W. Caban, *Architecting and Operating OpenShift Clusters*
https://doi.org/10.1007/978-1-4842-4985-7_2

2. High Availability

William Caban [1]

(1) Columbia, MD, USA

As we saw from Chapter 1 , OpenShift Container Platform is comprised of multiple elements build on top of Kubernetes. When designing production environments, we should understand the high availability (HA) built into the different elements of the platform. Each one of the HA elements can be scaled independently.

The desired level of HA for each platform element and how a cluster will be scaled out over time may have a direct influence in the initial design considerations.

In this chapter, we will cover the HA configurations, what may be considered the most relevant elements of the OCP architecture, but the reader should keep in mind there might be many other components which are not covered here.

Control Plane and Data Plane

From the OpenShift and Kubernetes perspective, there is a clear definition of the Control Plane, but, when it comes to the Data Plane, it is loosely defined and its definition is normally based on the context it is being used. To avoid confusion, this is the way we use the terms here:

- **OpenShift Control Plane** : The *OCP Control Plane* is comprised of the Kubernetes Control Plane [1] (Kubernetes Master [2] and the *kubelet* process in each node). For the purpose of this book, we are considering the OpenShift consoles, logging, metrics, and cluster monitoring services as part of this plane.

- **OpenShift Data Plane**: The term *OCP Data Plane*, even when not officially defined in the OKD and OCP documentation, is normally used to describe the traffic forwarding plane of the SDN layer.

NOTE The terms *Control Plane*, *Management Plane*, and *Data Plane* have a clear separation of concerns when used in computing, networking, and telecommunications systems. There is no direct mapping of the Kubernetes constructs into these concepts. Kubernetes, as a project, does not provide a clear separation of concerns between the functions that would normally go into the *Control Plane* and those that go into the *Management Plane*. When considering the OpenShift architecture, we could clearly map OpenShift components for each one of these planes.

For example, the *OpenShift Cluster Console* is what would normally be considered part of the *Management Plane*. Unfortunately, for those of us used to architecting solutions with these differentiations, the terms *Management Plane* and *Data Plane* have not been officially adopted by the OKD and OpenShift community.

HA for Control Plane

The elements of the OpenShift Control Plane are protected in different ways, and as such, to achieve high availability differs for each one.

HA FOR ETCD

The *etcd* database is one of the critical components of the Kubernetes. It is used to store status and details of the Kubernetes objects, store information and status of the *Nodes*, scheduler results, and much more.

From the technical point of view, *etcd* is a distributed *key-value* store using the *RAFT* consensus algorithm.

Because of the *consensus* required by the *RAFT algorithm* (see details in section “*RAFT Consensus Algorithm*”), the *etcd* service must be deployed in odd numbers to maintain *quorum*. For this reason, the minimum number of *etcd* instances for production environments is three.

NOTE Using one instance is considered a testing or demo environment as it is a single point of failure.

From the operational aspects of *etcd*, the *etcd* service is considered an *active-active* cluster. Meaning, an *etcd Client* can write to any of the *etcd* nodes and the cluster will replicate the data and maintain consistency of the data across the instances.

Failures of the etcd database can be classified under one of the following scenarios:

1. **Losing the *etcd Leader* or losing less than (N-1)/2 nodes of an N size**

etcd cluster: These are considered temporary failures from which the cluster recovers automatically.

2. **Losing *etcd* quorum**: This failure happens when the cluster loses more than (N-1)/2 nodes of the etcd cluster. This is a major failure as once the quorum is lost, the cluster is incapable of reaching consensus and cannot accept any additional update. When this failure happens, applications already running on OCP are unaffected. However, the platform functionality is limited to read-only operations. Under this failure scenario, it is not possible to take actions such as scaling an application up or down, changing deployment objects, and running or modifying builds.

3. **Losing the data of *etcd* cluster**: Losing the data from the *etcd* cluster will render the Kubernetes and OCP cluster unusable. The only way to recover from this failure scenario is by restoring the *etcd* data from backup.

From the deployment aspect, the *etcd* service can be colocated in the *Master Nodes* with other master services. It is a common practice to colocate the *etcd* service in the *Master Nodes*. In this case, a minimum of three Master nodes is required. The minimum of three Masters is because the *etcd* deployment must guarantee quorum so the *etcd RAFT* protocol can reach consensus in the case of a Node failure.

NOTE Up to OpenShift 3.11, there is the option to have external dedicated etcd Nodes. Starting with OpenShift 4.0, the etcd service will always be on the cluster.

From the implementation perspective, in OpenShift 3.11 the etcd instances are deployed as a series of privileged Pods running in the *kube-system* project[3] or namespace.[4] Additional details are covered in “HA for Masters Services” section.

RAFT Consensus Algorithm

The basis of the RAFT algorithm states that for any action (add, remove, update, etc.) to be accepted, there needs to be quorum. Quorum is decided by having a number of voting members greater than 50% of the total number of *etcd* instances or *Nodes*. For example, with three etcd Nodes, a minimum of two etcd Nodes are required to have quorum and achieve consensus.

The RAFT Consensus Algorithm consists of three states:

1. Follower

2. Candidate

3. Leader

There are two timeout settings which control the process of the election of a Leader node in the RAFT algorithm:

1. **Election Timeout**: The time a Follower waits before becoming a Candidate. This is a random number between 150ms and 300ms.

2. **Heartbeat Timeout**: Regular interval of time a Leader sends *Append Entries* messages to *Followers* to replicate logs.

All nodes start in the *Follower* state (see Step 1 of Figure 2-1). The nodes wait for *Election Timeout*. If a *Follower* doesn't hear from a *Leader* in *Election Timeout*, they can become *Candidate* (see Step 2 of Figure 2-1) and initiate new *Election Term*. The *Candidate* node votes for itself and *Request Votes* from the other nodes. If the receiving node hasn't voted yet in this *Election Term*, then it votes for the candidate and resets its *Election Timeout*. A *Candidate* becomes *Leader* if it gets the majority of the votes from the nodes (see Step 3 of Figure 2-1).

