
KubeCon
CloudNativeCon
Europe 2023
TiKV

How to Develop a Robust Operator for Day-2 (Lesson Learned on KubeVirt/HCO)

Simone Tiraboschi

Operators: day 2 challenges

Different **challenges**:

- Upgrades and platform upgrades
- Failure recovery
- Backup/Restore
- Scaling
- Adding API/features
- Replacing API/features
- Deprecating API/features
- Bugs, memory leaks, …
- …

GOAL: provide a **robust operator** that the user can **trust** for **fully automatic** and **continuous upgrades**

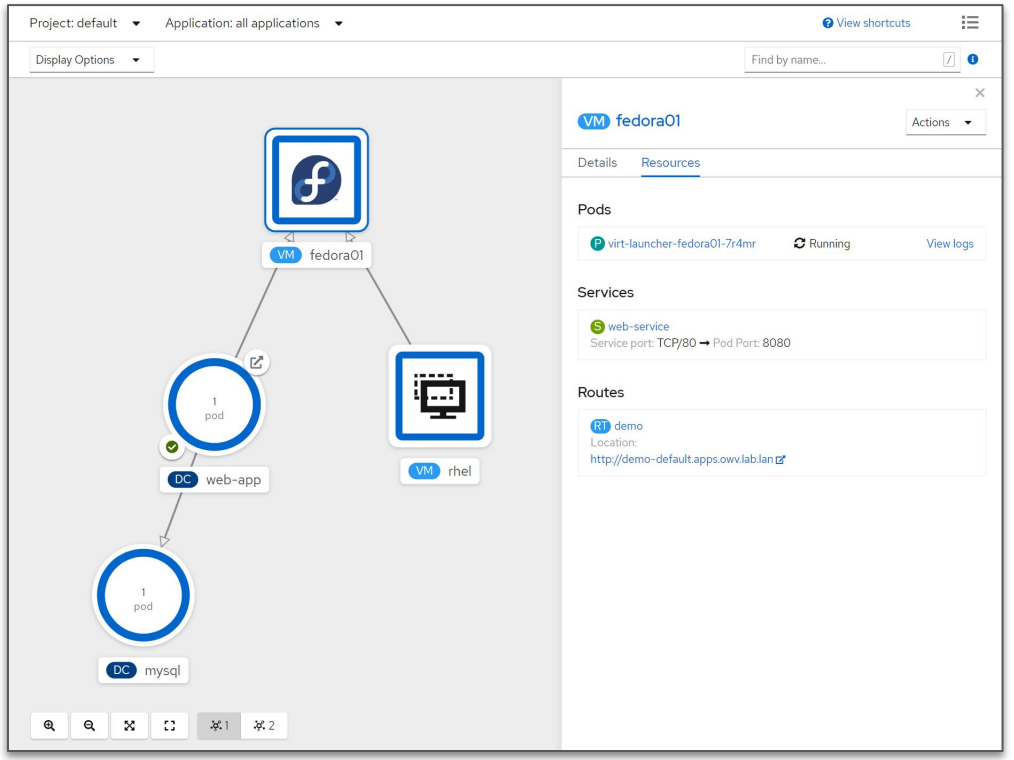
What's KubeVirt?

Kubernetes Virtualization API and runtime in order to define and manage **virtual machines**:

- Virtual machines
  - Running in **containers**
  - Using the **KVM** hypervisor
- **Scheduled**, deployed, and managed by **Kubernetes**
- **Integrated** with container orchestrator resources and services
  - Traditional Pod-like SDN connectivity and/or connectivity to external VLAN and other networks via multus
  - Persistent storage paradigm (PVC, PV, StorageClass)

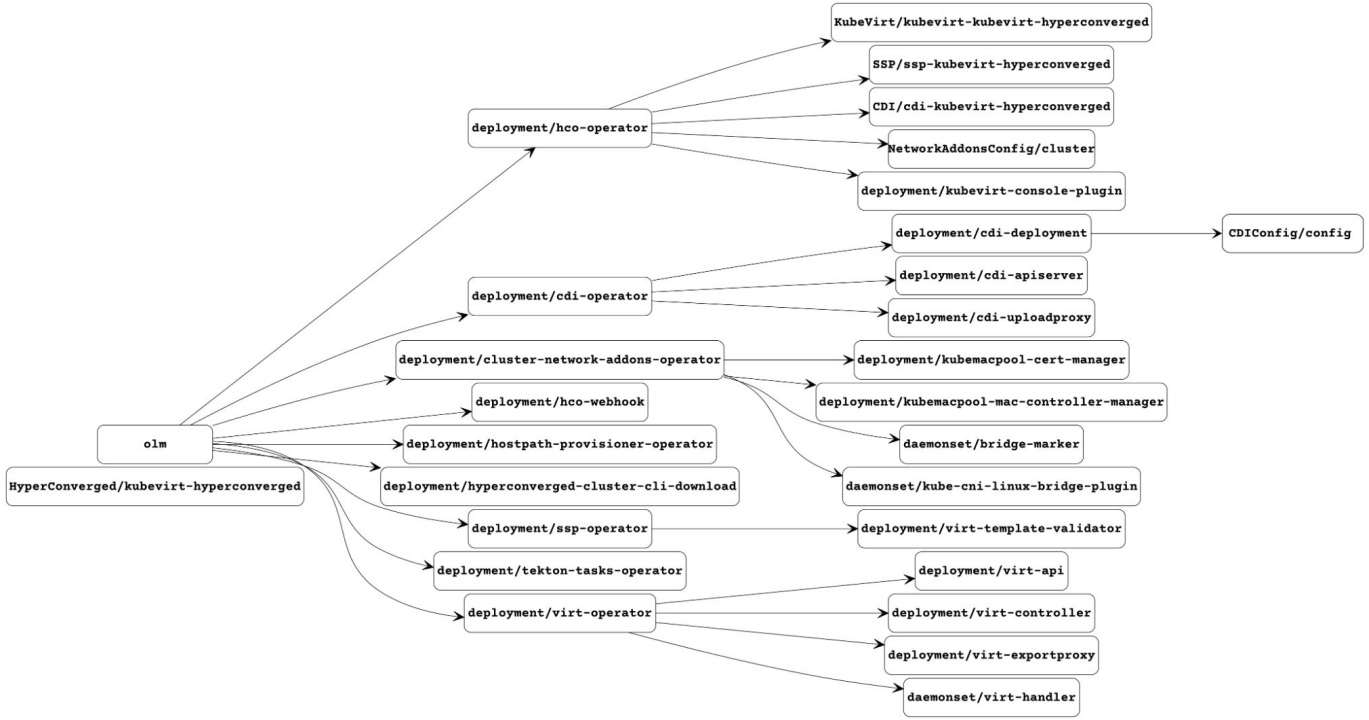
Why KubeVirt? Using VMs and containers together

- Follows **Kubernetes paradigms**:
  - Container Networking Interface (CNI)
  - Container Storage Interface (CSI)
  - Custom Resource Definitions (CRD, CR)
- Schedule, connect, and consume **VM** resources as **container-native**
- Virtual Machines connected to pod networks are **accessible** using standard Kubernetes methods:
  - **Service**
  - **Route**
  - **Ingress**
- **VM-to-pod**, and **vice-versa**, communication happens over SDN or ingress depending on network connectivity



What's the HyperConverged cluster Operator (HCO)?

