# Kubernetes / Docker / hybris POC on AWS

This document includes detailed information about running a hybris Accelerator Cluster in Docker containers.

This document describes how projects can use Docker for deploying Hybris Commerce Suite instances. The document is a result of a POC we performed on this topic, and we share the results here.

Please note that for this version of Hybris there is no official support for Docker, Kubernetes, or the way we used AWS, provided with Hybris Commerce Suite.

In the future releases we may add support for deployment and cluster management, but we cannot name any dates or give a guarantee on the technologies finally used.

**About this Document**

This document shows how to run a hybris Accelerator Cluster in Docker containers.

**Audience**: System/hybris Administrators, Consultants, Developers

**Validity**: 5.6.0 and higher

**Based on hybris version**: 5.6.0

**Installation Guide Resources**

Expand all Collapse all

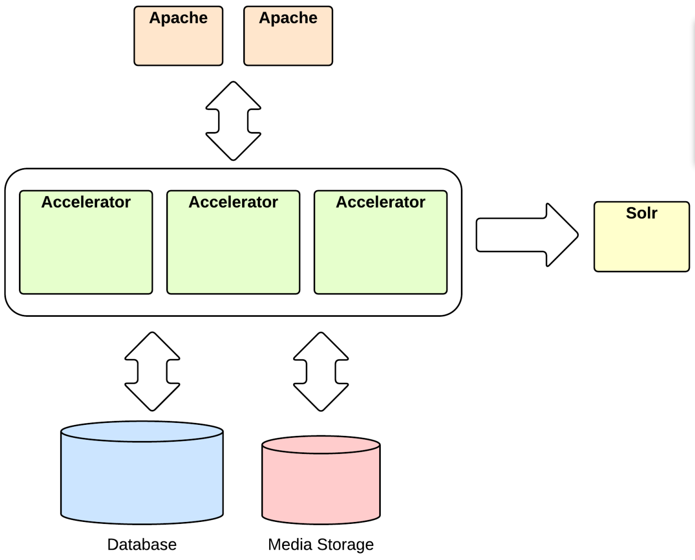
**See also**

- Initializing and Updating the hybris Commerce Suite
- Release 5 Documentation Home
- hybrs Download page
- Development Landscape - Release Strategy, History and EOL
- hybris Migration Guide
- hybris Administration Console - End User Guide
- Cluster - Technical Guide
- Build Framework

# Introduction

This document shows how to run a highly available and scalable hybris Accelerator Cluster in Docker containers. For this purpose, we use Amazon AWS instances with a CoreOS AMI. A script called *ycluster* creates and configures our cluster on AWS. CoreOS will be configured via cloudconfig scripts. After booting up, the instances are provisioned with Kubernetes services. We use Kubernetes to manage containers and networking within the cluster.

On this wiki page we're describing the processes in detail - from setting up the AWS instances to scaling up the hybris Accelerator.



## Technologies

### Amazon AWS

For our demo we're using Amazon Web Services as infrastructure provider. We're using **EC2 instances** within an Amazon **VPC** (Virtual Private Cloud) and **RDS MySQL** database instances. For BLOB Storage (hybris media and Docker Registry) we're using **S3**. Most of the infrastructure **se tup is scripted** in the ***yCluster*** script (see next topic).

### Docker

The Docker technology is based on Linux Kernel Containers. In the past Docker used LXC for managing containers. Since mid 2014 Docker has used its own Kernel interface called lib container. Docker is a suite for **packaging**, **shipping** and **running** applications, and managing its **network** configuration.

In this POC we're using Docker to package and deploy most cluster components (e.g. Accelerator, Solr, Apache). For privacy we have our own **D ocker Registry**, which stores all container images.

### CoreOS

CoreOS is a Linux distribution with the focus on **running containers**. Basically it comes just with the Kernel, systemd and some services such as etcd, fleet and Docker.

We use it as a very lightweight Linux base installation for our EC2 instances.

### Kubernetes

Kubernetes is a **cluster management** technology backed by Docker.

As a central piece of our POC, we're using it for the actual cluster configuration, roll-out and management (scaling).

## Entities

To understand Kubernetes, it's essential to know the entities available to define and run a cluster. Here's a brief overview:

### Pods

Kubernetes makes use of the so called ***Pods***.

Pods consist of one **or more** Docker containers residing on the same Docker host. Containers of a Pod share the same network stack and may also share directories.

You can use Pods to create application packages that bundle the application itself with peripheral services such as logging or monitoring agents. For example, you have an Apache container and want to transfer its logs to Logstash. The way to do it is to create a pod with an Apache container and a logstash-forwarder container. You would share an empty directory between those containers. The Apache logs into this directory and the logstash-forwarder reads those logs and sends them to the Logstash server.

**Kubernetes Pods**
Pods are bundles of containers.

Pods are assigned to hosts. All pod's containers run on the same host.

A pod's containers share the same networking stack, that is: the same IP address.

Containers of the same pod can share directories.

```
id: apache
kind: Pod
apiVersion: v1beta1
desiredState:
  manifest:
    version: v1beta1
    id: apache
    containers:
      - name: apache
        image: myregistry/apache
        cpu: 1000
        ports:
          - name: http
            containerPort: 80
        volumeMounts:
          - name: logs
            mountPath: /var/log/apache2
      - name: logstash-forwarder
        image: myregistry/logstash-forwarder
        volumeMounts:
          - name: logs
            mountPath: /var/log/apache2
        env:
          - key: logpath
            value: "/var/log/apache"
    volumes:
      - name: logs
        source:
          emptyDir: {}
labels:
  k8s-app: apache-pod
```

Here we define a Pod holding two containers: Apache and logstash-forwarder. Both share an empty directory *logs*. Further we label the Pod with `k8s-app: apache-pod`. We will use the label later for defining replication controllers and services.

## Replication Controllers

Pods themselves don't include any sort of high availability. If one process exits, the Pod will be stopped.

To ensure a specified number of defined Pod Templates are running, use *Replication Controllers (RC)*. Replication Controllers check if the specified number of Pods with a defined label are running. If there are not enough of them, Replication Controllers create a Pod based on its Pod Template. If there are too many of them, Replication Controllers terminate Pods.

You can use Replication Controllers to scale your application and provide high availability.

**Kubernetes Replication Controllers**
Replication Controllers (RC) ensure a specified number of Pods are running.

RCs are used to scale your application horizontally.

RCs restart Pods in the cluster when they have failed or the host has crashed.

You can realize high availability using Replication Controllers.

A Replication Controller that spins up 3 instances of our Apache Pod would look like this:

```
kind: ReplicationController
apiVersion: v1beta1
id: apache
namespace: default
labels:
  k8s-app: apache-rc
desiredState:
  replicas: 3
  replicaSelector:
    k8s-app: apache-pod
  podTemplate:
    desiredState:
      manifest:
        version: v1beta1
        id: apache
        containers:
          - name: apache
            image: myregistry/apache
            cpu: 1000
            ports:
              - name: http
                containerPort: 80
            volumeMounts:
              - name: logs
                mountPath: /var/log/apache2
          - name: logstash-forwarder
            image: myregistry/logstash-forwarder
            volumeMounts:
              - name: logs
                mountPath: /var/log/apache2
            env:
              - key: logpath
                value: "/var/log/apache"
        volumes:
          - name: logs
            source:
              emptyDir: {}
    labels:
      k8s-app: apache-pod
```

Notice that we use the Pod's label to define the members of a Replication Controller!

