

Kubernetes on Oracle Cloud Infrastructure

Overview and Manual Deployment Guide

ORACLE WHITE PAPER | FEBRUARY 2018





Disclaimer

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.



Table of Contents

Target Audience	4
Introduction	4
Overview of Kubernetes	4
Container Images	5
Application Deployment	5
Container Orchestration Features	6
Cluster Architecture for Kubernetes Manual Deployment on Oracle Cloud Infrastructure	7
etcd	8
Kubernetes Masters	8
Kubernetes Workers	9
Kubernetes Manual Deployment Guide for Oracle Cloud Infrastructure	9
Step 1: Create Oracle Cloud Infrastructure Resources	9
Step 2: Set Up a Certificate Authority and Create TLS Certificates	13
Step 3: Bootstrap an HA etcd Cluster	19
Step 4: Set Up RBAC	21
Step 5: Bootstrap an HA Kubernetes Control Plane	22
Step 6: Add a Worker	27
Step 7: Configure kubectl for Remote Access	34
Step 8: Deploy Kube-DNS	35
Step 9: Smoke Test	35
Appendix A: Security Rules	37



Target Audience

This document is intended for customers who are interested in learning how Kubernetes works by deploying it on Oracle Cloud Infrastructure or who have considerable experience with Kubernetes and want a baseline deployment process in order to create their own highly configured clusters.

This document assumes that the user is working from a computer running a macOS operating system and has an understanding of UNIX commands and file systems.

Users of this document should also be familiar with the fundamentals of the Oracle Cloud Infrastructure. For information, go to <https://docs.us-phoenix-1.oraclecloud.com/>. If this is the first time that you have used the platform, we recommend specifically the tutorial at <https://docs.us-phoenix-1.oraclecloud.com/Content/GSG/Reference/overviewworkflow.htm>.

Introduction

Kubernetes is the most popular container orchestration tool available today. Although the Kubernetes open-source project is still young and experiencing tremendous growth, when it is deployed properly Kubernetes can be a reliable tool for running container workloads in production.

This document presents a starting point for deploying a secure, highly available Kubernetes cluster on Oracle Cloud Infrastructure. The cluster created from the instructions in this document might be sufficient for your needs. However, if you want to configure your cluster beyond what is presented here, you'll need supplementary materials and white papers to address the various customization options and updates to Kubernetes.

Kubernetes is an incredibly fast-moving project, with frequent new releases and bug fixes. Accordingly, this document addresses deploying Kubernetes version 1.6 (the stable build at the time of writing). Future documents will address the process to upgrade the cluster made in this document.

Overview of Kubernetes

This section provides a brief introduction to Kubernetes. If you are already familiar with Kubernetes, you can skip this section.

Kubernetes is the most popular container orchestration tool available and is maintained by one of the fastest-growing open-source communities. The Kubernetes project originated within Google, a long-time user of massive numbers of containers. To manage these containers well, they needed to develop a system for container orchestration. Kubernetes combines the lessons that Google



learned from years of container usage into a single tool with an array of features that make container orchestration simple and adaptable to the wide variety of use cases in the technology industry. Since it became open source in July 2015, the capabilities of Kubernetes have continued to grow. Issues and new feature requests are tracked on the public [GitHub project](#) with new major versions released approximately every two months.

Containers are designed to solve problems with traditional application deployment, such as missing dependencies when installing an application, or trouble deploying applications on specific OS versions. Container orchestrators aim to solve problems with scaling these applications.

Container Images

The application, with all of its dependencies, is kept in a *container image*. When run by a container engine, such as Docker, the container image runs as a container. The process of creating this container image for an application is known as *containerization*. Containerization is beneficial in cases where the application would interact poorly with other applications if deployed on the same machine. The container provides a level of isolation that, although not fully multi-tenant, can prevent applications from causing problems with other applications running on the same physical host. For example, containers simplify Java application deployment by bundling the application's dependencies (like the specific Java Runtime Environment version) with the application in the container. Additionally, if the application runs on Linux, the container also abstracts the flavor and version of the Linux OS. All dependencies required to run the application residing in the OS can also be bundled in the container. As a result, a containerized application runs the same on Oracle Linux as it would on Ubuntu.

Application Deployment

After the application and its dependencies are bundled in a container image, the next step is to distribute that application to all the groups that need it. Assume that many teams need to use this application and that the scale of the teams' usage might change over time. In a traditional data center, scaling this application would likely require an analysis to estimate the resources needed to run this application across the company for the next quarter or perhaps the entire next fiscal year. The IT organization that manages these physical resources would need to order new equipment to satisfy the needs of the business. In the cloud, new resources on which to run the application can be acquired on-demand and in a greater variety of sizes (that is, a virtual machine with fewer cores rather than a whole physical machine); however, the organization still needs to manage application deployments to those resources and manage those deployments to respond to the needs of the business over time. Container orchestration simplifies the solution to this problem: the containerization of the application makes the application easy to deploy in a variety of



environments, and the container orchestrator provides a way to manage and scale that application as needed.

Container Orchestration Features

Kubernetes provides several features to accomplish the task of orchestrating containerized applications.

Declarative Management

Kubernetes is designed to be managed in a *declarative* manner rather than an *imperative* manner. An example of imperative management is installing a program via the `apt-get install` command, which imperatively installs the application in a certain location. When an update to the program is needed, you imperatively tell `apt-get` to update the application.

Declarative management is managing based on *desired state*. *Deployments* are a Kubernetes component designed to hold the state of an application. Defined in JSON or YAML, the Deployment contains information about the container image, such as its version, the number of containers that should exist within the Kubernetes cluster, and a variety of other properties that Kubernetes needs in order to deploy the application properly. You simply state the number of application instances that you want to have, and Kubernetes creates and maintains the necessary number of containers across your resources. If a container fails for any reason (for example, the virtual machine on which the container is running goes offline), Kubernetes automatically stands up a new one on a healthy node to maintain the state declared in the Deployment. Kubernetes constantly checks the actual state against the desired state and ensures that the two states match.

The declarative model provides a benefit over the imperative model, in which the system must always be told what to do. In the declarative model, Kubernetes does the work for you, as long as the definitions for the desired system state exists.



Rolling Upgrades and Rollbacks

Another benefit of deploying applications via Kubernetes Deployments is its built-in rolling upgrade and rollback functionality. For example, when a YAML file that defines the state of a Deployment is updated with a new version of the container image, Kubernetes recognizes this change and begins to shut down existing instances of the older version while creating new instances with the updated version. While Kubernetes performs this rolling upgrade, it continues to direct requests to running container instances, which normally results in zero downtime. Likewise, if there is an issue with the upgrade, Kubernetes performs a rollback as needed.

Load Balancer

To manage connections to the applications deployed as containers within Kubernetes, the Kubernetes Service component provides a software-defined load balancer within the Kubernetes cluster. For example, a deployment of three instances of the Java application might be accessed by means of a single point in the Kubernetes Service. As a result, if an instance becomes unavailable or if a rolling upgrade is performed, the application can still be accessed without interruption.

These basic components of Kubernetes make it an excellent way to orchestrate large numbers of containerized applications across a pool of resources. As you consider how best to use container orchestration to meet your needs, learn more about Kubernetes by reading their documentation at <https://kubernetes.io/docs>.

