Building a Controller Manager for your Cloud Platform

Chris Hoge, OpenStack Foundation
Fabio Rapposelli, VMware

Who are we?

- Chris Hoge
  - Strategic Program Manager, OpenStack Foundation
  - Co-lead, SIG-OpenStack
  - Co-lead, SIG-Cloud-Provider
  - PTL OpenStack Loci (lightweight container images for OpenStack
- Fabio Rapposelli
  - Staff Engineer, VMware
  - Co-lead, SIG-VMware

Part I: The State of Cloud Providers

What is this “in-tree” I keep hearing about?

As of Kubernetes 1.14, there are several in-tree cloud providers. When you download Kubernetes, you run these by default through direct configuration.

- AWS
- Azure
- Cloudstack
- GCE
- OpenStack
- OVirt
- Photon
- vSphere

Why is this a problem? (Or, where is *my* cloud provider?)

- Kubernetes should be an orchestration kernel, with drivers maintained independently by domain experts.
- For any particular deployment, there is a large body of code that is entirely irrelevant for that particular deployment scenario.
- Inclusion can imply endorsement or support for a select set of providers.
- Critical updates are tightly coupled to the Kubernetes release cycle.
- Development work has stopped on some providers, leaving code untested and unmaintained.

How do we fix cloud providers for everyone?

- To support independent work, a Cloud Controller Manager Interface was developed allow for external (“out of tree”) providers.
- Work is underway to make all cloud providers out of tree.
  - For supported in-tree providers, the provider code has been officially deprecated and their dependencies are being moved to staging.
  - With in-tree provider moved to staging after the upcoming release, they have a path to be removed in the following release. By the end of 2019, there will be no in-tree provider code.
  - Meanwhile, SIG-Cloud-Provider is working on a migration path to transition running clouds from in-tree to out-of-tree providers.
  - Unmaintained provider code will be removed completely.

Part II: Building a K8s Cloud Controller

Introduction to Cloud Controller Manager

The Cloud Controller Manager (CCM) replaces the Kube Controller Manager (KCM), and is daemon that embeds the following cloud-specific control loops:

- Node Controller
- Route Controller
- Service Controller

"Where's my volume controller?"

- Because of complexity of volume management, the CCM developers decided to not move volume control to the CCM and instead develop providers for Container Storage Interface (CSI).

Your Own Cloud Controller Manager!

In three easy steps.

1. Create a go package that satisfies the cloud provider interface.
2. Create a copy of the Cloud Controller Manager main.go and import your package, making sure there is an init block available.
3. There is no step 3*. There is no step 3!

* Except for building, testing, packaging, maintaining...

Cloud Provider Interface

- At a high level, to build an out-of-tree controller manager, you need to implement the Cloud Provider interface.
  - https://github.com/kubernetes/cloud-provider/blob/master/cloud.go
- Interfaces to implement (all are optional):
  - Load Balancer: cloud-specific ingress controller.
  - Instances: cloud-specific information about nodes in your cluster.
  - Zones: cloud-specific information about the host availability zones.
  - Clusters: cloud-specific information about running clusters.
  - Routes: cloud-specific information about networking.

```
Type Interface interface {
  // Initialize provides the cloud with a kubernetes client builder and may spawn goroutines
  // to perform housekeeping or run custom controllers specific to the cloud provider.
  // Any tasks started here should be cleaned up when the stop channel closes.
  Initialize(clientBuilder ControllerClientBuilder, stop <-chan struct{})

  // LoadBalancer returns a balancer interface. Also returns true if the interface is supported, false otherwise.
  LoadBalancer() (LoadBalancer, bool)

  // Instances returns an instances interface. Also returns true if the interface is supported, false otherwise.
  Instances() (Instances, bool)

  // Zones returns a zones interface. Also returns true if the interface is supported, false otherwise.
  Zones() (Zones, bool)

  // Clusters returns a clusters interface.  Also returns true if the interface is supported, false otherwise.
  Clusters() (Clusters, bool)

  // Routes returns a routes interface along with whether the interface is supported.
  Routes() (Routes, bool)

  // ProviderName returns the cloud provider ID.
  ProviderName() string

  // HasClusterID returns true if a ClusterID is required and set
  HasClusterID() bool
}
```

Cloud Controller Binary

The external cloud controller runs as a separate binary that interacts with the Kubernetes API service.

It is configured with a standard set of options, which can be extended to match the requirements of your cloud.