## Services

Kubernetes Services are defined as multiple Pods serving the same application. They have their own IP and redirect traffic in a round-robin fashion to their endpoints. For example, you would add all your Apache Pods to an Apache Service. The Pods would be the service's endpoints.

```
kind: Service
apiVersion: v1beta1
id: skydns
namespace: default
protocol: TCP
port: 80
portalIP: 10.255.0.23
containerPort: 80
labels:
  k8s-app: apache-svc
selector:
  k8s-app: apache-pod
```

Here we define an Apache service holding the Apache Pods. The service provides TCP port 80 and redirects it to the containers' port 80. We explicitly assign a Service IP in this example. If you don't do it, the service will get a random Service IP you can query via the Kubernetes API. To select the Apache Pods, we set up a selector querying for the label `k8s-app: apache-pod`.

With the service running you can connect to the Apaches via `10.255.0.23:80` from within the whole cluster. This is iptables magic done by *kube-proxy*. More on this at Kubernetes Service IP Routing.

### Packages

Kubernetes consists of multiple packages:

- kube-apiserver *(master only)*
- kube-controller-manager *(master only)*
- kube-scheduler *(master only)*
- kube-proxy
- kube-kubelet *(nodes only)*

#### kube-apiserver

The kube-apiserver provides the Kubernetes API that all other services connect to.

#### kube-scheduler

The kube-scheduler distributes the containers over the Kubernetes minions.

#### kube-controller-manager

The kube-controller-manager runs the replication controllers. It respawns failed containers and manages scaling.

#### kube-kubelet

The kube-kubelet service runs on the Kubernetes nodes (known as "minions"). It listens to the kube-apiserver and receives messages from the kube-scheduler and manages the local Docker service.

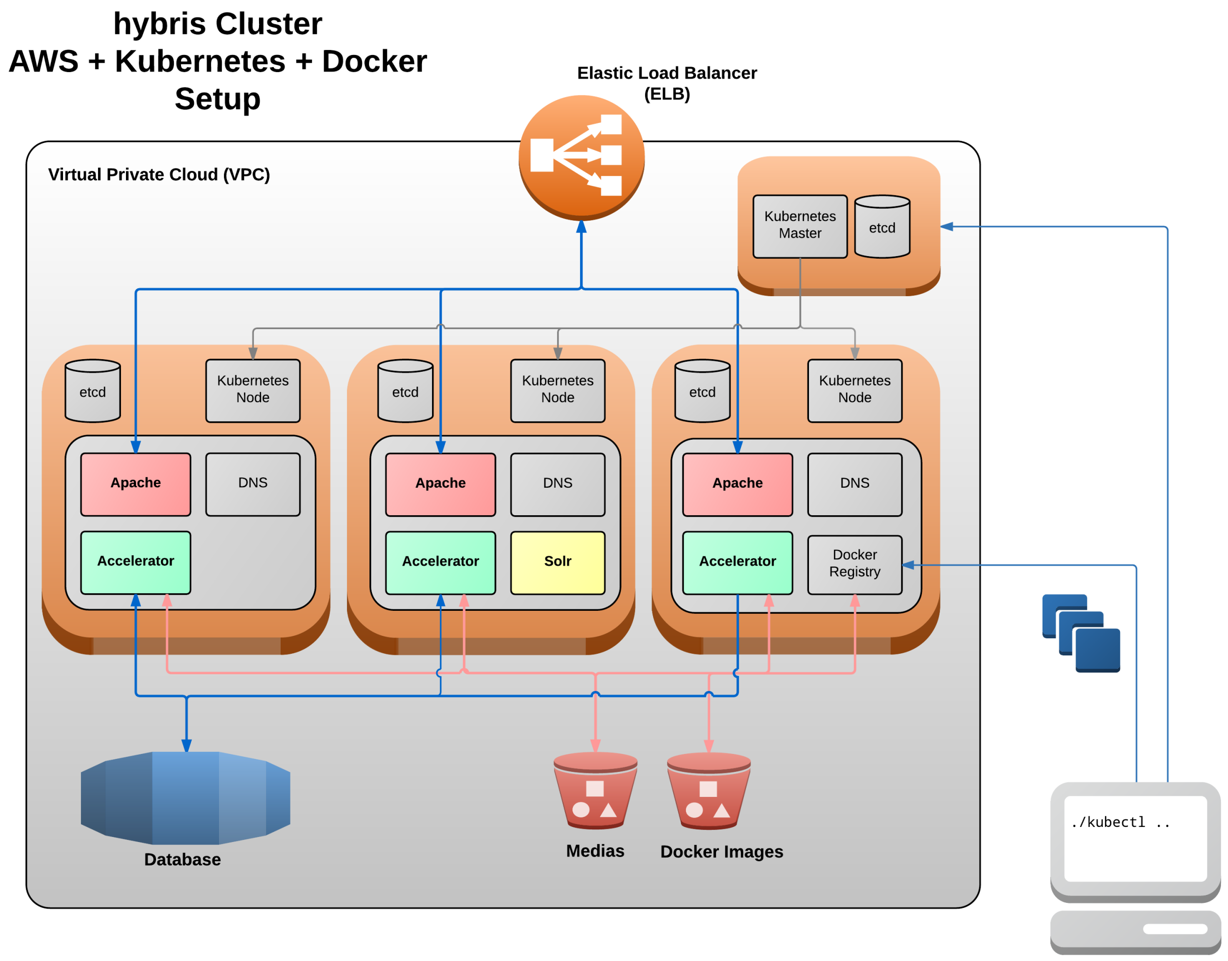
#### kube-proxy

The kube-proxy service is responsible for managing the local iptables configuration. It manages the service ip rules (see Kubernetes Service IP Routing).

## Architecture

We use an Amazon Virtual Private Cloud (VPC) as environment for our services. A VPC is like a private network within the Amazon Cloud. You may use private IP addresses and configure DHCP settings as well as routes. We attach an Elastic Load Balancer to the VPC. All http and https traffic will flow through this ELB. The ELB will also do the SSL offloading.

Kubernetes consists of two types of machines: One Kubernetes master and the Kubernetes nodes, called *minions*. The Kubernetes master uses etcd for persisting data. etcd is a distributed key-value store. We are running one etcd service on every node in our cluster to get HA. On the Kubernetes master instance we will install kube-apiserver, kube-scheduler, kube-controller-manager and kube-proxy. On the minions, we will have kube-kubelet and kube-proxy and a Docker installation.



We will setup a few peripheral services in the cluster: DNS and a Docker registry. The DNS will later serve DNS records for internal use of Kubernetes services. The Docker registry will be used to store our internal Docker containers, that we don't want to publish on Docker Hub. The Docker registry uses S3 as its BLOB store. The DNS as well as the Docker registry are applications that will be shipped in containers. They will be our first Kubernetes pods.

Further, we use an Amazon RDS MySQL database instance to run the hybris Accelerator.

