays flat.

Admit a batch. Wait for flat. Then the next.

The boundaries we added

Make start-up cheap, repair one master at a time, and never let everything restart at once.



CILIUM#44307
CNI start-up no longer explodes with endpoints · resolved in Cilium 1.18+

ROLLING REPAIR
one master at a time, never all at once

LEADER ELECTION
for the VPA recommender: one replica watches, not all

BUDGET RULE
startup probe ≥ ready time at maximum scale

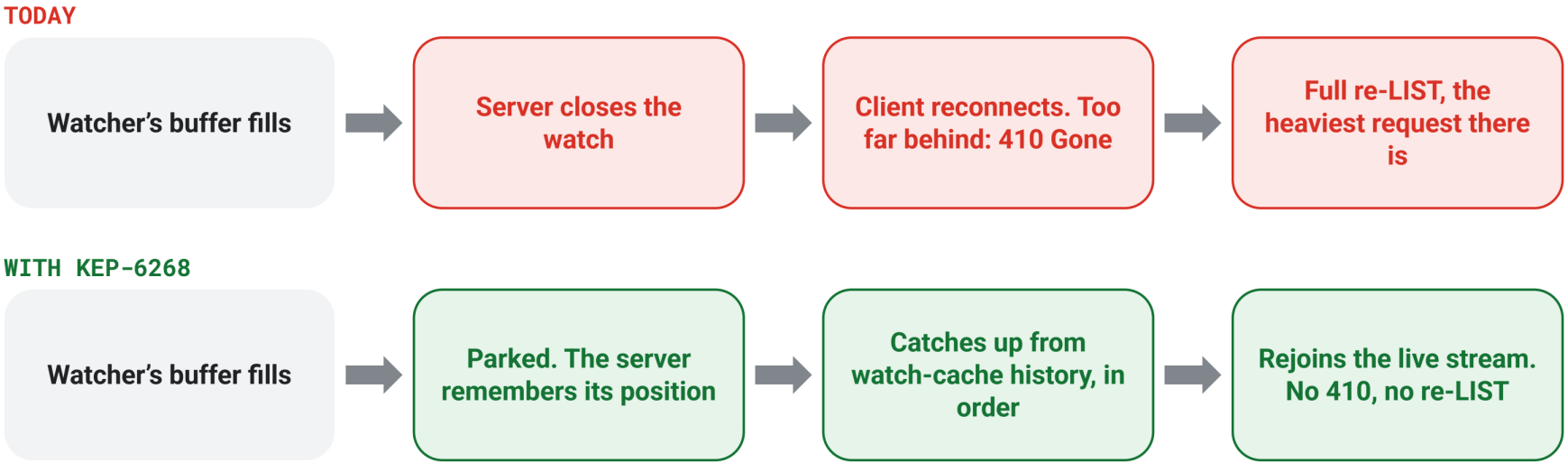
Recover in waves. Never all at once.

Connecting these issues

Watchers

KEP-6268: park a slow watcher, don’t kill it

Today a full buffer gets the watch closed, and the client comes back with a full LIST. Parked, it catches up from history.



Alpha, off by default (gate WatchCacheStallResume). KEP merged Oct 2026; code not landed yet (k/k#141450). Author: nami-ant (Anthropic); approved by Google reviewers. · Deep dive: Ryan Brewster, “Surviving the Churn”.

Slow is not dead. Park it, don’t kill it.

WHAT COMES NEXT

Big picture

Where this goes from here.

Patch on the spot. Upstream by default.

Fixes reach the largest clusters first, and every carried patch has a path home.

Hit at scale
an incident on the largest clusters

→ **Patch, qualified at scale**
tested where it broke

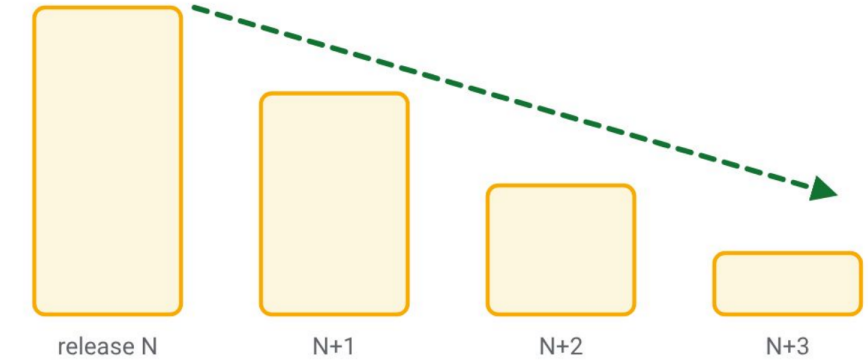
→ **Applied on the spot**
to the live clusters, in days

→ **Upstream PR**
reviewed in the open

→ **Next Kubernetes release**
everyone gets it

→ **Carried patch dropped**
nothing left to maintain

Carried patches per release



THE RULE WE WORK BY

Every carried patch needs a way home.