- **hyperconverged-cluster-operator (HCO)** goal: provide a **single entrypoint** for multiple operators - kubevirt, cdi, networking, ect... - where users can deploy and configure them in a single object
- HCO doesn't replace the Operator Lifecycle Manager, rather works with it.
- HCO is an **operator of operators** or AKA as *meta-operator* or *umbrella operator*
- HCO provides an **opinionated deployment** of KubeVirt and it's sibling operators: It's the easiest way to install KubeVirt on your cluster



Shipped as:

- KubeVirt HyperConverged Cluster Operator on operatorHub.io (https://operatorhub.io/operator/community-kubevirt-hyperconverged)
- OpenShift Community Operators for OKD (about 25 releases)
- OpenShift Virtualization on OpenShift (about 45 releases)

Single entry point CR

- Aggregate all the tunable knobs into a **single configuration object**, avoid spreading configuration in various config maps with implicit schema
- **Self explanatory** and **well defined** schema
- **Declarative**: What vs How
- State separation: **desired** state vs **observed** state

```
stirabos@t14s:~$ kubectl explain hco.spec
KIND:     HyperConverged
VERSION:  hco.kubevirt.io/v1beta1

RESOURCE: spec <Object>

DESCRIPTION:
     HyperConvergedSpec defines the desired state of HyperConverged

FIELDS:
   certConfig    <Object>
     certConfig holds the rotation policy for internal, self-signed certificates

   commonTemplatesNamespace    <string>
     CommonTemplatesNamespace defines namespace in which common templates will
     be deployed. It overrides the default openshift namespace.

   dataImportCronTemplates    <[]Object>
     DataImportCronTemplates holds list of data import cron templates (golden
     images)

   defaultCPUModel    <string>
     DefaultCPUModel defines a cluster default for CPU model: default CPU model
     is set when VMI doesn't have any CPU model. When VMI has CPU model set,
     then VMI's CPU model is preferred. When default CPU model is not set and
     VMI's CPU model is not set too, host-model will be set. Default CPU model
     can be changed when kubevirt is running.

   featureGates    <Object>
     featureGates is a map of feature gate flags. Setting a flag to `true` will
     enable the feature. Setting `false` or removing the feature gate, disables
     the feature.

   filesystemOverhead    <Object>
     FilesystemOverhead describes the space reserved for overhead when using
     Filesystem volumes. A value is between 0 and 1, if not defined it is 0.055
     (5.5 percent overhead)


…
```

Bad practice: abusing annotations

Annotations are not APIs:

- Schema?
- Validation?
- Status?
- Are we really advertising a feature?

```
apiVersion: hco.kubevirt.io/v1beta1
kind: HyperConverged
metadata:
 annotations:
   kubevirt.io/cpu-limit-to-request-ratio: "2"
   kubevirt.io/memory-limit-to-request-ratio: "1.5"
   deployOVS: "false"
 creationTimestamp: "2023-04-11T09:37:03Z"
 finalizers:
   - kubevirt.io/hyperconverged
 generation: 2
 labels:
   app: kubevirt-hyperconverged
 name: kubevirt-hyperconverged
 namespace: kubevirt-hyperconverged
```

"Nothing lasts longer than a temporary but dirty solution"

Reference: alway refer to the Kubernetes API Conventions doc from sig-architecture

https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md

Counterargument: annotations as an escape hatch

HCO is reconciling its operands to an opinionated state exposing only properly validated features.

And if we want to experiment/hack/… (eg. exposing an experimental feature of a sub-component) ?

```
apiVersion: hco.kubevirt.io/v1beta1
kind: HyperConverged
metadata:
 annotations:
   kubevirt.kubevirt.io/jsonpatch: |-
     [
       {
         "op":"add",
         "path":"/spec/configuration/developerConfiguration/featureGates/-",
         "value":"CPUManager"
       }
     ]
 name: kubevirt-hyperconverged
```

HCO will raise an alert reporting a tainted configuration

Even recursively…

```
metadata:
 annotations:
   kubevirt.kubevirt.io/jsonpatch : |-
     [
       {
         "op": "add",
         "path": "/spec/customizeComponents/patches",
         "value": [{
             "patch": "[{\"op\":\"add\",\"path\":\"/spec/template/spec/containers/0/command/-\",\"value\":\"--max-devices=250\"}]",
             "resourceName": "virt-handler",
             "resourceType": "Daemonset",
             "type": "json"
         }]
       }
     ]
```

Proper way: feature gates

Feature gates on HCO:

- Simple feature-flagging
- It could still become an anti-pattern if abused

```
// HyperConvergedFeatureGates is a set of optional feature gates to enable or di
// by default yet.
// +k8s:openapi-gen=true
type HyperConvergedFeatureGates struct {
  // Allow migrating a virtual machine with CPU host-passthrough mode. This shou
  // enabled only when the Cluster is homogeneous from CPU HW perspective
  // +optional
  // +kubebuilder:default=false
  // +default=false
  WithHostPassthroughCPU *bool `json:"withHostPassthroughCPU,omitempty"`

  // Opt-in to automatic delivery/updates of the common data import cron templat
  // There are two sources for the data import cron templates: hard coded list o
  // templates that can be added to the dataImportCronTemplates field. This feat
  // templates. It is possible to use custom templates by adding them to the dat
  // +optional
  // +kubebuilder:default=true
  // +default=true
  EnableCommonBootImageImport *bool `json:"enableCommonBootImageImport,omitempty

  // deploy resources (kubevirt tekton tasks and example pipelines) in Tekton ta
  // +optional
  // +kubebuilder:default=false
  // +default=false
  DeployTektonTaskResources *bool `json:"deployTektonTaskResources,omitempty"`
```

HCO feature gates: implementation details

Optional values

*bool: can be True, False or nil (it allows distinguishing unset from explicitly False)

```
// Allow migrating a virtual machine with CPU host-passthrough mode. This should be
// enabled only when the Cluster is homogeneous from CPU HW perspective
// +optional
// +kubebuilder:default=false
// +default=false
WithHostPassthroughCPU *bool `json:"withHostPassthroughCPU,omitempty"`
```

kubebuilder defaults in the openAPIV3Schema

```
schema:
 openAPIV3Schema:
   description: HyperConverged is the Schema for ...
   properties:
     ...
     spec:
       ...
       description: HyperConvergedSpec ...
       properties:
         ...
         featureGates:
           ...
           properties:
             ...
             withHostPassthroughCPU:
               default: false
               description: Allow migrating a ...
               type: boolean
```

Static defaulter-gen from k8s.io/code-generator

```
$ defaulter-gen ... --output-file-base zz_generated.defaults \
--output-package github.com/kubevirt/hyperconverged-cluster-operator/api/v1beta1
```

```
// Code generated by defaulter-gen. DO NOT EDIT.

// RegisterDefaults adds defaulters functions to the given scheme.
// Public to allow building arbitrary schemes.
// All generated defaulters are covering - they call all nested defaulters.
func RegisterDefaults(scheme *runtime.Scheme) error {
...


func SetObjectDefaults_HyperConverged(in *HyperConverged) {
  if in.Spec.FeatureGates.WithHostPassthroughCPU ==nil {
     var ptrVar1 bool = false
     in.Spec.FeatureGates.WithHostPassthroughCPU = &ptrVar1
  }
...
```

Defaulting

Static code generated defaulter functions	Defaults in the openAPIV3Schema	Admission Controlled Defaults	Controller-Assigned Defaults (aka Late Initialization)

The defaulter functions can be evaluated (explicitly) from go code without the need of a real APIServer. Useful in unit tests.
 	
`defaultScheme := runtime.NewScheme()`
`hcov1beta1.AddToScheme(defaultScheme)`
`hcov1beta1.RegisterDefaults(defaultScheme)`
`defaultHco := &hcov1beta1.HyperConverged{`
`  TypeMeta: metav1.TypeMeta{`
`  APIVersion: util.APIVersion,`
`  Kind: util.HyperConvergedKind,`
`  },`
`  ObjectMeta: metav1.ObjectMeta{`
`  Name: crName,`
`  }}`
`defaultScheme.Default(defaultHco)`
	Defaulting happens in the APIServer when reading from etcd
	
	***"Defaults applied when reading** data from etcd are not automatically written back to etcd. An update request via the API is required to persist those defaults back into etcd":* Be aware of Patch vs Update behaviours
		Implemented with a mutating admission webhook.
		
