KubeCon | CloudNativeCon

North America 2019

KubeCon | CloudNativeCon

North America 2019

Cloud Provider Deep Dive

Yassine Tijani
@yastij

Walter Fender
@cheftako

SIG Cloud Provider Deep Dive

Member of Technical Staff - VMware
yastij - GitHub
Sig Cloud Provider - contributor
Sig Cluster lifecycle - (kubeadm, CAPV)
Cloud Provider Extraction

Software Engineer - Google
Cheftako - GitHub
Sig Cloud Provider - Chair
Sig API Machinery - Contributor
Cloud Provider Extraction - Lead

SIG Cloud Provider Deep Dive

What is the CPI (Cloud Provider Interface) ?

- CPI is an interface that allows cloud providers to natively integrate with Kubernetes
- It’s implemented through a binary/container that is called Cloud-Controller-Manager (CCM)
- What does the interface provide:
    - Instance lifecycle
    - Load balancing
    - Routing Rules
    - Zones
    - ...

- Can be found here: https://github.com/kubernetes/cloud-provider/blob/master/cloud.go

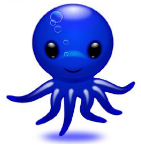
SIG Cloud Provider Deep Dive

How the community pushes changes to CPI ?

- The community extends the cloud interface
  - Modifications are done under staging/ `k8s.io/cloud-provider => ./staging/src/k8s.io/cloud-provider`
  - After the extraction, no provider code will be allowed in k/k

- Cut a kubernetes release that relies on the newly extended interface
- The CCM can choose to implement or return an error
- Ensure you are testing against the kubernetes supported versions

SIG Cloud Provider Deep Dive



Would a Cloud Provider ever want to run non standard controllers? Could that even work?

https://github.com/kubernetes/kubernetes/blob/master/pkg/controller/nodeipam/legacyprovider.go#L36

“`cloud cloudprovider.Interface`”

Google controls IPAM settings through the cloud provider interface.

https://github.com/kubernetes/kubernetes/blob/master/cmd/cloud-controller-manager/app/controllermanager.go#L279

```
“func newControllerInitializers() map[string]initFunc {
        controllers := map[string]initFunc{}
        controllers["cloud-node"] = startCloudNodeController”
```

Good start but need a better way to extend this.

SIG Cloud Provider Deep Dive

KubeCon | CloudNativeCon North America 2019

What about the default controllers running in the KCM?

https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/

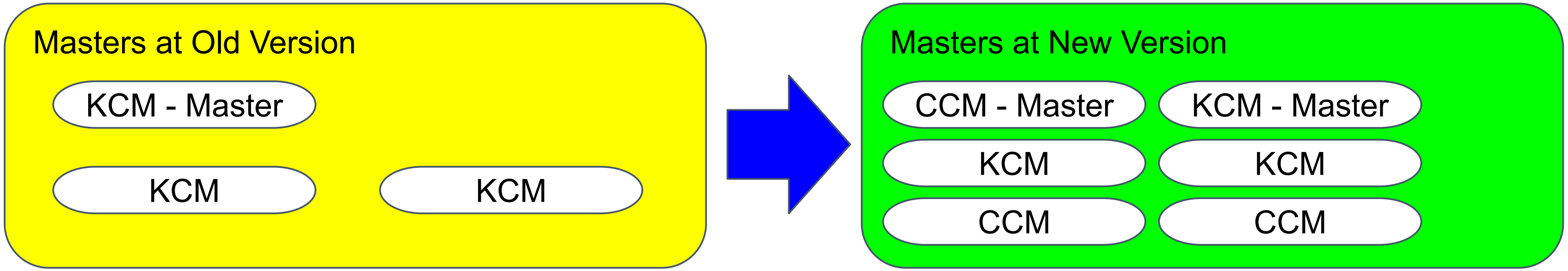
“--controllers stringSlice   Default: [*]
A list of controllers to enable. '*' enables all on-by-default controllers, 'foo' enables the controller named 'foo', '-foo' disables the controller named 'foo'.
All controllers: attachdetach, bootstrapsigner, cloud-node-lifecycle, clusterrole-aggregation, cronjob, csrapproving, csrcleaner, csrsigning, daemonset, deployment, disruption, endpoint, endpointslice, garbagecollector, horizontalpodautoscaling, job, namespace, **nodeipam**, nodelifecycle, persistentvolume-binder, persistentvolume-expander, podgc, pv-protection, pvc-protection, replicaset, replicationcontroller, resourcequota, root-ca-cert-publisher, route, service, serviceaccount, serviceaccount-token, statefulset, tokencleaner, ttl, ttl-after-finished
Disabled-by-default controllers: bootstrapsigner, endpointslice, tokencleaner”

```
/home/kubernetes/bin/kube-controller-manager --controllers=-nodeipam
```

