Kiyot

Kiyot is a Kubernetes CRI (container runtime interface) shim that uses Milpa to schedule pods and containers. The most widely known CRI implementation is Docker, which is the default CRI as of today in Kubernetes.

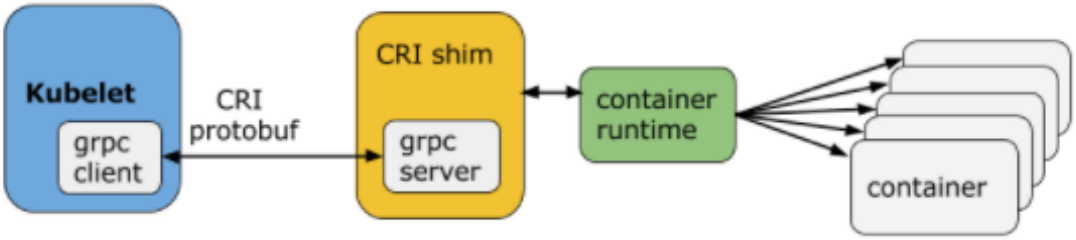

Figure 1: Kubernetes CRI

Architecture

Kiyot makes it possible to schedule unmodified Kuberetes workloads (pods, deployments, etc) in a cloud-native manner via Milpa. When launching a pod in Kubernetes, Kiyot will have Milpa provision a right-sized cloud compute instance for your pod, and terminate the instance when Kubernetes stops the pod.

Kubernetes uses the CRI to

- pull an image from a repository, inspect, and remove an image via ImageService, and
- manage the lifecycle of the pods and containers, as well as calls to interact with containers (exec/attach/port-forward) via RuntimeService.

Kiyot supports both services.

Installing Kiyot

Note: if you would like to set up a small test cluster for test driving Milpa and Kiyot with Kubernetes, see the Kiyot Tutorial section.

Assuming you already have a working Kubernetes cluster and would like to try nodeless Kubernetes, it is recommended to install Kiyot and Milpa directly on a Kubelet node. You are free to only deploy Kiyot and Milpa on a subset of the Kubelet nodes of your Kubernetes cluster. This way, you can use taints and tolerations to only schedule a part of your workload via Kiyot and Milpa.

For installing Kiyot, you need the self-extracting Milpa installer that includes Kiyot.

Assuming your Kubelet node is Debian or Ubuntu based, as root:

```
$ sudo apt-get update
$ sudo apt-get install -y awscli
# Download the Milpa installer.
$ curl -L https://download.elotl.co/milpa-installer-latest > /tmp/milpa-install.run
$ chmod 755 /tmp/milpa-install.run && sudo /tmp/milpa-install.run
```

