Inside of Kubernetes Controller

2019/09/27

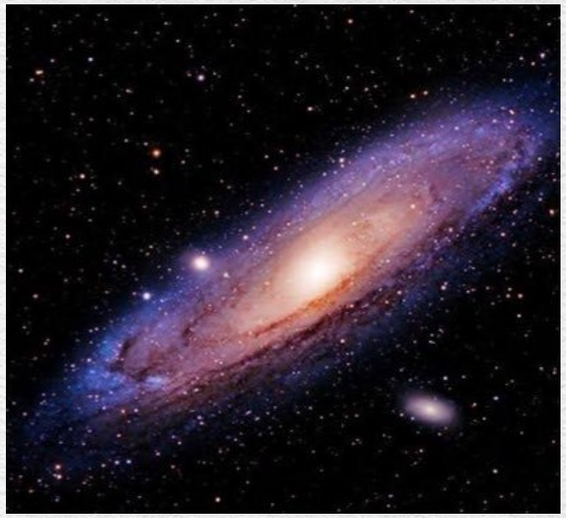
Kubernetes Meetup Tokyo #23 Operator Deep Dive

Who am I

Name: Kenta Iso(@go_vargo)

Job: Infrastructure Engineer

Kubernetes Lover



Mission: Make Kubernetes environment Improve

CKA & CKAD

Purpose & Target

Purpose

- People who know Kubernetes Resources can understand Controller mechanism(include Implement).

Target

- Kubernetes Controller Concept
- Kubernetes Controller mechanism, inside Controller Implement
- Components support Kubernetes Controller

= Controller's High Level Layer ~ Low Level Layer

Not Targeting

Not Target

- Kubernetes Custom Controller + CRD Detail

  ※ However, Understanding controller mechanism
  Helps you to build custom controller.

- Framework & SDK for Kubernetes Custom Controller

  e.g. Kubebuilder、Operator SDK
  controller-runtime, controller-tools

Note

This slide is impressed by **Programming Kubernetes**.

Programming Kubernetes https://programming-kubernetes.info/

Published by O'Reilly Media, Inc.
「Programming Kubernetes」 Author: Stefan Schimanski, Michael Hausenblas

Chap1~2: Kubernetes Concept, API Object
Chap3: client-go
Chap4: CRD
Chap5: code-generator
Chap6~7: Custom Controller
Chap8~9: Custom API Server, CRD Advanced

Book Review Blog(Japanese): https://go-vargo.hatenablog.com/entry/2019/08/05/201546

Agenda

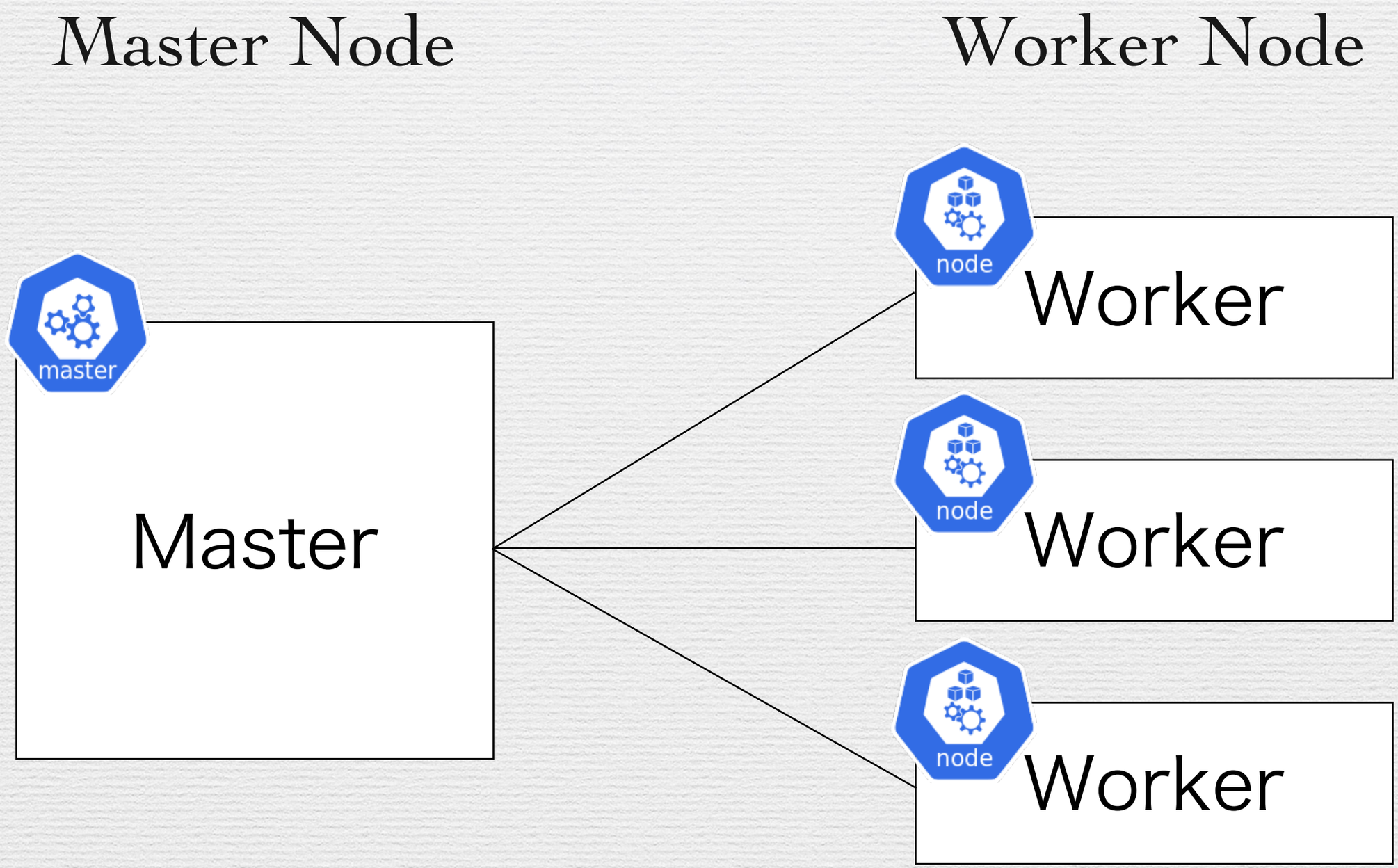
- What is Kubernetes Controller ?
- Control Loop(Reconciliation Loop)
- Controller Library & Components
- Client-Go
  - Informer
  - WorkQueue
- Controller's Cycle, Main Logic
- Controller Summary

- Reference

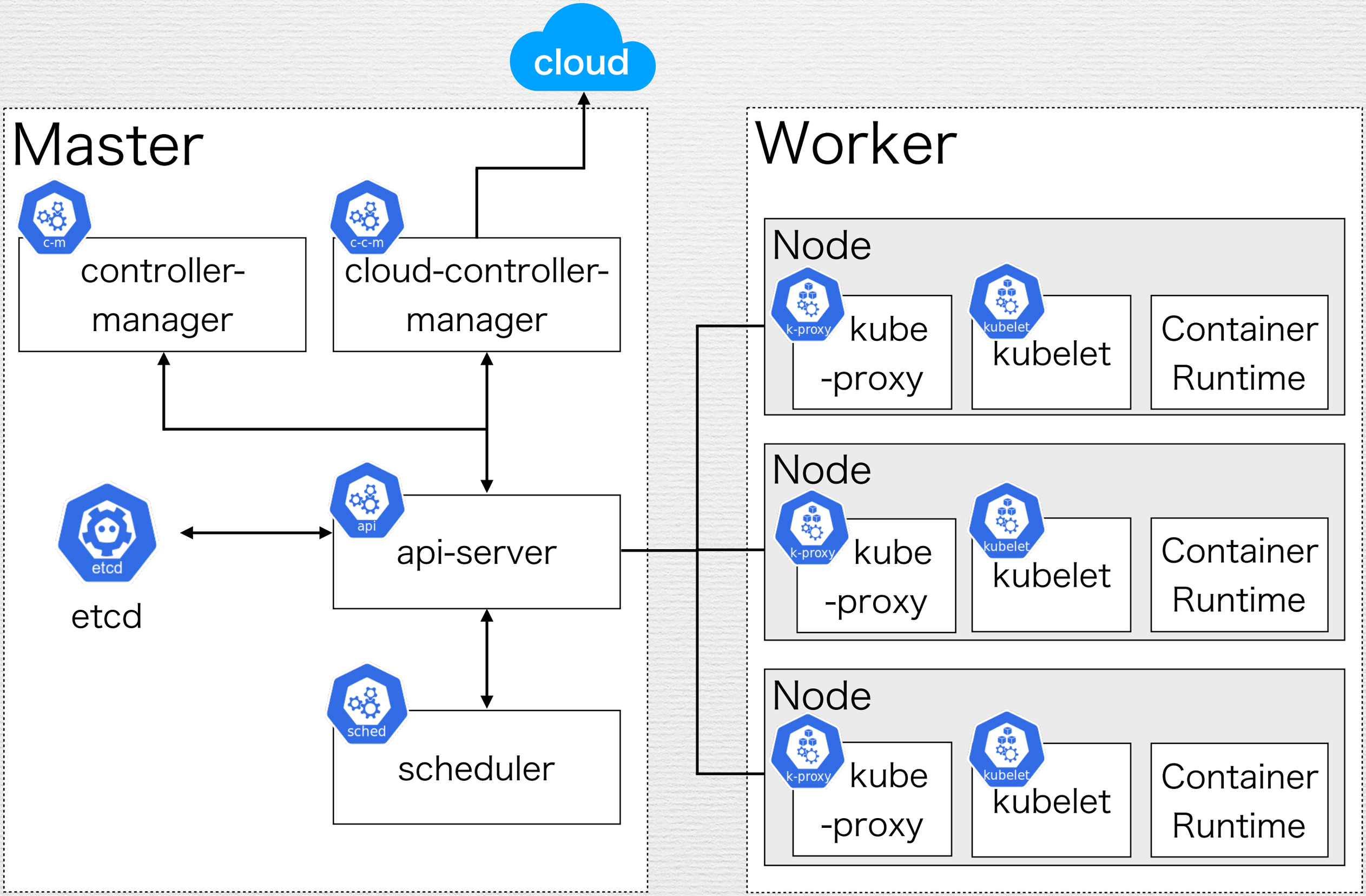
What is Kubernetes Controller ?

～ High Level Architecture ～

Kubernetes Architecture



Kubernetes High Level Architecture



Kubernetes Architecture

Kubernetes is Distributed Architecture
&
Distributed Components

Master:

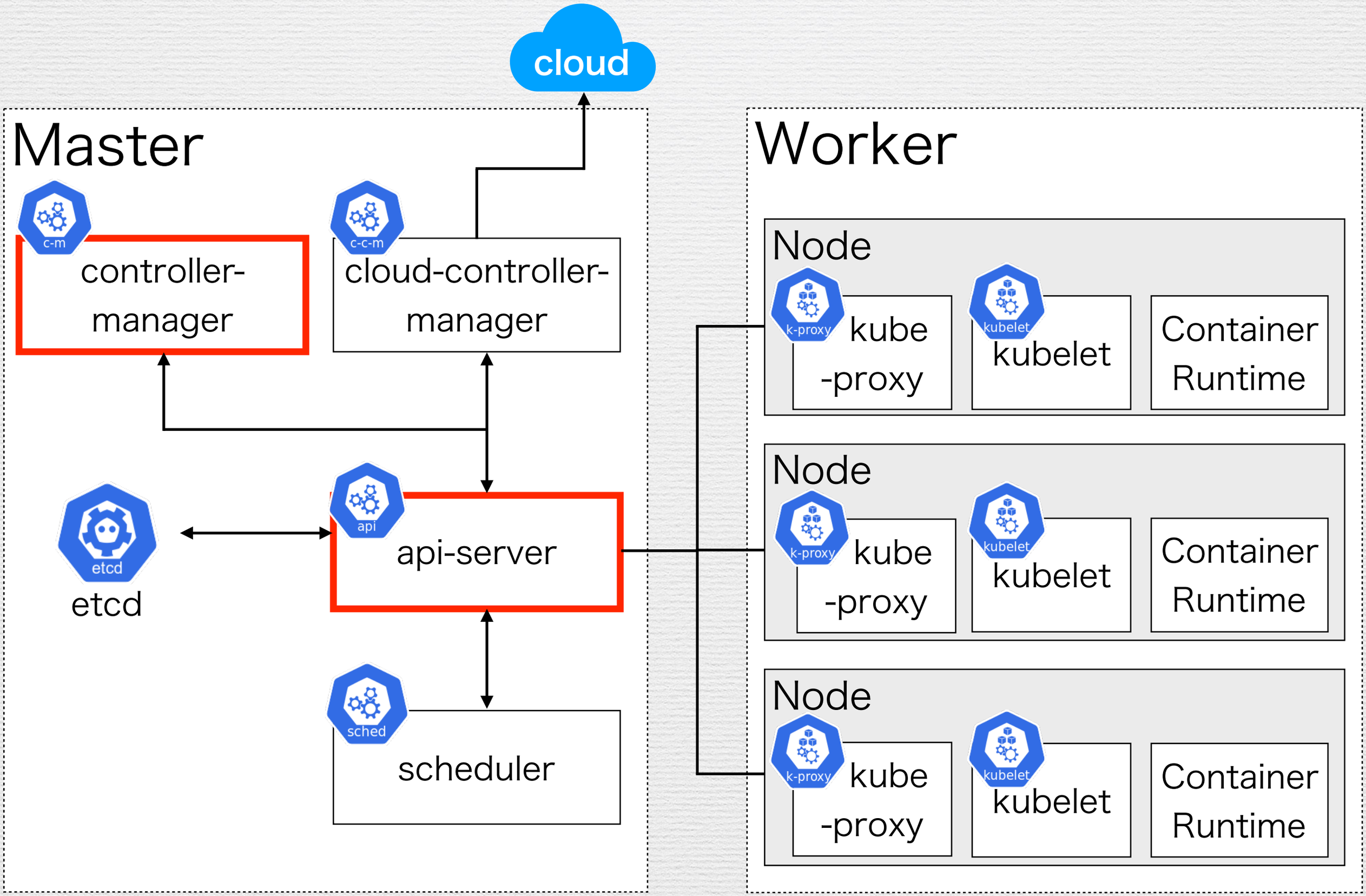
Auth・Authz, Resource(API Object) Management,

Container Scheduling: General management

Worker:

Container Execution

Kubernetes High Level Architecture



api-server / controller-manager

api-server:

api-server receives API Object's CREATE・UPDATE・DELTE (CRUD)requests and execute requests.

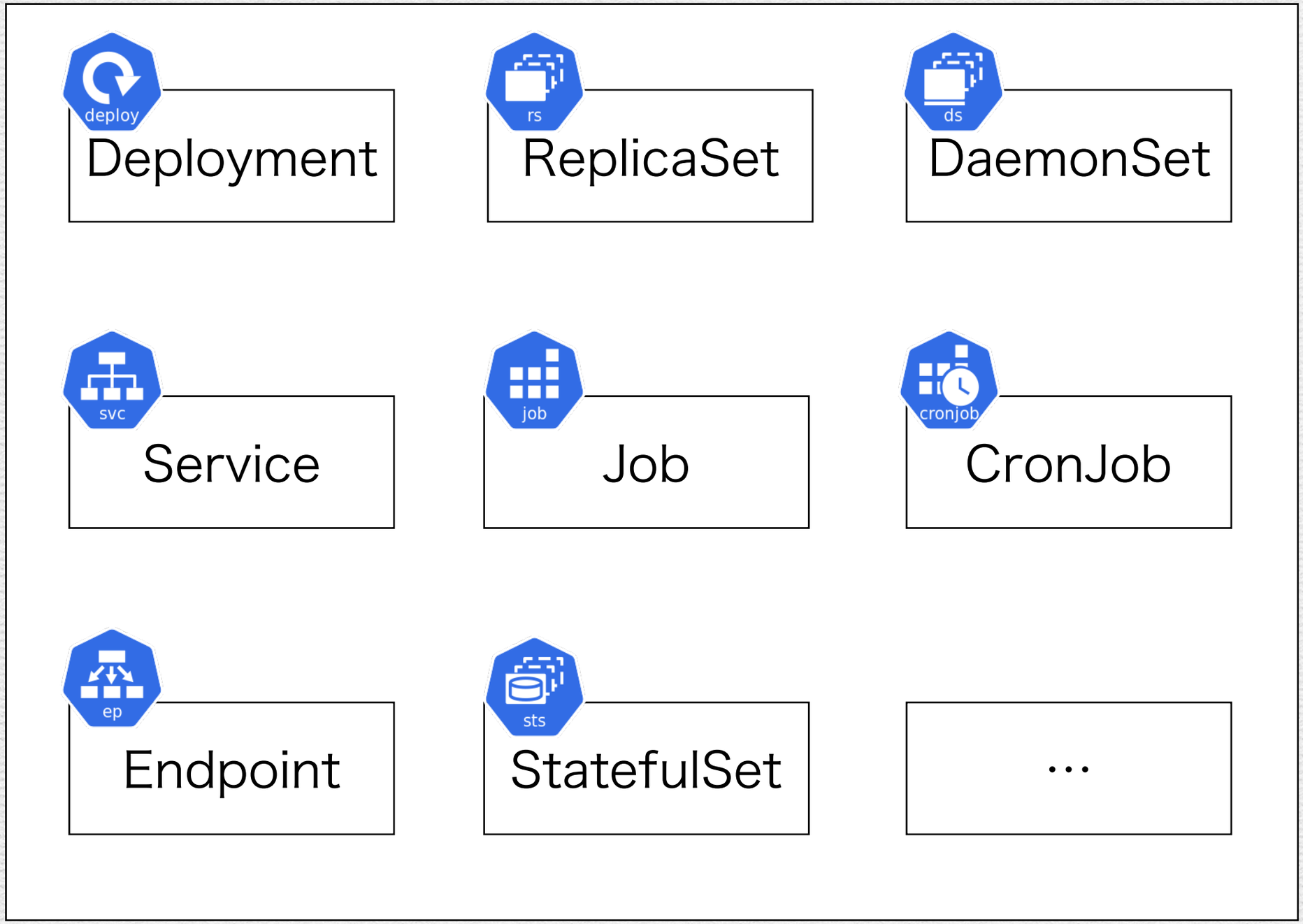
Executed Object data is persisted to etcd(DataStore).
※ component which accesses to etcd is only api-server

controller-manager:

Controller manages Resource(like Deployment, Service…).
controller-manager is a group of multiple controllers.

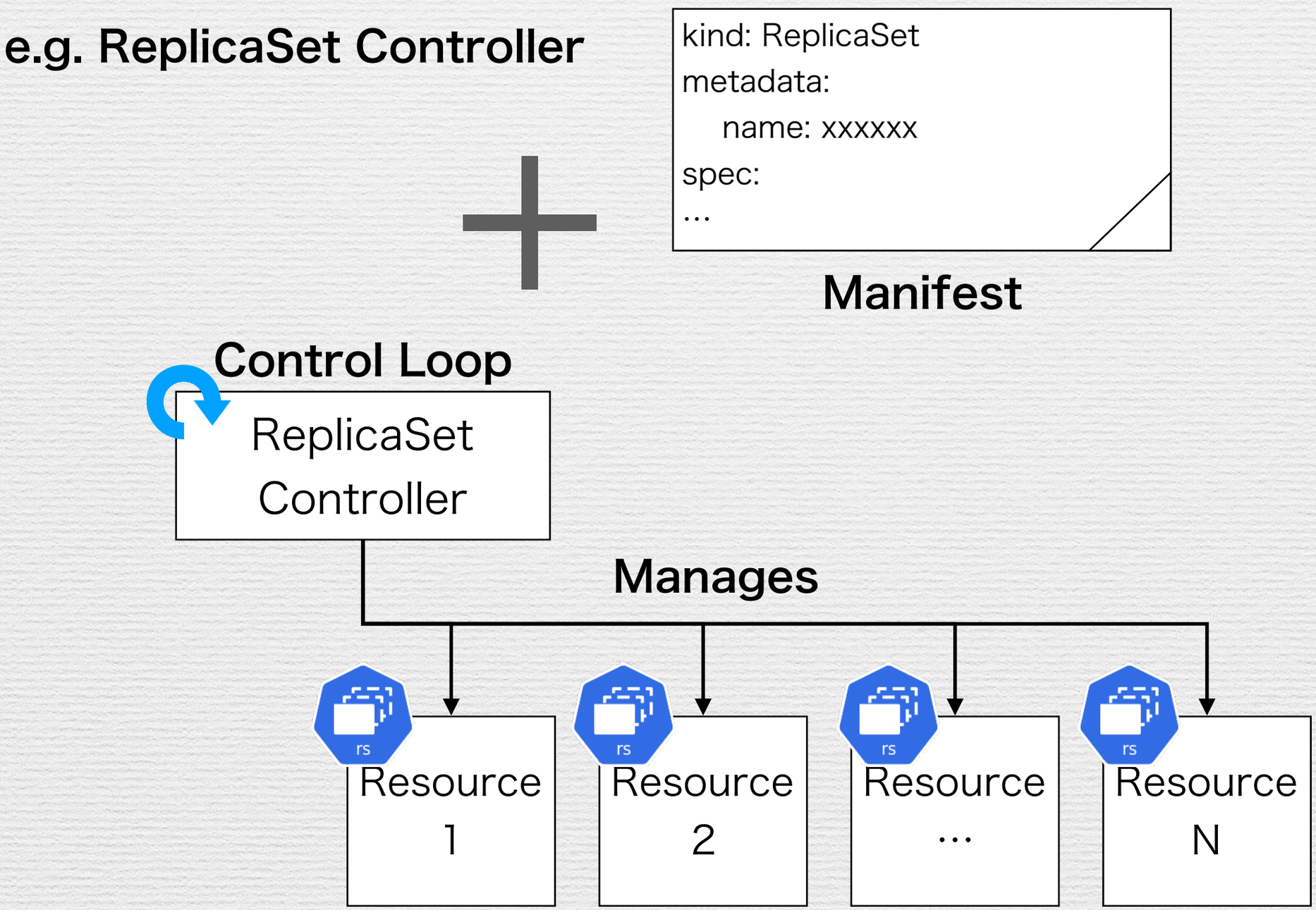
controller-manager

controller-manager: multiple controllers in one binary



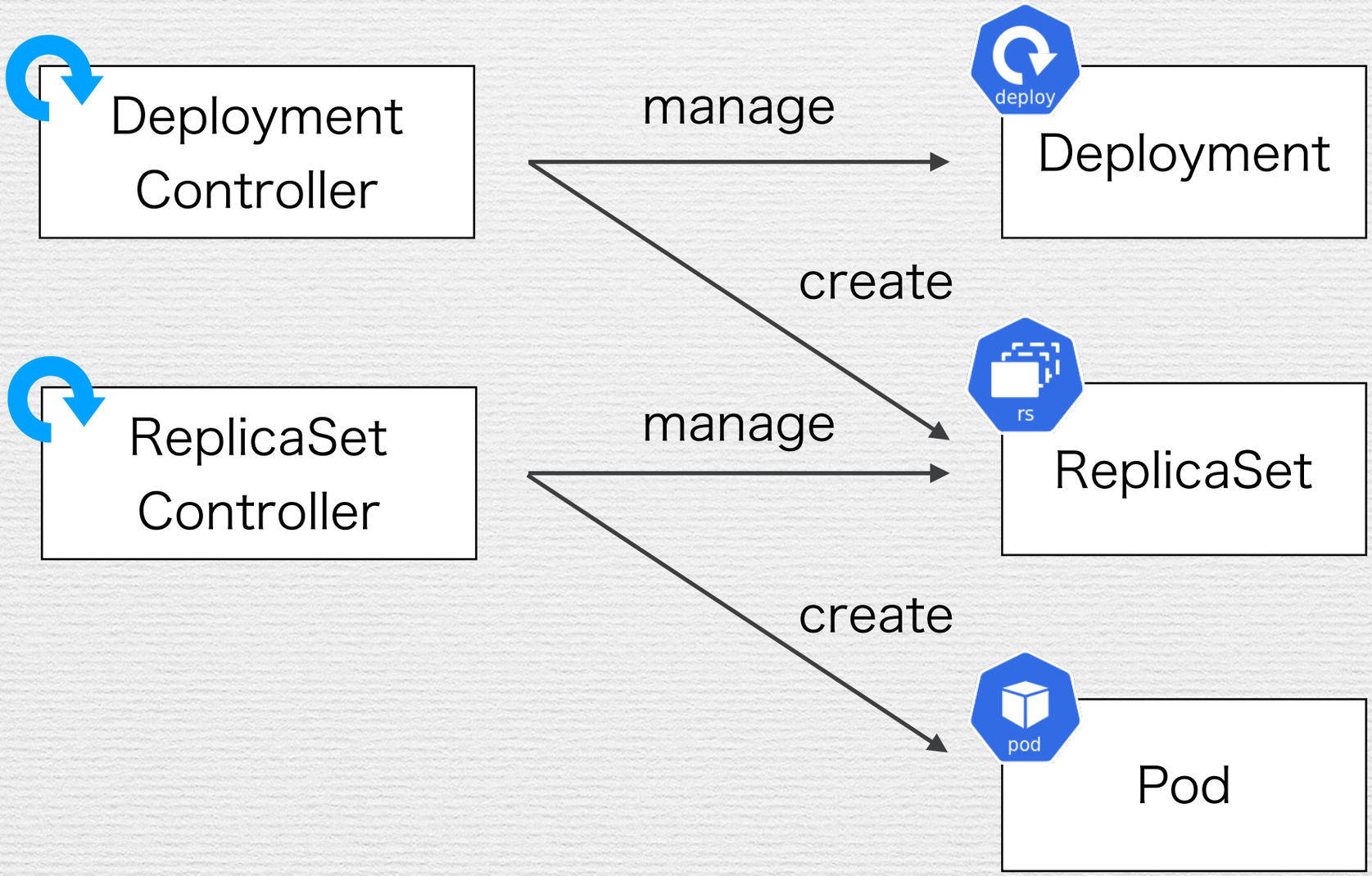
Controller と Resource

Controller manages Resource



Controller and Resource

One Controller manages one Resource



Reference: OwnerReference

There is mechanism called by「ownerReference」which parent resource tags child resource.

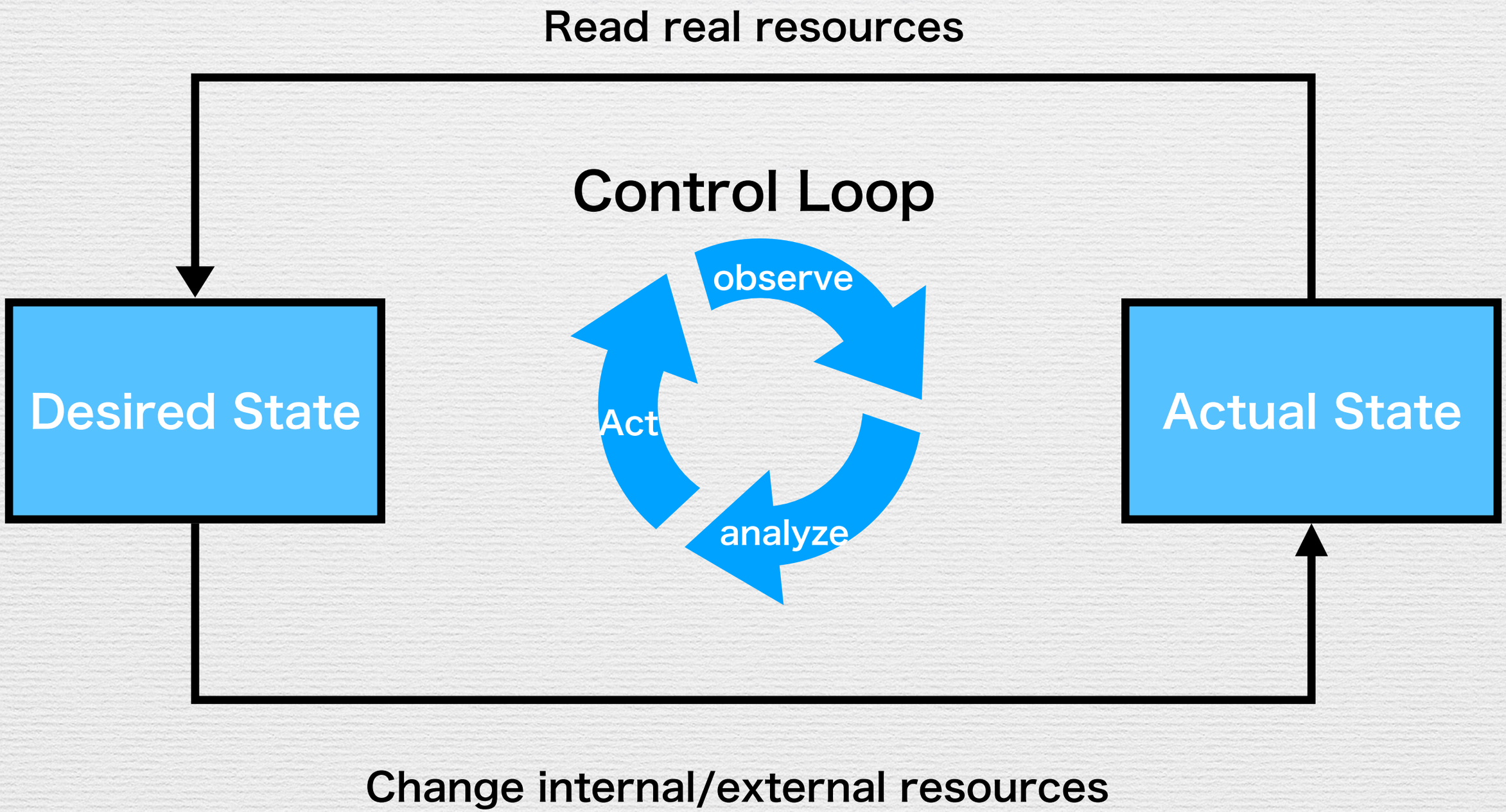
When parent resource is deleted, child resource is deleted by Garbage Collection(GC).

https://kubernetes.io/docs/concepts/workloads/controllers/garbage-collection/

Control Loop
(Reconciliation Loop)

Controller's Concept

Concept: Control Loop(Reconciliation Loop)



Control Loop(Reconciliation Loop)

Controller Loop is Controller's Concept.
※This is called by Reconciliation Loop

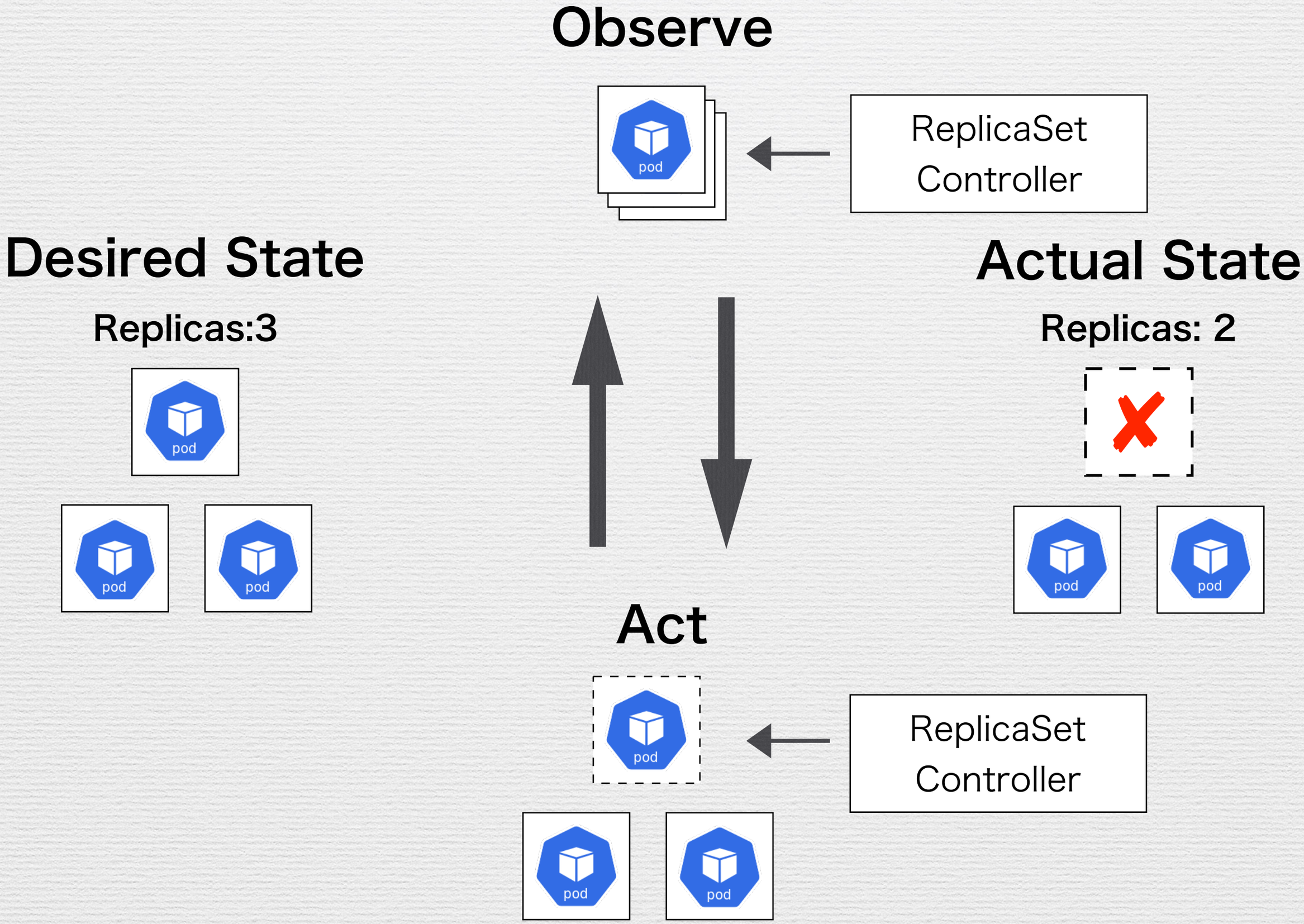
Controller Loop Flow:

1. Read Resource Actual State
2. Change Resource State to Desired State
3. Update Resource Status

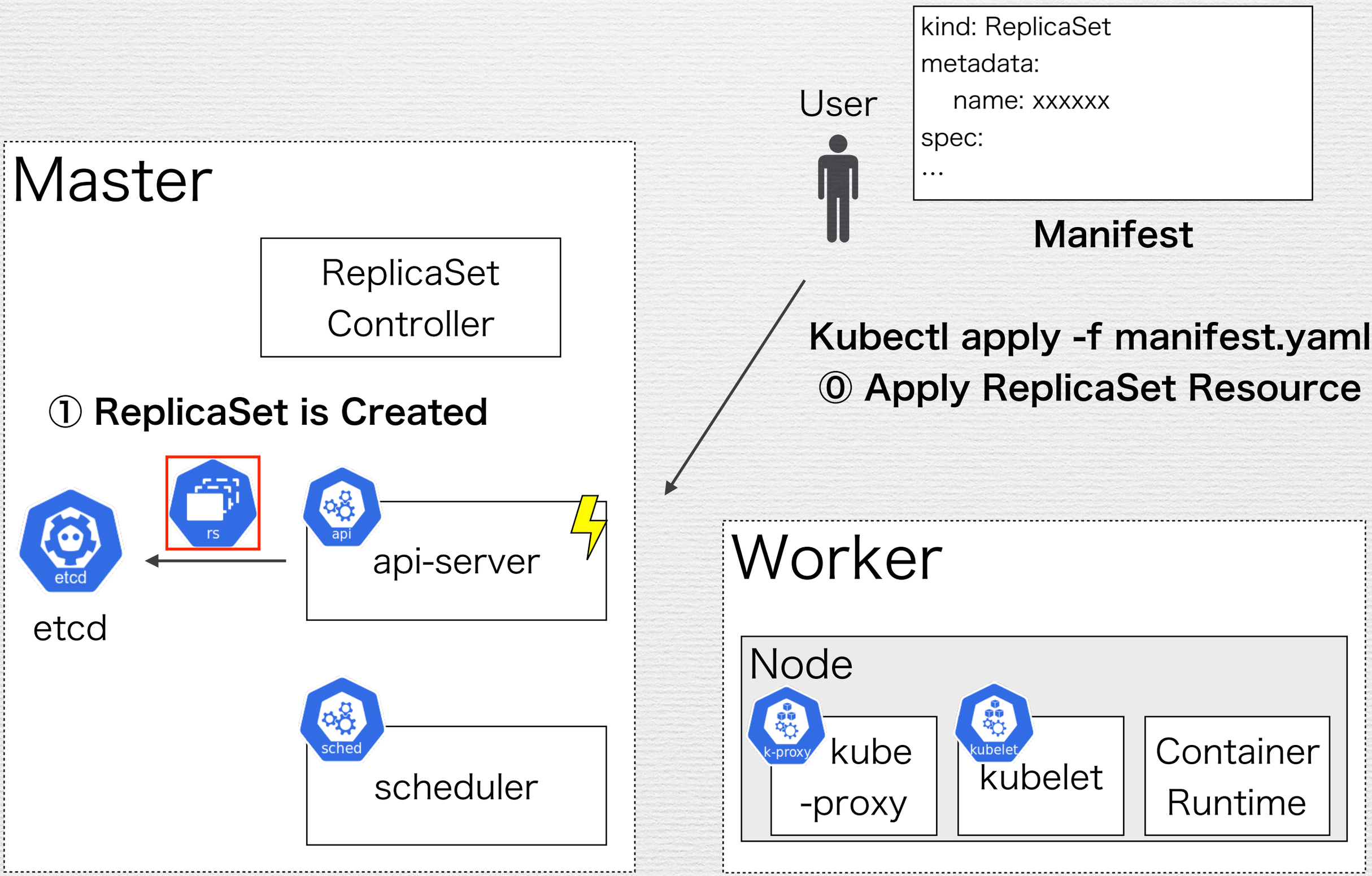
Loop

Declarative API realize immutable Infrastructure by <u>Control Loop</u>.