SIG Cloud Provider Deep Dive

CCM migration - Harder than it looks

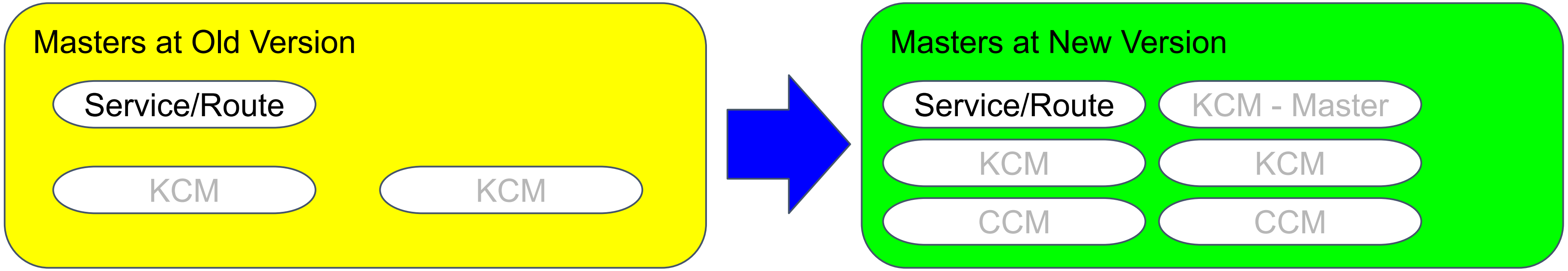
Let’s consider a HA Cluster with 3 Masters running KCM which I want to upgrade to 3 masters running both KCM and CCM.
Remember in the KCM only scenario I run the Route and Service controllers on the KCM.
On the KCM + CCM masters the Route and Service controllers run on the CCM.
We don’t consider it safe to run more than 1 instance of a controller in a cluster.
In the KCM only scenario we achieve this by using leader election to give us 1 master KCM and 2 hot standby.
In the KCM + CCM scenario they each use leader election to get 1 master and should not overlap on controllers.
What about **during** the upgrade? We could just take downtime, but that seems contrary to High Availability.

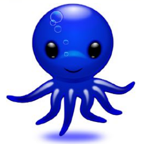


CCM migration - Harder than it looks

Let’s walk through the standard upgrade steps without upgrade/downgrade protection.

- 3 masters at version at the old version. The KCM master, running Service and Route Controllers.
- Take down an old master and bring up a new master at the new version.
- KCM master is still at the old version. We now have a CCM master at the new version. Both are running Service and Route Controllers.
- Take down another master at the old version. KCM master is still at the old version.
- Take down the last master at the old version. KCM master comes up at the new version not running the Service and Route Controllers.





CCM migration - Harder than it looks

Who needs to care? Anyone with HA clusters who wants to upgrade from “KCM only” to “KCM + CCM”.

★ Primarily “In-Tree Cloud Providers”

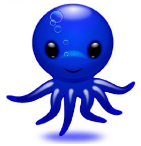
https://github.com/kubernetes/enhancements/blob/master/keps/sig-cloud-provider/20190422-cloud-controller-manager-migration.md
https://github.com/kubernetes/kubernetes/pull/84259

Essential idea is to have a migration lock.