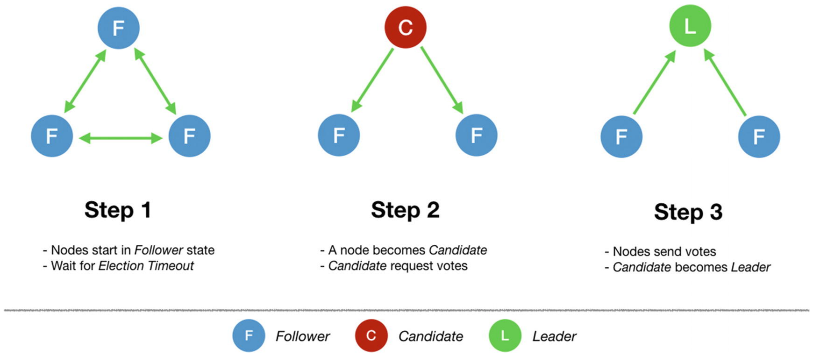
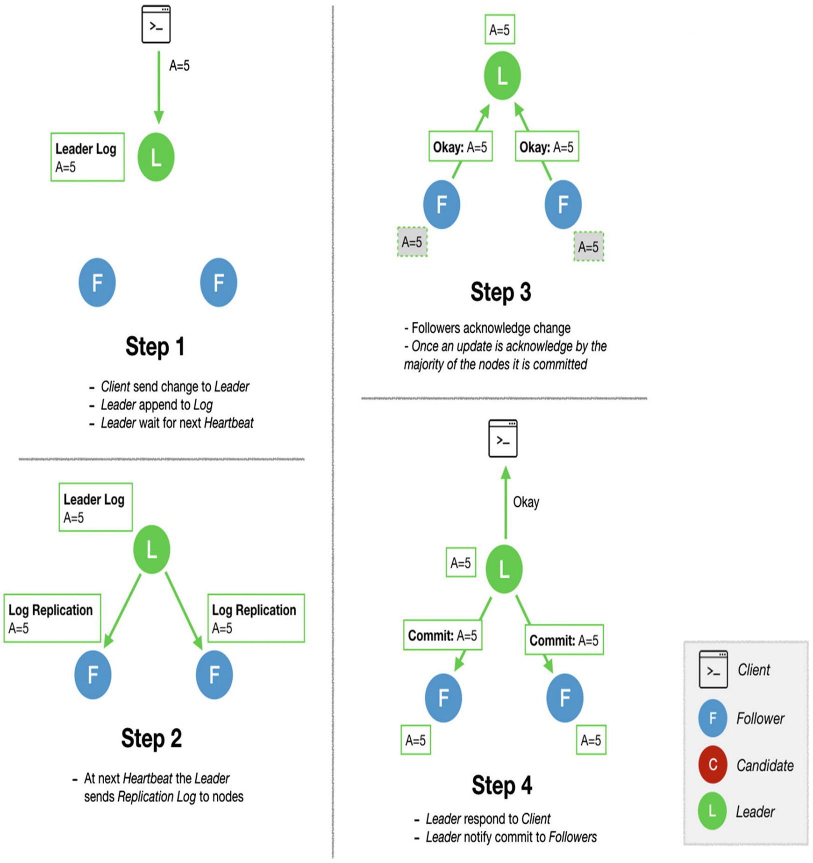

F
F
F
Step 1
- Nodes start in *Follower* state
- Wait for *Election Timeout*
C
F
F
Step 2
- A node becomes *Candidate*
- *Candidate* request votes
L
F
F
Step 3
- Nodes send votes
- *Candidate* becomes *Leader*
F Follower
C Candidate
L Leader

Figure 2-1 The RAFT algorithm Leader election process

Once a *Leader* is elected, all changes to the system go through the *Leader*. A client sends a change to the *Leader*. The *Leader* appends this to the *Replication Log* (see Step 1 of Figure 2-2). The change is sent to the *Followers* on the next *Heartbeat* (see Step 2 of Figure 2-2). Once an entry is committed and acknowledged by the majority of the *Followers* (see Step 3 of Figure 2-2), the cluster has reached *Consensus*. A response is sent to the client (see Step 4 of Figure 2-2).

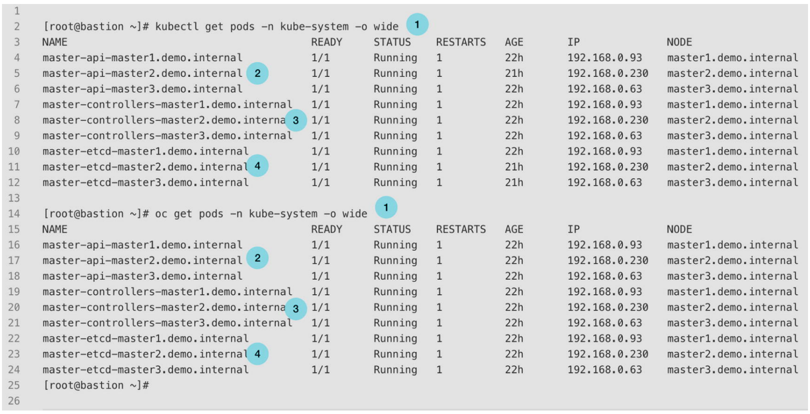


A=5
Leader Log
A=5
L
F
F
Step 1
- Client send change to Leader
- Leader append to Log
- Leader wait for next Heartbeat
Leader Log
A=5
L
Log Replication
A=5
Log Replication
A=5
F
F
Step 2
- At next Heartbeat the Leader sends Replication Log to nodes
A=5
L
Okay: A=5
Okay: A=5
F
F
A=5
A=5
Step 3
- Followers acknowledge change
- Once an update is acknowledge by the majority of the nodes it is committed
Okay
A=5
L
Commit: A=5
Commit: A=5
F
F
A=5
A=5
Step 4
- Leader respond to Client
- Leader notify commit to Followers
Client
F Follower
C Candidate
L Leader

Figure 2-2 Update value in RAFT algorithm

HA FOR MASTER SERVICES

When we talk about Master services , we are referring mainly to the API Server, the Controllers, and the etcd service. In OpenShift these services are deployed as privileged containers and pods (see Figure 2-3).



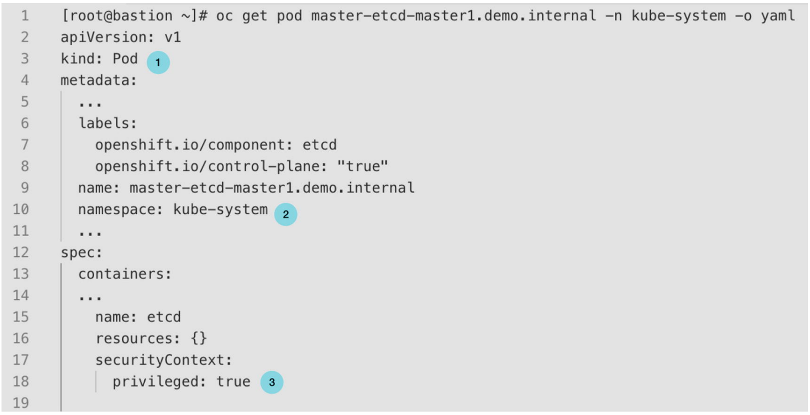
```
[root@bastion ~]# kubectl get pods -n kube-system -o wide
NAME                                       READY   STATUS    RESTARTS   AGE    IP              NODE
master-api-master1.demo.internal           1/1     Running   1          22h    192.168.0.93    master1.demo.internal
master-api-master2.demo.internal           1/1     Running   1          21h    192.168.0.230   master2.demo.internal
master-api-master3.demo.internal           1/1     Running   1          22h    192.168.0.63    master3.demo.internal
master-controllers-master1.demo.internal   1/1     Running   1          22h    192.168.0.93    master1.demo.internal
master-controllers-master2.demo.interna    1/1     Running   1          22h    192.168.0.230   master2.demo.internal
master-controllers-master3.demo.internal   1/1     Running   1          22h    192.168.0.63    master3.demo.internal
master-etcd-master1.demo.internal          1/1     Running   1          22h    192.168.0.93    master1.demo.internal
master-etcd-master2.demo.internal          1/1     Running   1          21h    192.168.0.230   master2.demo.internal
master-etcd-master3.demo.internal          1/1     Running   1          21h    192.168.0.63    master3.demo.internal

[root@bastion ~]# oc get pods -n kube-system -o wide
NAME                                       READY   STATUS    RESTARTS   AGE    IP              NODE
master-api-master1.demo.internal           1/1     Running   1          22h    192.168.0.93    master1.demo.internal
master-api-master2.demo.internal           1/1     Running   1          22h    192.168.0.230   master2.demo.internal
master-api-master3.demo.internal           1/1     Running   1          22h    192.168.0.63    master3.demo.internal
master-controllers-master1.demo.internal   1/1     Running   1          22h    192.168.0.93    master1.demo.internal
master-controllers-master2.demo.interna    1/1     Running   1          22h    192.168.0.230   master2.demo.internal
master-controllers-master3.demo.internal   1/1     Running   1          22h    192.168.0.63    master3.demo.internal
master-etcd-master1.demo.internal          1/1     Running   1          22h    192.168.0.93    master1.demo.internal
master-etcd-master2.demo.internal          1/1     Running   1          22h    192.168.0.230   master2.demo.internal
master-etcd-master3.demo.internal          1/1     Running   1          22h    192.168.0.63    master3.demo.internal
[root@bastion ~]#
```