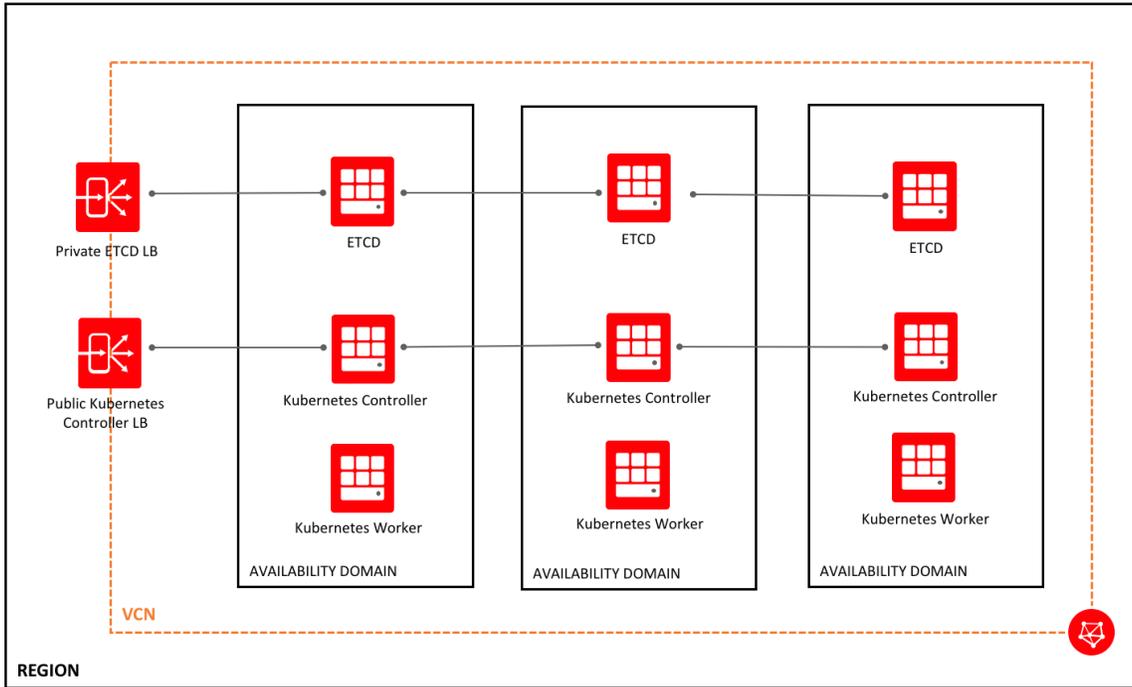
Cluster Architecture for Kubernetes Manual Deployment on Oracle Cloud Infrastructure

This section describes the components that make up a functioning Kubernetes cluster and explains how those components are deployed across the compute resources that make up the cluster. The architecture described in this section will be deployed in the “Kubernetes Manual Deployment Guide for Oracle Cloud Infrastructure” (the next section).

A Kubernetes cluster consists of three main components: etcd, Kubernetes masters (or controllers), and Kubernetes workers (or nodes). This guide explains how to create a highly available (HA) Kubernetes cluster with the following architecture:

- Three etcd nodes (across three availability domains in one region)
- Three Kubernetes masters (or controllers) (across three availability domains in one region)
- Three Kubernetes workers (or nodes) (across three availability domains in one region)

The steps in this guide produce an infrastructure similar to the one shown in the following diagram. The etcd cluster is configured to run on a separate set of compute resources from the Kubernetes cluster.



The following sections explain the components in greater detail.

etcd

etcd is a key-value store created by CoreOS. Kubernetes' state information is stored in the etcd cluster. This should not be confused with running an etcd cluster via Kubernetes; rather, this etcd cluster is helping to run Kubernetes itself.

In this guide, the etcd cluster is configured to run on a separate set of compute resources from the Kubernetes cluster. Running etcd on separate compute resources provides greater isolation between etcd and the components of the Kubernetes cluster.

Kubernetes Masters

The Kubernetes masters (or controllers) are machines (virtual or physical) that run the API server, controller manager, and scheduler components of the Kubernetes cluster.



Kubernetes Workers

The Kubernetes workers (or nodes) are machines (virtual or physical) that run the kubelet component of the Kubernetes cluster. The workers are the resources on which Kubernetes schedules containers (or pods).

Kubernetes Manual Deployment Guide for Oracle Cloud Infrastructure

This guide explains how to deploy and configure all the components and features required to run Kubernetes on Oracle Cloud Infrastructure. This guide assumes that you are starting with a “clean slate” environment; during the course of this guide, you will create all the resources that you need in Oracle Cloud Infrastructure.

This guide walks through the following tasks:

1. Create Oracle Cloud Infrastructure resources.
2. Generate certificates for Kubernetes components.
3. Bootstrap an HA etcd cluster.
4. Generate token and configuration files to authenticate components.
5. Bootstrap the HA Kubernetes masters (Kubernetes control plane).
6. Add a worker.
7. Configure remote access.
8. Deploy Kube-DNS.
9. Smoke test.

Step 1: Create Oracle Cloud Infrastructure Resources

This guide does *not* explain how to create the Oracle Cloud Infrastructure resources that you need to create the Kubernetes cluster. This section lists the required resources, but you must create them on your own by using the Oracle Cloud Infrastructure Console, CLI, API, SDKs, or the Terraform provider. For instructions, see the [Oracle Cloud Infrastructure documentation](#).

Networking

To create the Kubernetes cluster in this guide, you need the following Networking components. For more information about creating Networking components, see the [Networking section](#) of the Oracle Cloud Infrastructure documentation.

VCN

You need a single virtual cloud network (VCN) with the following values:

VCN Name	CIDR	Description
k8sOCI.oraclevcn.com	10.0.0.0/16	VCN used to host network resources for the Kubernetes cluster

Subnets

You need a VCN and at least one subnet per availability domain (three subnets total). In a production configuration, we recommend creating an etcd, master, and worker subnet per availability domain, which would result in nine subnets. For a test or learning deployment intended to be removed later, creating three subnets (one per availability domain) is sufficient. The following values describe the recommended configuration for a cluster in a region with three availability domains.

Use the following values:

Subnet Name	CIDR	Availability Domain	Description
publicETCDSubnetAD1.sub	10.0.20.0/24	AD1	Subnet used for etcd host in AD1
publicETCDSubnetAD2.sub	10.0.21.0/24	AD2	Subnet used for etcd host in AD2
publicETCDSubnetAD3.sub	10.0.22.0/24	AD3	Subnet used for etcd host in AD3
publicK8SMasterSubnetAD1.sub	10.0.30.0/24	AD1	Subnet used for Kubernetes masters in AD1
publicK8SMasterSubnetAD2.sub	10.0.31.0/24	AD2	Subnet used for Kubernetes masters in AD2
publicK8SMasterSubnetAD3.sub	10.0.32.0/24	AD3	Subnet used for Kubernetes masters in AD3
publicK8SWorkerSubnetAD1.sub	10.0.40.0/24	AD1	Subnet used to host Kubernetes workers in AD1
publicK8SWorkerSubnetAD2.sub	10.0.41.0/24	AD2	Subnet used to host Kubernetes workers in AD2
publicK8SWorkerSubnetAD3.sub	10.0.42.0/24	AD3	Subnet used to host Kubernetes workers in AD3

Security Lists

A production configuration should include the following security lists:

- `etcd_security_list`
- `k8sMaster_security_list`
- `k8sWorker_security_list`

For a list of the recommended security rules for each security list, see “Appendix A: Security Rules.”

Internet Gateway and Route Table

Your configuration should include one internet gateway and one route table rule that allows your Kubernetes cluster to access the internet through the internet gateway.

The route table rule should have a destination CIDR block of `0.0.0.0/0` and target type of `internet_gateway`. The target should be the internet gateway that you intend to use with your Kubernetes cluster.