Now we have the base for a hybris Accelerator cluster. From here we can deploy the hybris application and its ecosystem (for example Apache Reverse Proxy, Solr).

## yCluster Script

We have created the script *yCluster* that automates the infrastructure setup - except setting up the RDS instance and S3 bucket.

It automates the following tasks:

1. Create VPC
2. Creating DHCP options
3. Creating Internet Gateway
4. Setting up route tables
5. Creating subnets

6. Setting up security groups
7. Create an Elastic Load Balancer
8. Create an Elastic IP for the Kubernetes master
9. Create a Kubernetes Master
10. Create *n* Kubernetes Minions

### Configuration

Before executing the ycluster script, you have to define your cluster parameters in the **`ycluster.yml`** file:

```
aws:
  availability_zones:
  - eu-central-1a
  - eu-central-1b
  coreos_ami: ami-12ae980f
  key_name: <CHANGEME>
  region: eu-central-1
ycluster:
  kubernetes_version: 0.9.2
  master:
    disk_size: 40
    instance_type: t2.small
  minions:
    disk_size: 40
    instance_type: t2.small
    number: 3
  name: ycluster
```

You should always use the newest CoreOS AMI (https://coreos.com/docs/running-coreos/cloud-providers/ec2/).

Also you should insert your AWS keypair name. This has to be created by hand in advance.

If there is a new Kubernetes version, give it a try. Hopefully, it is compatible with the rest of your setup.

You can also define your minions' and your master's disk size. 40 GB is a good value for the beginning.

The number of minions can be scaled afterwards.

### Bootstrap

After you've configured your cluster in the **`ycluster.yml`**, you are ready to bootstrap the Kubernetes cluster. Export your AWS API keys to environment variables and execute ycluster:

```
export AWS_ACCESS_KEY_ID=AKIAFJHJDAKSKJ
export AWS_SECRET_ACCESS_KEY=djkasHLKHJ78kjhiu852/iufdsouodi

./ycluster bootstrap
```

yCluster will spin up your cluster. This takes about 5-10 minutes.

Afterwards, you can connect via SSH to your Kubernetes master. The master's IP is printed in the ycluster output.

To connect to the Kubernetes API, set up an ssh tunnel:

```
ssh -f -N -L 8080:localhost:8080 core@MASTERIP
```

Then you can work with kubectl from your local commandline:

```
./kubectl get minions
```

### Destroy

To destroy your cluster and terminate all instances you have created, execute

```
./ycluster destroy
```

### Scale Up

You can add another minion by executing

```
./ycluster create minion
```

## hybris Cluster Base Services

After you have set up the Kubernetes cluster, we recommend deploying a DNS service and a private Docker registry.

You will find four manifests in the directory *k8s-entities*. Just deploy them via *kubectl create*.

### Docker Registry

When you do want to use private container images, you should set up your own Docker Registry. For this we provide two Kubernetes manifests: *r egistry-rc.yml* and *registry-svc.yml*.

As the Registry uses an S3 bucket as its storage backend, you have to edit the registry-rc.yml and enter your AWS credentials and the S3 bucket name (3x <*CHANGEME*>):

```
kind: ReplicationController
apiVersion: v1beta1
id: registry
namespace: default
labels:
  k8s-app: "docker-registry"
desiredState:
  replicas: 1
  replicaSelector:
    k8s-app: "docker-registry"
  podTemplate:
    labels:
      k8s-app: "docker-registry"
    desiredState:
      manifest:
        version: v1beta2
        id: "docker-registry"
        containers:
          - name: "docker-registry"
            image: registry:latest
            env:
            - key: SETTINGS_FLAVOR
              value: s3
            - key: AWS_BUCKET
              value: <CHANGEME>
            - key: AWS_REGION
              value: eu-central-1
            - key: STORAGE_PATH
              value: "/registry"
            - key: AWS_KEY
              value: <CHANGEME>
            - key: AWS_SECRET
              value: <CHANGEME>
            - key: SEARCH_BACKEND
              value: sqlalchemy
            ports:
              - name: "docker-registry"
                containerPort: 5000
                protocol: TCP
              - name: "separate-registry-and-platform"
                hostPort: 9001
                containerPort: 9001
                protocol: TCP
          - name: "docker-registry-ui"
            image: atcol/docker-registry-ui
            env:
              - key: REG1
                value: 
http://docker-registry.default.ycluster.local:5000/v1/
            ports:
              - name: "docker-registry-ui"
                containerPort: 8080
                protocol: TCP
```

Now we can deploy the registry:

```
./kubectl create -f k8s-entities/registry-rc.yml
./kubectl create -f k8s-entities/registry-svc.yml
```

To push to your new registry, setup an ssh tunnel and push your image:

```
ssh -f -N -L 5000:registry.default.ycluster.local:5000 core@MASTERIP

# tag the image you want to upload with your registry hostname - as we are
using an ssh tunnel it is localhost:5000
docker tag -t localhost:5000/ubuntu:latest ubuntu:latest

# push your image
docker push localhost:5000/ubuntu:latest
```

Now you can use your images from within the Kubernetes cluster. The container images have names such as `registry.default.ycluster.local:5000/ubuntu:latest`

```
# so you can manually execute on the minions:
docker run -it registry.default.ycluster.local:5000/ubuntu:latest
```

### DNS Service

```
./kubectl create -f k8s-entities/dns-rc.yml
./kubectl create -f k8s-entities/dns-svc.yml
```

The DNS service will resolve service names to service IPs.

For example, you have set up a *solr* service with the service IP *10.255.23.42*. The DNS service will resolve the name *solr.default.ycluster.local* to this service IP.

The name structure is as follows:

```
service-name.namespace.ycluster.local
```

The DNS will be used within the Docker containers. It is not configured on the Kubernetes minions.

## hybris Accelerator related Services

## Apache Reverse Proxy

### Apache (yapache)

We have created an Apache container that loads its configuration from etcd.

All etcd keys below `/yapache` will be monitored. As soon as the value `/yapache/*.conf` is changed, the value will be written to `/etc/apache/sites-enabled/*.conf` and apache will be reloaded. Also if a new key/value pair is created, it will be added to the Apache configuration directory. When a pair is removed the corresponding configuration file will be deleted. Then Apache will be reloaded.

The Apache Pod exposes ports 8080 and 8443 to the host's outside. The Elastic Load Balancer uses those ports to route traffic inside the Kubernetes cluster.

```
kind: ReplicationController
apiVersion: v1beta1
id: yapache
namespace: default
labels:
  k8s-app: yapache
desiredState:
  replicas: 3
  replicaSelector:
    k8s-app: yapache
  podTemplate:
    labels:
      k8s-app: yapache
    desiredState:
      manifest:
        version: v1beta2
        id: yapache
        containers:
          - name: yapache
            image: registry.default.ycluster.local:5000/yapache:latest
            env:
              - key: ETCD_PEERS
                value: "http://172.31.0.10:4001"
            ports:
              - name: platform
                containerPort: 8080
                hostPort: 8080
                protocol: TCP
              - name: platform-secure
                containerPort: 8443
                hostPort: 8443
                protocol: TCP
```

### Apache Configuration Updater (yapache-k8s-watcher)

The *yapache-k8s-watcher* is responsible for keeping the Apache configuration in etcd `/yapache/*.conf` up to date. It watches the Kubernetes API for changes of the `hybris-platform` service endpoints. As soon as something changes, *yapache-k8s-watcher* generates a new configuration and saves it to `/yapache/hybris.conf`. As described above, the Apache watches this etcd directory and updates its configuration as soon as something changes.