ReplicaSet Control Loop Example

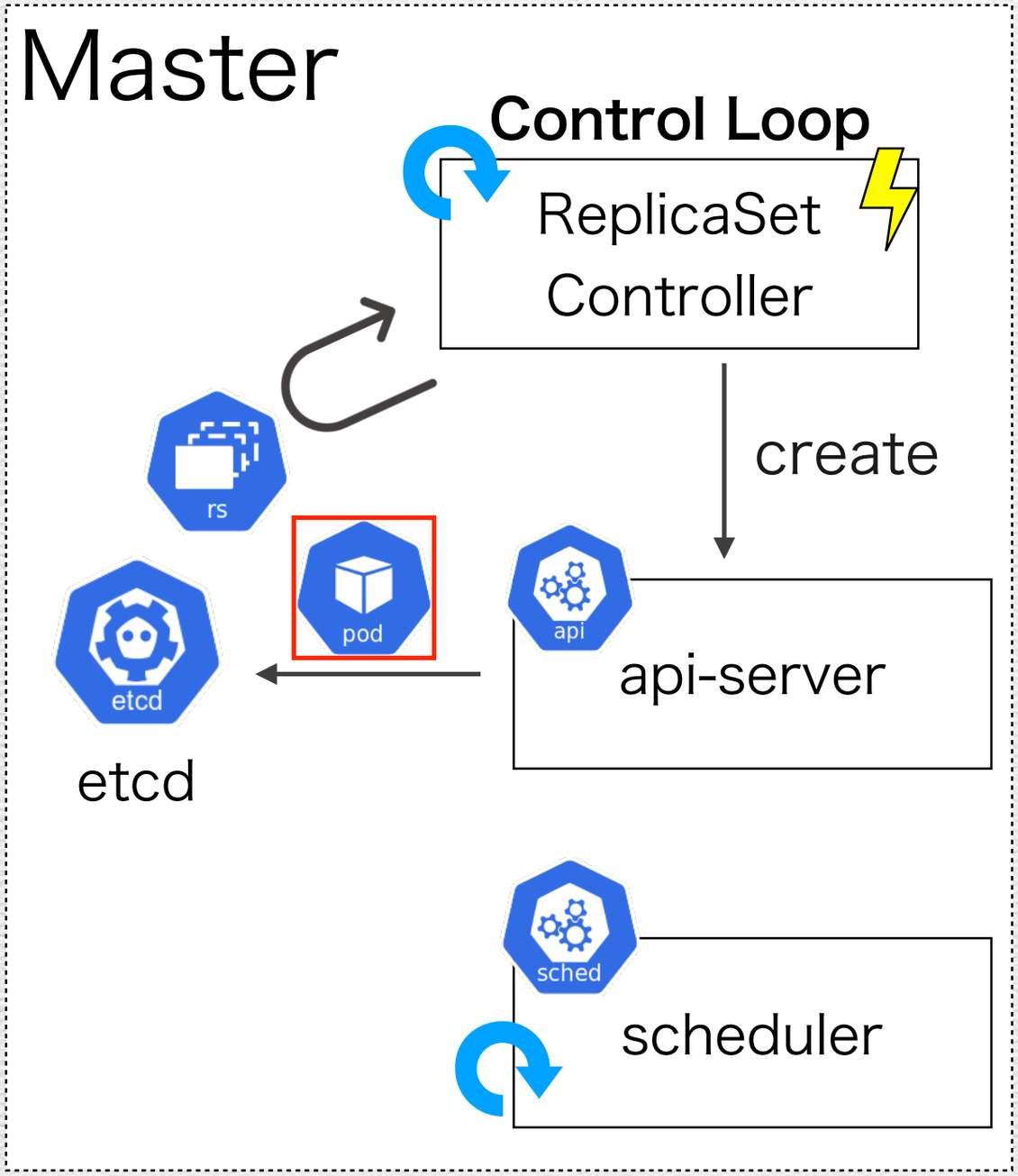


ReplicaSet Apply ～ Container Execution



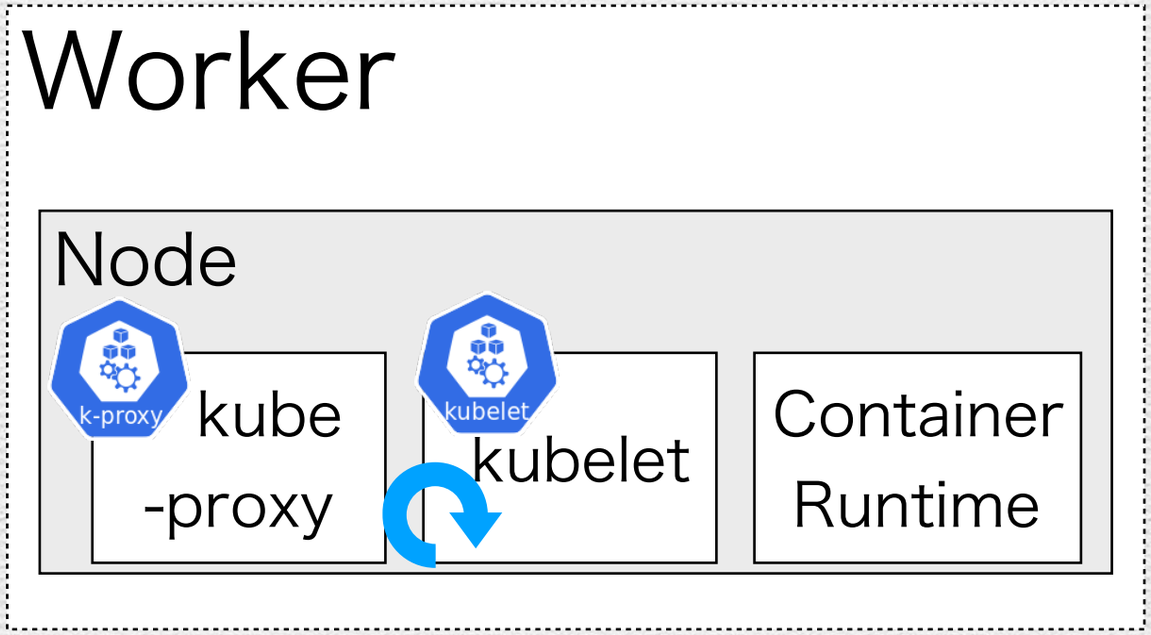
ReplicaSet Apply ～ Container Execution

② ReplicaSet Controller detects ReplicaSet creation



②’ Empty Pod is created which doesn’t have Spec.nodeName by Controller

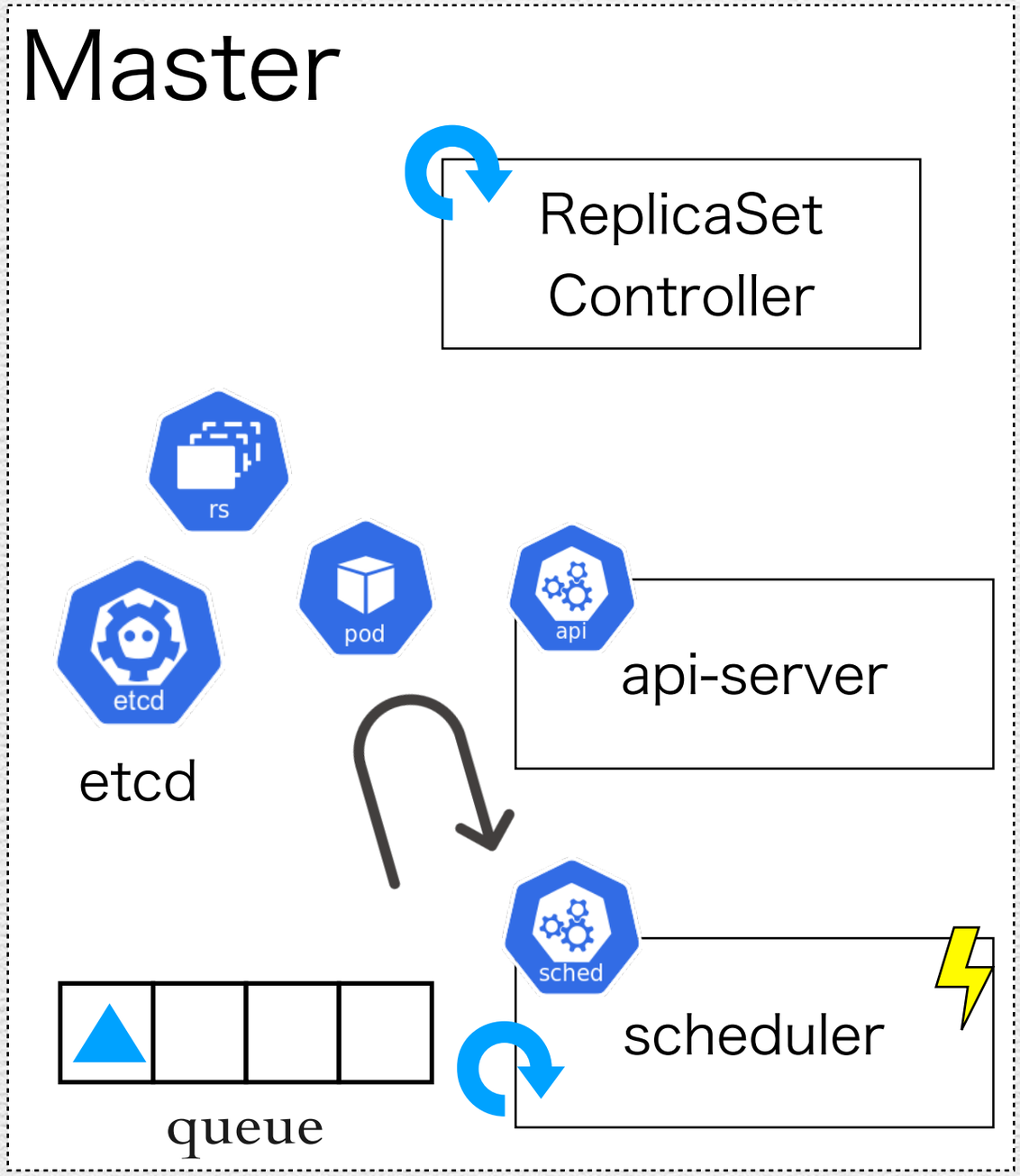
※Not yet delivered to Worker Node

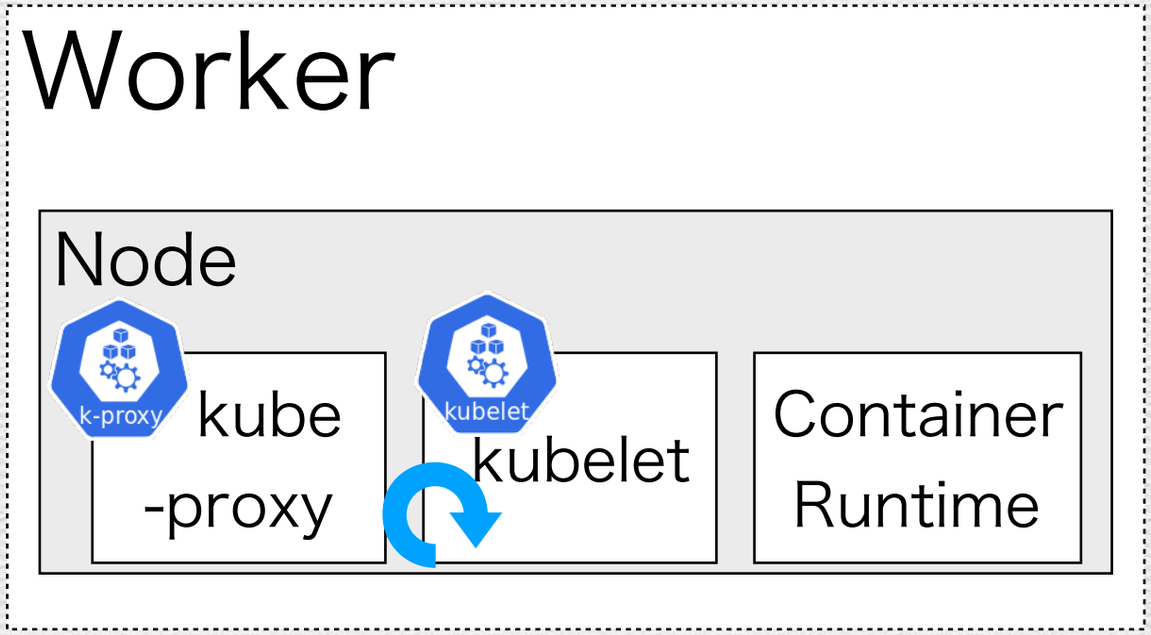


ReplicaSet Apply ～ Container Execution

③ Scheduler detects Pod creation

③' Scheduler enqueues to scheduling queue because pod.Spec.nodeName is empty.