Load Balancer

The recommended cluster configuration requires two load balancers. Create a private load balancer for your etcd nodes and a public load balancer for your Kubernetes masters. Populate the following table with your load balancer’s information to refer to throughout the guide.

In the guide, the public IP address of your Kubernetes master load balancer is referred to as `loadbalancer_public_ip`.

Use the following values:

Load Balancer Name	Load Balancer Type	Subnet1	Subnet2	Public IP	Private IP	Compute Resource Back Ends
lb-etcd	Private		Not applicable			Etcd1, Etcd2, Etcd3
lb-k8smaster	Public					KubeM1, KubeM2, KubeM3

Compute

Populate the following table to better track your Oracle Cloud Infrastructure resources for use with this guide. You can use any Compute instance shape for your compute resources. If you are exploring this technology via this guide, we recommend choosing small VM shapes such as VM.Standard1.1 and VM.Standard1.2.

For more information about creating Compute instances, see the [Compute section](#) of the Oracle Cloud Infrastructure documentation.

Compute Instance Name	Compute Instance Shape	Availability Domain	Subnet	Private IP Address	Public IP Address	Role in Kubernetes Cluster
Etcd1		AD1				etcd node
Etcd2		AD2				etcd node
Etcd3		AD3				etcd node
KubeM1		AD1				Kubernetes master
KubeM2		AD2				Kubernetes master
KubeM3		AD3				Kubernetes master
KubeW1		AD1				Kubernetes worker
KubeW2		AD2				Kubernetes worker
KubeW3		AD3				Kubernetes worker

In the guide, the values in the preceding table are referred to by the following names:

Compute Instance Name	Private IP Address	Public IP Address
Etcd1	etcd1_private_ip	etcd1_public_ip
Etcd2	etcd2_private_ip	etcd2_public_ip
Etcd3	etcd3_private_ip	etcd3_public_ip
KubeM1	kubem1_private_ip	kubem2_public_ip
KubeM2	kubem2_private_ip	kubem2_public_ip

Compute Instance Name	Private IP Address	Public IP Address
KubeM3	kubem3_private_ip	kubem3_public_ip
KubeW1	kubew1_private_ip	kubew1_public_ip
KubeW2	kubew2_private_ip	kubew2_public_ip
KubeW3	kubew3_private_ip	kubew3_public_ip

Step 2: Set Up a Certificate Authority and Create TLS Certificates

This step has three parts: install CFSSL, create certificates, and copy the certificate files to the host.

Install CFSSL

To generate certificates for use throughout the Kubernetes cluster, use the TLS certificate generating tool from CloudFlare, CFSSL. Run the commands in this section on your local computer (running macOS).

1. Download the cfssl package by using curl:

```
curl -O https://pkg.cfssl.org/R1.2/cfssl_darwin-amd64
```

2. Modify the permissions on the directory to make cfssl executable:

```
chmod +x cfssl_darwin-amd64
```

3. Move cfssl to your /usr/local/bin to add it to your path:

```
sudo mv cfssl_darwin-amd64 /usr/local/bin/cfssl
```

4. Download the cfssljson binary by using curl:

```
curl -O https://pkg.cfssl.org/R1.2/cfssljson_darwin-amd64
```

5. Modify the permissions on the directory to make cfssljson executable:

```
chmod +x cfssljson_darwin-amd64
```

6. Move cfssljson into your path:

```
sudo mv cfssljson_darwin-amd64 /usr/local/bin/cfssljson
```

Create Certificates

This section guides you through setting up the certificates that you need for your cluster. Run the commands in this section on your local computer (running macOS).

1. Create a configuration file that describes your certificate authority (CA). This file will be used as the authority for all keys related to the cluster.

```
cat > ca-config.json <<EOF
{
  "signing": {
    "default": {
      "expiry": "8760h"
    },
    "profiles": {
      "kubernetes": {
        "usages": ["signing", "key encipherment", "server auth", "client
auth"],
        "expiry": "8760h"
      }
    }
  }
}
EOF
```

2. Create a CA certificate signing request.

```
cat > ca-csr.json <<EOF
{
  "CN": "Kubernetes",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "Kubernetes",
      "OU": "CA",
      "ST": "Oregon"
    }
  ]
}
EOF
```

3. Generate a CA certificate and private key:

```
cfssl gencert -initca ca-csr.json | cfssljson -bare ca
```

The following files are created:

- ca-key.pem
- ca.pem

4. Create a client and server TLS certificate signing request for each Kubernetes worker node. Replace `$(instance)` with a name for the individual Kubernetes worker you are working with (for example, kubeW1, kubeW2, kubeW3).

```
for instance in worker-0 worker-1 worker-2; do
cat > ${instance}-csr.json <<EOF
{
  "CN": "system:node:${instance}",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "system:nodes",
      "OU": "Kubernetes The Hard Way",
      "ST": "Oregon"
    }
  ]
}
EOF
```

5. Generate a certificate and private key for each Kubernetes worker node. Run the following command once for each worker. Replace `$(instance)` with a name for the individual Kubernetes worker you are working with (for example, kubeW1, kubeW2, kubeW3). Replace `$(EXTERNAL_IP)` and `$(INTERNAL_IP)` with the public (external) and private (internal) IP addresses of the worker you are working with.

```
cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -hostname=${instance},${EXTERNAL_IP},${INTERNAL_IP} \
  -profile=kubernetes \
  ${instance}-csr.json | cfssljson -bare ${instance}
done
```

The following files are created:

- kubeW1-key.pem
- kubeW1.pem
- kubeW2-key.pem
- kubeW2.pem
- kubeW3-key.pem
- kubeW3.pem

6. Create the Admin Role Based Access Control (RBAC) certificate signing request. The Admin client certificate is used when you connect to the API server (master) via the admin role. This allows certain privileges under Kubernetes' native RBAC.

```
cat > admin-csr.json <<EOF
{
  "CN": "admin",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "system:masters",
      "OU": "Cluster",
      "ST": "Oregon"
    }
  ]
}
EOF
```

7. Generate the Admin client certificate and private key:

```
cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  admin-csr.json | cfssljson -bare admin
```

The following files are created:

- admin-key.pem
 - admin.pem
8. Create the kube-proxy client certificate signing request. This set of certificates is used by kube-proxy to connect to the Kubernetes master.

```
cat > kube-proxy-csr.json <<EOF
{
  "CN": "system:kube-proxy",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "system:node-proxier",
      "OU": "Cluster",
    }
  ]
}
```

```
    "ST": "Oregon"
  }
]
}
EOF
```

9. Generate the kube-proxy client certificate and private key:

```
cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  kube-proxy-csr.json | cfssljson -bare kube-proxy
```

The following files are created:

- kube-proxy-key.pem
- kube-proxy.pem

10. Create the Kubernetes server certificate. Replace `kubeMn_private_ip` with your master's private IP addresses and `loadbalancer_public_ip` with your load balancer IP address.

```
cat > kubernetes-csr.json <<EOF
{
  "CN": "kubernetes",
  "hosts": [
    "10.32.0.1",
    "kubeM1_private_ip",
    "kubeM2_private_ip",
    "kubeM3_private_ip",
    "loadbalancer_public_ip",
    "127.0.0.1",
    "kubernetes.default"
  ],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "Kubernetes",
      "OU": "Cluster",
      "ST": "Oregon"
    }
  ]
}
EOF
```

11. Generate the Kubernetes server certificate and private key:

```
cfssl gencert \  
-ca=ca.pem \  
-ca-key=ca-key.pem \  
-config=ca-config.json \  
-profile=kubernetes \  
kubernetes-csr.json | cfssljson -bare kubernetes
```

The following files are created::

- kubernetes-key.pem
- kubernetes.pem

12. Create etcd certificates as follows:

- A. Download the following project and follow the instructions to create certificates for each etcd node. Be sure to enter the private IP addresses for your etcd nodes in the **req-csr.pem** file under **config**.

