Networking

Guides

Configure Domain

Example Domain custom resource (CR)

Creating Domain by using the web console

Creating Domain by using the CLI

Use the Same Domain Across Multiple Clu

Subsequent Actions

Additional resources

Creating Certificates

Creating a certificate by using the web cor

Configure S

Why Service is

Example Cluste

Headless Servi

Configure Ingresses

Implementation Method

Example Ingress:

Creating a Ingress by using the CLI

Example: Loadl

Configure MetalLB

Prerequisites

Configure an External IP Address Pool by using the web console

Configure BGP Peers by using the web console

Configure an External IP Address Pool with L2Advertisement or BGPAdvertisement by using the CLI

Troubleshooting MetalLB

Configure GatewayAPI Gateway

Overview

Configure G

Overview

Prerequisites
Gateway Basics
Create Gateway
View Gateway Details

Prerequisites
Policy Attachme
Policy Attachme
Create Policies
icy
SPo
:Pol
affic

Configure GatewayAPI Route

Overview
Prerequisites
Configuration
View
Next Step
Related Tasks

Configure ALB

ALB
Frontend
Rule
Logs and Monitoring

Configure N

es
lotes
ks
Configuration

Configure CoreDNS

Overview
Configuration

How To

Tasks for Ingress-Nginx

Prerequisites
Max Connections
Request Timeout
Session Affinity (Sticky Sessions)
Header Modification
URL Rewrite

Tasks for Envoy Gateway

Overview
Prerequisites
Advanced Tasks
Related Documentation
More Configuration

Soft Data C

Prerequisites
Procedure
Verification

Configure Endpoint Health Checker

alb

Task: Migra

Trouble Shooting

How to Solve Inter-node Comm Find Who Cause the Error

Guides

Configure Domain

Example Domain custom resource (CR)

Creating Domain by using the web console

Creating Domain by using the CLI

Use the Same Domain Across Multiple Clu

Subsequent Actions

Additional resources

Creating Certificates

Creating a certificate by using the web cor

Configure S

Why Service is

Example Cluste

Headless Servi

Configure Ingresses

Implementation Method

Example Ingress:

Creating a Ingress by using the CLI

Example: Loadl

Configure MetalLB

Prerequisites

Configure an External IP Address Pool by using the web console

Configure BGP Peers by using the web console

Configure an External IP Address Pool with L2Advertisement or BGPAdvertisement by using the CLI

Troubleshooting MetalLB

Configure GatewayAPI Gateway

Overview

Prerequisites

Gateway Basics

Configure G

Overview

Prerequisites

Policy Attachme

Create Gateway

View Gateway Details

Policy Attachme

Create Policies

icy

SPo

:Pol

affic

Configure GatewayAPI Route

Overview

Prerequisites

Configuration

View

Next Step

Related Tasks

Configure ALB

ALB

Frontend

Rule

Logs and Monitoring

Configure N

es

lotes

ks

Configuration

Configure CoreDNS

Overview

Configuration

Configure Domain

Add domain name resources to the platform and allocate domains for use by all projects under a cluster or resources under a specific project. When creating a domain name, binding a certificate is supported.

> **NOTE**
>
> The domain names created on the platform should be resolved to the cluster's load balancing address before they can be accessed via the domain name. Therefore, you need to ensure that the domain names added on the platform have been successfully registered and that the domain names resolve to the cluster's load balancing address.

Successfully created and allocated domain names on the platform can be utilized in the following features of **Container Platform**:

- **Create Inbound Rules**: **Network Management** > **Inbound Rules** > **Create Inbound Rule**
- **Create Native Applications**: **Application Management** > **Native Applications** > **Create Native Application** > **Add Inbound Rule**
- **Add Listening Ports** for Load Balancing: **Network Management** > **Load Balancer Details** > **Add Listening Port**

Once the domain name is bound to a certificate, application developers can simply select the domain name when configuring the load balancer and inbound rules, allowing the use of the certificate that comes with the domain name for https support.

TOC

Example Domain custom resource (CR)

```yaml
# test-domain.yaml
apiVersion: crd.alauda.io/v2
kind: Domain
metadata:
  name: '${random-unique-name}'
  annotations:
    cpaas.io/secret-ref: developer.test.cn-xfd8x (1)
  labels:
    cluster.cpaas.io/name: global
    project.cpaas.io/name: demo
spec:
  name: developer.test.cn
  kind: full
```

(1) If certificates are enabled, an LTS-type Secret must be created in advance. The `secret-ref` is secret name.

Creating Domain by using the web console

1. Go to **Administrator**.

2. In the left navigation bar, click **Network Management** > **Domain Names**.
3. Click **Create Domain Name**.
4. Configure the relevant parameters according to the following instructions.

Parameter	Description
Type	• Domain: A complete domain name, e.g., `developer.test.cn`. • Wildcard Domain: A wildcard domain with a wildcard (*) character, e.g., `*.test.cn`, which includes all subdomains under the domain `test.cn`.
Domain	Enter a complete domain name or domain suffix based on the selected domain name type.
Allocate Cluster	If a cluster is allocated, you also need to select a project associated with the allocated cluster, such as all projects associated with the cluster.
Certificate	Includes the public key (tls.crt) and private key (tls.key) for creating a domain name-bound certificate. The project to which the certificate is allocated is the same as the bound domain name. **Notes**: • Binary file imports are not supported. • The bound certificate should meet the conditions of correct format, within the validity period, and signed for the domain name, etc. • After creating the bound certificate, the name format of the bound certificate is: domain name - random characters. • After creating the bound certificate, the bound certificate can be viewed in the certificate list, but updates and deletions of the bound certificate are only supported on the domain detail page. • After creating the bound certificate, updating the certificate content is supported, but replacing other certificates is not supported.

5. Click **Create**.

Creating Domain by using the CLI

```
kubectl apply -f test-domain.yaml
```

Use the Same Domain Across Multiple Clusters

You can configure the same domain to be used across multiple clusters by creating separate Domain resources in the global cluster with the identical `spec.name` value but different `cluster.cpaas.io/name` labels.

Configure via Web Console

1. Follow the steps in Creating Domain by using the web console.
2. Create two domain resources with the **same domain name** (e.g., `app.example.com`).
3. For each domain, select a different **Allocate Cluster** (e.g., cluster-a and cluster-b).

