Frontiir Multi-node HA K8s Cluster Setup Guide

Requirements

In the Philadelphia testbed, we use nine nodes in total for the setup: three for the HA Rancher server cluster, which acts as a cluster manager. It can be accessed using a FQDN that maps to a virtual IP address. The rest six of the nodes are for the HA downstream user cluster. All nodes are preinstalled with Ubuntu 18.04 x86_64 and run on an internal network 10.2.0.0/16 (interface eth0).Their detailed information are listed below:

IP Address	Host Name	Role
10.2.252.201	rancher1	Rancher server node (control plane, etcd, worker); Keepalived & HAproxy
10.2.252.202	rancher2	Rancher server node (control plane, etcd, worker) ; Keepalived & HAproxy
10.2.252.203	rancher3	Rancher server node (control plane, etcd, worker); Keepalived & HAproxy
10.2.252.204	master1	Control plane, etcd
10.2.252.205	master2	Control plane, etcd
10.2.252.206	master3	Control plane, etcd
10.2.252.207	worker1	Worker
10.2.252.208	worker2	Worker
10.2.252.209	worker3	Worker
10.2.252.101		Virtual IP address for Rancher server

Note: Control plane consists of API server, Scheduler and Controller Manager; Worker consists of Kubelet, kube-proxy and container runtimes.

We use rke 1.1.15 for the upstream cluster, which runs Rancher v.2.5.7; kubernetes

v1.20.4-rancher1-1 for the downstream cluster. Their node requirements can be checked from below links:
Node Requirements for Rancher Server Cluster
Node Requirements for Rancher Managed Clusters

Installation

Pre-configuration and Dependency Installation

On all nodes as well as our host machine (the one that has a browser installed), we add below record to **/etc/hosts**:

```
10.2.252.101 rancher.local.frontiir.net
```

Docker is required on all nodes. Follow the Docker Documentation for the installation steps. It is also required to add the current user ("ubuntu" in our case) to the "docker" group to facilitate later steps, where Docker needs to be executed as non-root user:

```
$ sudo usermod -aG docker $USER
```

The ntp (Network Time Protocol) package needs to be installed on rancher1/2/3. This prevents errors with certificate validation that can occur when the time is not synchronized between the client and server:

```
$ sudo apt-get update
$ sudo apt-get install ntp
```

Kubectl and Helm3 should be installed on all nodes to facilitate command-line interactions with clusters as well as kubernetes package management. Follow their documentations for the installations:
Install and Set Up kubectl on Linux
Install Helm

Setting Up the Cluster for Rancher Server with RKE

The HA Rancher server can be hosted on many types of kubernetes clusters such as RKE, K3s and hosted kubernetes (GKE, EKS, AKS). Here we use RKE. Note that the first RKE cluster can only be provisioned using a command-line installer named **rke**. Once we have a working Rancher server, we can provision other downstream RKE clusters from there using the docker commands generated by the Rancher server (as we shall see later).

On the host machine (the workstation used to access all nodes), download the rke binary from the latest available RKE release (we use rke v1.1.15), then make rke an executable:

```
$ sudo mv rke_linux-amd64 /usr/local/bin
```

```
$ sudo mv /usr/local/bin/rke_linux-amd64 /usr/local/bin/rke
$ sudo chmod +x /usr/local/bin/rke
$ rke --version    # verify the executable
```

In order to provision a rke cluster for the Rancher server, we also need to provide rke with a cluster configuration file. Below is our sample rancher-cluster.yml:

```
nodes:
  - address: 10.2.252.201
    user: ubuntu
    role: [controlplane, worker, etcd]
  - address: 10.2.252.202
    user: ubuntu
    role: [controlplane, worker, etcd]
  - address: 10.2.252.203
    user: ubuntu
    role: [controlplane, worker, etcd]

services:
  etcd:
    snapshot: true
    creation: 6h
    retention: 24h

# Required for external TLS termination with
# ingress-nginx v0.22+
ingress:
  provider: nginx
  options:
    use-forwarded-headers: "true"
```

It is important to note that for each node, the yml file has an configurable option **ssh_key_path**, which specifies the path on the host machine to the SSH private key used to authenticate to the node. It defaults to **~/.ssh/id_rsa**.

Now we run rke to deploy the kubernetes cluster specified as above:

```
$ rke up --config ./rancher-cluster.yml
```

When finished, it should print “Finished building Kubernetes cluster successfully”. A successful deployment should generate two new files: **kube_config_rancher-cluster.yaml** and **rancher-cluster.rkestate**. The first one is the Kubeconfig file for the cluster, which contains credentials for full access to the cluster. The second file is the Kubernetes Cluster State file, which contains credentials for full access to the cluster. Along with the **rancher-cluster.yml file**

we created earlier, make a copy of three of them and keep the copies in a safe place - they will be useful when we need to maintain, troubleshoot and upgrade the cluster later.
In order to interact with the cluster with kubectl, we need to copy the kube_config_rancher-cluster.yaml file from the host machine to Rancher server nodes, then do:

```
ubuntu@rancher1/2/3:~$ mkdir ~/.kube
ubuntu@rancher1/2/3:~$ mv kube_config_rancher-cluster.yaml config
ubuntu@rancher1/2/3:~$ mv config ~/.kube
ubuntu@rancher1/2/3:~$ chmod 600 ~/.kube/config # prevent the kubeconfig to be globally accessible
ubuntu@rancher1/2/3:~$ kubectl get nodes # verify kubectl
```

For more references on this topic, refer to Setting up a HA RKE Kubernetes Cluster.

Installing the Rancher Server

Follow the Install/Upgrade Rancher on a Kubernetes Cluster step by step for the installation. Several things to pay attention to: in step 2, it is recommended to select the stable version (we use v2.5.7) of Racher when aiming for production; in step 5, remember to click and expand the code block for installing cert-manager; in step 6, we choose the Rancher-generated certificates, and set the custom hostname to “rancher.local.frontiir.net”, which is the FQDN for the load balancer:

```
# Example command for installing Rancher with helm
helm install rancher rancher-stable/rancher \
--namespace cattle-system \
--set hostname=rancher.local.frontiir.net
```

Setting Up the Load Balancer for Rancher Server

A layer-4 load balancer is needed to direct traffic to one of the Rancher server nodes. Here we use keepalived and HAproxy on the server nodes themselves for such purpose - this co-located setup helps reduce the number of VMs needed to achieve the HA of both Rancher server and its load balancer.

First install keepalived and HAproxy on all Rancher server nodes:

```
ubuntu@rancher1/2/3:~$ sudo apt-get update
ubuntu@rancher1/2/3:~$ sudo apt-get install keepalived haproxy -y
```

Then edit the keepalived configuration in **/etc/keepalived/keepalived.conf** as root:

```
! /etc/keepalived/keepalived.conf
! Configuration File for keepalived
global_defs {
    router_id LVS_DEVEL
```

```
}
vrrp_script check_rancher_server {
  script "/etc/keepalived/check_rancher_server.sh"
  interval 3
  weight -2
  fall 10
  rise 2
}

vrrp_instance VI_1 {
    state MASTER
    interface eth0
    virtual_router_id 97
    priority 101
    authentication {
        auth_type PASS
        auth_pass zV#RE8c&dhYkyL7g
    }
    virtual_ipaddress {
        10.2.252.101
    }
    track_script {
        check_rancher_server
    }
}
```

Note that above is the keepalived configuration for rancher1. For rancher2/3, simply replace **MASTER** with **BACKUP** for “state” and replace **101** with **100** for “priority”. Please check with your network admin for the keepalived priority ID.

Also place a health checking script **check_rancher_server.sh** like below under **/etc/keepalived/** of each Rancher server node:

```
#!/bin/sh
RANCHER_SERVER_VIP=10.2.252.101
RANCHER_SERVER_DEST_PORT=443

errorExit() {
    echo "*** $*" 1>&2
    exit 1
}

curl --silent --max-time 2 --insecure
```

```
https://localhost:${RANCHER_SERVER_DEST_PORT}/ -o /dev/null || errorExit
"Error GET https://localhost:${RANCHER_SERVER_DEST_PORT}/"
if ip addr | grep -q ${RANCHER_SERVER_VIP}; then
    curl --silent --max-time 2 --insecure
https://${RANCHER_SERVER_VIP}:${RANCHER_SERVER_DEST_PORT}/ -o /dev/null ||
errorExit "Error GET
https://${RANCHER_SERVER_VIP}:${RANCHER_SERVER_DEST_PORT}/"
fi
```

Then apply above configuration to keepalived:

```
root@rancher1/2/3:~# systemctl enable keepalived --now
```

Similarly, edit the configuration of HAproxy in **/etc/haproxy/haproxy.cfg** as root:

```
global
    maxconn     4096
    log         /dev/log local0
    nbproc      2
    nbthread    4

defaults
    mode                tcp
    log                 global
    timeout connect     10s
    timeout client      30s
    timeout server      30s

listen rancher-http
    bind *:80
    balance roundrobin
    server rancher1 10.2.252.201:80 check
    server rancher2 10.2.252.202:80 check
    server rancher3 10.2.252.203:80 check

listen rancher-https
    bind *:443
    balance roundrobin
    server rancher1 10.2.252.201:443 check
    server rancher2 10.2.252.202:443 check
    server rancher3 10.2.252.203:443 check
```

