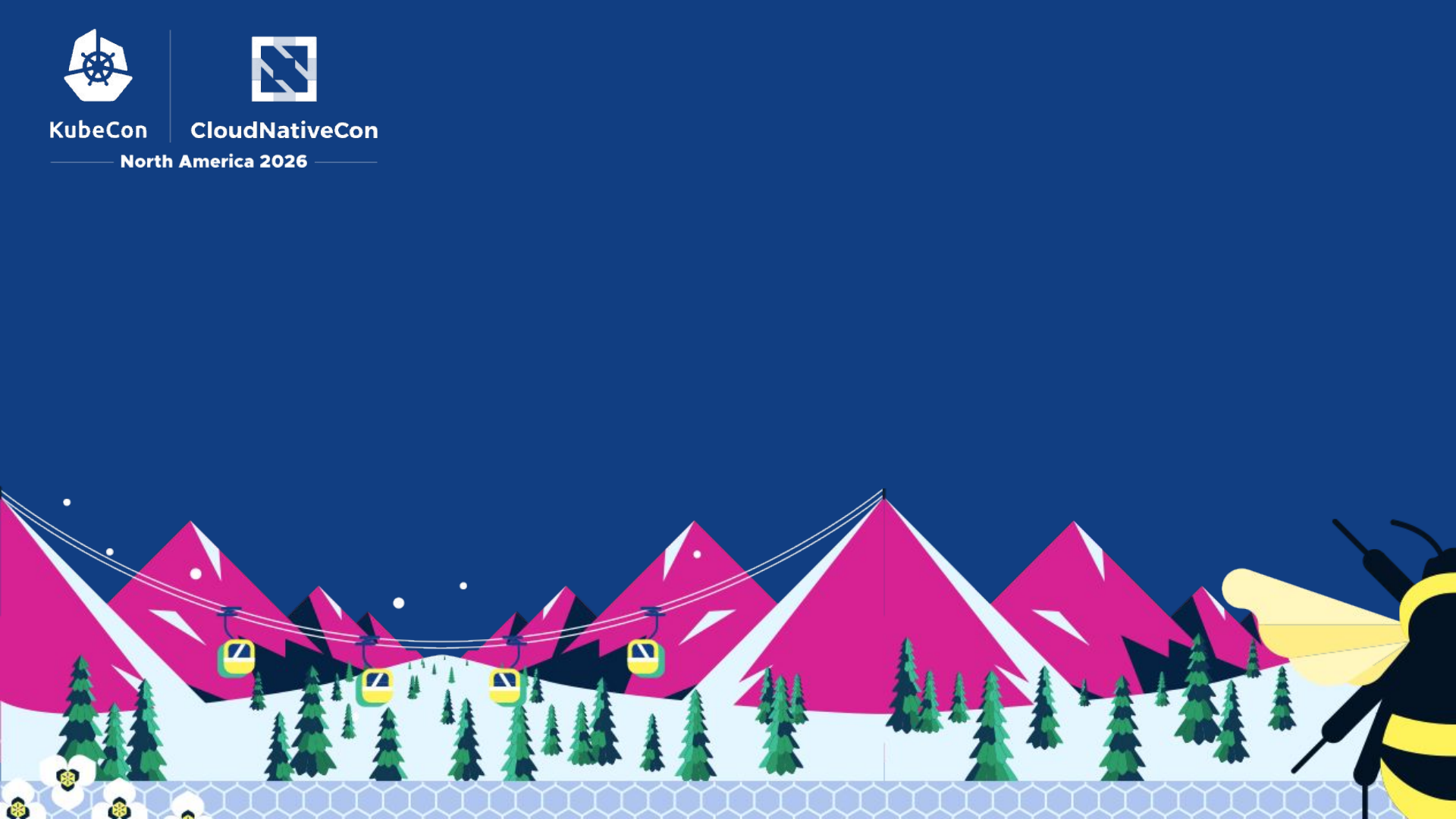


Green Dashboards, Broken Clusters: Taming Stale Watches in Kubernetes Operators

Alvaro Aleman and Michael Aspinwall




KubeCon
CloudNativeCon
North America 2026

Who we are

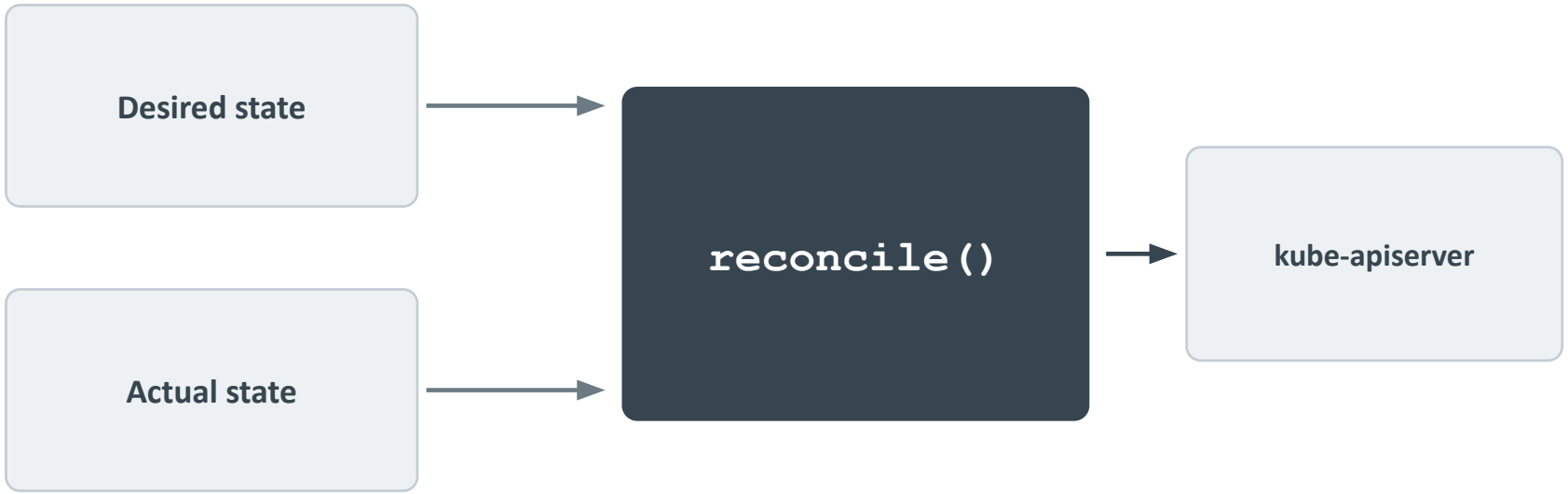


Michael Aspinwall
Google

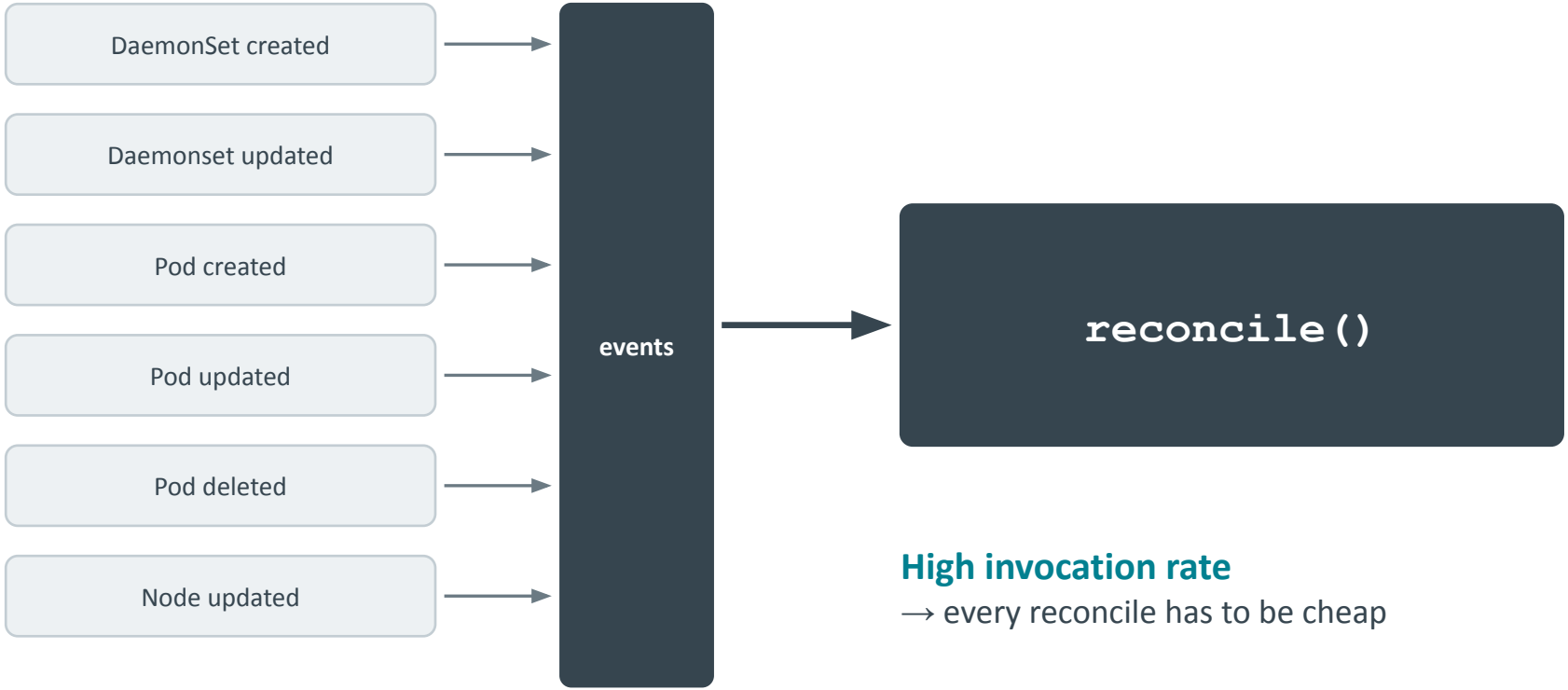


Alvaro Aleman
Baseten