		It allows implementing sophisticated logic to set default values which are not derived from the object in question.
		
		Default values are injected before reaching the APIServer so subsequent GETs of the resource will include the default values explicitly as they are stored there.
			Late initialization is when resource fields are set (asynchronously) by a system controller after an object is created/updated as a result of other actions of external events.
			
			Not strictly "defaulting" but a common pattern.
			
			It should handle race conditions and it will cause a subsequent reconciliation loop.

Golden rule: we never want to change or override a value that was provided by the user.

If they requested something invalid, they should get an error: validation!

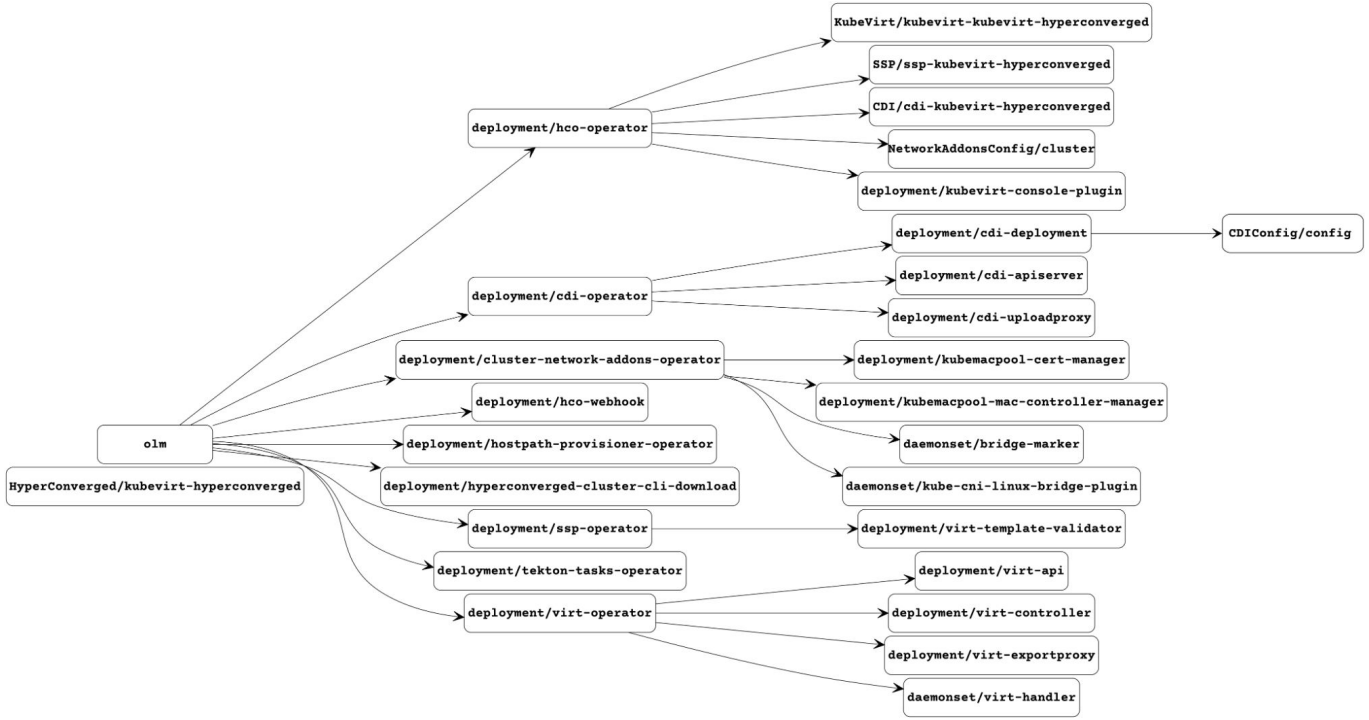
Validation

OpenAPI v3 schemas	Common Expression Language (CEL) validation rules for CRDs	Validating Admission Webhooks

• Metadata validation
• Schema validation
• Value validation (single field):
  ○ maxLength
  ○ maxItems
  ○ maxProperties
  ○ enum
  ○ minimum
  ○ Maximum
  ○ …
• String formats:
  ○ date
  ○ date-time
  ○ password
  ○ byte
  ○ Binary
  ○ …
	• Beta since Kubernetes 1.25
	• New use cases:
	  ○ A field vs another: `self.minReplicas <= self.replicas`
	  ○ Two sets are disjoint: `self.set1.all(e, !(e in self.set2))`
	  ○ Immutable field, once set: `self == oldSelf`
	• No need to develop a validating webhook
	• No need to deploy and keep a validating webhook running (certs...): you could avoid a point of failure
	• Custom error messages
		• An HTTP callback that receive admission request and can admit or deny it according to custom logic
		• Two distinct failure policies (unavailable webhook):
		  ○ `Ignore`: the check is skipped on webhook issues
		  ○ `Fail`: the admission will fail: it's a point of failure
		• Ideally an admission webhook should avoid any side effect but it could potentially alter external objects

Validation: HCO validating webhook

- Internal logic first
- **Delegated validation** with each component operator
  - HCO is a single entry point
  - But we don't want to duplicate component operator specific logic on HCO
  - **Dry-run mode propagation first**
  - Only if all the components admit the configuration change, this is really propagated
  - Not really an ACID Transaction
  - But still enough to prevent inconsistent configuration acroccos different components and the need to handle rollbacks
  - Implemented with goroutines



New APIs and APIs deprecation

Compatibility is hard!!!

Please refer to "Changing the API" document from sig-architecture:

https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api_changes.md

Quick hints:

- *An API change is considered compatible if it:*
  - adds new functionality that is not required for correct behavior (e.g., does not add a new required field)
  - does not change existing semantics, including:
    - the semantic meaning of default values and behavior
    - interpretation of existing API types, fields, and values
    - which fields are required and which are not
    - mutable fields do not become immutable
    - valid values do not become invalid
    - explicitly invalid values do not become valid
- If not, it's a non-backward compatible change and it requires:
  - new **API version**
  - proper **conversion mechanism**
  - proper **deprecation mechanism**

Metrics:

Refer to the Metrics Stability Framework (KEP-1209)

TA: introducing a new optional field with a sane/opinionated default is acceptable within the same API version

Warning: the OLM currently supports conversion webhooks only if the operator is deployed in AllNamespaces installMode (a sort of cluster singleton, CRDs are cluster scoped!)

Hint: use fuzzers to randomize inputs to detect conversion glitches (round trip)

Lesson learned: scaling

- **Watches** in controller-runtime are **expensive**
- An operator that works well on a **small cluster** can become unmanageable on a **huge cluster** with many objects
- Eg. watching all the configMaps or all the secrets on a cluster can lead a huge number of events to your operator
  a. Some **reconciliation loops** can be **skipped** filtering them with **predicates** (eventually custom ones)
  b. **Controller-runtime cache** can be **optimized** limiting its scope with **selectors** (labels, namespace…)

  Warning: filtered out objects cannot be read with controller-runtime client, if they are not in the cache you will get a NotFound error even for existing objects