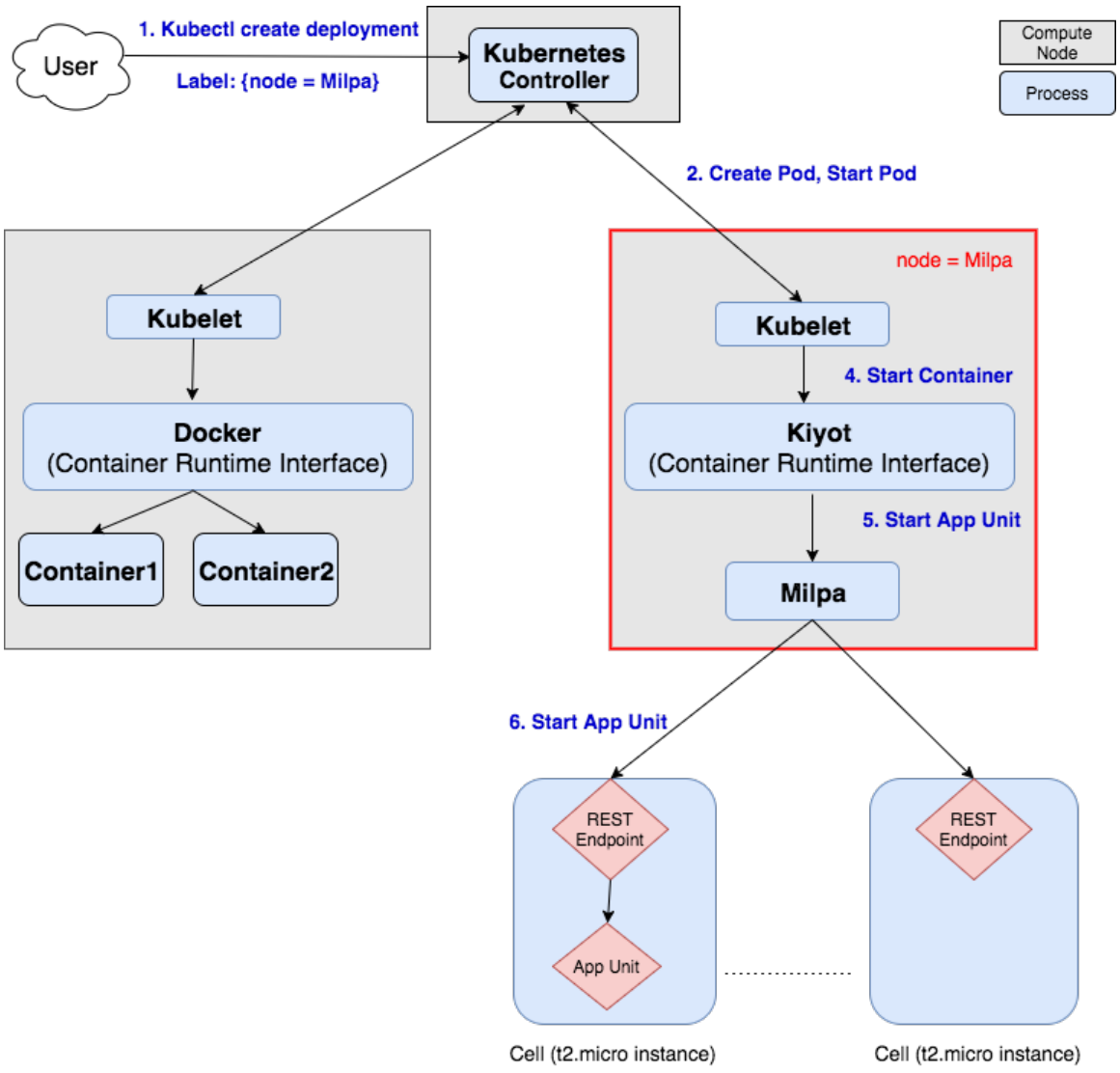

Figure 2: Kiyot Architecture

```
# Update kubelet config. Depending on your OS distribution and/or version, this might be different.
$ sudo sed -i -r 's#^DAEMON_ARGS="(.*)"$#DAEMON_ARGS="\1 --container-runtime=remote --container-runt
```

Edit the configuration file for Milpa (see the Milpa documentation for more details):

```
$ sudo vi /opt/milpa/etc/server.yml
```

and finally, restart Milpa and Kiyot:

```
$ sudo systemctl restart milpa; sleep 5; sudo systemctl restart kiyot
```

For more information on installing and configuring Milpa, please consult the Milpa documentation.

Using Taints and Tolerations with Kiyot

First, you need to add a taint to the Kubelet node running Kiyot and Milpa. Check the list of nodes in your Kubernetes cluster:

```
$ kubectl get nodes
NAME                            STATUS    ROLES     AGE       VERSION
ip-172-20-38-59.ec2.internal    Ready     node      1d        v1.10.7
ip-172-20-47-242.ec2.internal   Ready     master    1d        v1.10.7
```

Here, there is only one Kubelet node. Add a taint to it:

```
$ kubectl taint nodes ip-172-20-38-59.ec2.internal cri=kiyot:NoSchedule
node "ip-172-20-38-59.ec2.internal" tainted
```

Now, if you create a new pod without specifying a toleration matching this taint, that pod will not be scheduled to run on this Kubelet node:

```
$ cat pod.yml
---
apiVersion: v1
kind: Pod
metadata:
  name: memcached
spec:
  containers:
  - name: memcached
    image: memcached
$ kubectl create -f pod.yml
pod "memcached" created
$ kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
memcached   0/1       Pending   0          45s
```

Since there is only one Kubelet node in this cluster, the pod will remain pending. Remove the pod, and add the required tolerations to the manifest:

```
$ kubectl delete pod memcached
pod "memcached" deleted

$ vi pod.yml
[edit pod spec]
$ cat pod.yml
```

```
---
apiVersion: v1
kind: Pod
metadata:
  name: memcached
spec:
  containers:
  - name: memcached
    image: memcached
  tolerations:
  - key: "cri"
    operator: "Equal"
    value: "kiyot"
    effect: "NoSchedule"
```