<https://github.com/coreos/etcd/tree/v3.2.1/hack/tls-setup>

- B. Copy your etcd certificates from the **cert** directory into their own directory called **etcd-certs**. You will need this path later, so put the directory somewhere easy to remember (for example, **~/etcd-certs**)

- C. Use the `rename` command to rename your certificate authority files for etcd to reflect that they are for etcd:

```
brew install rename  
rename 's/ca/etcd-ca/' *
```

Copy CA Files to Hosts

1. Use a script similar to the following one to copy the necessary certificate files to the hosts. Replace `node_public_IP` with public IP addresses for workers, masters, and etcd nodes.

Filename: `copyCAs.sh`

```
for host in 'kubeW1_public_IP' 'kubeW2_public_IP' 'kubeW3_public_IP'; do  
  scp -i ~/.ssh/id_rsa ca.pem kube-proxy.pem kube-proxy-key.pem  
  ubuntu@${host}:~/  
done  
for host in 'kubeM1_public_IP' 'kubeM2_public_IP' 'kubeM3_public_IP' ; do  
  scp -i ~/.ssh/id_rsa ca.pem ca-key.pem kubernetes-key.pem kubernetes.pem  
  ubuntu@${host}:~/  
done  
for host in 'etcd1_public_IP' 'etcd2_public_IP' 'etcd3_public_IP'; do  
  scp -i ~/.ssh/id_rsa ca.pem ca-key.pem ubuntu@${host}:~/
```

```
done
```

2. Copy individual worker certificates to the corresponding workers. Each worker needs to have all the etcd certificates, the kubeconfig files, the ca.pem file, and its own certificates.

After the files have been copied to the workers, each worker should contain files similar to the following ones:

```
bootstrap.kubeconfig etcd1.csr etcd1.pem etcd2-key.pem etcd3.csr etcd3.pem etcd-ca-key.pem
kube-proxy-key.pem kube-proxy.pem kubew2-csr.json kubew2.kubeconfig proxy1.csr proxy1.pem

ca.pem etcd1-key.pem etcd2.csr etcd2.pem etcd3-key.pem etcd-ca.csr etcd-ca.pem
kube-proxy.kubeconfig kubew2.csr kubew2-key.pem kubew2.pem proxy1-key.pem
```

To organize these files, create a directory called **etcd-certs** in which to keep the etcd certificates. You can create this directory and move the appropriate files to it by using the following commands:

```
mkdir etcd-certs
mv *.* etcd-certs/
```

Step 3: Bootstrap an HA etcd Cluster

This step has three parts: install Docker, create an etcd directory, and provision the etcd cluster. Run the commands in this section on the etcd nodes unless otherwise instructed.

Install Docker on All Nodes

This guide uses the Docker installation script provided at get.docker.com. You can use another method to install Docker if you prefer.

1. Use curl to download the script:

```
curl -fsSL get.docker.com -o get-docker.sh
```

2. To run the script to install Docker, run the following command:

```
sh get-docker.sh
```

Create an etcd directory

Run the following command to create a directory for etcd to use to store its data:

```
sudo mkdir /var/lib/etcd
```

NOTE: If you have an issue while making your etcd cluster and need to redeploy your etcd servers, you must delete this directory or etcd will try to start with the old configuration. Delete it by using the following command: `sudo rm -rf /var/lib/etcd`

Provision the etcd Cluster

1. Set the following variables on each etcd node. The `NAME_n` and `HOST_n` variables provide the information required for the `CLUSTER` variable. For the `HOST_n` variables, replace `etcdn_private_ip` with your etcd node's private IP addresses.

```
NAME_1=etcd1
NAME_2=etcd2
NAME_3=etcd3
HOST_1= etcd1_private_ip
HOST_2= etcd2_private_ip
HOST_3= etcd3_private_ip

CLUSTER=${NAME_1}=https://${HOST_1}:2380,${NAME_2}=https://${HOST_2}:2380,
${NAME_3}=https://${HOST_3}:2380
DATA_DIR=/var/lib/etcd
ETCD_VERSION=latest
CLUSTER_STATE=new
THIS_IP=$(curl http://169.254.169.254/opc/v1/vnics/0/privateIp)
THIS_NAME=$(hostname)
```

2. Drop iptables.

Kubernetes and etcd make modifications to the iptables during setup. This step drops iptables completely to allow etcd to set them up as needed.

NOTE: This step does pose some security risk. Ensure that your security rules for your networking resources are sufficiently locked down before performing this step.

```
sudo iptables -F
```

3. Start the etcd containers by running the following code as-is on each node:

```
sudo docker run -d -p 2379:2379 -p 2380:2380 --volume=${DATA_DIR}:/etcd-data --volume=/home/ubuntu/etcd-certs:/etc/etcd --net=host --name etcd quay.io/coreos/etcd:${ETCD_VERSION} /usr/local/bin/etcd --name ${THIS_NAME} --cert-file=/etc/etcd/${THIS_NAME}.pem --key-file=/etc/etcd/${THIS_NAME}-key.pem --peer-cert-file=/etc/etcd/${THIS_NAME}.pem --peer-key-file=/etc/etcd/${THIS_NAME}-key.pem --trusted-ca-file=/etc/etcd/etcd-ca.pem --peer-trusted-ca-file=/etc/etcd/etcd-ca.pem --peer-client-cert-auth --client-cert-auth --initial-advertise-peer-urls https://${THIS_IP}:2380 --listen-peer-urls --listen-client-urls --advertise-client-urls --initial-cluster-token etcd-cluster-0 --initial-cluster ${CLUSTER} --initial-cluster-state new --data-dir=/etcd-data
```

NOTE: If you get the following error, some or all of the variables that you set in step 1 of this procedure are missing. Ensure that all of your variables have been set correctly.

```
"docker: invalid reference format"
```

4. Verify the cluster's successful deployment by running the following command on any etcd node:

```
sudo docker exec etcd etcdctl --ca-file=/etc/etcd/etcd-ca.pem --cert-file=/etc/etcd/${THIS_NAME}.pem --key-file=/etc/etcd/${THIS_NAME}-key.pem cluster-health
```

5. If the validation step does not work, clear your etcd files and try again:

```
sudo docker stop etcd
sudo docker rm etcd
sudo rm -rf ${DATA_DIR}
```

Step 4: Set Up RBAC

Run the commands in this section on your local computer running macOS.

