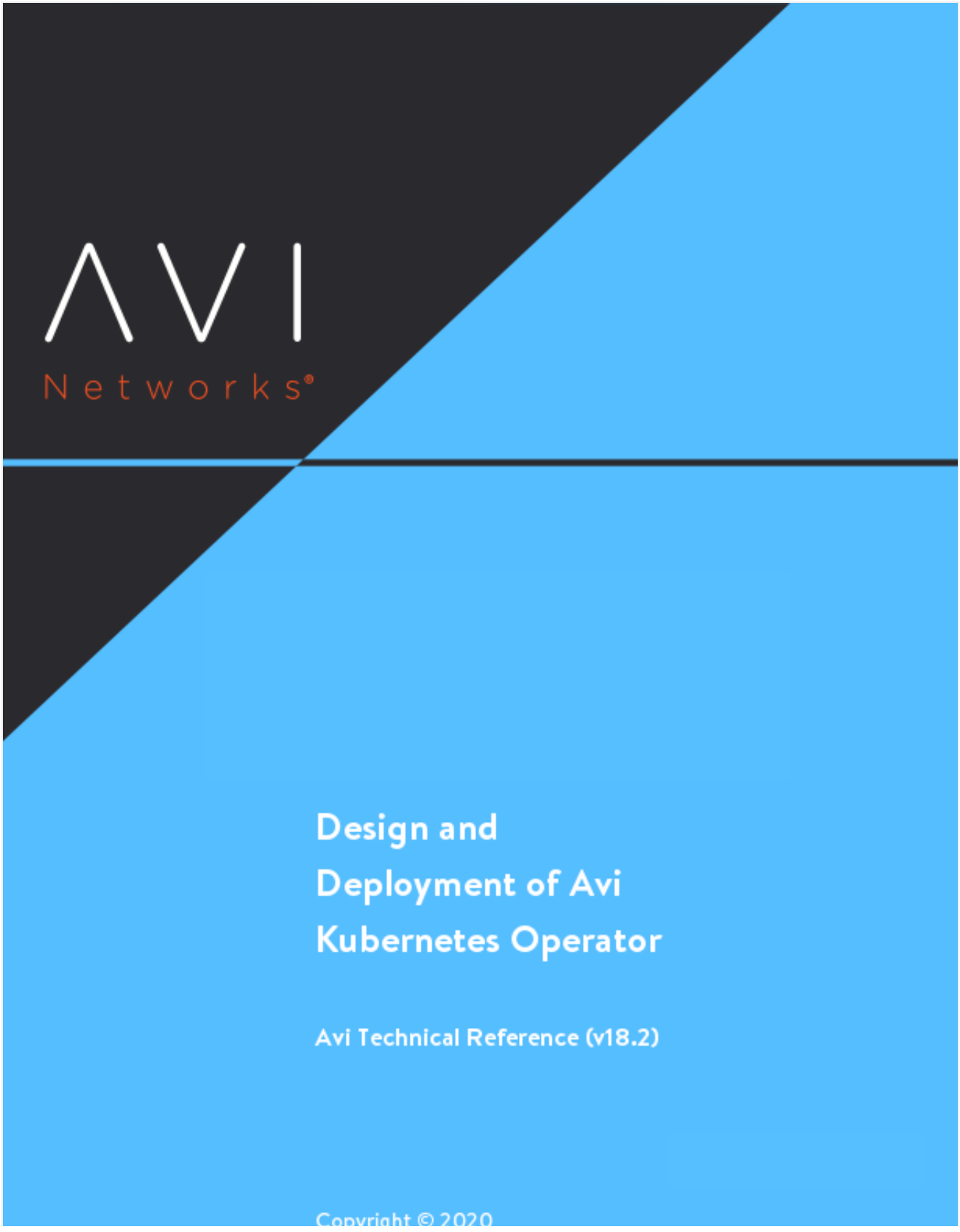

AVI
Networks®
Design and
Deployment of Avi
Kubernetes Operator
Avi Technical Reference (v18.2)
Copyright © 2020

