Deploying Applications on Kubernetes Clusters

Once you have created Kubernetes cluster using the Cisco Container Platform web interface, you can deploy containerized applications on top of it.

This chapter contains the following topics:

Workflow of Deploying Applications

Task	Related Section
Create Kubernetes clusters using the Cisco Container Platform web interface.	Creating Clusters on vSphere
Download the kubeconfig file that contains the cluster information and the certificates required to access clusters.	Downloading Kubeconfig File, on page 1
Use the kubectl utility to deploy the application and test the scenario.	Sample Scenarios, on page 2

Downloading Kubeconfig File

You must download the cluster environment to access the Kubernetes clusters using command line tools such as `kubectl` or using APIs.

Step 1 In the left pane, click **Clusters**.

Step 2 Click the **Download** icon corresponding to the cluster environment that you want to download.

The `kubeconfig` file that contains the cluster information and the certificates required to access clusters is downloaded to your local system.

Sample Scenarios

This topic contains a few sample scenarios of deploying applications.

Deploying a Pod with Persistent Volume

Tenant clusters are deployed with a default storage class named **standard**, and a default storage class provider named **vSphere provider**.

If you select a HyperFlex local network during cluster creation, HyperFlex storage class and storage class provisioner are created by default.

In Cisco container Platform 4.0+, when deployed with Hyperflex 4.0+, the following two HyperFlex provisioners are supported:

- **hyperflex**, the HyperFlex FlexVolume provisioner available with HyperFlex 3.5+
- **hyperflex-csi**, the HyperFlex Container Storage Interface (CSI) provisioner available with HyperFlex 4.0+

Note We recommended that you use the HyperFlex Container Storage Interface (CSI) plugin, which is a more robust framework.

Step 1 Configure a tenant Kubernetes cluster.

```
export KUBECONFIG=<Path to kubeconfig file>
```

Step 2 Verify if the storage cluster is created.

```
kubectl describe storageclass standard
```

```
  Name:                  standard
  IsDefaultClass:        Yes
  Provisioner:           kubernetes.io/vsphere-volume
  Parameters:            diskformat=thin
  ReclaimPolicy:         Delete
  VolumeBindingMode:     Immediate
```

On HyperFlex 4.0+, if you have selected a HyperFlex local network, additional storage classes are displayed when you run the following command:

```
kubectl get sc
```

```
  NAME                 PROVISIONER                    AGE
  hyperflex            hyperflex.io/hxvolume          22h
  hyperflex-csi        csi-hxcsi                      22h
  standard (default)   kubernetes.io/vsphere-volume   22h
```

Step 3 Create the persistent volume claim to request for storage.

```
cat <<EOF > pvc.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
```

```
  name: pv-claim
spec:
  storageClassName: standard
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
EOF
```

Note The `storageClassName` field is optional. For HyperFlex 4.0+, you must use **hyperflex-csi** as the storage class.

```
kubectl create -f pvc.yaml
persistentvolumeclaim "pv-claim" created
```

Note The HyperFlex storage class supports the ReadWriteOnce or ReadOnlyMany access modes and the vSphere storage class supports the ReadWriteOnce access mode.

Step 4 Verify if the persistent volume claim (pvc) is created.

```
kubectl describe pvc pv-claim
Name:          pv-claim
Namespace:     default
StorageClass:  standard
Status:        Bound
Volume:        hx-default-pv-claim-5c4e8978-cdd2-11e8-9a07-005056b8fd7b
Labels:
Annotations:   pv.kubernetes.io/bind-completed=yes
               pv.kubernetes.io/bound-by-controller=yes
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      3Gi
Access Modes:  RWO,ROX
Events:        \
```

Persistent Volume is automatically created and is bounded to this pvc.

Note When **VSPHERE** is used as the default storage class, a VMDK file is created inside the **kubevols** folder in the datastore which is specified during the creation of the tenant Kubernetes cluster.

Step 5 Create a pod that uses persistent volume claim with storage class.

```
cat <<EOF > pvc-pod.yaml
kind: Pod
apiVersion: v1
metadata:
  name: pvc-pod
spec:
  volumes:
    - name: pvc-storage
      persistentVolumeClaim:
       claimName: pv-claim
  containers:
    - name: pvc-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: pvc-storage
EOF

kubectl create -f pvc-pod.yaml
pod "pvc-pod" created
```

Step 6 Verify if the pod is up and running.

```
kubectl get pod pvc-pod

NAME      READY     STATUS    RESTARTS   AGE
pvc-pod   1/1       Running   0          16s
```

When **VSPHERE** is used as the default storage class, you can access vCenter and view the dynamically provisioned VMDKs of the pod.

Deploying Cafe Application with Ingress

This scenario shows how to deploy the NGINX or NGINX Plus Ingress controller, the Cafe application, and then configure load balancing for the Cafe application using the Ingress resource.

For more information on Ingress, see Load Balancing Kubernetes Services using NGINX.

Step 1 Download the required artifacts.

a) Go to the following URL:

https://github.com/nginxinc/kubernetes-ingress/tree/master/examples/complete-example

b) Download the following yaml files:

- cafe-ingress.yaml
- cafe-secret.yaml
- cafe.yaml

Step 2 Deploy the Ingress controller.

a) Save the public IP address of the Ingress controller in a shell variable.

```
$ IC_IP=XXX.YYY.ZZZ.III
```

b) Save the HTTPS port of the Ingress controller in a shell variable.

```
$ IC_HTTPS_PORT=<port number>
```

Step 3 Deploy the Cafe application by creating the coffee and the tea deployments and services.

```
$ kubectl create -f cafe.yaml
```

Step 4 Configure load balancing.

a) Create a secret with an SSL certificate and a key.

```
$ kubectl create -f cafe-secret.yaml
```

b) Create an Ingress resource.

```
$ kubectl create -f cafe-ingress.yaml
```

Step 5 Test the application.

a) To access the application, curl the coffee and the tea services.

Note You can use the curl `--insecure` option to turn off certificate verification of the self-signed certificate and the `--resolve` option to set the Host header of a request with cafe.example.com.

- Accessing the coffee application

```
  $ curl --resolve cafe.example.com:$IC_HTTPS_PORT:$IC_IP
https://cafe.example.com:$IC_HTTPS_PORT/coffee --insecure
  Server address: 10.12.0.18:80
  Server name: coffee-7586895968-r26zn
  ...
```

- Accessing the tea application

```
 $ curl --resolve cafe.example.com:$IC_HTTPS_PORT:$IC_IP
https://cafe.example.com:$IC_HTTPS_PORT/tea --insecure
  Server address: 10.12.0.19:80
  Server name: tea-7cd44fcb4d-xfw2x
  ...
```

b) Monitor the runtime activities of the application in one of the following ways:

1. If you are using NGINX, follow the instructions to access the NGINX status page.
2. If you are using NGINX Plus, go to the **Upstream** tab.