Configure via CLI

Create two Domain custom resources in the global cluster with identical `spec.name` but different `cluster.cpaas.io/name` labels:

> **NOTE**
>
> The domain name (`spec.name`) must be identical in both Domain resources, but the resource `metadata.name` should be unique. Both resources are created in the global cluster.

Domain for Cluster A:

```yaml
apiVersion: crd.alauda.io/v2
kind: Domain
metadata:
  name: '${random-unique-name}'
  labels:
    cluster.cpaas.io/name: cluster-a
    project.cpaas.io/name: project-a
spec:
  name: app.example.com  # Same domain name
  kind: full
```

Domain for Cluster B:

```yaml
apiVersion: crd.alauda.io/v2
kind: Domain
metadata:
  name: '${random-unique-name}'
  labels:
    cluster.cpaas.io/name: cluster-b
    project.cpaas.io/name: project-a
spec:
  name: app.example.com  # Same domain name as Cluster A
  kind: full
```

Subsequent Actions

- **Domain Registration**: Register the domain if the created domain has not been registered.
- **Domain Resolution**: Perform domain resolution if the domain does not point to the platform cluster's load balancing address.

Additional resources

- Configure Certificate

Creating Certificates

After the platform administrator imports the TLS certificate and assigns it to a specified project, developers with corresponding project permissions can use the certificate imported and assigned by the platform administrator when using inbound rules and load balancing functionalities. Subsequently, in scenarios such as certificate expiration, the platform administrator can update the certificate centrally.

> **NOTE**
>
> The certificate functionality is currently not supported for use in public cloud clusters. You can create TLS type secret dictionaries as needed within the specified namespace.

TOC

Creating a certificate by using the web console

1. Go to **Administrator**.
2. In the left navigation bar, click **Network Management** > **Certificates**.
3. Click **Create Certificate**.
4. Refer to the instructions below to configure the relevant parameters.

Parameter	Description
Assign Project	• All Projects: Assign the certificate for use in all projects associated with the current cluster. • Specified Project: Assign the certificate for use in the specified project. • No Assignment: Do not assign a project for now. After the certificate creation is completed, you can update the projects that can use the certificate through the **Update Project** operation.
Public Key	This refers to tls.crt. When importing the public key, binary files are not supported.
Private Key	This refers to tls.key. When importing the private key, binary files are not supported.

5. Click **Create**.

Configure Services

In Kubernetes, a Service is a method for exposing a network application that is running as one or more Pods in your cluster.

TOC

Why Service is Needed

1. Pods have their own IPs, but:
   - Pod IPs are not stable (they change if the Pod is recreated).
   - Directly accessing Pods becomes unreliable.
2. Service solves this by providing:
   - A stable IP and DNS name.
   - Automatic load balancing to the matching Pods.

Example ClusterIP type Service:

```yaml
# simple-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: ClusterIP (1)
  selector: (2)
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80 (3)
      targetPort: 80 (4)
```

1 The available type values and their behaviors are `ClusterIP`, `NodePort`, `LoadBalancer`, `ExternalName`

2 The set of Pods targeted by a Service is usually determined by a selector that you define.

3 `Service` port.

4 Bind `targetPort` of the Service to the Pod `containerPort`. In addition, you can reference `port.name` under the pod container.

Headless Services

Sometimes you don't need load-balancing and a single Service IP. In this case, you can create what are termed headless Services:

```
spec:
  clusterIP: None
```

Headless Services are useful when:

- You want to discover individual Pod IPs, not just a single service IP.
- You need direct connections to each Pod (e.g., for databases like Cassandra or StatefulSets).
- You're using StatefulSets where each Pod must have a stable DNS name.

Creating a service by using the web console

1. Go to **Container Platform**.
2. In the left navigation bar, click **Network** > **Services**.
3. Click **Create Service**.
4. Refer to the following instructions to configure the relevant parameters.

Parameter	Description
Virtual IP Address	If enabled, a ClusterIP will be allocated for this Service, which can be used for service discovery within the cluster. If disabled, a Headless Service will be created, which is usually used by **StatefulSet**.
Type	• **ClusterIP**: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. • **NodePort**: Exposes the Service on each Node's IP at a static port (the NodePort). • **ExternalName**: Maps the Service to the contents of the externalName field (for example, to the hostname api.foo.bar.example). • **LoadBalancer**: Exposes the Service externally using an external load balancer. Kubernetes does not directly offer a load balancing component; you must provide one, or you can integrate your Kubernetes cluster with a cloud provider.
Target Component	• **Workload**: The Service will forward requests to a **specific** workload, which matches the labels like `project.cpaas.io/name: projectname` and `service.cpaas.io/name: deployment-name`. • **Virtualization**: The Service will forward requests to a **specific** virtual machine or virtual machine group. • **Label Selector**: The Service will forward requests to a **certain type** of workload with specified labels, for example, `environment: release`.
Port	Used to configure the port mapping for this Service. In the following example, other podss within the cluster can call this Service via the virtual IP (if enabled) and TCP port *80*; the access requests will be forwarded to the externally exposed TCP port *6379* or *redis* of the target component's pods.

Parameter	Description
	• **Protocol**: The protocol used by the Service, supported protocols include: `TCP`, `UDP`, `HTTP`, `HTTP2`, `HTTPS`, `gRPC`. • **Service Port**: The service port number exposed by the Service within the cluster, that is, Port, e.g., *80*. • **Container Port**: The target port number (or name) that the service port maps to, that is, targetPort, e.g., *6379* or *redis*. • **Service Port Name**: Will be generated automatically. The format is `<protocol>-<service port>-<container port>`, for example: *tcp-80-6379* or *tcp-80-redis*.
Session Affinity	Session affinity based on the source IP address (ClientIP). If enabled, all access requests from the same IP address will be kept on the same server during load balancing, ensuring that requests from the same client are forwarded to the same server for processing.

5. Click **Create**.

Creating a service by using the CLI

```
kubectl apply -f simple-service.yaml
```

Create a service based on an existing deployment resource `my-app`.

```
kubectl expose deployment my-app \
  --port=80 \
  --target-port=8080 \
  --name=test-service \
  --type=NodePort \
  -n p1-1
```

Example: Accessing an Application Within the Cluste

```yaml
# access-internal-demo.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-clusterip
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
```

1. Apply this YAML:

```
kubectl apply -f access-internal-demo.yaml
```

2. Starting another Pod:

```
kubectl run test-pod --rm -it --image=busybox -- /bin/sh
```

3. Accessing the `nginx-clusterip` service in `test-pod` Pod:

```
wget -qO- http://nginx-clusterip
# or using DNS records created automatically by Kubernetes: <service-name>.<namespace>.svc.cluster.local
wget -qO- http://nginx-clusterip.default.svc.cluster.local
```

You should see a HTML response containing text like "Welcome to nginx!".