Design and Deployment of Avi Kubernetes Operator

view online

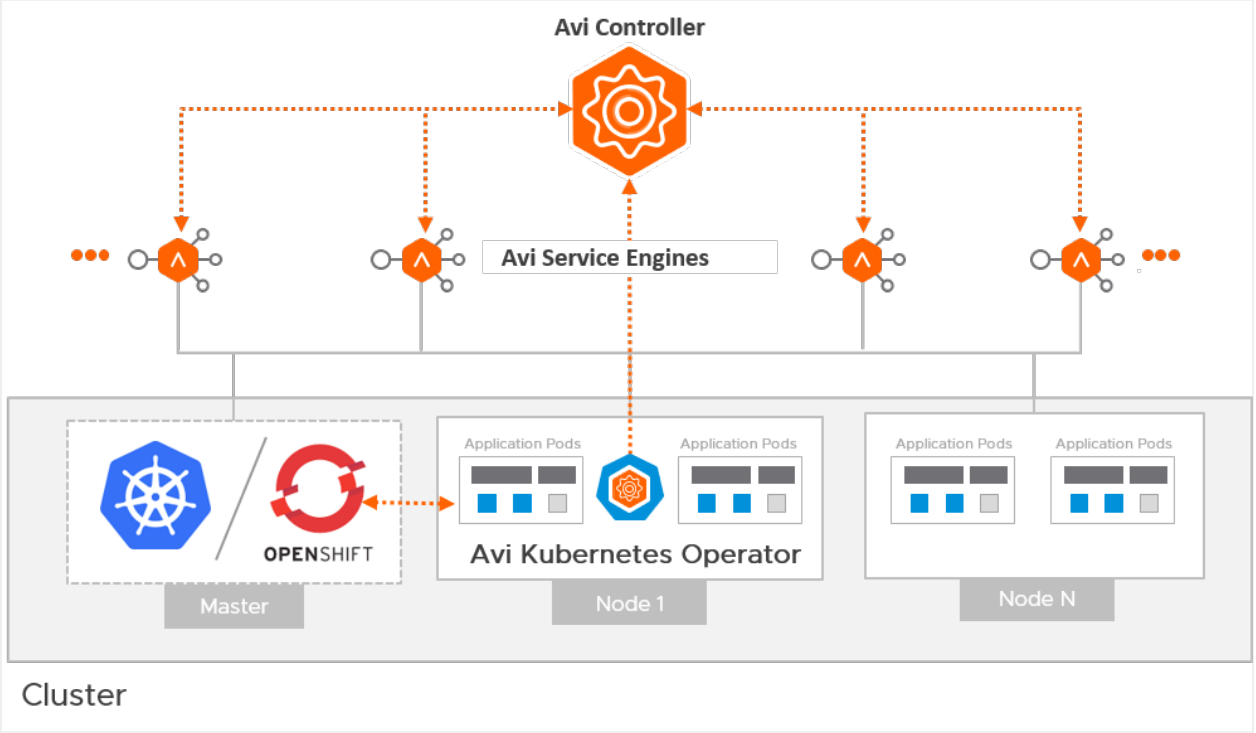
Overview

The Avi Kubernetes Operator (AKO) is a Kubernetes operator which works as an Ingress controller and performs Avi-specific functions in a Kubernetes environment with the Avi Controller. It translates Kubernetes objects to Avi Controller APIs.

This guide helps you understand the architectural overview and design considerations for AKO deployment.

Architecture

The Avi Deployment in Kubernetes for AKO comprises of the following main components: * The Avi Controller * The Service Engines (SE) * The Avi Kubernetes Operator (AKO)



The Avi Controller

The Avi Controller which is the central component of the Avi architecture is responsible for the following: * Control plane functionality like the: * Infrastructure orchestration * Centralized management * Analytics dashboard * Integration with the underlying ecosystem for managing the lifecycle of the data plane (Service Engines).

The Avi Controller does not handle any data plane traffic.

In Kubernetes environments, the Avi Controller is deployed outside the Kubernetes cluster, typically in the native type of the underlying infrastructure. However, it can be deployed anywhere as long as connectivity and latency requirements are satisfied.

The Avi Service Engines

The SEs implement data plane services of load balancing. For example, Web Application Firewall, DNS/GSLB, etc.

In Kubernetes environments, the SEs are deployed external to the cluster and typically in the native type of the underlying infrastructure.

The Avi Kubernetes Operator (AKO)

AKO is an Avi pod running in Kubernetes that provides an Ingress controller and Avi-configuration functionality. AKO remains in sync with the required Kubernetes objects and calls the Avi Controller APIs to deploy the Ingresses and Services via the Avi Service Engines.
AKO is deployed as a pod via Helm.

