Avi Kubernetes Operator Deployment Guide

Avi Technical Reference (v20.1)

Avi Kubernetes Operator Deployment Guide

view online

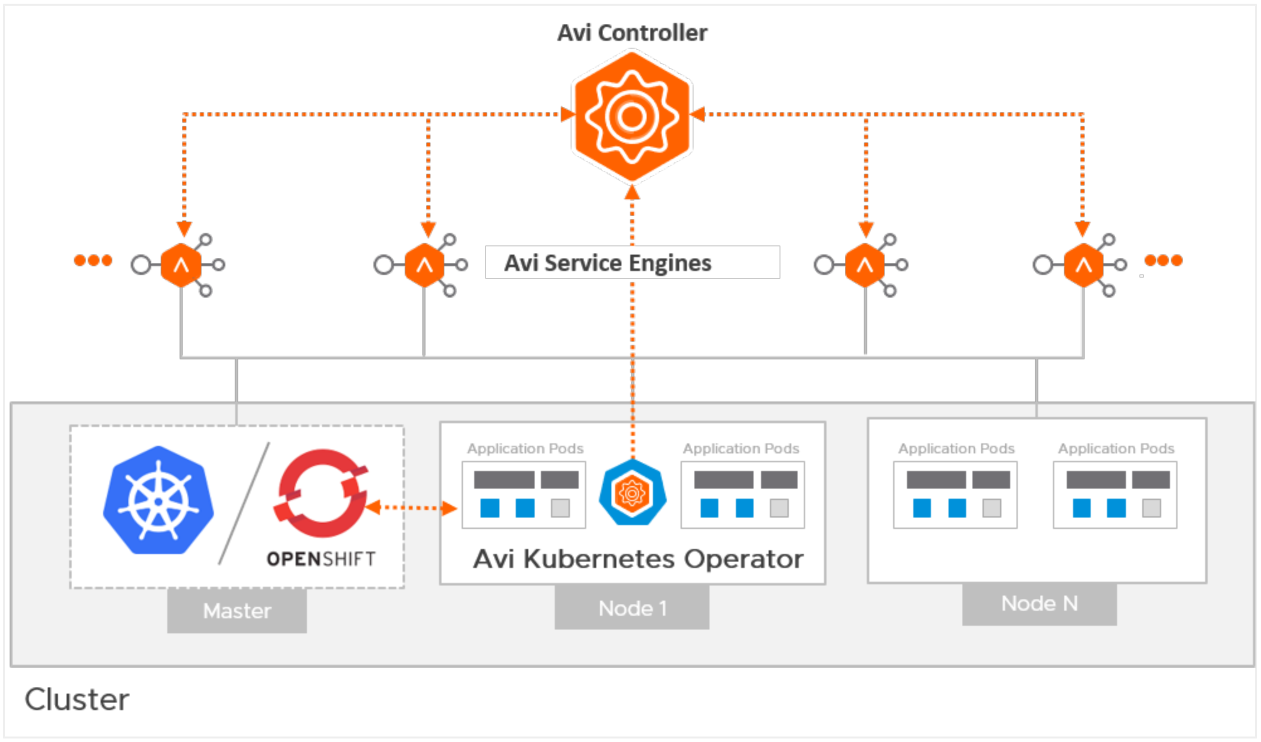
Overview

The Avi Kubernetes Operator (AKO) is an operator which works as an Ingress controller and performs Avi-specific functions in a Kubernetes/ OpenShift environment with the Avi Controller. It translates Kubernetes/ OpenShift 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/ OpenShift 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/ OpenShift environments, the Avi Controller is deployed outside the Kubernetes/ OpenShift 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/ OpenShift 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/ OpenShift 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/ OpenShift does not use the Kubernetes/ OpenShift cloud types. The integration with Kubernetes/ OpenShift and application-automation functions are handled by AKO and not by the Avi Controller.