Then test and apply above configuration to HAproxy:

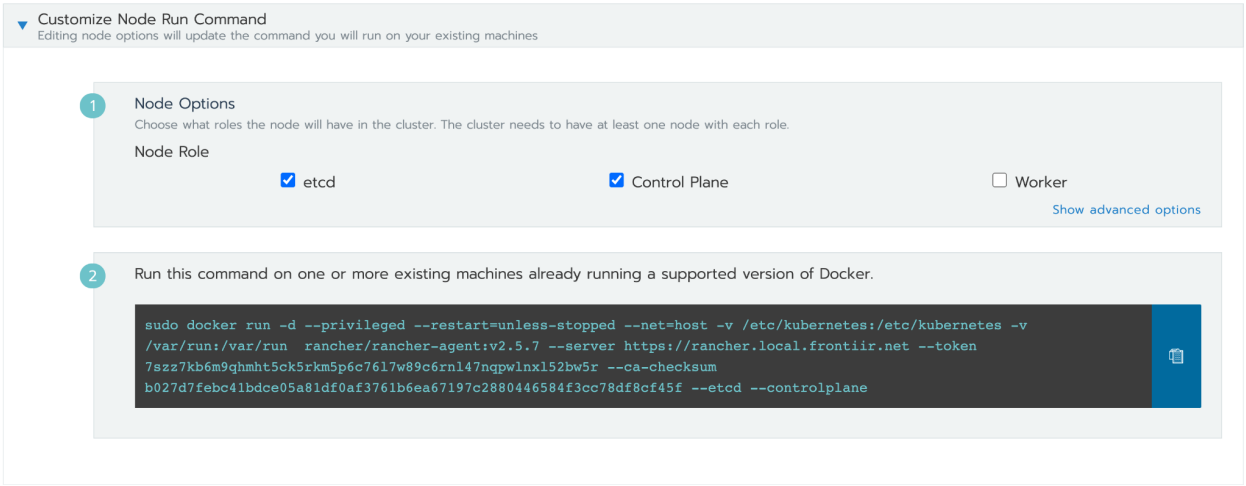
```
root@rancher1/2/3/:~# haproxy -c -f /etc/haproxy/haproxy.cfg
root@rancher1/2/3/:~# systemctl enable haproxy --now
```

Provisioning the Downstream Cluster

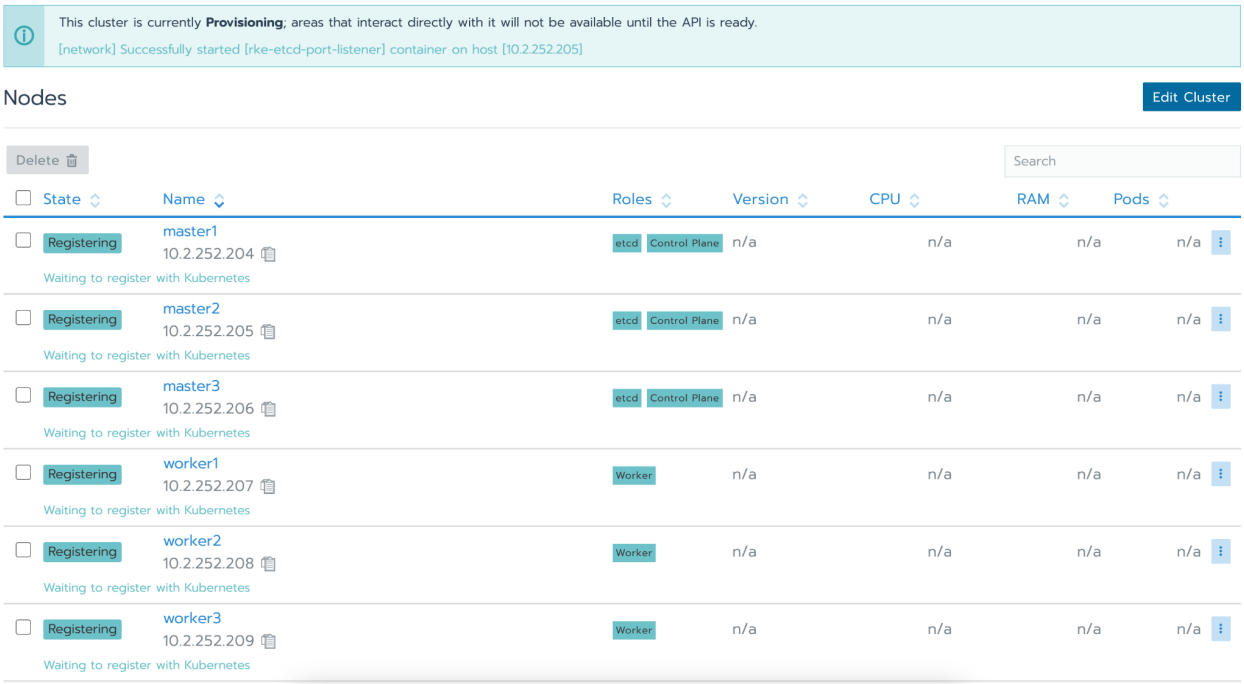
After correctly carrying out the above instructions, we can access the Rancher UI by entering “https://rancher.local.frontiir.net” using a browser on the host machine. The first user accessing the page will be granted the role of Cluster Owner (username: admin). Make sure to set up a strong password and keep it in a safe place. Once logged in, we can see a “local” cluster showing up in the “cluster manager” landing page, which corresponds to the RKE cluster we just set up for the Rancher server.

To bootstrap a downstream RKE cluster using the Rancher UI, simply click “Add Cluster” and select the “Existing Nodes” option under the "Create a new Kubernetes cluster” option. We recommend choosing **Canal** as the network provider (CNI) due to it makes possible project-level network isolation, which is a vital component of multi-tenancy. We also recommend enabling the “Authorized Cluster Endpoint” feature.

After choosing the basic configurations for the new cluster, click next and select the role(s) to assign to a node, a corresponding docker command will be generated. To register a node to the cluster, copy that command and run it in the terminal of that node. It’s recommended to set up all control plane nodes before the worker nodes. Below shows generating a command for registering a master node:



The provision process takes around 10 minutes. If anything goes wrong, look for the events described in Rancher UI as well as the logs of related docker containers (described in the Logs and Monitoring section). Below shows how the Rancher UI looks like when provisioning a cluster:



When the provision completes, retrieve the kubeconfig file from the “kubeconfig file” and place it into ~/.kube/config of each node (this same way as we did for the Rancher server nodes).

Important: Resolving the custom Rancher server domain in downstream clusters (related discussions: [Rancher Cattle Cluster Agent Could not Resolve Host]())

Regardless of the downstream cluster is launched by Rancher or imported, the following command needs to be applied to it to let cattle-cluster-agent able to resolve the Rancher server’s domain:

```
kubectl -n cattle-system patch deployments cattle-cluster-agent --patch '{
    "spec": {
        "template": {
            "spec": {
                "hostAliases": [
                    {
                      "hostnames":
                      [
                        "rancher.local.frontiir.net"
                      ],
                      "ip": "10.2.252.101"
                    }
                ]
            }
        }
```

```
    }
}'
```

If the downstream cluster is RKE launched by Rancher, we also need to apply below command:

```
kubectl -n cattle-system patch daemonsets cattle-node-agent --patch '{
 "spec": {
     "template": {
         "spec": {
             "hostAliases": [
                 {
                    "hostnames":
                      [
                        "rancher.local.frontiir.net"
                      ],
                    "ip": "10.2.252.101"
                 }
             ]
         }
     }
 }
}'
```

Authorized Cluster Endpoint (Good to Know)

In an unfortunate case, when the Rancher server becomes unavailable, the operations of downstream clusters would be **unaffected**. However the RKE clusters launched by the Rancher server would become unreachable from the default context of kubectl, due to the failure of cluster controller components as mentioned in the Rancher Architecture Page.

Luckily, we can bypass the Rancher server and directly interact with the downstream RKE clusters with a feature called “Authorized Cluster Endpoint” (this is enabled by default when creating the cluster). Then from any downstream master node, assuming a valid kubeconfig file is stored at the default location ~/.kube/, we may retrieve all the contexts like this:

```
ubuntu@master1/2/3:~$ kubectl config get-contexts
```

You can see that besides the “global” context which routes through the Rancher server, each master node also has its own context, which can be used as the authorized endpoints for directly interacting with the downstream cluster. Below command shows an example of switching a context to master node1:

```
ubuntu@master1/2/3: ~$ kubectl config set-context \
                          frontiir-pa-cluster1-master1
```

For more references on this part, see Access a Cluster with Kubectl and kubeconfig.

CI/CD with Gitlab and ArgoCD

The CI/CD workflow is exactly the same as the one described in the Frontiir Single Node Cluster Documentation. As we are no longer using Microk8s here, be sure to remove the “microk8s” prefix in each kubectl command as shown in above documentation.