Now, when you create the pod again, it will be scheduled to run on the Kubelet node via Kiyot and Milpa:

```
$ kubectl create -f pod.yml
pod "memcached" created
$ kubectl get pods
NAME        READY     STATUS              RESTARTS   AGE
memcached   0/1       ContainerCreating   0          9s
```

Eventually getting to a "Running" state once its cloud instance is up and the application has started:

```
$ kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
memcached   1/1       Running   0          1m
```

To remove the taint:

```
$ kubectl taint nodes ip-172-20-38-59.ec2.internal cri:NoSchedule-
```

Differences from Kubernetes with Docker

Kubernetes assumes that it launches containers, running locally on Kubelets (Kubernetes worker nodes), via the CRI. In reality, workloads will be scheduled to run in cloud instances instead. Since Kubernetes uses the CRI exclusively to interact with the "containers", however, Kiyot can seamlessly translate between Kubernetes and Milpa.

Since the Docker CRI (and other "local" CRIs) allows containers to interact directly with the host (i.e., the Kubelet node), there might still be a few surprises when using Kiyot.

Interacting with Namespaces on the Host

Kubernetes allows privileged containers to interact with the network, pid or IPC namespaces of the host. For example, a pod might request to share the network namespace with the host:

```
apiVersion: v1
kind: Pod
metadata:
  name: influxdb
spec:
  hostNetwork: true
  containers:
    - name: influxdb
      image: influxdb
```

This is usually reserved for very specific use cases (e.g. Kubernetes networking plugins), and application pods from end users rarely use these features. Sharing namespaces with the host is not supported by Kiyot, and the request to do so will be ignored by Kiyot and Milpa.

DaemonSets

A DaemonSet ensures that all (or some) Kubernetes nodes run a copy of a Pod. As nodes are added to the cluster, pods are added to them. As nodes are removed from the cluster, those pods are garbage collected.

If a pod from a DaemonSet is scheduled via Kiyot, it will run in a cloud instance, just like any other Milpa pod. This is likely not what you want, since DaemonSet pods are usually used to interact with the host (their Kubelet node) in some way (e.g. mapping files or directories from the host into the container, or interacting with host namespaces on the Kubelet node).

Pulling Images

When pulling an image, the image will get downloaded to the cloud instance that gets allocated to run the pod. This has the side effect that problems with pulling the image (invalid image, authentication problems, etc) will only occur once a cloud instance has been allocated to the pod, and the cloud instance tries to pull the image.

See the Troubleshooting section on how to get logs and more debug information on the status of the pod from Milpa to help diagnose this problem.

Mounting Volumes into Containers

EmptyDirs are fully supported in Milpa and Kiyot, so containers in your pod can share data via an EmptyDir volume.

Other kinds of volumes that are mounted into containers are only partially supported currently: they are always read only, and any updates performed inside a container will not be reflected in other containers if the volume is shared across multiple containers or multiple pods.

Logs

Currently, `kubectl logs` is only partially supported.

Tailing logs is fully supported (i.e. `kubectl logs -f <podname>`).