1. Download and install kubectl.

```
curl -O https://storage.googleapis.com/kubernetes-release/release/v1.6.0/bin/darwin/amd64/kubectl
chmod +x kubectl
sudo mv kubectl /usr/local/bin
```

2. Create and distribute the TLS bootstrap token as follows:

- A. Generate a token:

```
BOOTSTRAP_TOKEN=$(head -c 16 /dev/urandom | od -An -t x | tr -d ' ')
```

- B. Generate a token file:

```
cat > token.csv <<EOF
${BOOTSTRAP_TOKEN}, kubelet-bootstrap,10001,"system:kubelet-bootstrap"
EOF
```

- C. Distribute the token to each master. Replace `controllern` with the public IP address of each Kubernetes master.

```
for host in controller0 controller1 controller2; do
  scp -i ~/.ssh/id_rsa token.csv ubuntu@${host}:~/
done
```

3. Create the bootstrap kubeconfig file:

```
kubectl config set-cluster kubernetes-the-hard-way \
  --certificate-authority=ca.pem \
  --embed-certs=true \
```

```
--server=https://${LB_IP}:6443 \  
--kubeconfig=bootstrap.kubeconfig
```

```
kubectl config set-credentials kubelet-bootstrap \  
--token=${BOOTSTRAP_TOKEN} \  
--kubeconfig=bootstrap.kubeconfig
```

```
kubectl config set-context default \  
--cluster=kubernetes-the-hard-way \  
--user=kubelet-bootstrap \  
--kubeconfig=bootstrap.kubeconfig
```

```
kubectl config use-context default --kubeconfig=bootstrap.kubeconfig
```

4. Create the kube-proxy kubeconfig file:

```
kubectl config set-cluster kubernetes-the-hard-way \  
--certificate-authority=ca.pem \  
--embed-certs=true \  
--server=https://${LB_IP}:6443 \  
--kubeconfig=kube-proxy.kubeconfig
```

```
kubectl config set-credentials kube-proxy \  
--client-certificate=kube-proxy.pem \  
--client-key=kube-proxy-key.pem \  
--embed-certs=true \  
--kubeconfig=kube-proxy.kubeconfig
```

```
kubectl config set-context default \  
--cluster=kubernetes-the-hard-way \  
--user=kube-proxy \  
--kubeconfig=kube-proxy.kubeconfig
```

```
kubectl config use-context default --kubeconfig=kube-proxy.kubeconfig
```

5. Distribute the client kubeconfig files. Replace `kubeWn_public_ip` with the public IP address of each worker node.

```
for host in 'kubeW1_public_ip' 'kubeW2_public_ip' 'kubeW3_public_ip'; do  
  scp -i ~/.ssh/id_rsa bootstrap.kubeconfig kube-proxy.kubeconfig  
  ubuntu@${host}:~/  
done
```

Step 5: Bootstrap an HA Kubernetes Control Plane

Provision the Kubernetes masters. Run the following commands on the masters.

1. Copy the bootstrap token into place:

```
sudo mkdir -p /var/lib/kubernetes/  
sudo mv token.csv /var/lib/kubernetes/
```

2. If you did not copy the necessary certificates to the Kubernetes masters in “Step 2: Set Up a Certificate Authority and Create TLS Certificates,” do that now. You need the `ca.pem`, `ca-key.pem`, `kubernetes-key.pem`, and `kubernetes.pem` certificates.

To secure communication between the Kubernetes API server (on the masters) and `kubectl` (used to control Kubernetes from another machine) and the `kubelet` (on the workers), the TLS certificates created in Step 2 are used. Communication between the Kubernetes API server and `etcd` is also secured via TLS certificates created in Step 2.

3. Copy the TLS certificates to the Kubernetes configuration directory:

```
sudo mv ca.pem ca-key.pem kubernetes-key.pem kubernetes.pem
/var/lib/kubernetes/
```

4. Download the official Kubernetes release binaries by using `wget`:

```
wget https://storage.googleapis.com/kubernetes-
release/release/v1.7.0/bin/linux/amd64/kube-apiserver
wget https://storage.googleapis.com/kubernetes-
release/release/v1.7.0/bin/linux/amd64/kube-controller-manager
wget https://storage.googleapis.com/kubernetes-
release/release/v1.7.0/bin/linux/amd64/kube-scheduler
wget https://storage.googleapis.com/kubernetes-
release/release/v1.7.0/bin/linux/amd64/kubectl
```

5. Install the Kubernetes binaries:

```
chmod +x kube-apiserver kube-controller-manager kube-scheduler kubectl
sudo mv kube-apiserver kube-controller-manager kube-scheduler kubectl
/usr/bin/
```

Kubernetes API Server

1. To better organize the certificates, place all `etcd` certificates in their own directory by using the following commands:

```
sudo mkdir /var/lib/etcd
sudo cp *.* /var/lib/etcd/.
```

2. Capture the internal IP address of the machine:

```
INTERNAL_IP=$(curl http://169.254.169.254/opc/v1/vnics/0/privateIp)
```

3. Drop `iptables`.

Kubernetes and `etcd` make modifications to the `iptables` during setup. This step drops `iptables` completely to allow `etcd` to set them up as needed.

NOTE: This step does pose some security risk. Ensure that your security rules for your networking resources are sufficiently locked down before performing this step.

```
sudo iptables -F
```

4. Create the systemd unit file for the Kubernetes API server. This file instructs systemd on Ubuntu to manage the Kubernetes API server as a systemd service.

```
cat > kube-apiserver.service <<EOF
[Unit]
Description=Kubernetes API Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStart=/usr/bin/kube-apiserver \\\
  --admission-
control=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,
ResourceQuota \\\
  --advertise-address=${INTERNAL_IP} \\\
  --allow-privileged=true \\\
  --apiserver-count=3 \\\
  --audit-log-maxage=30 \\\
  --audit-log-maxbackup=3 \\\
  --audit-log-maxsize=100 \\\
  --audit-log-path=/var/lib/audit.log \\\
  --authorization-mode=RBAC \\\
  --bind-address=${INTERNAL_IP} \\\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\\
  --enable-swagger-ui=true \\\
  --etcd-cafile=/var/lib/etcd/etcd-ca.pem \\\
  --etcd-certfile=/var/lib/etcd/etcd3.pem \\\
  --etcd-keyfile=/var/lib/etcd/etcd3-key.pem \\\
  --etcd-servers=https://<etcd1 private ip>:2379,https://<etcd2 private
ip>:2379,https://<etcd3 private ip>:2379 \\\
  --event-ttl=1h \\\
  --experimental-bootstrap-token-auth \\\
  --insecure-bind-address=0.0.0.0 \\\
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\\
  --kubelet-client-certificate=/var/lib/kubernetes/kubernetes.pem \\\
  --kubelet-client-key=/var/lib/kubernetes/kubernetes-key.pem \\\
  --kubelet-https=true \\\
  --runtime-config=rbac.authorization.k8s.io/v1alpha1 \\\
  --kubelet-preferred-address-
types=InternalIP,ExternalIP,LegacyHostIP,Hostname \\\
  --service-account-key-file=/var/lib/kubernetes/ca-key.pem \\\
  --service-cluster-ip-range=10.32.0.0/16 \\\
  --service-node-port-range=30000-32767 \\\
  --tls-cert-file=/var/lib/kubernetes/kubernetes.pem \\\
  --tls-private-key-file=/var/lib/kubernetes/kubernetes-key.pem \\\
  --token-auth-file=/var/lib/kubernetes/token.csv \\\
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

5. Start the kube-apiserver service:

```
sudo mv kube-apiserver.service /etc/systemd/system/kube-apiserver.service
sudo systemctl daemon-reload
sudo systemctl enable kube-apiserver
sudo systemctl start kube-apiserver
sudo systemctl status kube-apiserver --no-pager
```

6. If your service reports an error, debug by using the following commands to ensure that it is bound to the ports it needs:

```
journalctl -xe
netstat -na | more
netstat -na | grep 6443
```

7. Run the following command to verify that your Kubernetes API Server is running:

```
kubectl get componentstatuses
```

The output from the command should look as follows:

```
ubuntu@Kubem3:~$ kubectl get componentstatuses
NAME                STATUS    MESSAGE                                           ERROR
scheduler           Unhealthy Get http://127.0.0.1:10251/healthz: dial tcp 127.0.0.1:10251: getsockopt: connection refused
controller-manager  Unhealthy Get http://127.0.0.1:10252/healthz: dial tcp 127.0.0.1:10252: getsockopt: connection refused
etcd-2              Healthy   {"health": "true"}
etcd-0              Healthy   {"health": "true"}
etcd-1              Healthy   {"health": "true"}
```

Kubernetes Scheduler

1. Create the systemd unit file for the Kubernetes scheduler. This file instructs systemd on Ubuntu to manage the Kubernetes scheduler as a systemd service.