There is one more caveat - since the private docker registry we currently working on is not secured (uses http instead of https), we need to add following entry to **/etc/docker/daemon.json** of each worker node, so that worker nodes can pull container images from our private registry (10.2.252.210:5555 for instance):

```
{
  "insecure-registries" : ["10.2.252.210:5555"]
}
```

Then restart the docker daemon with:

```
ubuntu@worker1/2/3:~$ sudo systemctl daemon-reload
ubuntu@worker1/2/3:~$ sudo systemctl restart docker
```

Service Discovery

There are various ways to expose a service to external users in kubernetes, including port-forwarding, the NodePort resource, the LoadBalancer resource and the Ingress Resource. Currently we are sticking to LoadBalancer for several reasons:

1. Port-forwarding and NodePort expose services using internal IP addresses instead of external ones.
2. We are able to provide our bare metal deployment with plenty of external IP addresses.
3. The annotations for ingress resources is tricky to set up for an application to work correctly.
4. The ingress controller only uses one IP address. While this is actually a desirable trait for cloud-hosted deployments, it could cause performance issues in bare metal deployments when used with MetalLB layer 2 mode load-balancing.

In traditional cloud-hosted clusters, when a LoadBalancer resource is created, it is automatically assigned with an external IP address. However this is not the case for bare metal deployed clusters. MetalLB is a popular open-source project aiming to solve this issue. One mode supported by MetalLB is layer 2 load-balancing, which relies on ARP/NDP. It is simple to set up, but may suffer from single-node bottlenecking, and potentially slow failover.
Follow the Installation Page to install MetalLB. Here the ConfigMap we provide is as follows:

```
# metallb-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
      - 10.2.252.102-10.2.252.120
```

After applying the above file using `kubectl apply -f metallb-config.yaml`, MetalLB starts watching for the creation of any LoadBalancer resource, and assigns the next available IP address from the pool we specified to that LoadBalancer. Try accessing the service by entering the assigned IP in a browser.

In certain cases we might want to change the IP address pool assigned to MetalLB. Note that simply reapplying the modified metallb-config.yaml file won't work, as it could break existing services. To force such refresh of IP rang, run:

```
kubectl delete po -n metallb-system --all
Kubectl apply -f metallb-config.yaml
```

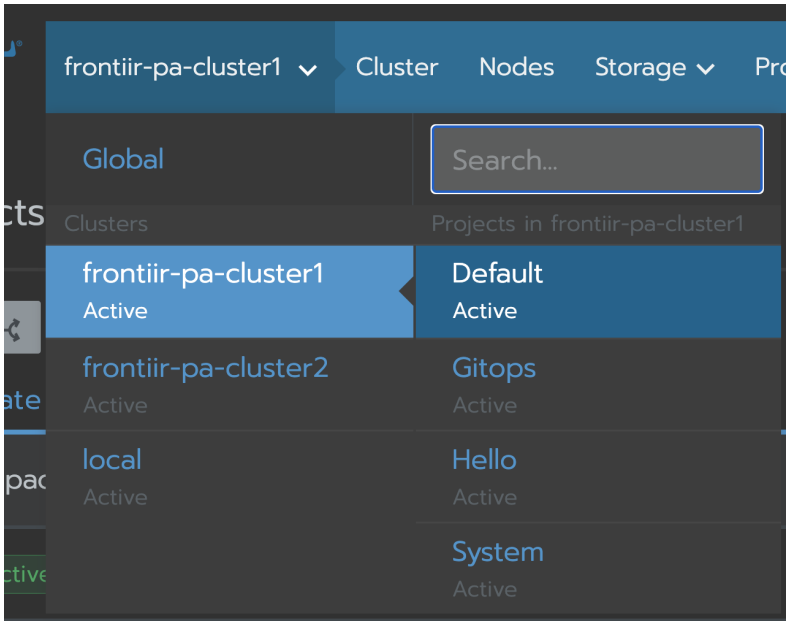
Multi-tenancy Support

To support basic soft multi-tenancy for our clusters, we take advantage of the Project and User concepts provided by Rancher. Below operations are carried out as Cluster Owner.

Creating a user: In the Cluster Manager's Global view, select "Users" under the "Security" dropdown menu. Click the "Add User" button at the top right, supply the new user with username and an initial password, then choose the global permissions for the user.

Creating a project: a project is a superset of namespaces, it can be used to group several related namespaces together. Each project can be viewed as a tenant of the cluster. In the cluster manager view of each specific cluster, select "Projects/Namespaces" at the top, and click the "Add Project" button at the top right. Aside from assigning the initial members that have access to the project, we can also define the resource quotas allowed for it, which prevents a single tenant of our cluster from using too much resources.

Assigning users to a project: To add a user to a specific project of a cluster, first select the project from the dropdown menu as shown below, then select the “Members” option in the top menu, and click the “Add Member” button on the top right corner.



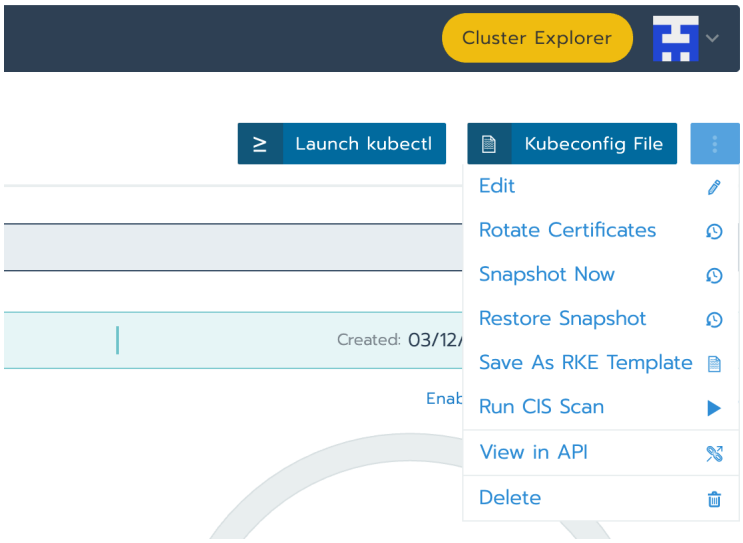
Enabling project network isolation: If the cluster uses canal as the network provider as suggested earlier, we may forbid communications between pods of different projects from “Edit Cluster” > “Kubernetes Options” > Select “Enabled” for “Project Network Isolation”.

Etcd Snapshot and Restore

Etcd is a crucial component for Kubernetes as it stores the entire state of the cluster: its configuration, specifications, and the statuses of the running workloads. When a user makes some unintentional change to the cluster and cannot easily undo it, etcd snapshots can be used to restore the cluster to the previous state. Moreover, etcd snapshots can also be used to recover the cluster to a healthy state when the etcd quorum is lost (disastrous scenario).

RKE clusters directly bootstrapped through the Rancher UI have the desirable feature of taking snapshot backups of etcd and storing them (locally or s3). Under “Edit Cluster” > “Advanced Options”, we can choose the frequency of recurring backup and the number of recent backups to keep. By default, the snapshots are stored at **/opt/rke/etcd-snapshots** of each etcd node.

It is recommended to backup the cluster before each time we want to make a change to our clusters. To manually initiate or restore a backup in the Rancher UI, use the dropdown menu as shown below inside each cluster’s cluster manager page:



Note that in an extreme case where all etcd nodes are broken, the etcd snapshots would also be lost, making it impossible to restore the cluster from snapshots. To prevent such a scenario, we need to either store the snapshots to s3 (supported by Rancher), or periodically backup the snapshots from an etcd node to an external machine.

Logging

Basic Logging with Command-line Interface

RKE clusters have different log locations than vanilla kubernetes clusters.
The logs of kubernetes system components are stored at **/var/lib/rancher/rke/log/**
The logs of each pod can be found at **/var/log/containers/**