Checking logs without tailing is only partially supported (i.e. `kubectl logs <podname>`): the first time `kubectl logs` is used, the log history will be empty, but Kiyot will start collecting logs for the pod. Subsequent calls to `kubectl logs` will be able to retrieve logs collected from that point on, until a timeout of 60 seconds (i.e. logs are not checked for 60 seconds).

```
$ kubectl logs date # No output, but Kiyot will start collecting logs.
$ kubectl logs -f date # Tailing logs works.
Wed Oct 17 21:35:10 UTC 2018
Wed Oct 17 21:35:11 UTC 2018
Wed Oct 17 21:35:12 UTC 2018
Wed Oct 17 21:35:13 UTC 2018
Wed Oct 17 21:35:14 UTC 2018
Wed Oct 17 21:35:15 UTC 2018
^C
$ kubectl logs date # Now Kiyot has collected logs for this pod.
Wed Oct 17 21:35:10 UTC 2018
Wed Oct 17 21:35:11 UTC 2018
Wed Oct 17 21:35:12 UTC 2018
Wed Oct 17 21:35:13 UTC 2018
Wed Oct 17 21:35:14 UTC 2018
Wed Oct 17 21:35:15 UTC 2018
Wed Oct 17 21:35:16 UTC 2018
Wed Oct 17 21:35:17 UTC 2018
Wed Oct 17 21:35:18 UTC 2018
Wed Oct 17 21:35:19 UTC 2018
$ date # A few minutes later.
Wed Oct 17 21:47:37 UTC 2018
$ kubectl logs date # This will show logs until the 60s timeout has passed.
Wed Oct 17 21:35:10 UTC 2018
Wed Oct 17 21:35:11 UTC 2018
Wed Oct 17 21:35:12 UTC 2018
Wed Oct 17 21:35:13 UTC 2018
Wed Oct 17 21:35:14 UTC 2018
Wed Oct 17 21:35:15 UTC 2018
Wed Oct 17 21:35:16 UTC 2018
Wed Oct 17 21:35:17 UTC 2018
Wed Oct 17 21:35:18 UTC 2018
```

```
Wed Oct 17 21:35:19 UTC 2018
Wed Oct 17 21:35:20 UTC 2018
Wed Oct 17 21:35:21 UTC 2018
Wed Oct 17 21:35:22 UTC 2018
[...]
Wed Oct 17 21:36:20 UTC 2018
Wed Oct 17 21:36:21 UTC 2018
Wed Oct 17 21:36:22 UTC 2018
```

Troubleshooting

In some cases, the information retrieved via `kubectl` might not be enough to debug what is happening with your workload. Interacting directly with Milpa might reveal more and help in debugging.

To interact with Milpa directly, you will need to ssh into the Kubelet. Ssh access to Kubelet nodes varies depending on the user's environment: some Kubernetes clusters use a bastion host for ssh access, while other setups may allow direct access via the Kubelet nodes' public IP address.

Once you are on the Kubelet node that runs Kiyot and Milpa, check that `milpactl` is working:

```
$ milpactl version
Milpa server version: {"Major":"1","Minor":"0","GitVersion":"1.0.0-beta.7","GitCommit":"43196699","G
```

Status of Pods

You can check pods via:

```
$ milpactl get pods
NAME                                    UNITS     RUNNING   STATUS
kube-dns-6c7dfbf97d-42k5k               3         3         Pod Running
kube-dns-6c7dfbf97d-kr9cg               3         3         Pod Running
kube-dns-6c7dfbf97d-n4vb9               3         3         Pod Running
kube-dns-6c7dfbf97d-tnxvv               4         3         Pod Running - Unit Terminated: ExitCod
kube-dns-6c7dfbf97d-wv9zm               3         3         Pod Running
kube-dns-autoscaler-6874c546dd-56ndq    1         1         Pod Running
kube-proxy-ip-172-20-38-59.ec2.internal 1         1         Pod Running
```

An individual pod via:

```
$ milpactl get pod kube-proxy-ip-172-20-38-59.ec2.internal
NAME                                    UNITS     RUNNING   STATUS        RESTARTS   NODE
kube-proxy-ip-172-20-38-59.ec2.internal 1         1         Pod Running   0          38c2b0b4-c1da
```

You can also get the output in json or yaml format, via adding `-ojson` or `-oyaml` to the command:

```
$ milpactl get pod kube-proxy-ip-172-20-38-59.ec2.internal -ojson
{
    "kind": "Pod",
    "apiVersion": "v1",
    "metadata": {
        "name": "kube-proxy-ip-172-20-38-59.ec2.internal",
        "uid": "98dfbb0f-3b36-41fc-8d34-c7f0f2e4e296",
        "namespace": "default",
        [...]
    },
    "spec": {
        "phase": "Running",
        "restartPolicy": "Never",
        "units": [
            {
                "name": "kube-proxy-1",
                "image": "k8s.gcr.io/kube-proxy:v1.10.7",
                "command": [
```

```
                    "/bin/sh",
                    "-c",
                    "mkfifo /tmp/pipe; (tee -a /var/log/kube-proxy.log \u003c /tmp/pipe \u0026 ) ; e
                ],
                "args": null,
                "env": [
                    {
                        "name": "KUBE_DNS_SERVICE_HOST",
                        "value": "100.64.0.10"
                    },
                    {
                        "name": "KUBE_DNS_PORT_53_UDP",
                        "value": "udp://100.64.0.10:53"
                    },
                    [...]
                ],
            }
        ],
        "initUnits": [],
        "imagePullSecrets": null,
        "instanceType": "c5.large",
        "spot": {
            "policy": "Never"
        },
        "resources": {
            "cpu": "0.10",
            "sustainedCPU": false
        },
        "placement": {
            "availabilityZone": ""
        },
    },
    "status": {
        "phase": "Running",
        "lastPhaseChange": "2018-10-16T17:59:27.894515581Z",
        "boundNodeName": "38c2b0b4-c1da-4792-ba37-78bbfba54a4c",
        "boundInstanceID": "i-0319c832acf318172",
        "addresses": [
            {
                "type": "PrivateIP",
                "address": "172.20.33.93"
            },
            {
                "type": "PrivateDNS",
                "address": "ip-172-20-33-93.ec2.internal"
            },
            {
                "type": "PublicIP",
                "address": "35.175.144.248"
            },
            {
```

```
                "type": "PublicDNS",
                "address": "ec2-35-175-144-248.compute-1.amazonaws.com"
            }
        ],
        "startFailures": 0,
        "spotFailure": false,
        "unitStatuses": [
            {
                "name": "kube-proxy-1",
                "state": {
                    "running": {
                        "startedAt": "2018-10-16T11:06:57-07:00"
                    }
                },
                "restartCount": 0,
                "image": "k8s.gcr.io/kube-proxy:v1.10.7"
            }
        ],
        "initUnitStatuses": []
    }
}
```