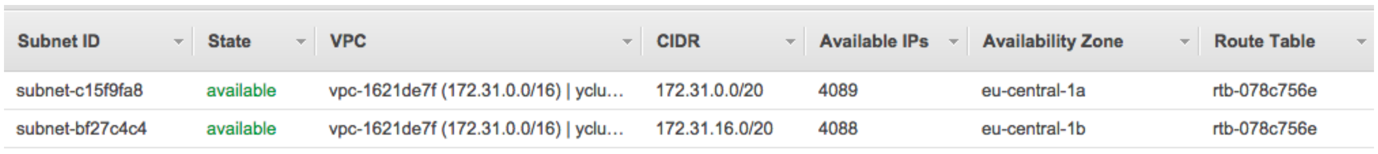
You can see/edit the Apache configuration template in the ycluster repository at `Dockerfiles/yapache-k8s-watcher/apacheconfig.template`

```
Listen 8080
Listen 8443
<VirtualHost *:8080>
    Header add Set-Cookie "ROUTEID=.%{BALANCER_WORKER_ROUTE}e; path=/"
env=BALANCER_ROUTE_CHANGED
    <Location /balancer-manager>
        SetHandler balancer-manager
    </Location>
    <Proxy balancer://ycluster>
    {{ range $endpoint := .Endpoints }}
        BalancerMember http://{{ $endpoint }}:9001 route={{ $endpoint }}
    {{ end }}
        ProxySet stickysession=ROUTEID
    </Proxy>
    ProxyPass / balancer://ycluster/ nocanon
    ProxyPassReverse / balancer://ycluster/
    ProxyTimeout 60
    ProxyPreserveHost On
</VirtualHost>
<VirtualHost *:8443>
    Header add Set-Cookie "ROUTEID=.%{BALANCER_WORKER_ROUTE}e; path=/"
env=BALANCER_ROUTE_CHANGED
    <Location /balancer-manager>
        SetHandler balancer-manager
    </Location>
    <Proxy balancer://ycluster>
    {{ range $endpoint := .Endpoints }}
        BalancerMember http://{{ $endpoint }}:9002 route={{ $endpoint }}
    {{ end }}
        ProxySet stickysession=ROUTEID
    </Proxy>
    ProxyPass / balancer://ycluster/ nocanon
    ProxyPassReverse / balancer://ycluster/
    ProxyTimeout 60
    ProxyPreserveHost On
</VirtualHost>
```

## Networking Background

### Subnets

We use multiple subnets within the Kubernetes environment. All Virtual Machines have an IP address in the `172.31.0.0/20` or `172.31.16.0/20` IPv4 subnet - depending on their EC2 availability zone:



| Subnet ID | State | VPC | CIDR | Available IPs | Availability Zone | Route Table |
|---|---|---|---|---|---|---|
| subnet-c15f9fa8 | available | vpc-1621de7f (172.31.0.0/16) \| yclu… | 172.31.0.0/20 | 4089 | eu-central-1a | rtb-078c756e |
| subnet-bf27c4c4 | available | vpc-1621de7f (172.31.0.0/16) \| yclu… | 172.31.16.0/20 | 4088 | eu-central-1b | rtb-078c756e |

For container IP addresses the subnet `10.0.0.0/8` is used. Every Virtual Machine has a `/24` subnet assigned from which it will allocate addresses to its containers. The container subnet is built from the VM's IPv4 address as follows:

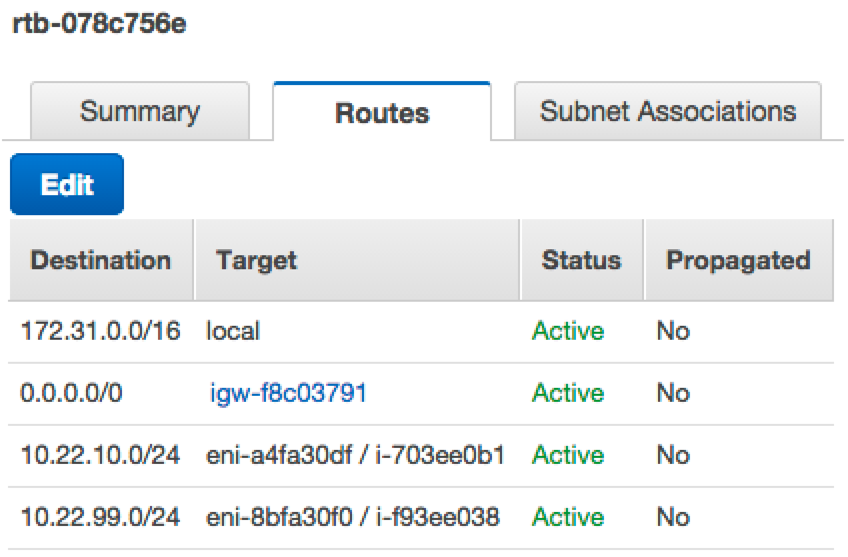
```
containerSubnet = "10.vmAddress[3].vmAddress[4].0/24"
```

For example, the VM with IP `172.31.16.23` is responsible for the subnet `10.16.23.0/24`.

The subnet `10.255.0.0/16` will be used for Kubernetes Service IPs.

## Routing

EC2 manages routing information within VPCs (Virtual Private Cloud) via VPC Route Tables. All subnets are registered in those Route Tables by the *ycluster* Script.



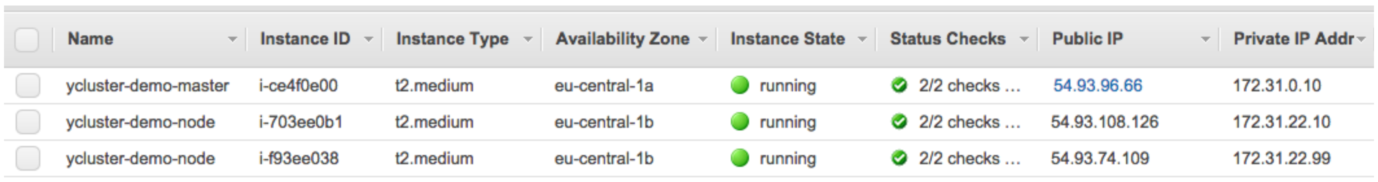
rtb-078c756e

Summary | Routes | Subnet Associations

Edit

| Destination | Target | Status | Propagated |
|---|---|---|---|
| 172.31.0.0/16 | local | Active | No |
| 0.0.0.0/0 | igw-f8c03791 | Active | No |
| 10.22.10.0/24 | eni-a4fa30df / i-703ee0b1 | Active | No |
| 10.22.99.0/24 | eni-8bfa30f0 / i-f93ee038 | Active | No |