Figure 2-3 The kube-system namespace or project

From what can be seen in Figure 2-3, the *kube-system* namespace or project host the containers Pods for the *API Server* (see #2 of Figure 2-3), the *Controllers* (see #3 of Figure 2-3), and the *etcd* (see #4 of Figure 2-3) instances. Each *Master Node* contains an *API Server Pod*, a *Controller Pods* and an *etcd Pods*.

NOTE The reader may notice the output in Figure 2-3 is the same when using the *kubectl* or the *oc*[5] command-line interfaces. The *kubectl* is the official Kubernetes CLI and *oc* is the OpenShift CLI. The oc CLI includes the *kubectl* and shares the same syntax. In addition to the standard features, the *oc* CLI extends capabilities and brings native support to OCP features like authentication, routes, DeploymentConfigs, ImageStreams, and others.

Looking into the details of one of these Pods, the *etcd Pods*, we can clearly see they are running as privileged containers (see #3 of Figure 2-4), and they are running in the *kube-system* (see #2 of Figure 2-4) namespace or project.

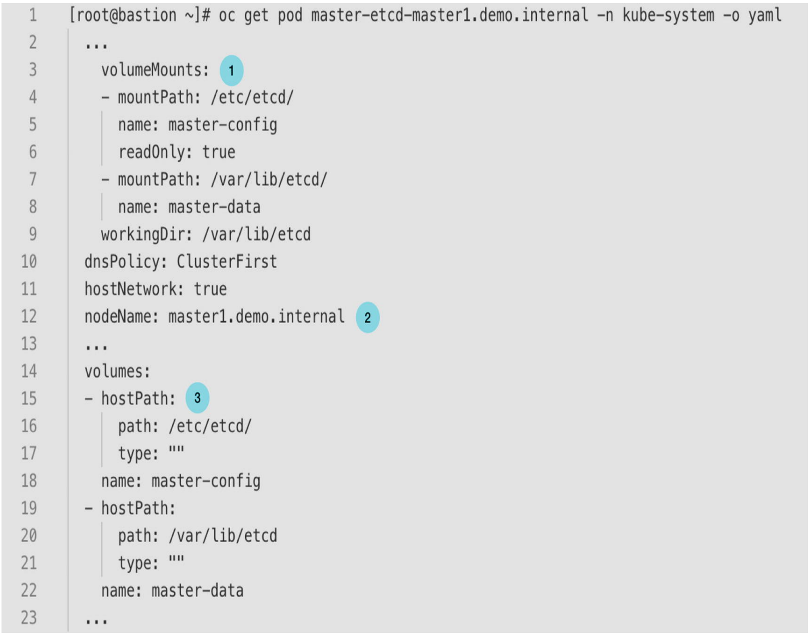


```
[root@bastion ~]# oc get pod master-etcd-master1.demo.internal -n kube-system -o yaml
apiVersion: v1
kind: Pod  1
metadata:
  ...
  labels:
    openshift.io/component: etcd
    openshift.io/control-plane: "true"
  name: master-etcd-master1.demo.internal
  namespace: kube-system  2
  ...
spec:
  containers:
  ...
    name: etcd
    resources: {}
    securityContext:
      privileged: true  3

```

Figure 2-4 Details of the etcd Pod definition highlighting the privileged mode

One of the reasons these Pods need the privileged access is because they access host resources. As we can see in Figures 2-5, 2-6, and 2-7, some host resources are mapped as volumes to the containers.

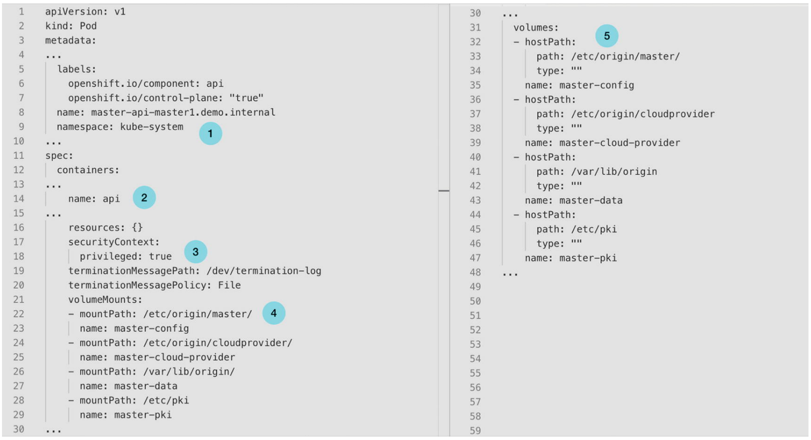
The details of the *etcd Pod* in Figure 2-5 highlight how paths from the Master Node (see #1 and #3) are mapped as volumes for the container (see #1).



```
[root@bastion ~]# oc get pod master-etcd-master1.demo.internal -n kube-system -o yaml
  ...
    volumeMounts:  1
    - mountPath: /etc/etcd/
      name: master-config
      readOnly: true
    - mountPath: /var/lib/etcd/
      name: master-data
    workingDir: /var/lib/etcd
  dnsPolicy: ClusterFirst
  hostNetwork: true
  nodeName: master1.demo.internal  2
  ...
  volumes:
  - hostPath:  3
      path: /etc/etcd/
      type: ""
    name: master-config
  - hostPath:
      path: /var/lib/etcd
      type: ""
    name: master-data
  ...
```

Figure 2-5 Details of etcd Pod highlighting host path mounts as volumes

Figure 2-6 (see #4 and #5) provides the detail of the host paths mounted by the *API server Pod* from the *Master Nodes*.