The Problem: Controller basics



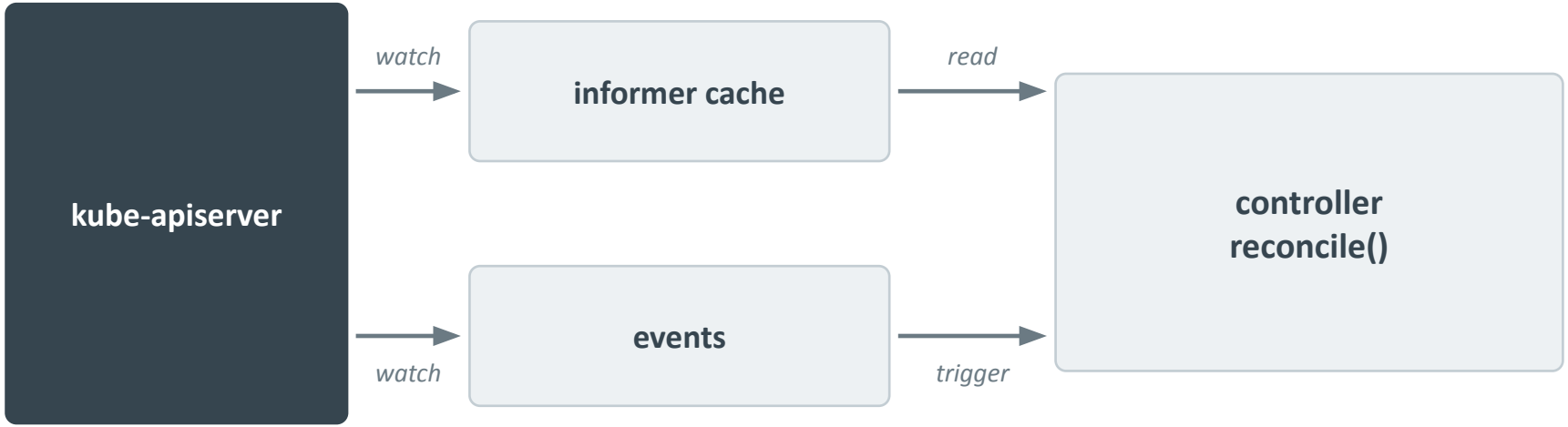
The Problem: Controller basics



High invocation rate

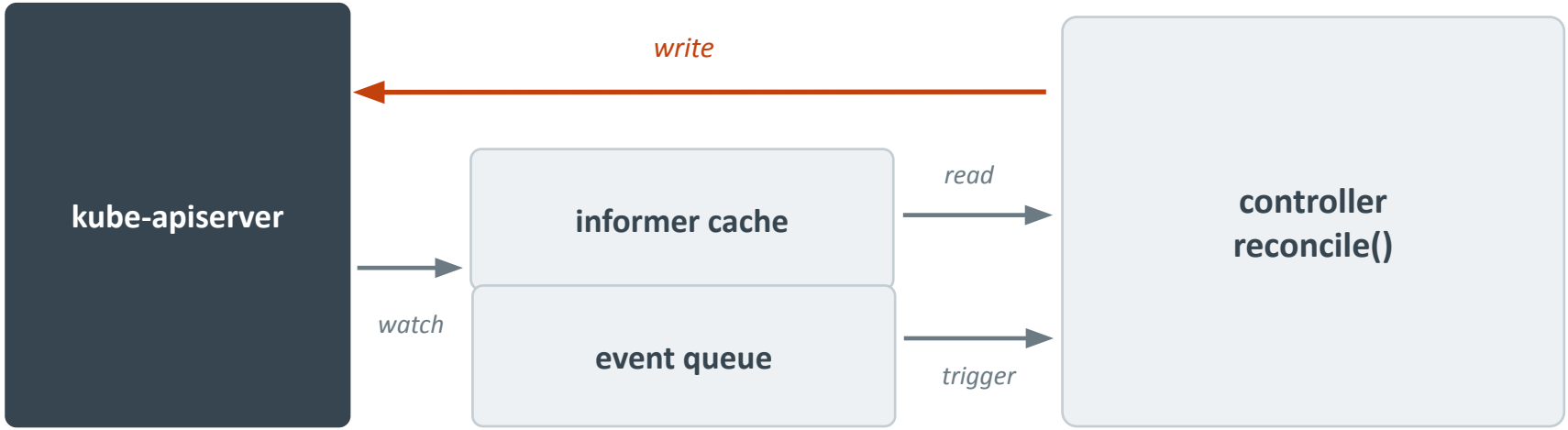
→ every reconcile has to be cheap

The Problem: Where data comes from



The same watch feeds both the cache the controller reads from and the events that trigger it.

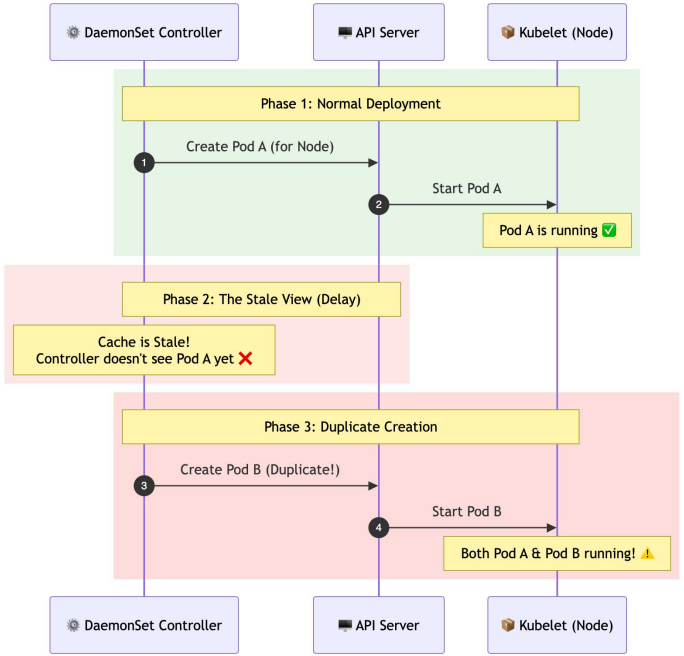
The Problem: Consistency



Not read-after-write consistent. A write lands in the cache only after a delay

Case Study - DaemonSet Controller

- DaemonSet controller relies on tracking Creates/Deletes on pods and waiting for those events before proceeding.
- ControllerExpectations time out after 5 minutes (TTL) to prevent indefinite stalls.
- Controller becomes more than 5 minutes stale
- DaemonSet controller reconciles again, making more pods on same node
    - Every 5 minutes another set of pods are created
- Triggers **infinite loop creating many pods per node until the cluster reaches a breaking point**. Ports are exhausted, crashloops occur, pods do not expect the same workload to exist on the node as itself.