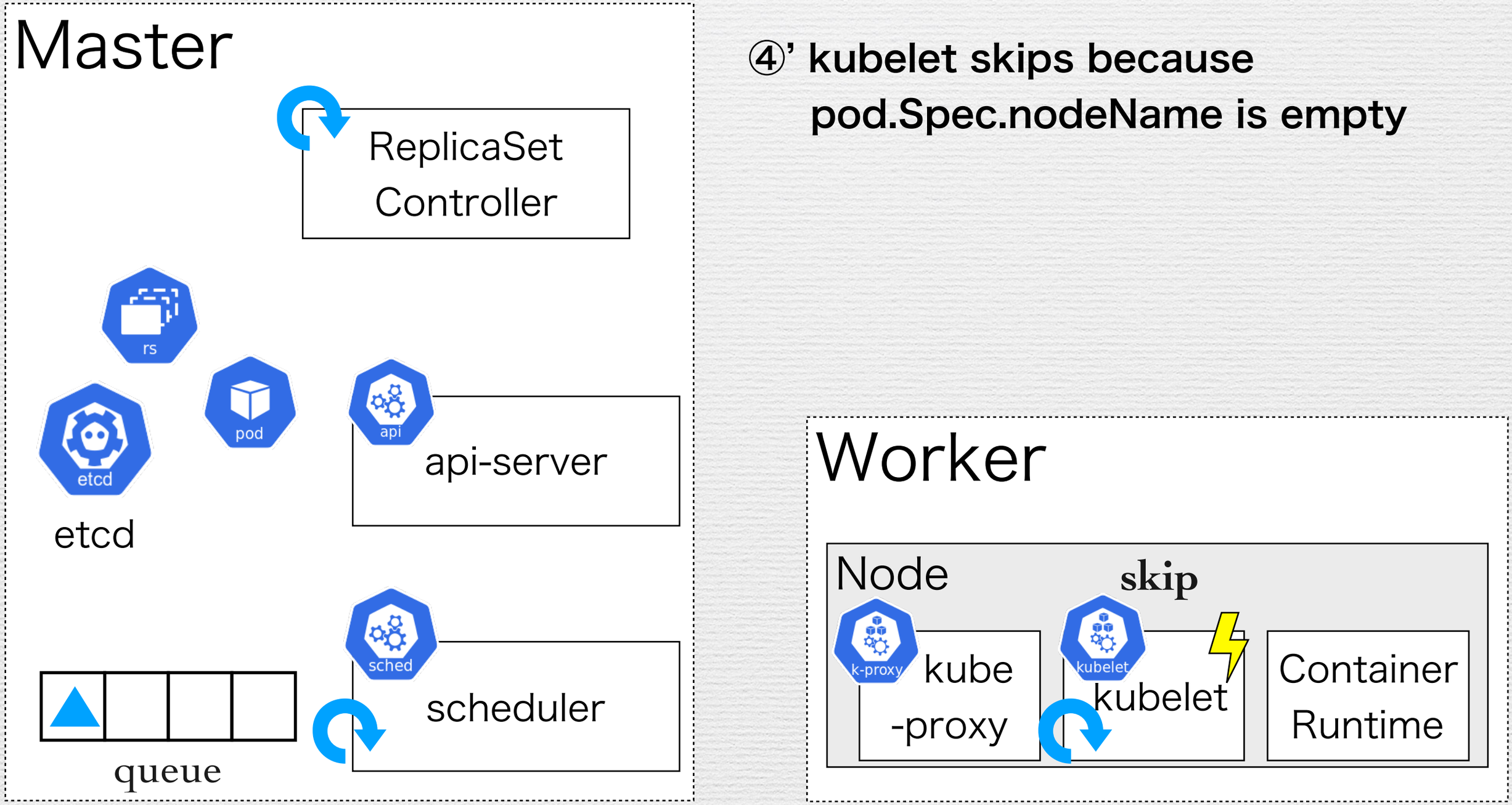




ReplicaSet Apply ～ Container Execution

④ kubelet also detects Pod creation

④’ kubelet skips because
pod.Spec.nodeName is empty

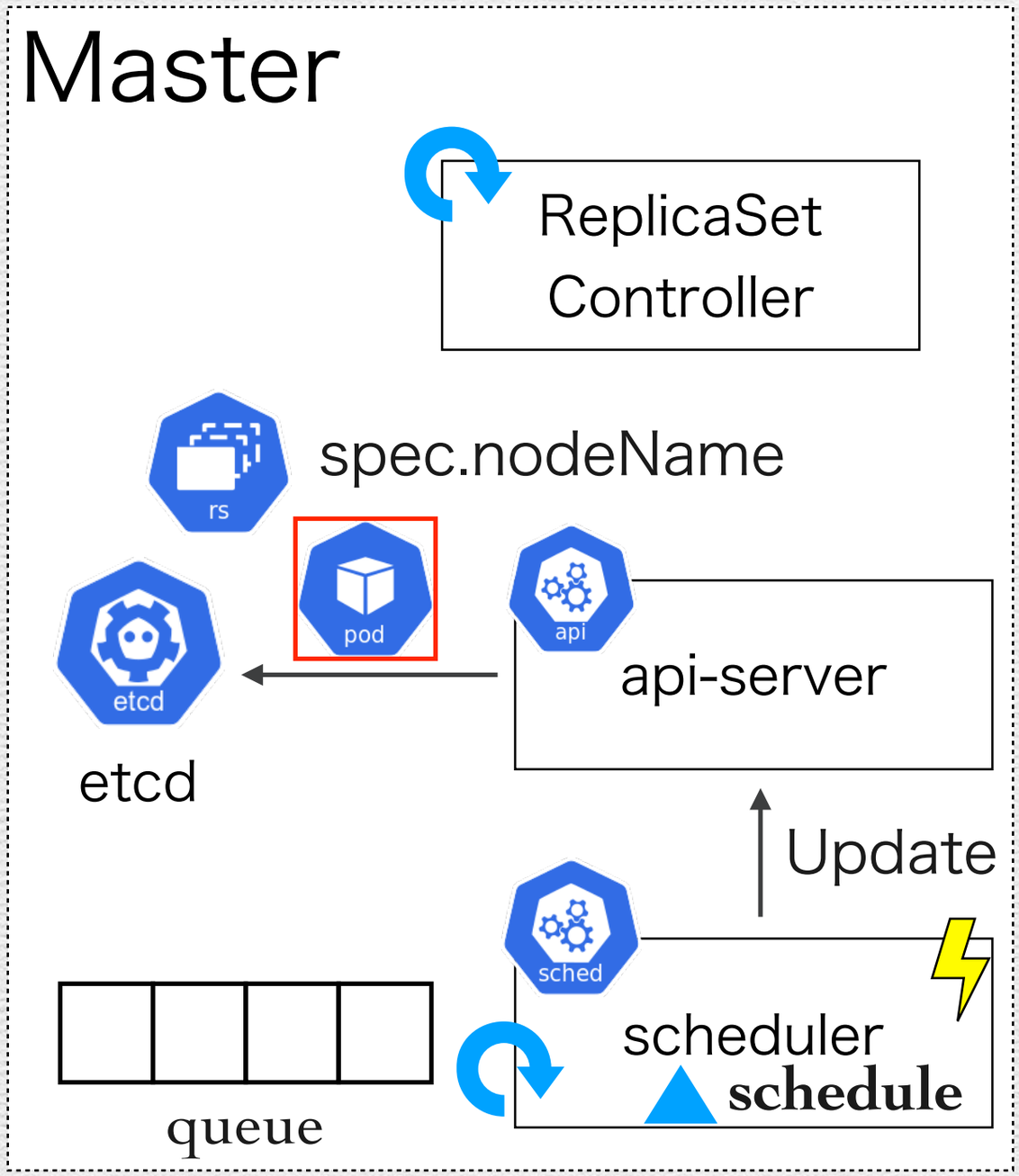


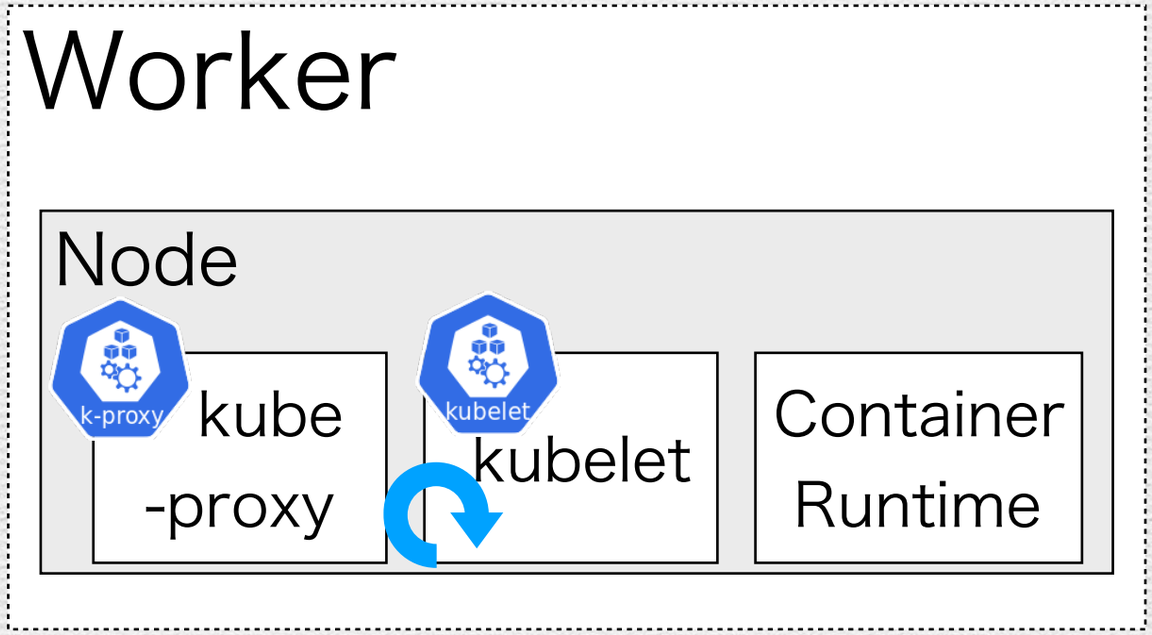
ReplicaSet Apply ～ Container Execution

⑤ Scheduler pops Pod from queue

⑤' Schedules pod to node which can be to assign to

⑤' Updates pod.Spec.nodeName

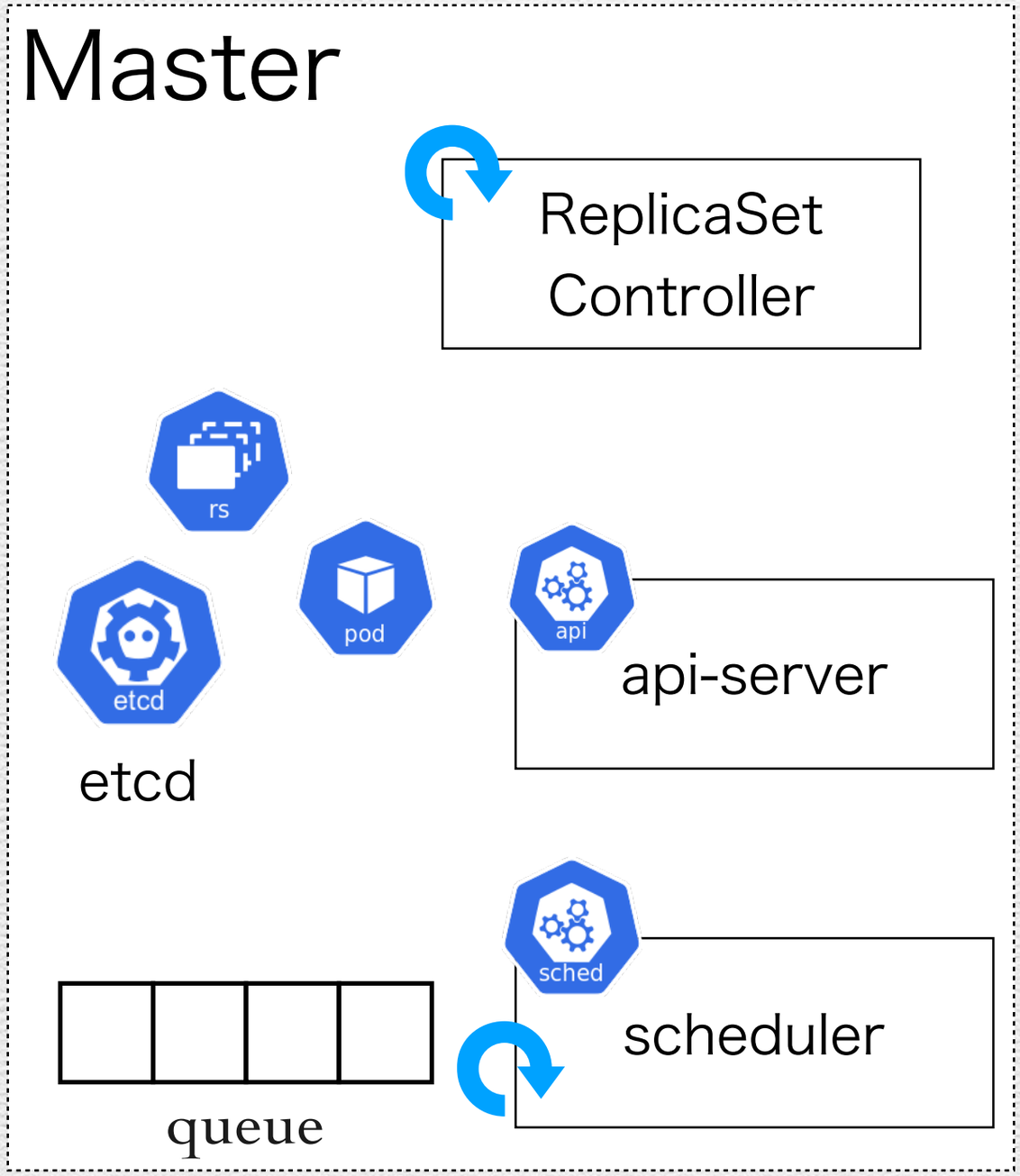


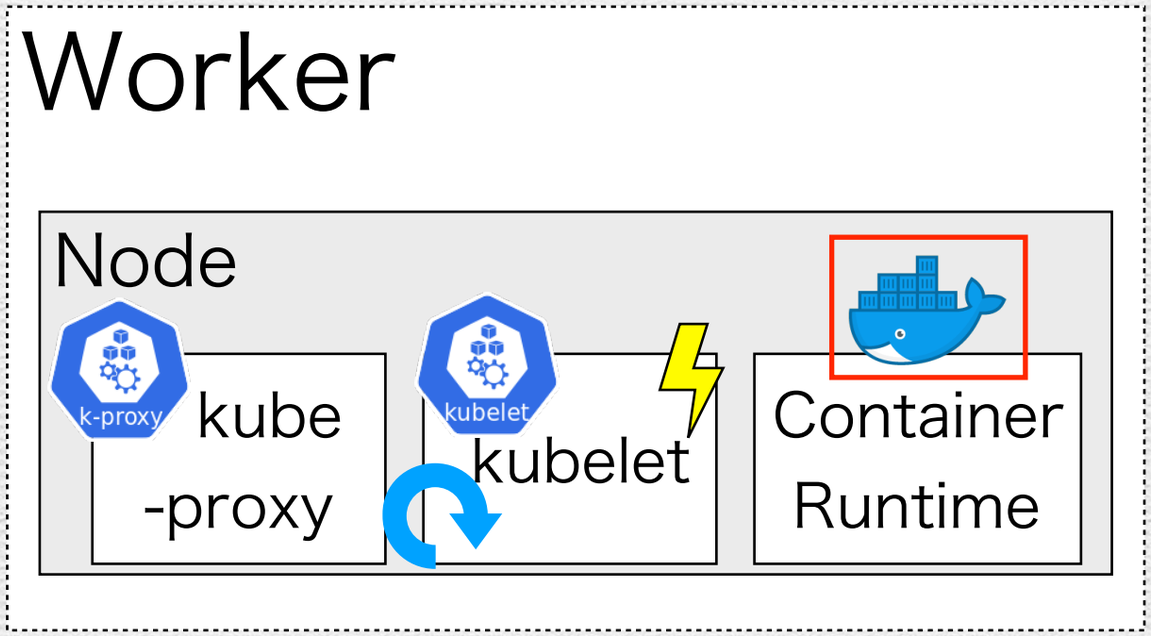


ReplicaSet Apply ～ Container Execution

⑥ kubelet detects Pod update

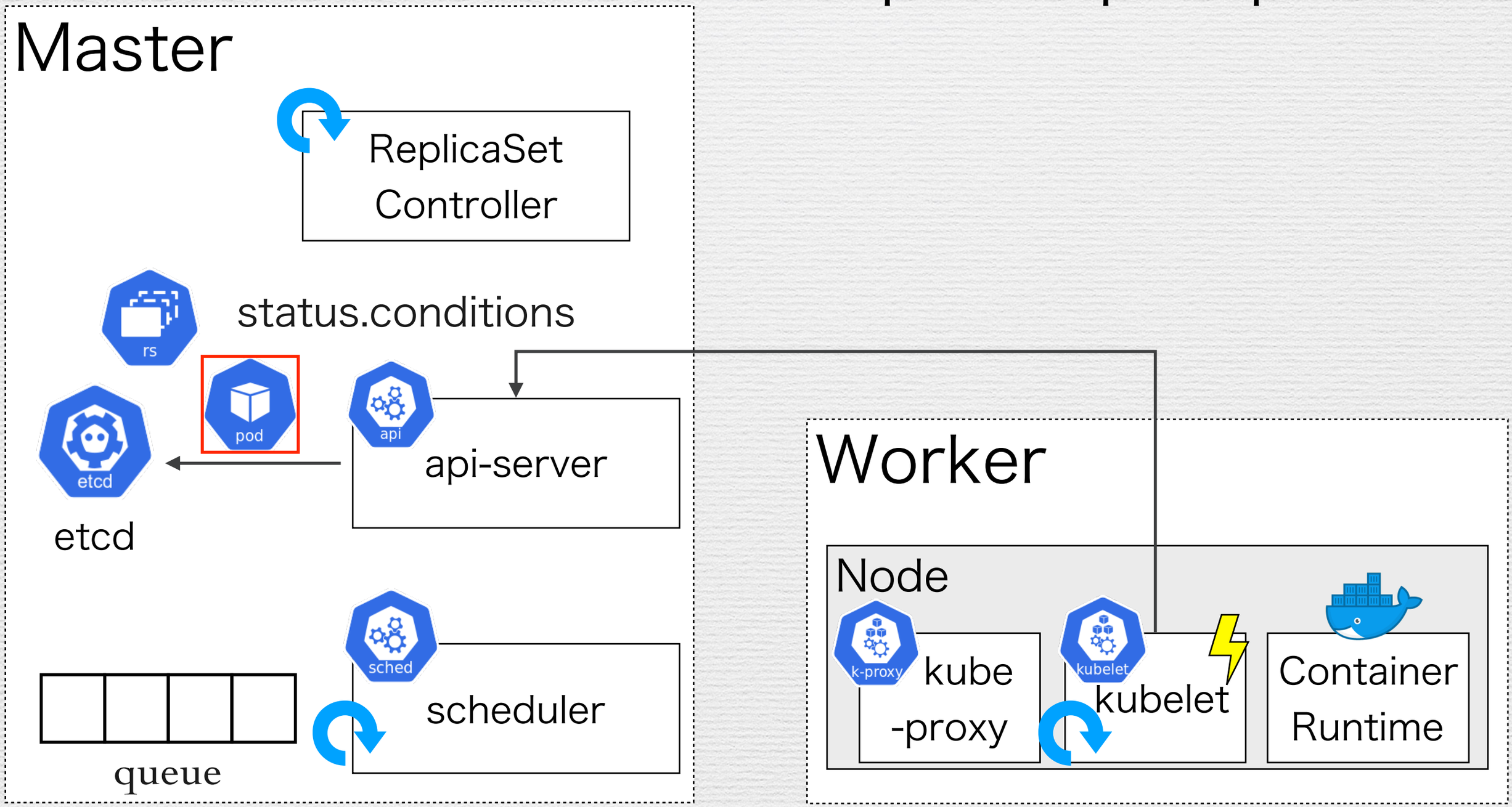
⑥’ kubelet starts up container





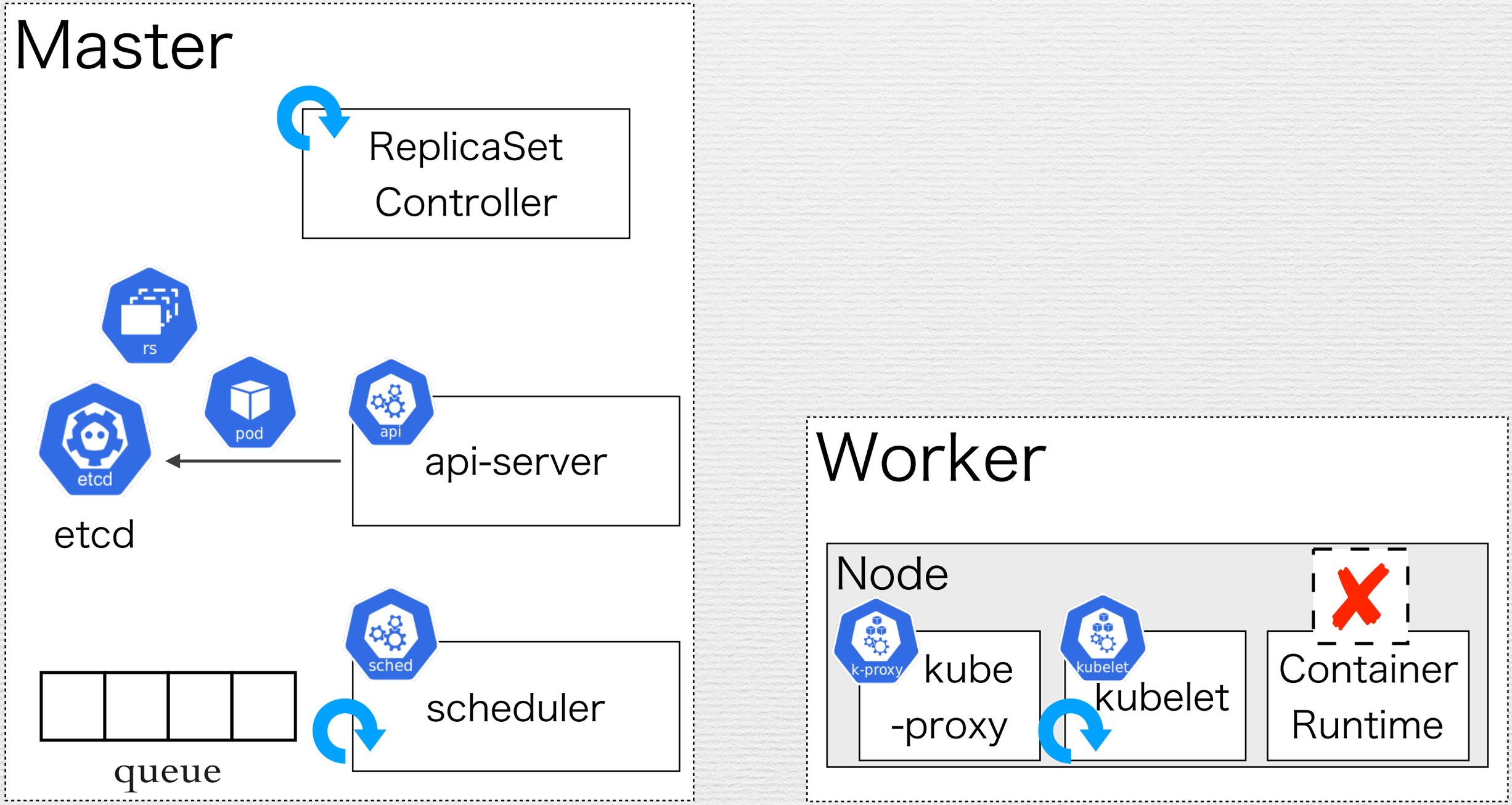
ReplicaSet Apply ～ Container Execution

⑦ kubelet sends request to api-server, and api-server updates pod.Status



ReplicaSet Apply ～ Container Execution

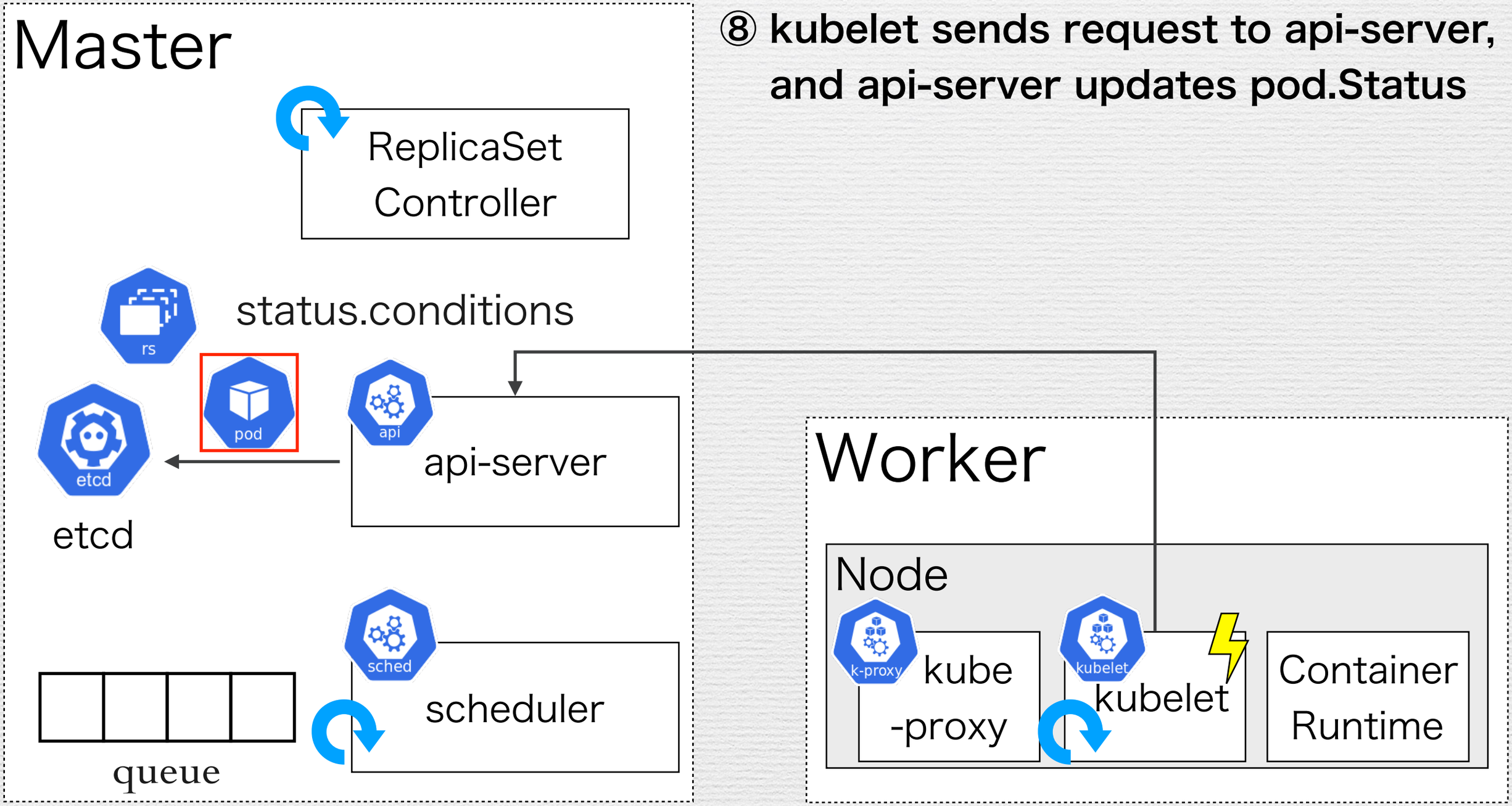
Now suppose the container died for some reason



ReplicaSet Apply ～ Container Execution

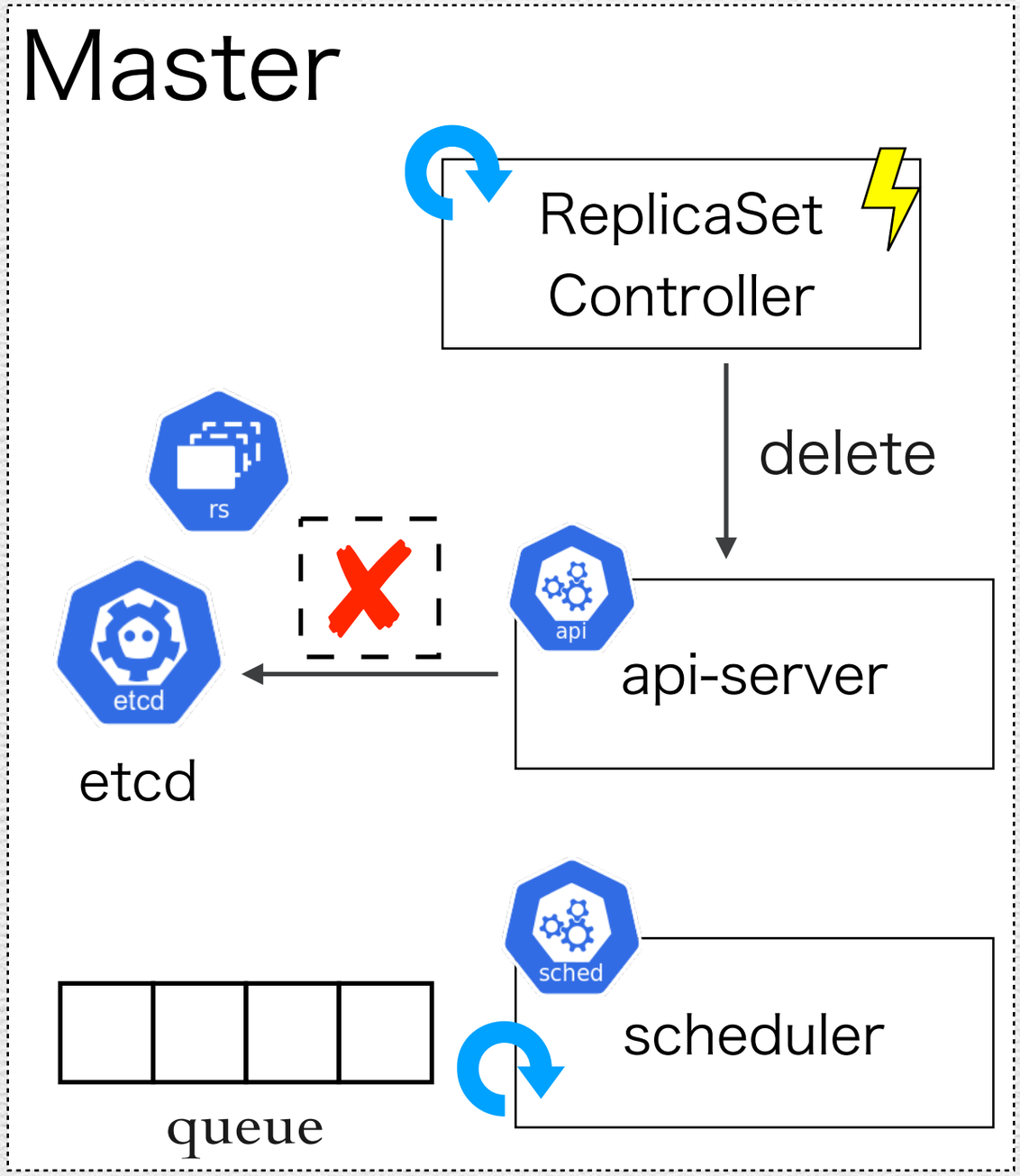
Pod's Terminating

⑧ kubelet sends request to api-server, and api-server updates pod.Status



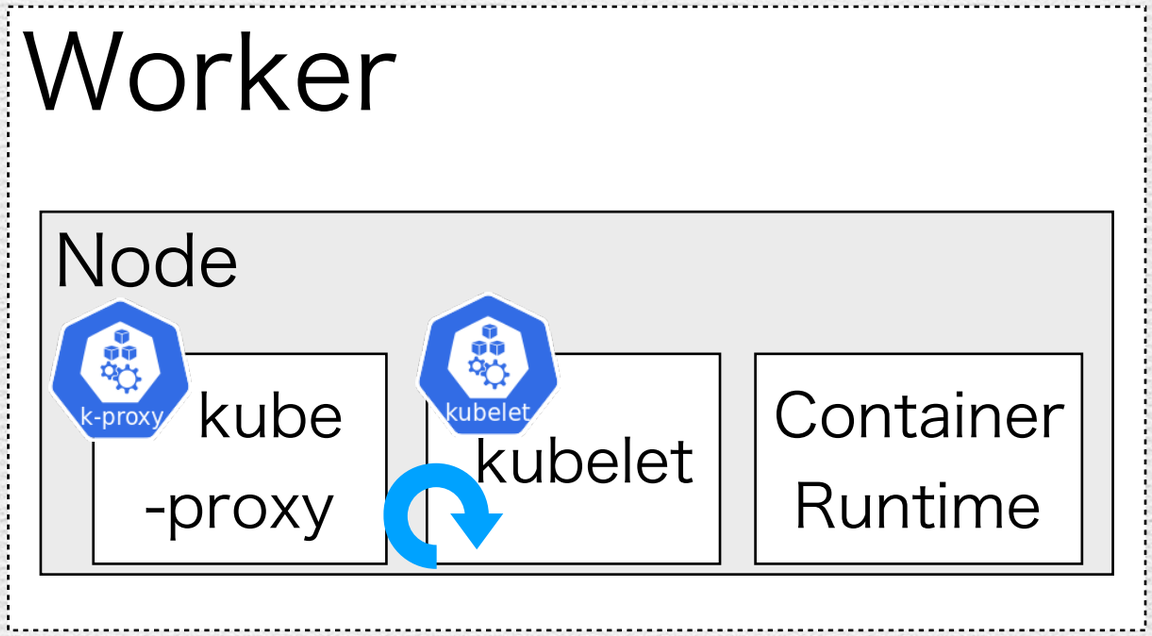
ReplicaSet Apply ～ Container Execution

Pod's Terminating



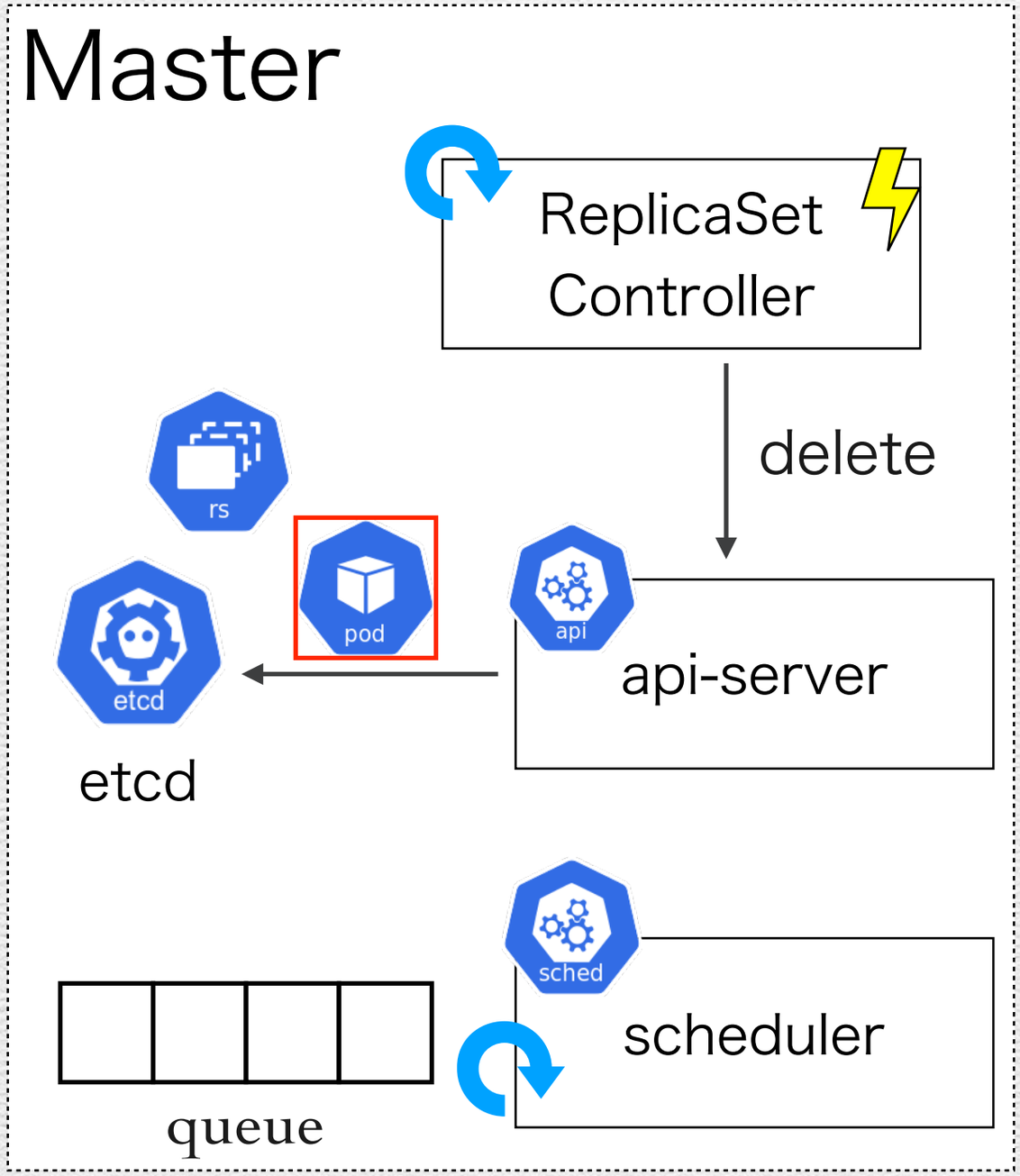
⑨ ReplicaSet Controller detects Pod update

⑨' ReplicaSet Controller deletes Pod



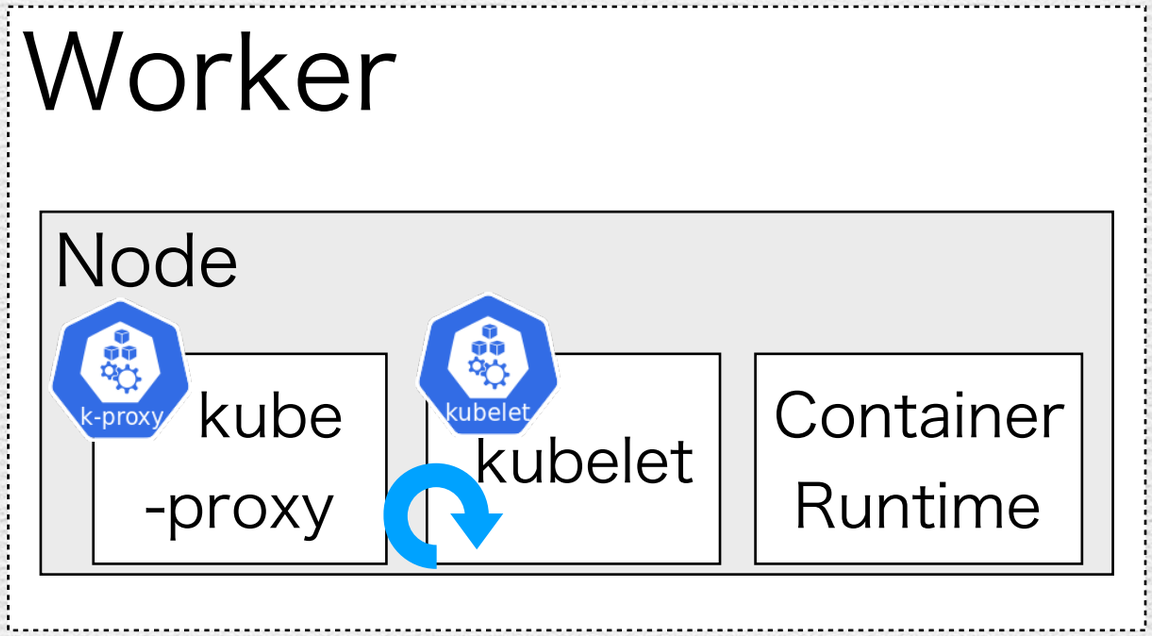
ReplicaSet Apply ～ Container Execution

Reconcile



⑩ ReplicaSet Controller Reconcile in order to become Desired State. Controller recreate Pod

And so, Loop this…
(②～⑩ repeat)



Appendix) Source Code

②

https://github.com/kubernetes/kubernetes/blob/release-1.16/pkg/controller/replicaset/replica_set.go#L487

③

https://github.com/kubernetes/kubernetes/blob/v1.16.0/pkg/scheduler/eventhandlers.go#L436

④

https://github.com/kubernetes/kubernetes/blob/v1.16.0/pkg/kubelet/config/apiserver.go#L33

⑤

https://github.com/kubernetes/kubernetes/blob/v1.16.0/pkg/scheduler/scheduler.go#L535

⑥

https://github.com/kubernetes/kubernetes/blob/release-1.16/pkg/kubelet/kuberuntime/kuberuntime_manager.go#L803

⑦

https://github.com/kubernetes/kubernetes/blob/release-1.16/pkg/kubelet/kubelet.go#L1527

⑧

https://github.com/kubernetes/kubernetes/blob/release-1.16/pkg/kubelet/kubelet.go#L2006

⑨

https://github.com/kubernetes/kubernetes/blob/v1.16.0/pkg/controller/replicaset/replica_set.go#L535

⑩

https://github.com/kubernetes/kubernetes/blob/release-1.16/pkg/controller/replicaset/replica_set.go#L487

Controller and Components

Each Component's Responsibility

- api-server: Resource CRUD
- scheduler: Resource Scheduling
- kubelet: Container starts up
- controller: Reconcile Resource

Each component concentrates on its responsibilities.

= **There is no Orchestra conductor**

who controls the whole and gives instructions.

Controller and Harmony

In Kubernetes, each component works cooperatively…
Do not mean

Even if component is not commanded, the whole consistency is maintained.

So…
Let's think of each component as a Single Controller.

Each Controller concentrates on running each Control Loop.
As a result, Strangely,
you can see that **<u>Kubernetes is in harmony as a whole.</u>**

Kubernetes is jazz improv

Kubernetes is more jazz improv than orchestration.

Co-founder Joe Beda

Core Kubernetes: Jazz Improv over Orchestration

https://blog.heptio.com/core-kubernetes-jazz-improv-over-orchestration-a7903ea92ca

Kubernetes is not an Orchestration.
As jazz improv, by players(Controllers) concentrating on each plays(Control Loop), the whole is consisted of.

Why Controller executes Control Loop?

When we think why Controller executes Control Loop,
「**Event**」is key factor. ※ e.g. Added, Modified, Deleted, Error…

For Kubernetes, which consists of distributed components,
Event is very important. It flows between each component.

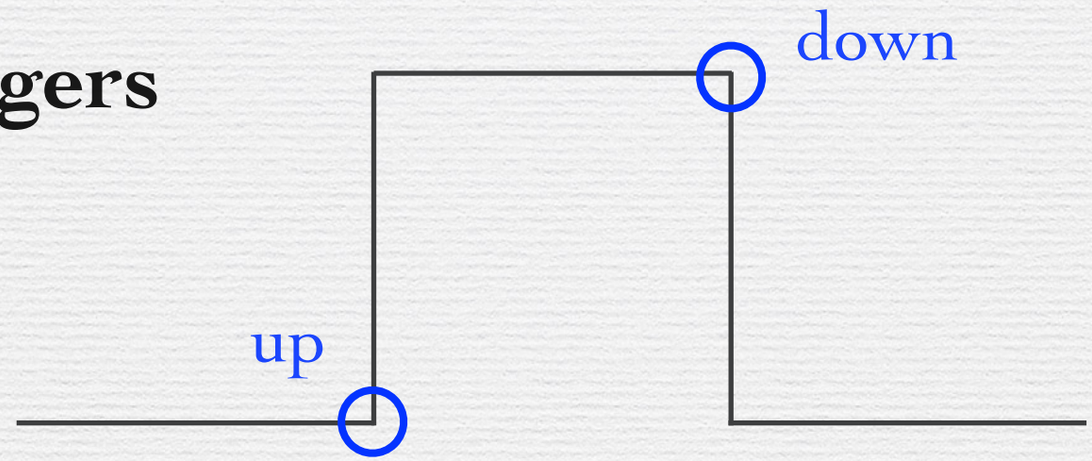
Reference: Events, the DNA of Kubernetes
https://www.mgasch.com/post/k8sevents/

There are two way of how we think event triggers

- **Edge-driven Triggers**
- **Level-driven Triggers**

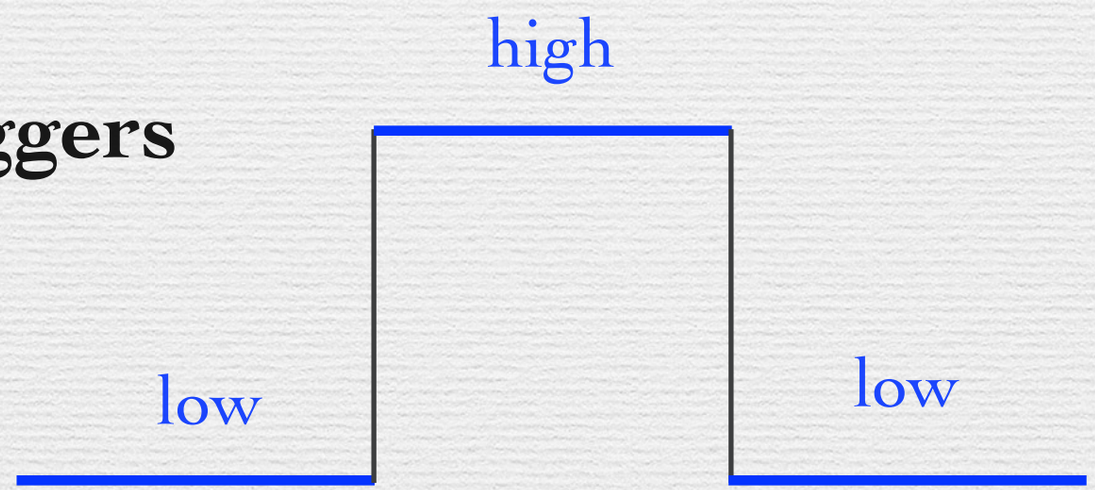
Appendix) Edge vs. Level

Edge-driven Triggers



Trigger when event occurs

Level-driven Triggers



Trigger when in a specific state

Reference: Level Triggering and Reconciliation in Kubernetes

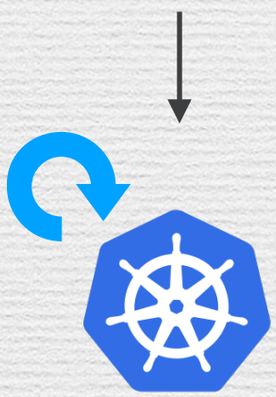
https://hackernoon.com/level-triggering-and-reconciliation-in-kubernetes-1f17fe30333d

Why Controller executes Control Loop?

In order to think that why controller executes control loop,
we suppose Kubernetes takes Procedure process, not Reconcile.