Multiple Kubernetes/ OpenShift Clusters

A single Avi Cloud can be used for integration with multiple Kubernetes/ OpenShift clusters, with each cluster running its own instance of AKO. Clusters in the `clusterIP` mode are separated on the data plane through unique SE groups per cluster.

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

Starting with version 1.2.1, AKO supports a separate SE group per Kubernetes/OpenShift cluster. Each cluster will need to be configured with a separate SE group. However, multiple SE groups within the same cluster is not supported. As a best practice, it is recommended to use non-default SE groups for every cluster. SE group per cluster is not a requirement if AKO runs in the `nodeport` mode.

Avi Controller Version

AKO version 1.2.1 supports Avi Controller releases 18.2.10, 20.1.1 and above only.
Versions prior to 18.2.10 are not supported.

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:

Pods are not Externally Routable ? Static Routing

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 Kubernetes 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/ OpenShift nodes, you can turn on static route programming in AKO. With this, AKO syncs the pod CIDR for each Kubernetes/ OpenShift node and programs static route on the Avi Controller for each Pod CIDR with the Kubernetes/ OpenShift node IP as the next hop. Prior to AKO 1.2.1 static routing per cluster for Pod Networks leveraged VRFs. Starting with AKO 1.2.1, static routing per cluster uses a new label-based routing scheme. No additional user configuration is required for this label-based scheme, however the upgrade from AKO 1.1.1 to 1.2.1 will be service impacting requiring an AKO restart.

Pods are not externally routable ? NodePort

In cases where direct load-balancing to the pods is not possible, NodePort based services can be used as the pool members in the Avi virtual service as end points. For this functionality, configure the services referenced by Ingresses/Routes as type *NodePort* and set the `configs.serviceType` parameter to enable NodePort based Routes/Ingresses. The `nodeSelectorLabels.key` and `nodeSelectorLabels.value` parameters are specified during the AKO installation to select the required Nodes from the cluster for load balancing. The required nodes in the cluster need to be labelled with the configured key and value pair.

Pod Subnet is Routable

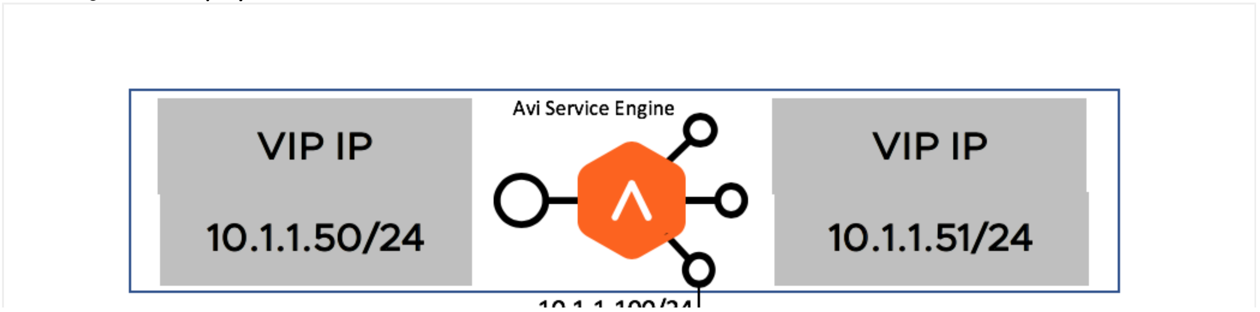
For CNIs like NSX-T CNI, AWS CNI (in EKS), Azure CNI (in AKS) etc., the pod subnet is externally routable. In this case no additional configuration is required to allow SEs to reach the Pod IPs. Set Static Route Programming to *Off* in the AKO configuration. SEs can be placed on any network and will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