Avi Cloud Considerations

Avi Cloud Type

The Avi Controller uses the Avi Cloud configuration to manage the SEs. This Avi Cloud is usually of the underlying infrastructure type, for ex. VMware vCenter Cloud, Azure Cloud, Linux Server Cloud etc.

Note: This deployment in Kubernetes does not use the Kubernetes cloud type. The integration with Kubernetes and application-automation functions are handled by AKO and not by the Avi Controller.

Multiple Kubernetes Clusters

A single Avi Cloud can be used for integration with multiple Kubernetes clusters, with each cluster running its own instance of AKO. Clusters are separated on SEs in the DataPlane by using VRF Contexts. Each Kubernetes cluster must be deployed in a separate VRF to prevent overlap of IP addresses etc.

Refer to the section VRF Configuration for Multi Cluster for more information.

IPAM and DNS

The IPAM and DNS functionality is handled by the Avi Controller via the Avi cloud configuration.

Refer to the Service Discovery Using IPAM and DNS article for more information on supported IPAM and DNS types per environment.

Service Engine Groups

A single Service Engine group (the default SE group) is supported per cloud.

Note: AKO v 0.9.1 currently does not support multiple Service Engine groups.

Avi Controller Version

AKO v 0.9.1 is supported with the Avi Controller versions 18.2.x - 18.2.6, 18.2.7, 18.2.8.

Network Considerations

Avi SE Placement / Pod Network Reachability

With AKO, the service engines are deployed outside the cluster. To be able to load balance requests directly to the pods, the pod CIDR must be routable from the SE. Depending on the routability of the Pod CNI used in the cluster, AKO can route using the following options:

Pod is Not Externally Routable

For CNIs like Canal, Calico, Antrea, Flannel etc., the pod subnet is not externally routable. In these cases the CNI assigns a pod CIDR to each node in the Kuberntes cluster. The pods on a node get IP assigned from the CIDR allocated for that node and is routable from within the node. In this scenario, the pod reachability depends on where the SE is placed.

If SE is placed on the same network as the Kubernetes nodes, you can turn on static route programming in AKO. With this, AKO syncs the pod CIDR for each Kubernetes node and programs static route on the Avi Controller for each Pod CIDR with the Kubernetes node IP as the next hop. Refer to the section VRF Configuration for Multi Cluster for more information.

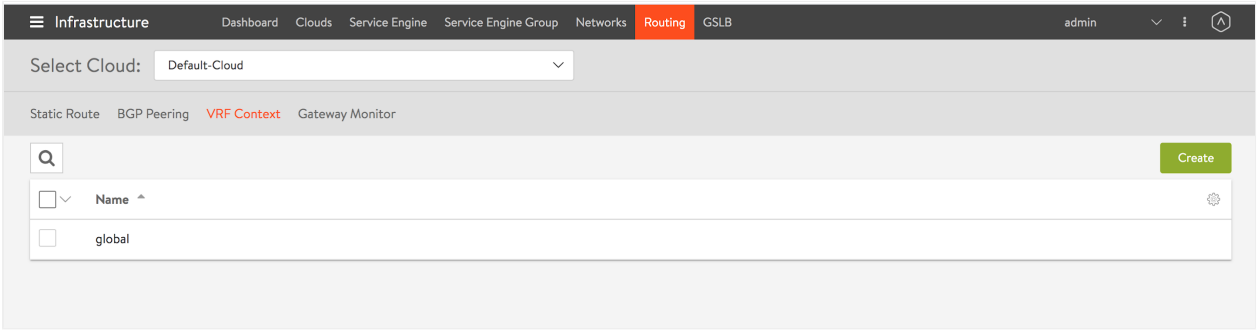
Pod Subnet is Routable

For CNIs like NSX-T CNI, AWS CNI (in EKS), Azure CNI (in AKS) etc., the pod subnet is externally routable. In this case no additional configuration is required to allow SEs to reach the Pod IPs. Set Static Route Programming to *Off* in the AKO configuration. SEs can be placed on any network and will be able to route the pods.

To know more about the CNIs supported in AKO v 0.9.1 click here.

VRF Configuration for Multi Cluster Use Cases If there are multiple Kubernetes clusters with non-routable CNIs, all clusters can have the same pod subnet. In this case, to allow the SE to route the traffic to a pod to the correct cluster, the admin must configure a VRF per cluster on the Avi cloud on the Avi Controller, and add the cluster?s node network to the corresponding VRF(assuming each Kubernetes cluster has a separate node network).

