



KubeCon



CloudNativeCon

Europe 2026

#KubeCon #CloudNativeCon

Generalizing Kubernetes Controller Sharding

LY Corporation

Tomoyuki Nakamura / OTSUKA, Motohiro



Agenda



KubeCon



CloudNativeCon

Europe 2026

1



Why scaling broke

2



What sharding can and cannot solve

3



Why our first architecture failed

4



How we pivoted

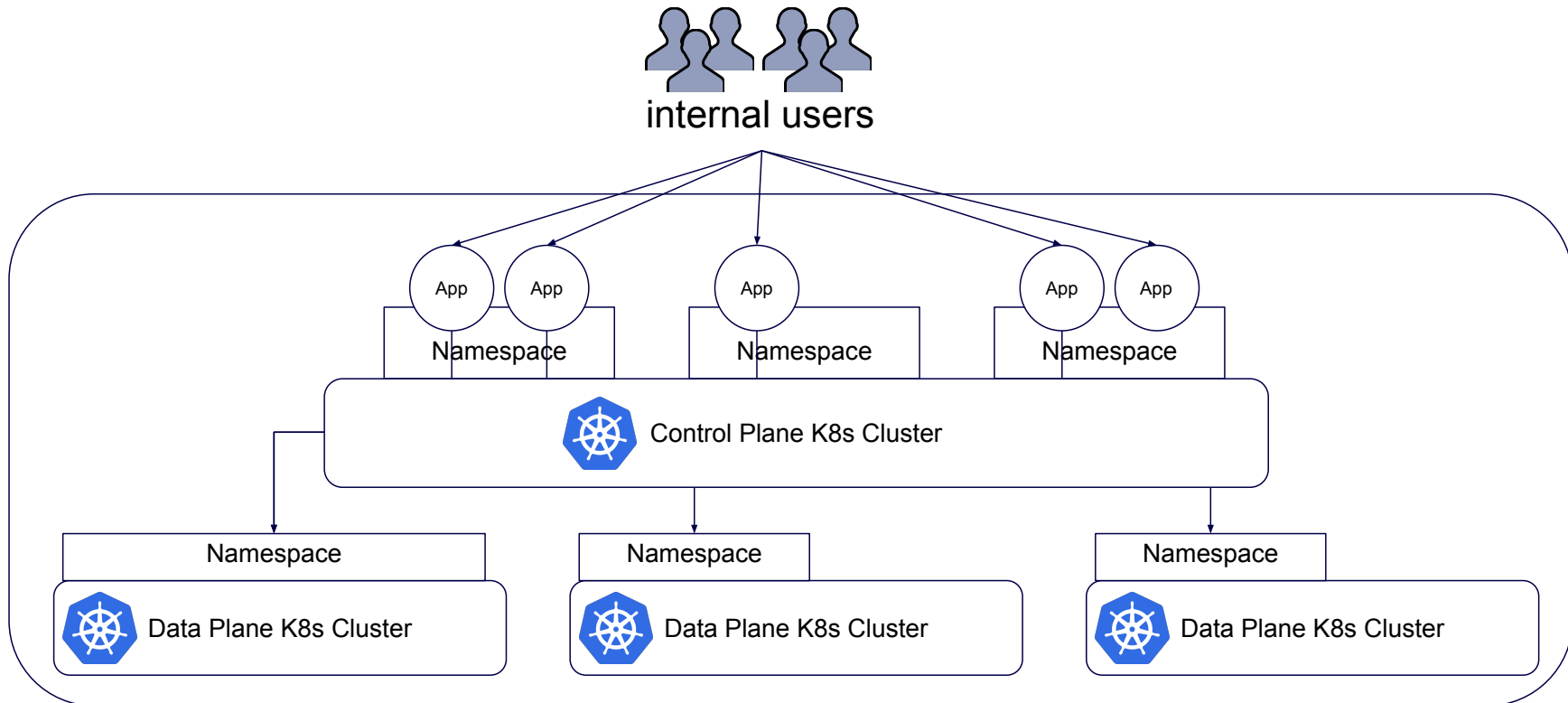
5



A reusable decision framework

**Our team develops and operates
a Kubernetes-based internal PaaS
for LY Corporation.**

Internal PaaS Architecture Overview



Motivation: When Controllers Stop Scaling



KubeCon



CloudNativeCon

Europe 2026

500k+

Managed Objects

Scale snapshot:

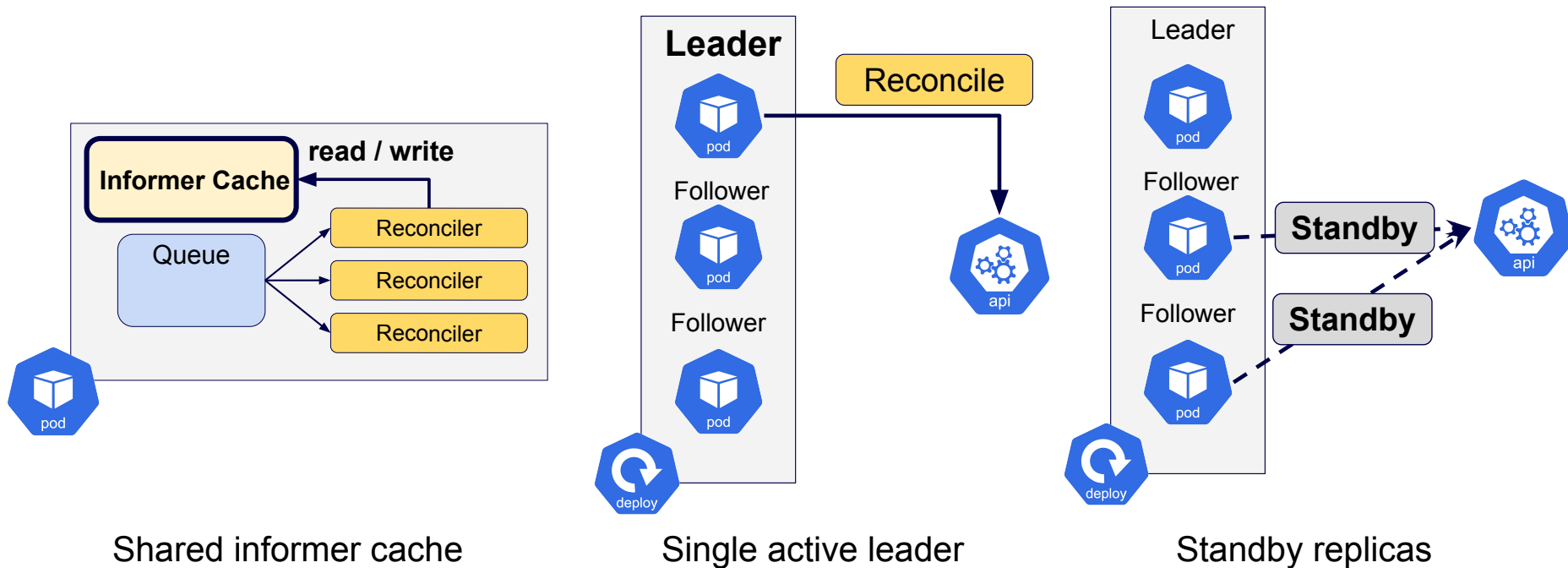
66,000 Pods, 5,000 Nodes, 3,000 Namespaces

Pain Before Sharding

- ⚠ Slow initial sync on resync/restart
- ⚠ Longer reconcile latency
- ⚠ Delayed user-visible propagation

Important: OOM was not the initial trigger. Performance degradation preceded memory limits.

Kubernetes controllers follow the same model



Efficient for vertical scaling but not for horizontal scaling

But not horizontally elastic

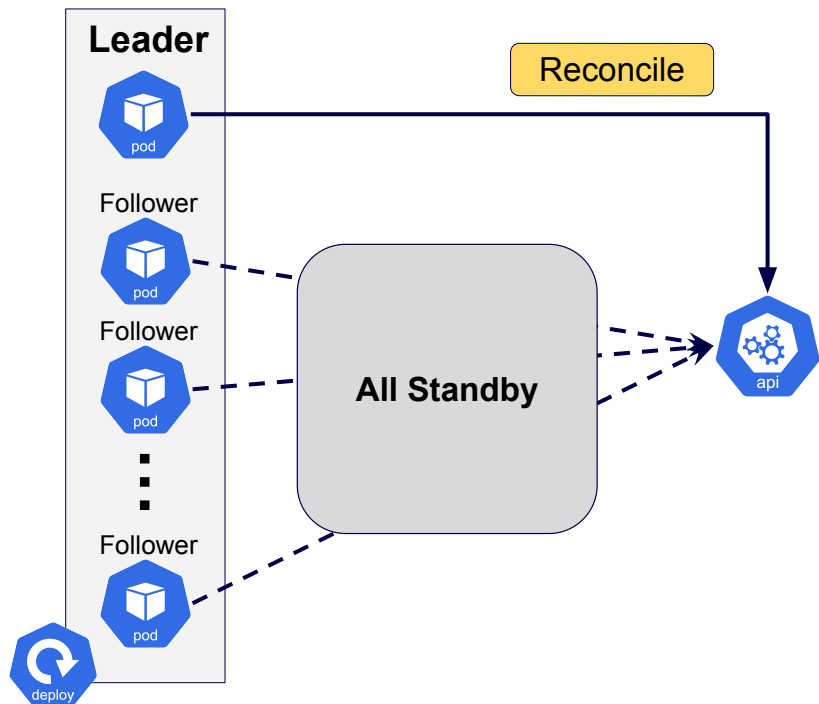


KubeCon



CloudNativeCon

Europe 2026



Leader election ensures only one replica reconciles

What Is Kubernetes Controller Sharding (KCS)?



KubeCon



CloudNativeCon

Europe 2026

The Project

Open source project by Tim Ebert

`github.com/timebertt/kubernetes-controller-sharding`

Goal: Make Kubernetes controllers horizontally scalable.

KCS shards reconciliation ownership, not the Kubernetes API.

Core Mechanism

- Hash-based shard selection from resource identity (namespace/name)
- Lease-based shard ownership
- Label-based assignment
- Parent/controlled-resource co-location
- Drain-based safe movement during rebalance

In practice: Objects labeled by shard, each shard scopes cache via label selectors, reducing memory usage.