Example: Accessing an Application Outside the Cluste

```yaml
# access-external-demo.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-nodeport
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30080
```

1. Apply this YAML:

```
kubectl apply -f access-external-demo.yaml
```

2. Checking Pods:

```
kubectl get pods -l app=nginx -o wide
```

3. curl Service:

```
curl http://{NodeIP}:{nodePort}
```

You should see a HTML response containing text like "Welcome to nginx!".

Of course, it is also possible to access the application from outside the cluster by creating a Service of type LoadBalancer.

Note: Please configure the LoadBalancer service beforehand.

```yaml
# access-external-demo-with-loadbalancer.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-lb-service
spec:
  type: LoadBalancer
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
```

1. Apply this YAML:

```
kubectl apply -f access-external-demo-with-loadbalancer.yaml
```

2. Get external ip address:

```
kubectl get svc nginx-lb-service
```

```
NAME            TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)
AGE
nginx-service   LoadBalancer   10.0.2.57        34.122.45.100   80:3000
5/TCP   30s
```

`EXTERNAL-IP` is the address you access from your browser.

```
curl http://34.122.45.100
```

You should see a HTML response containing text like "Welcome to nginx!".

If EXTERNAL-IP is `pending`, the Loadbalancer service is not currently deployed on the cluster.

Example: ExternalName type of Servce

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-external-service
  namespace: default
spec:
  type: ExternalName
  externalName: example.com
```

1. Apply this YAML:

```
kubectl apply -f external-service.yaml
```

2. Try to resolve inside a Pod in the cluster:

```
kubectl run test-pod --rm -it --image=busybox -- sh
```

then:

```
nslookup my-external-service.default.svc.cluster.local
```

You'll see that it resolves to `example.com`.

LoadBalancer Type Service Annotations

AWS EKS Cluster

For detailed explanations of the EKS LoadBalancer Service annotations, please refer to the Annotation Usage Documentation ↗ .

Key	Value	Description
service.beta.kubernetes.io/aws-load-balancer-type	external: Use the official AWS LoadBalancer Controller.	Specifies the controller for the LoadBalancer type. **Note**: Please contact the platform administrator in advance to deploy the AWS LoadBalancer Controller.
service.beta.kubernetes.io/aws-load-balancer-nlb-target-type	• instance: Traffic will be sent to the pods via NodePort. • ip: Traffic routes directly to the pods (the cluster must use Amazon VPC CNI).	Specifies how traffic reaches the pods.
service.beta.kubernetes.io/aws-load-balancer-scheme	• internal: Private network.	Specifies whether to use a private network or a public network.

Key	Value	Description
	• internet-facing: Public network.	
service.beta.kubernetes.io/aws-load-balancer-ip-address-type	• IPv4 • dualstack	Specifies the supported IP address stack.

Huawei Cloud CCE Cluster

For detailed explanations of the CCE LoadBalancer Service annotations, please refer to the Annotation Usage Documentation ↗ .

Key	Value
kubernetes.io/elb.id	

kubernetes.io/elb.autocreate	Example: `{"type":"public","bandwidth_name":"cce-bandwidth`
	`1551163379627","bandwidth_chargemode":"bandwidth","bandwi`
	`["cn-north-4b"],"l4_flavor_name":"L4_flavor.elb.s1.small"`
	
	Note: Please read the Filling Instructions ↗ first and adjust the exam

kubernetes.io/elb.subnet-id	

Key	Value
kubernetes.io/elb.class	• union: Shared load balancing. • performance: Exclusive load balancing, only supported in Kuberr
kubernetes.io/elb.enterpriseID	

Azure AKS Cluster

For detailed explanations of the AKS LoadBalancer Service annotations, please refer to the Annotation Usage Documentation ↗ .

Key	Value	Description
service.beta.kubernetes.io/azure-load-balancer-internal	• true: Private network. • false: Public network.	Specifies whether to use a private network or a public network.

Google GKE Cluster

For detailed explanations of the GKE LoadBalancer Service annotations, please refer to the Annotation Usage Documentation ↗ .

Key	Value	Description
networking.gke.io/load-balancer-type	Internal	Specifies the use of a private network.
cloud.google.com/l4-rbs	enabled	Defaults to public. If this parameter is configured, traffic will route directly to the pods.

Example: LoadBalancer with MetalLB BGP and Local Traffic Policy

This example demonstrates how to configure a LoadBalancer Service using MetalLB BGP mode with `externalTrafficPolicy: Local` to achieve active-active load balancing without extra network hops.

Benefits

- **Active-active load balancing**: Traffic is distributed across multiple nodes simultaneously

- **No extra network hops**: Direct routing to pods without intermediate node forwarding
- **Better performance**: `externalTrafficPolicy: Local` preserves source IP and reduces latency
- **High availability**: BGP route announcements ensure traffic reaches healthy nodes

Prerequisites

Before configuring the LoadBalancer Service, ensure you have:

1. **MetalLB deployed**: See Creating External IP Address Pool for installation
2. **BGP Peer configured**: See Creating BGP Peers for BGP setup
3. **External IP address pool**: Configure an IPAddressPool with BGPAdvertisement

Steps

Deploy your application with a LoadBalancer Service using `externalTrafficPolicy: Local`:

```yaml
# nginx-loadbalancer-local-demo.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-loadbalancer-local
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
```

Key Configuration Points

externalTrafficPolicy: Local

The `externalTrafficPolicy: Local` setting provides several benefits:

- **Source IP preservation**: Client source IP is maintained, enabling proper logging and security policies
- **Direct pod routing**: Traffic goes directly to pods without node-level forwarding

LoadBalancer with BGP

When using MetalLB with BGP mode:

- Routes are advertised from nodes specified in the BGPAdvertisement nodeSelectors
- The BGP peer receives these announcements and can route traffic accordingly
- Node selector alignment between BGPPeer and BGPAdvertisement ensures consistent routing

Deployment Steps

1. **Deploy the application**:

   ```
   kubectl apply -f nginx-loadbalancer-local-demo.yaml
   ```

2. **Verify the LoadBalancer Service**:

   ```
   kubectl get svc nginx-loadbalancer-local
   ```

   Expected output:

   ```
   NAME                        TYPE           CLUSTER-IP      EXTERNAL-IP   P
   ORT(S)        AGE
   nginx-loadbalancer-local LoadBalancer   10.0.2.57       4.4.4.3       8
   0:30005/TCP   30s
   ```

3. **Test the service**:

   ```
   curl http://4.4.4.3
   ```

Verification

- **Monitor service endpoints**: `kubectl get endpoints nginx-loadbalancer-local`
- **Check service status**: `kubectl describe svc nginx-loadbalancer-local`

Configure Ingresses

Ingress rules (Kubernetes Ingress) expose HTTP/HTTPS routes from outside the cluster to internal routing (Kubernetes Service), enabling control of external access to computing components.

Create an Ingress to manage the external HTTP/HTTPS access to a Service.

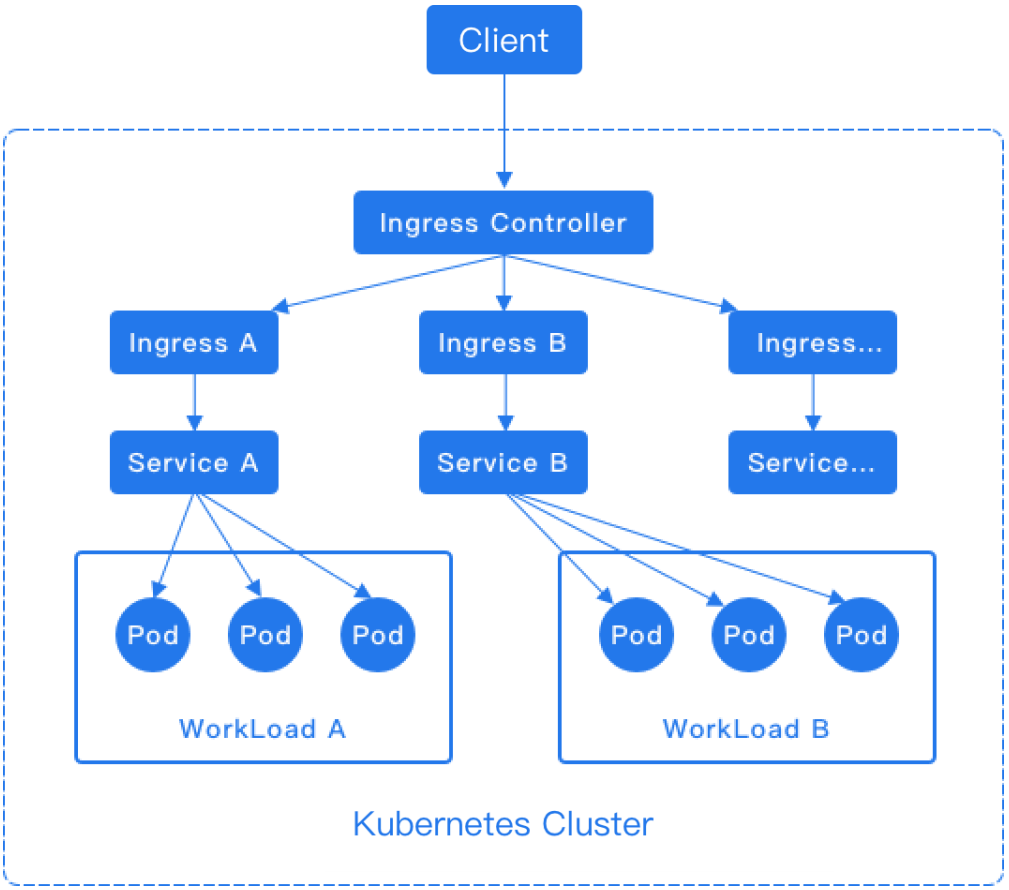
> **WARNING**
>
> When creating multiple ingresses within the same namespace, different ingresses **MUST NOT** have the same **Domain**, **Protocol**, and **Path** (i.e., duplicate access points are not allowed).

TOC

Implementation Method

Ingress rules depend on the implementation of the Ingress Controller, which is responsible for listening to changes in Ingress and Service. After a new Ingress is created, when the Ingress

Controller receives a request, it matches the forwarding rule from the Ingress and distributes the traffic to the specified internal routes, as shown in the diagram below.



NOTE

For the HTTP protocol, Ingress only supports the 80 port as the external port. For the HTTPS protocol, Ingress only supports the 443 port as the external port. The platform's load balancer will automatically add the 80 and 443 listening ports.

- Install ingress-nginx as ingress-controller via ingress-nginx-operator
- Install alb as ingress-controller via alb-operator

Example Ingress:

```yaml
# nginx-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx-ingress
  namespace: k-1
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: / 1
spec:
  ingressClassName: nginx 2
  rules:
    - host: demo.local 3
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: nginx-service
                port:
                  number: 80
```

1 To see more configurations please refer to nginx-configuration ↗.

2 `nginx` to using `ingress-nginx` controller, `$alb_name` to use alb as ingress controller.

3 If you only want to run ingress locally, configure the `hosts` beforehand.

Creating a Ingress by using the web console

1. Access the **Container Platform**.
2. In the left navigation bar, click **Network** > **Ingress**.
3. Click **Create Ingress**.
4. Reference the instructions below to configure certain parameters.

Parameter	Description
Ingress Class	Ingresses can be implemented by different controllers with different `IngressClass` name. If multiple ingress controllers are available on the platform, the user can select which one to use with this option.
Domain Name	Hosts can be precise matches (for example `foo.bar.com`) or a wildcard (for example `*.foo.com`). The domain names available are allocated by platform administrator.
Certificates	TLS secret or Certificates allocated by platform administrator.
Match Type and **Path**	• **Prefix**: Matches path prefixes, e.g., `/abcd` can match `/abcd/efg` or `/abcde`. • **Exact**: Matches exact paths, e.g., `/abcd`. • **Implementation specific**: If you are using a custom Ingress controller to manage the Ingress rules, you may choose to have the controller decide.
Service	External traffic will be forwarded to this Service.
Service Port	Specify which Service port the traffic will be forwarded to.

5. Click **Create**.

Creating a Ingress by using the CLI