A fix that only exists in one fork is a fix you will have to re-apply, re-test and re-explain on every upgrade.

illustrative; not patch counts

What comes next

How each PR reached our clusters

Three routes from a merged PR to a fix that is running. The middle one is the patch on the spot.

WRITTEN FROM OUR INCIDENTS

One of our two companies wrote it, upstream first.

- **k/k#129205**
  watch-cache window · Google · v1.33
- **k/k#136812**
  404 on delete · Anthropic · v1.36
- **k/k#140167**
  DaemonSet hot loop · Anthropic · v1.37
- **k/k#140896**
  stream cache LISTs · Anthropic · v1.38

BACKPORTED INTO OUR BUILDS

Someone else had written it. GKE carried it into our build within days.

- **k/k#130806**
  pod indexers · written at AWS · v1.34
- **k/k#134739**
  pod queuing · written at AWS · v1.35
- **k/k#137885**
  kubelet restart fix · community · 1.35 cherry-pick

ADOPTED AS A GATE

Alpha and off by default upstream. On for us.

- **k/k#138701**
  no storage fallback: retryable 429 · v1.37

Upstream by default. On the spot when it cannot wait.

Questions we don't have answers to yet

Open source is here to stay. Running it with heavy customization is the unsolved part.

How much customization is too much?



Is the limit node count, or blast radius × recovery time?



Deep dive: Thursday's panel, 11:50, this room

Do the largest users need a different Kubernetes?



Customize at a seam, not with a fork

Keep the API standard, swap the implementation behind it, bring what works back upstream.

SCHEDULING

Plugins + Workload API

Your logic as plugins. Gang scheduling API beta since 1.37.

framework + plugins

QUEUEING

Kueue admission checks

Your controller decides when a job starts, here or in another cluster.

one API, many implementations

CAPACITY

Capacity Buffer API

Declare spare capacity once; any autoscaler can provide it.

one API, many implementations

DEVICES

DRA device drivers

GPU and NIC vendors ship drivers outside core. Stable since 1.34.

out-of-tree drivers

INFERENCE

Inference endpoint picker

Routing logic as plugins. llm-d builds its router on it.

framework + plugins

WORKLOADS

Workload APIs as add-ons

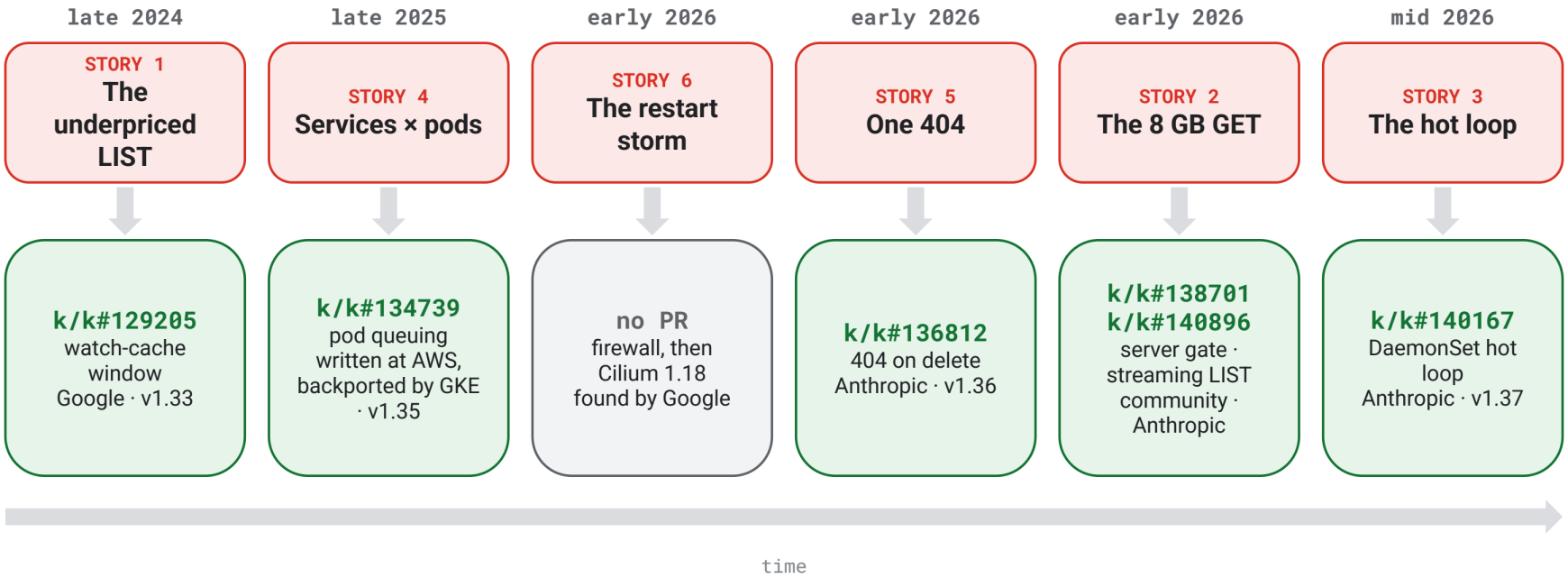
JobSet, LeaderWorkerSet and Agent Sandbox ship outside core.

out-of-tree APIs

Diverge at a seam. Upstream what works.