KCS Shard Assignment: Consistent Hashing Ring

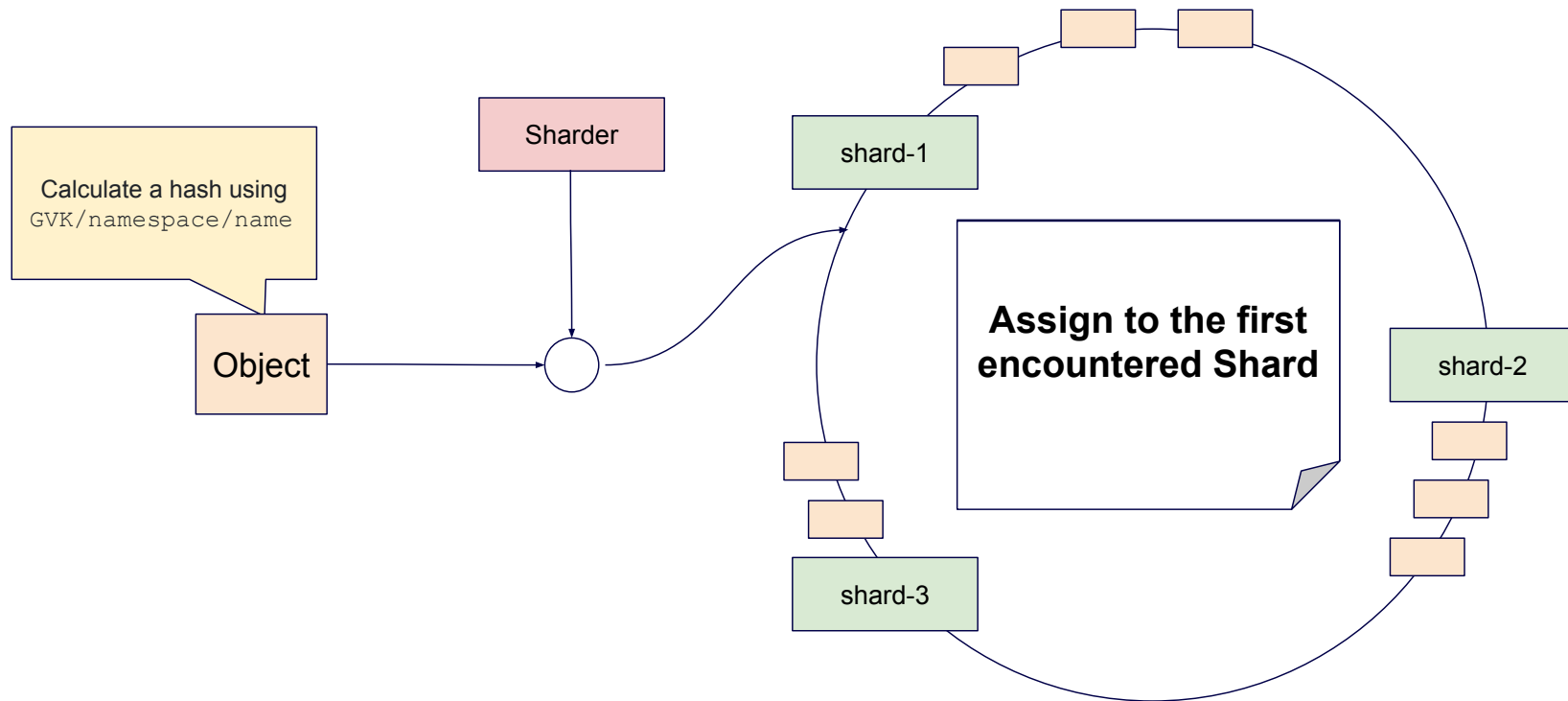


KubeCon



CloudNativeCon

Europe 2026



Parent
Resource

```
---  
apiVersion: paas.example.com/v1  
kind: App  
name: simple01  
namespace: ns-r00
```

Child
Resource

```
---  
apiVersion: paas.example.com/v1  
kind: AppChild  
name: simple01-xxxxx  
namespace: ns-r00  
ownerReferences:  
- apiVersion: paas.example.co/v1  
  blockOwnerDeletion: true  
  controller: true  
  kind: App  
  name: simple01
```

GrandChild
Resource

```
---  
apiVersion: paas.example.com/v1  
kind: AppGrandChild  
name: simple01-xxxxx-yyyy  
namespace: ns-r00  
ownerReferences:  
- apiVersion: paas.example.co/v1  
  blockOwnerDeletion: true  
  controller: true  
  kind: AppChild  
  name: simple01-xxxx
```

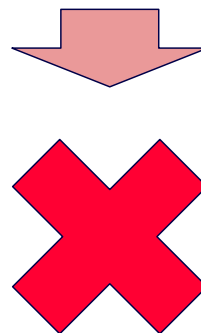
Calculate a hash using
Group / Kind / NS / Name

a7195ed2e

Calculate a hash using
GK/NS/Name of parent resource

a7195ed2e

Same Shard



KubeCon



CloudNativeCon

Europe 2026

Constraint: Parent-Hash Shard Scope



KubeCon



CloudNativeCon

Europe 2026

Implementation Constraint

- Shard is computed from hash of {parent namespace, parent name}
- Child can use OwnerReference to resolve parent
- Grandchild cannot directly resolve grandparent for that hash input

Practical Consequence

- Same-shard colocation is straightforward for parent + child
- It does not naturally extend to deeper dependency chains



Hypothesis: Namespace as Shard Boundary



KubeCon



CloudNativeCon

Europe 2026



- Given limitations, we shifted from object-level co-location to namespace-level ownership partitioning.
- At that time, App-related resources looked namespace-closed.

The Hypothesis

- Split reconcile ownership by namespace
- Reduce sync pressure
- Keep product behavior unchanged

First Attempt: Namespace-Level Sharding

Part 1: The Rationale and Fork



KubeCon



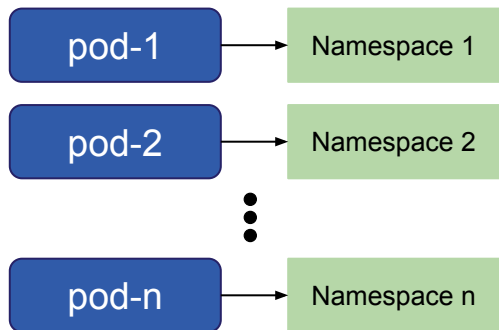
CloudNativeCon

Europe 2026

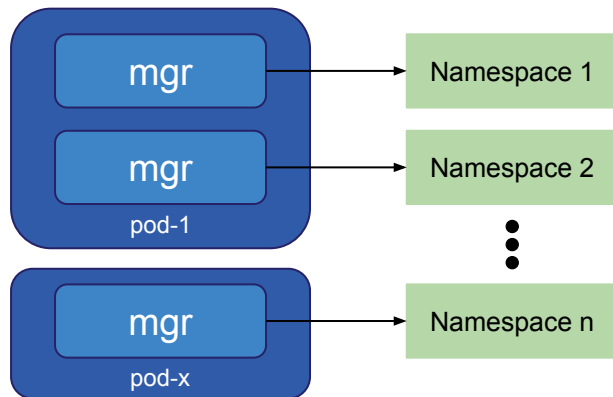
Why fork/adapt KCS?