Actual Kubernetes

```
kind: xxxxxxxxx
metadata:
    name: xxxxxx
spec:
...
```



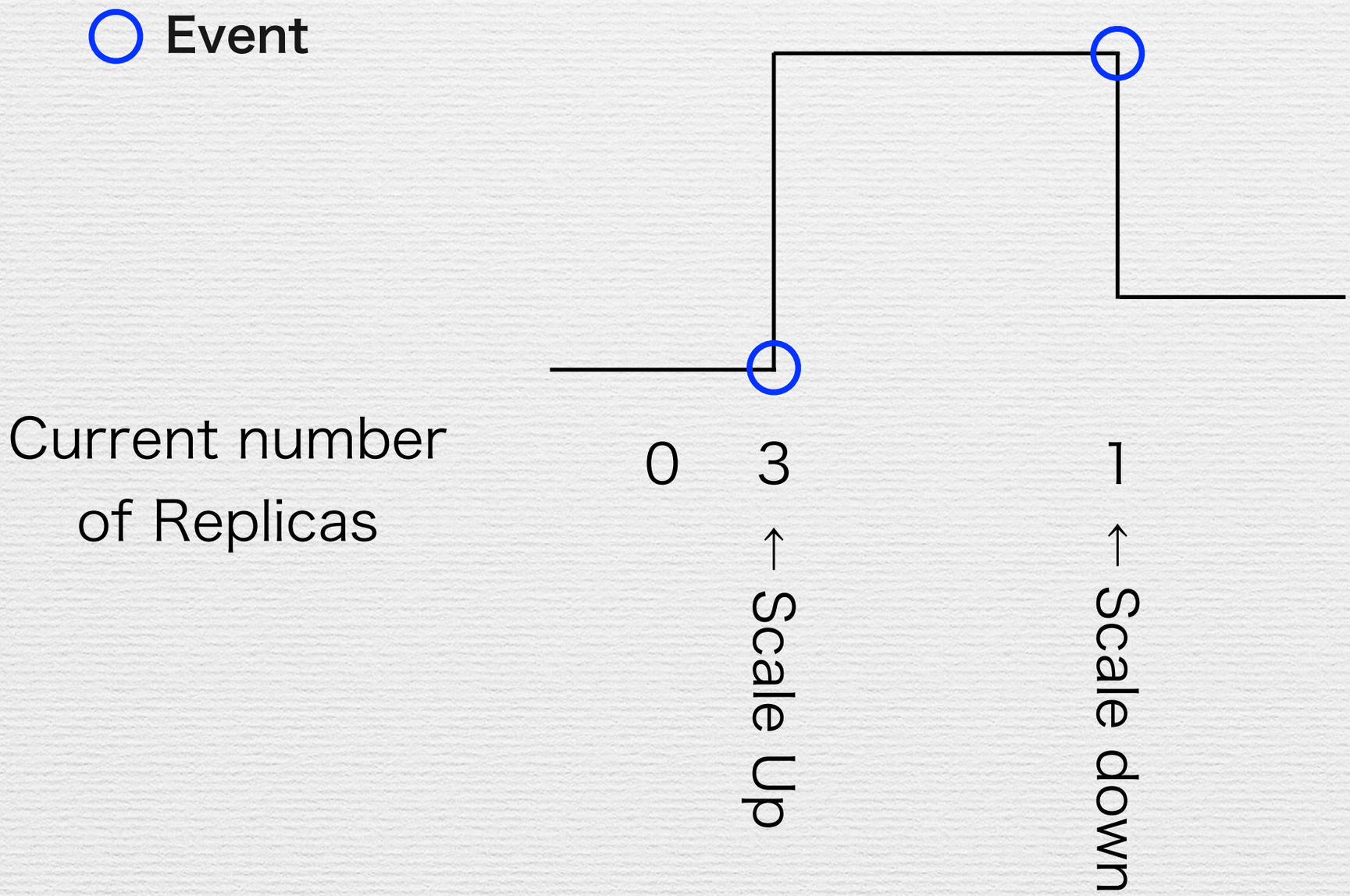
Reconcile

Assumption

Procedure

If Controller is Procedure

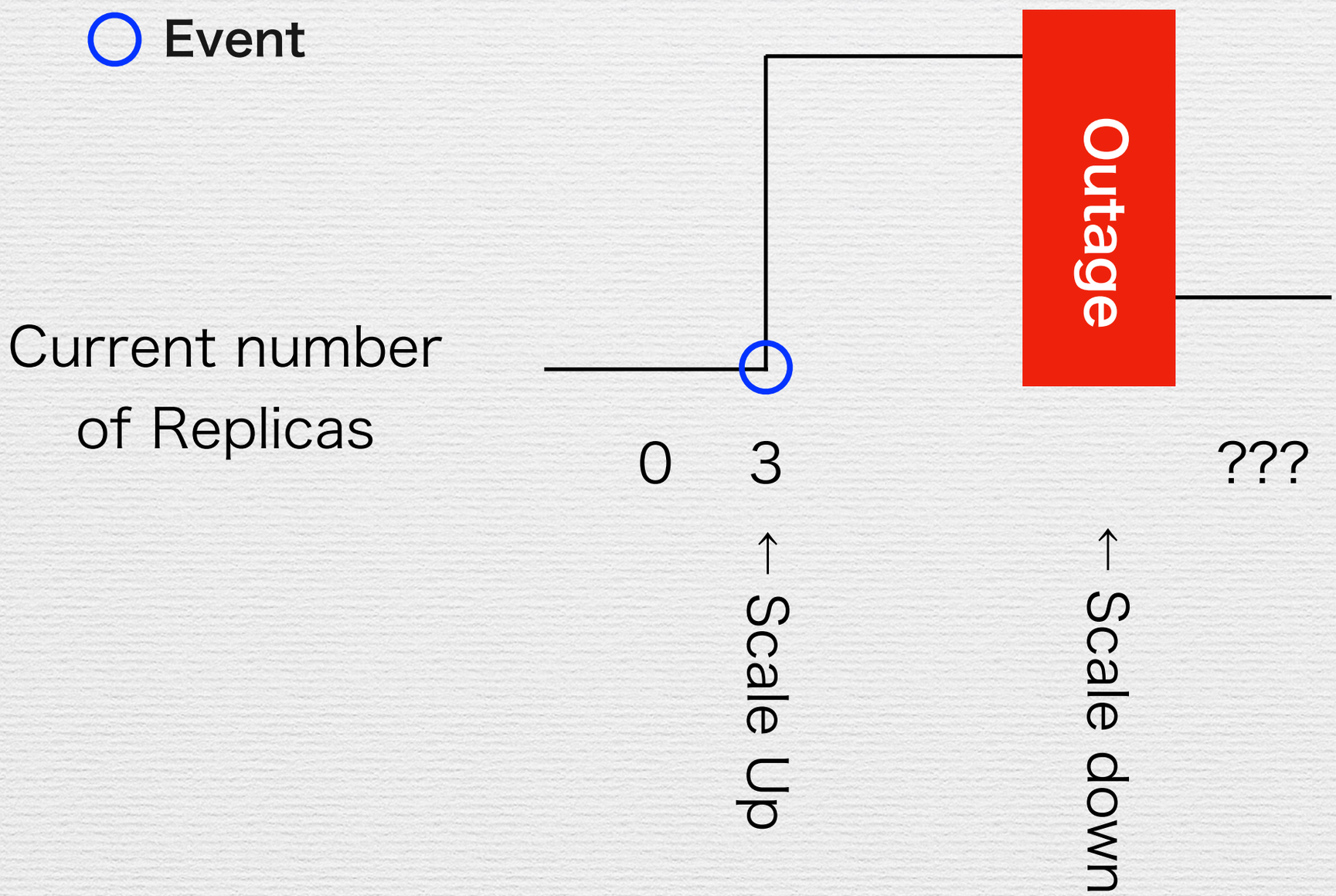
Controller takes Procedure process, the events are triggered as Edge-driven-trigger.



This seems like no problem, but there are weaknesses.

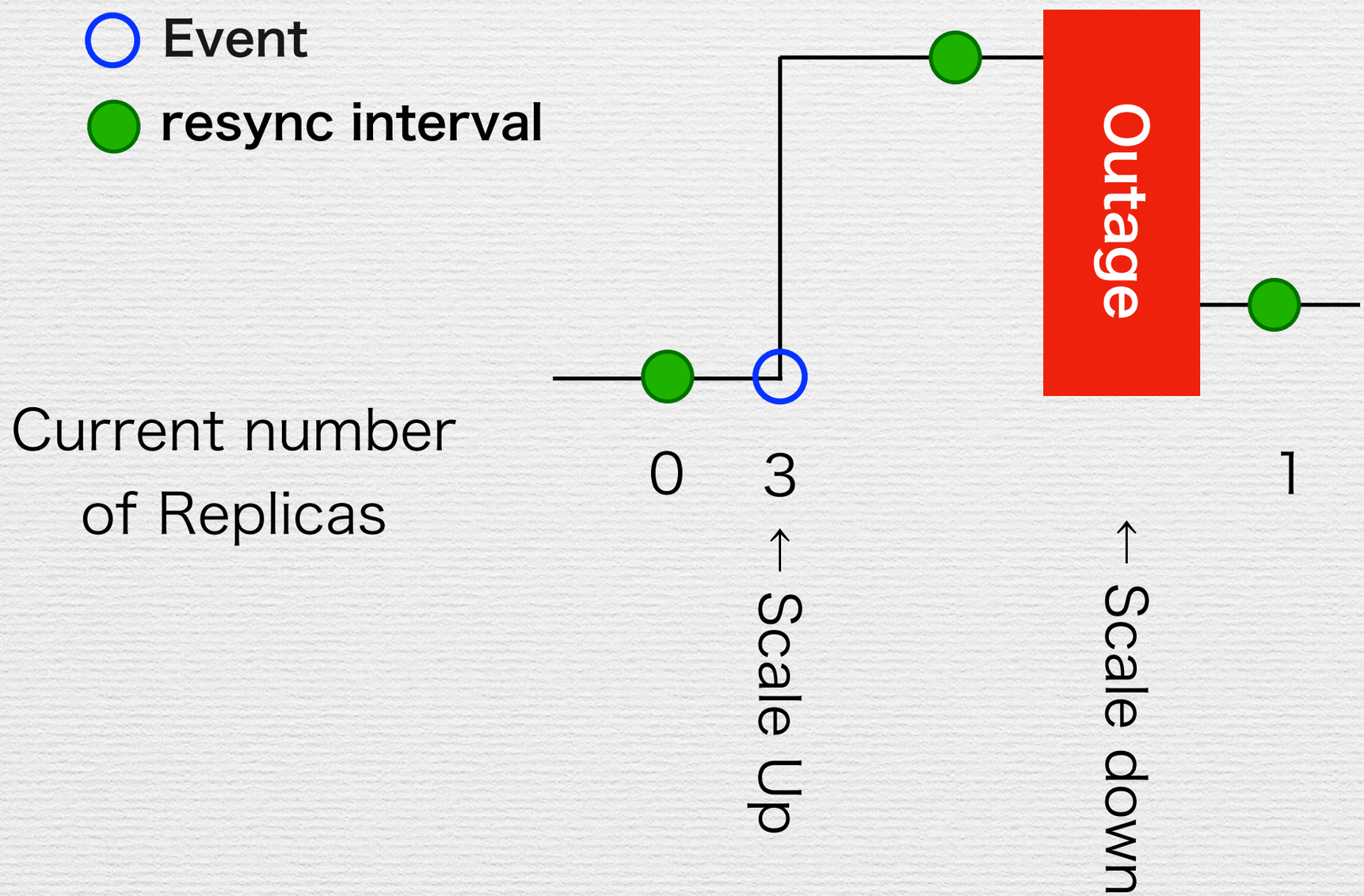
If there is no Reconcile

When there is temporary network outage or bug,
Event information is lost.



Resync Interval and Reconcile

Controller Reconcile every time Resync Interval.
So that Controller can bring the state closer to desired state.



Kubernetes = <u>Edge-driven Trigger + Level-driven Trigger</u>

Components which support Controller

~ Middle Level Architecture ~

Terminology

Kind:

Kind is the kind of API Object(e.g. Deployment, Service)

Resource:

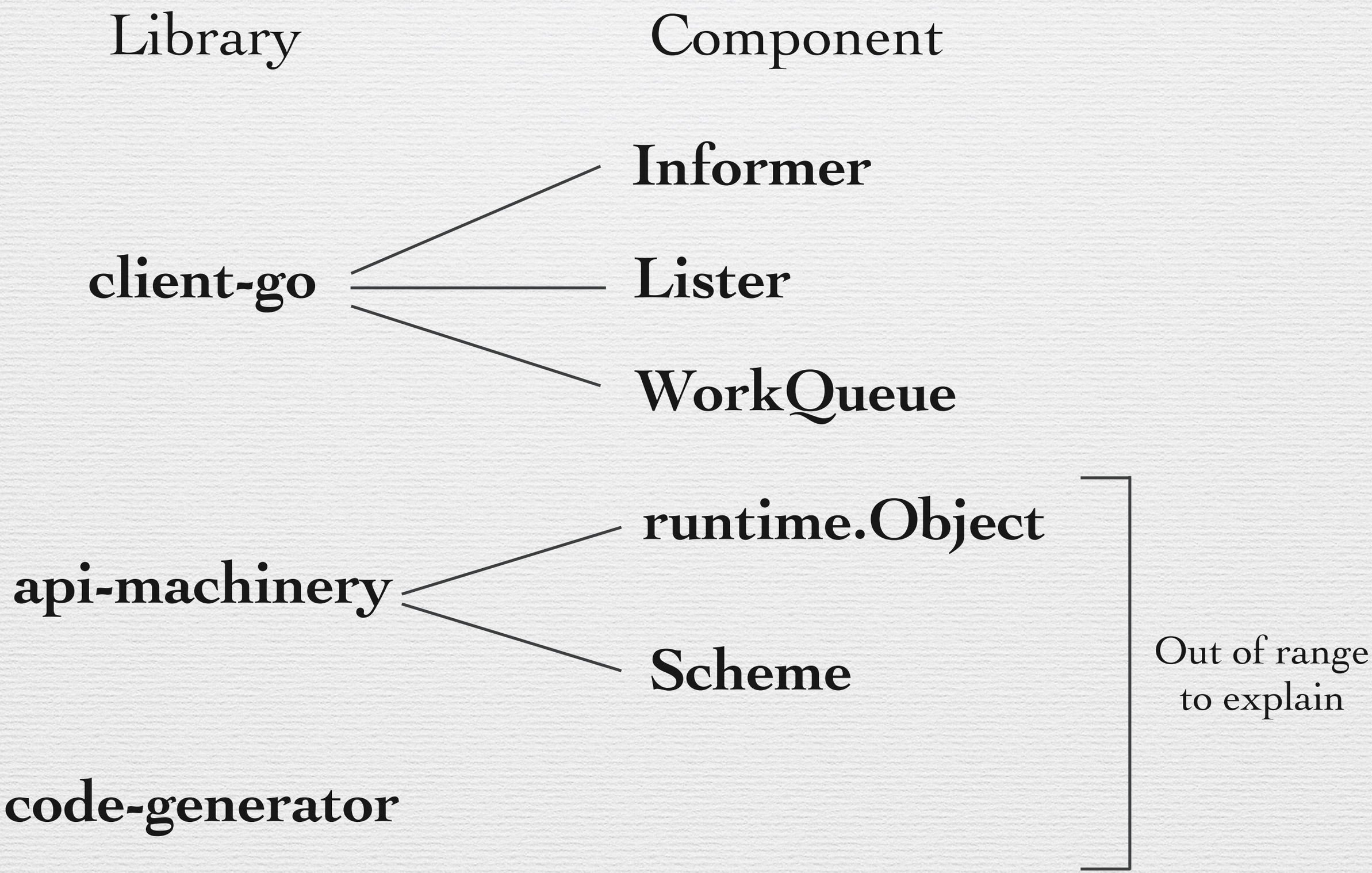
Resource is used in the same meaning as Kind.
This is used as HTTP Endpoint.
Resource is expressed in lower case and plural form
(e.g. pods, services)

Object:

An entity of created API Object.

This is persisted in etcd.

Library under the Controller



Appendix) Custom Controller SDK

Framework	**Kubebuilder**	**Operator SDK**
Library (High Level)	**controller-runtime**	**controller-tools**
Library (Low Level)	**client-go**	**api-machinery** **etc…**
Component	**Informer** **Lister** **WorkQueue**	**Scheme** **runtime.Object** **etc…**

Library under the Controller

client-go:

Kubernetes official client Library

This is used to Kubernetes development

api-machinery:

Kubernetes API Object & Kubernetes API like Object Library

e.g. conversion, decode, encode, etc…

Controller manages API Object, so this is needed.

code-generator:

Informer, Lister, clientset, DeepCopy source code generator

This is used to Custom Controller development mainly.

Component under Controller

Detail of each component will be described later.

Informer:

Watch an Object Event and stores data to in-memory-cache

Lister:

Getter object data from in-memory-cache

WorkQueue:

Queue which store Control Loop item

runtime.Object:

API Object Interface

Scheme:

Associate Go Type with Kubernetes API

Out of range to explain

client-go Informer

~ Low Level Architecture ~

client-go と Informer

Library **client-go**

Component **Informer**

Reflector DeltaFIFO Indexer

Store Lister

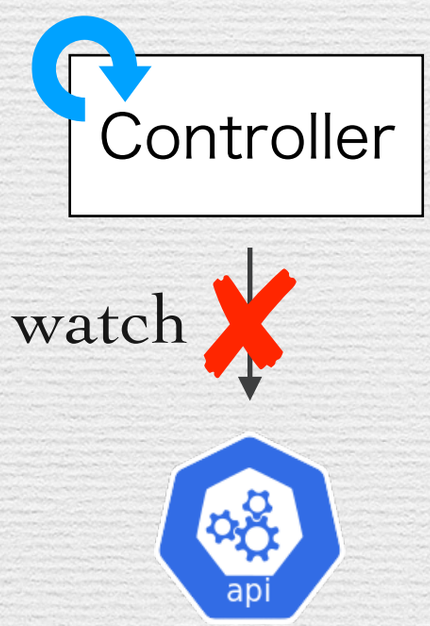
Informer

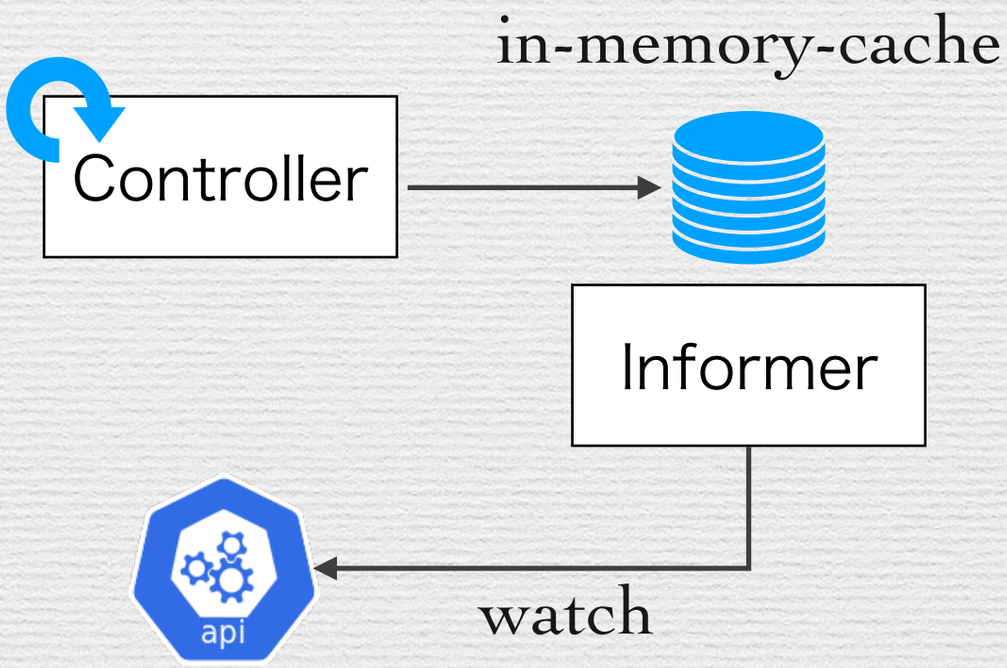
Informer watches Object Event(Added, Updated, Deleted…)

When controller inquiries object status to api-server every time to monitor Object changes, api-server is high loaded.

⇒ Informer stores object data to in-memory-cache.

By Controller referring to cache, this problem is solved.





Appendix) Informer Sample Code

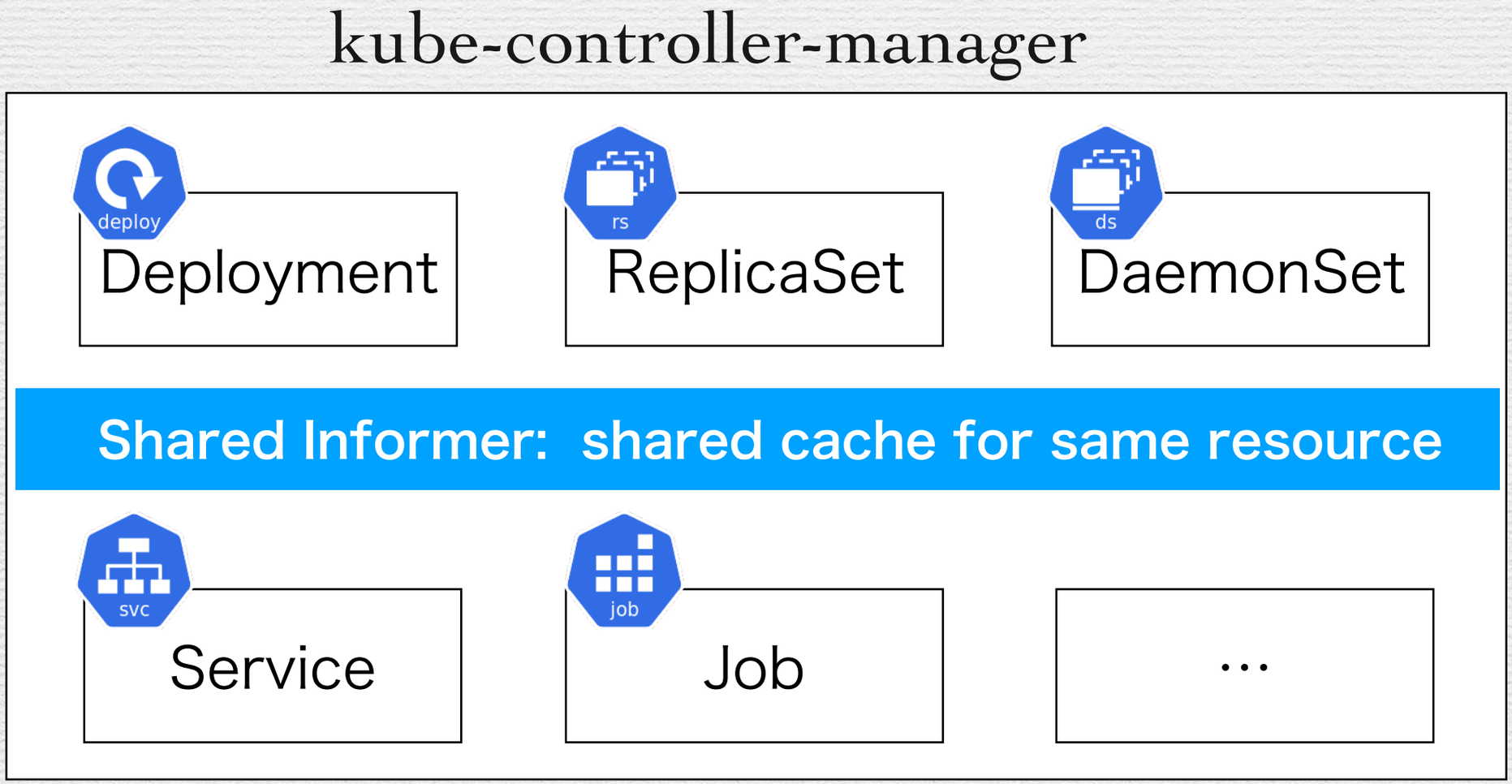
```
func main() {
    ...
    clientset, err := kubernetes.NewForConfig(config)
    // Create InformerFactory
    informerFactory := informers.NewSharedInformerFactory(clientset, time.Second*30)

    // Create pod informer by informerFactory
    podInformer := informerFactory.Core().V1().Pods()

    // Add EventHandler to informer
    podInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
      AddFunc: func(new interface{}) {   log.Println("Added")   },
      UpdateFunc: func(old, new interface{}) {   log.Println("Updated")   },
      DeleteFunc: func(old interface{}) {   log.Println("Deleted")   },
    })

    // Start Go routines
    informerFactory.Start(wait.NeverStop)
    // Wait until finish caching with List API
    informerFactory.WaitForCacheSync(wait.NeverStop)

    // Create Pod Lister
    podLister := podInformer.Lister()
    // Get List of pods
    _, err = podLister.List(labels.Nothing())
    …
}
```

https://github.com/govargo/kubecontorller-book-sample-snippet/blob/master/02/podinformer/podinformer.go

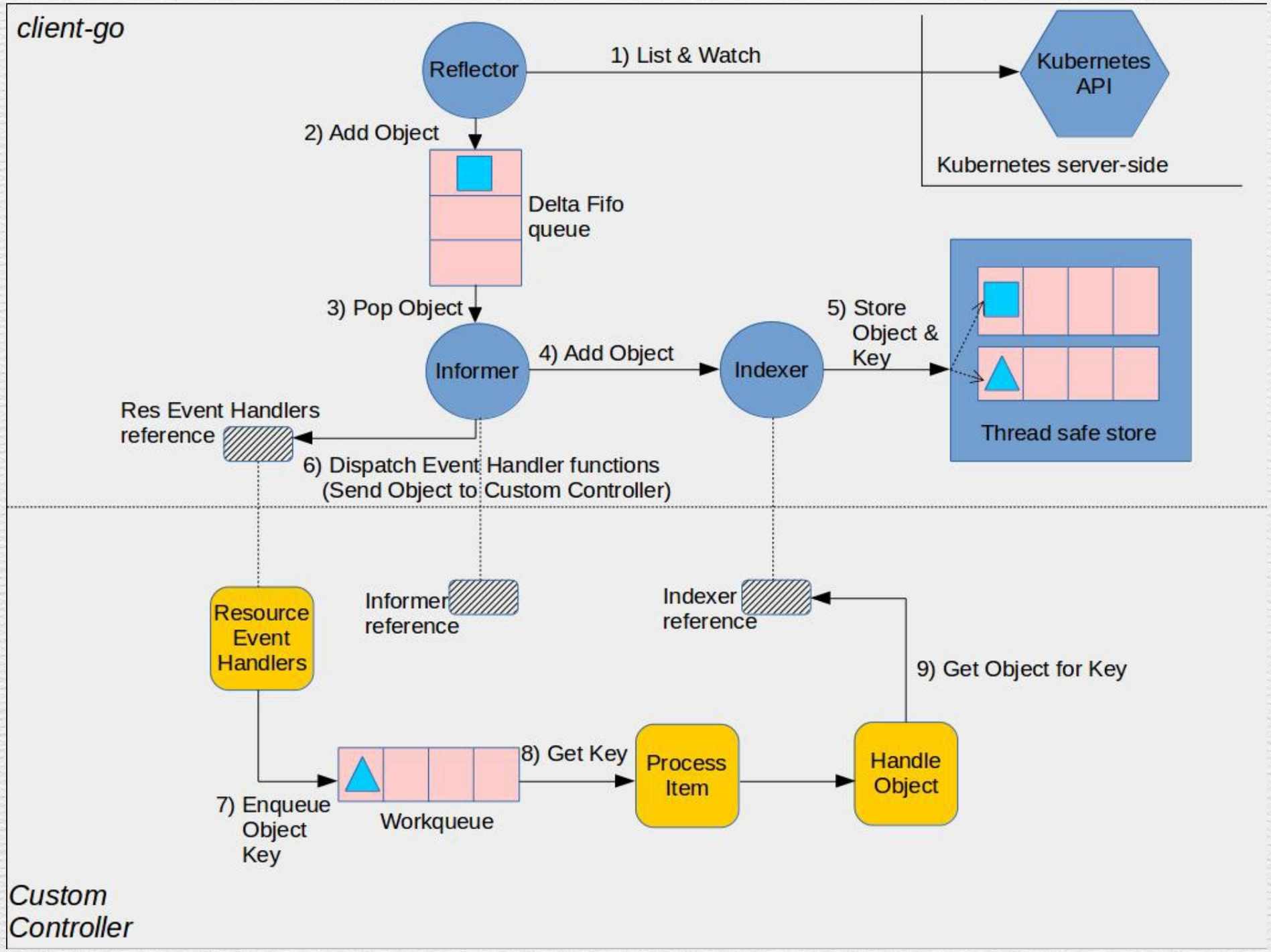
Appendix) Shared Informer

When we use Informer, we don't use Informer itself.
Instead we use Shared Informer.

Shared Informer shares same Resource in single binary.



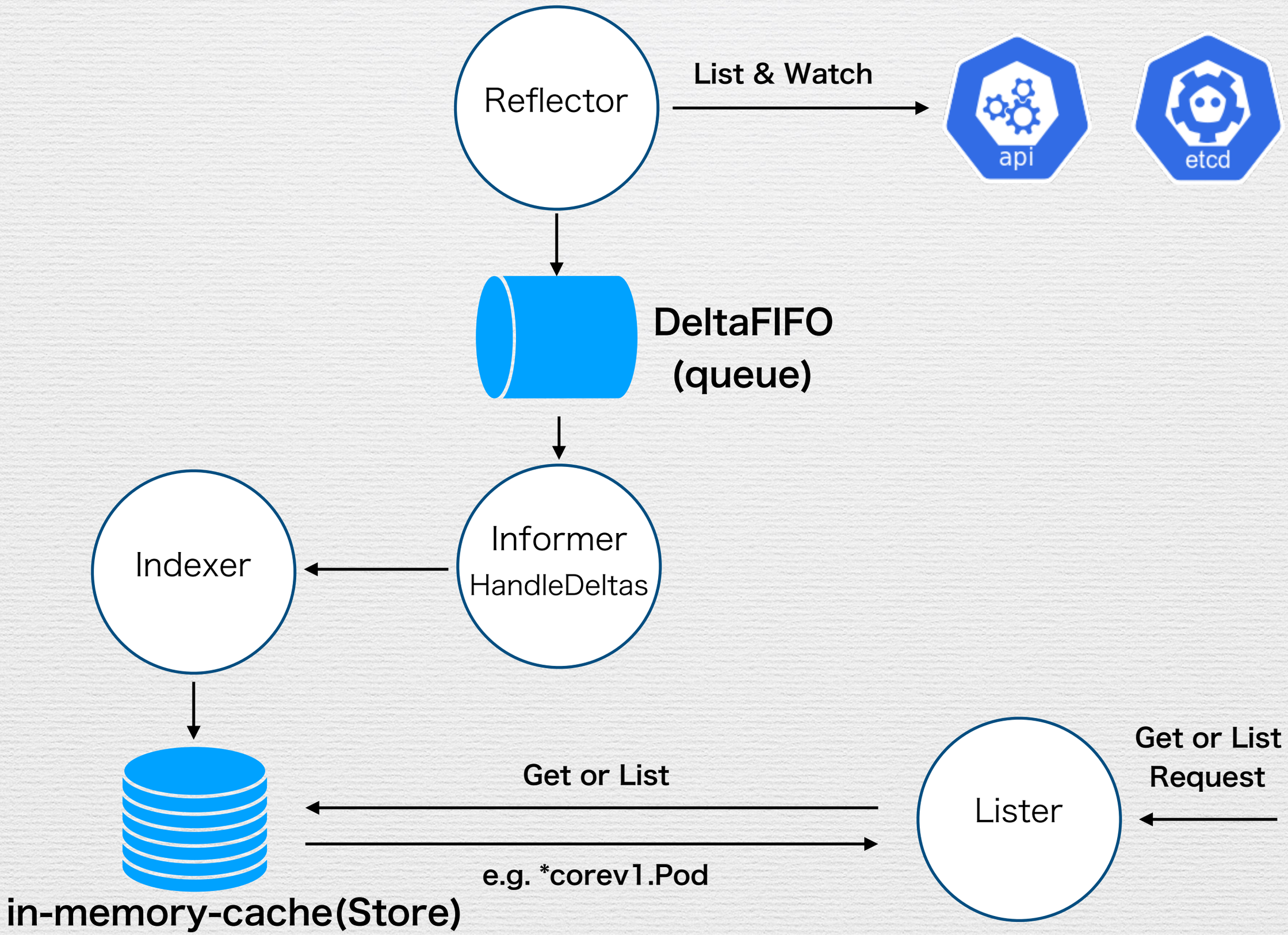
Informer and WorkQueue Overview



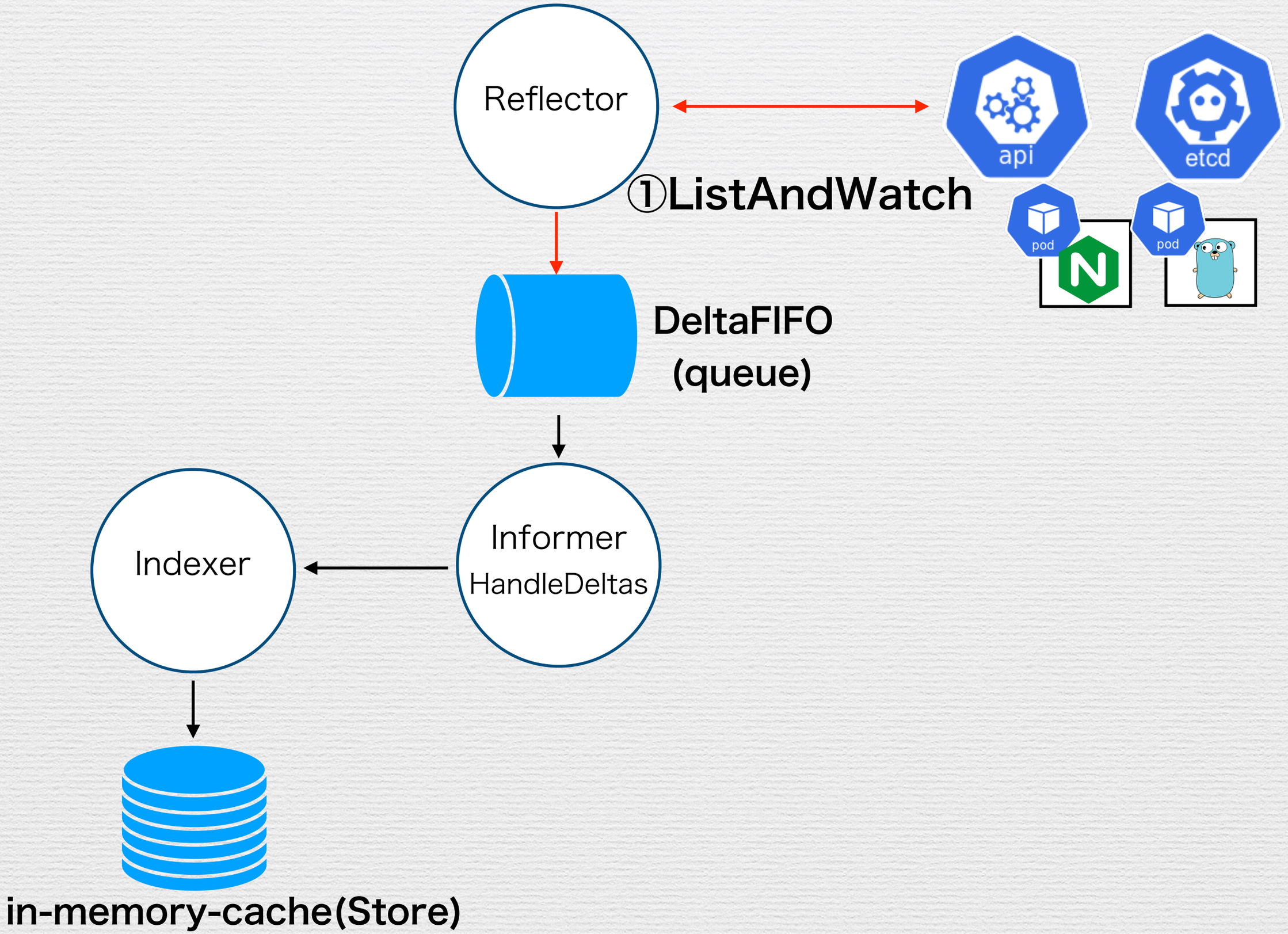
Reference:

https://github.com/kubernetes/sample-controller/blob/master/docs/controller-client-go.md