Here you can see the `172.16.0.0/16` subnet that holds all EC2 instances. Further there is a route to the internet (`0.0.0.0/0`) via an EC2 internet gateway. The last two lines show routes to container the subnets `10.22.10.0/24` and `10.22.99.0/24`. These container subnet routes use EC2 instances as targets.



| Name | Instance ID | Instance Type | Availability Zone | Instance State | Status Checks | Public IP | Private IP Addr |
|---|---|---|---|---|---|---|---|
| ycluster-demo-master | i-ce4f0e00 | t2.medium | eu-central-1a | ● running | ✓ 2/2 checks … | 54.93.96.66 | 172.31.0.10 |
| ycluster-demo-node | i-703ee0b1 | t2.medium | eu-central-1b | ● running | ✓ 2/2 checks … | 54.93.108.126 | 172.31.22.10 |
| ycluster-demo-node | i-f93ee038 | t2.medium | eu-central-1b | ● running | ✓ 2/2 checks … | 54.93.74.109 | 172.31.22.99 |

The instance overview shows that the instance `i-803ee0b1` that is associated with the subnet `10.22.10.0/24` has the Private IP address `172.31.22.10` and the instance `i-f98ee038` that is the target for the `10.22.99.0/24` route has the private IP address `172.31.22.99`. This follows the schema described under *Subnets*.

## Kubernetes Service IP Routing

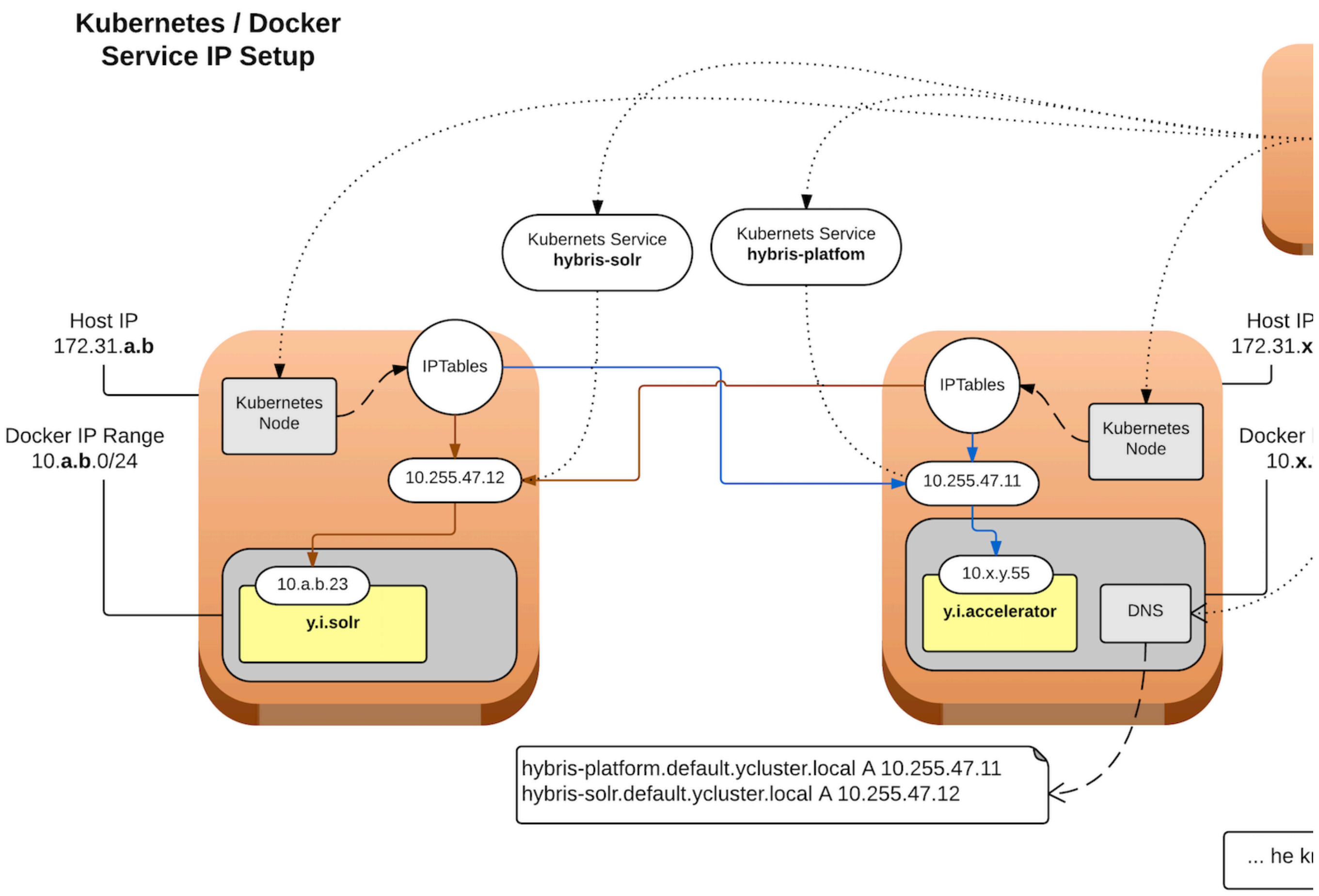
Kubernetes Service IP addresses reside in the subnet `10.255.0.0/16`. This subnet cannot be found in the routing table. Service IP addresses are virtual. They will be mapped by *iptables* on the Kubernetes minions to their target IP addresses.

A *Kubernetes Service* has multiple endpoints - containers that expose this service. The Kubernetes executable *kube-proxy* creates DNAT *iptables* entries that redirect traffic to those addresses to the service's endpoint addresses. If there are multiple endpoints, *kube-proxy* will distribute the traffic using round-robin.

```
$ iptables-save
# ...
-A KUBE-PORTALS-CONTAINER -d 10.255.0.2/32 -p udp -m comment --comment
skydns -m udp --dport 53 -j REDIRECT --to-ports 33968
-A KUBE-PORTALS-CONTAINER -d 10.255.188.255/32 -p tcp -m comment --comment
hybris-platform -m tcp --dport 9001 -j REDIRECT --to-ports 50084
-A KUBE-PORTALS-HOST -d 10.255.0.2/32 -p udp -m comment --comment skydns -m
udp --dport 53 -j DNAT --to-destination 172.31.0.10:33968
-A KUBE-PORTALS-HOST -d 10.255.188.255/32 -p tcp -m comment --comment
hybris-platform -m tcp --dport 9001 -j DNAT --to-destination
172.31.0.10:50084
```

Here you can see the iptables rules for the services *skydns* and *hybris-portal*. Those rules can be found on every Kubernetes minion. If you connect to the service IP `10.255.0.2` on Port `53/UDP` iptables will reroute the traffic to the container with the IP `172.31.0.10` and port 33968.