To configure VRF context, 1. From the Avi UI, navigate to Infrastructure > Routing. 2. Select the required cloud by clicking on the Select Cloud drop down list. 3. Click on the VRF Context tab. 4. Click on Create.



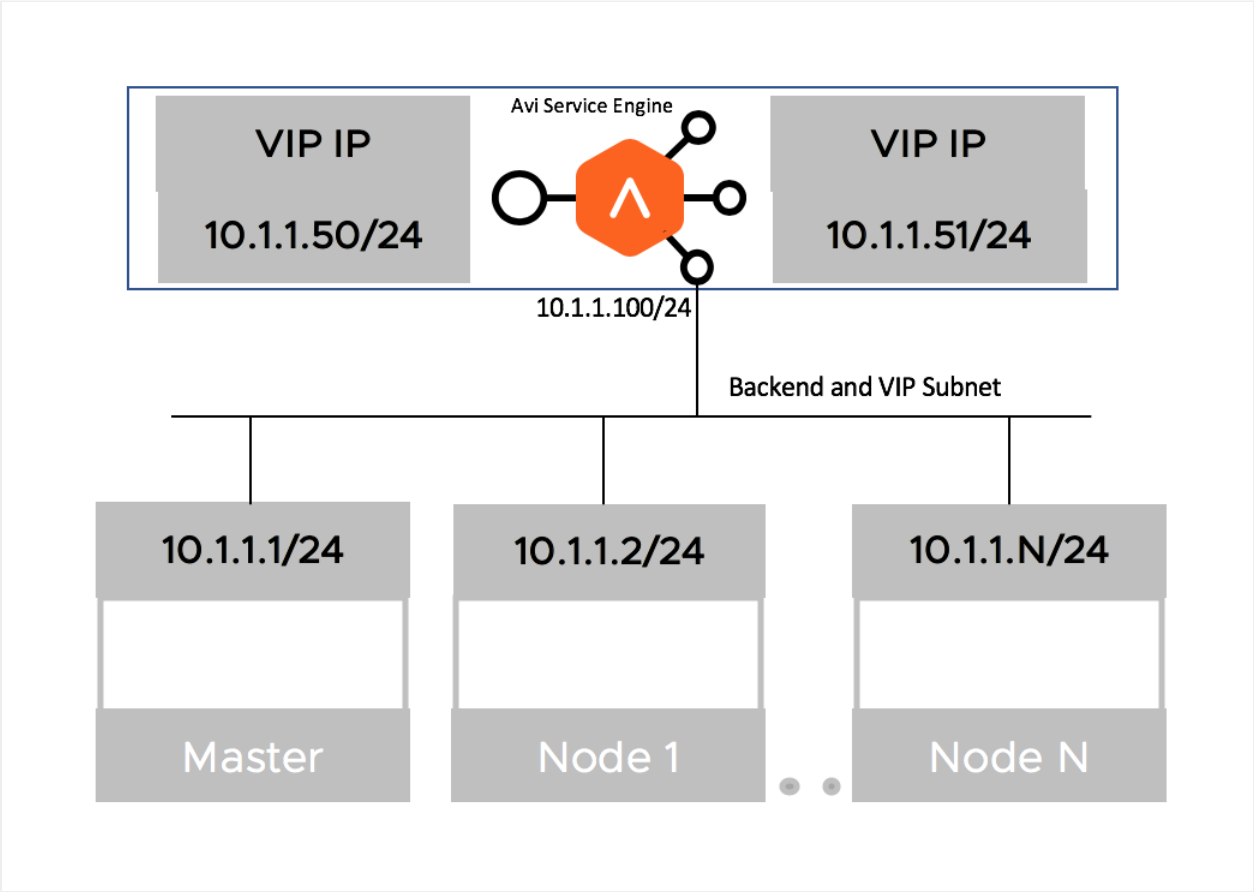
5. Enter the Name of the VRF context. 6. Click on Save.

The VRF name must be configured in AKO during installation. AKO creates the virtual service in the correct VRF context, for the Ingress and LB service objects on the cluster.