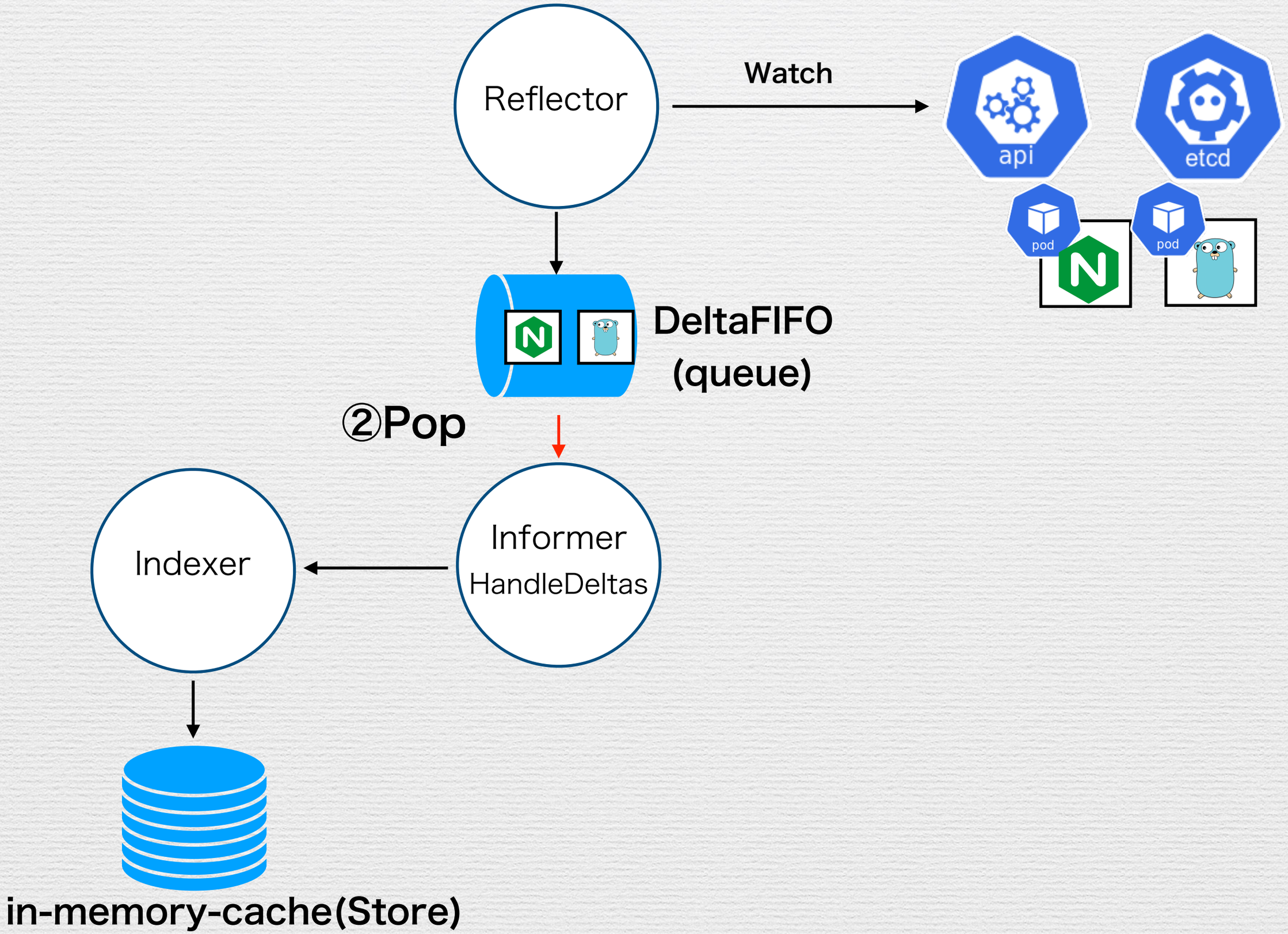
Detail of Informer



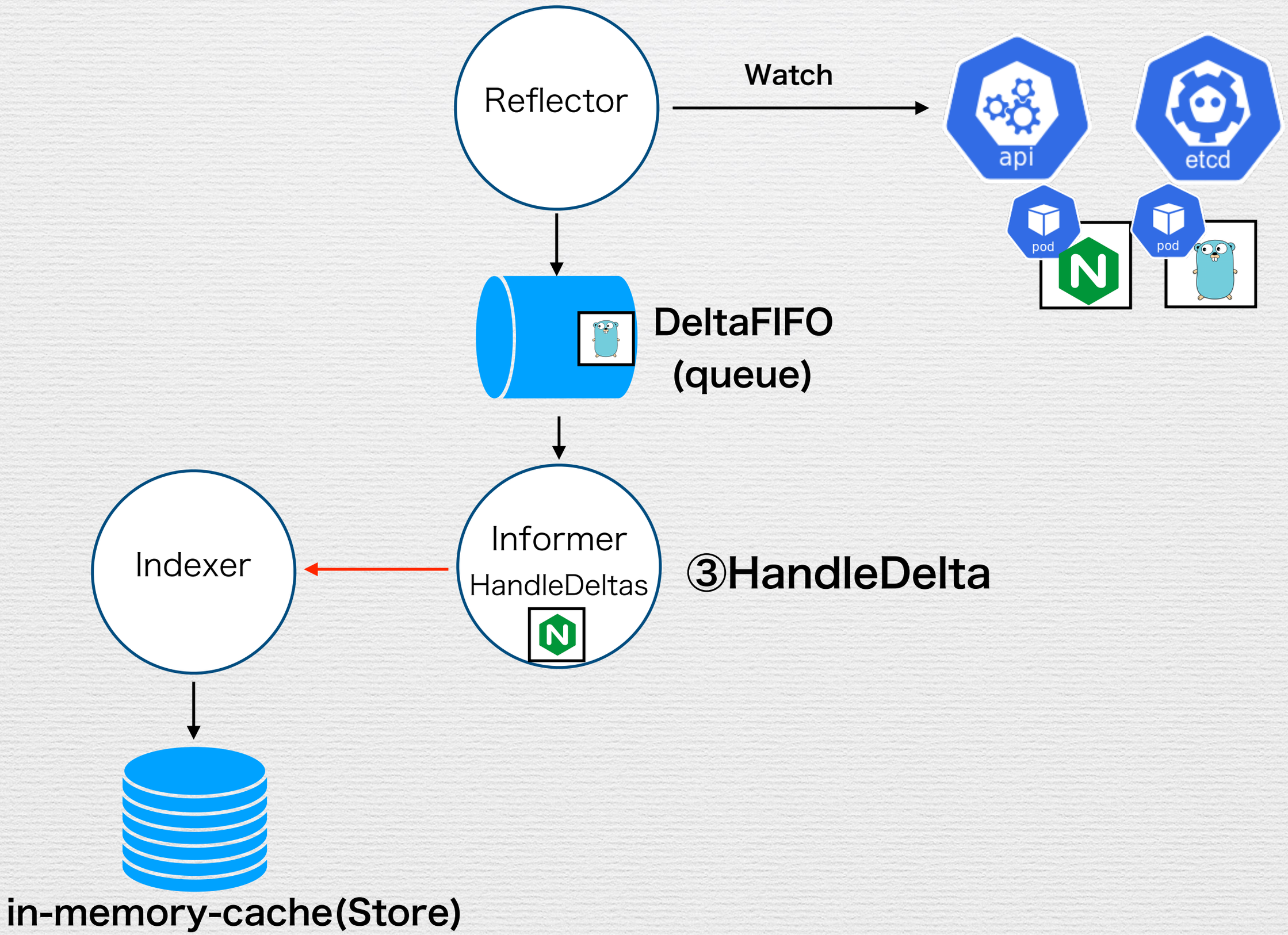
Detail of Informer ~Cache Flow~



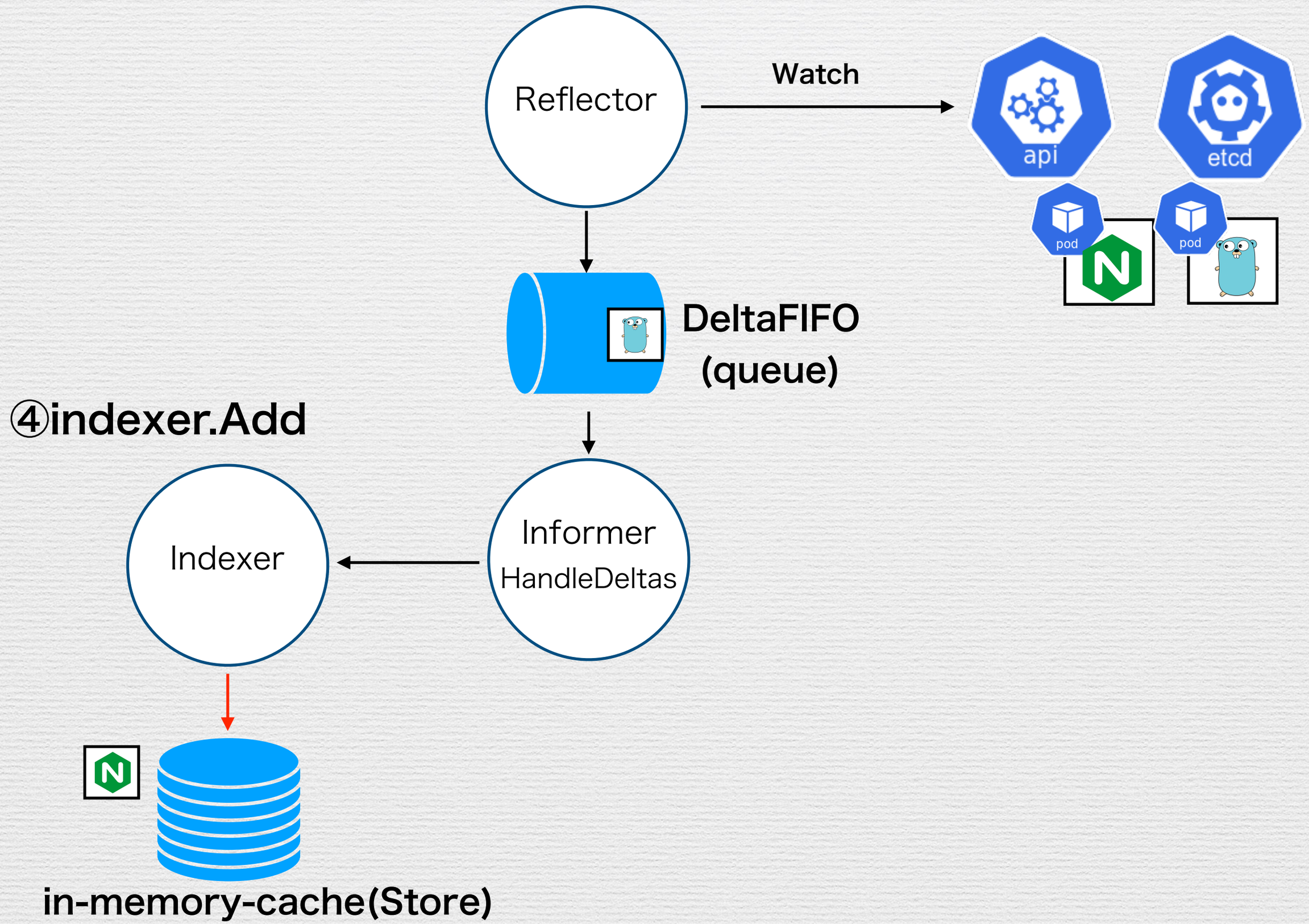
Detail of Informer ~Cache Flow~



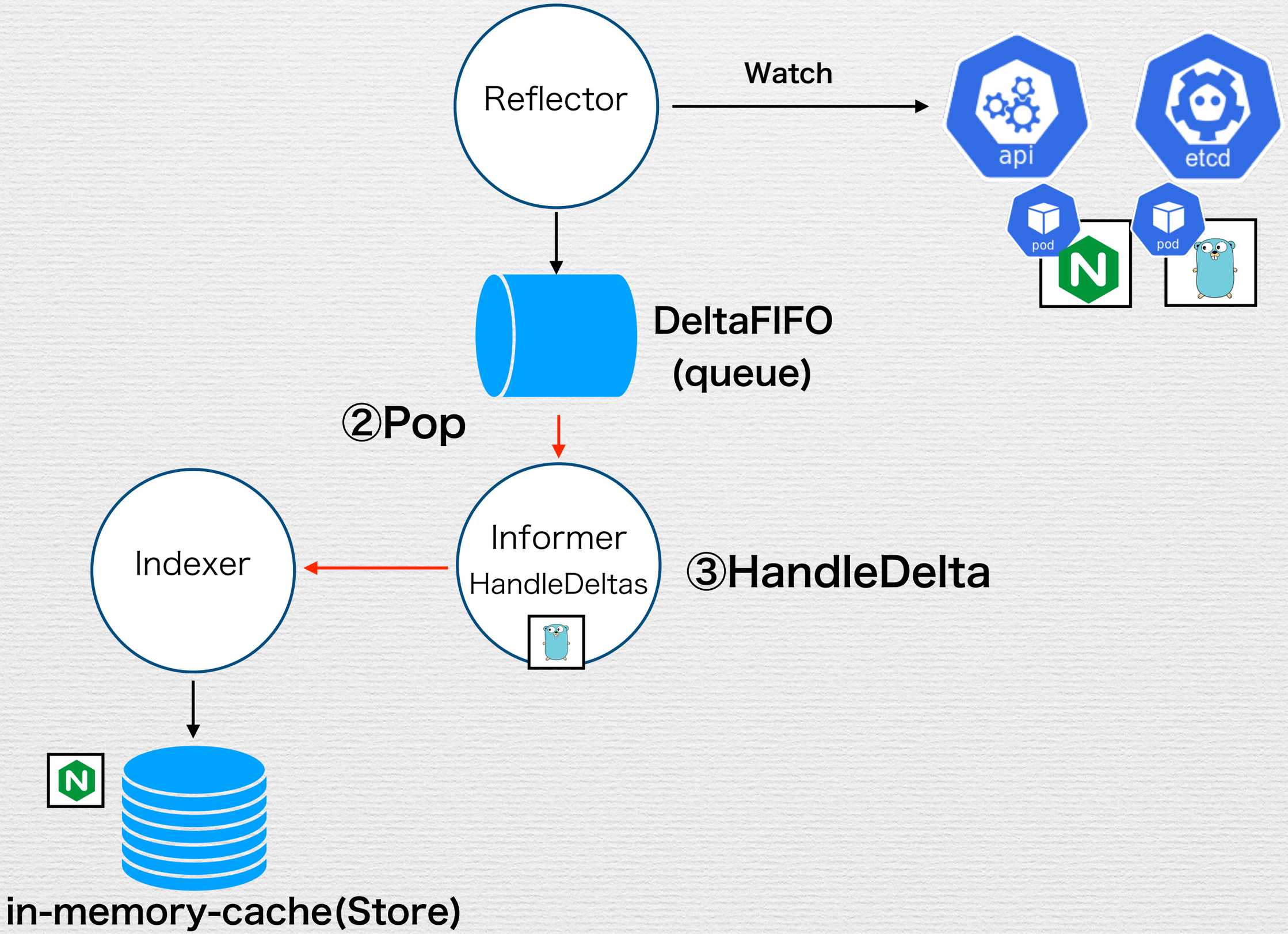
Detail of Informer ~Cache Flow~



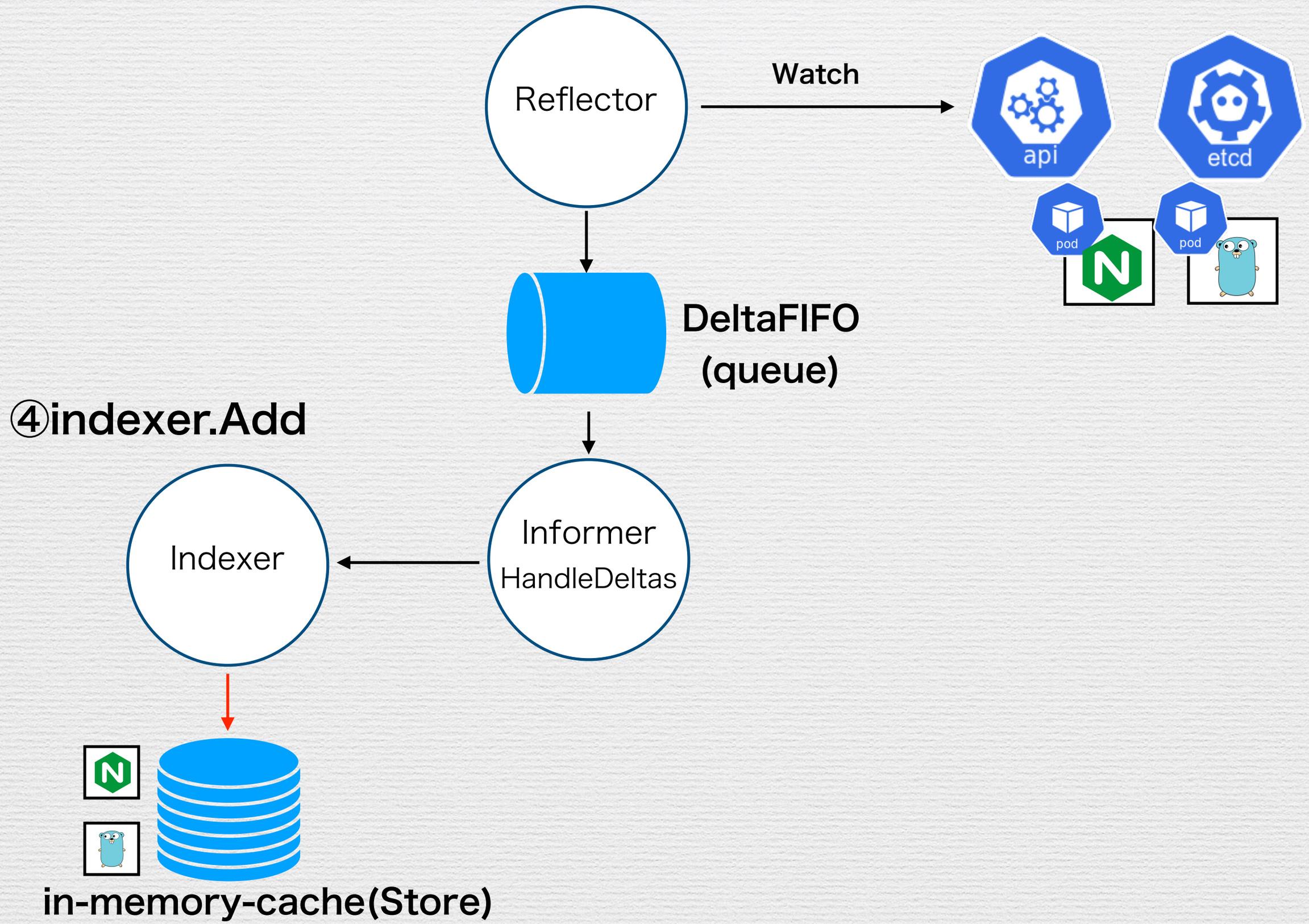
Detail of Informer ~Cache Flow~



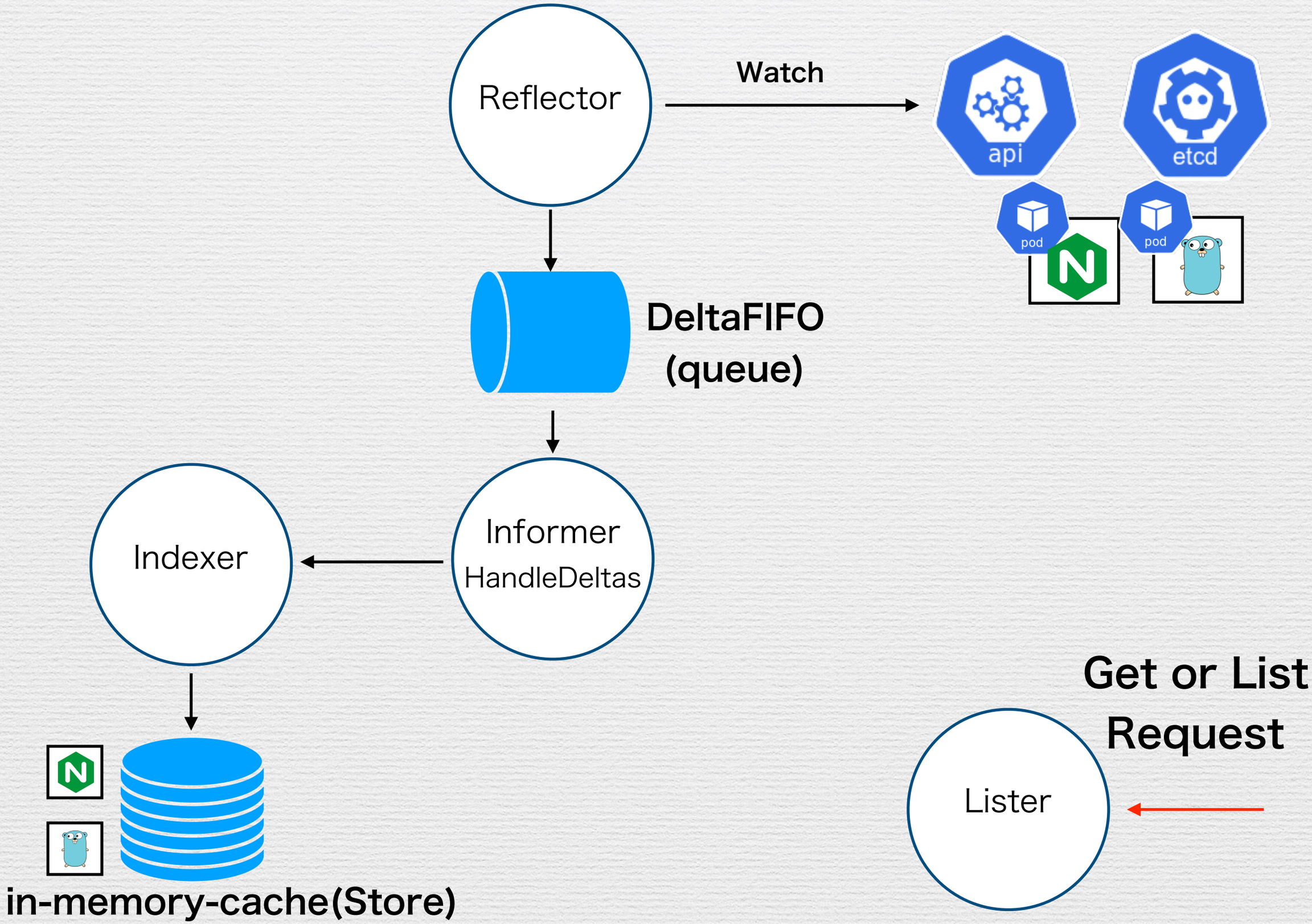
Detail of Informer ~Cache Flow~



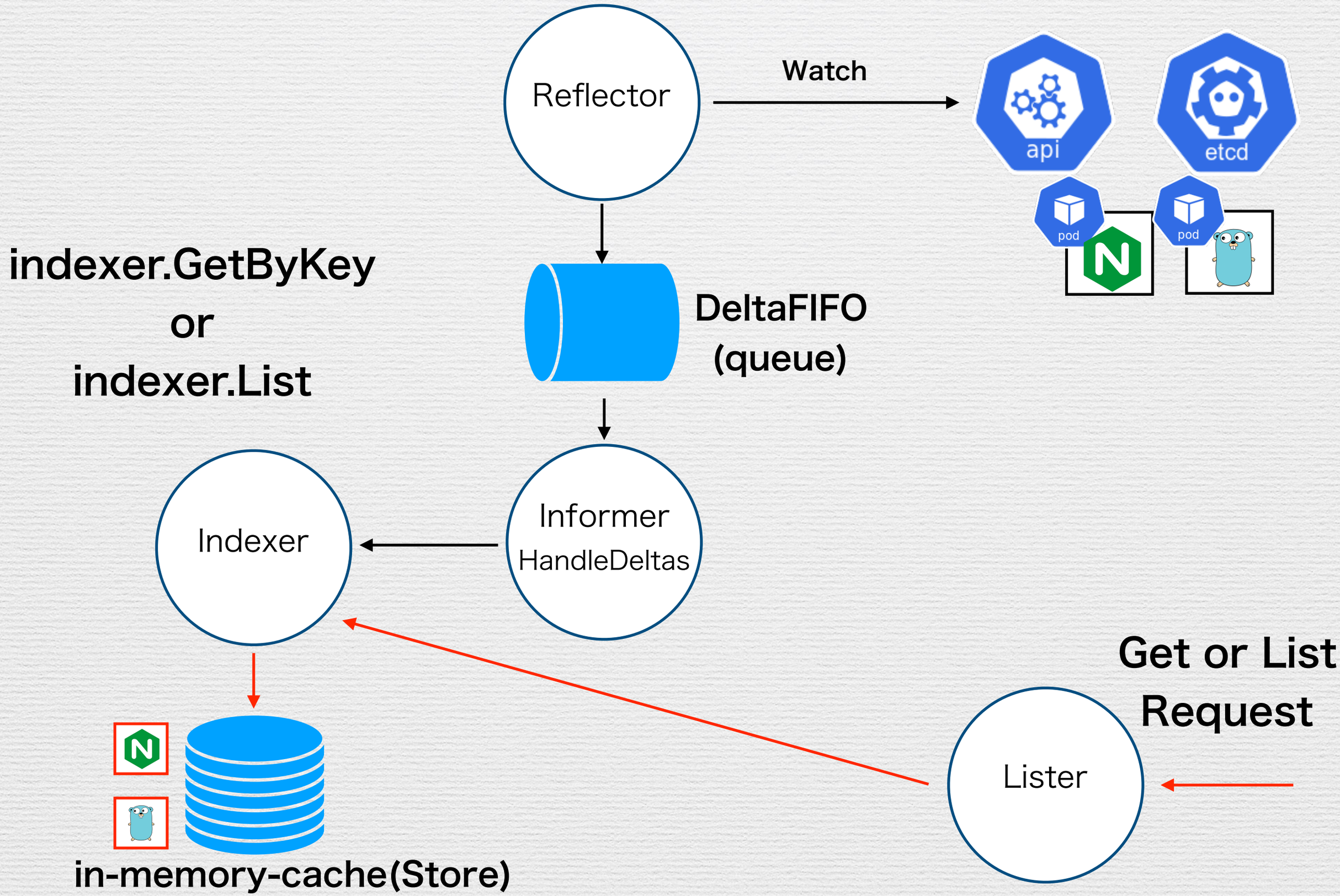
Detail of Informer ~Cache Flow~



Detail of Informer ~Cache Flow~



Detail of Informer ~Cache Flow~



Appendix) Informer cache Source Code

①

https://github.com/kubernetes/client-go/blob/release-13.0/tools/cache/reflector.go#L188

②

https://github.com/kubernetes/client-go/blob/release-13.0/tools/cache/controller.go#L153

③

https://github.com/kubernetes/client-go/blob/release-13.0/tools/cache/shared_informer.go#L455

④

https://github.com/kubernetes/client-go/blob/release-13.0/tools/cache/shared_informer.go#L464

Informer and Component

Informer:

Watch an Object Event and stores data to in-memory-cache

Reflector:

ListAndWatch api-server

DeltaFIFO:

FIFO Queue which enqueue object data temporarily

Indexer:

Getter / Setter for in-memory-cache

Store:

in-memory-cache

Lister:

Getter object data from in-memory-cache via Indexer

client-go WorkQueue

~ Low Level Architecture ~

client-go と WorkQueue

Library

client-go

Component

WorkQueue

RateLimitingQueue

DelayedQueue

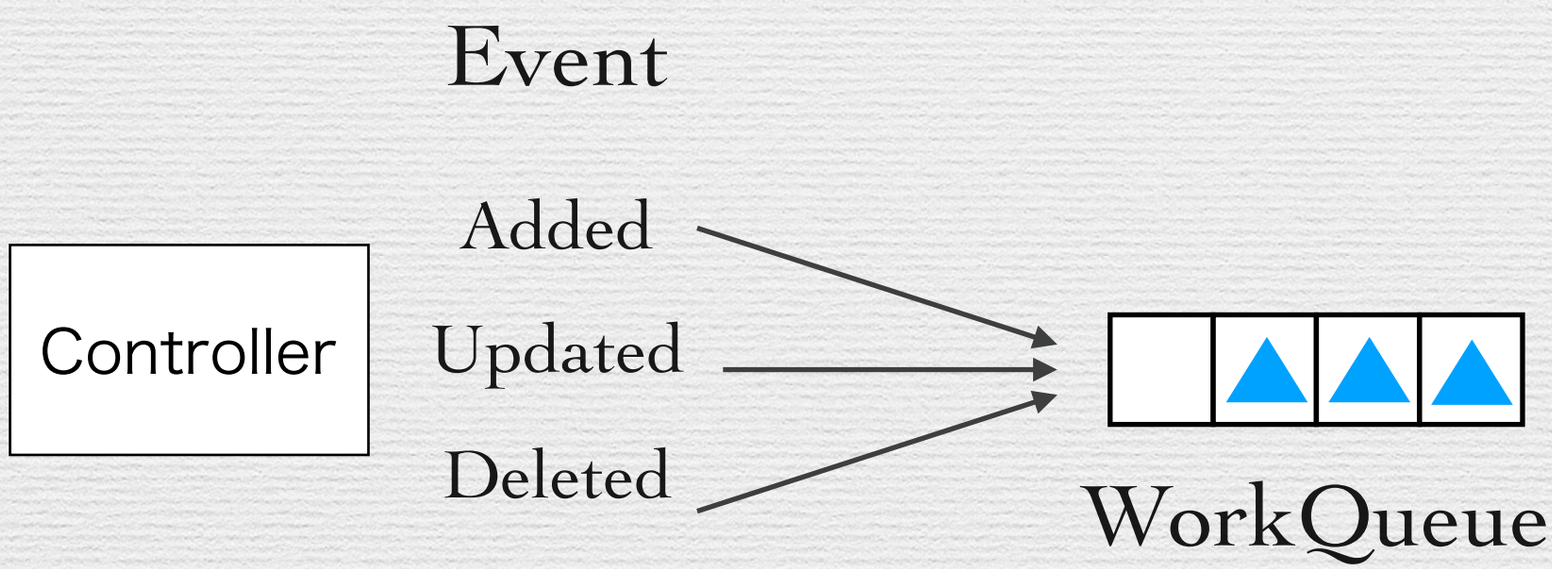
…

WorkQueue

WorkQueue is another queue different from DeltaFIFO.

WorkQueue is used in order to store item of Contrl Loop.
Reconcile will be executed as many times as the number stored in WorkQueue.

Pure Controller enqueues item to WorkQueue when Event occurs.