Key Lessons Learned

- Controllers don't always take staleness into account when running their reconcile loops
- There are many ways that staleness can cause issues, rather than fixing each occurrence, we need a way to handle it at the source
- We need detection to see that our system is stale with metrics and alerts
- Controllers need a way to see if their previous actions have been acted upon in their cache
- Current controller tooling does not expose these concepts to authors

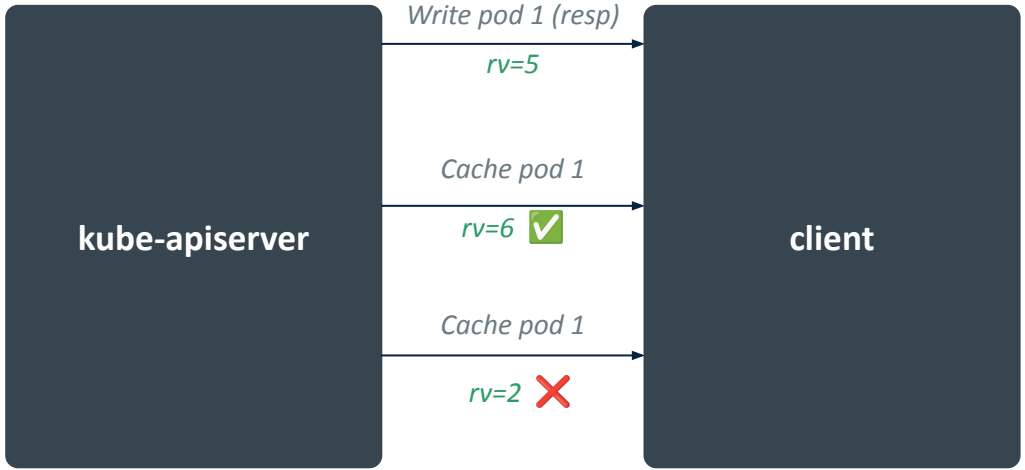
Work Done

- 3 Primary Workstreams in K8s 1.36
- Work continues in 1.37+
- Tracked in KEP-5647: http://kep.k8s.io/5647



Comparable Resource Version

- Resource versions of an object defined as only comparable by equality
- No concept of newer/older
- Define concepts of comparison for resource version http://kep.k8s.io/5504
- Simple, but requires conformance to ensure storage providers use monotonically increasing resource versions



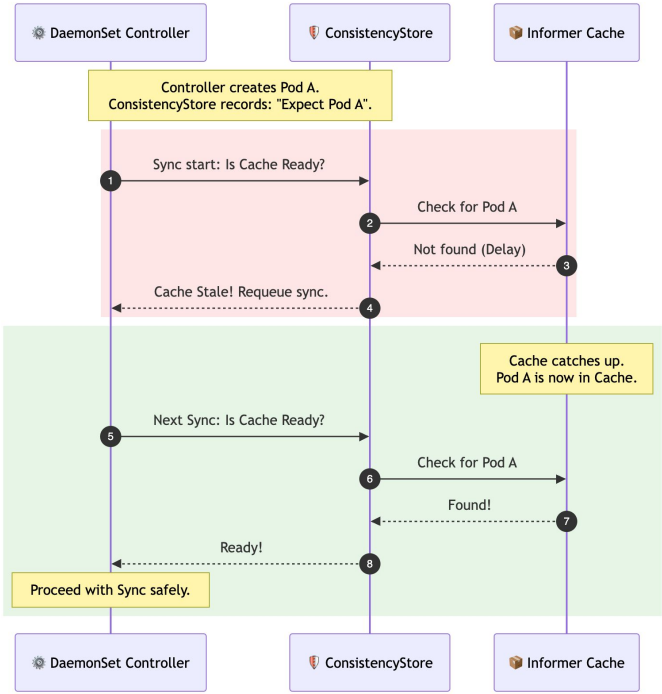
Informer Metrics

- Create metrics tracking cache staleness and informer backlog
- Instrument each GVK store in client-go based controllers to expose their resource version
- Compare with apiserver RV for lag, or use staleness monitor to measure lag in seconds!
- `informer_*` metrics in https://kubernetes.io/docs/reference/instrumentation/metrics/
- Being used to validate performance in high scale clusters https://github.com/kubernetes/perf-tests/tree/master/perflens



Consistency Store

- Enter the Consistency Store!
- Middle layer between reconciler and cache
- Every write, update the store with latest written RV
- On reconcile, ensure our local writes have been read back
- Compares written RV vs. store's LastStoreSyncResourceVersion() (via AtomicFIFO & bookmarks)
- Onboarded in KCM: DaemonSet, Job, StatefulSet, ReplicaSet, ReplicationController, HPA



Consistency Store Cont.