- PaaS dependency graph did not close on parent/controlled resources.
- Upstream assumptions alone couldn't partition ownership cleanly.
- Required **namespace-level isolation** for the full domain.

Option 1: Pod per Namespace



Option 2: KCS-based Namespace Sharding



First Attempt: Namespace-Level Sharding

Part 2: Reality vs Expectation



KubeCon



CloudNativeCon

Europe 2026

The Implementation

For each namespace, we instantiated a controller-runtime manager:

- Unique cache and informer set per namespace.
- Goal: **Clean isolation** of reconcile scope.

Initial Expectation:

Cheaper and more efficient than a Pod-per-namespace approach.

Perspective	Outcome
Manager Cost	In-process cost $\approx$ Pod cos
Complexity	Multiplied the already multiplicative structure
Performance	Practical overhead far exceeded expectations

Explosion of Goroutines

3,500 → 500,000

- app-operator entered OOM restart loops
- CPU steal and memory saturation
- **Incident scale snapshot:**
 - Namespace: 3,000
 - App: 10,000

Validation and Cost Model



KubeCon



CloudNativeCon

Europe 2026

Repeated measurements in validation environment.

Metric: pprof heap inuse (inuse_space)

$$\text{heap_inuse} \approx 59 + 0.725 \times (\text{Cluster} \times \text{Namespace}) + 0.73 \times \text{App}$$

Practical dominant driver in the first-attempt architecture:

Namespace-correlated manager growth. Growth was linear vs namespace count, but namespace count itself was hard to cap operationally. At large counts, absolute resource usage exceeded expectations.