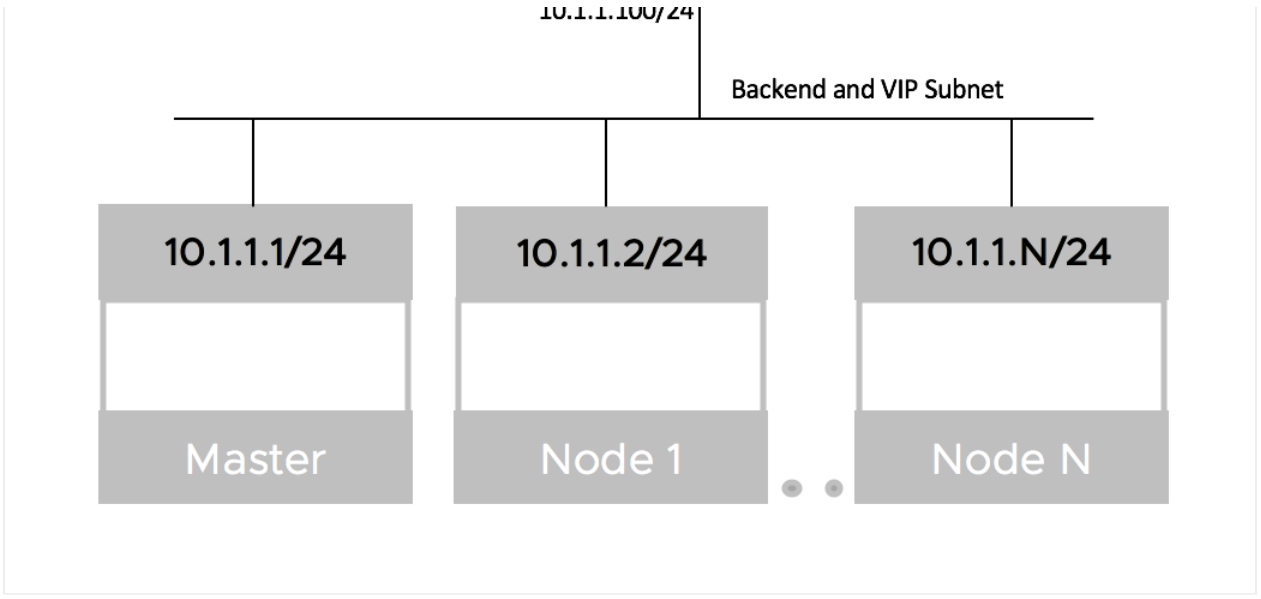
To know more about the CNIs supported in AKO version 1.2.1 click here.