- The lock defines which controllers it runs across (Service and Route)
- It defines which Manager runs the controllers at which release. (KCM at 1.20, CCM at 1.21)

While holding the Master Lock the Manager gets the migration lock. Only then starting the migrating controllers.
Requires you upgrade/downgrade through all minor releases. Already a requirement.
All patch releases of a minor version must have the same configuration.
Allows for both upgrade and downgrade.
Each migration should use a different migration lock name. (Eg 1_20_to_1_21 vs 1_21_to_1_22)

SIG Cloud Provider Deep Dive



Removing Cloud Provider from Persistent Volumes

Container Storage Interface is an initiative being pushed by SIG Storage (Thank You! We still have to care though)

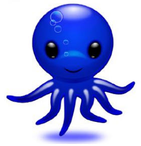
Most volume types are an additional driver which needs to run on each Node that might need that volume types
We also need a controller for each volume type, where does that run? How do we control versions?

When the Cloud Providers have been extracted; we will still have cloud provider volume type shims to CSI
New volume types are going to be added over time and some will be cloud provider specific
These need to be folded into both OSS, managed and test deployment systems

Future features such as snapshot?

https://github.com/kubernetes/community/blob/master/contributors/design-proposals/storage/csi-migration.md

CSI Driver Developer Guide for Migration



Power of Shims or What about CP volume types?

How do we migrate from the existing in tree cloud provider volume types to the CSI system?
Delete the volume types and recreate as CSI drivers of the correct type? A little too final, no good rollback!

Add a flag to allow the CP volume type to go to either the CP code in process or via the matching CSI driver.
Flag starts defaulted to false and is used for testing.
Flag is flipped to default to true and is used to roll back if there is a problem.
Once everyone has the shim pointing at the CSI driver; the flag and in process CP code can be removed.

Eventually we could even remove the shim and just have the CSI driver

Provider-specific Networking

Integrating the Kubernetes networking with my provider, why should I even care ?

- Native IP Address Management (IPAM)
- Native Routing: integrate packet routing natively with your networking infrastructure

The cluster networking has to satisfy the following rules:

- pods on a node can communicate with all pods on all nodes without NAT
- agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node
- pods in the host network of a node can communicate with all pods on all nodes without NAT

SIG Cloud Provider Deep Dive

KubeCon | CloudNativeCon North America 2019

Native IPAM, how to ?

Native Pod CIDRs (optional)

- Controller-based
  - fills the `node.Spec.PodCIDR` field
  - Reconciles multiple infrastructure resources (e.g. Subnets)
  - Can manage configuration declaratively
  - Have centralized Pod CIDR management
  - NOTE: you should disable the node ipam controller as show in the previous slides

Native Pod IPs

- DaemonSet-based
  - doesn’t need to manage a complex configuration
  - Uses node-local resources (e.g. using `node.Spec.PodCIDR` or secondary network interfaces if not)
  - Can use the device plugin extension to leverage resource-based scheduling
  - Decentralized Pod IPAM

SIG Cloud Provider Deep Dive

Native Routing, how to ?

- Variant 1: Relying on native Pod IPs
    - native Pod IPs obtained through secondary Network Interfaces of the node or a native Pod CIDR
    - Native IPs makes pods routable so nothing is required

SIG Cloud Provider Deep Dive

Other routing alternatives

- Variant 2 : Provider’s route tables
  - CNI typically kubenet (plumbing to route packets using network namespaces, veth, ARP, routes/default gateways..)
  - CPI **(typically this is the Route interface that you implemented)**
  - Route controller
  - Kube-controller-manager’s `--configure-cloud-routes` flag
  - Convenient for users that do not need native Pod IPs

- Variant 3: Another variant is to in-line routes on the nodes
  - Nodes on the same subnet
  - Disable configuring cloud routes by the route controller
  - Your DaemonSet handles routes management at the node level