Two years. Six incidents. The PRs that saved them.

In time order, not talk order. Red is what broke. Green is the upstream PR. Grey means no PR saved it.

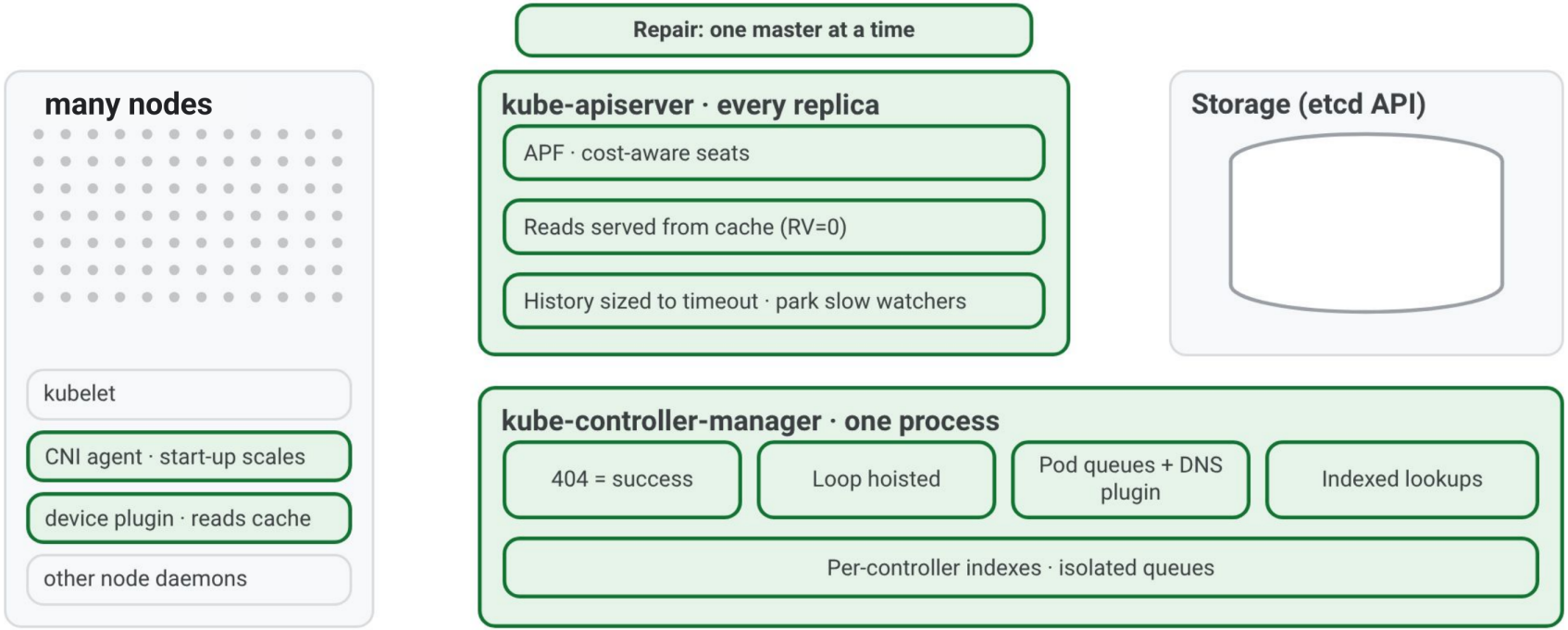


Five of the six ended in an upstream PR. The sixth ended with a firewall.

Close

Every fix added a boundary

Same map, same nodes: now each failure stays where it starts.



Close

Thank you

Jian Cheung · Anthropic

Maciek Różacki · Google

Questions?

See also: Thursday 11:50 · same room · “The Scheduling Stack” panel

See also: Ryan Brewster (Anthropic) · “Surviving the Churn” talk



How Anthropic Almost Melted GKE (And the PRs That Saved It)

The PRs that saved it

Leave this up for questions. Every row is public.

Story	What broke	PR	Who wrote it	Release
1	A multi-GB LIST was underpriced; 75 s of history	`k/k#129205`	Google	v1.33
2	One node's pods read every pod: 8 GB	`k/k#138701` · `k/k#140896`	community · Anthropic	v1.37 · v1.38
3	DaemonSet controller re-parsed per node	`k/k#140167`	Anthropic	v1.37
4	Every pod update × every Service	`k/k#134739`	AWS, backported by GKE	v1.35
5	One 404 aborted a whole scale-down	`k/k#136812`	Anthropic	v1.36
6	Most node agent restarted in lockstep	`none: firewall, then Cilium 1.18`	Google found it (cilium#44307)	GKE 1.35
Next	A slow watcher is killed, then re-LISTs	`KEP-6268` · `k/k#141450`	Anthropic, with Google review	alpha, not landed

Close