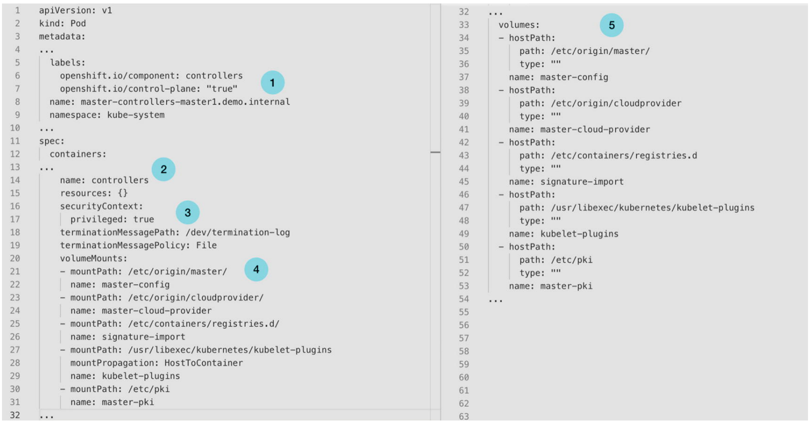


```
apiVersion: v1
kind: Pod
metadata:
...
  labels:
    openshift.io/component: api
    openshift.io/control-plane: "true"
  name: master-api-master1.demo.internal
  namespace: kube-system
...
spec:
  containers:
...
    name: api
...
    resources: {}
    securityContext:
      privileged: true
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /etc/origin/master/
      name: master-config
    - mountPath: /etc/origin/cloudprovider/
      name: master-cloud-provider
    - mountPath: /var/lib/origin/
      name: master-data
    - mountPath: /etc/pki
      name: master-pki
...
```

```
...
  volumes:
  - hostPath:
      path: /etc/origin/master/
      type: ""
    name: master-config
  - hostPath:
      path: /etc/origin/cloudprovider
      type: ""
    name: master-cloud-provider
  - hostPath:
      path: /var/lib/origin
      type: ""
    name: master-data
  - hostPath:
      path: /etc/pki
      type: ""
    name: master-pki
...
```

Figure 2-6 Details of API server Pod highlighting host path mounts as volumes

Similarly, Figure 2-7 highlights the host paths from the Master Node (see #4 and #5) mounted by the *Controllers Pod*.



```
apiVersion: v1
kind: Pod
metadata:
...
  labels:
    openshift.io/component: controllers
    openshift.io/control-plane: "true"
  name: master-controllers-master1.demo.internal
  namespace: kube-system
...
spec:
  containers:
...
    name: controllers
    resources: {}
    securityContext:
      privileged: true
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /etc/origin/master/
      name: master-config
    - mountPath: /etc/origin/cloudprovider/
      name: master-cloud-provider
    - mountPath: /etc/containers/registries.d/
      name: signature-import
    - mountPath: /usr/libexec/kubernetes/kubelet-plugins
      mountPropagation: HostToContainer
      name: kubelet-plugins
    - mountPath: /etc/pki
      name: master-pki
...
```

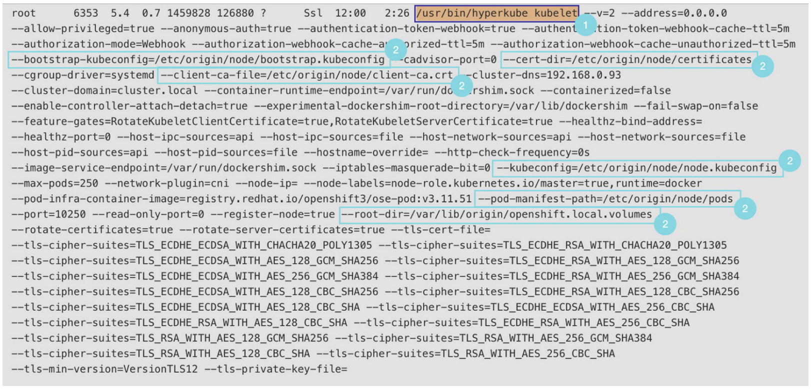
```
...
  volumes:
  - hostPath:
      path: /etc/origin/master/
      type: ""
    name: master-config
  - hostPath:
      path: /etc/origin/cloudprovider
      type: ""
    name: master-cloud-provider
  - hostPath:
      path: /etc/containers/registries.d
      type: ""
    name: signature-import
  - hostPath:
      path: /usr/libexec/kubernetes/kubelet-plugins
      type: ""
    name: kubelet-plugins
  - hostPath:
      path: /etc/pki
      type: ""
    name: master-pki
...
```

Figure 2-7 Details of Controllers Pod highlighting host path mounts as volumes

In all these cases, the configuration files, certificates, and other information reside on the Master Node but are consumed directly by these privileged *Pods* which are core components of the *Control Plane*.

One of the missing elements of the *Control Plane* not running as a *Pod* or as privileged *Container* is the *kubelet* service. The *kubelet* service runs as a traditional privileged process on the *Master Node* (see Figure 2-8).

NOTE The reader may notice the *hyperkube* binary used to invoke the *kubelet* service (see #1 of Figure 2-8). The *hyperkube*[6] is the all-in-one binary with all the Kubernetes server components: *kube-apiproxy, kubelet, kube-scheduler, kube-controller-manager, kube-proxy.*



```
root      6353  5.4  0.7 1459828 126880 ?      Ssl  12:00   2:26 /usr/bin/hyperkube kubelet --v=2 --address=0.0.0.0
--allow-privileged=true --anonymous-auth=true --authentication-token-webhook=true --authent 1 tion-token-webhook-cache-ttl=5m
--authorization-mode=Webhook --authorization-webhook-cache-a 2 orized-ttl=5m --authorization-webhook-cache-unauthorized-ttl=5m
--bootstrap-kubeconfig=/etc/origin/node/bootstrap.kubeconfig -cadvisor-port=0 --cert-dir=/etc/origin/node/certificates 2
--cgroup-driver=systemd --client-ca-file=/etc/origin/node/client-ca.crt -cluster-dns=192.168.0.93
--cluster-domain=cluster.local --container-runtime-endpoint=/var/run/do 2 rshim.sock --containerized=false
--enable-controller-attach-detach=true --experimental-dockershim-root-directory=/var/lib/dockershim --fail-swap-on=false
--feature-gates=RotateKubeletClientCertificate=true,RotateKubeletServerCertificate=true --healthz-bind-address=
--healthz-port=0 --host-ipc-sources=api --host-ipc-sources=file --host-network-sources=api --host-network-sources=file
--host-pid-sources=api --host-pid-sources=file --hostname-override= --http-check-frequency=0s
--image-service-endpoint=/var/run/dockershim.sock --iptables-masquerade-bit=0 --kubeconfig=/etc/origin/node/node.kubeconfig 2
--max-pods=250 --network-plugin=cni --node-ip= --node-labels=node-role.kubernetes.io/master=true,runtime=docker
--pod-infra-container-image=registry.redhat.io/openshift3/ose-pod:v3.11.51 --pod-manifest-path=/etc/origin/node/pods 2
--port=10250 --read-only-port=0 --register-node=true --root-dir=/var/lib/origin/openshift.local.volumes 2
--rotate-certificates=true --rotate-server-certificates=true --tls-cert-file=
--tls-cipher-suites=TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305 --tls-cipher-suites=TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305
--tls-cipher-suites=TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 --tls-cipher-suites=TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
--tls-cipher-suites=TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 --tls-cipher-suites=TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
--tls-cipher-suites=TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 --tls-cipher-suites=TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
--tls-cipher-suites=TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA --tls-cipher-suites=TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
--tls-cipher-suites=TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA --tls-cipher-suites=TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
--tls-cipher-suites=TLS_RSA_WITH_AES_128_GCM_SHA256 --tls-cipher-suites=TLS_RSA_WITH_AES_256_GCM_SHA384
--tls-cipher-suites=TLS_RSA_WITH_AES_128_CBC_SHA --tls-cipher-suites=TLS_RSA_WITH_AES_256_CBC_SHA
--tls-min-version=VersionTLS12 --tls-private-key-file=
```

Figure 2-8 Details of the kubelet process running in a node

In a multimaster deployment, the default is to use *native* high availability (HA) to determine how to load balance the API requests across the *Master Nodes*. This *native* HA method takes advantage of the built-in native HA master capabilities in OCP and can be used with any load balancing solution.

