KubeCon CloudNativeCon North America 2026

#KubeCon #CloudNativeCon

GitOps Meltdown

How a commit triggered a cascading delete of 39 clusters in production

Deepak Goel
CTO Cloud Native



GitOps assumes the configuration converges to Git

What happens when the desired state is wrong?

39

workload clusters deleted, with all their applications

1

management cluster driving all of them through GitOps

~8h

between the first and second wave of deletion

AGENDA

What this talk will cover

* Background	Declarative platforms, Cluster API, GitOps and Flux
* The platform	How 39 clusters were wired to one management cluster
* The incident	Two waves of deletion, minute by minute
* Recovery	What worked, what didn’t, and what saved the data
* Lessons	Shrinking the blast radius and staying recoverable

BACKGROUND · DECLARATIVE PLATFORMS

Declarative by default, reconciled forever

spec
Desired state
What you declare in YAML: what should exist and how it should look.

→

controller
Reconcile loop
Observe the world, diff it against the spec, act to close the gap. Repeat.

→

status
Actual state
What the API server, the nodes and the infrastructure report right now.

The feature: drift is corrected automatically, without a human in the loop.

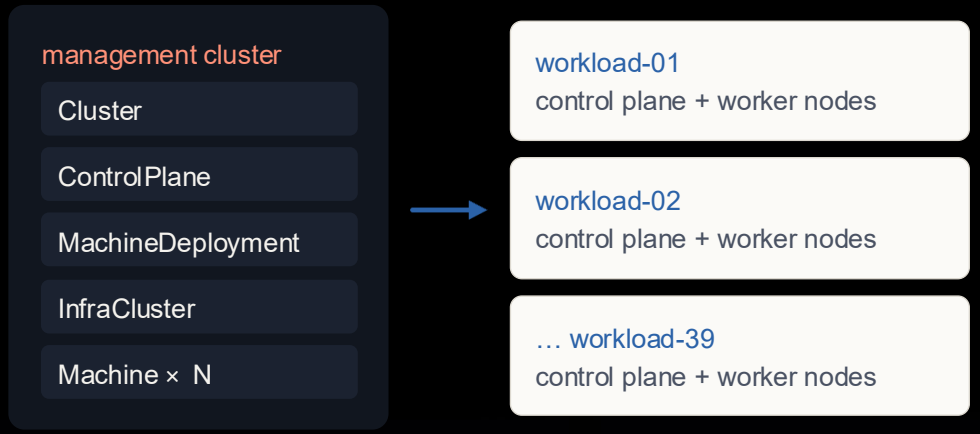
The risk: a wrong desired state is applied just as faithfully, at machine speed.

Cluster API: Managing Clusters as Code

A **management cluster** runs the Cluster API controllers and stores a set of objects for every workload cluster.

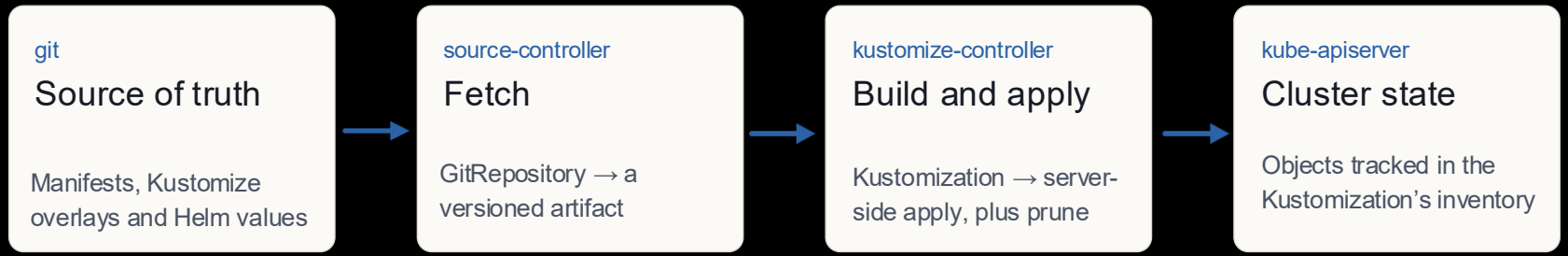
Infrastructure providers turn those objects into VMs, control planes and nodes.

Delete the Cluster object and its finalizer tears down every machine behind it.



BACKGROUND · FLUX

Pull, build, apply, then keep reconciling



Flux usually manages itself. Its own controllers and CRDs are objects in a Kustomization it reconciles. Remember this for later.

BACKGROUND · FLEET MANAGEMENT

Putting it together: a fleet driven by commits



One commit can create a fleet.

One commit can delete one.

Flux remembers everything it applied

Every Kustomization records what it applied in .status.inventory.

With prune: true, anything that disappears from the source is deleted from the cluster.