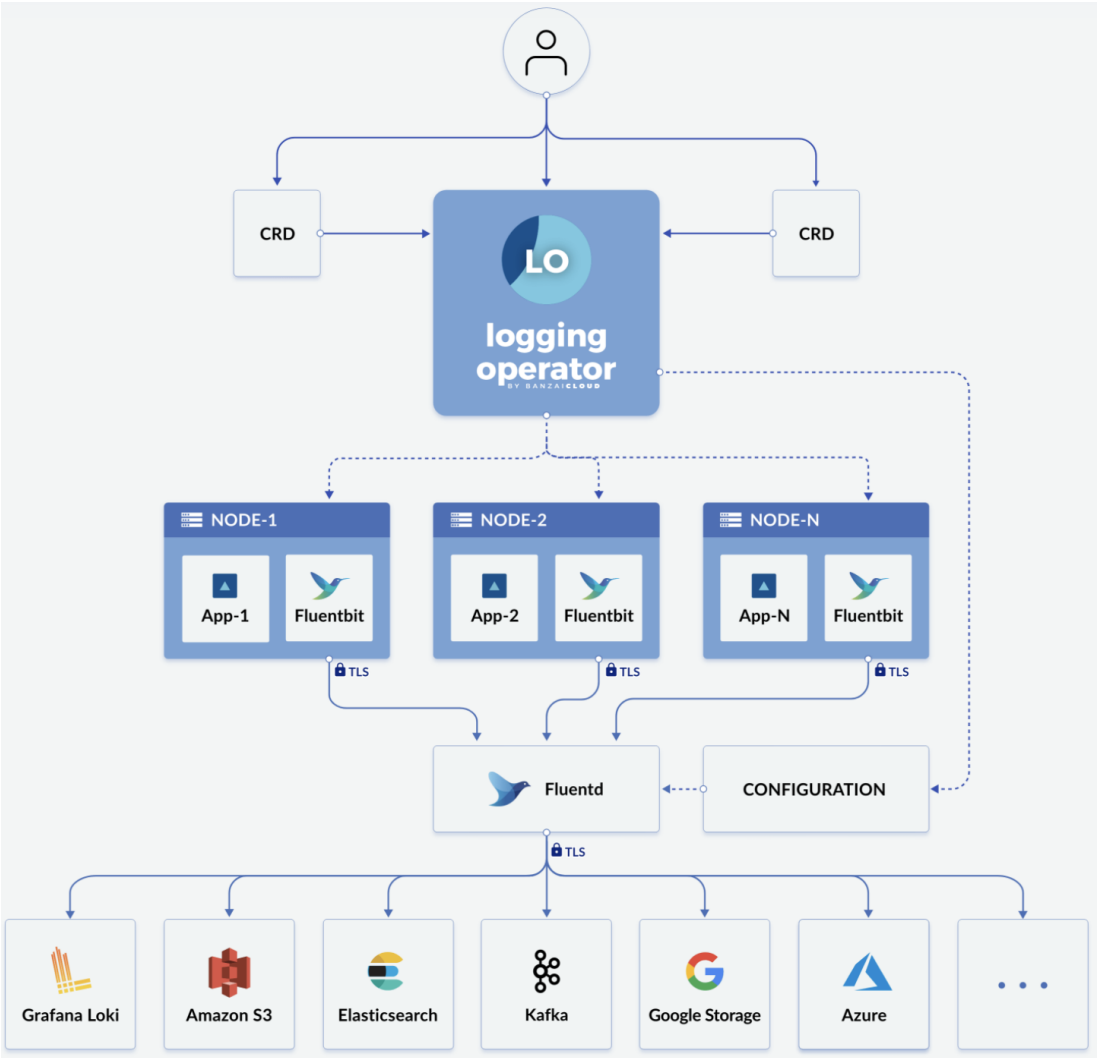
Below is an example of checking the etcd logs of our cluster:

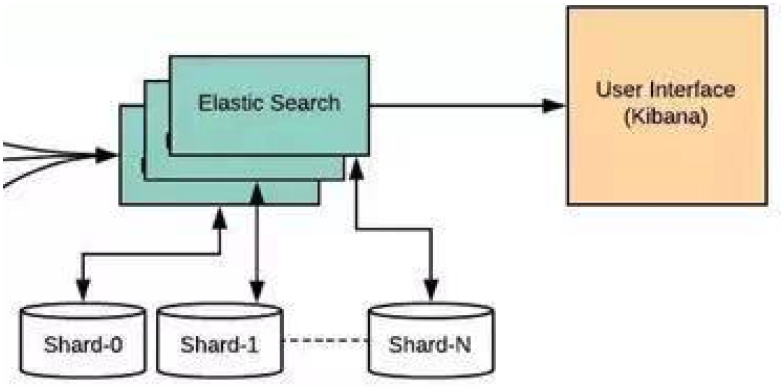
```
ubuntu@master1:/var/lib/rancher/rke/log$ ls
etcd_1dc7a1837ade9aa54d56e185678b82858843fc74c2cbd91983da9e7c3ea85470.log
kube-apiserver_14f13eb430dd0db46e9ceda1305e2c48a0cb5b1443ef4ea6015bf153a1a10bea.log
kube-controller-manager_d9d1a402ec200f9f4f91eb87a27bfe0589b3159ced33e1cc14f92c3b792
565d8.log
kubelet_a78153bfb9da6b7026312fe0af569355b95e43c464c029305d71b2bafebbf621.log
kube-proxy_d6af6f08b4fc9f3fd85bc61b43404e180e73fb33d86de825c3da56c545eb41cd.log
kube-scheduler_90f6a27d2a517d40e60b64c86da7745ad7803e519c8d109df526805e6108036d.log

ubuntu@master1:/var/lib/rancher/rke/log$ sudo tail -f
etcd_1dc7a1837ade9aa54d56e185678b82858843fc74c2cbd91983da9e7c3ea85470.log
{"log":"2021-03-25 15:46:44.375766 I | etcdserver/api/etcdhttp: /health OK (status
code 200)\n","stream":"stderr","time":"2021-03-25T15:46:44.37608045Z"}
{"log":"2021-03-25 15:46:59.391249 I | etcdserver/api/etcdhttp: /health OK (status
```

```
code 200)\n","stream":"stderr","time":"2021-03-25T15:46:59.391769565Z"}
{"log":"2021-03-25 15:47:11.667011 I | mvcc: store.index: compact
2533363\n","stream":"stderr","time":"2021-03-25T15:47:11.667540124Z"}
{"log":"2021-03-25 15:47:11.688020 I | mvcc: finished scheduled compaction at
2533363 (took
17.970738ms)\n","stream":"stderr","time":"2021-03-25T15:47:11.68824635Z"}
```


Advanced Logging with the EFK Stack

A centralized, cluster-level logging system can help us quickly analyze the logs produced by all pods running on our cluster. A common solution is based on the EFK stack (Elasticsearch, Fluentd and Kibana). Below is the architecture of the logging solution (note that Kibana should appear under Elasticsearch in the diagram; reference: Rancher logging documentation):

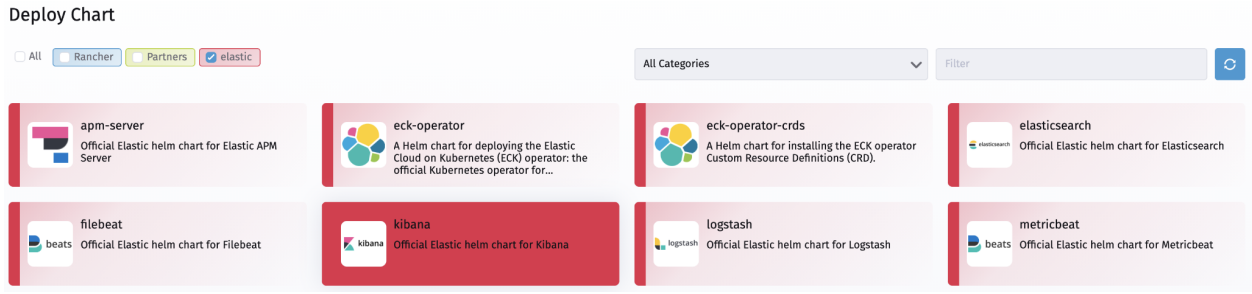




First we deploy the Logging Operator and Fluentbit/Fluentd as shown in the above diagram. Go to “Cluster Explorer” > “Apps and Marketplace” > “Charts”, then install the “Logging” chart provided by the Rancher helm chart repository. There should appear a new “Logging” option

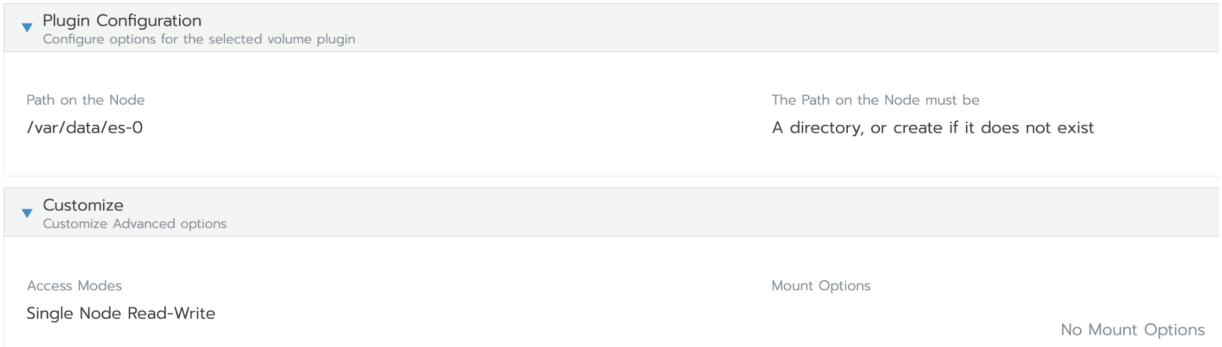
under the “Cluster Explorer” page. All the new resources will be created under the **cattle-logging-system** namespace.