```
kube-scheduler.service
cat > kube-scheduler.service <<EOF
[Unit]
Description=Kubernetes Scheduler
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
[Service]
ExecStart=/usr/bin/kube-scheduler \\\
  --leader-elect=true \\\
  --master=http://${INTERNAL_IP}:8080 \\\
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

2. Start the kube-scheduler service:

```
sudo mv kube-scheduler.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable kube-scheduler
sudo systemctl start kube-scheduler
sudo systemctl status kube-scheduler --no-pager
```

3. Run the following command to verify that the Kubernetes scheduler is running:

```
kubectl get componentstatuses
```

The output from the command should look as follows:

```
ubuntu@kubem3:~$ kubectl get componentstatuses
NAME                STATUS    MESSAGE                                     ERROR
scheduler           Healthy   ok
controller-manager  Unhealthy Get http://127.0.0.1:10252/healthz: dial tcp 127.0.0.1:10252: getsockopt: connection refused
etcd-1              Healthy   {"health": "true"}
etcd-2              Healthy   {"health": "true"}
etcd-0              Healthy   {"health": "true"}
```

Kubernetes Controller Manager

1. Create the systemd unit file for the Kubernetes controller manager. This file instructs systemd on Ubuntu to manage the Kubernetes controller manager as a systemd service.

```
cat > kube-controller-manager.service <<EOF
[Unit]
Description=Kubernetes Controller Manager
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStart=/usr/bin/kube-controller-manager \\\
  --address=0.0.0.0 \\\
  --allocate-node-cidrs=true \\\
  --cluster-cidr=10.200.0.0/16 \\\
  --cluster-name=kubernetes \\\
  --cluster-signing-cert-file=/var/lib/kubernetes/ca.pem \\\
  --cluster-signing-key-file=/var/lib/kubernetes/ca-key.pem \\\
  --leader-elect=true \\\
  --master=http://${INTERNAL_IP}:8080 \\\
  --root-ca-file=/var/lib/kubernetes/ca.pem \\\
  --service-account-private-key-file=/var/lib/kubernetes/ca-key.pem \\\
  --service-cluster-ip-range=10.32.0.0/16 \\\
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

2. Start the kube-controller-manager service:

```
sudo mv kube-controller-manager.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable kube-controller-manager
sudo systemctl start kube-controller-manager
sudo systemctl status kube-controller-manager --no-pager
```

3. Run the following command to verify that the Kubernetes controller manager is running:

```
kubectl get componentstatuses
```

The output from the command should look as follows:

```
[ubuntu@kubem1:~]$ kubectl get componentstatuses
NAME                STATUS    MESSAGE                                 ERROR
controller-manager  Healthy  ok
scheduler           Healthy  ok
etcd-1              Healthy  {"health": "true"}
etcd-0              Healthy  {"health": "true"}
etcd-2              Healthy  {"health": "true"}
```

Step 6: Add a Worker

This step has the following parts: generate a kubeconfig file for each worker, generate a kubeconfig file for kube-proxy, configure the files on the workers, and install several components.

Generate a kubeconfig File for Each Worker

Be sure to use the client certificate (created in “Step 2: Set Up a Certificate Authority and Create TLS Certificates”) that matches each worker’s node name.

Use the following script to generate a kubeconfig file for each worker node:

```
for instance in worker-0 worker-1 worker-2; do
  kubectl config set-cluster kubernetes-the-hard-way \
    --certificate-authority=ca.pem \
    --embed-certs=true \
    --server=https://${LB_IP}:6443 \
    --kubeconfig=${instance}.kubeconfig

  kubectl config set-credentials system:node:${instance} \
    --client-certificate=${instance}.pem \
    --client-key=${instance}-key.pem \
    --embed-certs=true \
    --kubeconfig=${instance}.kubeconfig

  kubectl config set-context default \
    --cluster=kubernetes-the-hard-way \
    --user=system:node:${instance} \
    --kubeconfig=${instance}.kubeconfig

  kubectl config use-context default --kubeconfig=${instance}.kubeconfig
done
```

The following files are created:

- worker-0.kubeconfig
- worker-1.kubeconfig
- worker-2.kubeconfig

Generate a kubeconfig File for Kube-Proxy

Use the following commands to generate a kubeconfig file for kube-proxy to use to connect to the master:

```
kubectl config set-cluster kubernetes-the-hard-way \  
  --certificate-authority=ca.pem \  
  --embed-certs=true \  
  --server=https://{LB_IP}:6443 \  
  --kubeconfig=kube-proxy.kubeconfig
```

```
kubectl config set-credentials kube-proxy \  
  --client-certificate=kube-proxy.pem \  
  --client-key=kube-proxy-key.pem \  
  --embed-certs=true \  
  --kubeconfig=kube-proxy.kubeconfig
```

```
kubectl config set-context default \  
  --cluster=kubernetes-the-hard-way \  
  --user=kube-proxy \  
  --kubeconfig=kube-proxy.kubeconfig
```

```
kubectl config use-context default --kubeconfig=kube-proxy.kubeconfig
```

Configure the Files on the Workers

Log in to each worker and run the following file move commands:

```
sudo mkdir /var/lib/kubernetes/  
sudo mkdir /var/lib/kubelet/  
sudo mkdir /var/lib/kube-proxy/  
sudo mv bootstrap.kubeconfig /var/lib/kubelet  
sudo mv kube-proxy.kubeconfig /var/lib/kube-proxy  
sudo mv ca.pem /var/lib/kubernetes/  
sudo mv $(hostname)-key.pem $(hostname).pem /var/lib/kubelet/  
sudo mv $(hostname).kubeconfig /var/lib/kubelet/kubeconfig  
sudo mv ca.pem /var/lib/kubernetes/
```

Install Flannel

1. Install Flannel by using the following commands:

```
wget https://github.com/coreos/flannel/releases/download/v0.6.2/flanneld-  
amd64 -O flanneld && chmod 755 flanneld  
sudo mv flanneld /usr/bin/flanneld
```

2. Configure the flannel service by creating the following

/etc/systemd/system/flanneld.service file:

```
[Unit]
Description=Flanneld overlay address etcd agent
[Service]
Type=notify
EnvironmentFile=-/usr/local/bin/flanneld
ExecStart=/usr/bin/flanneld -ip-masq=true -iface 10.0.0.11 --etcd-
cafile=/home/ubuntu/etcd-ca.pem --etcd-certfile=/home/ubuntu/etcd3.pem --
etcd-keyfile=/home/ubuntu/etcd3-key.pem --etcd-
endpoints=https://10.0.0.7:2379,https://10.0.1.7:2379,https://10.0.2.6:237
9 --etcd-prefix=/kube/network
Restart=on-failure
```

3. Start the flannel service:

```
sudo systemctl daemon-reload
sudo systemctl restart flanneld
sudo systemctl enable flanneld
sudo systemctl status flanneld --no-pager
```

The last command checks the status of the service and should give output similar to the following example:

```
ubuntu@kubew1:~$ sudo systemctl status flanneld --no-pager
flanneld.service - Flanneld overlay address etcd agent
   Loaded: loaded (/etc/systemd/system/flanneld.service; static; vendor
   preset: enabled)
   Active: active (running) since Fri 2017-09-15 17:40:06 UTC; 1s ago
   Main PID: 1904 (flanneld)
     Tasks: 10
    Memory: 10.6M
       CPU: 292ms
    CGroup: /system.slice/flanneld.service
           └─1904 /usr/bin/flanneld -ip-masq=true -iface 10.0.0.11 --etcd-
cafile=/home/ubuntu/etcd-ca.pem --etcd-certfile=/home/ubuntu/etcd3.pem --
etcd-keyfile=/home/ubuntu/etcd3-key.pem --etcd-
endpoints=https://10.0.0.7:2379,https://10.0.1...

Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.283241 01904 ipmasq.go:47] Adding iptables rule: ! -
s 10.200.0.0/16 -d 10.200.0.0/16 -j MASQUERADE
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.285301 01904 manager.go:246] Lease acquired: 10.200.63.0/24
Sep 15 17:40:06 kubew1 systemd[1]: Started Flanneld overlay address etcd
agent.
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.285666 01904 network.go:58] Watching for L3 misses
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.285698 01904 network.go:66] Watching for new subnet leases
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.297509 01904 network.go:153] Handling initial subnet events
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.297538 01904 device.go:163] calling GetL2List()
dev.link.Index: 3
```

```
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.297617 01904 device.go:168] calling
NeighAdd: 10.0.0.9, 86:64:67:a6:29:e5
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.297682 01904 device.go:168] calling NeighAdd: 10.0.2.9,
e6:76:2b:98:c6:ab
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.297728 01904 device.go:168] calling NeighAdd: 10.0.0.8,
a6:67:22:ab:2f:8b
```

Install the Container Networking Interface (CNI)

1. Create the following files for the CNI service:

```
/etc/systemd/system/cni-bridge.service
```

```
[Unit]
Requires=network.target
Before=docker.service
[Service]
Type=oneshot
ExecStart=/usr/local/bin/cni-bridge.sh
RemainAfterExit=true
```

```
/usr/local/bin/cni-bridge.sh
```

```
#!/bin/bash

set -x

/sbin/ip link add name cni0 type bridge
/sbin/ip addr add $(grep '^FLANNEL_SUBNET' /run/flannel/subnet.env | cut -
d= -f2) dev cni0
/sbin/ip link set dev cni0 up
```

2. Run the following block of commands to install and start the CNI service.

```
sudo su
mkdir -p /opt/cni/bin /etc/cni/net.d
chmod +x /usr/local/bin/cni-bridge.sh
curl -L --
retry 3 https://github.com/container networking/cni/releases/download/v0.5.
2/cni-amd64-v0.5.2.tgz -o /tmp/cni-plugin.tar.gz
tar xzf /tmp/cni-plugin.tar.gz -C /opt/cni/bin/
printf '{\n  "name": "podnet",\n  "type": "flannel",\n  "delegate":
{\n    "isDefaultGateway": true\n  }\n}\n' >/etc/cni/net.d/10-
flannel.conf
chmod +x /usr/local/bin/cni-bridge.sh
systemctl enable cni-bridge && systemctl start cni-bridge
exit
```

3. Run the following command to check the status of the service:

```
sudo systemctl status cni-bridge
```

The output should be similar to the following example:

```
ubuntu@kubew3:~$ sudo systemctl status cni-bridge.service
cni-bridge.service
  Loaded: loaded (/etc/systemd/system/cni-bridge.service; static; vendor
  preset: enabled)
  Active: active (exited) since Fri 2017-09-15 18:20:25 UTC; 27s ago
  Process: 1940 ExecStart=/usr/local/bin/cni-bridge.sh (code=exited,
  status=0/SUCCESS)
  Main PID: 1940 (code=exited, status=0/SUCCESS)

Sep 15 18:20:25 kubew3 systemd[1]: Starting cni-bridge.service...
Sep 15 18:20:25 kubew3 cni-bridge.sh[1940]: + /sbin/ip link add name cni0
  type bridge
Sep 15 18:20:25 kubew3 cni-bridge.sh[1940]: ++ grep '^FLANNEL_SUBNET'
  /run/flannel/subnet.env
Sep 15 18:20:25 kubew3 cni-bridge.sh[1940]: ++ cut -d= -f2
Sep 15 18:20:25 kubew3 cni-bridge.sh[1940]: + /sbin/ip addr add
  10.200.63.1/24 dev cni0
Sep 15 18:20:25 kubew3 cni-bridge.sh[1940]: + /sbin/ip link set dev cni0
  up
Sep 15 18:20:25 kubew3 systemd[1]: Started cni-bridge.service.
```

```
ubuntu@kubew1:~$ ifconfig
cni0      Link encap:Ethernet  HWaddr 32:2a:0a:be:35:a2
          inet addr:10.200.63.1  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::302a:aff:febe:35a2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:648 (648.0 B)
```

Install Docker

1. Install Docker as follows:

```
wget https://get.docker.com/builds/Linux/x86_64/docker-1.12.6.tgz
tar -xvf docker-1.12.6.tgz
sudo cp docker/docker* /usr/bin/
```

2. Create the Docker service by creating the following

/etc/systemd/system/docker.service file:

```
[Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.io
After=network.target firewall.service cni-bridge.service
Requires=ckey cni-bridge.service

[Service]
ExecStart=/usr/bin/dockerd \
  --bridge=cni0 \
  --iptables=false \
  --ip-masq=false \
```

```
--host=unix:///var/run/docker.sock \  
--insecure-registry=registry.oracledx.com \  
--log-level=error \  
--storage-driver=overlay  
Restart=on-failure  
RestartSec=5  
  
[Install]  
WantedBy=multi-user.target
```

3. Start the Docker service:

```
sudo systemctl daemon-reload  
sudo systemctl enable docker  
sudo systemctl start docker  
sudo docker version  
sudo docker network inspect bridge
```

4. Ensure that Docker is properly configured by running the following command:

```
sudo docker network inspect bridge
```

The output should show that Docker is configured to use the cni0 bridge.

Install Kubelet

1. Use the following commands to download kubelet version 1.7.4:

```
wget -q --show-progress --https-only --  
timestamping https://storage.googleapis.com/kubernetes-  
release/release/v1.7.4/bin/linux/amd64/kubectl  
wget -q --show-progress --https-only --  
timestamping https://storage.googleapis.com/kubernetes-  
release/release/v1.7.4/bin/linux/amd64/kube-proxy  
wget -q --show-progress --https-only --  
timestamping https://storage.googleapis.com/kubernetes-  
release/release/v1.7.4/bin/linux/amd64/kubelet
```

2. Install kubelet:

```
chmod +x kubectl kube-proxy kubelet  
sudo mv kubectl kube-proxy kubelet /usr/bin/
```

3. Create the following `/etc/systemd/system/kubelet.service` file:

```
/etc/systemd/system/kubelet.service  
[Unit]  
Description=Kubernetes Kubelet  
Documentation=https://github.com/GoogleCloudPlatform/kubernetes  
After=docker.service  
Requires=docker.service  
  
[Service]  
ExecStart=/usr/bin/kubelet \  
--allow-privileged=true \  

```

```

--cluster-dns=10.32.0.10 \
--cluster-domain=cluster.local \
--container-runtime=docker \
--image-pull-progress-deadline=2m \
--kubeconfig=/var/lib/kubelet/kubeconfig \
--require-kubeconfig=true \
--network-plugin=cni \
--pod-cidr=10.200.88.0/24 \
--serialize-image-pulls=false \
--register-node=true \
--runtime-request-timeout=10m \
--tls-cert-file=/var/lib/kubelet/kubew2.pem \
--tls-private-key-file=/var/lib/kubelet/kubew2-key.pem \
--hostname-override= kubew2.sub08071631352.kubevcn.oraclevcn.com \
--v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target

```

4. Drop iptables before starting the kubelet service