Difference in the Number of Manager Instances Before and After Sharding



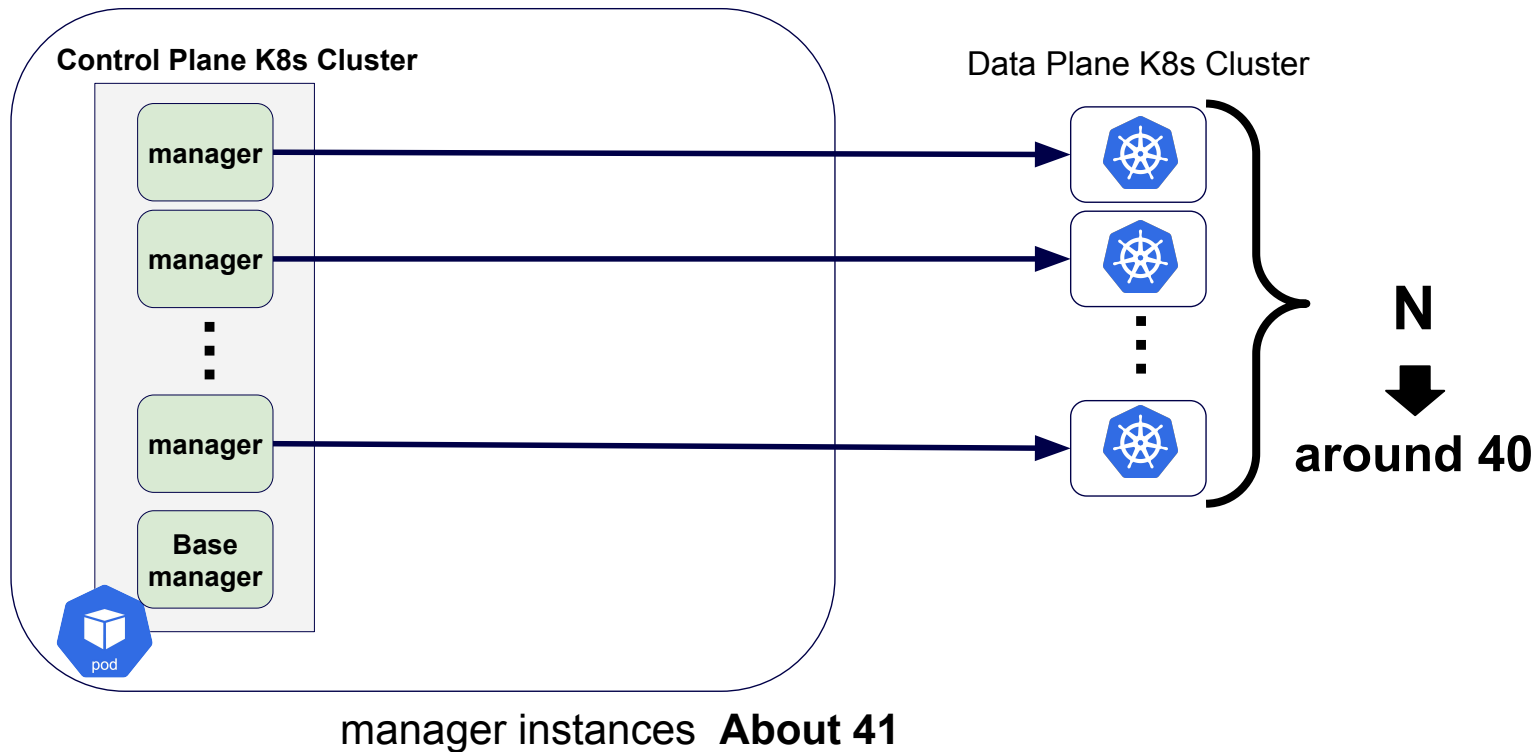
KubeCon



CloudNativeCon

Europe 2026

Before Sharding



Difference in the Number of Manager Instances Before and After Sharding



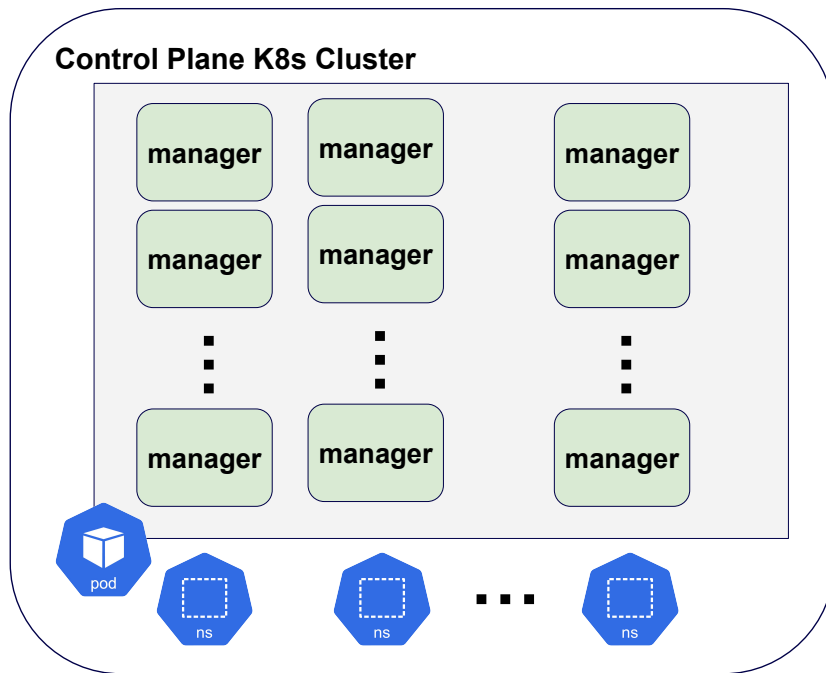
KubeCon



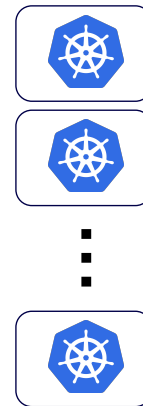
CloudNativeCon

Europe 2026

After Sharding



Data Plane K8s Cluster



Namespace sharding created a manager for **every** namespace

Difference in the Number of Manager Instances Before and After Sharding



KubeCon



CloudNativeCon

Europe 2026

Before Sharding:



Manager instances = Data Plane K8s Cluster ≈ 40

After Sharding:



Manager instances = Namespaces $\times$ Data Plane K8s Cluster

$\approx 3,000 \times 40 \approx 120,000$

API Watch Semantics Still Matter



KubeCon



CloudNativeCon

Europe 2026

- Kubernetes watch is namespaced or all-namespaces scoped
- No native arbitrary namespace-set watch primitive
- This constraint multi-namespace sharding designs in general

But in our first-attempt architecture,
primary amplification came from

.

API Watch Semantics Still Matter



KubeCon



CloudNativeCon

Europe 2026

- Kubernetes watch is namespaced or all-namespaces scoped



Note: Sharded Watch (v1.36+)

- **KEP 5866** (Server-Side Sharded List and Watch) merged March 17, 2026
- Enables hash-based range sharding for list/watch operations
 - Example: `selector=shardRange(object.metadata.uid, 0, 8)`
 - Partitions watch streams by UID hash value ranges
 - Alpha feature gate: `ShardedListAndWatch`
- Initial implementation: UID-based sharding (namespace support planned)

Reference: [KEP-5866](#)

Parent resource

```
---
apiVersion: paas.example.com/v1
kind: App
name: simple01
namespace: ns-r00
```

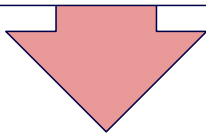
Child resource

```
---
apiVersion: paas.example.com/v1
kind: AppChild
name: simple01-xxxxx
namespace: ns-r00
ownerReferences:
- apiVersion: paas.example.co/v1
  blockOwnerDeletion: true
  controller: true
  kind: App
  name: simple01
```

Grandchild resource

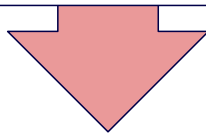
```
---
apiVersion: paas.example.com/v1
kind: AppGrandChild
name: simple01-xxxxx-yyyy
namespace: ns-r00
ownerReferences:
- apiVersion: paas.example.co/v1
  blockOwnerDeletion: true
  controller: true
  kind: AppChild
  name: simple01-xxxx
```

Compute the hash value from
Group / Kind / Namespace / Name



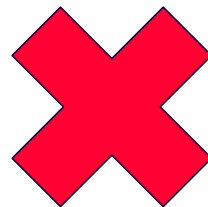
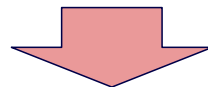
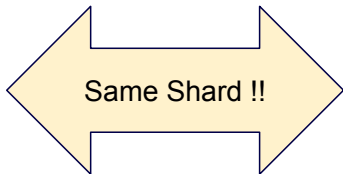
a7195ed2e

Compute the hash value from the parent's
Group / Kind / Namespace / Name via
OwnerReference



a7195ed2e

Same Shard !!