Next deploy Elasticsearch and Kibana. Under “Cluster Explorer” > “Apps and Marketplace” > “Chart Repositories”, select “Create” and add the official elastic helm chart repository from there by entering the index URL: **https://helm.elastic.co**. Now go back to “Charts” and there should appear a new checkbox “elastic”, we can install Elasticsearch and Kibana from here:



Several notes on deployment:

- Deploy both Elasticsearch and Kibana to the cattle-logging-system namespace for consistency.
- By default Elasticsearch deploys a StatefulSet of 3 replicas, where each replica claims a persistent volume of size 30Gi. Customize these values based on your needs inside “Values YAML” before installing, otherwise the new pods will appear “pending” due to insufficient resources.
- Persistent volumes can be manually provisioned under “Cluster Manager” > “Storage” > “Persistent Volumes”. At the time of writing, we provisioned three persistent volumes for each replica of Elasticsearch using **HostPath** (this may not be our final PV solution). Make sure to select “Single Node Read-Write” as the Access Mode. Below is a sample of our configuration:



- Suppose the HostPath **/var/data/es-0** is assigned to worker1, then inside worker1 we need to manually do: `ubuntu@worker1:~$ sudo chown 1000:1000 /var/data/es-0` and so on for the other two replicas.
- It is recommended to deploy the Kibana service as a NodePort/LoadBalancer instead of ClusterIP. ClusterIP requires port-forwarding which often disconnects when testing.
- Consider reducing the CPU requests of kibana from 1 to 0.5 to save CPU resources.
- Fluent-bit config needs to be modified to eliminate warning/error logs related to incorrect time format coming from cattle-logging-system itself (add detailed configmap below).

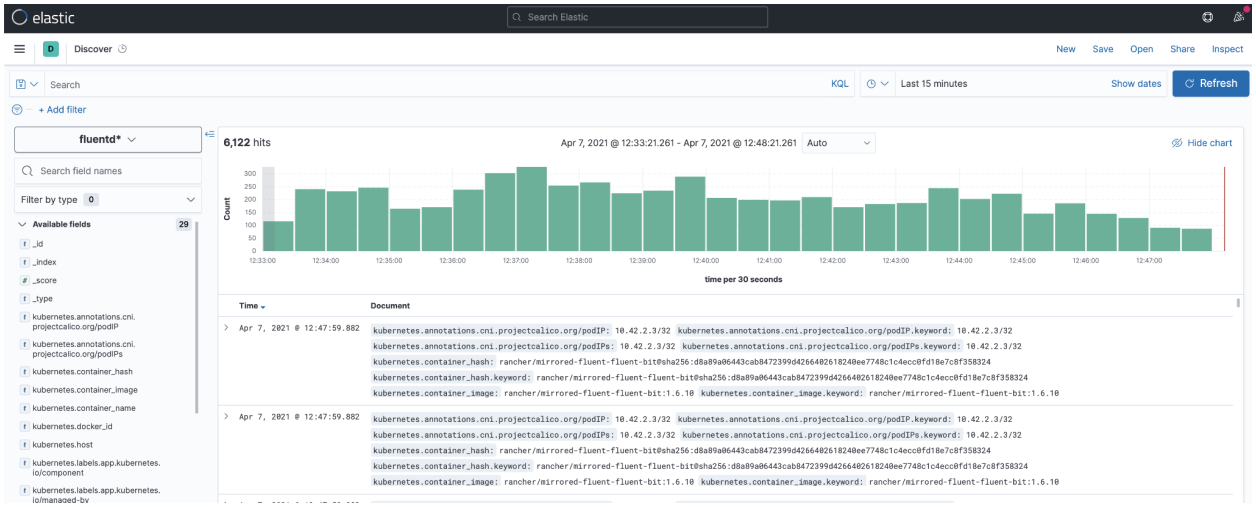
Now that we have deployed the EFK stack on our cluster, a few more steps are needed to create the logging pipeline. First we define a Elasticsearch ClusterOutput by applying below yaml to the cluster:

```
apiVersion: logging.banzaicloud.io/v1beta1
kind: ClusterOutput
metadata:
    name: "frontiir-k8s-es"
    namespace: "cattle-logging-system"
spec:
    elasticsearch:
      host: <ClusterIP of elasticsearch-master service>
      port: 9200
      scheme: http
```

Next we configure all logs to go to the above ClusterOutput by creating a corresponding ClusterFlow:

```
apiVersion: logging.banzaicloud.io/v1beta1
kind: ClusterFlow
metadata:
    name: "all-logs"
    namespace: "cattle-logging-system"
spec:
  globalOutputRefs:
    - "frontiir-k8s-es"
```

Finally open the Kibana service in the browser, go to “Discover”. Create an Elasticsearch index pattern named “fluentd*”, then select “time” as the time field. We should be able to see all logs of our cluster now, more filtering can be performed in Kibana using KQL:



By default, logging indices are kept for seven days in elasticsearch. This retention period can be adjusted (reference).

Monitoring

The default Rancher UI supports basic functionalities such as monitoring the health of nodes/core kubernetes components and reporting the events related to each kubernetes object. For advanced monitoring, Rancher integrates Prometheus and Grafana. Go to “Cluster Explorer” > “Apps and Marketplace”, then install the “Monitoring” chart provided by the Rancher helm chart repository.The “Monitoring” feature should appear in the upper-left menu of “Cluster Explorer” page. New objects related to monitoring will be created under the **cattle-monitoring-system** and **cattle-dashboards** namespaces.

Grafana UI

Grafana allows us to easily visualize the Prometheus metrics with dashboards, we can access it in “Cluster Explorer” > “Monitoring” > “Overview” > “Grafana”. Some of the useful dashboards include “etcd”, “Kubernetes / Computer Resources / Namespace(Workload)” and “Kubernetes / Computer Resources / Namespace(Networking)”.

Alerts

TODO: slack webhook example and how it can be applied to rocket chat; email needs smtp configured.
Doc: https://rancher.com/docs/rancher/v2.5/en/monitoring-alerting/

Failure Recovery

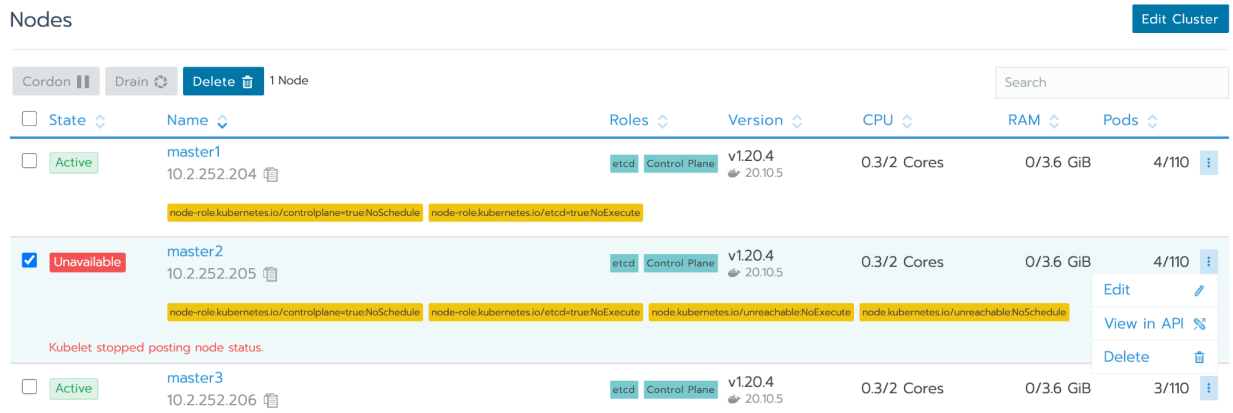
Failure Recovery of Downstream RKE Cluster

A node failure can be classified into temporary failure and permanent failure. Examples of temporary failures include machine reboot, network instability, docker containers restart, etc. RKE is capable of automatically handling such temporary failures by performing regular health checks under the hood. Once the node is back online, the cluster resumes normal operation without the need of any human intervention.

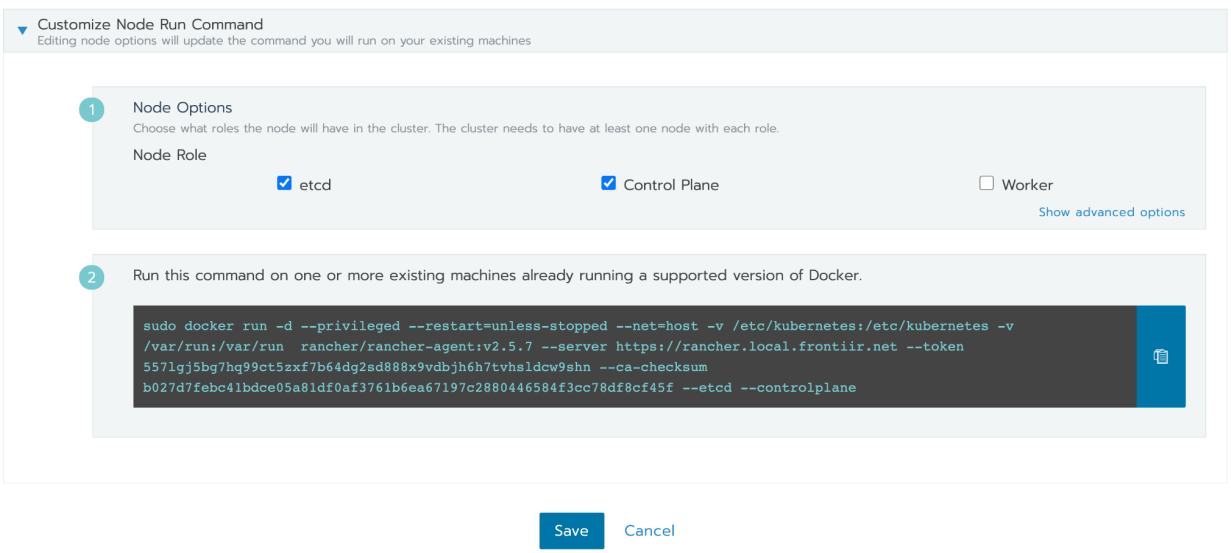
Permanent node failures could also happen, for example, the physical machine that hosts nodes is broken. In such cases, we would need to remove the previous broken nodes from the RKE cluster, then add back new nodes with corresponding roles. This can be easily performed in the Rancher UI.