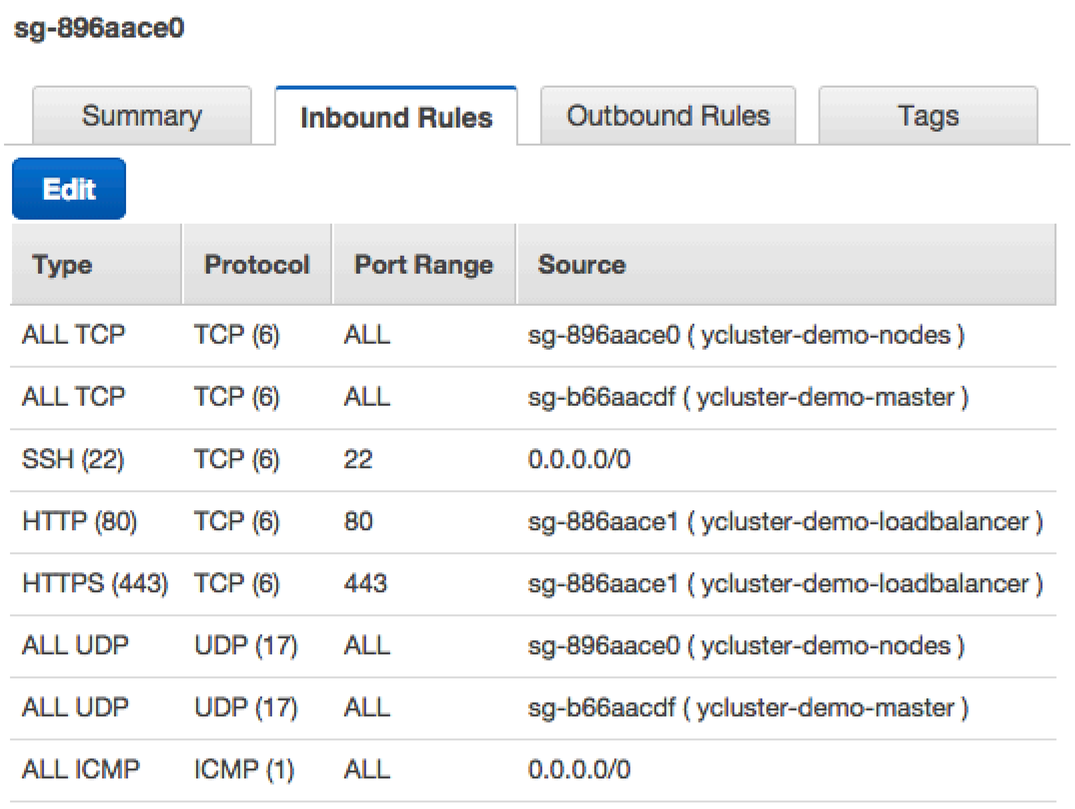
**Service IPs**
iptables will only reroute ports that are specified in the Kubernetes Service definition. **This means you *cannot* ping Service IPs!**


Kubernetes / Docker Service IP Setup
Kubernets Service hybris-solr
Kubernets Service hybris-platfom
Host IP 172.31.a.b
Docker IP Range 10.a.b.0/24
Kubernetes Node
IPTables
10.255.47.12
10.a.b.23
y.i.solr
IPTables
Kubernetes Node
10.255.47.11
10.x.y.55
y.i.accelerator
DNS
Host IP 172.31.x
Docker
10.x.
hybris-platform.default.ycluster.local A 10.255.47.11
hybris-solr.default.ycluster.local A 10.255.47.12
... he k

## Security Groups

### Kubernetes Nodes

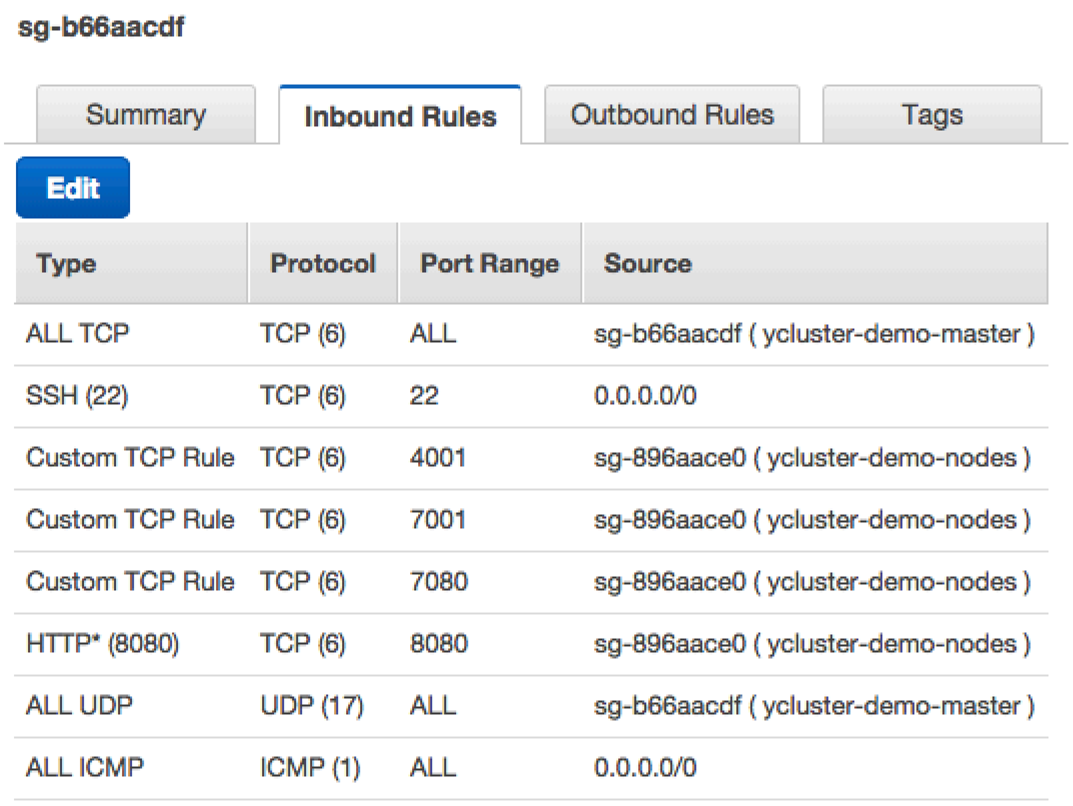
- Allow all TCP and UDP traffic from other Kubernetes Nodes
- Allow all TCP and UDP traffic from the Kubernetes Master
- Allow SSH from everywhere
- Allow HTTP and HTTPS from the Load Balancer
- Allow all ICMP from everywhere



| Type | Protocol | Port Range | Source |
|---|---|---|---|
| ALL TCP | TCP (6) | ALL | sg-896aace0 ( ycluster-demo-nodes ) |
| ALL TCP | TCP (6) | ALL | sg-b66aacdf ( ycluster-demo-master ) |
| SSH (22) | TCP (6) | 22 | 0.0.0.0/0 |
| HTTP (80) | TCP (6) | 80 | sg-886aace1 ( ycluster-demo-loadbalancer ) |
| HTTPS (443) | TCP (6) | 443 | sg-886aace1 ( ycluster-demo-loadbalancer ) |
| ALL UDP | UDP (17) | ALL | sg-896aace0 ( ycluster-demo-nodes ) |
| ALL UDP | UDP (17) | ALL | sg-b66aacdf ( ycluster-demo-master ) |
| ALL ICMP | ICMP (1) | ALL | 0.0.0.0/0 |