Deployment Modes

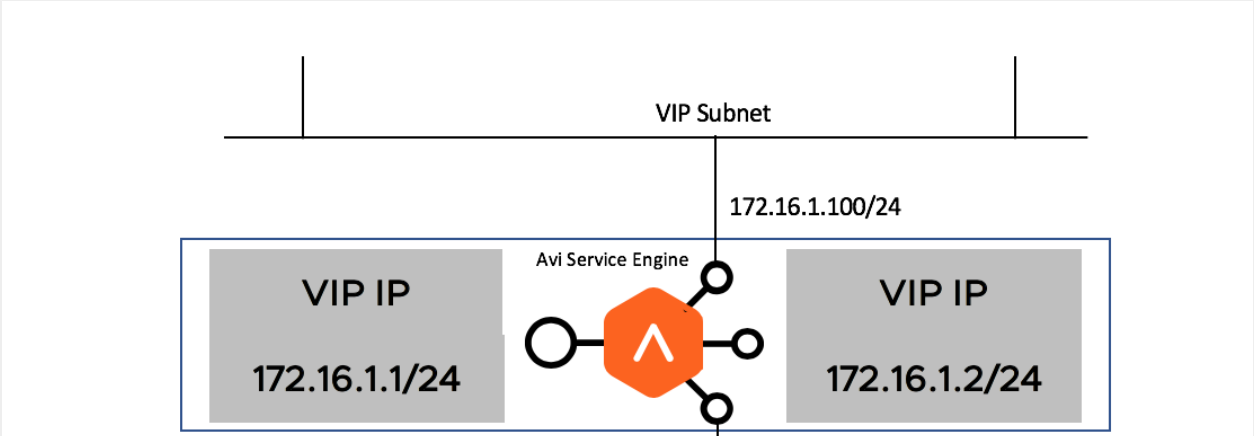
Single Arm Deployment

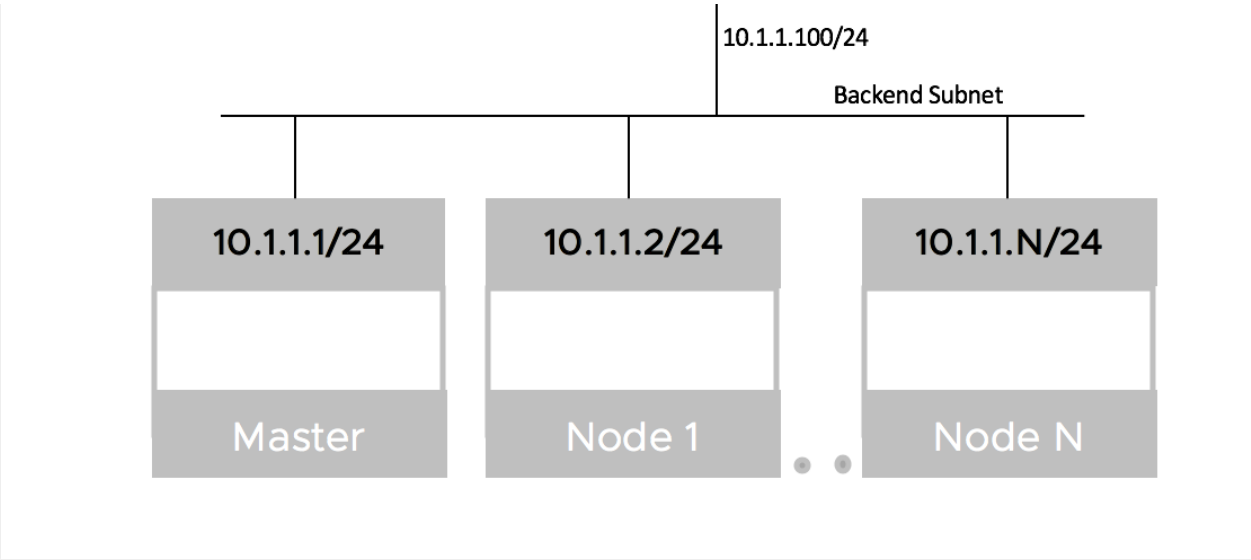
The deployment in which the virtual IP (VIP) address and the Kubernetes cluster are in the same network subnet is called a *Single Arm Deployment.*



Two-Arm Deployment

When the virtual IP (VIP) address and the Kubernetes cluster are in different network subnets, then the deployment is a *Two-Arm deployment*





AKO v 0.9.1 supports both Single-Arm and Two-Arm deployments with *vCenter Cloud* in *write-access* mode.

Handling of Kubernetes and Avi Objects

This section outlines the object translation logic between AKO and the Avi Controller.