Pivot: Sharding & K8s API Principles



KubeCon



CloudNativeCon

Europe 2026

Split the Monolith



Design Principles

- ✓ **Native Label Sharding:** Leverages built-in K8s watch/list mechanics.
- ✓ **API Compliance:** Do not fight namespace watch semantics (all-or-single).

Fixed Manager Strategy

- ✓ **Label-based:** One manager handles all namespaces efficiently.
- ✓ **Cluster-scoped:** Fixed managers per Pod (1 Base + 1 Cluster).

CRITICAL PATH OUTCOME

ObjectNum + ManagerBaseline

Shard count can increase **without** increasing the underlying manager count.

Final Pattern A: app-operator



KubeCon



CloudNativeCon

Europe 2026

Applying upstream-style KCS by focusing on strict **parent/controlled domain** boundaries.

- ✓ **Isolation:** Parent resources and their direct children are co-located.

- ✓ **Resources:**

- ✓ • App (Parent)
- ✓ • AppChildA, B, C...

This fits KCS assumptions perfectly.



Final Pattern B: deploy-operator — Before



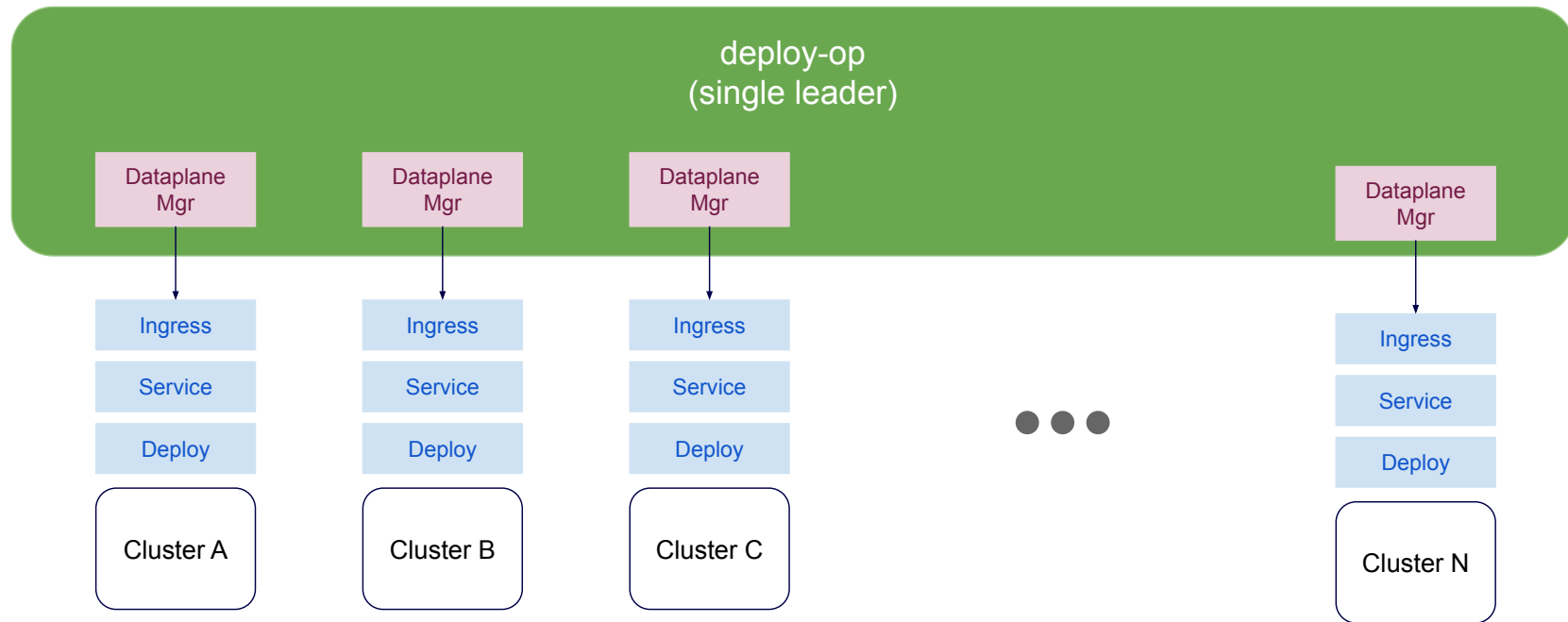
KubeCon



CloudNativeCon

Europe 2026

Before the pivot: one operator Pod, many Cluster Managers. All cluster caches in one process.