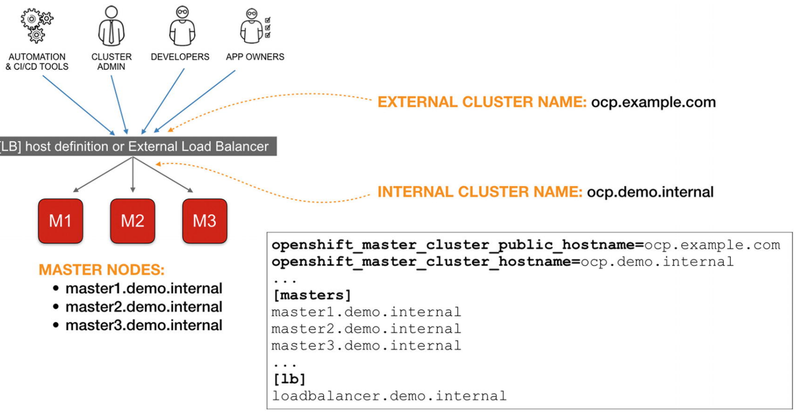
Each Master Node runs all the master server components. Accessing the API server at Master Nodes does not require session awareness or stickiness. Each Master Node answers to the cluster internal name, the cluster external name, and its own hostname.

The OpenShift advanced installation using *openshift-ansible* supports the definition of an *[lb]* section in the inventory file which automatically installs and configures an HAProxy to act as the load balancing solution for the *Master Nodes*.

NOTE The *[lb]* definition ONLY manages or load balances traffic toward the *Master Nodes*. It does NOT load balance traffic toward the *Infrastructure Nodes* or applications running on the OpenShift cluster.

To better illustrate this configuration, refer to Figure 2-9. As seen in Figure 2-9, there is the concept of an *External Cluster Name* and *Internal Cluster Name*, and each *Master Node* has their own assigned *FQDN*.

The *External Cluster Name* is defined in the advanced installation *inventory* file by the *openshift_master_cluster_public_hostname* variable. Similarly, the internal cluster name is specified by the *openshift_master_cluster_hostname* variable.



```
openshift_master_cluster_public_hostname=ocp.example.com
openshift_master_cluster_hostname=ocp.demo.internal
...
[masters]
master1.demo.internal
master2.demo.internal
master3.demo.internal
...
[lb]
loadbalancer.demo.internal
```

Figure 2-9 The native HA and load balancing for Master Nodes

Any external load balancer can be used to load balance the traffic among the *Master Nodes*. The requirements for using external load balancer are simple:

1. Define a virtual IP or VIP to represent the cluster.

2. Configure the VIP for SSL passthrough.

3. Configure the VIP to listen to the port specified by the openshift_master_api_port variable of the inventory file. If no port is specified, the API server will listen in port 8443 in every Master Node.

> **NOTE** In some load balancer might require a different *external VIP* and an *internal VIP*. Other load balancers will handle both external and internal cluster names with a single VIP.

4. Configure the DNS to resolve the *External Cluster Name* to the *external VIP* and the *Internal Cluster Name* to the *internal VIP.*

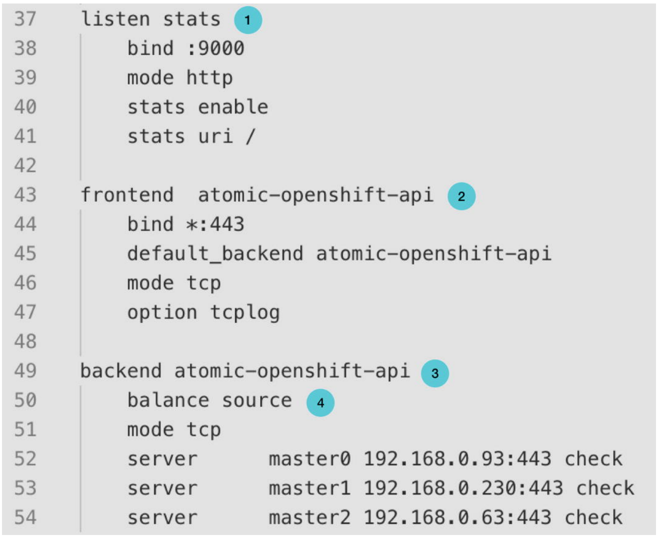
The HA styles for each of the master services can be summarized as in Table 2-1. Some services handle their internal HA, while others are completely active-active HA.

Table 2-1 The Native HA of Master Services

Role	HA Style	Notes
etcd	Active-Active	The *etcd* service is highly redundant and using the RAFT algorithm to maintain data replication and consistency. By default, in OpenShift, this is only accessible from within the cluster. There is no external access or exposure to the *etcd* service.
API Server	Active-Active	Any *Master Node* can handle requests to the API Server. The external load balancer can choose the preferred method to distributing the load. When using the *[lb]* host, the *HAProxy* distributes the traffic using the *source* balancing mode which is based on the hash of the source IP address making the request.

Role	HA Style	Notes
Controllers and Schedulers	Active-Passive	One *Controller* instance is elected as the cluster leader at a time. Each API Server handling a request interacts with their local *Controller* instance. The local *Controller* instance is aware and communicates with the leader *Controller* which is the only instance scheduling and controlling Pods in the cluster at any given time.

The specific configuration for the *[lb]* hosts is shown in Figure 2-10. As it can be seen, the HAProxy is deployed to listen on *openshift_master_api_port*, in this example port 443 (#2 of Figure 2-10). The load balancing is a simple TCP passthrough (#3 of Figure 2-10) toward the *Master Nodes*. The load balancing mode is a *source* (#4 of Figure 2-10) which balances based on the resulting hash of the source IP address making the request.



```
listen stats  1
    bind :9000
    mode http
    stats enable
    stats uri /

frontend  atomic-openshift-api  2
    bind *:443
    default_backend atomic-openshift-api
    mode tcp
    option tcplog

backend atomic-openshift-api  3
    balance source  4
    mode tcp
    server      master0 192.168.0.93:443 check
    server      master1 192.168.0.230:443 check
    server      master2 192.168.0.63:443 check
```

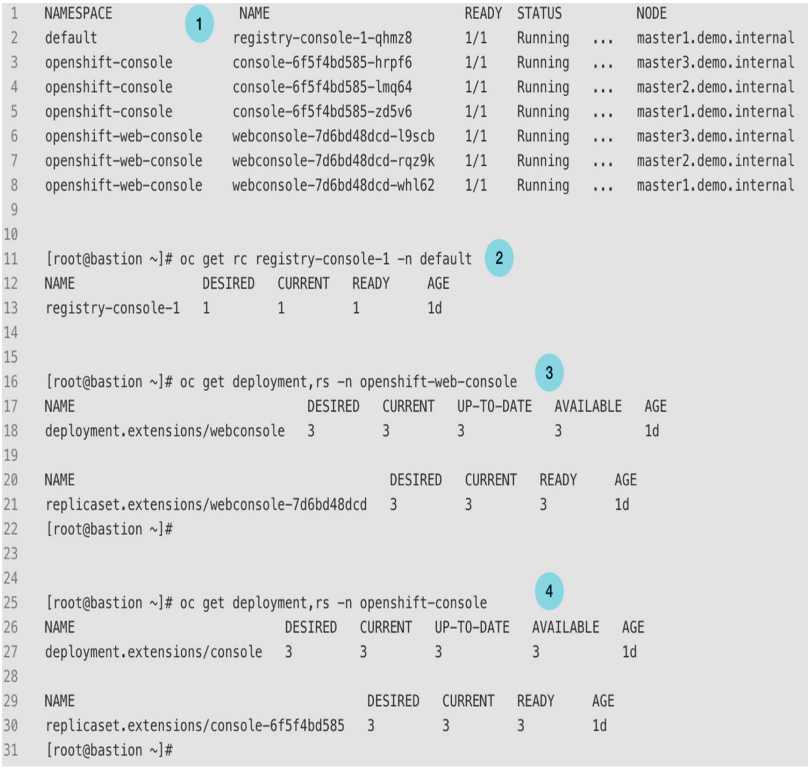
Figure 2-10 Relevant HAProxy configuration for the [lb] host

Beyond what can be achieved by the load balancers, the system takes care of restarting any of the *Containers* and *Pods* providing the master services just like it will do to remediate deviations from the desired configuration or state for any other *Pod* running an application in Kubernetes.