```
kubectl apply -f nginx-ingress.yaml
```

Configure Subnets

TOC

IP Allocation Rules

> **NOTE**
>
> If a project or namespace is assigned multiple subnets, an IP address will be randomly selected from one of the subnets.

- Project Allocation:
  - If a project is not bound to a subnet, Pods in all namespaces under that project can only use IP addresses from the default subnet. If there are insufficient IP addresses in the default subnet, the Pods will not be able to start.
  - If a project is bound to a subnet, Pods in all namespaces under that project can only use IP addresses from that specific subnet.
- Namespace Allocation:

- If a namespace is not bound to a subnet, Pods in that namespace can only use IP addresses from the default subnet. If there are insufficient IP addresses in the default subnet, the Pods will not be able to start.
- If a namespace is bound to a subnet, Pods in that namespace can only use IP addresses from that specific subnet.

Calico Network

Creating subnets in the Calico network to achieve finer granularity of network isolation for resources within the cluster.

Constraints and Limitations

In an IPv6 cluster environment, the subnets created within the Calico network, by default, use VXLAN encapsulation. The ports required for VXLAN encapsulation differ from those of IPIP encapsulation. You need to ensure that UDP port 4789 is open.

Example Subnet custom resource (CR) with Calico Network

```yaml
# test-calico-subnet.yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: test-calico
spec:
  cidrBlock: 10.1.1.1/24
  default: false (1)
  ipipMode: Always (2)
  natOutgoing: true (3)
  private: false
  protocol: Dual
  v4blockSize: 30
```

(1) When `default` If true, use VXLAN encapsulation.

2. See Encapsulation Mode parameters and Encapsulation Protocol parameters.
3. See Outbound Traffic NAT parameters.

Creating a Subnet in the Calico network by using the web console

1. Go to **Administrator**.
2. In the left navigation bar, click **Network Management** > **Subnets**.
3. Click **Create Subnet**.
4. Refer to the following instructions to configure the relevant parameters.

Parameter	Description
CIDR	After allocating the subnet to a project or namespace, the container groups within the namespace will randomly use IP addresses within this CIDR for communication. **Note**: For the correspondence between CIDR and BlockSize, please refer to Reference Content.
Encapsulation Protocol	Select the encapsulation protocol. **IPIP** is not supported in dual-stack mode. • **IPIP**: Implements inter-segment communication using the IPIP protocol. • **VXLAN (Alpha)**: Implements inter-segment communication using the VXLAN protocol. • **No Encapsulation**: Directly connected through routing forwarding.

Parameter	Description
Encapsulation Mode	When the encapsulation protocol is **IPIP** or **VXLAN**, the encapsulation mode must be set, defaulting to **Always**. • **Always**: Always enable IPIP / VXLAN tunnels. • **Cross Subnet**: Enable IPIP / VXLAN tunnels only when the host is in different subnets; direct connection via routing forwarding when the host is in the same subnet.
Outbound Traffic NAT	Choose whether to enable outbound traffic NAT (Network Address Translation), which is enabled by default. It is primarily used to set the access addresses exposed to the external network when the subnet container group accesses the external network. When outbound traffic NAT is enabled, the host IP will be used as the access address for the current subnet container group; when not enabled, the IPs of the container groups in the subnet will be directly exposed to the external network.

5. Click **Confirm**.
6. On the subnet details page, select **Actions** > **Allocate Project** / **Allocate Namespace**.
7. Complete the configuration and click **Allocate**.

Creating a Subnet in the Calico network by using the CLI

```
kubectl apply -f test-calico-subnet.yaml
```

Reference Content

The dynamic matching relationship between CIDR and blockSize is shown in the table below.

CIDR	blockSize Size	Number of Hosts	Size of a Single IP Pool
`prefix<=16`	26	1024+	64
`16<prefix<=19`	27	256~1024	32
`prefix=20`	28	256	16
`prefix=21`	29	256	8
`prefix=22`	30	256	4
`prefix=23`	30	128	4
`prefix=24`	30	64	4
`prefix=25`	30	32	4
`prefix=26`	31	32	2
`prefix=27`	31	16	2
`prefix=28`	31	8	2
`prefix=29`	31	4	2
`prefix=30`	31	2	2
`prefix=31`	31	1	2

NOTE

Subnet configurations with prefixes greater than 31 are not supported.

Kube-OVN Network

Creating a subnet in the Kube-OVN Overlay Network to achieve more granular network isolation of resources in the cluster.

NOTE

The platform has a built-in **join** subnet for communication between nodes and Pods; please avoid conflicts in network segments between **join** and newly created subnets.

Example Subnet custom resource (CR) with Kube-OVN Overlay Network

```yaml
# test-overlay-subnet.yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: test-overlay-subnet
spec:
  default: false
  protocol: Dual
  cidrBlock: 10.1.0.0/23
  natOutgoing: true (1)
  excludeIps: (2)
    - 10.1.1.2
  gatewayType: distributed (3)
  gatewayNode: '' (4)
  private: false
  enableEcmp: false (5)
```

(1) See Outbound Traffic NAT parameters.

(2) See Reserved IP parameters.

(3) See Gateway Type parameters. The available values are `distributed` or `centralized`.

(4) See Gateway Nodes parameters.

(5) See ECMP parameters. Provided that you contact the administrator to enable the feature gate.

Creating a Subnet in the Kube-OVN Overlay Network by using the web console

1. Go to **Administrator**.
2. In the left navigation bar, click on **Network Management** > **Subnet**.
3. Click on **Create Subnet**.
4. Refer to the following instructions to configure the related parameters.

Parameter	Description
Network Segment	After assigning the subnet to the project or namespace, IPs within this segment will be randomly allocated for use by Pods.
Reserved IP	The set reserved IP will not be automatically allocated. For example, it can be used as the IP address for computing components' **fixed IP**.
Gateway Type	Select the type of gateway for the subnet to control the outbound traffic. - **Distributed**: Each host in the cluster can act as an outbound node for Pods on the current host, enabling distributed egress. - **Centralized**: All Pods in the cluster use one or more specific hosts as outbound nodes, facilitating external auditing and firewall control. Setting multiple centralized **gateway nodes** can achieve high availability.
ECMP (Alpha)	When choosing a **Centralized** gateway, the ECMP feature can be used. By default, the gateway operates in master-slave mode, with only the master gateway processing traffic. When enabling ECMP (Equal-Cost Multipath Routing), outbound traffic will be routed through multiple equal-cost paths to all available gateway nodes, thereby increasing the total throughput of the gateway. **Note**: Please enable ECMP-related features in advance.
Gateway Nodes	When using a **Centralized** gateway, select one or more specific hosts as gateway nodes.
Outbound Traffic NAT	Choose whether to enable outbound traffic NAT (Network Address Translation). By default, it is enabled. It is mainly used to set the access address exposed to the external network when the Pods in the subnet access the internet. When outbound traffic NAT is enabled, the host IP will be used as the

Parameter	Description
	access address for the Pods in the current subnet; when not enabled, the IPs of the Pods within the subnet will be directly exposed to the external network. In this case, using a centralized gateway is recommended.

5. Click **Confirm**.
6. On the subnet details page, select **Actions** > **Allocate Project** / **Namespace**.
7. Complete the configuration and click **Allocate**.

Creating a Subnet in the Kube-OVN Overlay Network by using the the CLI