Operator : Cluster Manager = 1 : N

Final Pattern B: deploy-operator — After



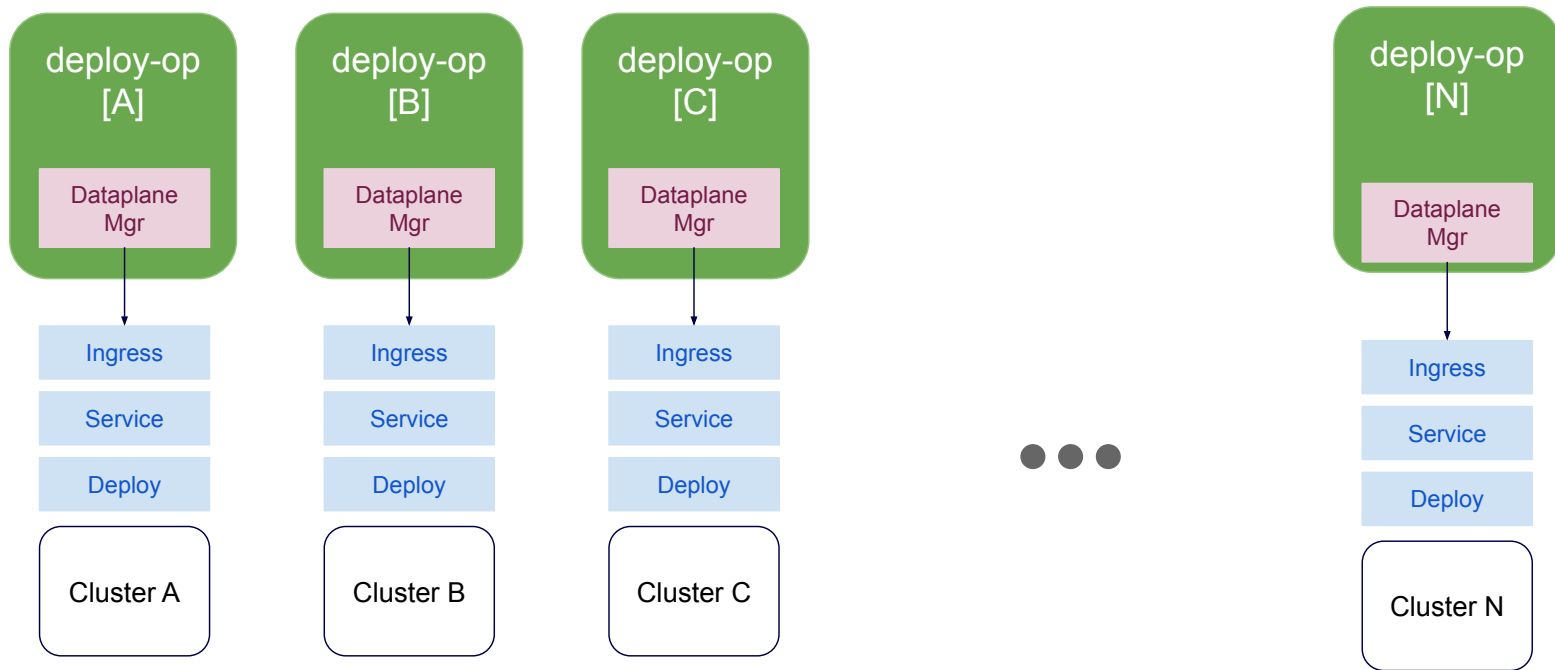
KubeCon



CloudNativeCon

Europe 2026

After the pivot: one operator instance per dataplane cluster.



Operator : Cluster Manager = 1 : 1

Production Validation: Pivoted Architecture



KubeCon



CloudNativeCon

Europe 2026

Before sharding (1 Pod)

Total: concentrated 1 process

First Attempt

OOM: namespace-correlated amplification

Pivoted (N Pods)

Shard 1

Shard 2

Shard 3

- ✓ Per-shard base overhead exists.
- ✓ Total usage comparable to pre-sharding.
- ✓ No namespace amplification.

- ✓ Failure blast radius is bounded.
- ✓ Stable continuous operation.
- ✓ Reconcile latency improved.

Production Learning 1: Ownership Alignment



KubeCon



CloudNativeCon

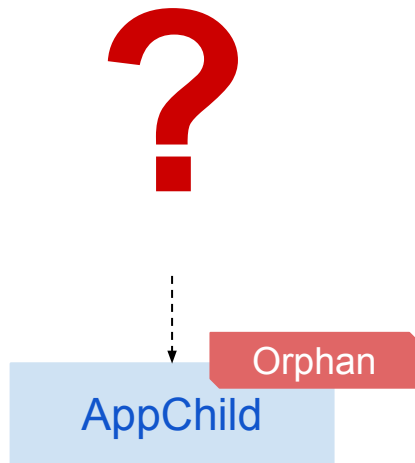
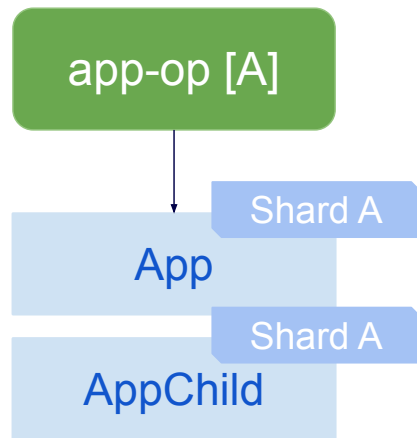
Europe 2026

KCS computes child shard from parent identity via OwnerReference. If user creates resource directly without OwnerReference, no shard is assigned and it's silently ignored.

In Our PaaS: Pre-created AppChild has no OwnerReference to App -> No shard assigned -> Ignored.

Fix: App controller adopts pre-created AppChild, sets OwnerReference -> Shard assignment realigns.

Shard-filtered reconciliation assumes ownership alignment. Break it, and behavior changes silently.