Deployment Modes

Single Arm Deployment

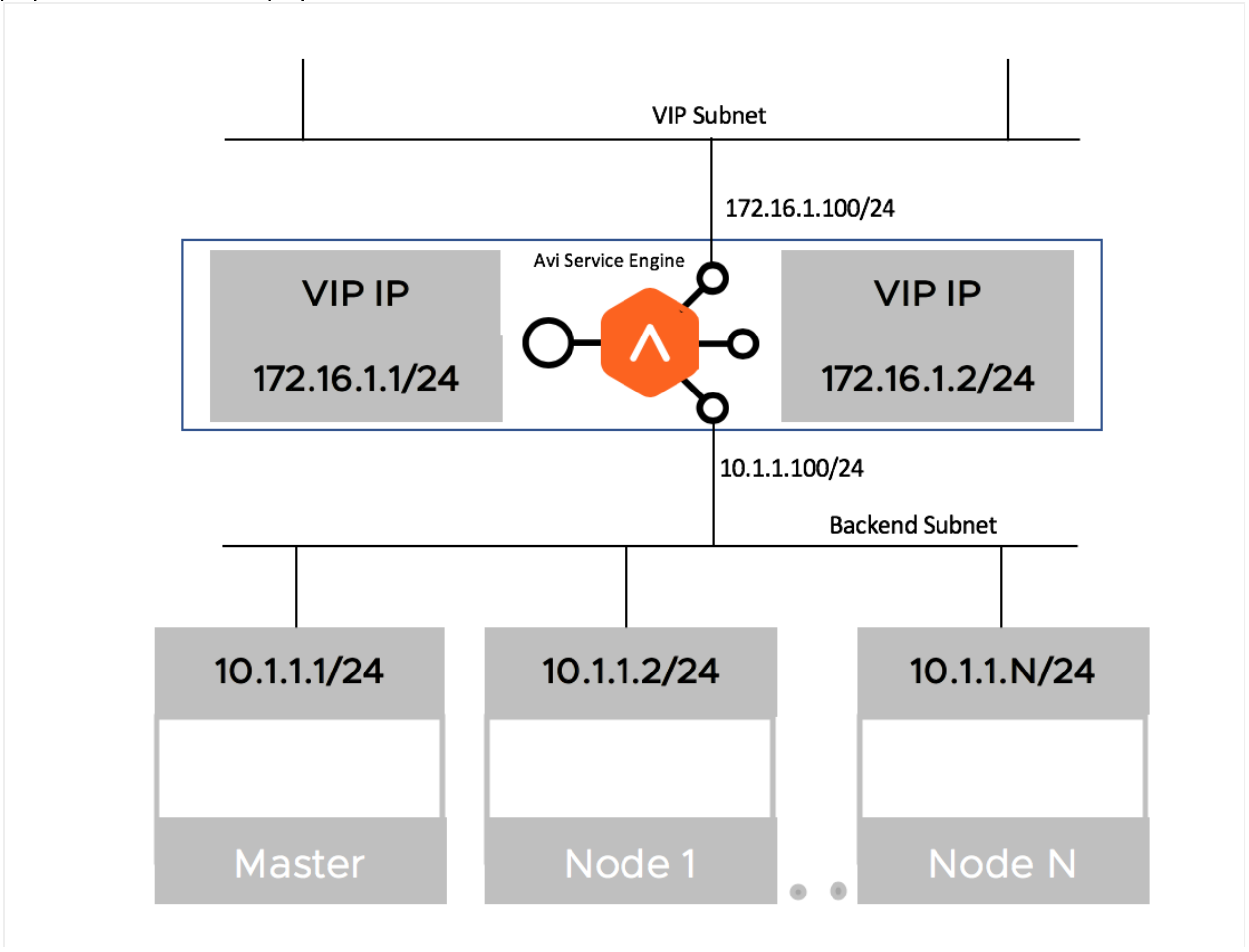
The deployment in which the virtual IP (VIP) address and the Kubernetes/ OpenShift cluster are in the same network subnet is called a *Single Arm Deployment*.





Two-Arm Deployment

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



AKO version 1.2.1 supports both Single-Arm and Two-Arm deployments with *vCenter Cloud* in *write-access* mode.

Handling of Kubernetes/ OpenShift 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/ OpenShift.
An example of such a service object in Kubernetes/ OpenShift 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/ OpenShift 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.

Service of Type NodePort

A service of Type NodePort can be used to send traffic to the pods using nodeports. This can be used where the option of static IP in VRF Context is not feasible.
This feature supports ingress/route attached to Service of type `NodePort`.

AKO will function either in the NodePort mode or in ClusterIP mode.

A new parameter serviceType has been introduced as configuration in AKO's values.yaml.
To use this feature, set the `serviceType` to `NodePort`.

Parameter	Description	Default
configs.serviceType	Type of Service to be used as backend for Routes/Ingresses	ClusterIP
nodeSelectorLabels.key	Key used as a label based selection for the nodes in NodePort mode	
nodeSelectorLabels.value	Value used as a label based selection for the nodes in NodePort mode	

By default, Kubernetes/ OpenShift configures the node port for any service of type `LoadBalancer`.
If the `config.serviceType` is set to `NodePort`, AKO would use NodePort as backend for service of type Loadbalancer instead of using Endpoints.
This is the default behaviour with `config.serviceType` set to `ClusterIP`.

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 path matches are by default Longest Prefix Matches (LPM). 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". However, a "/" path on the FQDN can still be programmed to point uniquely to a different pool without conflicts.

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).

OpenShift Route

In Openshift cluster, AKO can be used to configure routes. Ingress configuration is not supported. Currently the shard mode supported for openshift route is hostname.