One field that might reveal more information is `unitStatuses` (and `initUnitStatuses` for init containers). In case of a failure, this field contains more information on what happened. For example, in case of an image pull failure due to a non-existent image:


```
"unitStatuses": [
    {
        "name": "memcached-0",
        "state": {
            "waiting": {
                "reason": "Error pulling image for unit memcached-0: pulling image library/non-exist
                "startFailure": true
            }
        },
        "restartCount": 0,
        "image": "non-existent-image:latest"
    }
],
```


Logs

You can use `milpactl logs` to check the logs of a pod or a unit (container) inside the pod:

```
$ milpactl logs mypod
[logs from the first container/unit in the pod named "mypod"]
```

To check the logs from a specific unit:

```
$ milpactl logs mypod -u mycontainer
[logs from container "mycontainer" in the pod named "mypod"]
```

You can also tail logs via `milpactl logs -f`.

If you have the ID of the cloud instance the pod is bound to (check the `NODE` column in the output from `milpactl get pod <podname>`), you can also check the logs of the agent running on the cloud instance and managing the pod. For example:

```
$ milpactl logs 5878820f-ebcf-4035-b7d9-9843ba071604
[logs from agent]
```

Using `milpactl logs` does not have the limitations `kubectl logs` has.

Support bundle

If rootcausing your failure leads to dead end and you need to contact Elotl Support, please include following payload while reaching out to Elotl.

- Sanitized `/opt/milpa/etc/server.yml` after removing your AWS keys.
- Kiyot logs gathered via `journalctl -u kiyot -l --no-pager`.
- Milpa logs gathered via `journalctl -u milpa -l --no-pager`.
- Milpa dump gathered via `milpactl dump all > milpaDump.txt`.

Kiyot Tutorial

Introduction

This tutorial will walk you through the following steps.

1. Create a 1.10.7 Kubernetes cluster on AWS, with a worker node using Kiyot as the CRI instead of Docker. (`1.10.7` is the version we have tested. Other versions of Kubernetes that support CRI API versions `v1alpha1` or `v1alpha2` should also work.)
2. Deploy two applications:

- Kubernetes dashboard
- Guestbook app

3. Remove the cluster

Prerequisites

1. Please make sure your IAM user has following permissions: `AmazonEC2FullAccess`, `AmazonRoute53FullAccess`, `AmazonS3FullAccess`, `IAMFullAccess`, `AmazonVPCFullAccess`, `AmazonDynamoDBFullAccess`
2. Create a new AWS EC2 instance with Ubuntu 16.04 (`ami-759bc50a`) and log into it. The remaining steps assume you are logged in to your new instance.
3. Install the following packages:

```
$ sudo apt-get update
$ sudo apt --assume-yes install python
$ sudo apt install python-pip
$ pip install pyyaml --user
```

4. Set the default region to `us-east-1`.

```
$ export AWS_DEFAULT_REGION=us-east-1
```

5. Set your AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY.

```
$ export AWS_ACCESS_KEY_ID=xxxx
$ export AWS_SECRET_ACCESS_KEY=xxxx
```

6. Create a new ssh keypair by running below command (accept defaults).

```
$ ssh-keygen -t rsa -b4096
```

Installation

1. Download the kops installer on your EC2 instance.
2. Extract `kops-latest.tar.gz`.

```
$ tar xvf kops-latest.tar.gz && cd kops
```

3. Edit `server.yml`. Set a unique `clusterName`, configure your cloud access credentials and add your license details under `license`.

```
$ vi server.yml
```

4. Run the provisioning script. This will create a new Kubernetes cluster with one master and one worker node, and configure the worker node to use Kiyot as its CRI.