KMS Provider for encryption at rest

- A Key Management System (KMS) Is a service that allows to manage all your keys in a central place
- Kubernetes relies on envelope encryption using Key Encryption Keys (KEK) and Data Encryption Keys (DEKs)
- Provider can leverage their KMS to encrypt data at rest

```
// KeyManagementServiceServer is the server API for KeyManagementService service.
type KeyManagementServiceServer interface {
	// Version returns the runtime name and runtime version of the KMS provider.
	Version(context.Context, *VersionRequest) (*VersionResponse, error)
	// Execute decryption operation in KMS provider.
	Decrypt(context.Context, *DecryptRequest) (*DecryptResponse, error)
	// Execute encryption operation in KMS provider.
	Encrypt(context.Context, *EncryptRequest) (*EncryptResponse, error)
}
```

- NOTE: data is still stored on etcd (changing backend storage is hard, due to some etcd semantics being leaked through the API)

SIG Cloud Provider Deep Dive

KMS Provider for encryption at rest

- Your end-user would provide a versioned configuration file to Kubernetes

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
- secrets
- somecrd
providers:
- kms:
    name: myKmsPlugin
    endpoint: unix:///tmp/socketfile.sock
    cachesize: 100
    timeout: 3s
- identity: {}
```

SIG Cloud Provider Deep Dive

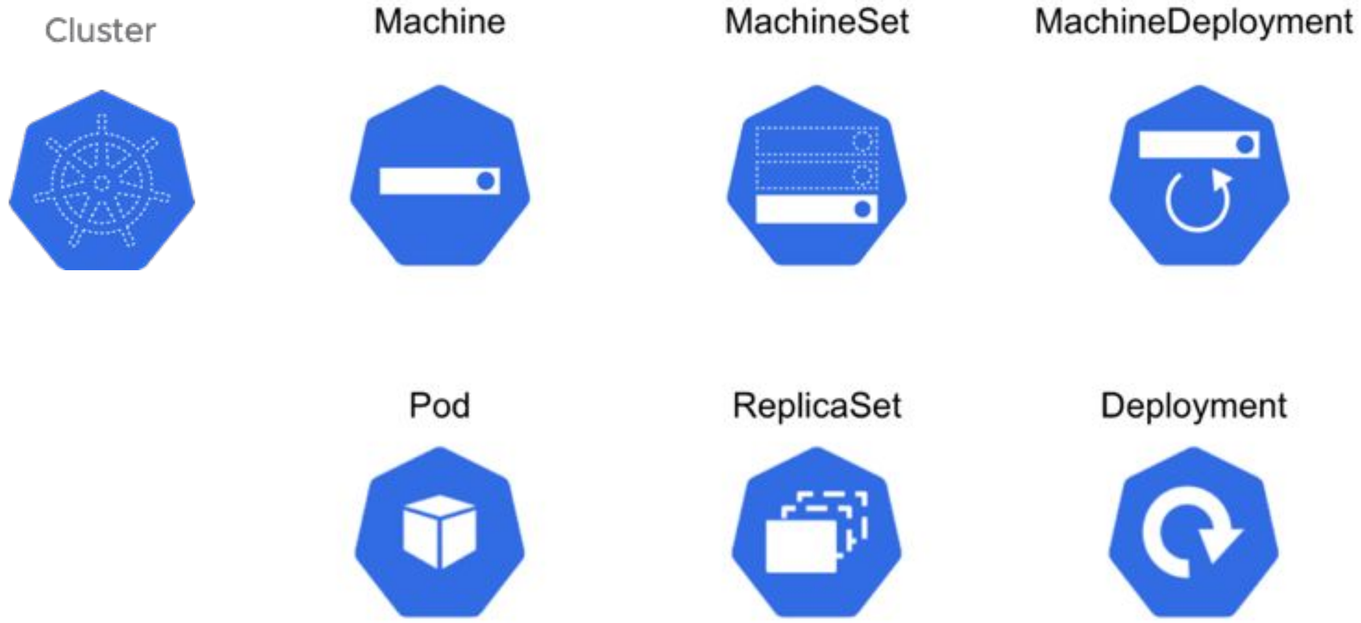
Cluster API: declarative Cluster Management

Lifecycle management of clusters is hard:

- Lots and Lots of steps to deploy fully functioning clusters
- Automated at some point, but requires a lot a wrapping around the existing tools
- No great way to handle KaaS use case
- No common UX across infrastructure providers

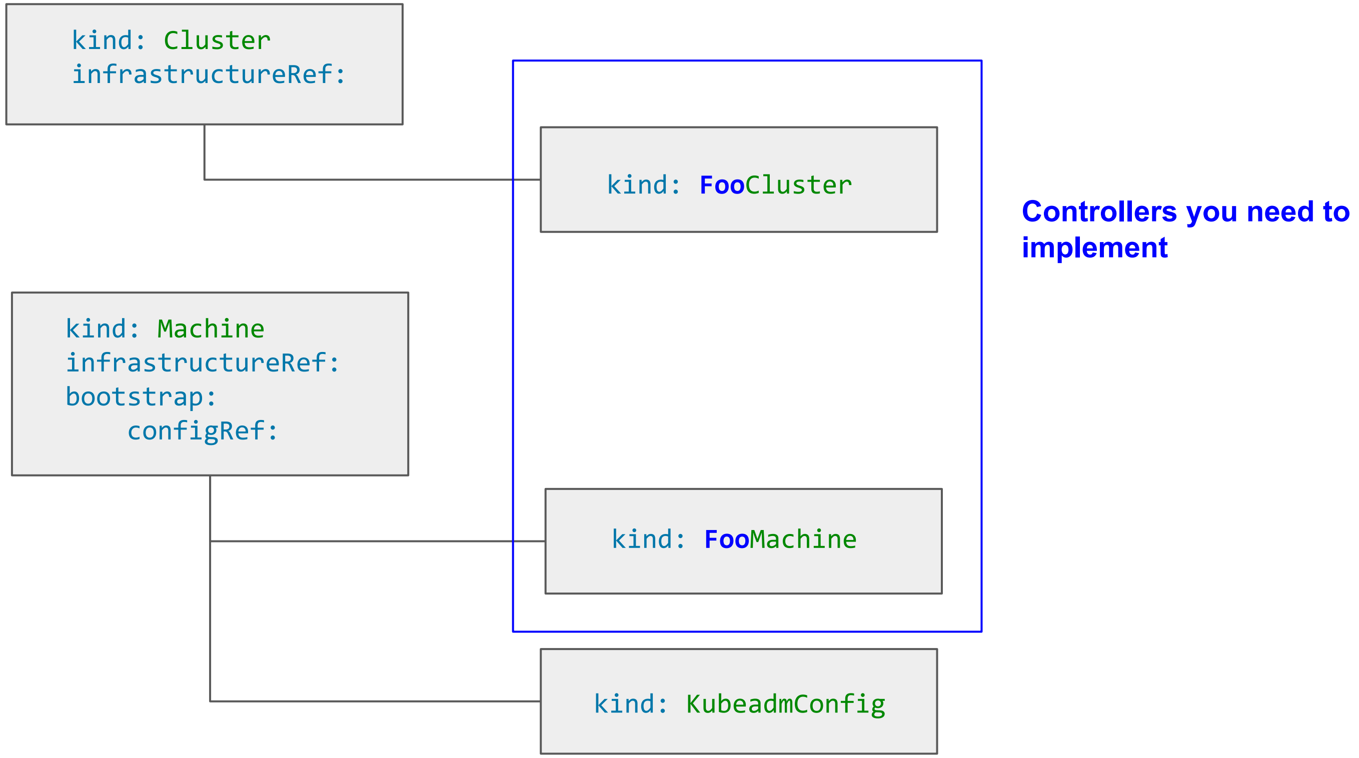
SIG Cloud Provider Deep Dive

Cluster API: What if we cloud manage infrastructure as k8s APIs ?



SIG Cloud Provider Deep Dive

Cluster API: declarative Cluster Management



SIG Cloud Provider Deep Dive

Konnectivity Server or remove the SSH Tunnel already!

Today the mechanism to communicate from master to cluster when IP routing isn't possible is SSH Tunnels. This solution forces us link cloud provider into the Kubernetes API Server. It solves a very specific problem and conflates traffic egress issues with serving APIs.

We need a more general solution. We need to get cloud provider out of Kubernetes API Server. We need to separate egress issues from serving APIs. For this we have created Konnectivity Server and Agent. We also have an egress lookup service in the Kubernetes API Server.

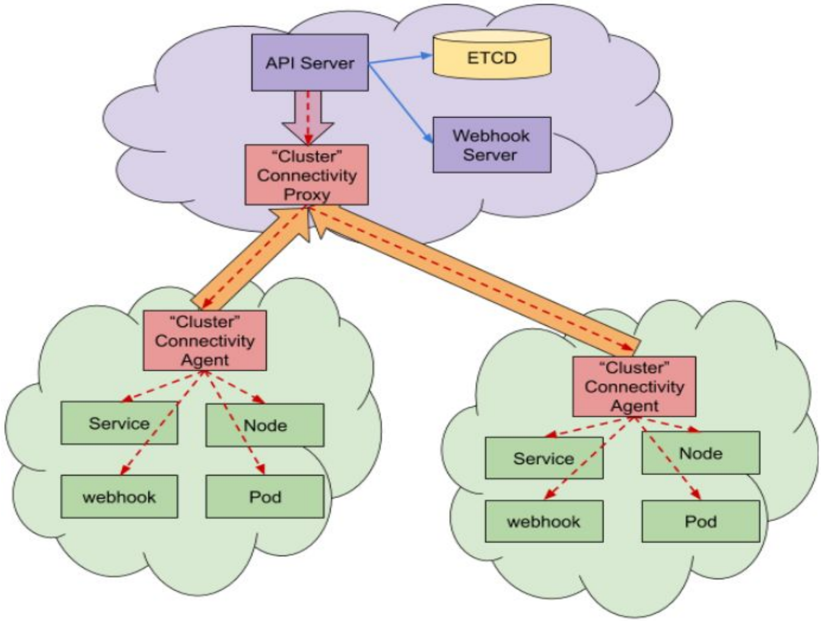
https://github.com/kubernetes-sigs/apiserver-network-proxy.

https://github.com/kubernetes/enhancements/blob/master/keps/sig-api-machinery/20190226-network-proxy.md

SIG Cloud Provider Deep Dive

Konnectivity Server and Agent