```
iptables -F
```

5. Start the kubelet service:

```

sudo systemctl daemon-reload
sudo systemctl enable kubelet
sudo systemctl start kubelet
sudo systemctl status kubelet --no-pager

```

Install kube-proxy

1. Create the following `/etc/systemd/system/kube-proxy.service` file for the kube-proxy service:

```

[Unit]
Description=Kubernetes Kube Proxy
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStart=/usr/bin/kube-proxy \
  --cluster-cidr=10.200.0.0/16 \
  --kubeconfig=/var/lib/kube-proxy/kube-proxy.kubeconfig \
  --proxy-mode=iptables \
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target

```

2. Start the kube-proxy service:

```
sudo systemctl daemon-reload
sudo systemctl enable kube-proxy
sudo systemctl start kube-proxy
sudo systemctl status kube-proxy --no-pager
```

Remove a worker

If you need to remove a worker, you can do so by using the following commands. Replace `nodename` with the name of your Kubernetes worker node from `kubectl get nodes`.

1. Remove any pods from the node:

```
kubectl drain nodename
```

2. Remove the worker node:

```
kubectl delete node nodename
```

NOTE: These actions do not delete the instance. If you want to delete the Oracle Cloud Infrastructure Compute instance, you must do that via the CLI or Console.

Step 7: Configure kubectl for Remote Access

`kubectl` is the command line tool used to control and manage Kubernetes clusters. By installing and configuring `kubectl` on your local computer, you can manage your Kubernetes clusters easily through your computer, rather than logging in to the cluster or some other remote location to manage the clusters. If you want to manage your Kubernetes cluster from a computer other than your local one, run these steps on that computer.

This step enables you to connect to your cluster in Oracle Cloud Infrastructure. Run the following commands on your local computer, replacing `${LB_IP}` with your load balancer's IP address.

```
kubectl config set-cluster kubernetes-the-hard-way \
  --certificate-authority=ca.pem \
  --embed-certs=true \
  --server=https://${LB_IP}:6443

kubectl config set-credentials admin \
  --client-certificate=admin.pem \
  --client-key=admin-key.pem

kubectl config set-context kubernetes-the-hard-way \
  --cluster=kubernetes-the-hard-way \
  --user=admin

kubectl config use-context kubernetes-the-hard-way
```

Step 8: Deploy Kube-DNS

1. Deploy the kube-dns cluster add-on:

```
kubectl create -f https://storage.googleapis.com/kubernetes-the-hard-way/kube-dns.yaml
```

The output should be similar to the following example:

```
Service account "kube-dns" created configmap "kube-dns" created service "kube-dns" created deployment "kube-dns" created
```

2. List the pods created by the kube-dns deployment:

```
kubectl get pods -l k8s-app=kube-dns -n kube-system
```

The output should be similar to the following example:

NAME	READY	STATUS	RESTARTS	AGE
kube-dns-3097350089-gq015	3/3	Running	0	20s
kube-dns-3097350089-q64qc	3/3	Running	0	20s

Step 9: Smoke Test

This section walks you through a quick smoke test to ensure the cluster is working as expected.

1. Create an nginx deployment with three replicas by using the following command:

```
kubectl run nginx --image=nginx --port=80 --replicas=3
```

The output should look as follows:

```
deployment "nginx" created
```

2. Run the following command to see the pods that your deployment created and ensure that they are in a Running state:

```
kubectl get pods -o wide
```

The output should be similar to the following example:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nginx-158599303-bt144-worker-ad1-0	1/1	Running	0	18s	10.99.49.5	k8s-
nginx-158599303-ndxtc-worker-ad2-0	1/1	Running	0	18s	10.99.49.3	k8s-
nginx-158599303-r2801-worker-ad3-0	1/1	Running	0	18s	10.99.49.4	k8s-

3. Create a service to connect to your nginx deployment.

```
kubectl expose deployment nginx --type NodePort
```

4. View the service that you created. Note that `--type=LoadBalancer` is not currently supported in Oracle Cloud Infrastructure.

```
kubectl get service
NAME          CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
kubernetes   10.21.0.1     <none>         443/TCP          1h
nginx        10.21.62.159 <nodes>        80:30050/TCP     1h
```

At this point, you must either *manually* create a load balancer in BMC that routes to the cluster (in this example, that would be 10.21.62.159:80) or expose the node port (in this case, 30050) publicly via Oracle Cloud Infrastructure's security lists. In this guide, you do the latter.

5. Modify the worker's security list, allowing ingress traffic to 30050.
6. Get the `NodePort` that was set up for the `nginx` service:

```
NodePort=$(kubectl get svc nginx --output=jsonpath='{range .spec.ports[0]}{.nodePort}')
```

7. Get the `worker_public_ip` value for one of the workers from the UI.
8. Test the `nginx` service with these values by using `curl`:

```
curl http://${worker_public_ip}:${NodePort}
```

The output should look like the following example:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
  body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
  }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed
and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Appendix A: Security Rules

This appendix outlines the security rules for the following security lists:

- **etcd:** etcd_security_list.sl
- **Kubernetes masters:** k8sMaster_security_list.sl
- **Kubernetes workers:** k8sWorker_security_list.sl

The specific ingress and egress rules for each of these security lists are provided in the following tables. All rules are stateful.

ETCD SECURITY LIST INGRESS RULES

Source	IP Protocol	Source Port Range	Destination Port Range
10.0.0.0/16	TCP	All	ALL
10.0.0.0/16	TCP	All	22
10.0.0.0/16	TCP	All	2379-2380

ETCDSECURITY LIST EGRESS RULES

Destination	IP Protocol	Source Port Range	Destination Port Range
0.0.0.0/0	All	All	All

KUBERNETES MASTER SECURITY LIST INGRESS RULES

Destination	IP Protocol	Source Port Range	Destination Port Range
10.0.0.0/16	All	All	All
10.0.0.0/16	TCP	All	3389
10.0.0.0/16	TCP	All	6443
10.0.0.0/16	TCP	All	22

KUBERNETES MASTER SECURITY LIST EGRESS RULES

Destination	IP Protocol	Source Port Range	Destination Port Range
0.0.0.0/0	All	All	All


KUBERNETES WORKER SECURITY LIST INGRESS RULES

Destination	IP Protocol	Source Port Range	Destination Port Range
10.0.0.0/16	TCP	All	All
10.0.0.0/16	TCP	All	22
10.0.0.0/16	UDP	All	30000-32767

KUBERNETES WORKER SECURITY LIST EGRESS RULES

Destination	IP Protocol	Source Port Range	Destination Port Range
0.0.0.0/0	All	All	All



Oracle Corporation, World Headquarters

500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries

Phone: +1.650.506.7000
Fax: +1.650.506.7200

CONNECT WITH US

-  blogs.oracle.com/oracle
-  facebook.com/oracle
-  twitter.com/oracle
-  oracle.com

Integrated Cloud Applications & Platform Services

Copyright © 2018, Oracle and/or its affiliates. All rights reserved. This document is provided **for** information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0218

Kubernetes on Oracle Cloud Infrastructure
February 2018
Author: Kaslin Fields



Oracle is committed to developing practices and products that help protect the environment