   ```
   $ ./provision.sh
   ```

5. Check that you have a functional Kubernetes cluster.

```
# kubectl cluster-info
Kubernetes master is running at https://api-root-2ok55yq4-k8s-loc-jfh3dn-1605922318.us-east-1.elb.am
KubeDNS is running at https://api-root-2ok55yq4-k8s-loc-jfh3dn-1605922318.us-east-1.elb.amazonaws.co

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

```
# kubectl get nodes
NAME                           STATUS    ROLES     AGE       VERSION
ip-172-20-35-65.ec2.internal   Ready     node      18m       v1.10.7
ip-172-20-47-96.ec2.internal   Ready     master    18m       v1.10.7
```

Tutorials

Kubernetes Dashboard

Deploy the Kubernetes Dashboard using the following steps.

Step 1

Grant admin privilege to Dashboard

Create `dashboard-admin.yaml` and apply it:

```
$ cat << EOF > dashboard-admin.yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: kubernetes-dashboard
  labels:
    k8s-app: kubernetes-dashboard
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: kubernetes-dashboard
  namespace: kube-system
EOF
$ kubectl create -f dashboard-admin.yaml
```

Step 2

Follow the instructions to deploy the Kubernetes Dashboard.

Wait for the dashboard deployment to be `AVAILABLE`:

```
$ kubectl -n kube-system get deployment kubernetes-dashboard
NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kubernetes-dashboard   1         1         1            1           3m
```

Step 3 (Optional)

If you want to access dashboard via a browser on your local machine, you will need to use the default kubeconfig context from your EC2 instance. You can view it on your EC2 instance via:

```
$ kubectl config view --raw
```

Add the output from above to `$HOME/.kube/config` on your local machine. You can now run `kubectl proxy` locally to access your Kubernetes cluster.

Step 4

Access your dashboard by pointing your browser at

`http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/`

Note: this will only work if you did not skip step 3.

Guestbook Application

Step 1

Start up the Redis master

Step 2

Start up the Redis slaves

Step 3

Set up the Guestbook Frontend Deployment.

Once the frontend deployment is created, create the frontend service. You will need to change the type of the service from `type: NodePort` to `type: LoadBalancer` (so that you won't need to open up the port via a security group to be able to access the service from the outside).

a) Download `frontend-service.yaml`.

```
$ wget https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/applicat
```

b) Edit `frontend-service.yaml`, comment out `type: NodePort`, and uncomment `type: LoadBalancer`.

c) Apply the yaml to create the frontend service.

```
$ kubectl apply -f frontend-service.yaml
```

Step 4

Verify that the application is working

If your `EXTERNAL-IP` is truncated, you can get it via:

```
$ kubectl get service frontend --output jsonpath='{.status.loadBalancer.ingress[0].hostname}'
a50c6dfe3bd2911e899db02678f5841c-612345793.us-east-1.elb.amazonaws.com
```

It might take a few minutes for the ELB endpoint to become available.

Step 5

Scale the web frontend

Step 6

Clean up

Terminate the Dashboard Application

Delete the dashboard service and deployment:

```
$ kubectl -n kube-system delete service kubernetes-dashboard
service "kubernetes-dashboard" deleted

$ kubectl -n kube-system delete deployment kubernetes-dashboard
deployment.extensions "kubernetes-dashboard" deleted
```

Tear Down the Cluster

The provisioning script will create a script for removing the Kubernetes cluster and all associated resources, named `delete-<USERNAME>-<CLUSTER_NAME>-<TIMESTAMP>.sh` (e.g. `delete-ubuntu-vo3g666q.k8s.local-1537482934.sh`). Run this script on the provisioning EC2 instance to destroy your cluster.

```
$ cd $HOME/kops && ./delete-ubuntu-vo3g666q.k8s.local-1537482934.sh
```