- 3 main primitives in **ConsistencyStore**
- **WroteAt** marks a write that must be read back before the next reconcile
- **Clear** marks a resource as untracked, usually on deletion
- **EnsureReady** returns an error if all writes have not been read back
  - We thought about making this blocking as well, but other objects may be ready to reconcile, blocking could starve workers accidentally.
- General pattern, mark writes in reconciliation loop, and ensure ready before entering, backing off exponentially on error

ConsistencyStore in Code


```
func (dsc *DaemonSetsController) syncDaemonSet(ctx context.Context, key string) error {
     if err := dsc.consistencyStore.EnsureReady(nsName); err != nil {
          return err // Requeues with backoff until cache catches up
     }
     // ...
     pod, err := dsc.kubeClient.CoreV1().Pods(ns).Create(ctx, newPod, metav1.CreateOptions{})
     if err == nil {
          dsc.consistencyStore.WroteAt(nsName, ds.UID, podGR, pod.ResourceVersion)
     }
}
```

Case Study Resolved: DaemonSet

- Replaces brittle wall-clock expectation expiry with deterministic RV comparisons
- When pod informer lags >5m, **EnsureReady** returns a ConsistencyError instead of expiring
- Exposes `stale_sync_skips_total` metric so operators can observe and alert on cache lag

Before: ControllerExpectations (5m TTL)

- Informer watch lags > 5 minutes
- 5m expectation TTL expires silently
- Duplicate Pod B, C, D created every 5m
- Host ports exhausted & nodes crashloop ❌

After: ConsistencyStore (RV Check)

- Informer watch lags > 5 minutes
- LastStoreSyncRV() < WroteRV detected
- Reconcile skipped & requeued with backoff
- Zero duplicate pods created ✅

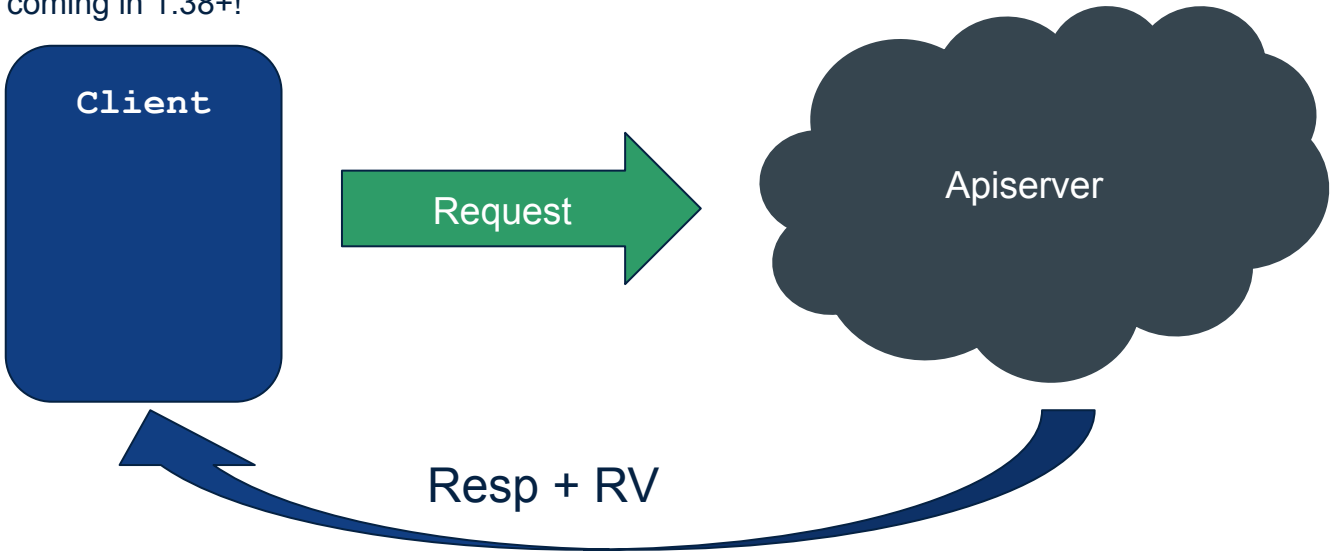
KubeCon CloudNativeCon
North America 2026