Deleting the Kustomization itself garbage-collects its whole inventory

Flux can't tell "removed on purpose" from "missing by mistake."

```yaml
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: clusters
spec:
  path: ./clusters
  prune: true  # delete what leaves Git
status:
  inventory:
    entries:
    - id: fleet_workload-01_cluster.x-k8s.io_Cluster
    - id: fleet_workload-02_cluster.x-k8s.io_Cluster
    # … 37 more Cluster objects
```

Deleting an object is a process

01
DELETE request
The API server sets deletionTimestamp. The object is now Terminating.

→

02
Finalizers run
Owning controllers clean up external resources, then remove their finalizer.

→

03
Garbage collection
Dependents found through ownerReferences go too.

→

04
Object removed
Only once the finalizer list is empty does the object leave etcd.

No controller, no cleanup. If the controller that owns a finalizer is gone, the object stays Terminating indefinitely.

BACKGROUND · STORAGE LIFECYCLE

Where the data lives

StatefulSet
whenDeleted: Retain

Its claim retention policy decides whether PVCs outlive it when it's deleted or scaled down.

→

PVC
ownerReferences

Removed by its owner, by garbage collection or by hand. Deleting it releases the PV.

→

PV
reclaimPolicy

Inherited from the StorageClass. Delete is the default for dynamically provisioned volumes.

→

Backing volume
CSI DeleteVolume

The driver removes the disk, and the storage system decides what that really means.

Each layer has its own deletion semantics. In a recovery, you go looking for the lowest layer that still has your bytes.

THE PLATFORM

One management cluster, 39 workload clusters

Management cluster

Flux controllers
Reconcile the platform, the fleet and themselves

In-cluster Git server
Operator-managed, on a PVC: Flux's source of truth

Cluster API
39 Cluster objects and all their machines

→

Workload clusters × 39



Each runs production applications, delivered by the same GitOps pipelines.

underneath VMs with block storage

backups Velero + backup appliance · 24 h RPO

THE HIDDEN COUPLING

The control plane managed its own lifeline

Source inside the blast radius

Flux's source of truth, the in-cluster Git server, lived in the same cluster Flux could prune.

A controller that can delete itself

Flux reconciled its own controllers. Pruning them stops everything mid-flight.

One tree owned everything

A single reconciliation hierarchy reached the platform and all 39 workload clusters.

None of this is exotic: a self-managing flux-system Kustomization is what a standard bootstrap gives you.

Day 0

The trigger: one deleted Kustomization

While changing the configuration of another object, an admin with elevated privileges **unintentionally deleted the top-level Flux Kustomization.**

That Kustomization's inventory held everything beneath it: the platform, Flux itself and **all 39 cluster resources** with their applications.

With prune enabled and the default deletion policy, Flux did what it was built to do: it garbage-collected the whole inventory.

Day 0

Wave 1: Flux starts pruning

01 Applications start terminating.

02 The in-cluster Git server is deleted: repository data gone, its volumes unmounted.

03 Flux deletes its own controllers mid-sequence.

04 The teardown stalls: orphaned resources, stuck finalizers, nothing left to finish the job.

12

minutes from the first prune to a stalled, half-deleted platform

THE TWIST

Flux deleted itself, the only reason 39 clusters survived the first wave.

A stalled delete is not a safe state. Every object still carried its deletion intent, waiting for a controller to come back.

Day 0

Wave 2: restoring Flux resumed the delete

restore

Flux comes back

Restored by hand while rebuilding the management cluster.

→

reconcile

Same desired state

It pulled the latest configuration, deletion markers still in place.

→

prune

Job finished

Everything that remained was deleted, including all 39 workload clusters.

Restore order matters. Bringing the reconciler back before fixing the desired state replays the incident.

TIMELINE

Four days, end to end

Day 0 · 13:03

Trigger

Privileged change marks 39 clusters and their apps for deletion

Day 0 · 13:15

Wave 1 stalls

Git server deleted; Flux deletes itself mid-prune

Day 0 · 21:00

Wave 2

Restored Flux resumes pruning; all 39 clusters deleted

Day 1 – 2

Clusters back

39 of 39 workload clusters redeployed through GitOps

Day 2 – 4

Apps back

Tiered restore in order of business criticality

Orange: destruction · Blue: recovery ·

RECOVERY · DAY 1 – 2

Rebuilding clusters from Git

39/39

workload clusters redeployed



The declarative model that amplified the failure also made the rebuild repeatable: clusters came back through the same GitOps pipelines.

RECOVERY · DAY 2 – 4

The backups weren’t ready for this

12 h

Age of the most recent Velero snapshots when they were needed.

Partial

The backup appliance couldn’t restore native Kubernetes application layers.

Untested

The end-to-end recovery pipeline had never been verified in practice.