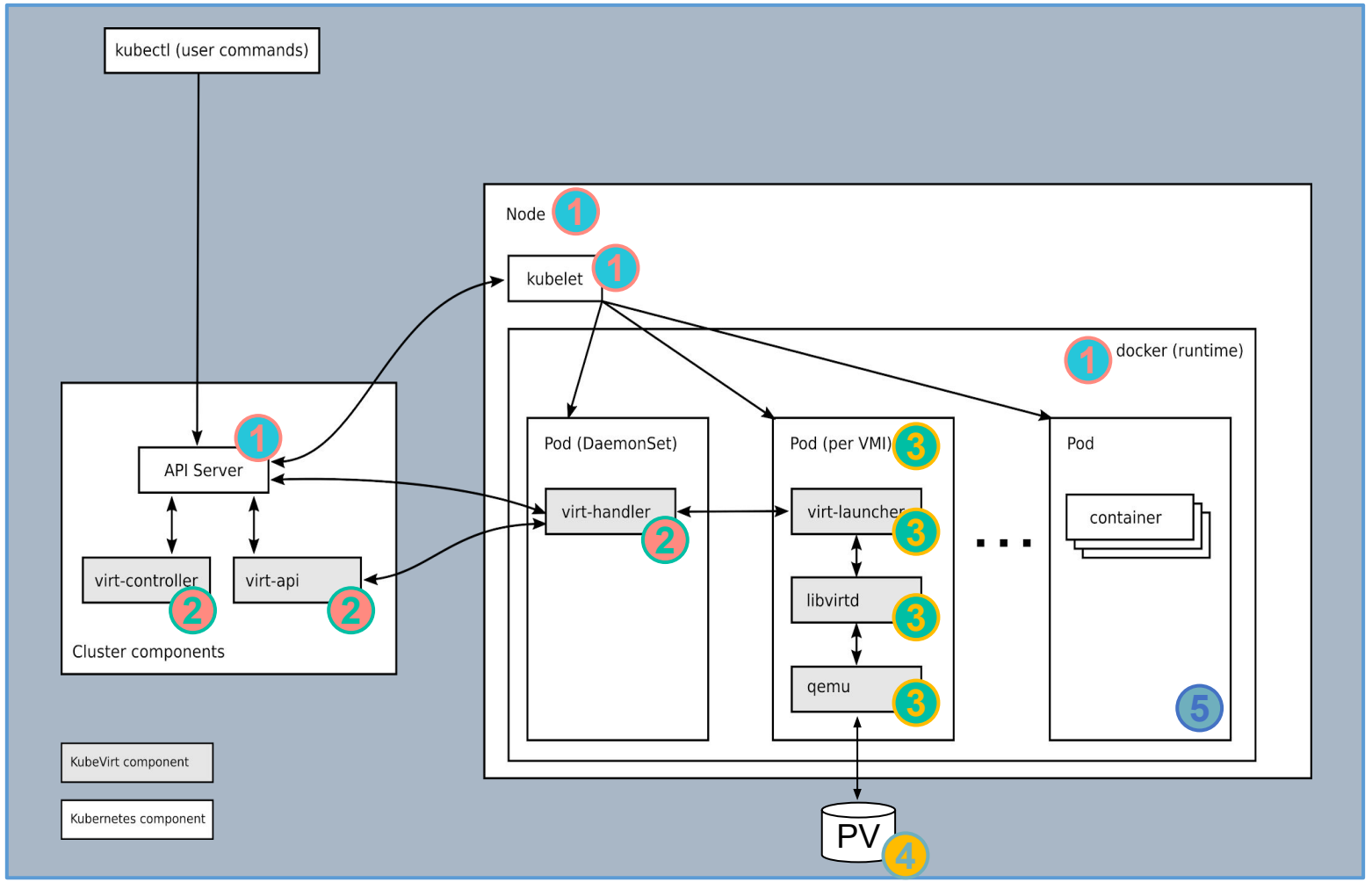
- **Limit API burden**: eg. update the status only once at the end of the reconciliation loop
- Measure your operator with metrics on real clusters!

```
// Restricts the cache's ListWatch to specific fields/labels per
// GVK at the specified object to control the memory impact
// this is used to completely overwrite the NewCache
// function so all the interesting objects should be
// explicitly listed here
func getNewManagerCache(operatorNamespace string) cache.NewCacheFunc {
  namespaceSelector := fields.Set{
     "metadata.namespace": operatorNamespace}.AsSelector()
  labelSelector := labels.Set{
     hcoutil.AppLabel: hcoutil.HyperConvergedName}.AsSelector()
  labelSelectorForNamespace := labels.Set{
     hcoutil.KubernetesMetadataName: operatorNamespace}.AsSelector()
  return cache.BuilderWithOptions(
     cache.Options{
        SelectorsByObject: cache.SelectorsByObject{
           ...
           &schedulingv1.PriorityClass{}: {
              Label: labels.SelectorFromSet(
                 labels.Set{
                    hcoutil.AppLabel: hcoutil.HyperConvergedName}),
           },
           &corev1.ConfigMap{}: {
              Label: labelSelector,
           },
           &corev1.Service{}: {
              Field: namespaceSelector,
           },
           &corev1.Endpoints{}: {
              Field: namespaceSelector,
           },
           ...
```

Upgrades in HCO/KubeVirt

Different topics:

1. Platform/node OS upgrades
2. HCO/KubeVirt control plane upgrades
3. KubeVirt workload upgrades
4. VMs Guest OS upgrades
5. Others (eg. golden images)



1. Platform/node OS upgrades

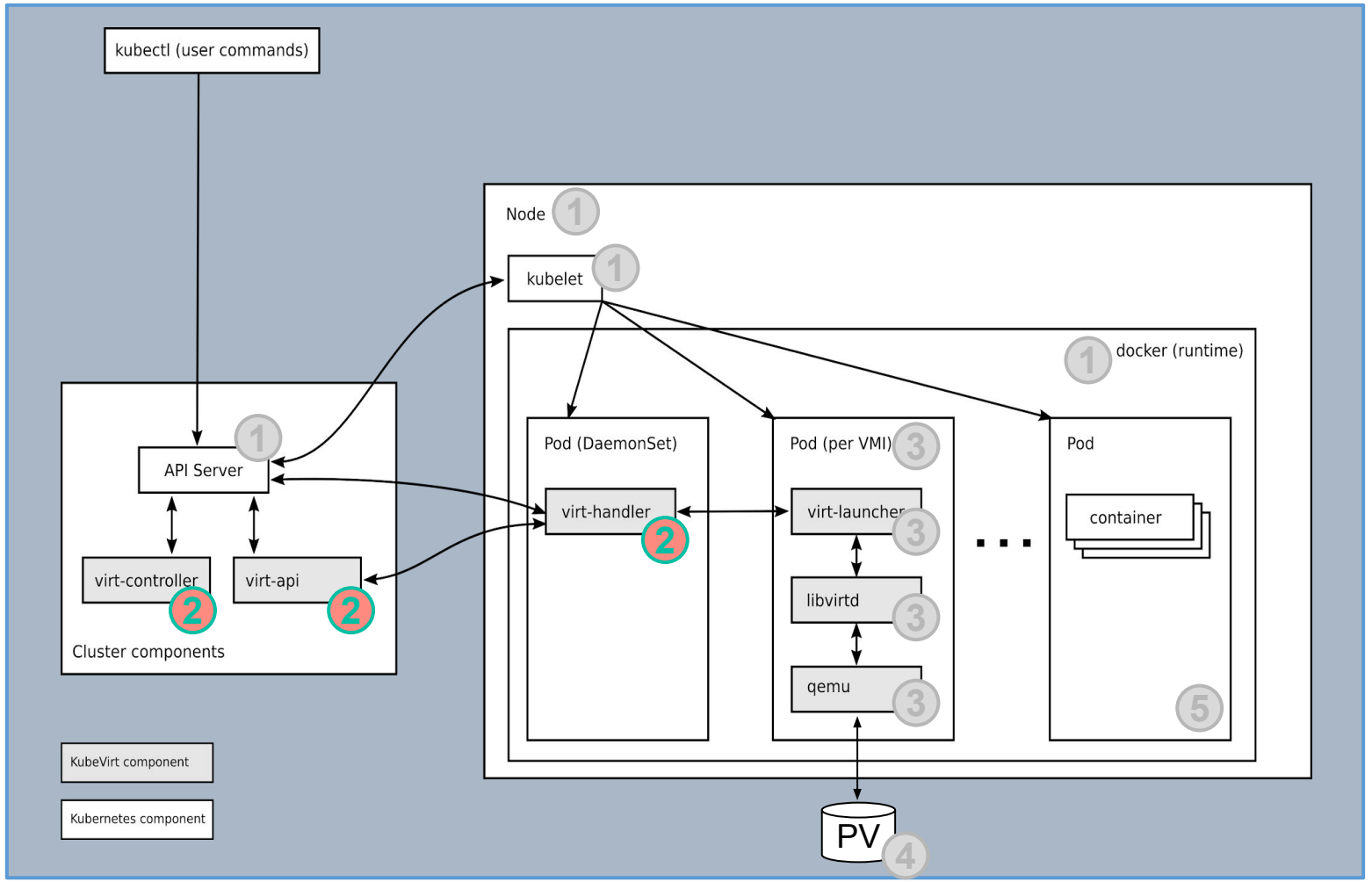
- A **platform upgrade** will (probably) require a **node drain** to **upgrade** its OS
- VMs/VMIs:
  - are **not** like **generic stateless pods** that can be quickly destroyed and recreated on a different node
  - should be **graceful shutdown** (when possible), see: `terminationGracePeriodSeconds`
    - Virt-handler is responsible for both signaling the virtual machine to shutdown and ensuring the virtual machine is forced off
    - Virt-launcher intercepts signals (such as SIGTERM) sent to it by the kubernetes runtime and notifies virt-handler to begin the graceful shutdown process
  - VMs can be **live-migrated** (under certain conditions, from a pod to another pod on another node): see `evictionStrategy`
- **VMIs** are **protected** by a **PodDisruptionBudget** with `minAvailable: 1`
- During live-migration the **migration process** is **protected** by a **second PDB** with `minAvailable: 2` (protecting **source** and **target pods**)
- A VM that could not be successfully migrated nor terminated could **block** a **node drain** and require a **cluster admin intervention** (an alert will be raised)
- The number of `parallelMigrationsPerCluster` and `parallelOutboundMigrationsPerNode` can be tuned from HCO
- As operator authors, we can potentially set an annotation (`olm.maxOpenShiftVersion`) on the bundle to eventually prevent the CVO to upgrade OCP/OKD to the next platform version if not compatible with our current release. An alert will be raised