HA FOR OPENSHIFT CONSOLES

The OpenShift consoles are deployed as Kubernetes objects and use *Services*, *ReplicationController,* or *Deployment* objects to maintain HA. Consider the output shown in Figure 2-11. #1 of Figure 2-11 lists the *Pods* corresponding to each of the *Consoles*: *registry-console*, *openshift-web-console,* and *openshift-console.*

The HA for the *Container Registry Console* is achieved by the Service named *registry-console* and the ReplicationController named registry-console-1 (see #2 of Figure 2-11). The HA for the developer console (openshift-web-console) is achieved by a *Service* named *webconsole* and a *Deployment* object named *webconsole* with its corresponding *ReplicaSet* (see #3 of Figure 2-11). Finally, the HA for the OpenShift operations console (openshift-console) is achieved by a Service named *console* and a Deployment object named *console* with its corresponding ReplicaSet (see #4 of Figure 2-11).



```
NAMESPACE              NAME                           READY  STATUS         NODE
default                registry-console-1-qhmz8       1/1    Running   ...  master1.demo.internal
openshift-console      console-6f5f4bd585-hrpf6       1/1    Running   ...  master3.demo.internal
openshift-console      console-6f5f4bd585-lmq64       1/1    Running   ...  master2.demo.internal
openshift-console      console-6f5f4bd585-zd5v6       1/1    Running   ...  master1.demo.internal
openshift-web-console  webconsole-7d6bd48dcd-l9scb    1/1    Running   ...  master3.demo.internal
openshift-web-console  webconsole-7d6bd48dcd-rqz9k    1/1    Running   ...  master2.demo.internal
openshift-web-console  webconsole-7d6bd48dcd-whl62    1/1    Running   ...  master1.demo.internal


[root@bastion ~]# oc get rc registry-console-1 -n default
NAME                DESIRED   CURRENT   READY     AGE
registry-console-1  1         1         1         1d


[root@bastion ~]# oc get deployment,rs -n openshift-web-console
NAME                             DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deployment.extensions/webconsole  3         3         3            3           1d

NAME                                         DESIRED   CURRENT   READY     AGE
replicaset.extensions/webconsole-7d6bd48dcd   3         3         3         1d
[root@bastion ~]#


[root@bastion ~]# oc get deployment,rs -n openshift-console
NAME                           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deployment.extensions/console  3         3         3            3           1d

NAME                                      DESIRED   CURRENT   READY     AGE
replicaset.extensions/console-6f5f4bd585  3         3         3         1d
[root@bastion ~]#
```

Figure 2-11 OpenShift Console Pods, ReplicationControllers, and Deployments

With the use of the native Kubernetes constructs to protect these Consoles, there is no additional configuration required for its HA.

HA FOR LOGGING, METRICS, AND MONITORING

The OpenShift Monitoring, Logging, and Metrics services are comprised of multiple elements, all of which are deployed and managed as Kubernetes objects: *Service*, *DaemonSet*, *Deployment*, *ReplicationController,* and *DeploymentConfig*. As such, these mechanisms take care of maintaining the high availability for each one of these services. The *OpenShift Monitoring* components are deployed on the *openshift-monitoring Namespace* or *Project*.

NOTE A *DeploymentConfig* or *OpenShift Deployment Configuration*[7] is an OpenShift-specific object that predates *Kubernetes Deployment*. The

DeploymentConfig was built on *ReplicationController* to support the development and deployment lifecycle of an application. In addition to the capabilities of the *Deployment*, the *DeploymentConfig* provides the ability to specify deployment strategies (i.e., rolling strategy, recreate strategy, etc.) to change or upgrade an application; ability to set up triggers to automatically change, redeploy, or upgrade an application and the deployment strategy to use during the transition; and the ability to define hooks to be run before or after creating the *ReplicationController*.

Even when the system takes care of maintaining the availability of these services, it is good to understand how these services are deployed should there be a need for troubleshooting.

OpenShift Monitoring

The *OpenShift Monitoring* is a cluster monitoring solution comprised of *Prometheus*[8] with its plugin ecosystem and *Grafana* for the dashboards. OpenShift uses the *Cluster Monitoring Operator*[9] to configure, deploy, and maintain the *OpenShift Monitoring* stack.

The elements of OpenShift Monitoring are illustrated in Figure 2-12. The details of each component are described in the following list:

- **Prometheus**: Prometheus itself is an Open Source project for monitoring and alerting.
- **Prometheus Operator**: A *Kubernetes Operator* to create, configure, and manage *Prometheus* and *Alertmanager* instances. In *OpenShift Monitoring*, this component is deployed as a *Deployment* which creates a *ReplicaSet (RC).* The RC maintains one *prometheus-operator Pod* running in any of the *Infrastructure Nodes.*
- **Cluster Monitoring Operator**: Watches the deployed monitoring components and resources of the OpenShift Monitoring and ensures they are up to date. This element is deployed as a Deployment which creates the ReplicaSet *(RC).* The RC maintains one *cluster-monitoring-operator Pod* running in any of the *Infrastructure Nodes.*

- ***prometheus-k8s***: The actual Prometheus instances responsible for monitoring and alerting on cluster and OpenShift components. This component is deployed as a *StatefulSet* and maintains a copy in every *Infrastructure Node*.

- **Alertmanager**: A global cluster component for handling alerts generated by all the *Prometheus* instances in the particular cluster. This element is deployed as a *StatefulSet* and maintains two prometheus-k8s Pods across any of the *Infrastructure Nodes.*

- **node-exporter**: Prometheus exporter or agent deployed on every Node to collect metrics from its hardware and Operating System. This element is deployed as a *DaemonSet*. There is one *node-exporter Pod* in every *Node* of the cluster.

- **kube-state-metrics**[10]: Prometheus exporter or plugin to convert metrics from Kubernetes objects into metrics consumable by *Prometheus*. This is deployed as a *Deployment* which creates a *ReplicaSet* and runs a *kube-state-metric Pod* in any of the *Infrastructure Nodes*.