Insecure Routes

```
apiVersion: v1
kind: Route
metadata:
  name: route1
spec:
  host: routehost1.avi.internal
  path: /foo
  to:
    kind: Service
    name: avisvc1
```

For insecure routes, AKO creates a Shared virtual service, pool group, and DataScript like insecure ingress. For pool name, the route configuration differs from ingress configuration. The service name is appended at the end of pool name. ### Shared

Virtual Service Pool Names for Route The formula to derive the Shared virtual service pool name for route is as follows:
`poolname = clusterName + "--" + hostname + "-" + namespace + "-" + routeName + "-" + serviceName`

Insecure Route with Alternate Backends

A route can also be associated with multiple services denoted by alternate backends. The requests that are handled by each service is governed by the service weight.

```
apiVersion: v1
kind: Route
metadata:
  name: route1
spec:
  host: routehost1.avi.internal
  path: /foo
  to:
    kind: Service
    name: avisvc1
    weight: 20
  alternateBackends:
  - kind: Service
    name: avisvc2
    weight: 10
```

For each backend of a route, a new pool is added. All such pools are added with same priority label - `hostname/path`. In case of the example mentioned above, two pools would be added with priority - `routehost1.avi.internal/foo`.
The ratio for a pool is the same as the weight specified for the service in the route.

Secure Route with Edge Termination

```
apiVersion: v1
kind: Route
metadata:
  name: secure-route1
spec:
  host: secure1.avi.internal
  path: /bar
  to:
    kind: Service
    name: avisvc1
  tls:
    termination: edge
    key: |-
    -----BEGIN RSA PRIVATE KEY-----
    ...
    ...
    -----END RSA PRIVATE KEY-----
    certificate: |-
    -----BEGIN CERTIFICATE-----
    ...
    ...
    -----END CERTIFICATE-----
```

Secure route is configured in Avi like secure ingress. An SNI virtual service is created for each hostname and for each host path, one pool group is created. However, for alternate backends, multiple pools are added in each pool group. Also, unlike Secure Ingresses, no redirect policy is configured for secure route for insecure traffic.

SNI Pool Names for Route

The formula to derive the SNI virtual service's pools for route is as follows:
`poolname = clusterName + "--" + namespace + "-" + hostname + "_" + path + "-" + routeName + "-" serviceName`

Secure Route with Termination Reencrypt

```
apiVersion: v1
  kind: Route
metadata:
  name: secure-route1
spec:
  host: secure1.avi.internal
  to:
    kind: Service
    name: service-name
  tls:
    termination: reencrypt
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
    destinationCACertificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

In case case of reencrypt, an SNI virtual service is created for each hostname and each host/path combination corresponds to a pool group in Avi. SSL is enabled in each pool for such virtual services with SSL profile set to `System-Standard`. In additon, if the `destinationCACertificate` is specified, a PKI profile with the `destinationCACertificate` is created for each pool.

Secure Route Insecure Edge Termination Policy: Redirect

```
apiVersion: v1
  kind: Route
metadata:
  name: secure-route1
spec:
  host: secure1.avi.internal
  to:
    kind: Service
    name: service-name
  tls:
    termination: edge
    insecureEdgeTerminationPolicy: redirect
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

In addition to the secure SNI virtual service, for this type of route, AKO creates a redirect policy on the shared parent of 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).

Secure Route Insecure Edge Termination Policy: Allow

```
apiVersion: v1
  kind: Route
metadata:
  name: secure-route1
spec:
  host: secure1.avi.internal
  to:
    kind: Service
    name: service-name
  tls:
    termination: edge
    insecureEdgeTerminationPolicy: Allow
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

If the `insecureEdgeTerminationPolicy` is *Allow*, then AKO creates an SNI VS for the hostname; also a pool is created for the same hostname which is added as member is pool group of the parent Shared virtual service. This enables the host to be accessed via both http(80) and https(443) port.

Passthrough Route

With passthrough routes, secure traffic is sent to the backend pods without TLS termination in AVI. A set of shared L4 Virtual Services are created by AKO to handle all TLS passthrough routes. Number of shards can be configured in helm with the flag `passthroughShardSize` in values.yaml. These virtual services would listen on port 443 and have one L4 ssl datascript each. Name of the virtual service would be of the format `clustername-- 'Shared-Passthrough'-shardnumber`. A number of shards can be configured using the flag `passthroughShardSize` while installation using helm.