The idea is we optionally run side cars in the master and cluster. The Kubernetes API Server can be configured to egress via the the konnectivity server running on the master. It will then forward the traffic via a secure tunnel to the konnectivity agent running on the cluster. The requests will then be placed on the cluster network. Each of these components are independently swappable. So it can be extended to support things like communicating over a VPN connection.


API Server
ETCD
Webhook Server
"Cluster" Connectivity Proxy
"Cluster" Connectivity Agent
Service
Node
webhook
Pod
"Cluster" Connectivity Agent
Service
Node
webhook
Pod

SIG Cloud Provider Deep Dive

Supporting All Cloud Providers Will Remain An Ongoing Effort

What is Kubernetes?

“Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.”

For it to remain portable and extensible for all Cloud Providers we will have to keep extending and refining the systems which interface with Cloud Providers. The same is true for it to remain widely available.

The range of workloads change. The features of Cloud Providers grows. The tooling provided by Kubernetes for extensibility is accelerating. All of which means we need to stay on top of the cloud provider interoperability layer. In addition the number of Cloud Providers is increasing. Not just traditional Cloud Providers but Providers who are looking to run Kubernetes in their own Data Centers for themselves.

Supporting all Cloud Providers is an community effort which is with us for the forseeable future.

SIG Cloud Provider Deep Dive

Spare