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
 ...
 name: rhel9-wu0yeqpj1cywx0p6
 namespace: simone
spec:
 ...
 running: true
 template:
   ...
   spec:
     evictionStrategy: LiveMigrate
     ...
     terminationGracePeriodSeconds : 180
     volumes:
       - dataVolume:
           name: rhel9-wu0yeqpj1cywx0p6
         name: rootdisk
     ...
status:
 conditions:
   - lastProbeTime: null
     lastTransitionTime: null
     message: |-
       cannot migrate VMI: PVC
       rhel9-wu0yeqpj1cywx0p6 is not shared,
       live migration requires that all PVCs
       must be shared (using ReadWriteMany access
mode)
     reason: DisksNotLiveMigratable
     status: "False"
     type: LiveMigratable
```

Upgrades in HCO/KubeVirt

Different topics:

1. Platform/node OS upgrades
2. HCO/KubeVirt control plane upgrades
3. KubeVirt workload upgrades
4. VMs Guest OS upgrades
5. Others (eg. golden images)



2. HCO/KubeVirt control plane upgrades 1/4

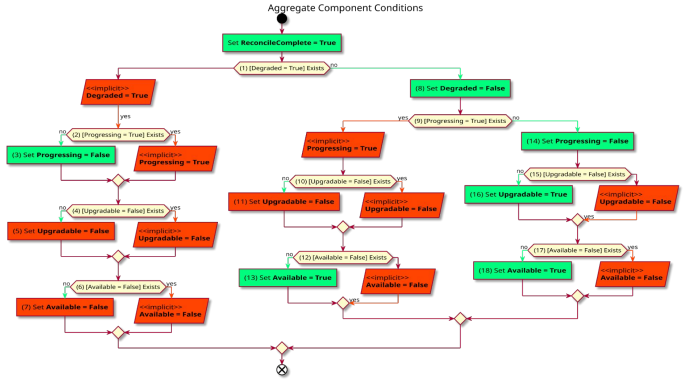
KubeCon CloudNativeCon Europe 2023

Upgrade detection logic:

- **HCO** and its **sibling operators** (Kubevirt, Containerized Data Importer, Cluster Network Addons Operator …) are **shipped together** in an opinionated and validated bundle
- CRDs are update by the OLM before operator deployment, no rollbacks
- HCO is **aware**, from the bundle configuration, of its **expected version** and the **versions** of its **sibling operators**:

```
- name: KUBEVIRT_VERSION
 value: v0.59.0
- name: CDI_VERSION
 value: v1.56.0
- name: NETWORK_ADDONS_VERSION
 value: v0.85.0
- name: SSP_VERSION
 value: v0.17.0
- name: TTO_VERSION
 value: v0.5.0
 ...
```

- **Each component operator** is **reporting** its **observed version** on the status on its CR
- Aggregating **Available**, **Progressing** and **Degraded conditions** from its sibling operators and observing their `status.observedVersion` against its expected value, HCO can monitor and the upgrade process




```
status:
 conditions:
   - lastHeartbeatTime: "2023-04-11T09:37:31Z"
     lastTransitionTime:
"2023-04-11T09:37:31Z"
     message: Deployment Completed
     reason: DeployCompleted
     status: "True"
     type: Available
   - lastHeartbeatTime: "2023-04-11T09:37:31Z"
     lastTransitionTime:
"2023-04-11T09:37:31Z"
     status: "False"
     type: Progressing
   - lastHeartbeatTime: "2023-04-11T16:16:53Z"
     lastTransitionTime:
"2023-04-11T16:16:53Z"
     status: "False"
     type: Degraded
 observedVersion: v1.56.0
 operatorVersion: v1.56.0
 phase: Deployed
 targetVersion: v1.56.0
```

2. HCO/KubeVirt control plane upgrades 1/4

Canary deployments (reliability and resiliency vs upgrade speed)

- From OpenShift Conventions (https://github.com/openshift/enhancements/blob/master/CONVENTIONS.md):
  All daemonsets of OpenShift components, especially those which are not limited to control-plane nodes, **should use the `maxUnavailable` rollout strategy** to **avoid slow updates over large numbers of compute nodes**.
  a. Use 33% `maxUnavailable` if you are a workload that has no impact on other workload. This ensure that if there is a bug in the newly rolled out workload, 2/3 of instances remain working. Workloads in this category include the spot instance termination signal observer which listens for when the cloud signals a node that it will be shutdown in 30s. At worst, only 1/3 of machines would be impacted by a bug and at best the new code would roll out that much faster in very large spot instance machine sets.
  b. Use 10% `maxUnavailable` in all other cases, most especially if you have ANY impact on user workloads. This limits the additional load placed on the cluster to a more reasonable degree during an upgrade as new pods start and then establish connections.
- For **virt-handler** we are even **stricter**:
  safely upgrade one virt-handler (one node) and then proceed with rest in batches, making the upgrade procedure faster (but still safe):
  a. update virt-handler damonset with `maxUnavailable` =1
  b. patch daemonSet with new version
  c. wait for a new virt-handler to be ready
  d. set `maxUnavailable` =10%
  e. start the rollout of the new virt-handler again
  f. wait for all nodes to complete the rollout
  g. set `maxUnavailable` back to 1

2. HCO/KubeVirt control plane upgrades 3/4

Be declarative if/when possible

- An **operator** is the **bridge** between a declarative world (its APIs) and what should be performed (imperative world)
- An operator is naturally imperative, but…
- Some operations (patching a set of fields, deleting other object) especially in the upgrade scenario can be describe with existing formalisms
- Why?
  - **Better readability** of the changes
  - **Better control** over the actual execution of the changes to the world
  - **Less code**
  - Easily coverable with **unit tests**
  - We can have **different configurations** for **different release lanes**
  - …
- We use **lists of changes** (jsonPatch) to the CR for HCO
- ... matching a semverRange of interested source versions
- All the **required changes** are **identified first** and then **evaluated** in **dry-run mode** and then **eventually applied to proceed in the upgrade process**
- With the same logic we keep a list of objects to be eventually deleted
- The list of allowed actions is still managed with the RBAC configuration for the Service Account used by the operator: please avoid wildcards there!!!