A starting template is provided in the Kubernetes CCM directory.

https://github.com/kubernetes/kubernetes/blob/master/cmd/cloud-controller-manager

```
import (
    "fmt"
    "math/rand"
    "os"
    "time"

    "k8s.io/component-base/logs"
    "k8s.io/kubernetes/cmd/cloud-controller-manager/app"
    "<my_cloud_provider>"
    _ "k8s.io/kubernetes/pkg/util/prometheusclientgo" // load all the prometheus client-go plugins
    _ "k8s.io/kubernetes/pkg/version/prometheus"     // for version metric registration
)

func main() {
    rand.Seed(time.Now().UnixNano())

    command := app.NewCloudControllerManagerCommand()

    // TODO: once we switch everything over to Cobra commands, we can go back to calling
    // utilflag.InitFlags() (by removing its pflag.Parse() call). For now, we have to set the
    // normalize func and add the go flag set by hand.
    // utilflag.InitFlags()
    logs.InitLogs()
    defer logs.FlushLogs()

    if err := command.Execute(); err != nil {
        fmt.Fprintf(os.Stderr, "error: %v\n", err)
        os.Exit(1)
    }
}
```

Two Types of Testing

At a minimum, your implementation needs two types of testing:

- Unit tests with appropriate mocks to guarantee your implementation behaves as expected functionally.
- End to end (e2e) testing with the provider enabled on an instance of your cloud for full integration and conformance testing.

If you want to enable release gating against your cloud, both must be implemented and e2e must be reporting to test-grid.

Running A Cloud Provider (the application)

Arguments to kube-api-server:

- `--cloud-provider=external`

Arguments to start your CCM binary

- `--cloud-provider=<your cloud provider name>`
- `--cloud-config=<path to your cloud configuration>`

The similarity in options comes from the scaffolding code. This comes at the cost of a lot of unneeded options also being pulled over to the provider.

Running A Cloud Provider (in production)

When running in production, use a daemonset!

Your cluster behavior will change in a few ways. Notably:

- “kubelets specifying `--cloud-provider=external` will add a taint `node.cloudprovider.kubernetes.io/uninitialized` with an effect `NoSchedule` during initialization.”
- “Cloud information about nodes in the cluster will no longer be retrieved using local metadata, but instead all API calls to retrieve node information will go through cloud controller manager. ...for larger clusters you may want to consider if cloud controller manager will hit rate limits since it is now responsible for almost all API calls to your cloud from within the cluster.”

```yaml
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cloud-controller-manager
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: system:cloud-controller-manager
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: cloud-controller-manager
  namespace: kube-system
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  labels:
    k8s-app: cloud-controller-manager
  name: cloud-controller-manager
  namespace: kube-system
spec:
  selector:
    matchLabels:
      k8s-app: cloud-controller-manager
  template:
    metadata:
      labels:
        k8s-app: cloud-controller-manager
    spec:
      serviceAccountName: cloud-controller-manager
      containers:
      - name: cloud-controller-manager
        # for in-tree providers we use
k8s.gcr.io/cloud-controller-manager
        # this can be replaced with any other image for out-of-tree
providers
        image: k8s.gcr.io/cloud-controller-manager:v1.8.0
        command:
        - /usr/local/bin/cloud-controller-manager
        - --cloud-provider=<YOUR_CLOUD_PROVIDER>
        # Add your own cloud provider here!
        - --leader-elect=true
        - --use-service-account-credentials
        # these flags will vary for every cloud provider
        - --allocate-node-cidrs=true
        - --configure-cloud-routes=true
        - --cluster-cidr=172.17.0.0/16
      tolerations:
      # this is required so CCM can bootstrap itself
      - key: node.cloudprovider.kubernetes.io/uninitialized
        value: "true"
        effect: NoSchedule
      # this is to have the daemonset runnable on master nodes
      # the taint may vary depending on your cluster setup
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      # this is to restrict CCM to only run on master nodes
      # the node selector may vary depending on your cluster setup
      nodeSelector:
        node-role.kubernetes.io/master: ""
```

Implementation Details

- What library will you use to interact with your cloud?
- How do you want to handle authentication and authorization?
  - It's likely that your cloud controller isn't the only thing interacting with your cloud.
  - For example, you may have a Cluster API provider or storage provider in the works.
  - Consolidate your efforts, otherwise you'll wind up with fragmented and inconsistent auth configuration methods across your projects.