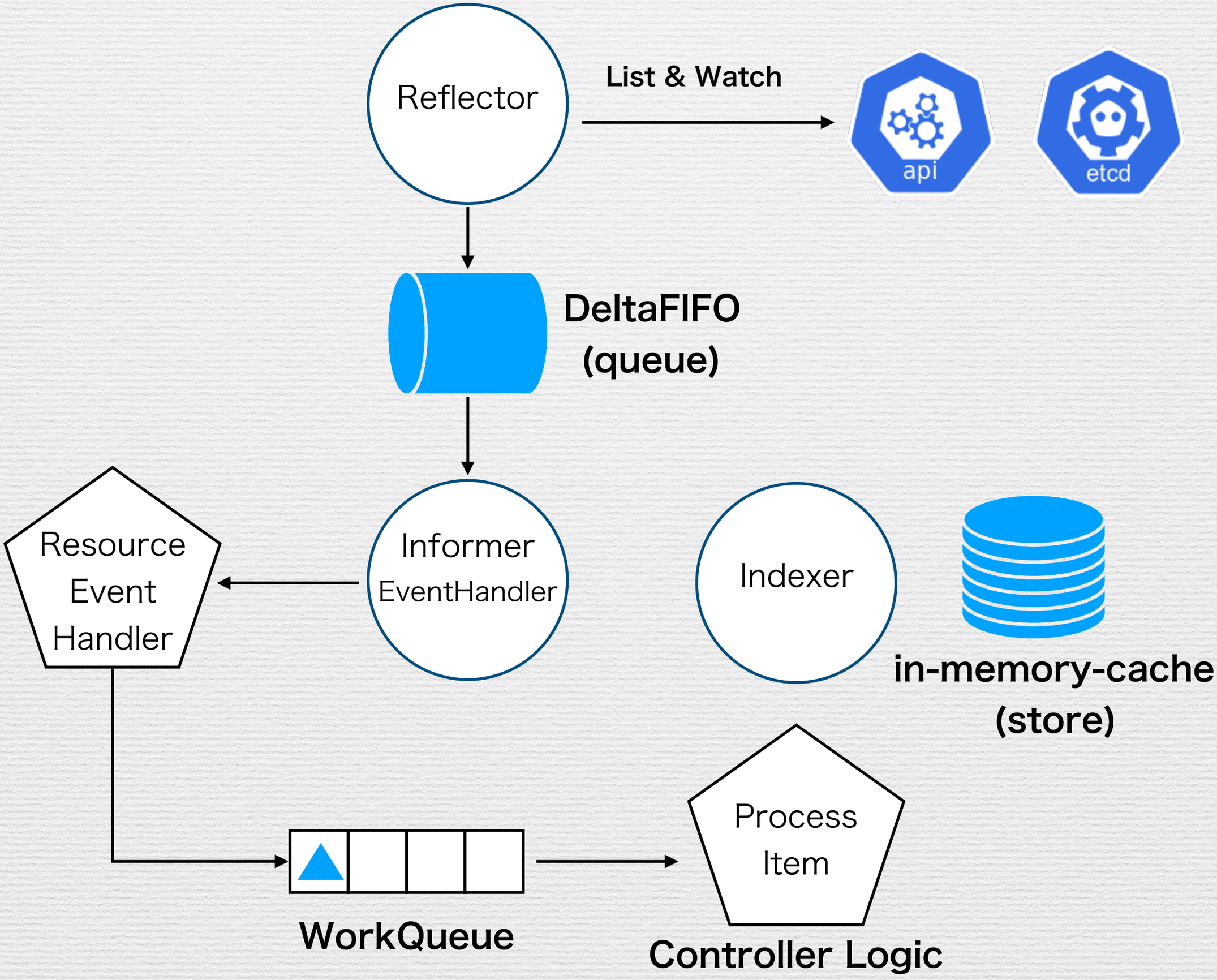


Appendix) WorkQueue Sample Code

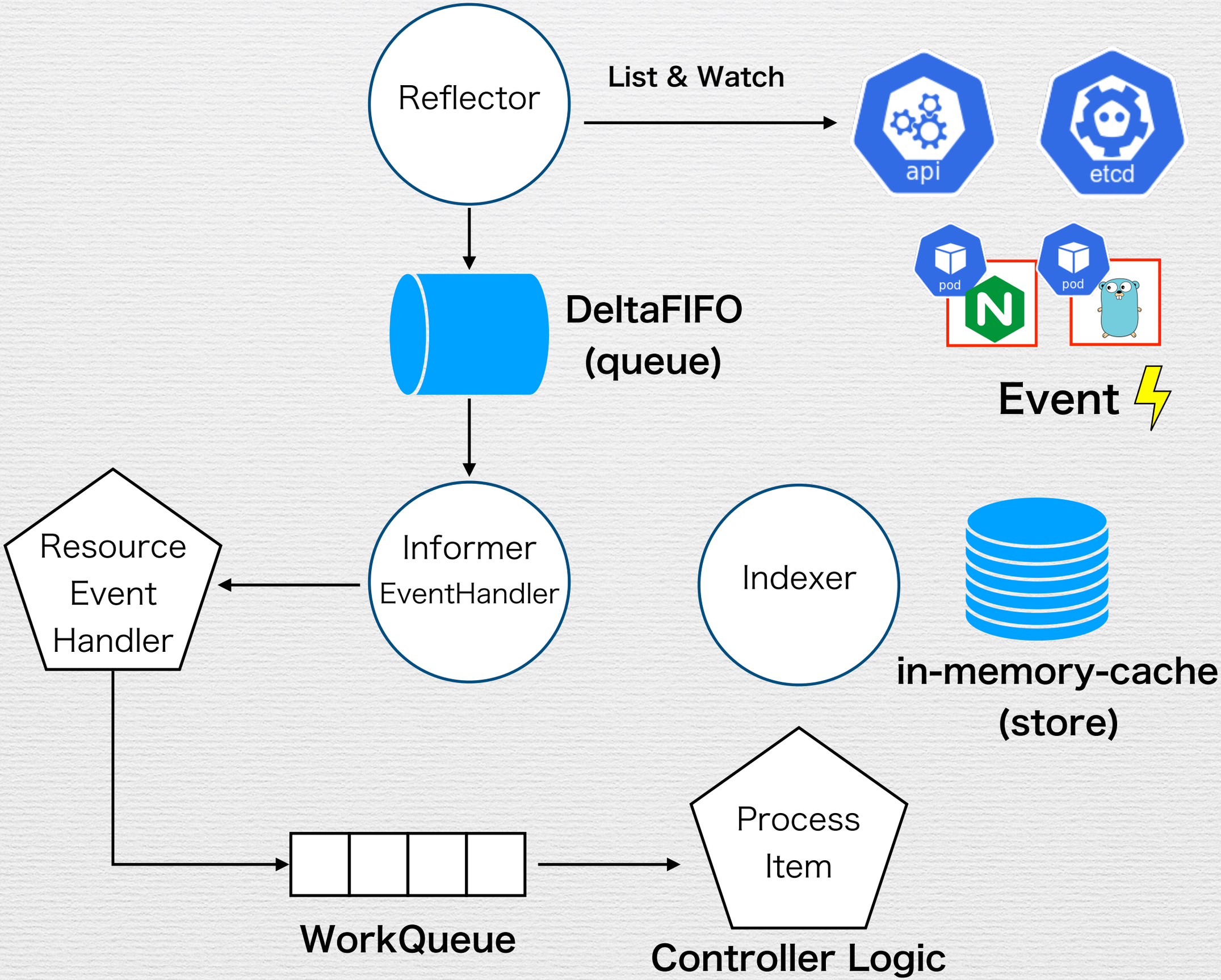
```
func main() {
    ...
    clientset, err := kubernetes.NewForConfig(config)
    // Create InformerFactory
    informerFactory := informers.NewSharedInformerFactory(clientset, time.Second*30)

    // Create pod informer by informerFactory
    podInformer := informerFactory.Core().V1().Pods()

    // Create RateLimitQueue
    queue := workqueue.NewRateLimitingQueue(workqueue.DefaultControllerRateLimiter())
   // shutdown when process ends
   defer queue.ShutDown()

    // Add EventHandler to informer
    podInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
      AddFunc: func(old interface{}) {
                var key string
                var err error
                if key, err = cache.MetaNamespaceKeyFunc(old); err != nil {
                      runtime.HandleError(err)
                      return
                }
                queue.Add(key)
                log.Println("Added: " + key)
          },
      UpdateFunc: func(old, new interface{}) { …  },
      DeleteFunc: func(old interface{}) {  …  },
    })
   …
}
```

https://github.com/govargo/kubecontorller-book-sample-snippet/blob/master/02/workqueue/enqueuePod.go

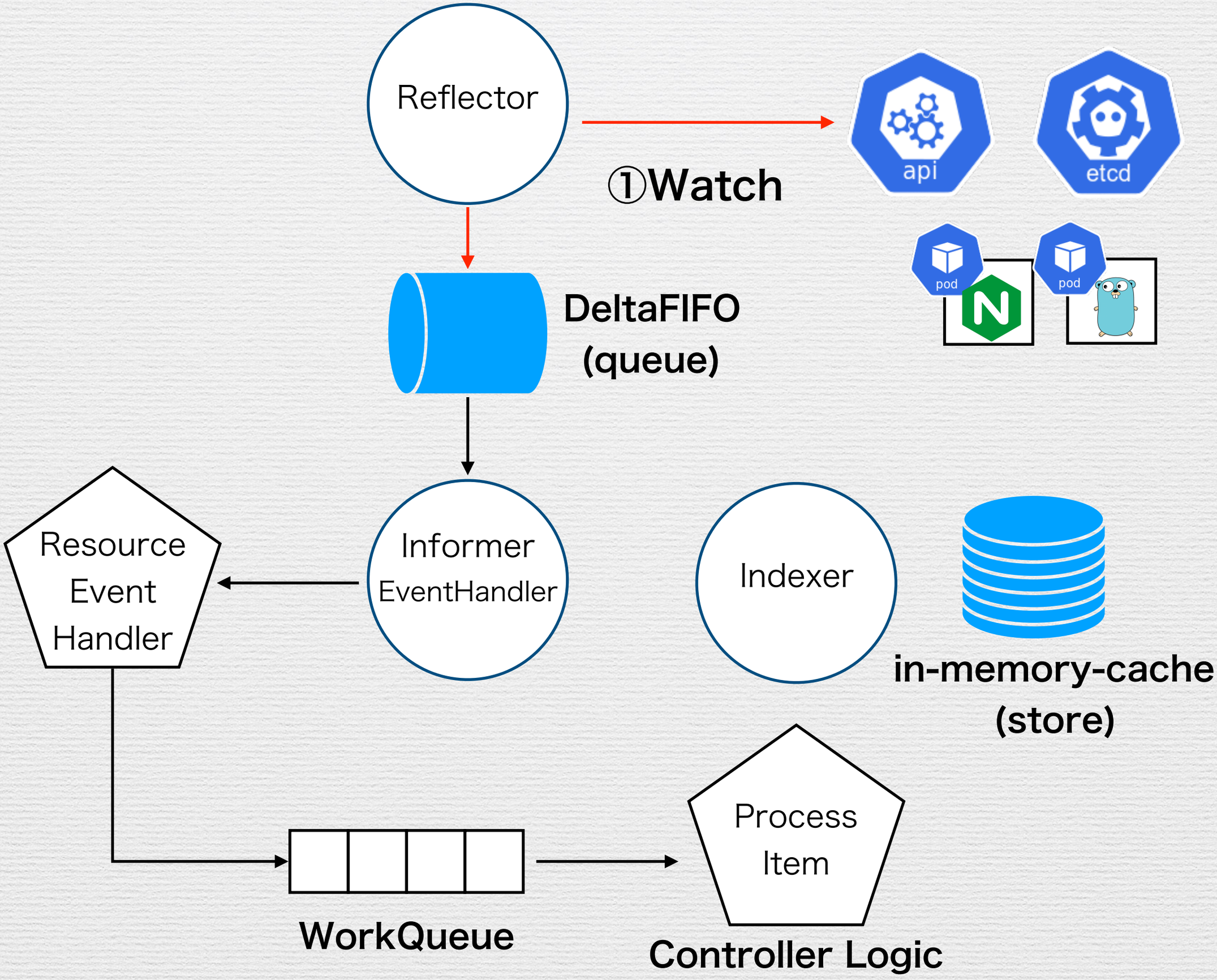
Detail of WorkQueue



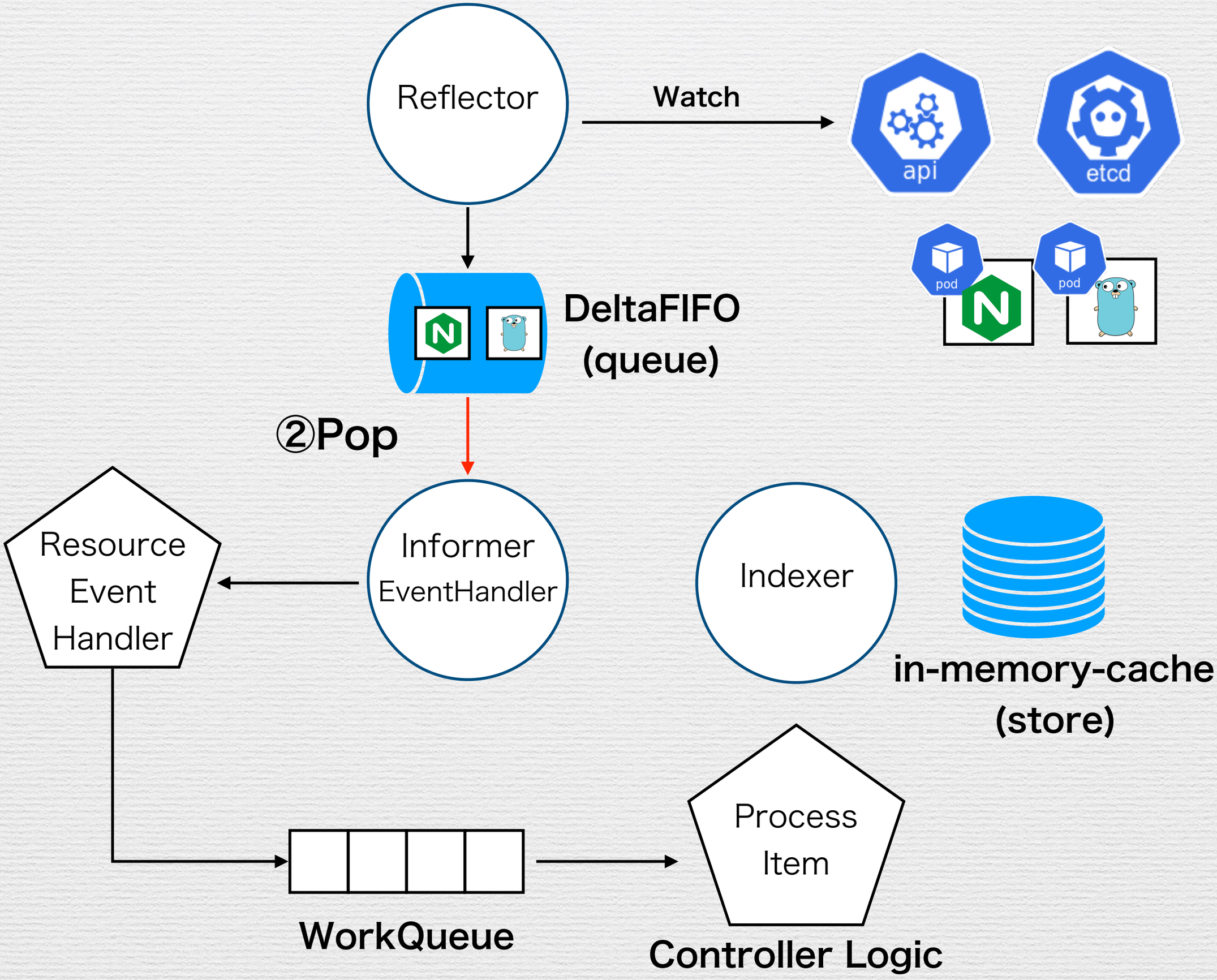
Detail of WorkQueue ~Enqueue~



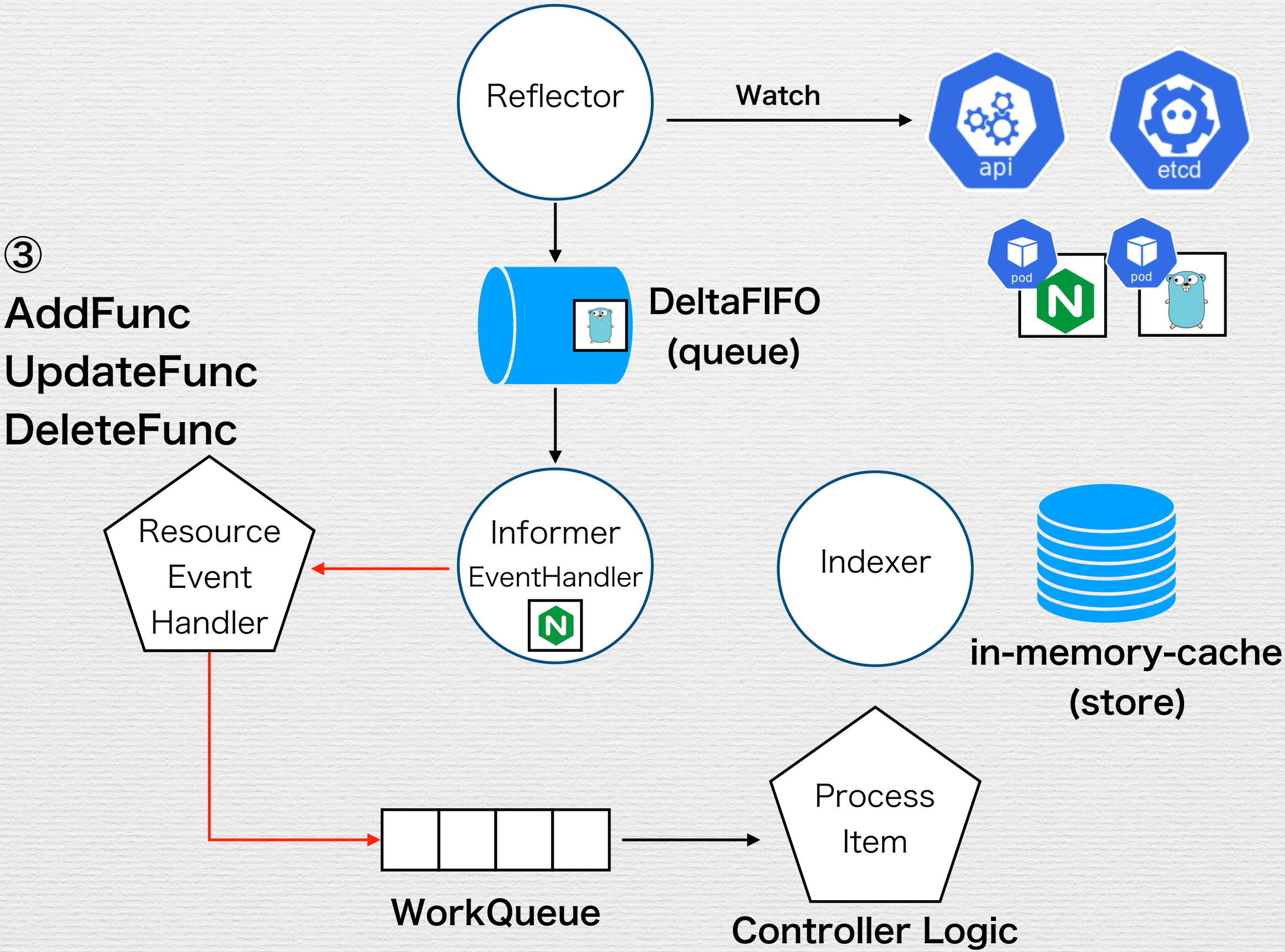
Detail of WorkQueue ~Enqueue~



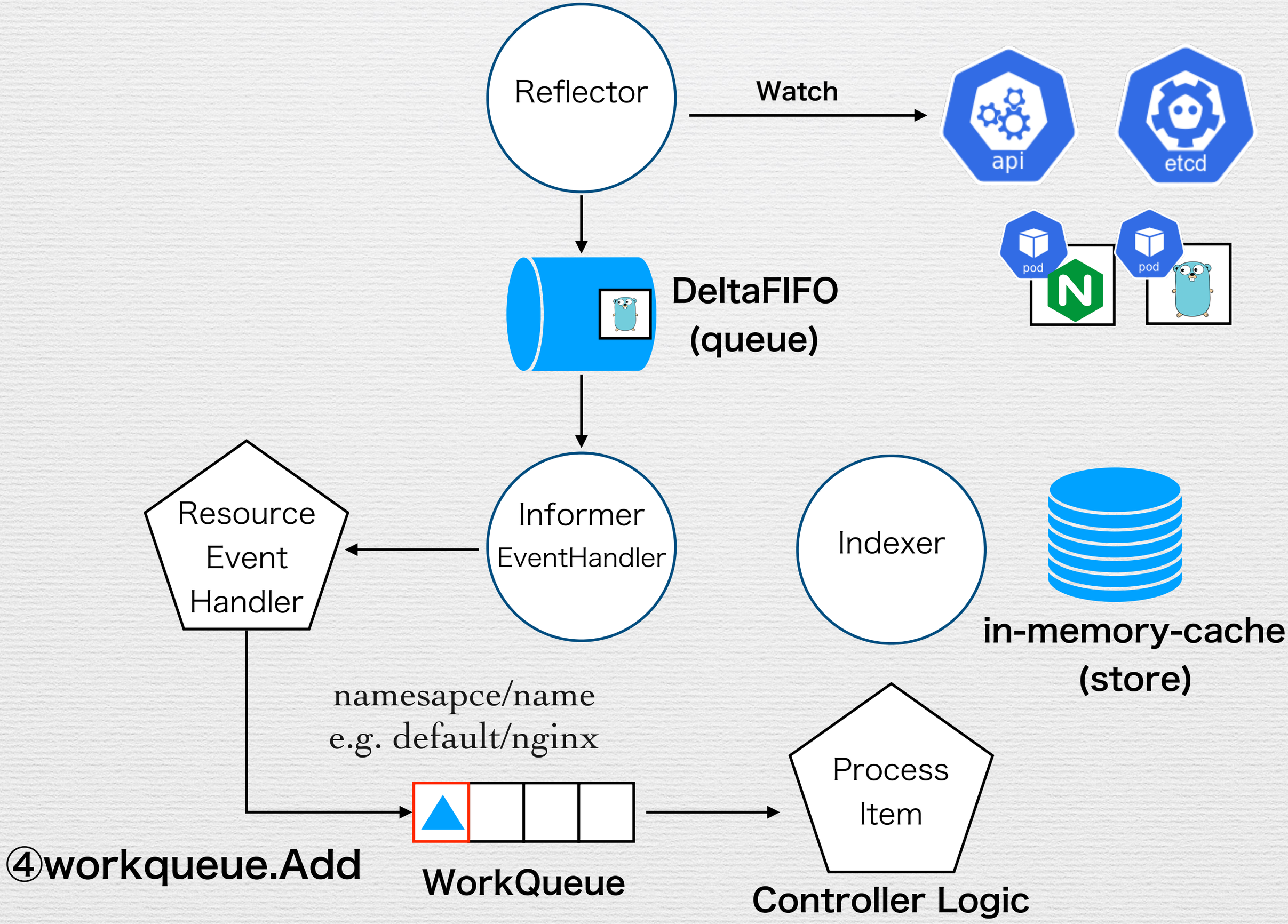
Detail of WorkQueue ~Enqueue~



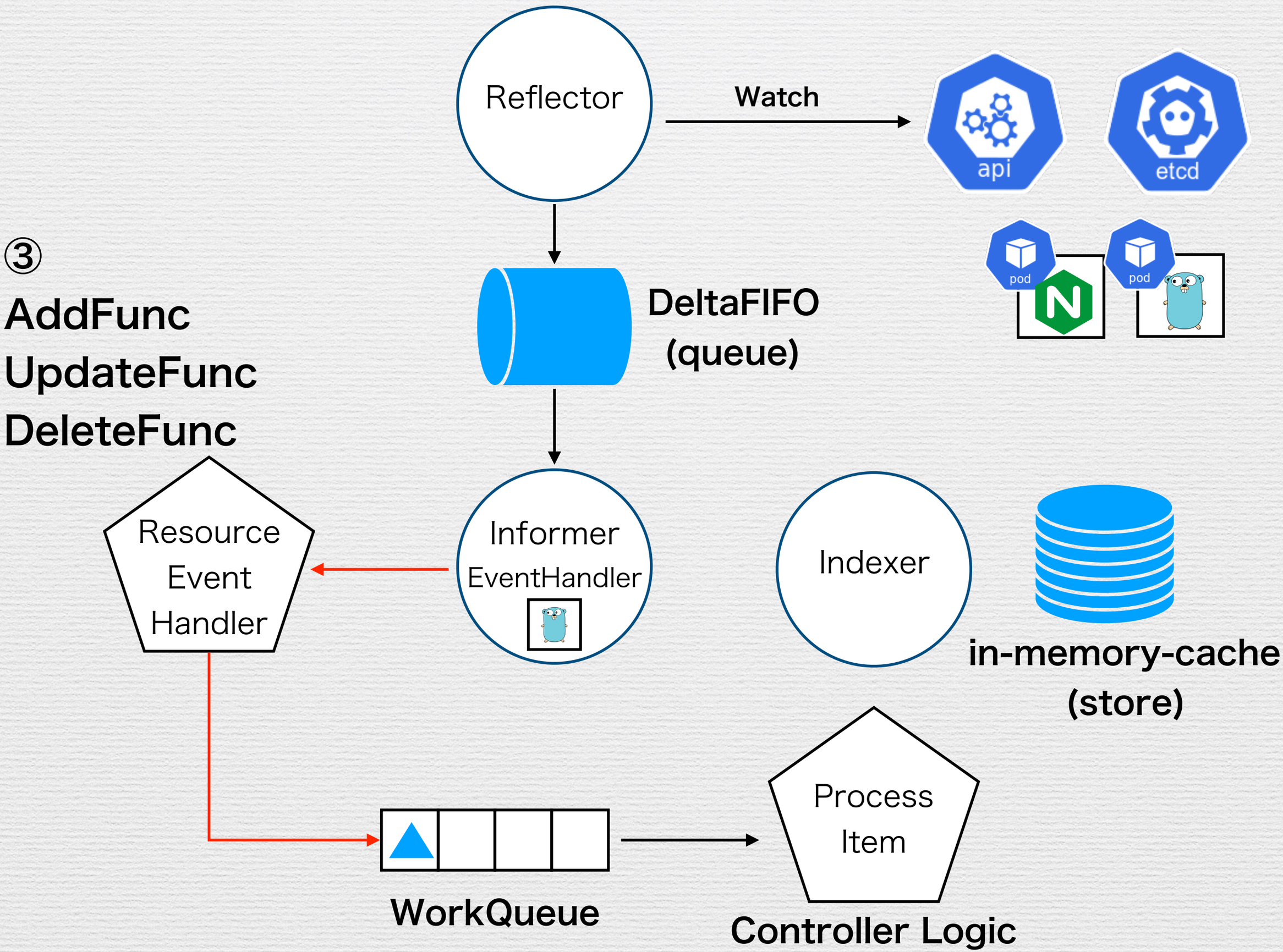
Detail of WorkQueue ~Enqueue~



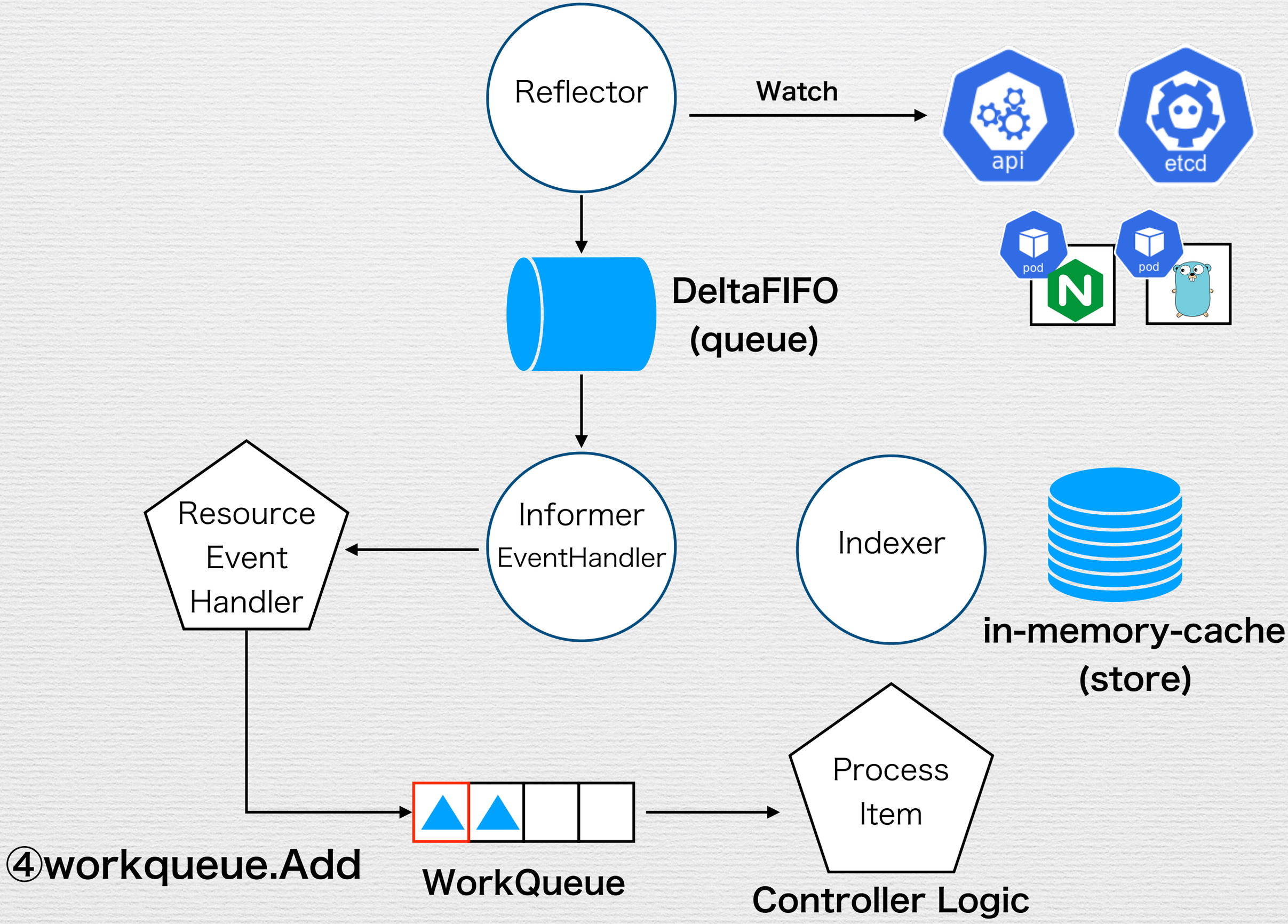
Detail of WorkQueue ~Enqueue~



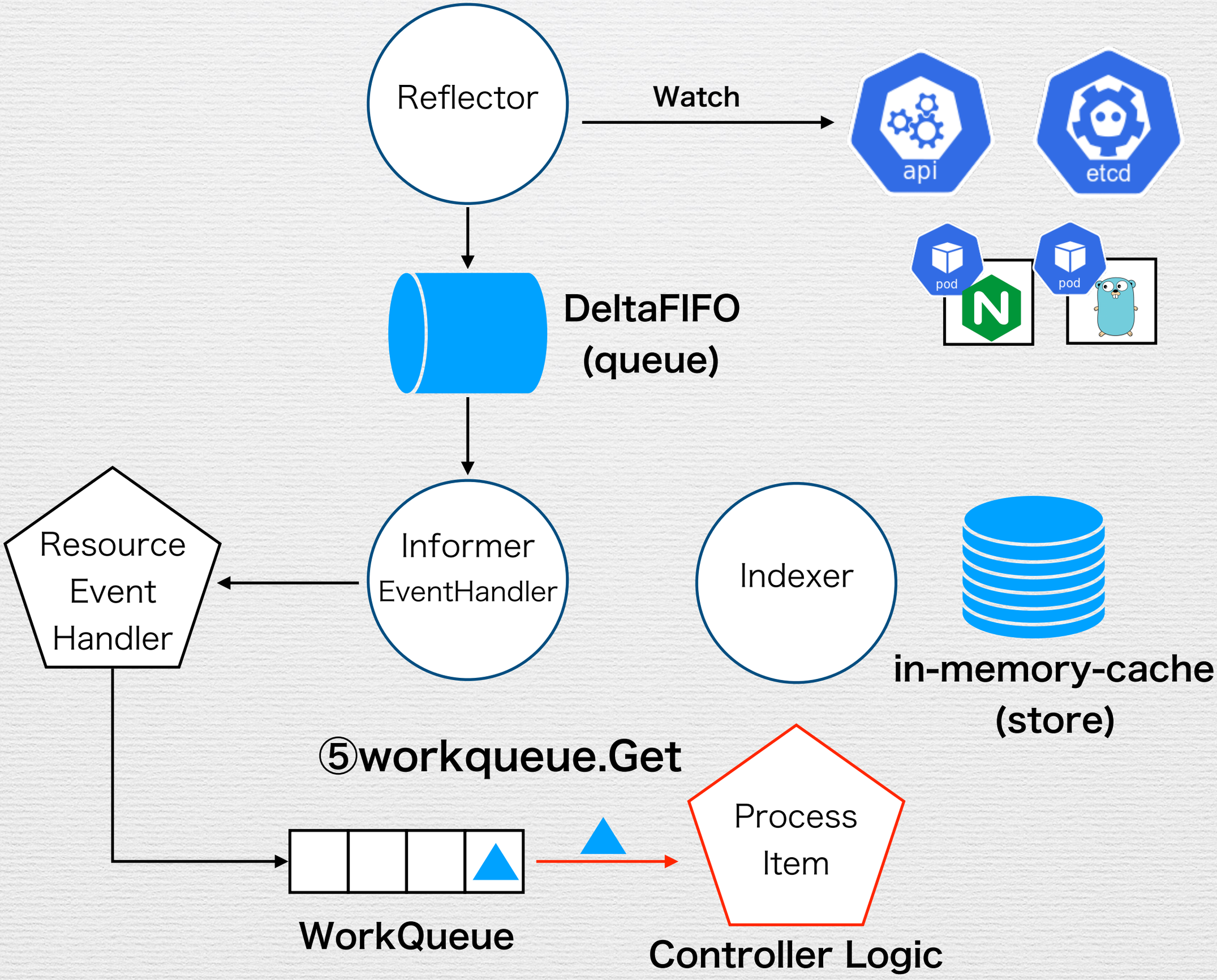
Detail of WorkQueue ~Enqueue~



Detail of WorkQueue ~Enqueue~



Detail of WorkQueue ~Dequeue~



Appendix) Informer enqueue Source Code

①

https://github.com/kubernetes/client-go/blob/master/tools/cache/reflector.go#L267

②

https://github.com/kubernetes/client-go/blob/release-13.0/tools/cache/controller.go#L153

③

https://github.com/kubernetes/client-go/blob/release-13.0/tools/cache/controller.go#L198

https://github.com/kubernetes/kubernetes/blob/release-1.16/pkg/controller/replicaset/replica_set.go#L153

※ ReplicaSet Controllerの場合

④

https://github.com/kubernetes/kubernetes/blob/release-1.16/pkg/controller/replicaset/replica_set.go#L417

※ ReplicaSet Controllerの場合

⑤

https://github.com/kubernetes/kubernetes/blob/release-1.16/pkg/controller/replicaset/replica_set.go#L438

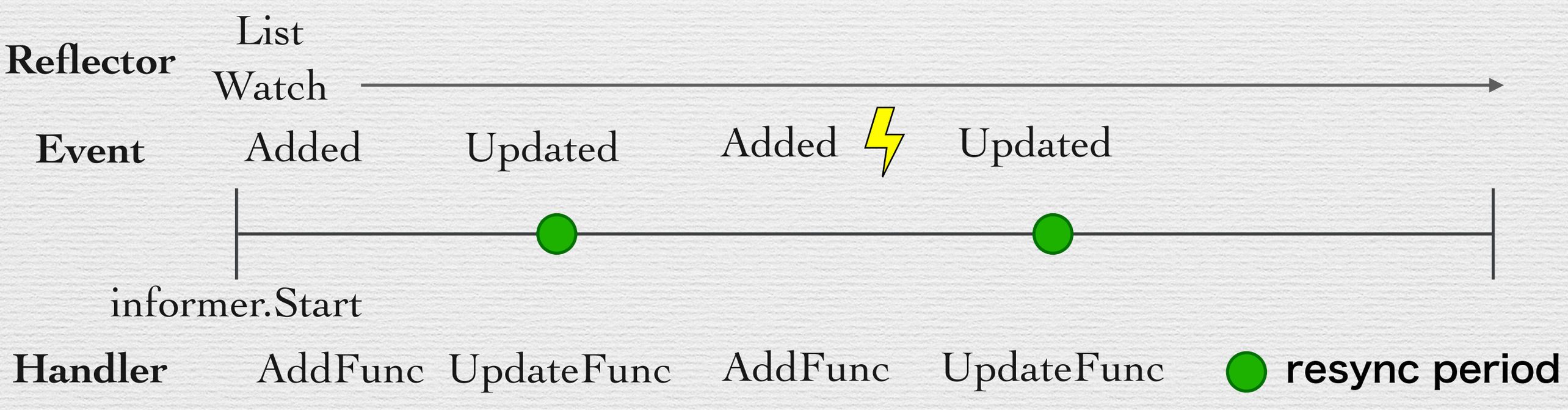
※ ReplicaSet Controllerの場合

Appendix) Informer Resync Period

Resync Period is option of InformerFactory.