```
{
 "hcoCRPatchList": [
   {
     "semverRange": ">=1.4.0 <=1.5.0",
     "jsonPatch": [
       {
         "op": "replace",
         "path": "/spec/featureGates/sriovLiveMigration",
         "value": true
       }
     ]
   },
   {
     "semverRange": ">=1.4.0 <1.5.0",
     "jsonPatch": [
       {
         "op": "test",
         "path": "/spec/liveMigrationConfig/bandwidthPerMigration",
         "value": "64Mi"
       },
       {
         "op": "remove",
         "path": "/spec/liveMigrationConfig/bandwidthPerMigration"
...
 "objectsToBeRemoved": [
   {
     "semverRange": "<=1.6.0",
     "groupVersionKind": {
       "group": "",
       "version": "v1",
       "kind": "ConfigMap"
     },
     "objectKey": {
       "name": "v2v-vmware",
       "namespace": "kubevirt-hyperconverged"
...
```

2. HCO/KubeVirt control plane upgrades 4/4

Communicating Operator Conditions to OLM

In the past **Operator-Lifecycle-Manager (OLM)** was **inferring** the **state** of an operator from the state of Kubernetes resources that define the operator (the **ready status** of operator pods …): generally working but we were abusing the `ready` status for something else (ambiguity)

Now HCO uses the **operator condition** (new CustomResourceDefinition) to explicitly report back to the OLM `upgradeable=false`, this will ensure that the OLM will not allow another overlapping upgrade till the previous one is marked as completed or failed. Useful for long lasting upgrades.

(unsafe) fail-forward upgrades

In the past with Operator-Lifecycle-Manager (OLM) there was no easy option to recover from an upgrade declared as Failed.
Now the cluster admin as an option to forcefully enable an upgrade to the next version even if an intermediate step is declared as failed.

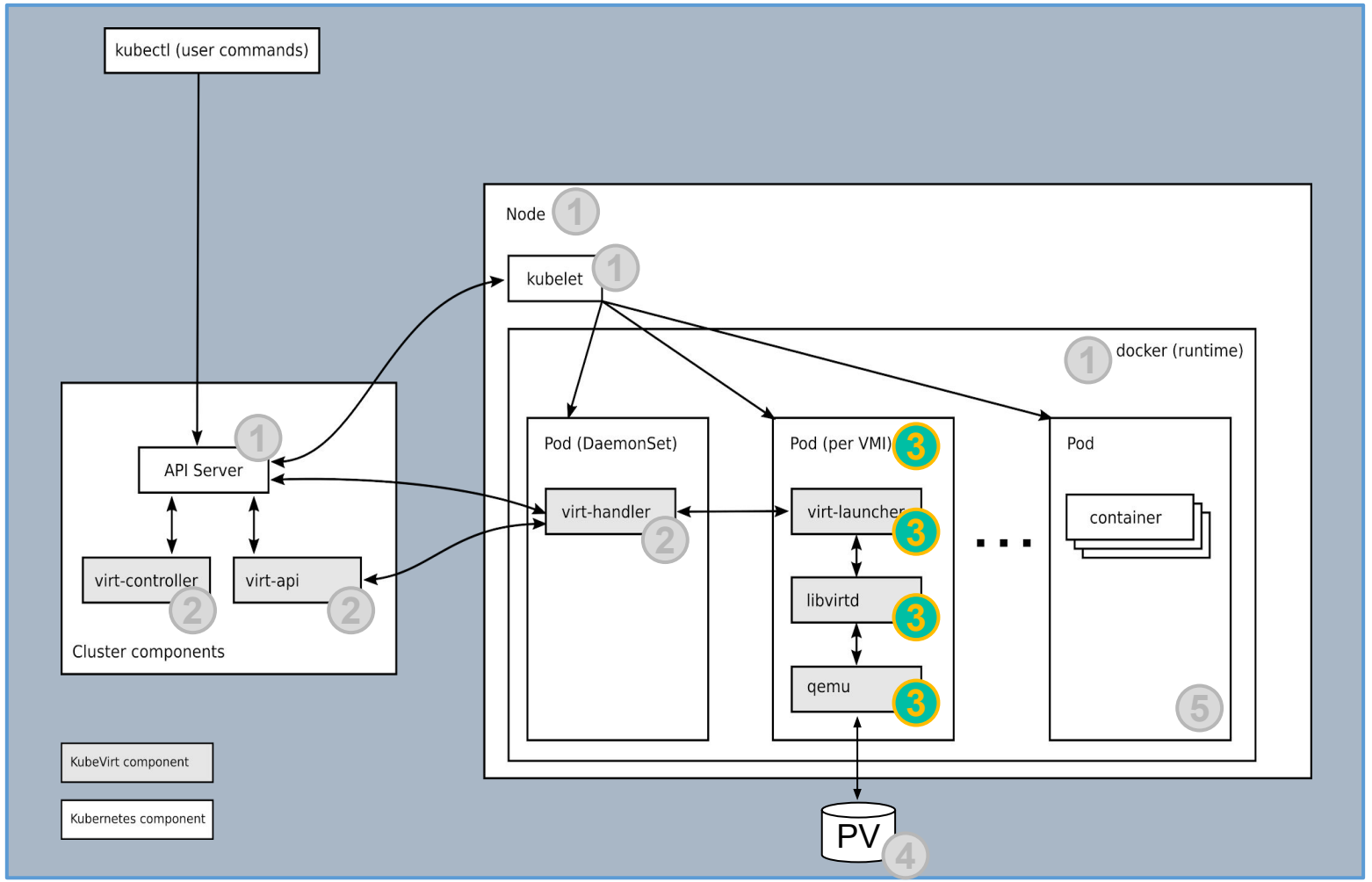
Warning: UnsafeFailForward upgrades are ultimately unsafe and should only be used in specific circumstances