Note: An important assumption is that the cluster still maintains its etcd quorum. Without a working etcd quorum, the cluster becomes read-only, write operations such as removing/adding nodes become impossible. That being said, it's still possible to restore the cluster with valid etcd snapshots. More experiments need to be carried out on losing the quorum.

Below shows a scenario where we simulate node failure by first destroying the VM of master2. After a while the UI reports marks it as unavailable, and we delete the node:



After removing the broken node, we need to prepare a clean node, and do two things on it: resolve the Rancher server domain, and install Docker on it. Steps can be found in the "Pre-configuration and Dependency Installation" section. Then we can add the node from UI: click "Edit Cluster", scroll to the bottom and select the role(s) to assign to the new node, copy the docker command generated by the UI and run it in the new node:



After registering the new node, make sure to install kubectl on it and copy the kubeconfig file to the default location (~/.kube/config).

Failure Recovery of Upstream Rancher Cluster

Since Rancher v2.5, it is recommended to backup and restore Rancher using the **rancher-backup** operator. Before installing the operator, make sure the Rancher cluster is able to provision a persistent volume of at least 2 GiB with single node read-write access mode.

From the **"local"** cluster, go to "Cluster Explorer" > "Apps and Marketplace", then install the "Rancher Backups" app with the default storage location correctly set. The "Rancher Backups" feature should appear in the upper-left menu of "Cluster Explorer" page. From there you may manually create backups (one-time or recurring) and restores for Rancher. Refer to the official documentation for more details.

Appendix

Below we describe an alternative way of setting up a downstream kubernetes cluster using **kubeadm**, a tool recommended by the official kubernetes community. The cluster can be imported to Rancher, however imported clusters cannot enjoy some of the benefits provided by Rancher, such as simple etcd snapshots and certificate rotation, security scan. We also have some trouble getting MetalLB correctly working on a kubeadm cluster, so right now we prioritize the RKE clusters.

A. Install HA k8s cluster through kubeadm **(version 1.20.2)**

Requirements

At least 6 nodes are required for the setup, with 3 being the master (control plane) nodes and 3 being the worker nodes. Each node should meet below requirements:

1. Ubuntu 16.04+ or CentOS 7+ (here we use Ubuntu 18.04 x86_64)
2. At least 2GB RAM
3. At least 2 cores CPU
4. Full network connectivity
5. Unique hostname, MAC_address and product_uuid
6. Certain ports are available (check below url for details)
7. Swap and firewall are disaled

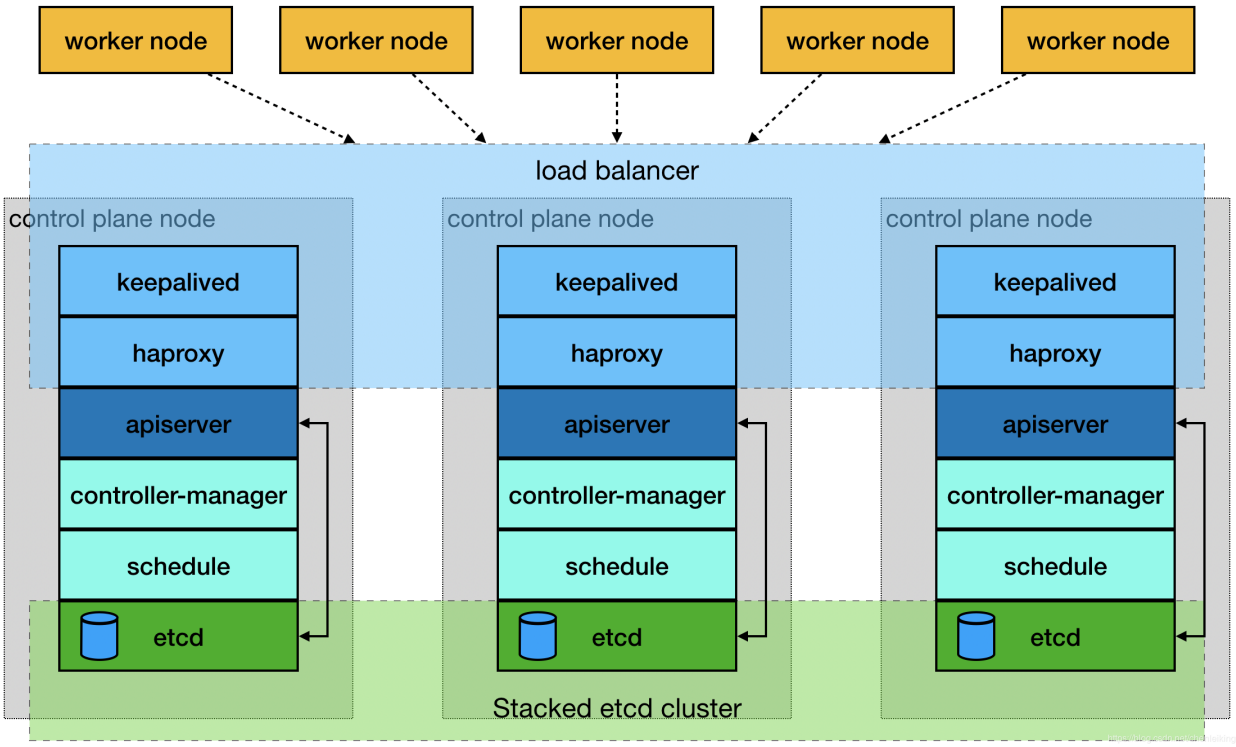
To get a minimum viable sample up and running, we use the kubeadm tool for bootstrapping the cluster. More details can be found in this kubeadm installation guide.

Below is the detailed information for each node in the PA office testbed:

IP Address	Host Name	Role
10.2.251.201	k8s-master1	Keepalived & HAproxy, master, etcd

10.2.251.202	k8s-master2	Keepalived & HAproxy, master, etcd
10.2.251.203	k8s-master3	Keepalived & HAproxy, master, etcd
10.2.170.171	k8s-worker1	worker
10.2.85.171	k8s-worker2	worker
10.2.0.171	k8s-worker3	worker
10.2.252.110		Virtual IP address

Architecture



Above diagram shows the topology of the HA Kubernetes cluster. Note that here we choose a stacked topology where the distributed data storage cluster provided by etcd is stacked on top of the cluster formed by the nodes managed by kubeadm that run control plane components. Such topology requires less nodes when compared to the external etcd topology, but suffers from the risk of failed coupling. Relevant discussions can be found in [this page describing options for HA topology.]

The kube-apiserver instances are exposed to worker nodes using a load balancer, shown in light blue in above diagram. Note that the load balancer itself also needs to be highly available. As this is not taken care of by kubernetes or kubeadm, we handle it separately using a common

solution based on Keepalived + Haproxy, which provides load balancing from a virtual IP. More information can be found here.

Installation and Configuration (running all commands as root user)

a. Configure /etc/hosts and run prerequisite commands and install docker runtime(for all worker nodes and master nodes)

Add below records to master node

```
root@all_node$ cat >> /etc/hosts << EOF
10.2.251.110 cluster.kube.com
10.2.251.201 k8s-master1
10.2.251.202 k8s-master2
10.2.251.203 k8s-master3
EOF
```

Add below record to worker node

```
root@all_node$ cat >> /etc/hosts << EOF
10.2.251.110 cluster.kube.com
EOF
```

Run below commands on all nodes

```
root@all_node$ swapoff -a      # this can close the swap temporarily
                # use 'swapon --show' to check swap
# Use below command to stop the firewall if needed
root@all_node$ ufw disable
# load br_netfilter first (by default this is disabled on PA office VMs)
root@all_node$ modprobe br_netfilter

# ensure net.bridge.bridge-nf-call-iptables is set to 1 in sysctl config
root@all_node$ cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
br_netfilter
EOF

root@all_node$ cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF

# use below cmd to check the rule
# sudo sysctl --system

# Install the docker run time
```

```
root@all_node$ apt-get update && apt-get install -y \
  apt-transport-https ca-certificates curl software-properties-common gnupg2

root@all_node$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
apt-key --keyring /etc/apt/trusted.gpg.d/docker.gpg add -

root@all_node$ add-apt-repository \
  "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) \
  stable"

root@all_node$ apt-get update && apt-get install -y \
  containerd.io=1.2.13-2 \
  docker-ce=5:19.03.11~3-0~ubuntu-$(lsb_release -cs) \
  docker-ce-cli=5:19.03.11~3-0~ubuntu-$(lsb_release -cs)

# Set up the Docker daemon
root@all_node$ cat <<EOF | sudo tee /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF

root@all_node$ mkdir -p /etc/systemd/system/docker.service.d
# Start the docker service
root@all_node$ systemctl enable docker
root@all_node$ systemctl daemon-reload
root@all_node$ systemctl restart docker

# Install kubeadm, kubectl, kubelet
root@all_node$ apt-get update && apt-get install -y apt-transport-https curl
root@all_node$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo
apt-key add -
root@all_node$ cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
root@all_node$ apt-get update
root@all_node$ apt-get install -y kubelet kubeadm kubectl

# this command is used to mark a package as held up, will not have auto update
root@all_node$ apt-mark hold kubelet kubeadm kubectl
```

b. Configure HA (running on all master nodes only)