Service of Type Load Balancer

AKO creates a Layer 4 virtual service object in Avi corresponding to a service of type `loadbalancer` in Kubernetes.
An example of such a service object in Kubernetes is as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: avisvc-lb
  namespace: red
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 8080
    name: eighty
  selector:
    app: avi-server
```

AKO creates a dedicated virtual service for this object in Kubernetes that refers to reserving a virtual IP for it. The layer 4 virtual service uses a pool section logic based on the ports configured on the service of type `loadbalancer`. In this case, the incoming port is port 80 and hence the virtual service listens on this port for client requests.
AKO selects the pods associated with this service as pool servers associated with the virtual service.

Insecure Ingress

Consider the following example of an insecure hostname specification from a Kubernetes Ingress object:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
    - host: myinsecurehost.avi.internal
      http:
        paths:
        - path: /foo
          backend:
            serviceName: service1
            servicePort: 80
```

For insecure host/path combinations, AKO uses a Sharded virtual service logic. Here, based on either the namespace of this Ingress or the hostname value (myhost.avi.internal), a pool object is created on a Shared virtual service. A shared virtual service typically denotes a virtual service in Avi that is shared across multiple Ingresses.

A priority label is associated to the pool group against its member pool (that is created as a part of this Ingress), with the priority label `myhost.avi.internal/foo`.

An associated DataScript object with this shared virtual service is used to interpret the host FQDN/path combination of the incoming request. The corresponding pool is chosen based on the priority label as mentioned above.

The paths specified are interpreted as *STARTSWITH* checks. This means for this particular host/path if pool X is created then, the matchrule can be interpreted as - "If the host header equals `myhost.avi.internal` and path STARTSWITH `foo` then route the request to pool X".

Secure Ingress

Consider the following example of an secure Ingress object:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: my-ingress
spec:
  tls:
  - hosts:
    - myhost.avi.internal
    secretName: testsecret-tls
  rules:
    - host: myhost.avi.internal
      http:
        paths:
        - path: /foo
          backend:
            serviceName: service1
            servicePort: 80
```

SNI Virtual Service per Secure Hostname

AKO creates an SNI child virtual service to a parent shared virtual service for the secure hostname. The SNI virtual service is used to bind the hostname to an `sslkeycert` object. The `sslkeycert` object is used to terminate the secure traffic on Avi's service engine. In the above example the `secretName` field denotes the secret asssociated with the hostname `myhost.avi.internal`. AKO parses the attached secret object and appropriately creates the `sslkeycert` object in Avi. The SNI virtual service does not get created if the secret object does not exist in Kubernetes corresponding to the reference specified in the Ingress object.

Traffic Routing Post SSL Termination

On the SNI virtual service, AKO creates `httppolicyset` rules to route the terminated (insecure) traffic to the appropriate pool object using the host/path specified in the rules section of this Ingress object.

Redirect Secure Hosts from HTTP to HTTPS

Additionally, for these hostnames, AKO creates a redirect policy on the shared virtual service (parent to the SNI child) for this specific secure hostname. This allows the client to automatically redirect the HTTP requests to HTTPS if they are accessed on the insecure port (80).

AKO Created Object Naming Conventions

In the current AKO model, all Kubernetes cluster objects are created on the admin tenant in Avi. This is true even for multiple Kubernetes clusters managed through a single Avi cloud (like the vCenter cloud).

Each virtual service/pool/pool group has to be unique to ensure there are no conflicts between similar object types.

AKO uses a combination of elements from each Kubernetes object to create a corresponding object in Avi that is unique for the cluster.

L4 Virtual Service

Use the following formula to derive a virtual service name:

```
vsName = clusterName + "--" + namespace + "--" + svcName
```

Here,
* `vrfName` is the value specified in values.yaml during install. * `svcName` refers to the service object's name in Kubernetes. * `namespace` refers to the namespace on which the service object is created.

L4 Pool

Use the following formula to derive L4 pool names:

```
poolname = vsName + "-" + listener_port
```

Here,
* `listener_port` refers to the service port on which the virtual service listens on. * The number of pools is directly associated with the number of listener ports configured in the Kubernetes service object.

L4 Pool Group

Use the following formula to derive the L4 pool group names for L4 virtual services:


```
poolgroupname = vsName + "-" + listener_port
```

Here,
* `vsName` is the virtual service's name. * `listener_port` refers to the service port on which the virtual service listens on.

Shared Virtual Service

The shared virtual service names are derived based on a combination of fields to keep it unique per Kubernetes cluster. This is the only object in Avi that does not derive it's name from any of the kubernetes objects.

The formula to derive the shared virtual service name is as follows:

```
ShardVSName = clusterName + "--Shared-L7-" + <shardNum>
```

Here,
* `clusterName` is the value specified in values.yaml during install. * `shardNum` is the number of the shared VS generated based on either hostname or namespace based shards.

Shared Virtual Service Pool

Use the following formula to derive the Shared virtual service pool group name:

```
poolgroupname = clusterName + "--" + priorityLabel + "-" + namespace + "-" + ingName
```

Here,
* `clusterName` is the value specified in values.yaml during install. * `priorityLabel` is the host/path combination specified in each rule of the Kubernetes Ingress object. * `ingName` refers to the name of the ingress object. * `namespace` refers to the namespace on which the ingress object is found in Kubernetes.

Shared Virtual Service Pool Group

Use the following formula to derive the shared virtual service pool group name:

```
poolgroupname = vsName
```

Here, * `vsName` is the virtual service's name.

Name of the shared virtual service is the same as the shared virtual service name.

SNI Child Virtual Service

The SNI child virtual service's naming varies between different sharding options.

Hostname Shard

```
vsName = clusterName + "--" + sniHostName
```

Namespace shard

```
vsName = clusterName + "--" + ingName + "-" + namespace + "-" + secret
```

The difference in naming is done because with namespace based sharding only one SNI child is created per ingress/per secret object but in hostname based sharding each SNI virtual service is unique to the hostname specified in the Ingress object.

SNI Pool

Use the following formula to derive the SNI virtual service's pool names:

```
poolname = clusterName + "--" + namespace + "-" + host + "_" + path + "-" + ingName
```

Here, the host and path variables denote the secure hosts' hostname and path specified in the ingress object.

SNI Pool Group

Use the following formula to derive the SNI virtual service's pool group names:

```
poolgroupname = clusterName + "--" + namespace + "-" + host + "_" + path + "-" + ingName
```

Some of these naming conventions can be used to debug/derive corresponding Avi object names that can be used as a tool for first level troubleshooting.

Annotations

AKO v 0.9.1 does not support annotations.

Multi-Tenancy

AKO v 0.9.1 does not currently support Multi-Tenancy.

AKO Support

```
<th>Kubernetes Version</th>
<th>CNI</th>
<th>Avi Cloud</th>
<th>Object Type</th>
<th>Avi Controller</th>
```

```
<td>v1.14</td>
<td>Flannel/Canal/Calico</td>
<td>vCenter</td>
<td>extensionsv1/ingress</td>
<td>18.2.6 onwards</td>
```

```
<td>v1.14 to v1.17</td>
<td>Flannel/Canal/Calico</td>
<td>vCenter</td>
<td>Service of type loadlbalancer</td>
<td>18.2.6 onwards</td>
```

```
<td>v1.15 to v1.17</td>
<td>Flannel/Canal/Calico</td>
<td>vCenter</td>
```

```
<td>v1/ingress</td>
<td>18.2.6 onwards</td>
```

Features Supported

The following features are supported in AKO v 0.9.1:

```
<th>Feature Name</th>
<th>Component</th>
```

```
<td>Service sync of type L4</td>
<td>AKO</td>
```

```
<td>Ingress sync of type L7</td>
<td>AKO</td>
```

```
<td>SE static route programming</td>
<td>AKO</td>
```

```
<td>AKO reboots/retry</td>
<td>AKO</td>
```

AKO install/ upgrade

Features Currently Not Supported

AKO v 0.91 does not support the following:

- OpenShift 3.x /OpenShift 4.x
- Any ecosystems/environments/CNIs other than the ones mentioned as supported here.
- Avi Annotations
- Non-default SE Groups
- Egress pod
- Custom resources definitions (CRD)
- Multiple tenants
- AKO does not support ingresses which do not have either hostname or a path defined.
- Ingresses in Kubernetes will be deployed only as Sharded L7 VSs in Avi. Dedicated VSs will not be supported for Ingresses.
- Services of type:LoadBalancer will only be deployed as dedicated L4 VSs in Avi.
- Changing of Sharding

Document Revision History

Date	Change Summary
June 30, 2020	Published the Design Guide for AKO version 0.9.1
April 30, 2020	Published the Design Guide for AKO (Tech Preview)

Related Reading

- Install Avi Kubernetes Operator
- Compatability Guide for AKO