```
kubectl apply -f test-overlay-subnet.yaml
```

Underlay Network

Creating subnets in the Kube-OVN Underlay network not only enables finer-grained network isolation for resources but also provides a better performance experience.

> **INFO**
>
> The container network in Kube-OVN Underlay requires support from the physical network. Please refer to the best practices Preparing the Kube-OVN Underlay Physical Network to ensure network connectivity.

Usage Instructions

The general process for creating subnets in the Kube-OVN Underlay network is: Add Bridge Network > Add VLAN > Create Subnet.

1. Default Network Card Name.
2. Configure Network Card by Node.

Add Bridge Network by using the web console (Optional)

```
# test-provider-network.yaml
kind: ProviderNetwork
apiVersion: kubeovn.io/v1
metadata:
  name: test-provider-network
spec:
  defaultInterface: eth1 (1)
  customInterfaces: (2)
    - interface: eth2
      nodes:
        - node1
  excludeNodes:
    - node2
```

(1) Default Network Card Name.

(2) Configure Network Card by Node.

A bridge network refers to a bridge, and after binding the network card to the bridge, it can forward container network traffic, achieving intercommunication with the physical network.

Procedure:

1. Go to **Administrator**.
2. In the left navigation bar, click **Network Management** > **Bridge Network**.
3. Click **Add Bridge Network**.
4. Configure the relevant parameters based on the following instructions.
   Note:

   - *Target Pod* refers to all Pods scheduled on the current node or Pods in namespaces bound to specific subnets scheduled to the current node. This depends on the scope of the subnet under the bridge network.
   - The nodes in the Underlay subnet must have multiple network cards, and the network card used by the bridge network must be exclusively assigned to the Underlay and cannot carry other traffic, such as SSH. For example, if the bridge network has three nodes planning for eth0, eth0, eth1 for exclusive use by the Underlay, then the default network card can be set as eth0, and the network card for node three can be eth1.

Parameter	Description
Default Network Card Name	By default, the target Pod will use this as the bridge network card for intercommunication with the physical network.
Configure Network Card by Node	The target Pods on the configured nodes will bridge to the specified network card instead of the default network card.
Exclude Nodes	When nodes are excluded, all Pods scheduled to these nodes will not bridge to any network card on these nodes. **Note**: Pods on excluded nodes will not be able to communicate with the physical network or cross-node container networks, and care should be taken to avoid scheduling related Pods to these nodes.

5. Click **Add**.

Add Bridge Network by using the CLI

```
kubectl apply -f test-provider-network.yaml
```

Add VLAN by using the web console (Optional)

```
# test-vlan.yaml
kind: Vlan
apiVersion: kubeovn.io/v1
metadata:
  name: test-vlan
spec:
  id: 0 1
  provider: test-provider-network 2
```

1 VLAN ID.

2 Bridge network reference.

The platform has a pre-configured **ovn-vlan** virtual LAN, which will connect to the **provider** bridge network. You can also configure a new VLAN to connect to other bridge networks, thereby achieving network isolation between VLANs.

Procedure:

1. Navigate to **Administrator**.
2. In the left navigation bar, click **Network Management** > **VLAN**.
3. Click **Add VLAN**.
4. Configure the relevant parameters based on the following instructions.

Parameter	Description
VLAN ID	The unique identifier for this VLAN, which will be used to differentiate different virtual LANs.
Bridge Network	The VLAN will connect to this bridge network for intercommunication with the physical network.

5. Click **Add**.

Add VLAN by using the CLI

```
kubectl apply -f test-vlan.yaml
```

Example Subnet custom resource (CR) with Kube-OVN Underlay Network

```yaml
# test-underlay-network.yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: test-underlay-network
spec:
  default: false
  protocol: Dual
  cidrBlock: 11.1.0.0/23
  gateway: 11.1.0.1
  excludeIps:
    - 11.1.0.3
  private: false
  allowSubnets: []
  vlan: test-vlan (1)
  enableEcmp: false
```

(1) VLAN reference.

Creating a Subnet in the Kube-OVN Underlay Network by using the web console

> **NOTE**
>
> The platform also pre-configures a **join** subnet for communication between nodes and Pods in Overlay transport mode. This subnet will not be used in Underlay transport mode, so it is crucial to avoid IP segment conflicts between **join** and other subnets.

Procedure:

1. Navigate to **Administrator**.
2. In the left navigation bar, click **Network Management** > **Subnet**.
3. Click **Create Subnet**.
4. Configure the relevant parameters based on the following instructions.

Parameter	Description
VLAN	The VLAN to which the subnet belongs.
Subnet	After assigning the subnet to a project or namespace, IPs within the physical subnet will be randomly allocated for use by Pods.
Gateway	The physical gateway within the above subnet.
Reserved IP	The specified reserved IP will not be automatically assigned. For example, it can be used as the IP for the compute component **fixed IP**.

5. Click **Confirm**.
6. On the subnet details page, select **Action** > **Assign Project** / **Namespace**.
7. Complete the configuration and click **Assign**.

Creating a Subnet in the Kube-OVN Underlay Network by using the CLI