1. Configure Keepalived

```
# Install keepalive
root@all_master_node$ cat >> /etc/sysctl.conf << EOF
net.ipv4.ip_forward = 1
EOF

root@all_master_node$ sysctl -p

root@all_master_node$ apt install -y keepalived

root@all_master_node$ cat > /etc/keepalived/keepalived.conf << EOF
! Configuration File for keepalived

global_defs {
   router_id LVS_DEVEL
}

vrrp_script check_haproxy {
    script "killall -0 haproxy"
    interval 3
    weight -2
    fall 10
    rise 2
}

vrrp_instance VI_1 {
    state {STATE}
    interface {INTERFACE}
    virtual_router_id {VIRTUAL_ROUTER_ID}
    priority {PRIORITY}
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass {AUTH_PASS}
    }
    virtual_ipaddress {
        {VIP}
    }
    track_script {
        check_haproxy
    }
}
EOF
root@all_master_node$ service keepalived start
```

Parameter:

{STATE}: The state of current node, should be **MASTER** or **BACKUP**

{INTERFACE}: The network interfaces used to communicate with other nodes, please use `**ifconfig**` or `**ip addr**` to check.
{VIRTUAL_ROUTER_ID}: Id for virtual_router, same router id can communicate with each other
{PRIORITY}: The device with the highest priority within the group becomes the primary
{AUTH_PASS}: password for authentication, can be manually set
{VIP}: virtual ip address

Below is my version of config file:

```
global_defs {
   router_id LVS_DEVEL
}

vrrp_script check_haproxy {
    script "killall -0 haproxy"
    interval 3
    weight -2
    fall 10
    rise 2
}

vrrp_instance VI_1 {
    state MASTER
    interface eth0
    virtual_router_id 96
    priority 250
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 35f18af7190d51c9f7f78f37300a0cbd
    }
    virtual_ipaddress {
        10.2.251.110
    }
    track_script {
        check_haproxy
    }
}
```

This is the setting for k8s-master1 node, for the other two master nodes, what only need to be changed are **state** and **priority**.

c. HAproxy install and configuration (for all master nodes)

```
# Install and configure haproxy
root@all_master_node$ cat >> /etc/sysctl.conf << EOF
net.ipv4.ip_nonlocal_bind = 1
```

```
EOF

root@all_master_node$ sysctl -p

root@all_master_node$ apt install -y haproxy

root@all_master_node$ cat > /etc/haproxy/haproxy.cfg << EOF
global
    # to have these messages end up in /var/log/haproxy.log you will
    # need to:
    #
    # 1) configure syslog to accept network log events.  This is done
    #    by adding the '-r' option to the SYSLOGD_OPTIONS in
    #    /etc/sysconfig/syslog
    #
    # 2) configure local2 events to go to the /var/log/haproxy.log
    #   file. A line like the following can be added to
    #   /etc/sysconfig/syslog
    #
    #    local2.*                       /var/log/haproxy.log
    #
    log         127.0.0.1 local2

    chroot      /var/lib/haproxy
    pidfile     /var/run/haproxy.pid
    maxconn     4000
    user        haproxy
    group       haproxy
    daemon

    # turn on stats unix socket
    stats socket /var/lib/haproxy/stats

defaults
    mode                    tcp
    log                     global
    option                  redispatch
    retries                 3


listen https-apiserver
    bind    {VIP}:{PORT}
    mode        tcp
```

```
    balance     roundrobin
    timeout server 15s
    timeout connect 15s
    server  {SERVER HOSTNAME} {SERVER IP}:6443 check port 6443 inter 5000
fall 5
    ...

EOF

root@all_master_node$ service haproxy start
```

Below is a sample of HAproxy configuration file

```
root@all_master_node$ cat > /etc/haproxy/haproxy.cfg << EOF
global
    # to have these messages end up in /var/log/haproxy.log you will
    # need to:
    #
    # 1) configure syslog to accept network log events.  This is done
    #    by adding the '-r' option to the SYSLOGD_OPTIONS in
    #    /etc/sysconfig/syslog
    #
    # 2) configure local2 events to go to the /var/log/haproxy.log
    #   file. A line like the following can be added to
    #   /etc/sysconfig/syslog
    #
    #    local2.*                       /var/log/haproxy.log
    #
    log         127.0.0.1 local2

    chroot      /var/lib/haproxy
    pidfile     /var/run/haproxy.pid
    maxconn     4000
    user        haproxy
    group       haproxy
    daemon

    # turn on stats unix socket
    stats socket /var/lib/haproxy/stats

defaults
    mode                    tcp
    log                     global
    option                  redispatch
    retries                 3
```

```
listen https-apiserver
    bind 10.2.251.110:6443
    mode        tcp
    balance     roundrobin
    timeout server 15s
    timeout connect 15s
    server  k8s-master1 10.2.251.201:6443 check port 6443 inter 5000 fall 5
    server  k8s-master2 10.2.251.202:6443 check port 6443 inter 5000 fall 5
    server  k8s-master3 10.2.251.203:6443 check port 6443 inter 5000 fall 5

EOF
```

After successfully installing the Keepalived + HAproxy, can `ping` the VIP from anywhere inside the private network, even if the master node is down, the VIP is still responsive. You can also use below command to check the status of Keepalived + HAproxy

```
root@all_master_node$ systemctl status keepalived
root@all_master_node$ systemctl status haproxy
```

d. Initial the Kubernetes (on one master node)

Using below configuration file to initial a new cluster, I wrote it into a new file called **kubeadm-config.yml**

```
apiVersion: kubeadm.k8s.io/v1beta1
kind: ClusterConfiguration
kubernetesVersion: v1.20.2
apiServer:
  certSANs:
    - "cluster.kube.com"
controlPlaneEndpoint: "cluster.kube.com:6443"
networking:
  podSubnet: "10.244.0.0/16"
```

Then running the command below(please notice it runs under **ubuntu** user)

```
ubuntu@node1$ sudo kubeadm init --config=kubeadm-config.yaml --upload-certs
```

Wait for several minutes until the initial configuration is done. If there are errors during the process, then running below command to clear the installation

```
ubuntu@node1$ sudo kubeadm reset -f
```

e. Add other master nodes and worker nodes

After you successfully run the `kubeadm init` command. You will get the printing below

```
# Below is all that using to join the cluster
To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

You can now join any number of the control-plane node running the following command
on each as root:

  kubeadm join cluster.kube.com:6443 --token 68yz08.1oeuf9sxjbf8o3jw \
    --discovery-token-ca-cert-hash
sha256:3c2fe5e68c81b1112f557b4dde63a2b24ce9b2089a0d5650210388fc7ff753d0 \
    --control-plane --certificate-key
64ed079cc2dab8d08a6a028fce323abf277c85cdb8e588a50e06208e6b667983

Please note that the certificate-key gives access to cluster sensitive data, keep
it secret!
As a safeguard, uploaded-certs will be deleted in two hours; If necessary, you can
use
"kubeadm init phase upload-certs --upload-certs" to reload certs afterward.

Then you can join any number of worker nodes by running the following on each as
root:

kubeadm join cluster.kube.com:16443 --token 68yz08.1oeuf9sxjbf8o3jw \
    --discovery-token-ca-cert-hash
sha256:3c2fe5e68c81b1112f557b4dde63a2b24ce9b2089a0d5650210388fc7ff753d0
```

Follow the command above to Add other master nodes and worker nodes, **please note that the certificate-key will expire in TWO hours**, using the command in the instruction to renew the key, after this step, your `kubectl` command will basically work.

f. Install the CNI plugin

```
ubuntu@master_node1$ curl -O
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
kubectl apply -f kube-flannel.yml
```

g. Add more nodes in the future

First run all prerequisite commands according to previous steps, please note that add worker node and add master node is different. Then, running below command on a current master to get the discovery token

```
root@master_node1/2/3$ kubeadm token create --print-join-command
```

For adding a worker node, directly run the command generated through above command on the new worker node you want to add

For adding a master node, please run below command on the current master node to get the new cert (cert will expire in 2 hours), and use new cert to join to cluster

```
root@master_node1/2/3$ kubeadm init phase upload-certs --upload-certs

# after getting the new cert, replace certificate key with the latest cert you generated
root@new_master_node$ kubeadm join cluster.kube.com:6443 --token 68yz08.1oeuf9sxjbf8o3jw \
    --discovery-token-ca-cert-hash sha256:3c2fe5e68c81b1112f557b4dde63a2b24ce9b2089a0d5650210388fc7ff753d0 \
    --control-plane --certificate-key <NEW_CERT_KEY>
```