Informer watches object events to api-server.
After Resync Period has passed, no matter what event has occurred,
UpdateFunc is called back.
As a result, Reconcile is executed again.

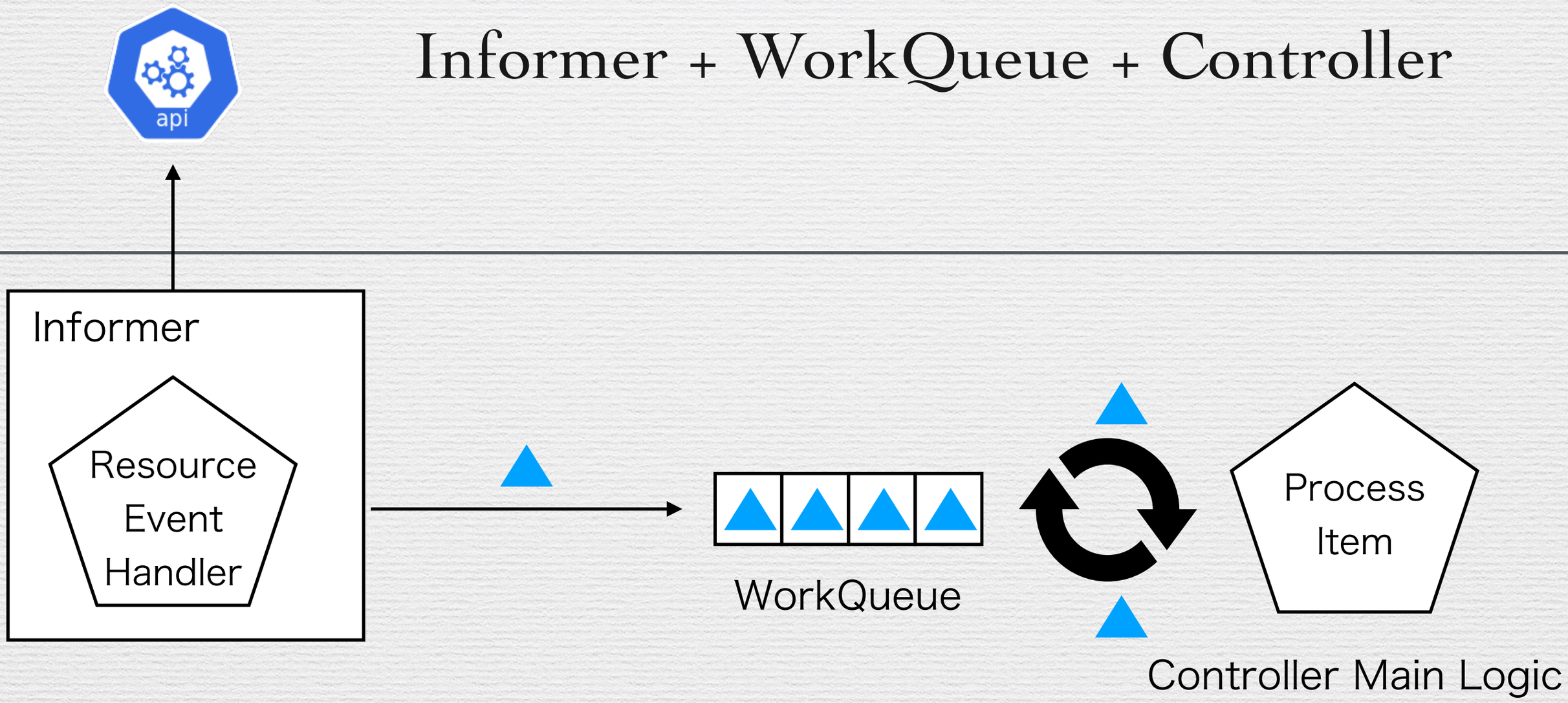
※This time, Resync refers in-memory-cache(not api-server).
Resync(cache sync) and Relist(list from api-server) is different.



Controller's Cycle
Main Logic

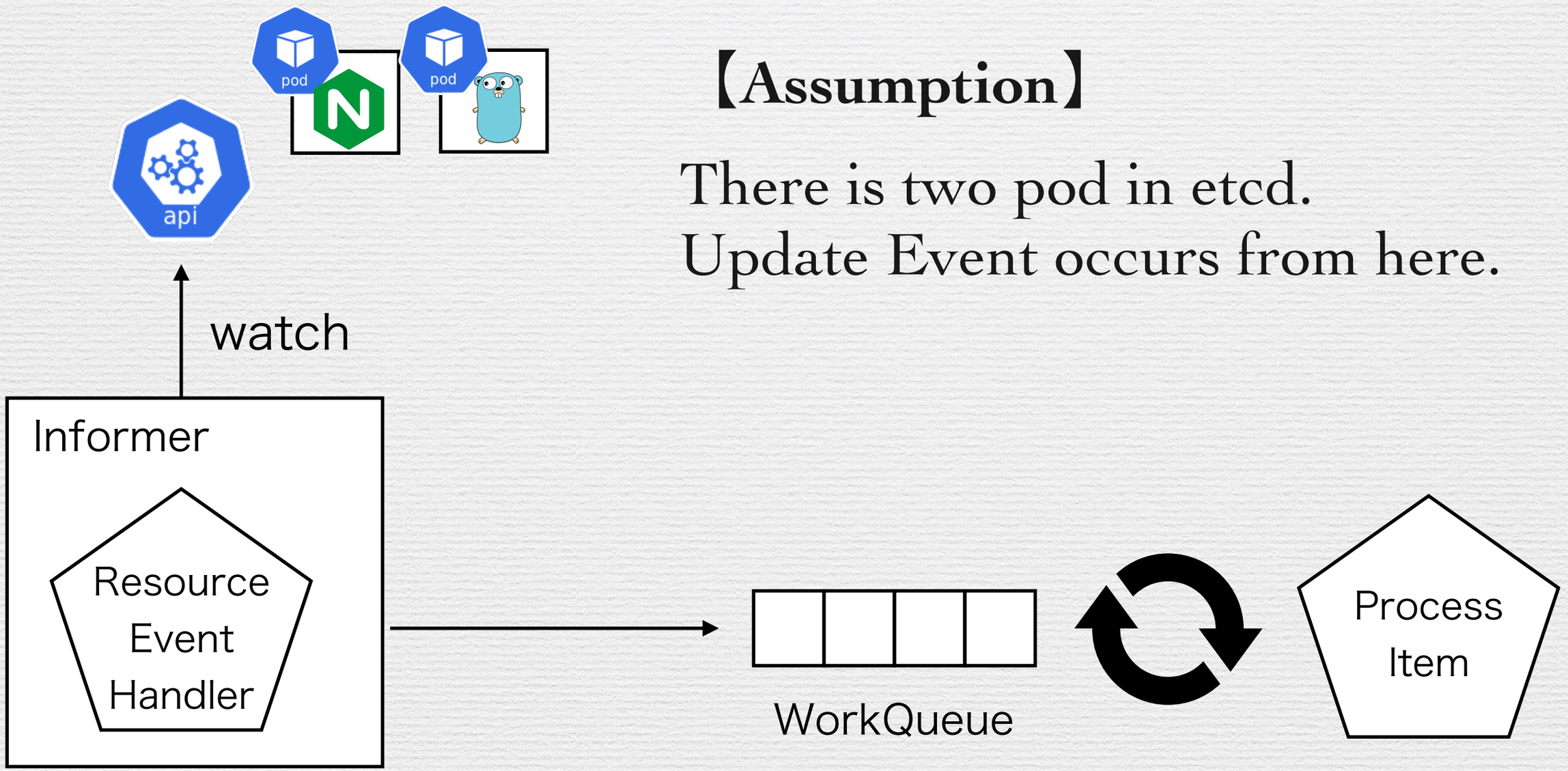
Controller's Cycle

Let's confirm Reconcile flow.
Informer + WorkQueue + Controller



Controller

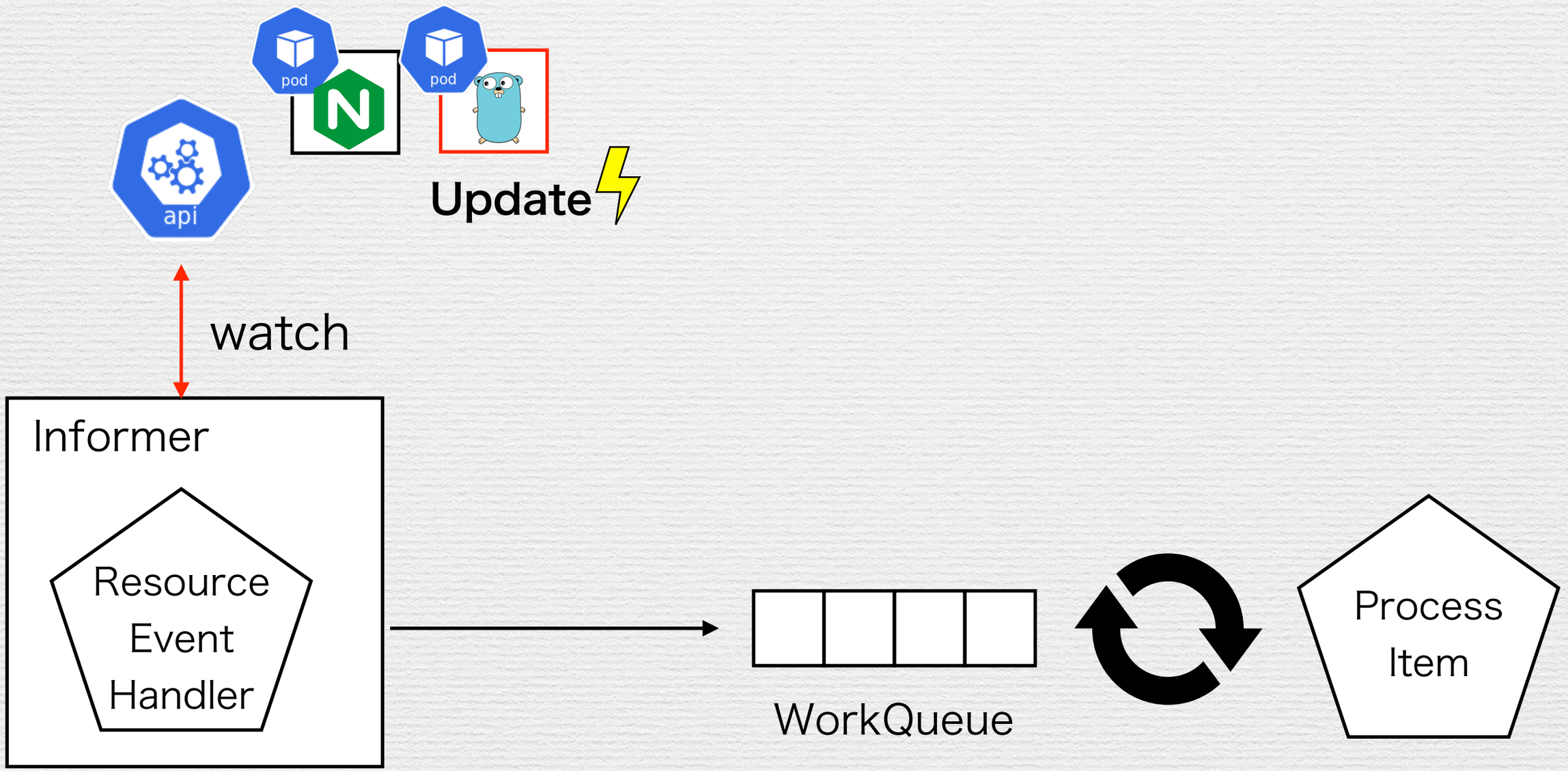
Controller's Cycle



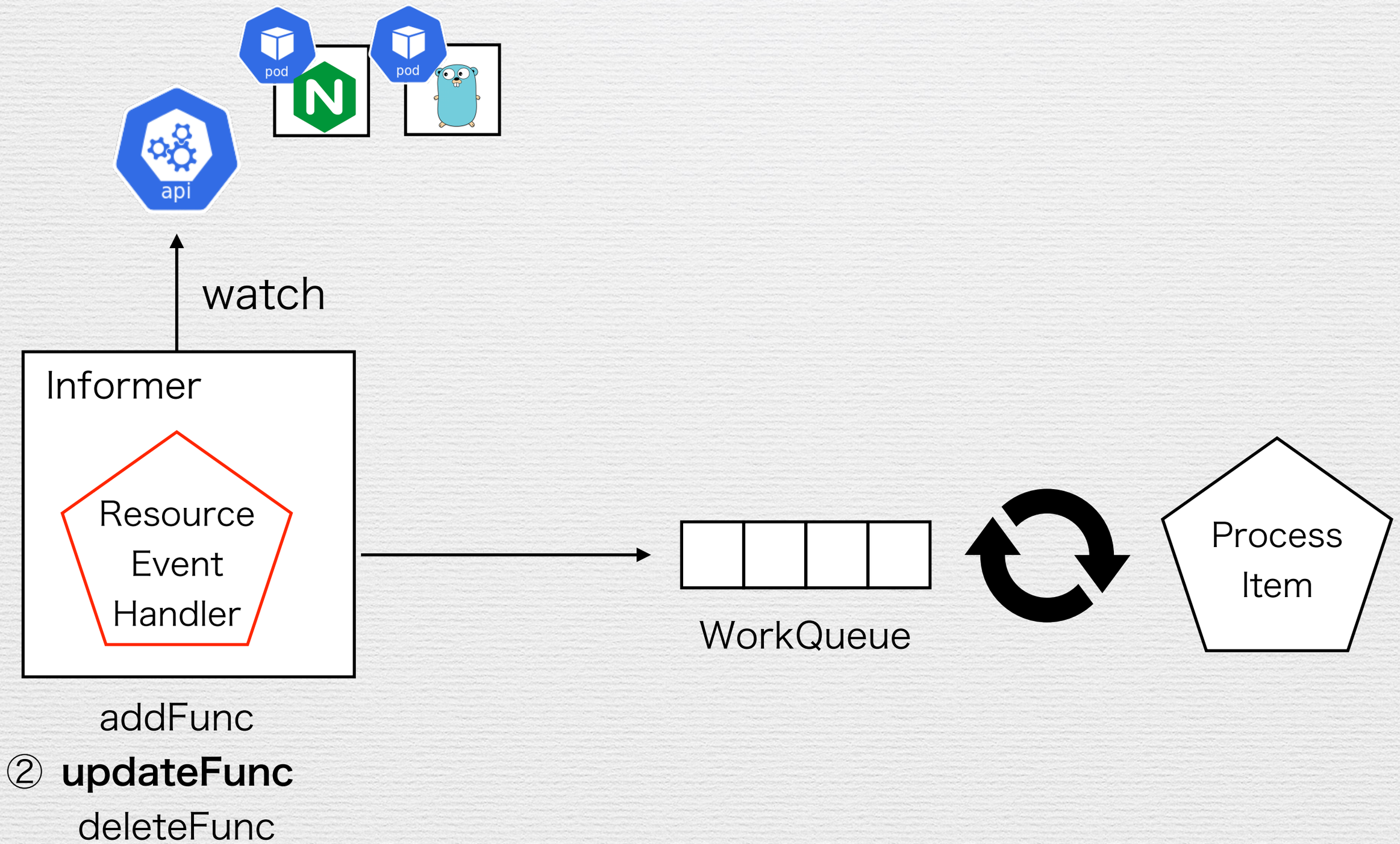
【Assumption】

There is two pod in etcd.
Update Event occurs from here.