- **Grafana**[11]: An extensible Open Source metrics analytics and visualization suite. This element is deployed as a *Deployment* which creates a *ReplicaSet* and runs a *Grafana Pod* in any of the *Infrastructure Nodes.*

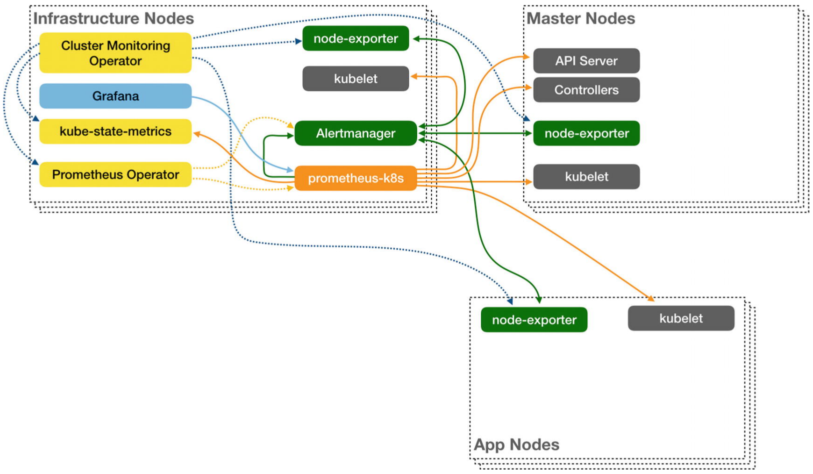


Figure 2-12 The OpenShift Monitoring architecture

Metrics

What is considered *OpenShift Metrics* are the original OpenShift components used to collect metrics information from *Containers*, *Pods,* and *Nodes* across the entire OpenShift cluster. These collected metrics are then available over the OpenShift Console or can be exported to an external system. These metrics can also be used for the *Horizontal Pod Autoscaler (HPA)*[12] to scale the number of *Pods* in a *ReplicationController* or *ReplicaSet* based.

NOTE OpenShift 3.11 is the last version supporting the traditional *OpenShift Metrics* service.[13] These are being deprecated in OpenShift 4.0. Most of the functionalities are replaced by the *OpenShift Monitoring* solution based on the Prometheus project, and the remaining functionality is superseded by the *Kubernetes Metrics Server.*

All the components of the traditional OpenShift Metrics are deployed as Kubernetes *ReplicationControllers* on the *openshift-infra Namespace* or *Project.* This service consists of the following components:

1. **Heapster**[14] : A service for the monitoring and analysis of compute, memory, and network resource utilization and performance for Kubernetes. Collects the information from the Kubelet APIs.

NOTE Kubernetes *kubelet* embeds *cAdvisor*[15] which autodiscovers all containers in the machine and collects CPU, memory, filesystem, and network usage statistics. *cAdvisor* also provides the overall machine usage by analyzing the “root” Container on the machine.

2. **Hawkular Metrics** : This is the metric storage engine for Hawkular. It uses the Cassandra database as the metric datastore.

3. **Cassandra**: The Cassandra database is used to store the metrics data.

The interaction between all these components is illustrated in Figure 2-13.

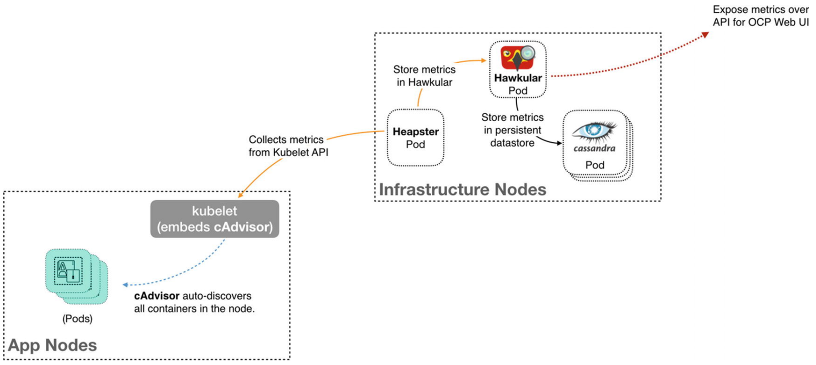

Expose metrics over API for OCP Web UI
Store metrics in Hawkular
Hawkular Pod
Heapster Pod
Store metrics in persistent datastore
cassandra Pod
Collects metrics from Kubelet API
Infrastructure Nodes
kubelet (embeds cAdvisor)
cAdvisor auto-discovers all containers in the node.
(Pods)
App Nodes

Figure 2-13 The OpenShift Metrics architecture (deprecated in OCP 4.0)

Metrics Server

Metrics Server[16] is a cluster-wide aggregator of resource usage data like Container CPU and memory utilization. The Metrics Server collects metrics from the Kubelet API of each node. The resource usage metrics are made available in Kubernetes through the Metrics API. It supersedes the Heapster service in OpenShift 4.0 and beyond.

The *Metrics Server* is considered the prerequisite for some advanced Kubernetes features or capabilities like the *Horizontal Pod Autoscaler (HPA)*, the Kubernetes scheduler, and other functionalities that require access to metrics[17] from nodes and Pods. In OpenShift, this service runs as a *Deployment* which creates a *ReplicaSet* to maintain a *metrics-server Pod* in one of the *Master Nodes*.

Logging

The *OpenShift Logging* service aggregates logs for the OpenShift platform services, *Nodes*, *Containers*, and applications. The OpenShift Logging service shown in Figure 2-14 is comprised of the following components:

- **Elasticsearch (ES)**: A NoSQL database with multitenant full-text search and analytics engine. This component is deployed in the *openshift-logging Namespace* or *Project* as *DeploymentConfig* which creates a *ReplicationController* to run the requested number of *Pods*. The cluster administrator should rightsize[18] the Elasticsearch deployment to the requirements of the specific environment.

- **FluentD**: Data collection software that gathers logs from the *Nodes* and feeds them to the *Elasticsearch* database. This element is deployed in the *openshift-logging Namespace* as a *DaemonSet*. There is a *logging-fluentd Pod* in *every* Node of the cluster.

- **Kibana**: An analytics and visualization Web UI for Elasticsearch. It enables the creation of visualizations and dashboards for monitoring *Container* and *Pods* logs by *Deployment, Namespace, Pod, and Container*. Kibana is deployed in the *openshift-logging Namespace* as a *DeploymentConfig* which creates a *ReplicationController* to run and maintain the *logging-kibana Pod* running on an *Infrastructure Node*.

- **Curator**[19]: Allows administrators to configure scheduled maintenance operations for the Elasticsearch database. These are performed automatically on per-project basis. This component is deployed into the *openshift-logging Namespace* as a *Kubernetes CronJob* object and runs the *logging-curator Pod* on one of the Infrastructure Nodes.

- **Eventrouter**: Watches Kubernetes events, formats them to JSON, and outputs them to STDOUT to be ingested by FluentD. The *logging-eventrouter Pod* is deployed to *default Namespace* or *Project* as a *DeploymentConfig* where it creates a *ReplicationController* and runs the *Pod* on an *Infrastructure Node.*

The first three components together (Elasticsearch, FluentD, and Kibana) are known as the ***EFK** stack*.