Production Learning 2: Shard Identity Stability



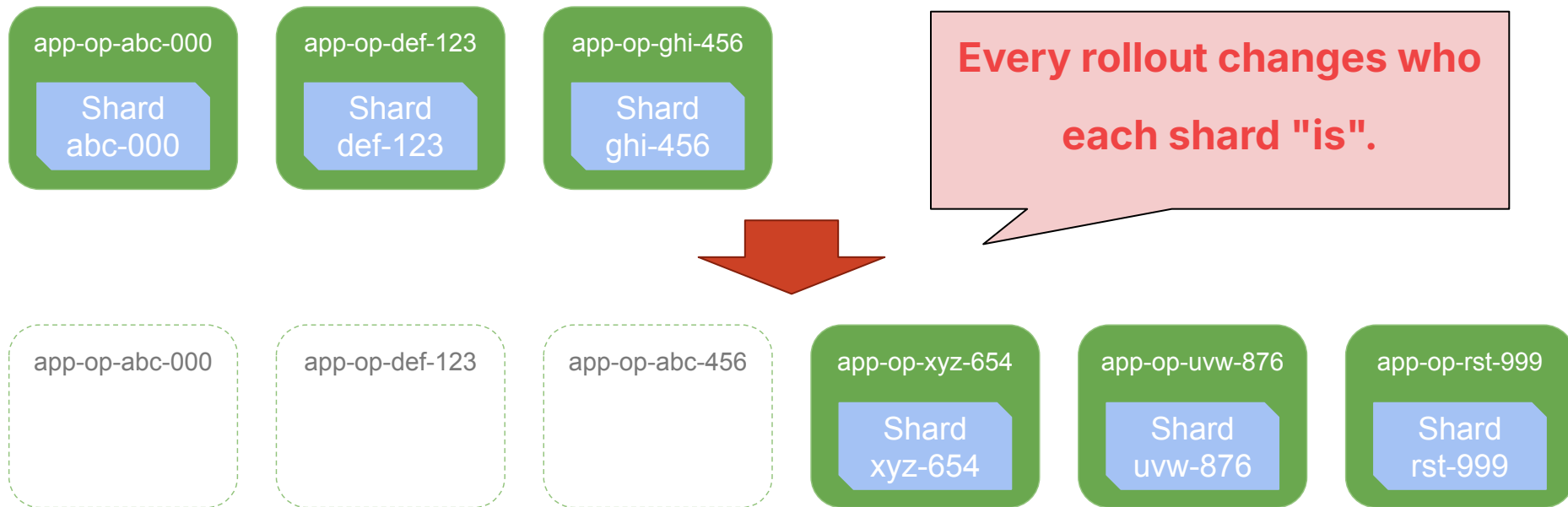
KubeCon



CloudNativeCon

Europe 2026

Shard identity = Pod name (how KCS identifies each shard)



The Write Storm



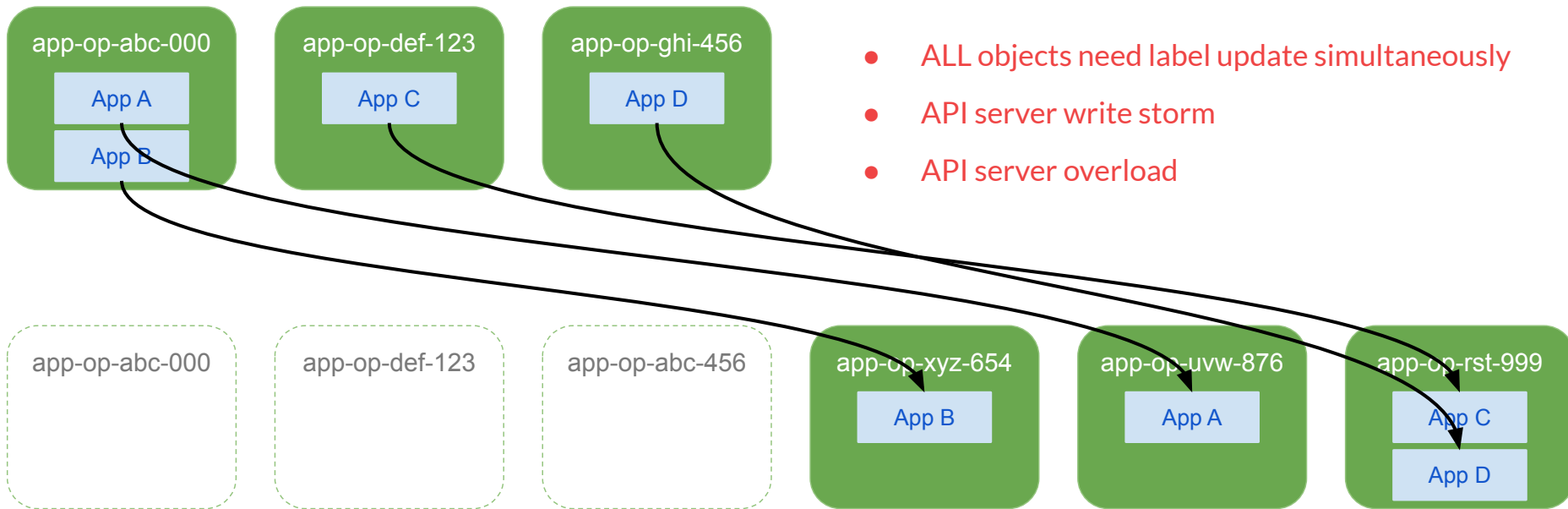
KubeCon



CloudNativeCon

Europe 2026

All managed objects carry a shard label. When shard identity changes, all labels must be rewritten.



Fix: StatefulSet for Stable Shard Identity



KubeCon



CloudNativeCon

Europe 2026

StatefulSet rollout (shard identity == stable ordinal):

Before Rollout



After Rollout



The Fix

- Move app-operator to StatefulSet
- Tune leader-election-related settings

The Result

- Pod names unchanged -> identities preserved
- No label rewrite needed
- Zero additional API server load

Generalized Principles



KubeCon



CloudNativeCon

Europe 2026



Respect API Semantics

Work within Kubernetes API design principles (watch semantics, label-based selection)



Fix Manager Count

Keep manager count per process fixed — do not let it scale with external dimensions



Shard by Architecture

Choose sharding axis based on your architecture's dimension ratio



Stable Shard Identity

Stabilize shard identity in rollout design



Operational Risk First

Evaluate operational risk as a first-class concern



Sharding is not about splitting work.

Thank you !!!



KubeCon



CloudNativeCon

Europe 2026