Future Work in Kubernetes



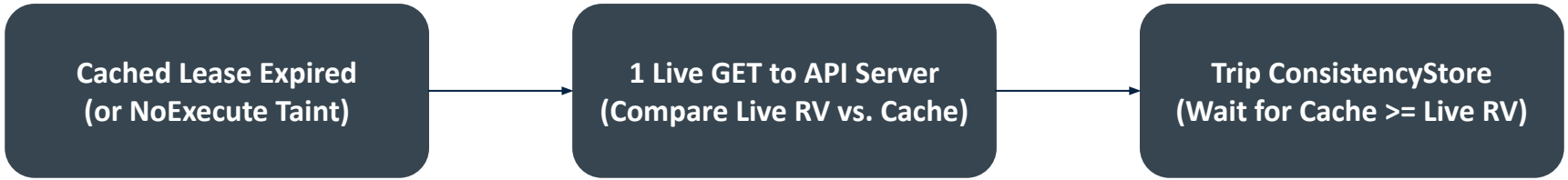
Universality of Resource Version

- All API responses should return a resource version of the object returned/modified
- Currently not the case (Pod Evictions, Bindings, Deletions, etc…)
- Without universal responses, hard to make a generic framework
- Will be coming in 1.38+!



Circuit Breaker Patterns

- Controllers that depend on time based assertions are effectively broken when staleness exceeds the threshold
- Node Lifecycle (Leases) and Taint Eviction controllers are prime examples
- Check staleness of cache before marking a node unhealthy with a live get
- If stale, trip ConsistencyStore circuit breaker to skip checks until cache catches up to live RV (preventing API server overload).



Testing Harnesses

- Property based testing for kubernetes controllers
- Inject failures based on failure scenarios seen in production, but in a controlled manner
- Ensure that properties hold, even with a suite of failures that can have new scenarios added to it
- Onboard controllers onto this framework, and catch failures before they are even seen in production!
- Merges 2 sets of expertise, controller author expectations and test scenario authors



KubeCon CloudNativeCon
North America 2026

Controller Runtime



Intro controller-runtime

- Library to build controllers
- Used in the code generated by kubebuilder and operator-sdk
- Introduces a few abstractions
- Makes the common use-cases simple and the advanced use-cases possible
- Does not change the basic design pattern

Intro controller-runtime


```go
type Controller struct {
    queue workqueue.TypedRateLimitingInterface[cache.ObjectName]
}

func New(factory informers.SharedInformerFactory) (*Controller, error) {
    c := &Controller{
        queue: workqueue.NewTypedRateLimitingQueue(
            workqueue.DefaultTypedControllerRateLimiter[cache.ObjectName]()),
    }
    _, err := factory.Core().V1().Pods().Informer().AddEventHandler(
        cache.ResourceEventHandlerFuncs{
            AddFunc:    func(obj any) { c.enqueue(obj) },
            UpdateFunc: func(_, obj any) { c.enqueue(obj) },
            DeleteFunc: func(obj any) { c.enqueue(obj) },
        })
    return c, err
}

func (c *Controller) Run(ctx context.Context) {
    defer c.queue.ShutDown()
    for c.processNext(ctx) {
    }
}

func (c *Controller) processNext(ctx context.Context) bool {
    key, shutdown := c.queue.Get()
    if shutdown {
        return false
    }
    defer c.queue.Done(key)

    if err := c.reconcile(ctx, key); err != nil {
        c.queue.AddRateLimited(key)
        return true
    }
    c.queue.Forget(key)
    return true
}

func (c *Controller) sync(ctx context.Context, key cache.ObjectName) error {
    return nil
}
```

```go
type reconciler struct{}

func New(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&corev1.Pod{}).
        Complete(&reconciler{})
}

func (r *reconciler) Reconcile(ctx context.Context, req reconcile.Request) (reconcile.Result, error) {
    return reconcile.Result{}, nil
}
```