```yaml
apiVersion: operators.coreos.com/v2
kind: OperatorCondition
metadata:
 name: kubevirt-hyperconverged-operator.v4.12.2
 namespace: openshift-cnv
 ownerReferences:
   - apiVersion: operators.coreos.com/v1alpha1
     blockOwnerDeletion: false
     controller: true
     kind: ClusterServiceVersion
     name: kubevirt-hyperconverged-operator.v4.12.2
spec:
 conditions:
   - lastTransitionTime: "2023-04-12T21:00:19Z"
     message: ""
     reason: Upgradeable
     status: "True"
     type: Upgradeable
 deployments:
   - hco-operator
   - hco-webhook
   - hyperconverged-cluster-cli-download
   ...
 serviceAccounts:
   - hyperconverged-cluster-operator
   - cluster-network-addons-operator
   - kubevirt-operator
   ...
status:
 conditions:
   - lastTransitionTime: "2023-04-12T21:00:19Z"
     message: ""
     observedGeneration: 7
     reason: Upgradeable
     status: "True"
     type: Upgradeable
```

Upgrades in HCO/KubeVirt

Different topics:

1. Platform/node OS upgrades
2. HCO/KubeVirt control plane upgrades
3. KubeVirt workload upgrades
4. VMs Guest OS upgrades
5. Others (eg. golden images)



3. KubeVirt workload upgrades

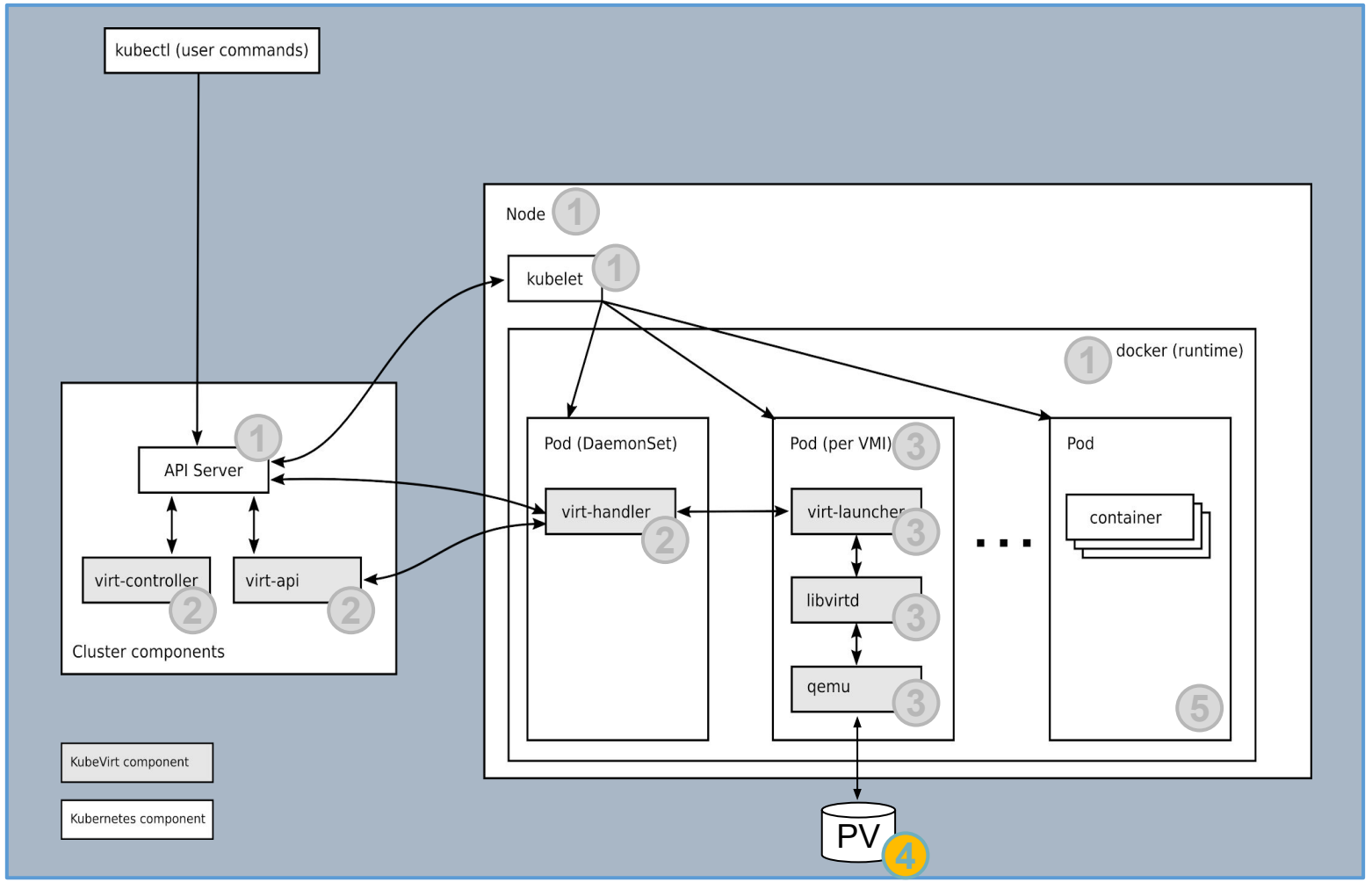
- **Virt-launcher image** is used by each individual Virtual Machine Instance **(VMI) pod**
- It contains:
  - libvirt
  - Qemu
  - Other utilities …
- We want to keep them up to date!
- HCO exposes an API to let the cluster admin tune it
- Two possible **strategies**:
  - **LiveMigrate** (less disruptive)
  - **Evict**
- It works in **batches**
- **After** the control plane upgrade is completed, and **async**!
- **un-upgradeable VMIs** (not live-migrateable and not supposed to be evicted) are **tagged** with an annotation and an alert is raised
- It doesn't affect VMs guest OS

```
type HyperConvergedWorkloadUpdateStrategy  struct {
  // WorkloadUpdateMethods  defines the methods that can be used to
  // disrupt workloads during automated workload updates.
  // When multiple methods are present, the least disruptive method takes
  // precedence over more disruptive methods.
  // For example if both LiveMigrate and Evict
  // methods are listed, only VMs which are not
  // live migratable will be restarted/shutdown.
  // An empty list defaults to no automated workload updating.
  //
  // +listType=atomic
  // +kubebuilder:default={"LiveMigrate"}
  // +default=["LiveMigrate"]
  WorkloadUpdateMethods []string `json:"workloadUpdateMethods"`

  // BatchEvictionSize  Represents the number of VMIs that can be
  // forced updated per the BatchShutdownInterval interval
  //
  // +kubebuilder:default=10
  // +default=10
  // +optional
  BatchEvictionSize *int `json:"batchEvictionSize,omitempty"`