### Kubernetes Master

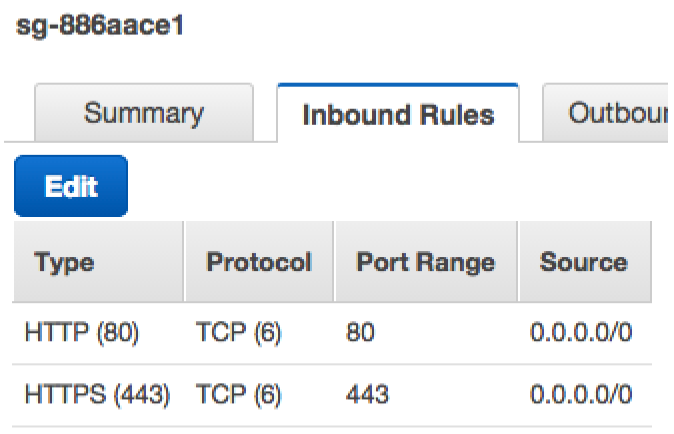
- Allow all TCP and UDP traffic from other Kubernetes Masters
- Allow SSH from everywhere
- Allow accessing etcd from the Kubernetes Nodes
- Allow accessing Kubernetes APIs from the Kubernetes Nodes
- Allow all ICMP from everywhere



| Type | Protocol | Port Range | Source |
|---|---|---|---|
| ALL TCP | TCP (6) | ALL | sg-b66aacdf ( ycluster-demo-master ) |
| SSH (22) | TCP (6) | 22 | 0.0.0.0/0 |
| Custom TCP Rule | TCP (6) | 4001 | sg-896aace0 ( ycluster-demo-nodes ) |
| Custom TCP Rule | TCP (6) | 7001 | sg-896aace0 ( ycluster-demo-nodes ) |
| Custom TCP Rule | TCP (6) | 7080 | sg-896aace0 ( ycluster-demo-nodes ) |
| HTTP* (8080) | TCP (6) | 8080 | sg-896aace0 ( ycluster-demo-nodes ) |
| ALL UDP | UDP (17) | ALL | sg-b66aacdf ( ycluster-demo-master ) |
| ALL ICMP | ICMP (1) | ALL | 0.0.0.0/0 |

### Load Balancer

- Allow HTTP and HTTPS from everywhere

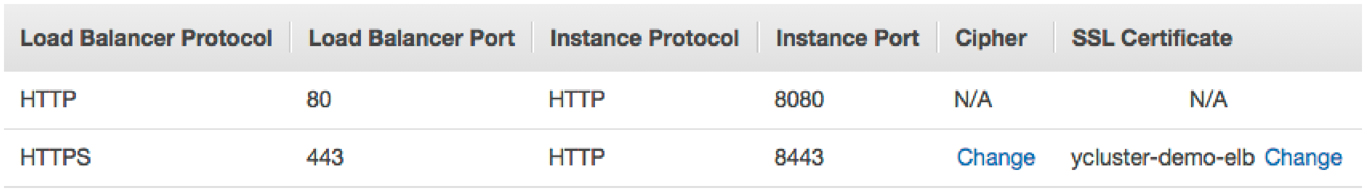


| Type | Protocol | Port Range | Source |
|---|---|---|---|
| HTTP (80) | TCP (6) | 80 | 0.0.0.0/0 |
| HTTPS (443) | TCP (6) | 443 | 0.0.0.0/0 |

## Load Balancer

We use an Amazon Elastic Load Balancer (ELB) to redirect incoming HTTP and HTTPS traffic to Apache containers that are running on the Kubernetes Minions.

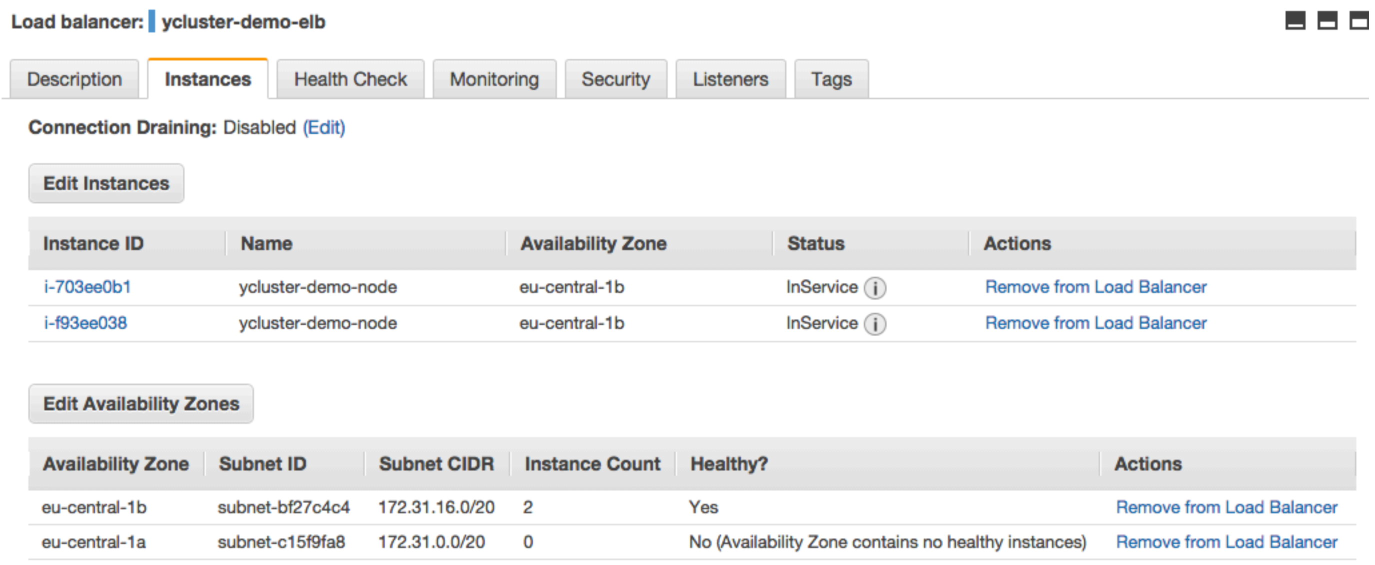
Every Apache Container exposes the ports 8080 and 8443 on its host VM. So if you connect the VM from the outside on port 8080/8443, you will actually connect to the Apache container. We use this to add the VMs as the ELB endpoints.



| Load Balancer Protocol | Load Balancer Port | Instance Protocol | Instance Port | Cipher | SSL Certificate |
|---|---|---|---|---|---|
| HTTP | 80 | HTTP | 8080 | N/A | N/A |
| HTTPS | 443 | HTTP | 8443 | Change | ycluster-demo-elb Change |

Further we do SSL offloading on the Elastic Load Balancer. All traffic between ELB and VM/Container is unencrypted. We redirect the traffic that was encrypted to port 8443 whereas the unencrypted traffic will go to 8080. This way, we can distinguish between secure and insecure requests without additional headers.

The Elastic Load Balancer distributes incoming traffic in a round-robin fashion to the Apache containers. We don't do sticky sessions on the ELB. This will be handled by the Apaches.