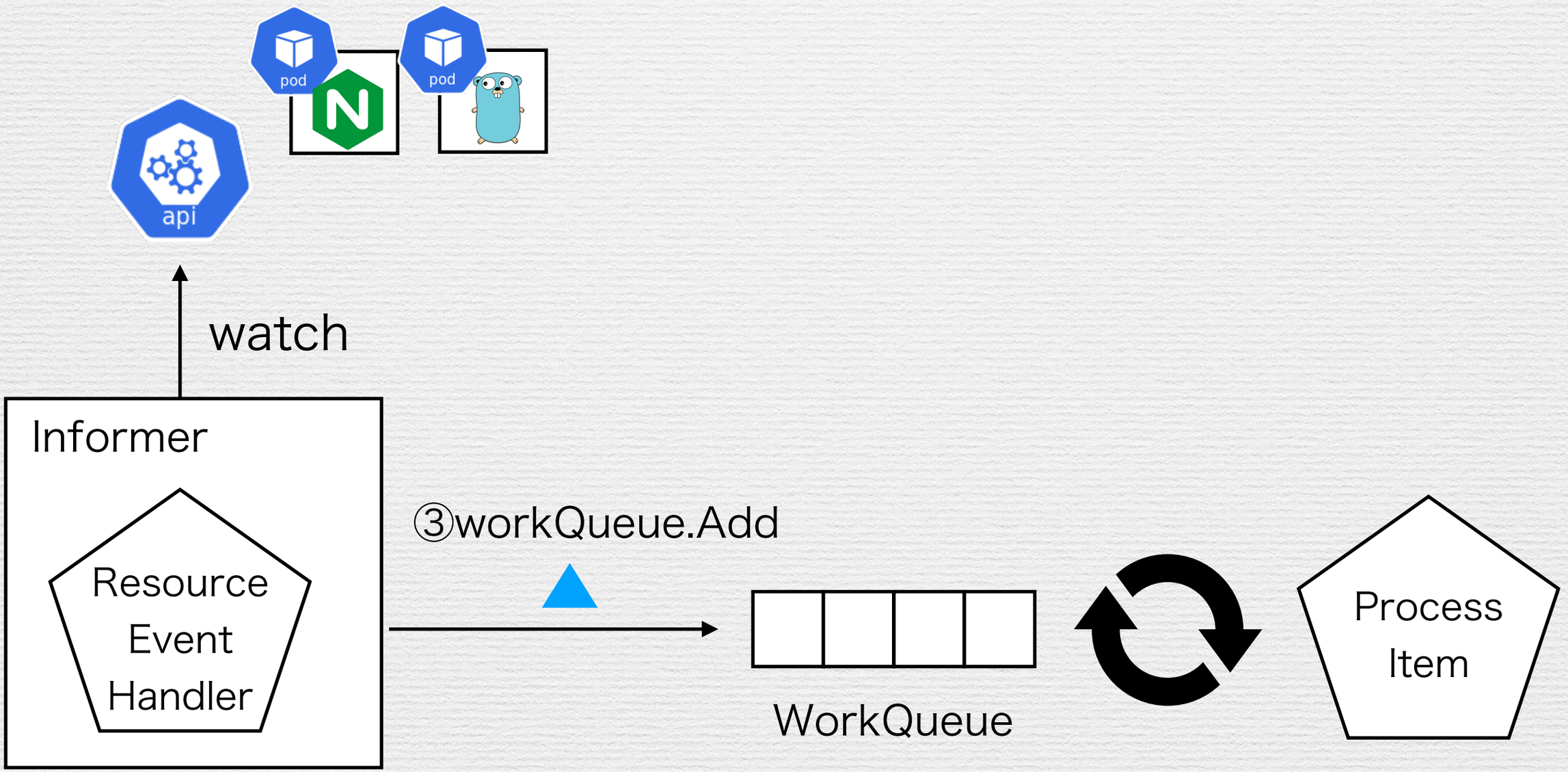
Controller's Cycle



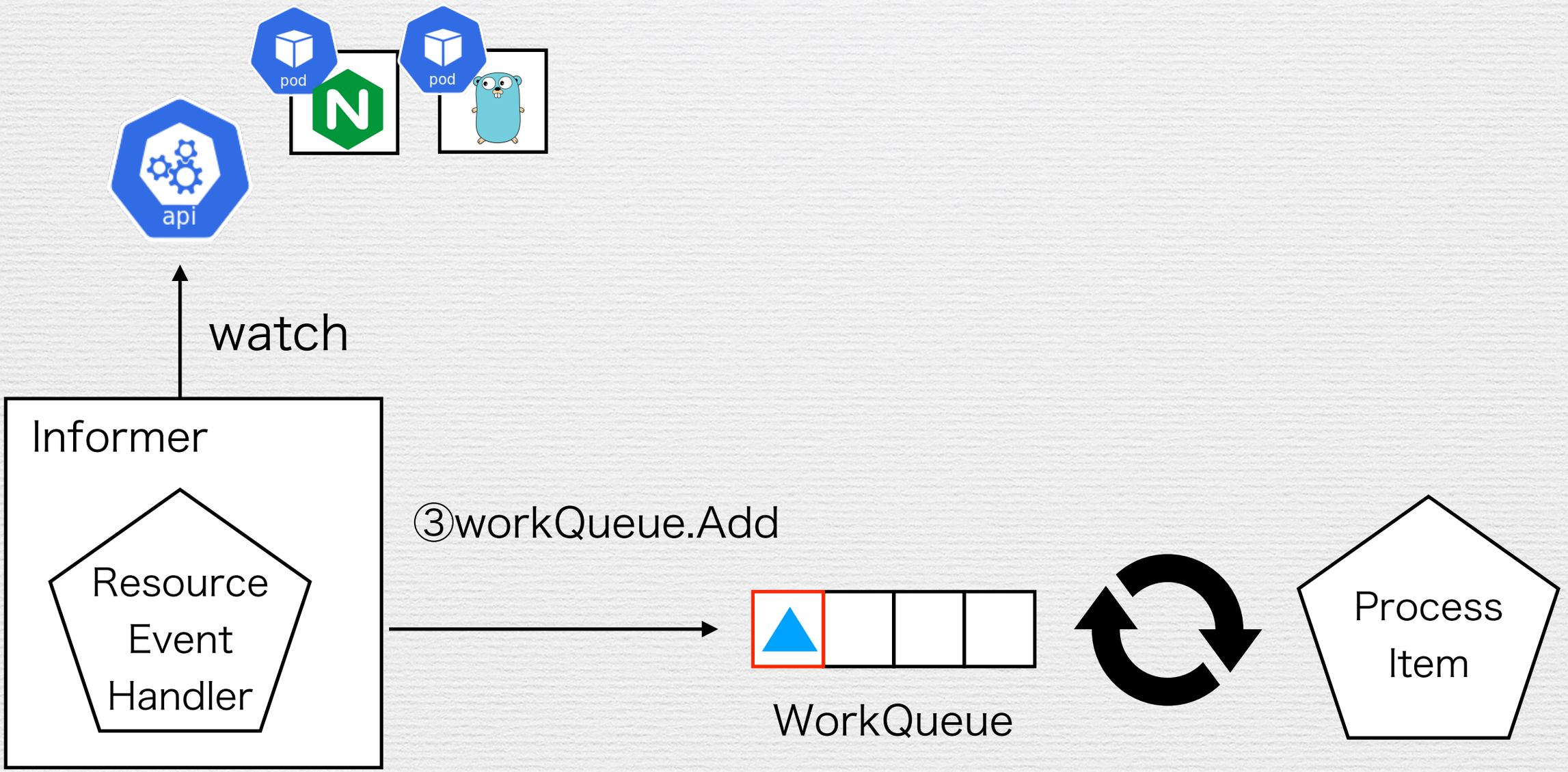
Controller's Cycle



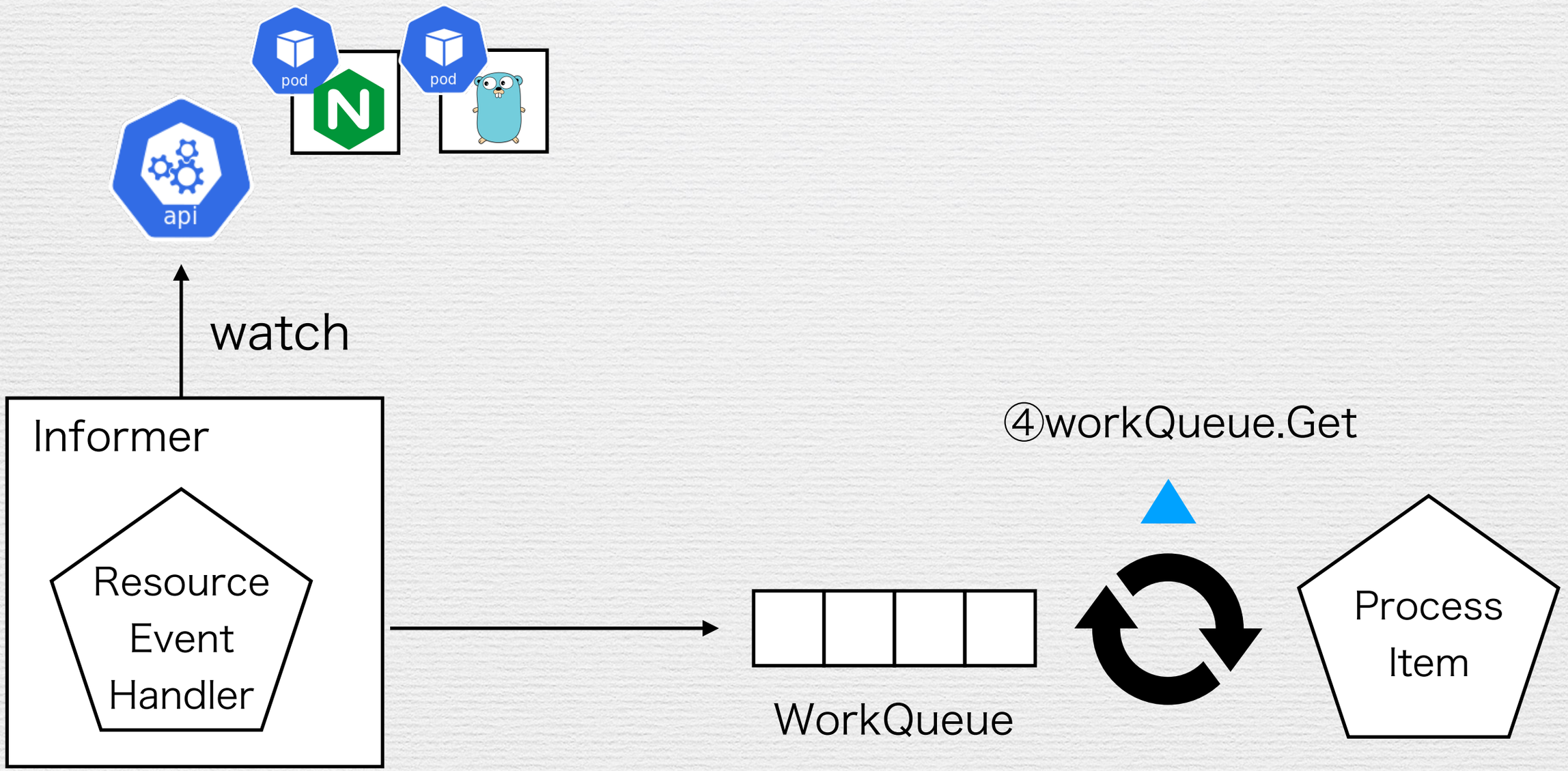
Controller's Cycle



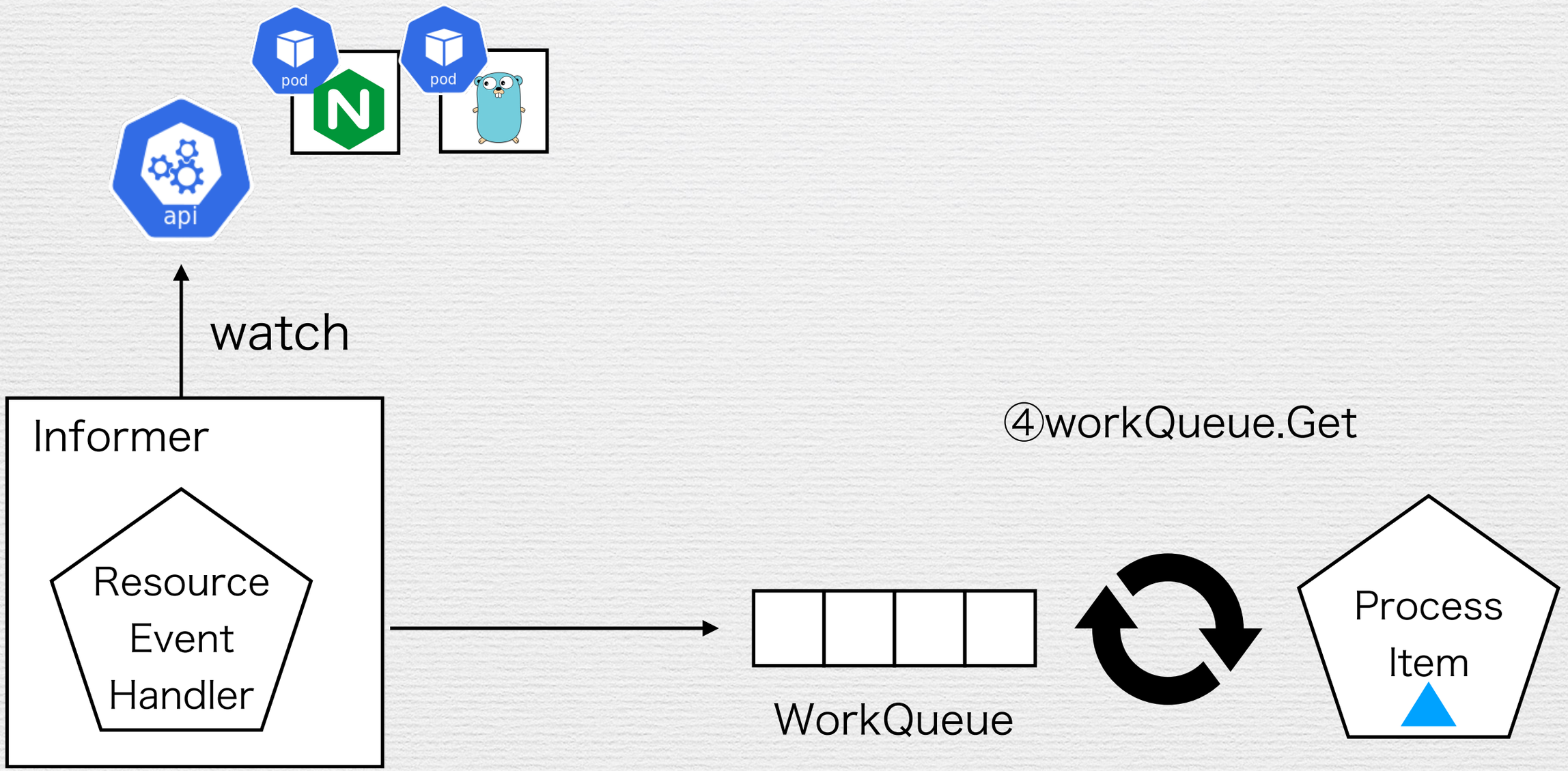
Controller's Cycle



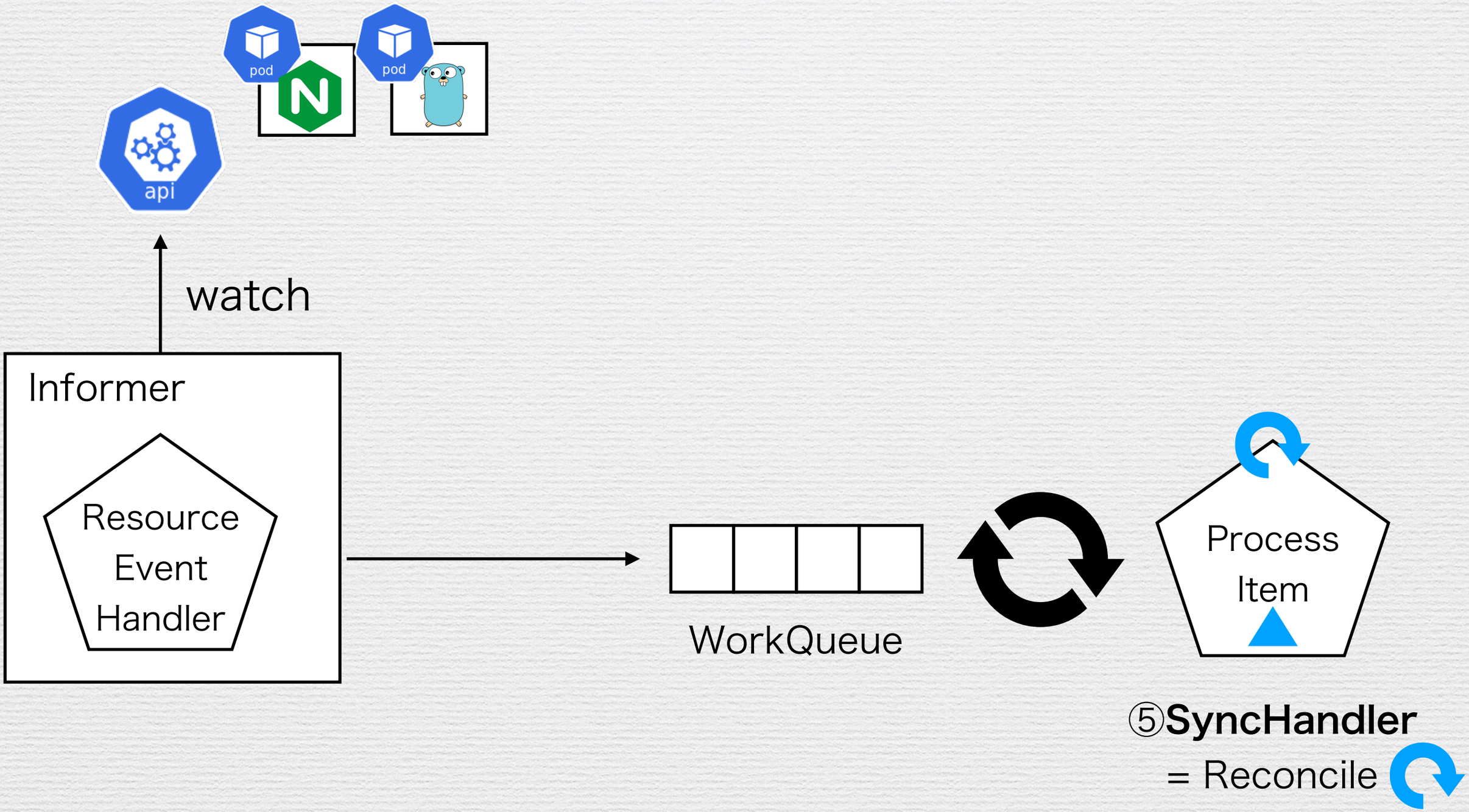
Controller's Cycle



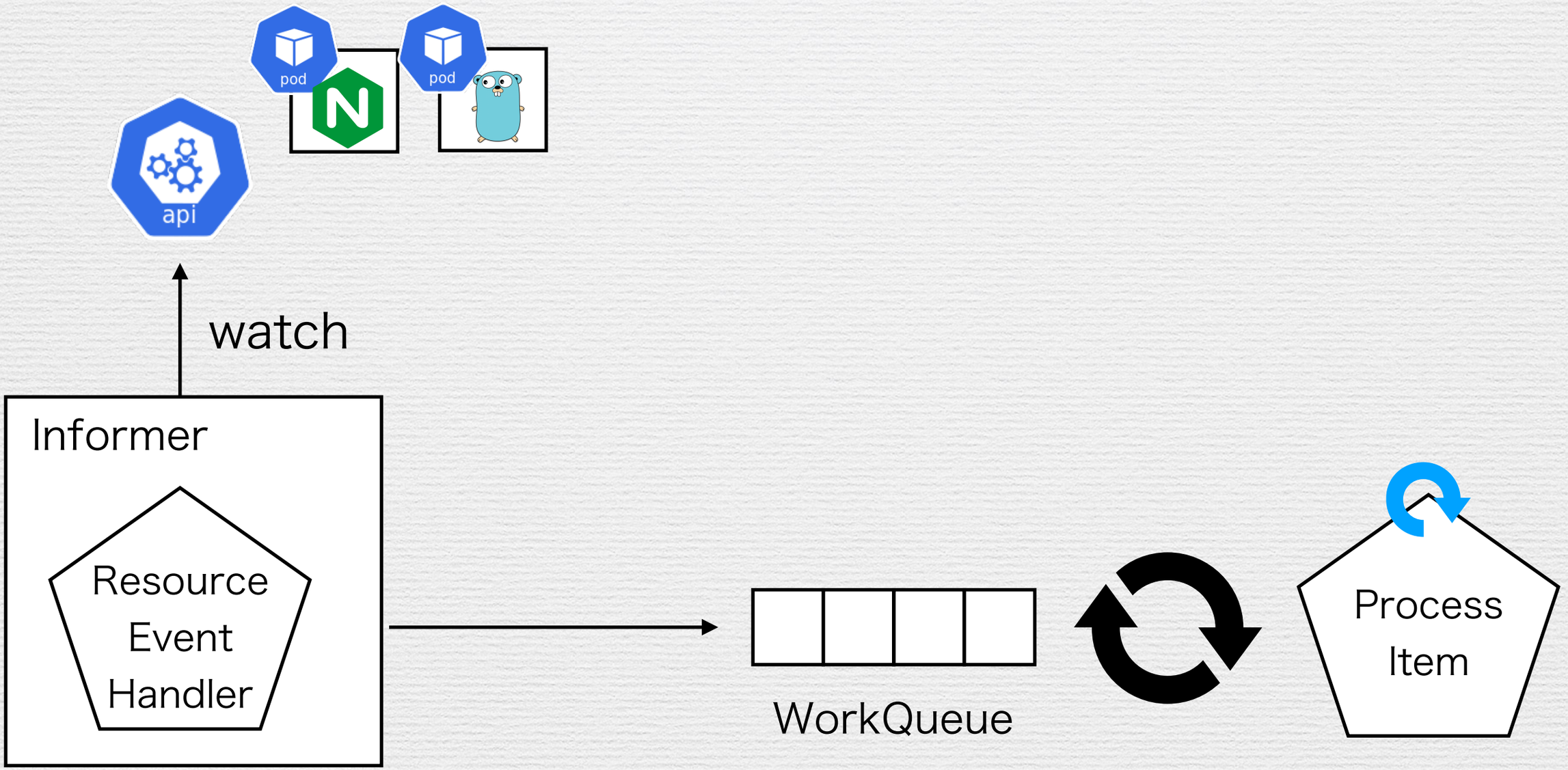
Controller's Cycle



Controller's Cycle



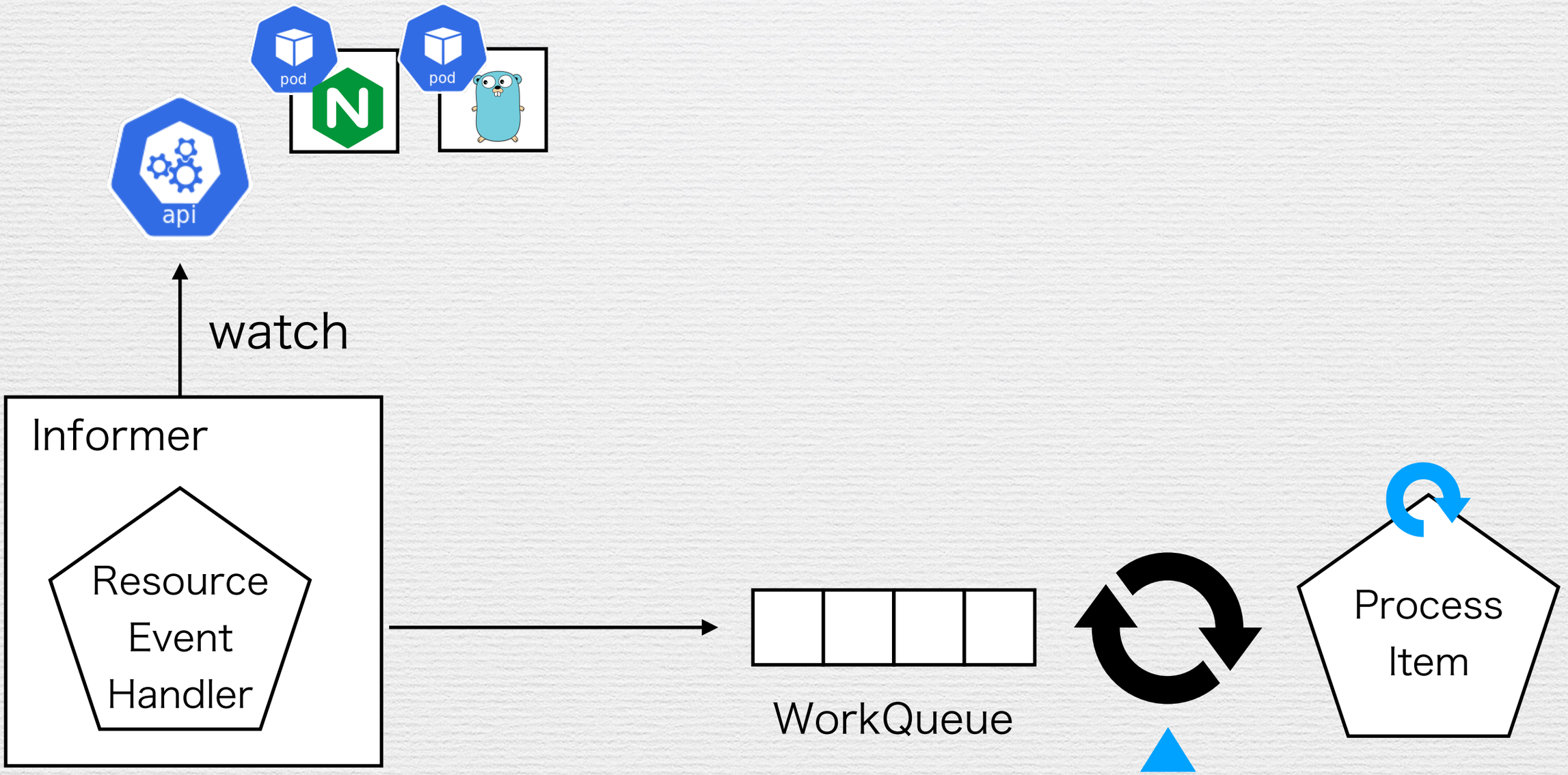
Controller's Cycle



When Reconcile finished successfully → ⑥workQueue.Forget
⑦workQueue.Done

The Item of Control Loop is removed
from WorkQueue completely

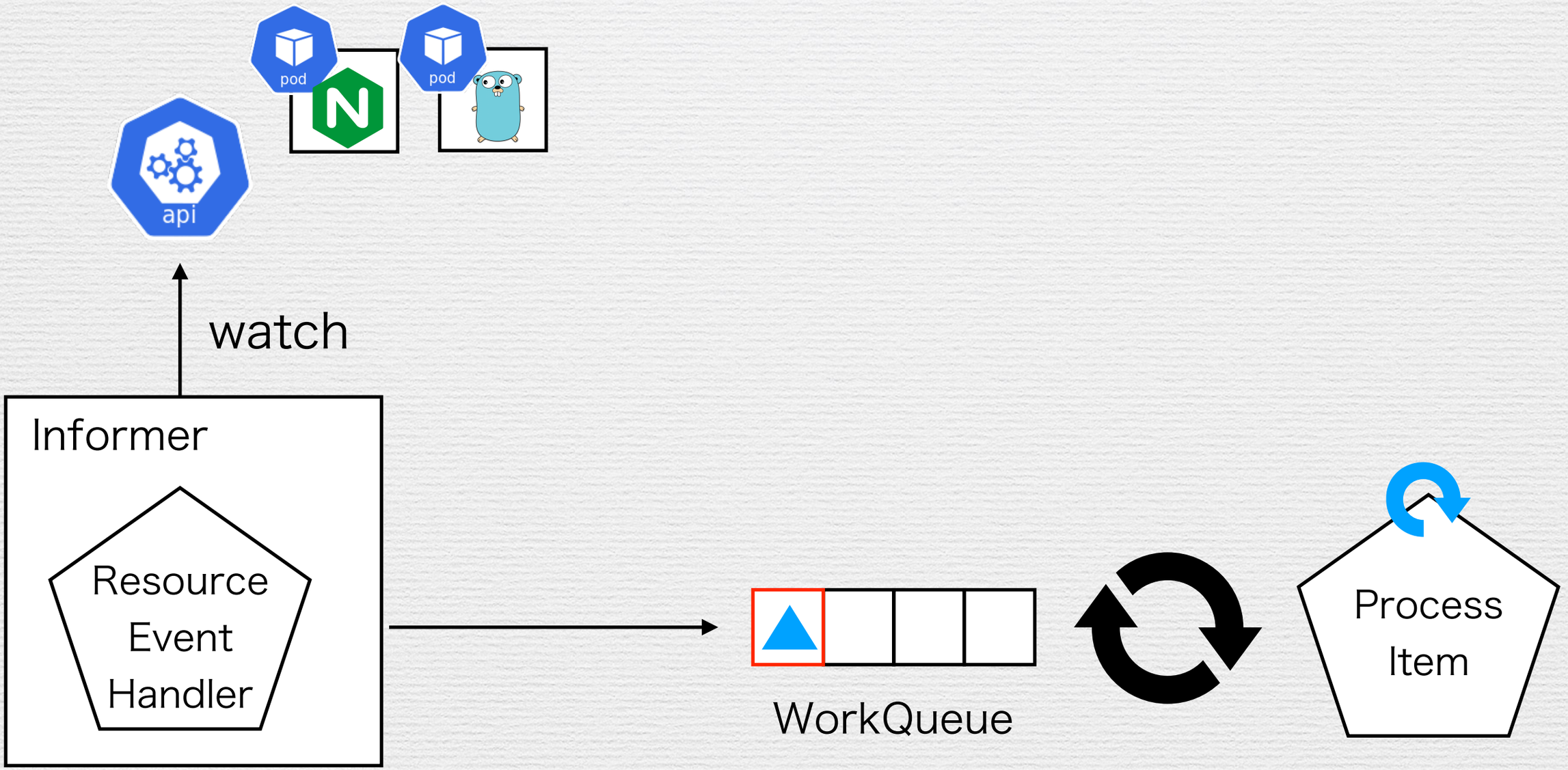
Controller's Cycle



When Reconcile ends with Error ⟶ ⑤' workQueue.AddRateLimited

Controller requeue item to WorkQueue.
And Reconcile will be executed again.

Controller's Cycle



When Reconcile ends with Error ⟶ ⑤' workQueue.AddRateLimited

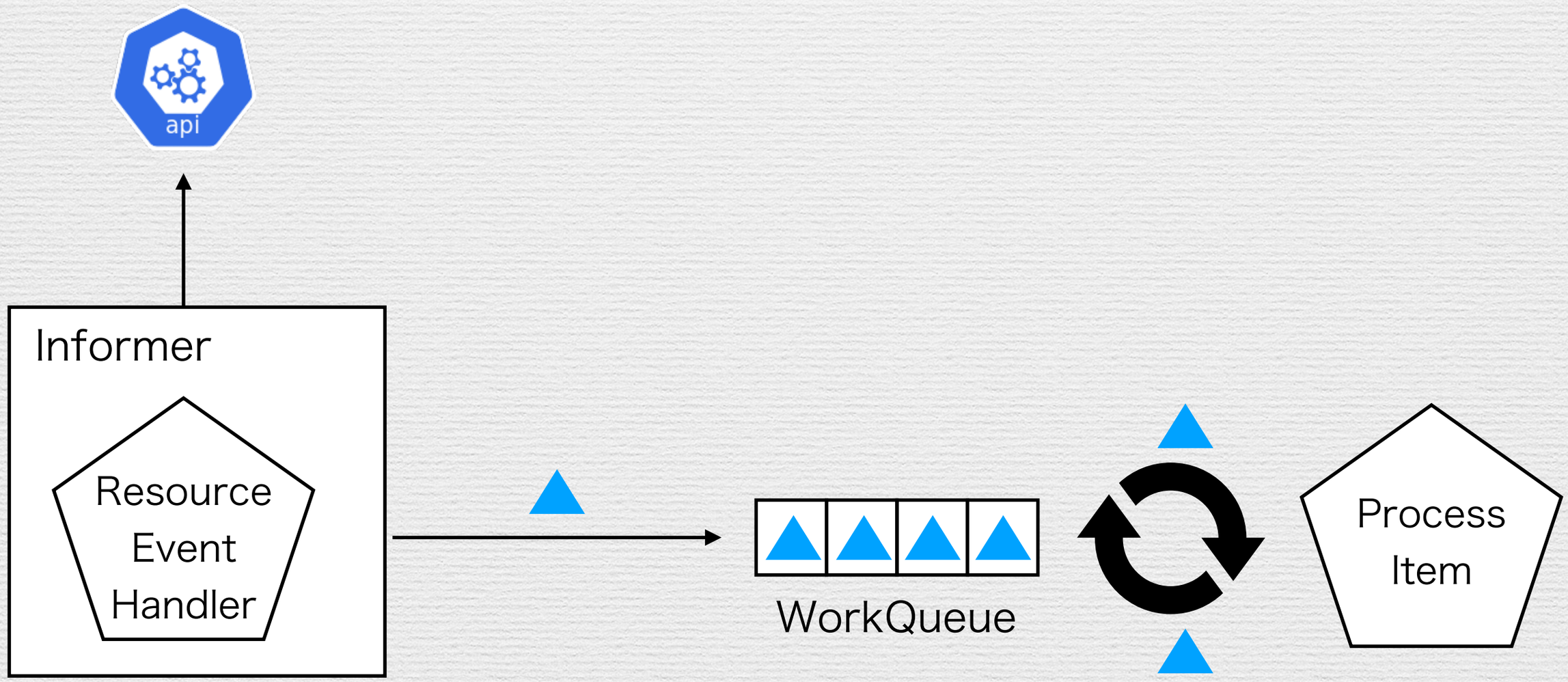
Controller requeue item to WorkQueue.
And Reconcile will be executed again.

Controller's Cycle

(Left Side of this slide)
Every time an event occurs, items continue to store in WorkQueue.

(Right Side of this slide)
Controller processes items in WorkQueue and executes Reconcile.

This loop continues endlessly until the Controller stops.



Controller's Basic Strategy

Read from In-memory-cache. Write to api-server.

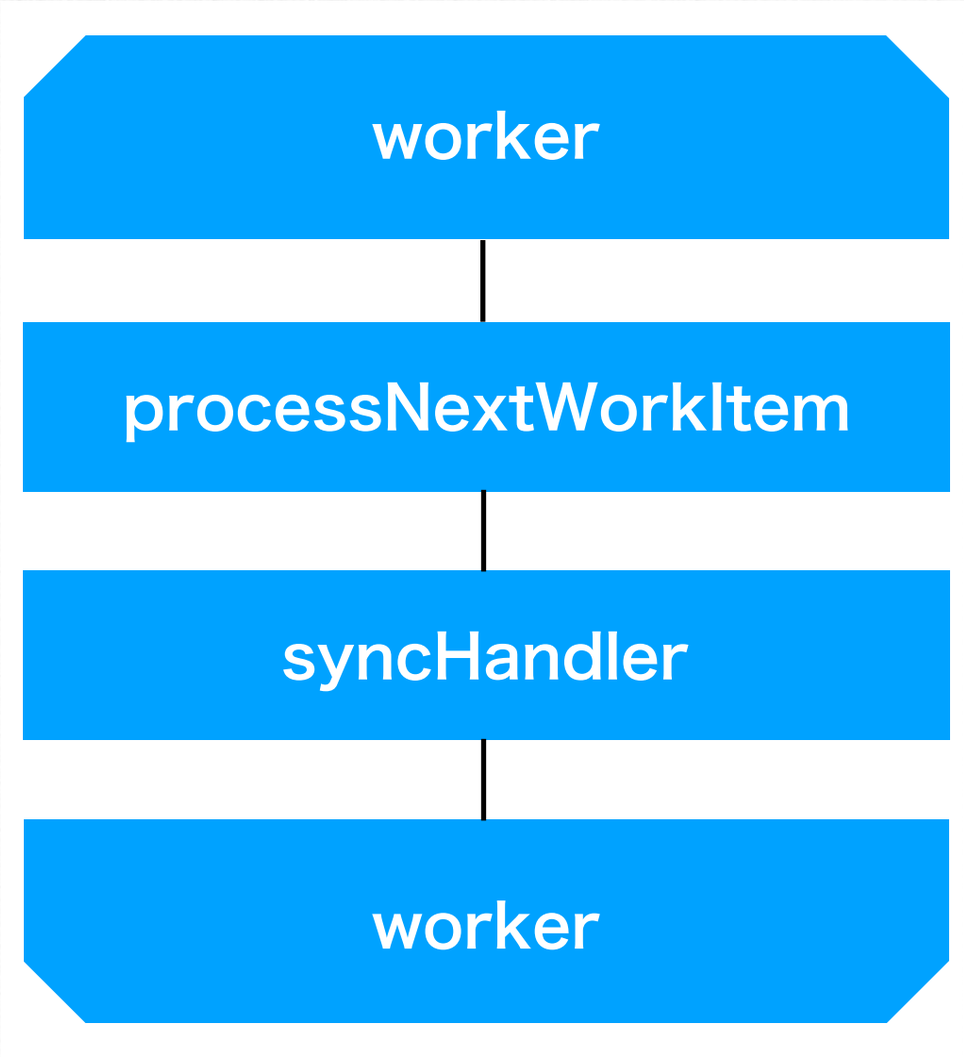
※ However if we update object in cache directly,
it is very difficult to guarantee its consistency.
So, we use DeepCopy(get clone data), when we update object.

e.g. kubernetes/pkg/controller/replicaset/replica_set.go

```
rs = rs.DeepCopy()
newStatus := calculateStatus(rs, filteredPods, manageReplicasErr)

// Always updates status as pods come up or die.
updatedRS, err := updateReplicaSetStatus(rsc.kubeClient.AppsV1(). ⇨
ReplicaSets(rs.Namespace), rs, newStatus)
```

https://github.com/kubernetes/kubernetes/blob/release-1.15/pkg/controller/replicaset/replica_set.go#L611

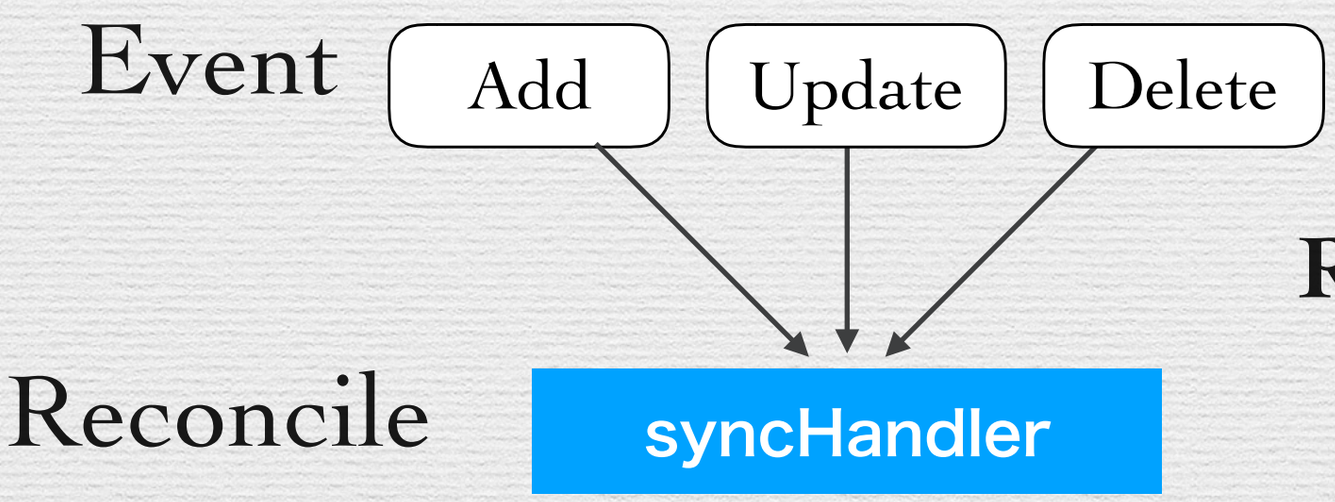
Controller's Main Logic



worker:
Endless Loop of processNextWorkItem

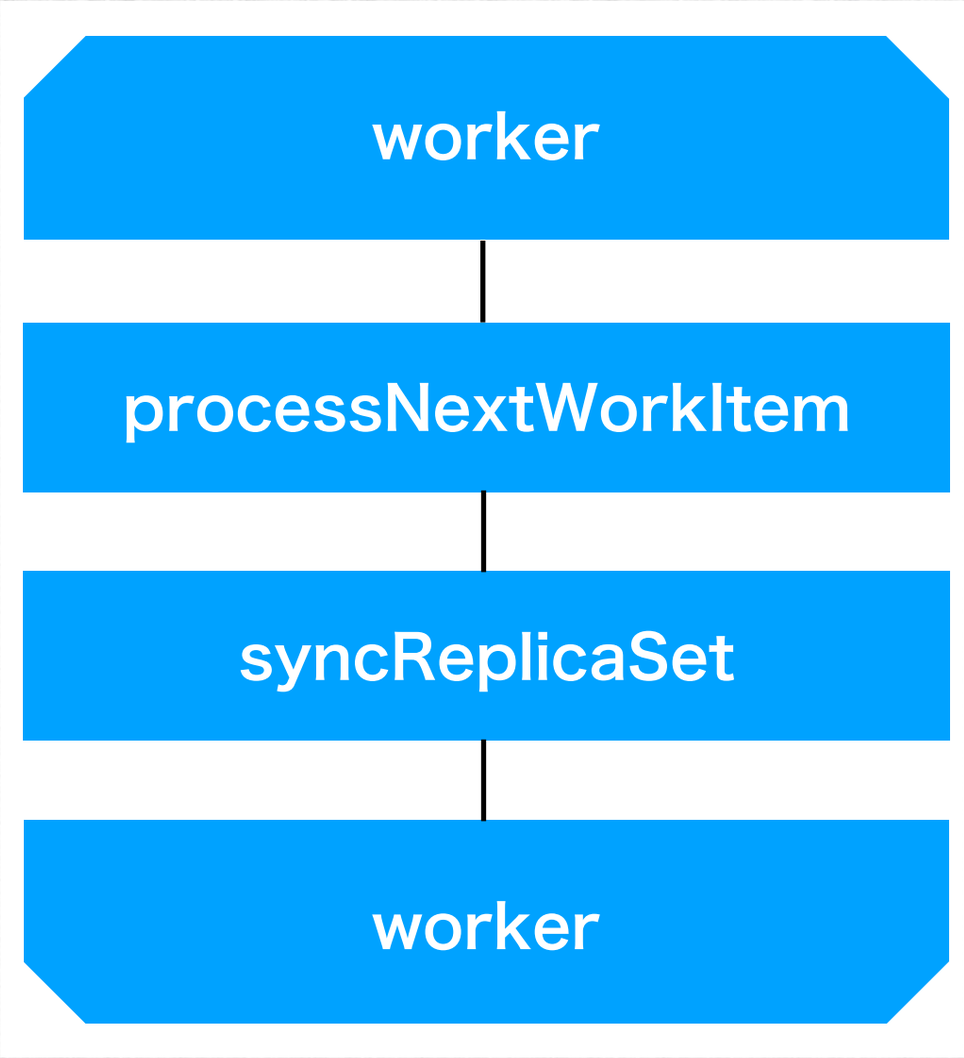
processNextWorkItem:
Operate WorkQueue(Get, Add) and Call Reconcile Logic

syncHandler:
This is equal to Reconcile Logic



Reconcile regardless of Event Type

Appendix) ReplicaSet Controller Soure Code



ReplicaSet Controller

Kubernetes v1.16

worker:

https://github.com/kubernetes/kubernetes/blob/release-1.16/pkg/controller/replicaset/replica_set.go#L432

processNextWorkItem:

https://github.com/kubernetes/kubernetes/blob/release-1.16/pkg/controller/replicaset/replica_set.go#L437

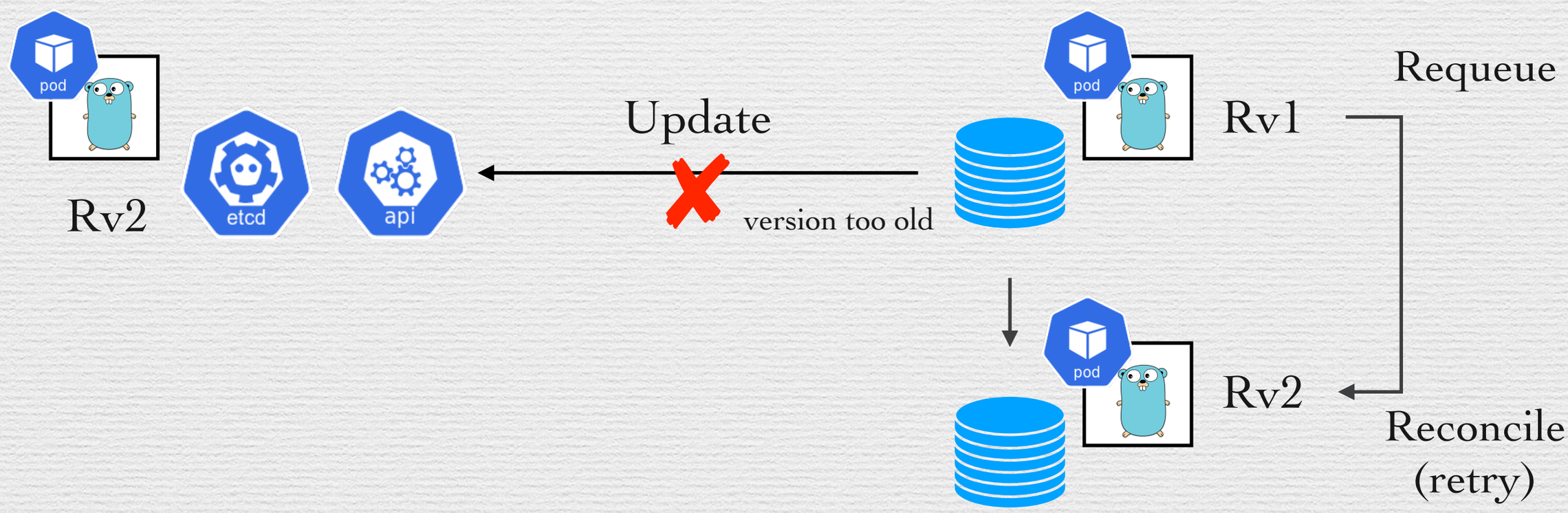
syncReplicaSet:

https://github.com/kubernetes/kubernetes/blob/release-1.16/pkg/controller/replicaset/replica_set.go#L562

Appendix) Sync of in-memory-cache and etcd

Informer synchronizes object data from etcd to in-memory-cache.

You may think whether in-memory-cache data is different from data of etcd.
It's no problem. Object has resourceVersion.
If resourceVersion of etcd and in-memory-cache is different, Error occurs when Controller updates object state.
Controller requeue and Reconcile until Reconcile finishes successfully.



Terminology(review)

Informer:

Watch Object Event, and store object data to in-memory-cache
Add items of Control Loop to WorkQueue via EventHandler

Lister:

Getter object data from in-memory-cache via Indexer

WorkQueue:

Queue which store items of Control Loop
This items is target of Reconcile Logic.
If error has occurs when Reconcile ends,
Controller requeue item to WorkQueue.
And Controller executes Reconcile again.

Controller Summary

Controller Summary

- Controller realizes declarative API by Control Loop (Reconciliation Loop)

- Kubernetes has distributed component.
  Event associates each component.

- client-go, apimachinery, code-generator are Library for Controller.

- Informer has two important role.
  ① Store object data to in-memory-cache
  ② Add items to WorkQueue via EventHandler

- Items which are stored in WorkQueue is processed by Reconcile.

Step up to Deep Dive

- Sample Controller

Link: https://github.com/kubernetes/sample-controller

```
mkdir -p $GOPATH/src/k8s.io && cd  $GOPATH/src/k8s.io && git clone https://github.com/kubernetes/sample-controller.git
export GO111MODULE=on
go build -o sample-controller .
./sample-controller -kubeconfig $HOME/.kube/config
```

- Kubernetes/Kubernetes

  - Deployment Controller

    https://github.com/kubernetes/kubernetes/blob/release-1.16/pkg/controller/deployment/deployment_controller.go

  - ReplicaSet Controller

    https://github.com/kubernetes/kubernetes/blob/release-1.16/pkg/controller/replicaset/replica_set.go

- Make Custom Controller(+ CRD)

  Kubebuilder: https://book.kubebuilder.io/

  Operator SDK: https://github.com/operator-framework/operator-sdk

Thank you!!!

Reference

Reference

- **Web Article**

- https://kubernetes.io/docs/concepts/architecture/nodes/
- https://kubernetes.io/docs/concepts/workloads/controllers/garbage-collection/
- https://github.com/kubernetes/community/blob/master/contributors/devel/sig-api-machinery/controllers.md
- A deep dive into Kubernetes controllers (https://engineering.bitnami.com/articles/a-deep-dive-into-kubernetes-controllers.html)
- Core Kubernetes: Jazz Improv over Orchestration (https://blog.heptio.com/core-kubernetes-jazz-improv-over-orchestration-a7903ea92ca)
- Events, the DNA of Kubernetes(https://www.mgasch.com/post/k8sevents/)

- **Presentation(Japanes)**

- Kubernete Meetup Tokyo #18 - Kubebuilder/controller-runtime 入門 (https://www.slideshare.net/pfi/kubernete-meetup-tokyo-18-kubebuildercontrollerruntime)
- Kubernetesのソースコードリーディング入門 (https://speakerdeck.com/smatsuzaki/kubernetesfalsesosukodorideinguru-men)

- **Book**

- Programming Kubernetes (https://programming-kubernetes.info/)

Reference

- **Repository**

- Kubernetes(https://github.com/kubernetes/kubernetes)
- Sample Controller(https://github.com/kubernetes/sample-controller)
- client-go(https://github.com/kubernetes/client-go)
- apimachinery(https://github.com/kubernetes/apimachinery)
- codegenerator(https://github.com/kubernetes/code-generator)
- what-happens-when-k8s(https://github.com/jamiehannaford/what-happens-when-k8s)