```
kubectl apply -f test-underlay-network.yaml
```

Related Operations

When both Underlay and Overlay subnets exist in a cluster, you can configure the Automatic Intercommunication Between Underlay and Overlay Subnets as needed.

Subnet Management

Updating Gateway by using the web console

This includes changing the outbound traffic method, gateway nodes, and NAT configuration.

1. Go to **Administrator**.

2. In the left sidebar, click on **Network Management** > **Subnets**.
3. Click the name of the subnet.
4. Select **Action** > **Update Gateway**.
5. Update the parameter configurations; refer to the Parameter Description for details.
6. Click **OK**.

Updating Gateway by using the CLI

```
kubectl patch subnet test-overlay-subnet --type=json -p='[
  {"op": "replace", "path": "/spec/gatewayType", "value": "centralized"},
  {"op": "replace", "path": "/spec/gatewayNode", "value": "192.168.66.21
0"},
  {"op": "replace", "path": "/spec/natOutgoing", "value": true},
  {"op": "replace", "path": "/spec/enableEcmp", "value": true}
]'
```

Updating Reserved IPs by using the web console

The gateway IP cannot be removed from the reserved IPs, while other reserved IPs can be edited, deleted, or added freely.

1. Go to **Administrator**.
2. In the left sidebar, click on **Network Management** > **Subnets**.
3. Click the name of the subnet.
4. Select **Action** > **Update Reserved IP**.
5. After completing the updates, click **Update**.

Updating Reserved IPs by using the CLI

```
kubectl patch subnet test-overlay-subnet --type=json -p='[
  {
    "op": "replace",
    "path": "/spec/excludeIps",
    "value": ["10.1.0.1", "10.1.1.2", "10.1.1.4"]
  }
]'
```

Assigning Projects by using the web console

Assigning subnets to specific projects helps teams better manage and isolate network traffic for different projects, ensuring that each project has sufficient network resources.

1. Navigate to **Administrator**.
2. In the left sidebar, click on **Network Management** > **Subnets**.
3. Click the name of the subnet.
4. Select **Action** > **Assign Project**.
5. After adding or removing projects, click **Assign**.

Assigning Projects by using the CLI

```
kubectl patch subnet test-overlay-subnet --type=json -p='[
  {
    "op": "replace",
    "path": "/spec/namespaceSelectors",
    "value": [
      {
        "matchLabels": {
          "cpaas.io/project": "cong"
        }
      }
    ]
  }
]'
```

Assigning Namespaces by using the web console

Assigning subnets to specific namespaces allows for finer network isolation.

Note: The assignment process will rebuild the gateway, and outbound data packets will be discarded! Please ensure no business applications are currently accessing external clusters.

1. Navigate to **Administrator**.
2. In the left sidebar, click on **Network Management** > **Subnets**.
3. Click the name of the subnet.
4. Select **Action** > **Assign Namespace**.
5. After adding or removing namespaces, click **Assign**.

Assigning Namespaces by using the CLI

```
kubectl patch subnet test-overlay-subnet --type=json -p='[
  {
    "op": "replace",
    "path": "/spec/namespaces",
    "value": ["cert-manager"]
  }
]'
```

Expanding Subnets by using the web console

When the reserved IP range of a subnet reaches its usage limit or is about to be exhausted, it can be expanded based on the original subnet range without affecting the normal operation of existing services.

1. Navigate to **Administrator**.
2. In the left sidebar, click on **Network Management** > **Subnets**.
3. Click the name of the subnet.
4. Select **Action** > **Expand Subnet**.
5. Complete the configuration and click **Update**.

Expanding Subnets by using the CLI

```
kubectl patch subnet test-overlay-subnet --type=json -p='[
  {
    "op": "replace",
    "path": "/spec/cidrBlock",
    "value": "10.1.0.0/22"
  }
]'
```

Managing Calico Networks

Support for assigning projects and namespaces; for details, please refer to the project assignment and namespace assignment.

Delete Subnet by using the web console

NOTE

- When a subnet is deleted, if there are still container groups using the IPs within the subnet, the container groups can continue to run and the IP addresses will remain unchanged, but they will be unable to communicate over the network. The container groups can be rebuilt to use IPs within the default subnet, or assign a new subnet to the namespace where the container groups reside for usage.
- The default subnet cannot be deleted.

1. Go to **Administrator**.
2. In the left navigation bar, click **Network Management** > **Subnets**.
3. Click ⋮ > **Delete**, and proceed with the deletion.

Delete Subnet by using the CLI

```
kubectl delete subnet test-overlay-subnet
```

Configure MetalLB

TOC

Prerequisites

Please ensure that you have read the Installation documentation before proceeding.

Configure an External IP Address Pool by using the web console

1. Go to **Administrator**.
2. In the left navigation bar, click **Network Management** > **External IP Address Pool**.
3. Click **Create External IP Address Pool**.
4. Refer to the following instructions to configure certain parameters.

Parameter	Description
Type	• L2: Communication and forwarding based on MAC addresses, suitable for small-scale or local area networks that require simple and fast layer 2 switching, with advantages in simple configuration and low latency. • BGP (Alpha): Routing and forwarding based on IP addresses, using BGP protocol to exchange routing information, suitable for large-scale networks requiring complex routing across multiple autonomous systems, with advantages in high scalability and reliability.
IP Resources	Support input in CIDR and IP range formats. Click **Add** to support multiple entries, examples as follows: **CIDR**: `192.168.1.1/24`. **IP Range**: `192.168.2.1` ~ `192.168.2.255`.
Available Nodes	In L2 mode, available nodes are those used to carry all VIP traffic; in BGP mode, available nodes are those used to carry VIPs, establish BGP connections with peers, and announce routes externally. • **Node Name**: Select available nodes based on node names. • **Label Selector**: Select available nodes based on labels. • **Show Node Details**: View final available nodes in a list format. **Note**: • When using BGP type, the available nodes are the next-hop nodes; ensure that the selected available nodes are a subset of the BGP Connection Nodes. • You can configure either the label selector or the node name separately to choose available nodes; if both are configured simultaneously, the final available nodes are the intersection of both.
BGP Peers	Select BGP peers; please refer to BGP Peers for specific configurations.

5. Click **Create**.

Configure BGP Peers by using the web console

1. Go to **Administrator**.
2. In the left navigation bar, click **Network Management** > **BGP Peers**.
3. Click **Create BGP Peer**.
4. Refer to the instructions below to configure the parameters.

Parameter	Description
Local AS Number	The AS number of the AS where the BGP-connected node resides. **Note**: If there are no special requirements, it is recommended to use an IBGP configuration, meaning the local AS number should be consistent with the peer AS number.
Peer AS Number	The AS number of the AS where the BGP peer resides.
Peer IP	The IP address of the BGP peer, which must be a valid IP address capable of establishing a BGP connection.
Local IP	The IP address of the BGP-connected node. When the BGP-connected node has multiple IPs, select the specified local IP to establish a BGP connection with the peer.
Peer Port	The port number of the BGP peer.
BGP-Connected Node	The node that establishes the BGP connection. If this parameter is not configured, all nodes will establish BGP connections.
eBGP Multi-Hop	Allows the establishment of BGP sessions between BGP routers that are not directly connected. When this feature is enabled, the default TTL value of BGP packets is 5, allowing the establishment of BGP peer relationships across multiple intermediate network devices, making network design more flexible.

Parameter	Description
RouterID	A 32-bit numeric value (usually represented in dotted-decimal format, similar to IPv4 address format) used to uniquely identify a BGP router in the BGP network, generally used for establishing BGP neighbor relationships, detecting routing loops, selecting optimal paths, and troubleshooting network issues.

5. Click **Create**.

Configure an External IP Address Pool with L2Advertisement or BGPAdvertisement by using the CLI