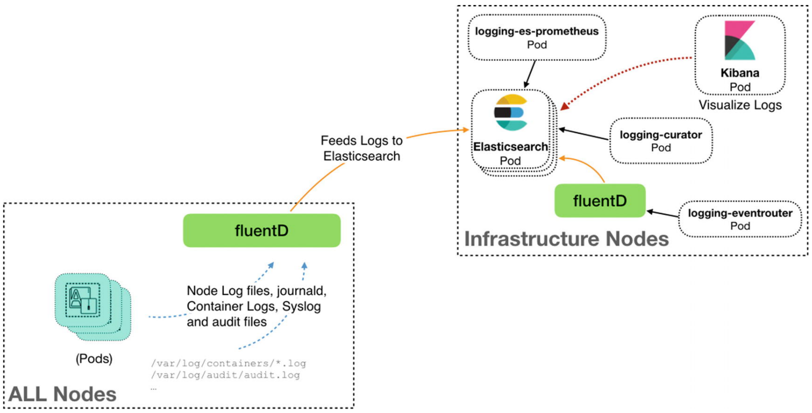


Figure 2-14 The OpenShift Logging Service

HA for Data Plane

As mentioned before, there is no official definition of the *OpenShift Data Plane* even though the term is normally used to refer to the traffic forwarding plane of the SDN layer.

As with any other networking architecture, from the SDN layer perspective, we can talk about north-south traffic and east-west traffic. From the OpenShift perspective, the north-south traffic refers to the external traffic arriving into the cluster or the inbound traffic toward the applications hosted on the platform. The east-west traffic refers to the traffic within the cluster.

In a future chapter, we will go into details on how the different SDN options and capabilities move traffic within the cluster (east-west traffic) and the specific features they may provide. For now, this section focuses on the inbound traffic (north-south traffic) arriving to the applications deployed on the cluster. By default, the traffic toward the applications running on the cluster goes through the OpenShift Routers.

HA FOR OPENSHIFT ROUTER

The *OpenShift Router* is an OpenShift component used to expose *Services* running on the cluster to external clients. It does this by generating a unique *FQDN* and handling requests to it by steering the traffic to the appropriate *Service*. The *OpenShift Routers* are deployed in the *default Namespace* or *Project* as a *DeploymentConfig* which creates a *ReplicationController*. The ReplicationController maintains the number of *router Pod* specified by *openshift_hosted_router_replicas* in the inventory file. These *Routers* are deployed to the *Infrastructure Nodes.* This behavior can be modified by specifying a different *Node* label selector for the Pods using the *openshift_router_selector* variable in the inventory file. If not specified, the default number of replicas is set to one.

In case of failure of a *Router*, the *DeploymentConfig* takes care of correcting the environment by creating a new one.

In a later chapter, we will cover the *OpenShift Router Sharding* capabilities, and we are going to see some of the advanced techniques that can be used to distribute *Routes* among different *Routers* or even dedicate *Routers* to specific *Namespaces* or *Projects*.

HA for Container Registry

The *OpenShift Container Registry (OCR)* is the default internal *Container* image registry used by the cluster to store *Container* images built with one of the supported build strategies, or among other things, to maintain a copy of *Container* images running in the environment.

The OpenShift Container Registry is deployed into the *default Namespace* or *Project* using a *DeploymentConfig* which creates a *ReplicationController* used to run and maintain the desired number of *docker-registry Pod* running. Alternatively, the cluster admin can choose to deploy the OCR as a DaemonSet.[20]

When not specified, by default, the installer will deploy one *docker-registry Pod* running on the *Infrastructure Nodes.* The number of *docker-registry Pods* to deploy and the *Nodes* selectors to use to deploy the Pod can be specified by using the *openshift_hosted_registry_replicas* and *openshift_registry_selector* variables, respectively, in the advanced installer *inventory* file.

If no persistent storage options are specified for the registry, the default is to use ephemeral storage and all data will be lost when the *Pod* is restarted.

When using multiple replicas, the persistent storage must support the *ReadWriteMany*[21] storage access mode. The supported storage[22] backends for the Registry range from GlusterFS to S3 compatible services.

CAUTION In production environments, the OpenShift Container Registry should NOT use NFS as the storage backend.

Summary

The OpenShift architecture is designed for high availability of every one of its components. Since these elements are built on top of Kubernetes using the Kubernetes constructs, they benefit from the resiliency provided by these. As it can be seen from this chapter, when the OpenShift cluster is deployed with multiple Master, Infrastructure, and Application Nodes, the availability of all the other internal elements of the platform is achieved with Kubernetes itself.

With the abstraction layers created by Kubernetes and the OpenShift platform, Chapter 3 describes the traffic flow with different overlay SDNs when components communicate inside the platform vs. when applications communicate outside the platform.

Footnotes

1 See official definition here: https://kubernetes.io/docs/concepts/#kubernetes-control-plane

2 API Server, Controllers, Scheduler, and etcd database

3 OpenShift Projects are Kubernetes Namespaces used to organize and manage content in isolation for a community of users. https://docs.openshift.com/container-platform/3.11/dev_guide/projects.html

4 Kubernetes Namespaces are used to divide cluster resources among multiple projects and users. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

5 For details about the differences between *oc* and *kubectl*, visit https://docs.openshift.com/container-

platform/3.11/cli_reference/differences_oc_kubectl.html

6 Refer to the GitHub project for additional details:

https://github.com/kubernetes/kubernetes/tree/master/cluster/images/

7 For more information about *Deployments* and deployment strategies, visit https://docs.openshift.com/container-platform/3.11/dev_guide/deployments/how_deployments_work.html

8 Prometheus is an Open Source project for monitoring and alerting. Additional information can be found at https://prometheus.io/docs/introduction/overview/

9 The Cluster Monitoring Operator is an Open Source Kubernetes Operator to manage a Prometheus-based cluster monitoring stack. More information can be found here: https://github.com/openshift/cluster-monitoring-operator

10 Additional details about the metrics collected by this agent can be found at https://github.com/prometheus/node_exporter

11 Additional information can be found at https://grafana.com/grafana

12 Kubernetes Horizontal Pod Autoscaler (HPA) automatically scales the number of Pods. For more information, visit https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

13 See Release Notes for OpenShift 3.11 at https://docs.openshift.com/container-platform/3.11/release_notes/ocp_3_11_release_notes.html#ocp-311-major-changes-in-40

14 Heapster was deprecated in Kubernetes 1.11 in favor of Metrics Server and has been retired in Kubernetes 1.13. Additional details are available here: https://github.com/kubernetes-retired/heapster/blob/master/docs/deprecation.md

15 Additional information on how *cAdvisors* are embedded in *kubelet* is available at https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/#cadvisor

16 For more details about the Kubernetes Metrics Server, visit https://kubernetes.io/docs/tasks/debug-application-cluster/core-metrics-pipeline/#metrics-server

17 Additional information on use cases and scalability of the Metrics Server is available at https://github.com/kubernetes/community/blob/master/contributors/des
proposals/instrumentation/metrics-server.md

18 For guidelines on rightsizing the Elasticsearch database, visit https://docs.openshift.com/container-platform/3.11/install_config/aggregate_logging_sizing.html

19 For details on how to configure and use Curator, visit https://docs.openshift.com/container-platform/3.11/install_config/aggregate_logging.html#configuring-curator

20 Deploying the Registry as a DaemonSet: https://docs.openshift.com/container-platform/3.11/install_config/registry/deploy_registry_existing_cluste
daemonset

21 Persistent volume access modes supported by OpenShift are described here: https://docs.openshift.com/container-platform/3.11/architecture/additional_concepts/storage.html#pv-access-modes

22 The full list of supported storage for registry is available at https://docs.openshift.com/container-platform/3.11/install_config/registry/deploy_registry_existing_cluste
for-the-registry