Reads in client-go


```
type Controller struct {
	clientset kubernetes.Interface
	podLister corelisters.PodLister
}

func (c *Controller) sync(ctx context.Context, ns, name string) error {
	pod, err := c.podLister.Pods(ns).Get(name)
}
```

Reads in controller-runtime


```
type reconciler struct {
	client client.Client
}

func (r *reconcile) Reconcile(ctx context.Context, req ctrl.Request)
(reconcile.Result, error) {
	pod := &corev1.Pod{}
	err := r.client.Get(ctx, req.NamespacedName, pod)
}
```

controller-runtime client

We can’t change all the controllers

So we changed the client

Goals

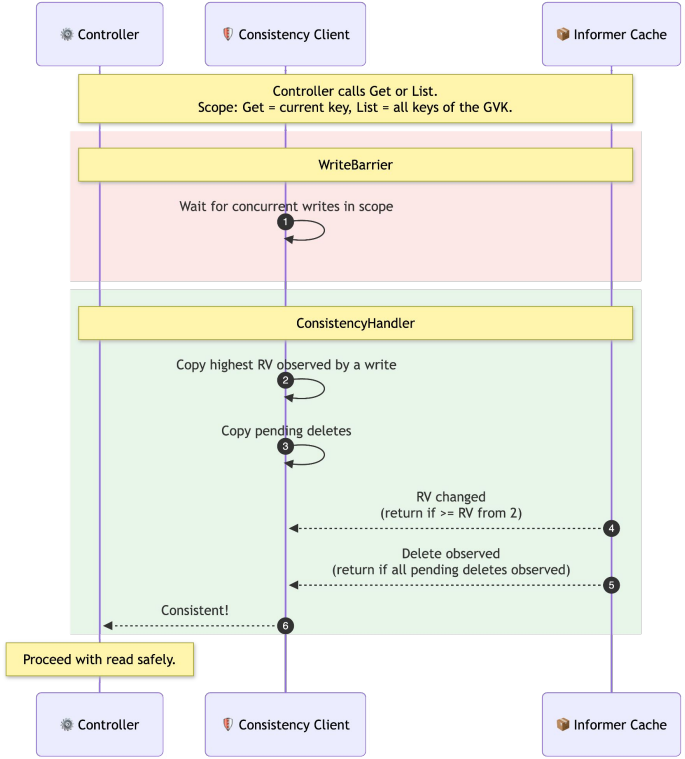
- Any read from the client (`Get` or `List`) contains all writes performed through the same client that were committed by the server at the time of the read
- Reads do not get blocked on writes that started after they started
- Writes themselves do not get blocked waiting for them to be observable by reads

Non-Goals

- Any sort of consistency with writes originating from a client in another binary
- Consistency between multiple clients constructed using the same informer cache
- Consistency for writes that succeeded, but where the information if the write succeeded didn't make it back to the client
- Support DeleteAllOf
- Dealing with configurations where a Transform strips off required fields like RV or UID

Consistent Client

- Layer on top of the existing client and cache
- Writes update the last observed RV or add deletes
- Reads block synchronously
- Get waits for the given key, List for all keys in the GVK
- WriteBarrier to block until concurrent writes are finished
- ConsistencyHandler to block until all writes have made it into the cache



Future Work and Open Issues

- Tracked in #3320
- More fine-granular config as to what writes to wait on
- Making it work with the multiNamespaceCache
- Handling `GenerateName`
- Support cache configurations where the cache doesn’t contain all objects
- Fail writes that use optimistic concurrency clientside

Key Takeaways & Links

- Stale watches are inevitable at scale, wall-clock timeouts cannot guarantee consistency
- Comparable ResourceVersion (http://kep.k8s.io/5504) unlocks client-side freshness checks
- Upstream Kubernetes & client-go (http://kep.k8s.io/5647):
  - Non-blocking **ConsistencyStore** (WroteAt / EnsureReady) + Live-GET Circuit Breakers
  - Track watch lag via `informer_store_resource_version`
- controller-runtime Consistent Client:
  - Drop-in client wrapper blocking Get / List via **WriteBarrier** & **ConsistencyHandler**
  - Trades some efficiency for simplicity, can be activated by just setting a boolean

KubeCon CloudNativeCon
North America 2026

Questions?