```
apiVersion: v1
   kind: Route
 metadata:
   name: passthrough-route1
 spec:
   host: pass1.avi.internal
   to:
     kind: Service
     name: service-name
   tls:
     termination: edge
     insecureEdgeTerminationPolicy: Allow
```

For each passthrough host, one unique pool group is created with name `clustername-fqdn` and the pool group is attached to the DataScript of the virtual service that is derived by the sharding logic. In this case, a pool group with name `clustername-pass1.avi.internal` is created.

For each backend of a TLS passthrough route, one pool is created with ratio as per the route spec and is attached to the corresponding pool group.

If the `insecureEdgeTerminationPolicy`is redirect, another virtual service is created for each shared L4 VS, to handle insecure traffic on port 80. HTTP Request polices would be added in this VS for each FQDN with `insecureEdgeTermination` policy set to redirect. Both the virtual services listening on port 443 and 80 have a common virtual service VIP. This allows the DNS virtual service to resolve the hostname to one IP address consistently. The name of the insecure shared virtual service would be of the format `clustername--'Shared-Passthrough'-shard-number-'insecure'`.

For passthrough routes, the `insecureEdgeTerminationPolicy: Allow` is not supported in OpenShift.

Multi-Port Service Support in Openshift

A service in OpenShift can have multiple ports. In order for a Service to have multiple ports, OpenShift mandates them to have a name. To use such a service, the user must specify the `targetPort` within the port in route spec. The value of the `targetPort` can be integer value of the target port or name of the port. If the backend service has only one port, then the port field in route can be skipped, but it can not be skipped if the service has multiple ports. For example, consider the following service:

```
apiVersion: v1
     kind: Service
     metadata:
       labels:
         run: avisvc
     spec:
       ports:
       - name: myport1
         port: 80
         protocol: TCP
         targetPort: 80
       - name: myport2
         port: 8080
         protocol: TCP
         targetPort: 8080
       selector:
         app: my-app
       type: ClusterIP
```

In order to use this service in a route, the route spec can look like one of the following:

```
apiVersion: v1
    kind: Route
    metadata:
      name: route1
    spec:
      host: routehost1.avi.internal
      path: /foo
      port:
        targetPort: 8080
      to:
        kind: Service
        name: avisvc1
    apiVersion: v1
    kind: Route
    metadata:
      name: route1
    spec:
      host: routehost1.avi.internal
      path: /foo
      port:
        targetPort: myport2
      to:
        kind: Service
        name: avisvc1
```

AKO Created Object Naming Conventions

In the current AKO model, all Kubernetes/ OpenShift cluster objects are created on the admin tenant in Avi. This is true even for multiple Kubernetes/ OpenShift 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/ OpenShift 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/ OpenShift. * `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/ OpenShift 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/ OpenShift cluster. This is the only object in Avi that does not derive it's name from any of the Kubernetes/ OpenShift 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 version 1.2.1 does not support annotations. However, the `HTTPRule` and `HostRule CRDs` can be leveraged to customize the Avi configuration. Refer to the Custom Resource Definitions article for more information on what parameters can be tweaked.

Multi-Tenancy

AKO version 1.2.1 does not currently support Multi-Tenancy.

AKO Support

Refer to the the Compatibility Guide for information on supportability of features and environments with AKO.

Features Supported

The following features are supported in AKO version 1.2.1:

```
<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

Document Revision History

Date	Change Summary
September 18, 2020	Published the Design Guide for AKO version 1.2.1
July 20, 2020	Published the Design Guide for AKO version 1.2.1 (Tech Preview)

Related Reading

- Install Avi Kubernetes Operator
- Compatibility Guide for AKO