Part III: Examples

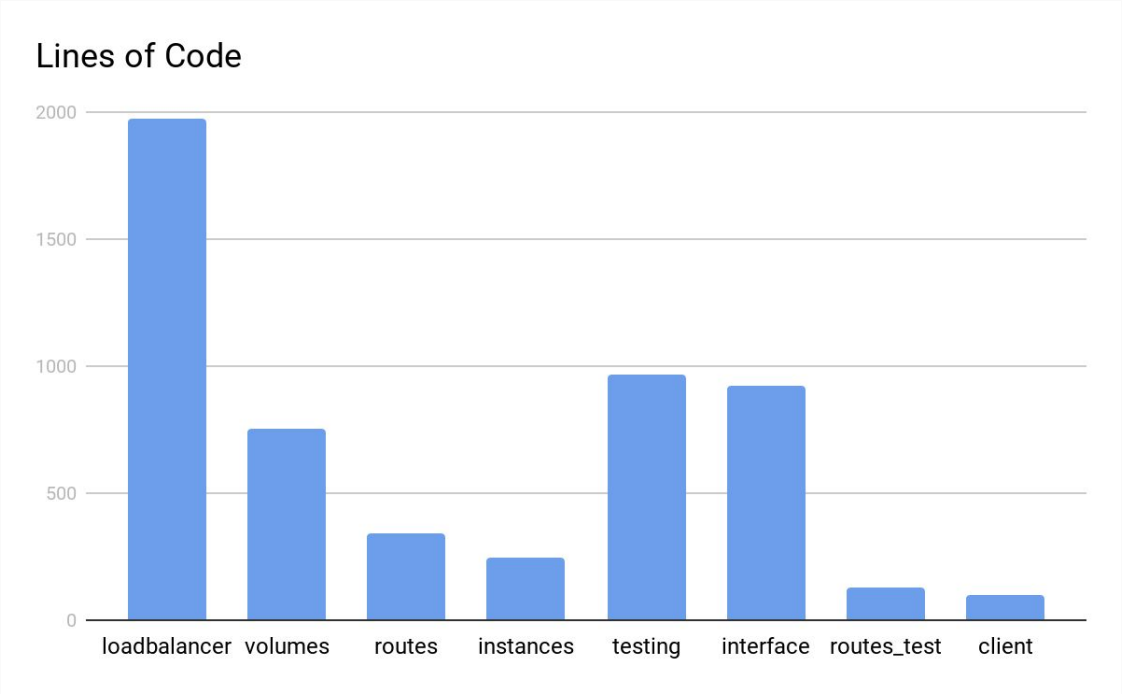
Cloud Provider OpenStack

http://git.k8s.io/cloud-provider-openstack

Some implementation details:

- Gopher Cloud is the OpenStack SDK we use for the provider
  - It's the same SDK used in the Terraform provider.
- We have a number of different sub-controllers that are independent of one another and are activated through configuration..
- Zones are not implemented, as they don't have a single authorization mechanism across the multitude of public and private OpenStack clouds.

Cloud Provider OpenStack (by the numbers)



Cloud Provider OpenStack (testing)

- OpenStack reuses existing tools for standing up various test environments, allowing for a matrix of compatibility testing.
- A test OpenStack cloud is brought up using DevStack, that standard integrated test environment within the OS community.
- `local-up-cluster.sh` is used to launch a Kubernetes cluster on the OpenStack cluster using the custom cloud controller manager.
- `e2e.go` is used to run conformance testing on the cluster, with results reported back to Test Grid.
- This is orchestrated with Ansible playbooks launched on a Zuul cluster.

Cloud Provider vSphere

http://git.k8s.io/cloud-provider-vsphere

- Cloud Provider VMware uses govmomi (github.com/vmware/govmomi)
  - vSphere API types package is HUGE (~23MB when compiled), another valid reason to keep cloud SDKs out of core K8s
- Currently does not implement:
  - `LoadBalancer()`
  - `Clusters()`
  - `Routes()`
- Supports vSphere versions from 6.5 onwards
- Shares secrets configuration with vSphere-CSI driver for storage

Cloud Provider vSphere (Zones support)

- vSphere availability zones / fault domains are science on their own
  - There are **books** about it…
- Cloud Provider VMware uses vSphere tags to map the zones and regions constructs into vSphere datacenter objects.
  - BYOT – Bring Your Own Topology
- Enables zones support for storage volumes with the CSI provider

Cloud Provider vSphere (What about storage?)

- As the `volumeController` control loop is not implemented in the cloud provider interface, we need a CSI plugin to provide storage functionalities
- When transitioning from In-Tree cloud provider to Out-of-Tree, we need both CCM and CSI to provide the same level of functionality
- This adds another dimension to the support matrix: K8s version, CCM version and CSI plugin version (on top of the vSphere version…)

Cloud Provider vSphere (Future: vSphere on AWS)

- Plain vSphere does not have a load balancer facility, so you have to BYO and integrate it.
- vSphere on AWS is integrated with several AWS native services, including Elastic Load Balancer
- Currently discovering how to enable consumption of services from multiple cloud providers within the same K8s cluster (e.g. vSphere cloud provider + AWS cloud provider)

SIG Cloud Provider

SIG Cloud Provider's mission is to simplify, develop, and maintain cloud provider integrations as extensions, or add-ons, to Kubernetes clusters

Important KEPs:

- Cloud Controller Manager https://git.io/fjBk2
- TestGrid conformance https://git.io/fjBkV
- Removing In-Tree Cloud Provider Code https://git.io/fjBkr

MUCHAS GRACIAS!

Questions?