  // BatchEvictionInterval  Represents the interval to wait
  // before issuing the next batch of shutdowns
  //
  // +kubebuilder:default="1m0s"
  // +default="1m0s"
  // +optional
  BatchEvictionInterval *metav1.Duration
`json:"batchEvictionInterval,omitempty"`
}
```

Upgrades in HCO/KubeVirt

Different topics:

1. Platform/node OS upgrades
2. HCO/KubeVirt control plane upgrades
3. KubeVirt workload upgrades
4. VMs Guest OS upgrades
5. Others (eg. golden images)



3. VMs Guest OS upgrades

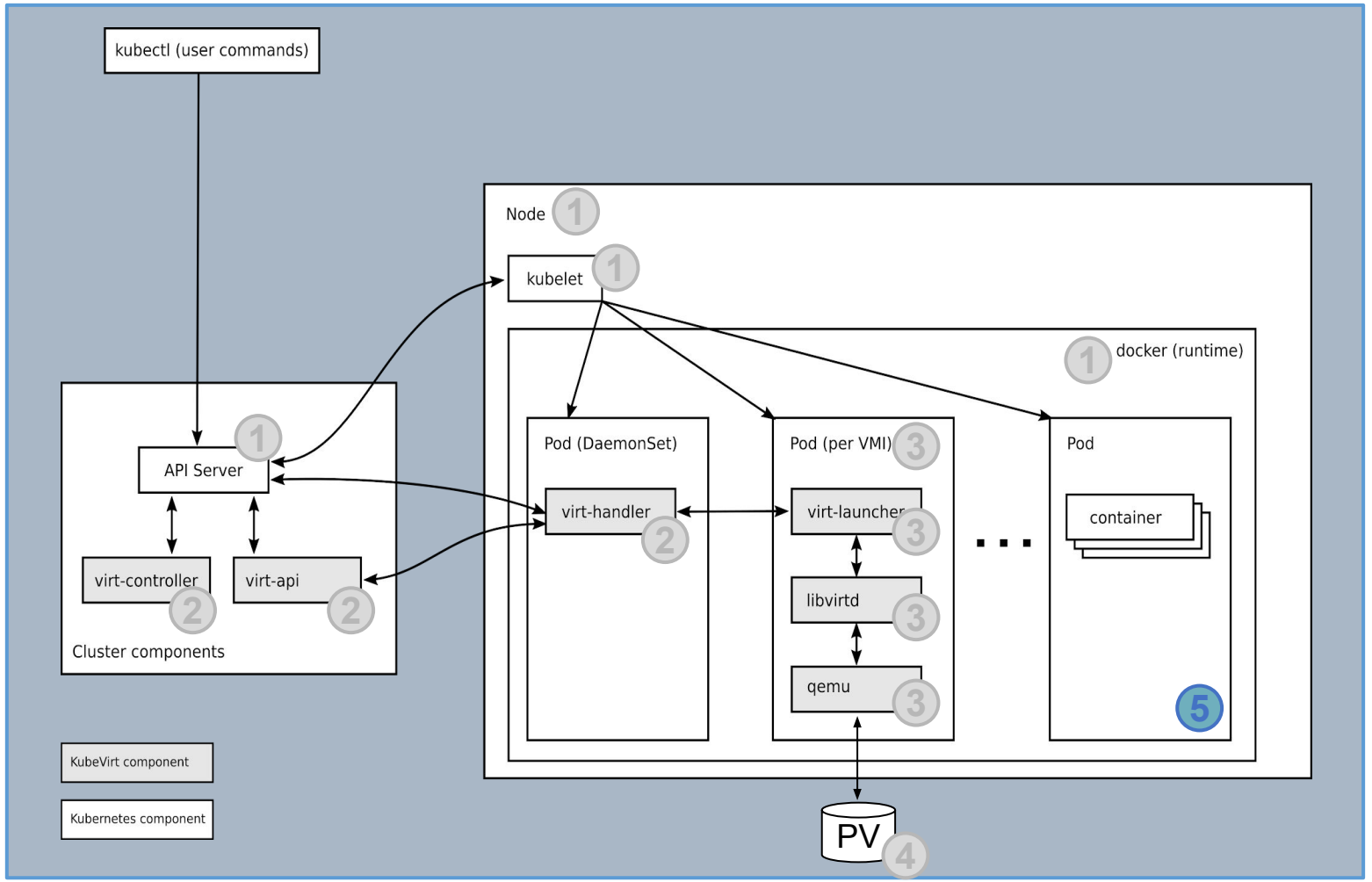
- Guest OSs are not strictly part of / or managed by HCO/Kubevirt
- **Many external tools** can be used to manage them
- … but we also have a **cloud native integration**!

- **Tekton**, AKA OpenShift **Pipelines**, is a cloud-native, continuous integration and continuous delivery (CI/CD) solution for Kubernetes
- KubeVirt Tekton tasks provides Kubevirt specific Tekton tasks (building blocks), which focus on:
  - Creating
  - Updating
  - Managing resources of KubeVirt (VMs, DataVolumes, DataSources, Templates…)
  - Executing commands in VMs
  - Manipulating disk images with libguestfs tools
  - …
- You can easily create Tekton Pipelines to take care of your beloved VMs using KubeVirt Tekton tasks "blocks"
- Examples available on https://github.com/kubevirt/kubevirt-tekton-tasks/tree/main/examples/pipelines

Upgrades in HCO/KubeVirt

Different topics:

1. Platform/node OS upgrades
2. HCO/KubeVirt control plane upgrades
3. KubeVirt workload upgrades
4. VMs Guest OS upgrades
5. Others (eg. golden images)



4. Others: golden images

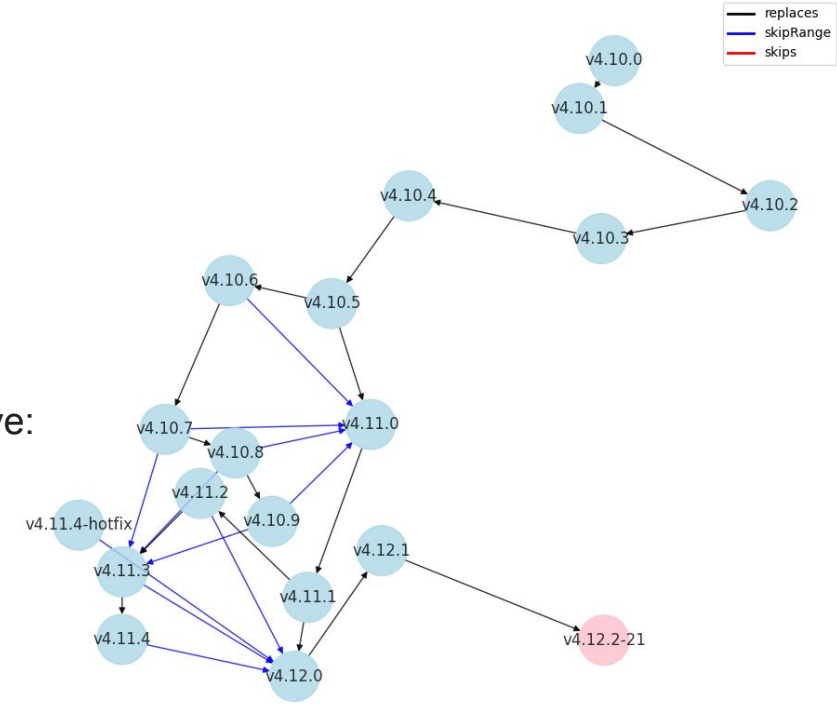
- Hyperscalers (like AWS, GCP, Azure, IBM Cloud…) provide:
  - root disk images for commonly used operating systems (**golden images**)
  - **continuous updates** of those images
- We want to offer the same feature to our Kubevirt users -> `dataImportCron`
- The `dataImportCron` is ensuring that whenever a new version of our cloud golden image is available (on a cloud registry) it will be updated also in our cluster
- **Popular images** (Fedora, Centos-Stream, RHEL, …) are available **out of the box** in the **opinionated** deployment, **custom** one can be added
- `retentionPolicy` and `garbageCollect` can be tuned by cluster admin, an older image could be preferred over the last one
- Notes:
  - Not a pipeline for updating disks attached to existing VMs
  - The **golden image update process** is completely **independent** from HCO/Kubevirt **releases**

```yaml
apiVersion: hco.kubevirt.io/v1beta1
kind: HyperConverged
metadata:
 name: kubevirt-hyperconverged
spec:
 dataImportCronTemplates:
   - metadata:
       name: custom-image1
     spec:
       schedule: "0 */12 * * *"
       template:
         spec:
           source:
             registry:
               url: docker://myprivateregistry/custom1
       managedDataSource: custom1
       retentionPolicy: "All"
   - metadata:
       name: centos-stream9
     spec:
       schedule: "5 */12 * * *"
       template:
         spec:
           source:
             registry:
               url: docker://quay.io/containerdisks/centos-stream:9
       managedDataSource: centos-stream9
       retentionPolicy: "None"
       garbageCollect: Outdated
```

Test! Test! Test!

Test! Test! Test!

- We cannot really prevent bugs, but we can always test more (**automated testing is the key**!)
- If we'd expect our user to confidently enable automatic upgrades,
  we should offer only properly validate paths
- To keep it manage, let's keep the **upgrade matrix small**
  - **Less upgrade edges**
  - **Longer upgrade paths**… but we'd expect our user to confidently upgrade more often!
- OLM verbs to define the upgrade graph (Replaces/Skips/SkipRange) are not always so intuitive:
  Let's plot the upgrade graph
  - Initially we implement our own tool
  - Now the `opm` tool (*Operator Package Manager*) can do the same (`opm alpha render-graph`)



And observe!

- Instrument your operator with relevant metrics
- And collect remote traces with OpenTelemetry (or similar)
  - You can track which versions your users are using
  - How often they are upgrading
  - How long does it take (and much more…)
  - You can discover issue that you cannot forecast on your development clusters
  - -> **observability-integrated/driven development**
- You can design canary release strategies releasing (slowly rolling out) just to select users first
  on different channels…

reference:
https://sdk.operatorframework.io/docs/best-practices/observability-best-practices/

KubeCon | CloudNativeCon
Europe 2023

Please scan the QR Code above
to leave feedback on this session