> A backup strategy is a hypothesis until you’ve restored from it.

RECOVERY · DAY 2 – 4

What saved the data: the storage layer

Kubernetes objects	deleted
Machines and VMs	deleted
Block storage	kept in recycle bin

The underlying block storage was not erased. It stayed in the storage system's recycle bin.

Recovery came from a layer nobody had planned to use. Know your lowest layer, and how long it keeps deleted data.

RECOVERY PLAYBOOK

Restore without replaying the incident

01

Freeze reconciliation

Bring controllers back with Kustomizations suspended, or kustomize-controller scaled to zero.

`flux suspend kustomization --all`

02

Fix desired state first

Revert the bad change in Git and confirm no deletion intent remains anywhere.

`git revert <sha>`

03

Restore the source

Bring back the Git server and its data before any reconciler reads from it.

04

Diff before you apply

Compare what Flux would do with what's running now.

`flux diff kustomization clusters --path ./clusters`

05

Resume with pruning off

Reconcile with prune: false until the inventory matches, then turn pruning back on.

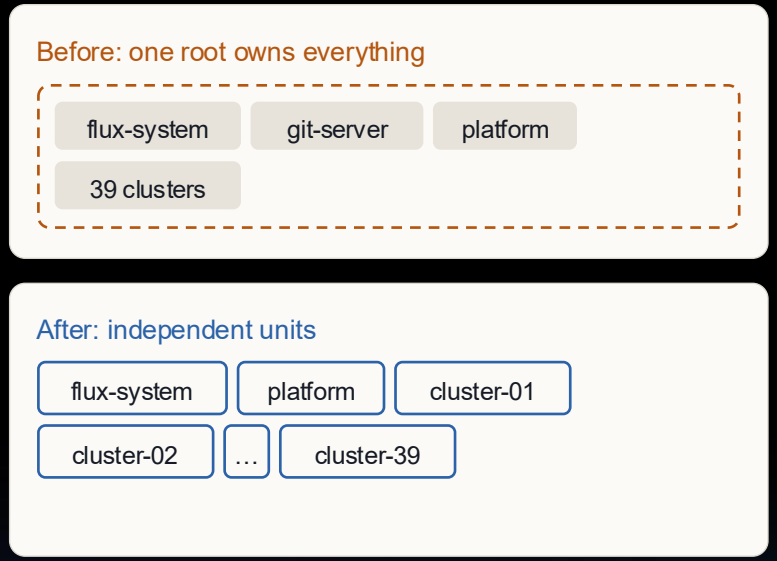
06

Recover data bottom-up

Start from the lowest layer that kept the bytes: storage, then snapshots, then app backups.

LESSON 1

Shrink the blast radius



One Kustomization per cluster so one bad path can’t reach the whole fleet.

Flux is bootstrapped and protected separately.

LESSON 2

Make bulk deletion hard

Mark Cluster objects, the Git server and flux-system with **prune: disabled**.

Set **deletionPolicy: Orphan** on Kustomizations that own infrastructure.

Deny DELETE on Cluster objects at admission, with ValidatingAdmissionPolicy or a policy engine, unless a reviewed label says otherwise.

```
# Never prune, even if it leaves Git
metadata:
  annotations:
    kustomize.toolkit.fluxcd.io/prune: disabled
---
# Deleting the Kustomization orphans,
# it doesn't delete
spec:
  prune: true
  deletionPolicy: Orphan
```

LESSON 3

Protect stateful infrastructure

reclaimPolicy: Retain

On StorageClasses for critical data: deleting a PVC must not delete the bytes.

whenDeleted: Retain

In each StatefulSet's claim retention policy, so volumes outlive the workload.

Git outside the blast radius

Host or mirror the source of truth outside the cluster it drives.

Snapshots + recycle bin

Storage-level protection with a known retention window, and someone who knows how to use it.

LESSON 4

Watch the watchers

Alert on the pipeline itself

Ingestion rate drops, Fluent Bit retries and errors, Loki rejecting writes.

Audit logs off-cluster

Ship API server audit logs to storage the cluster can’t delete.

Page on mass prunes

Send Flux events through notification-controller to an external sink.

Silence is a signal

Treat “no logs” as an incident, not as a quiet night.

LESSON 5

A backup you haven’t restored is a hope

RPO target

≤1 h

down from 24 h

RTO target

≤30 min

for the platform layer

Test Kubernetes-layer restores end to end, on a schedule.

Prove each tool restores what you think, application layer included.

Run a game day: lose the management cluster and time the rebuild.

THE ONE THING TO REMEMBER

Automation is only as safe as the desired state it trusts.

Bound the blast radius.

Guard the deletes.

Rehearse the recovery.

KubeCon CloudNativeCon
North America 2026

Thank You