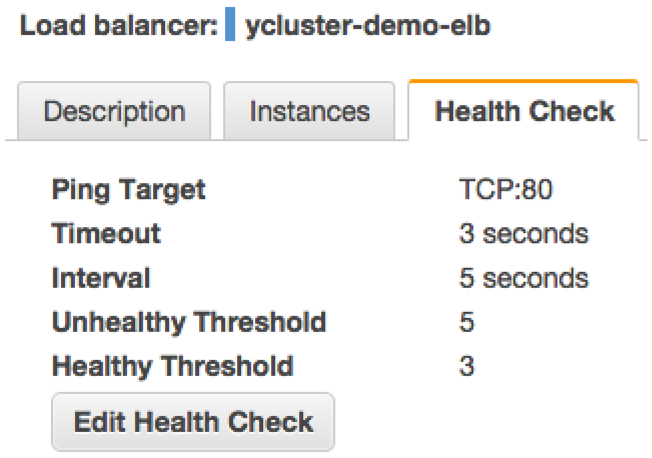
**Load balancer:** ycluster-demo-elb

**Connection Draining:** Disabled (Edit)

| Instance ID | Name | Availability Zone | Status | Actions |
|---|---|---|---|---|
| i-703ee0b1 | ycluster-demo-node | eu-central-1b | InService | Remove from Load Balancer |
| i-f93ee038 | ycluster-demo-node | eu-central-1b | InService | Remove from Load Balancer |

| Availability Zone | Subnet ID | Subnet CIDR | Instance Count | Healthy? | Actions |
|---|---|---|---|---|---|
| eu-central-1b | subnet-bf27c4c4 | 172.31.16.0/20 | 2 | Yes | Remove from Load Balancer |
| eu-central-1a | subnet-c15f9fa8 | 172.31.0.0/20 | 0 | No (Availability Zone contains no healthy instances) | Remove from Load Balancer |

The ELB performs health checks on the Apaches. For our use case, it is sufficient to check if the target Port 8080 is open. We don't want to check the Tomcat behind the Apache, we just want to check the Apache.



**Load balancer:** ycluster-demo-elb

| | |
|---|---|
| Ping Target | TCP:80 |
| Timeout | 3 seconds |
| Interval | 5 seconds |
| Unhealthy Threshold | 5 |
| Healthy Threshold | 3 |

## CoreOS

CoreOS is the Linux base we rely on in our cluster. CoreOS's job is to provide etcd, the Kubernetes services and Docker. We use cloud-config scr ipts to setup the CoreOS hosts during startup.

### Cloud-Config

## Master

Via Cloud-Config we install our SSH keys on the master machine.

Besides, we add some services to systemd:

- etcd
- fleet
- docker
- kube-apiserver
- kube-controller-manager
- kube-scheduler
- kube-proxy

and two custom helper services:

- etcd-environment
- kube-register

*etcd-environment* is a BASH script that gets some network environment information and writes them to `/etc/network-environment` so they can be used as environment variables later.

**etcd-environment.sh**

```
#!/bin/bash
set -e
set -x

ENV_FILE=/etc/network-environment

source /etc/environment

PRIVATE_IPV4=$COREOS_PRIVATE_IPV4
PUBLIC_IPV4=$COREOS_PUBLIC_IPV4

function setVar {
  echo $1=$2 >> $ENV_FILE
  source $ENV_FILE
}

setVar PRIVATE_IPV4  $PRIVATE_IPV4

setVar DEFAULT_IPV4  $PRIVATE_IPV4
setVar ETH0_IPV4     $PRIVATE_IPV4
setVar LO_IPV4       127.0.0.1

setVar PUBLIC_IPV4   $PUBLIC_IPV4


setVar DOCKER_BIP    "10.`echo $PRIVATE_IPV4 | cut -d. -f3,4`.1/24"
setVar DOCKER_SUBNET "10.`echo $PRIVATE_IPV4 | cut -d. -f3,4`.0/24"

setVar FLEET_PUBLIC_IP $PRIVATE_IPV4
setVar FLEET_METADATA "DOCKER_SUBNET=$DOCKER_SUBNET,HOSTNAME=$HOSTNAME"
```

*kube-register* reads the nodes that are registered in fleet and adds them to the Kubernetes cluster as new minions.

Here you see an example of a cloud-config file. The full cloud-config files for the nodes we're using in the cluster can be found in the ycluster repository. Please note that the cloud-config files in the ycluster repo are templates that are filled by the ycluster script during the cluster bootstrap / creating a node!

```
#cloud-config
ssh_authorized_keys:
  - <YOUR_SSH_KEY_HERE>
  - <YOUR_SSH_KEY_HERE>
coreos:
  units:
  - name: etcd.service
    command: start
    content: |
      [Unit]
      Description=etcd Server
      [Service]
      ExecStart=/bin/etcd \
      --addr $private_ipv4:4001 \
      --bind-addr 0.0.0.0:4001 \
      --peer-addr $private_ipv4:7001 \
      --peer-bind-addr 0.0.0.0:7001 \
      --discovery {{ ycluster.etcd_discovery }}
      Restart=always
      RestartSec=10s
  - name: etcd-environment.service
    command: start
    content: |
      [Unit]
      Description=Get Machine Configuration Environment Variables
      Requires=etcd.service
      After=etcd.service
      [Service]
      ExecStartPre=/usr/bin/mkdir -p /opt/bin
      ExecStartPre=/usr/bin/wget -O /opt/bin/etcd-get-environment
https://s3.eu-central-1.amazonaws.com/hybris-docker/tools/etcd-environment
/2014-12-26/etcd-environment.sh
      ExecStartPre=/usr/bin/chmod +x /opt/bin/etcd-get-environment
      ExecStartPre=/usr/bin/sleep 2
      ExecStartPre=/opt/bin/etcd-get-environment
      ExecStart=/usr/bin/echo "Machine configuration written"
      RemainAfterExit=yes


...
```

## Nodes

On the nodes, we install the following services:

- etcd
- etcd-environment
- fleet
- docker
- kube-proxy

- kube-kubelet

Let's have a look at the Docker service as we have to define the node's subnet here:

```
...

  - name: docker.service
    command: start
    content: |
      [Unit]
      Description=Docker Application Container Engine
      Documentation=http://docs.docker.io
      Requires=etcd-environment.service
      After=etcd-environment.service
      Requires=registry-hosts-entry.service
      After=registry-hosts-entry.service
      [Service]
      EnvironmentFile=/etc/network-environment
      ExecStartPre=/sbin/iptables -t nat -A POSTROUTING -s${DOCKER_BIP} !
-o docker0 ! -d 10.0.0.0/8 -j MASQUERADE
      ExecStart=/usr/bin/docker -d --bip=${DOCKER_BIP} -H fd:// --dns
10.255.0.2 --ip-masq=false
      [Install]
      WantedBy=multi-user.target

...
```

In the *[Service]* section we see that systemd will load the network information from */etc/network-environment* so we can use the variables later.

Before starting Docker, we add iptables rules to enable NAT from the Docker containers to the network outside of the 10.0.0.0/8 subnet (aka Internet). Then we start the Docker daemon and define it's bridge IP (which will define the subnet where the Docker containers' IP addresses will be selected from) and to use Kubernetes SkyDNS service within the containers. We select `--ip-masq=false` because we have set up the iptables NAT manually.