System log monitoring

a. Load balancer monitoring

You can through `systemctl status` to monitor the both Keepalived and HAproxy status, also, these logs can be check in **/var/log/haproxy** and **/var/log/syslog(Keepalived)**

b. Cluster Monitoring

Using below command to check the nodes status

```
ubuntu@master_node$ kubectl get nodes
```

Using below command to check detail status of a node

```
ubuntu@master_node$ kubectl describe node <NodeName>
```

Using below command to check the system component status

```
ubuntu@master_node$ kubectl get pods -n kube-system
```

Using below command to track the system pod status

```
ubuntu@master_node$ kubectl logs <PodName> -n kube-system
```

Reference

1. Official setup docs:

https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/
2. Reference blog:
https://www.kubeclusters.com/docs/How-to-Deploy-a-Highly-Available-kubernetes-Cluster-with-Kubeadm-on-CentOS7
3. Load balancer instruction:
https://github.com/kubernetes/kubeadm/blob/master/docs/ha-considerations.md#options-for-software-load-balancing
4. Reference blog:
https://www.liquidweb.com/kb/how-to-install-kubernetes-using-kubeadm-on-ubuntu-18/
5. External etcd reference blog:
https://medium.com/velotio-perspectives/demystifying-high-availability-in-kubernetes-using-kubeadm-3d83ed8c458b#:~:text=In%20a%20single%20master%20setup,entire%20cluster%20will%20be%20lost

B. Common k8s command

kubectl context and configuration

```
kubectl config view # Show Merged kubeconfig settings.
kubectl config view -o jsonpath='{.users[].name}'    # display the first user
kubectl config view -o jsonpath='{.users[*].name}'   # get a list of users
kubectl config get-contexts                          # display list of contexts
kubectl config current-context                       # display the current-context
kubectl config use-context my-cluster-name           # set the default context to
my-cluster-name
```

kubectl apply and create object

```
kubectl apply -f ./my-manifest.yaml            # create resource(s)
kubectl apply -f ./my1.yaml -f ./my2.yaml      # create from multiple files
kubectl apply -f ./dir                         # create resource(s) in all manifest
files in dir
kubectl apply -f https://git.io/vPieo          # create resource(s) from url
kubectl create deployment nginx --image=nginx  # start a single instance of nginx

# create a Job which prints "Hello World"
kubectl create job hello --image=busybox -- echo "Hello World"

# create a CronJob that prints "Hello World" every minute
kubectl create cronjob hello --image=busybox   --schedule="*/1 * * * *" -- echo
"Hello World"

kubectl explain pods                           # get the documentation for pod
```

```
manifests
```

kubectl get and view resources

```
# Get commands with basic output
kubectl get services                          # List all services in the namespace
kubectl get pods --all-namespaces             # List all pods in all namespaces
kubectl get pods -o wide                      # List all pods in the current
namespace, with more details
kubectl get deployment my-dep                 # List a particular deployment
kubectl get pods                              # List all pods in the namespace
kubectl get pod my-pod -o yaml                # Get a pod's YAML

# Describe commands with verbose output
kubectl describe nodes my-node
kubectl describe pods my-pod

# List Services Sorted by Name
kubectl get services --sort-by=.metadata.name

# List pods Sorted by Restart Count
kubectl get pods --sort-by='.status.containerStatuses[0].restartCount'

# List PersistentVolumes sorted by capacity
kubectl get pv --sort-by=.spec.capacity.storage

# Show labels for all pods (or any other Kubernetes object that supports labelling)
kubectl get pods --show-labels

# Compares the current state of the cluster against the state that the cluster
would be in if the manifest was applied.
kubectl diff -f ./my-manifest.yaml
```

kubectl update resources

```
kubectl set image deployment/frontend www=image:v2               # Rolling update
"www" containers of "frontend" deployment, updating the image
kubectl rollout history deployment/frontend                      # Check the
history of deployments including the revision
kubectl rollout undo deployment/frontend                         # Rollback to the
previous deployment
kubectl rollout undo deployment/frontend --to-revision=2         # Rollback to a
specific revision
kubectl rollout status -w deployment/frontend                    # Watch rolling
update status of "frontend" deployment until completion
kubectl rollout restart deployment/frontend                      # Rolling restart
```

```
of the "frontend" deployment

cat pod.json | kubectl replace -f -                              # Replace a pod
based on the JSON passed into std

# Force replace, delete and then re-create the resource. Will cause a service
outage.
kubectl replace --force -f ./pod.json

# Create a service for a replicated nginx, which serves on port 80 and connects to
the containers on port 8000
kubectl expose rc nginx --port=80 --target-port=8000

kubectl label pods my-pod new-label=awesome                      # Add a Label
kubectl annotate pods my-pod icon-url=http://goo.gl/XXBTWq       # Add an
annotation
kubectl autoscale deployment foo --min=2 --max=10                # Auto scale a
deployment "foo"
```

kubectl scale resources

```
kubectl scale --replicas=3 rs/foo                                 # Scale a
replicaset named 'foo' to 3
kubectl scale --replicas=3 -f foo.yaml                            # Scale a
resource specified in "foo.yaml" to 3
kubectl scale --current-replicas=2 --replicas=3 deployment/mysql  # If the
deployment named mysql's current size is 2, scale mysql to 3
kubectl scale --replicas=5 rc/foo rc/bar rc/baz                   # Scale multiple
replication controllers
```

kubectl delete resource

```
kubectl delete -f ./pod.json                                              # Delete
a pod using the type and name specified in pod.json
kubectl delete pod,service baz foo                                        # Delete
pods and services with same names "baz" and "foo"
kubectl delete pods,services -l name=myLabel                              # Delete
pods and services with label name=myLabel
kubectl -n my-ns delete pod,svc --all                                      # Delete
all pods and services in namespace my-ns,
```

kubectl interacting with running pods

```
kubectl logs my-pod                                 # dump pod logs (stdout)
kubectl logs -l name=myLabel                        # dump pod logs, with label
```

```
name=myLabel (stdout)
kubectl logs my-pod --previous                      # dump pod logs (stdout) for a
previous instantiation of a container
kubectl logs my-pod -c my-container                 # dump pod container logs
(stdout, multi-container case)
kubectl logs -l name=myLabel -c my-container        # dump pod logs, with label
name=myLabel (stdout)
kubectl logs my-pod -c my-container --previous      # dump pod container logs
(stdout, multi-container case) for a previous instantiation of a container
kubectl logs -f my-pod                              # stream pod logs (stdout)
kubectl logs -f my-pod -c my-container              # stream pod container logs
(stdout, multi-container case)
kubectl logs -f -l name=myLabel --all-containers    # stream all pods logs with
label name=myLabel (stdout)
kubectl run -i --tty busybox --image=busybox -- sh  # Run pod as interactive shell
kubectl run nginx --image=nginx -n
mynamespace                                         # Run pod nginx in a specific
namespace

# Run pod nginx and write its spec into a file called pod.yaml
kubectl run nginx --image=nginx --dry-run=client -o yaml > pod.yaml

kubectl attach my-pod -i                            # Attach to Running Container
kubectl port-forward my-pod 5000:6000               # Listen on port 5000 on the
local machine and forward to port 6000 on my-pod
kubectl exec my-pod -- ls /                         # Run command in existing pod
(1 container case)
kubectl exec --stdin --tty my-pod -- /bin/sh        # Interactive shell access to a
running pod (1 container case)
kubectl exec my-pod -c my-container -- ls /         # Run command in existing pod
(multi-container case)
kubectl top pod POD_NAME --containers               # Show metrics for a given pod
and its containers
kubectl top pod POD_NAME --sort-by=cpu              # Show metrics for a given pod
and sort it by 'cpu' or 'memory'
```

kubectl interacting with deployment and svc

```
kubectl logs deploy/my-deployment                       # dump Pod logs for a
Deployment (single-container case)
kubectl logs deploy/my-deployment -c my-container       # dump Pod logs for a
Deployment (multi-container case)

kubectl port-forward svc/my-service 5000                # listen on local port
5000 and forward to port 5000 on Service backend
kubectl port-forward svc/my-service 5000:my-service-port  # listen on local port
```

```
5000 and forward to Service target port with name <my-service-port>

kubectl port-forward deploy/my-deployment 5000:6000       # listen on local port
5000 and forward to port 6000 on a Pod created by <my-deployment>
kubectl exec deploy/my-deployment -- ls                   # run command in first
Pod and first container in Deployment (single- or multi-container cases)
```

kubectl interacting with node and cluster

```
kubectl cordon my-node              # Mark my-node as unschedulable
kubectl drain my-node               # Drain my-node in preparation for maintenance
kubectl uncordon my-node            # Mark my-node as schedulable
kubectl top node my-node            # Show metrics for a given node
kubectl cluster-info                # Display addresses of the master and services
kubectl cluster-info dump           # Dump current cluster state to stdout
kubectl cluster-info dump --output-directory=/path/to/cluster-state   # Dump
current cluster state to /path/to/cluster-state

# If a taint with that key and effect already exists, its value is replaced as
specified.
kubectl taint nodes foo dedicated=special-user:NoSchedule
```

For more command, please see [kubectl-cheatsheet]