```yaml
# ippool-with-L2advertisement.yaml
kind: IPAddressPool
apiVersion: metallb.io/v1beta1
metadata:
  name: test-ippool
  namespace: metallb-system
spec:
  addresses:
    - 13.1.1.1/24
  avoidBuggyIPs: true
---
kind: L2Advertisement
apiVersion: metallb.io/v1beta1
metadata:
  name: test-ippool
  namespace: metallb-system
spec:
  ipAddressPools:
    - test-ippool 1
  nodeSelectors:
    - matchLabels: {}
      matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
            - 192.168.66.210
```

BGP mode

```yaml
# ippool-with-bgpadvertisement.yaml
kind: IPAddressPool
apiVersion: metallb.io/v1beta1
metadata:
  name: test-pool-bgp
  namespace: metallb-system
spec:
  addresses:
    - 4.4.4.3/23
  avoidBuggyIPs: true
---
kind: BGPAdvertisement
apiVersion: metallb.io/v1beta1
metadata:
  name: test-pool-bgp
  namespace: metallb-system
spec:
  ipAddressPools:
    - test-pool-bgp
  nodeSelectors:
    - matchLabels:
        alertmanager: 'true'
  peers:
    - test-bgp-example
```

```
kubectl apply -f ippool-with-L2advertisement.yaml -f ippool-with-bgpadver
tisement.yaml
```

Troubleshooting MetalLB

Symptom	Possible Cause	Resolution
No external IP assigned	No valid IPAddressPool or pool misconfigured	Verify IP range and namespace
Pods CrashLoop	Speaker or Controller RBAC missing	Check Operator permissions

Symptom	Possible Cause	Resolution
BGP not established	ASN mismatch or peer unreachable	Check `BGPPeer` spec and network routes
L2 not working	Wrong VLAN or ARP filtering	Use `arping` to verify broadcast reachability

To see more Troubleshooting MetalLB ↗

Configure GatewayAPI Gateway

TOC

Overview

This document explains how to configure a `Gateway` after the Envoy Gateway operator and `EnvoyGatewayCtl` are ready. A `Gateway` defines how traffic enters the gateway, while the companion `EnvoyProxy` controls how the underlying Envoy data plane is deployed.

In the recommended workflow, this document comes after Envoy Gateway Operator and before Configure GatewayAPI Route.

Prerequisites

Please ensure that you have completed the following before proceeding:

1. Read Envoy Gateway Operator to understand the basic concepts and resource relationships
2. Install the Envoy Gateway operator and create an `EnvoyGatewayCtl`

> **NOTE**
>
> This document first explains the main Gateway concepts and then shows how to create a `Gateway`. If you are already familiar with these concepts, you can skip to the Create Gateway section.

Gateway Basics

What is a Gateway

A `Gateway` is the entry point for traffic entering your cluster. It defines how external requests are received and routed to your backend services. A `Gateway` mainly defines:

- **Listeners**: Define the ports, protocols, and hostnames the gateway listens on
- **GatewayClass**: Selects which gateway controller manages this `Gateway`
- **Infrastructure reference**: References an `EnvoyProxy` that controls how the underlying Envoy data plane is deployed

Gateway Exposure (Service Type)

Service Type configures how the gateway is exposed through the underlying Envoy Service. There are three modes: LoadBalancer, NodePort, and ClusterIP.

In YAML, this setting is configured in the companion `EnvoyProxy` resource at `.spec.provider.kubernetes.envoyService.type`.

LoadBalancer (Recommended)

The advantage is ease of use and high-availability load balancing capabilities. To use LoadBalancer, the cluster must have LoadBalancer support, which can be enabled via MetalLB.

When using MetalLB, you can specify a static VIP through service annotations. In the Web Console, use the **Service Annotation** field:

```
metallb.universe.tf/address-pool: ADDRESS_POOL_NAME
# Or specify a specific IP directly
metallb.universe.tf/loadBalancerIPs: VIP_IP
```

For more details, see How To Specify a VIP When Using MetalLB.

NodePort

The advantage is that it doesn't require any external dependencies.

However, using NodePort has these disadvantages:

- When using NodePort, Kubernetes assigns NodePort port numbers that differ from the service's own ports. You must use the NodePort port number for access, not the service port.
- The service can be accessed via any node IP address in the cluster, which may pose potential security risks.

How to Get the Correct Port When Using NodePort

In the Gateway details page, when Service Type is NodePort, the listener list displays the NodePort column showing the assigned port numbers. You can also use the following command:

```
kubectl get svc -n ${ENVOYGATEWAYCTL_NS} -l gateway.envoyproxy.io/owning-gateway-name=${GATEWAY_NAME} -o=jsonpath="{.items[0].spec.ports[?(@.port==${PORT})].nodePort}"
```

The output is the NodePort.

ClusterIP

Very convenient if you don't need external exposure.

Listener Configuration

A Listener defines the port and protocol for the gateway to listen on. In HTTP or HTTPS protocols, different hostnames can be treated as different listeners.

You cannot create a listener with a conflicting port, protocol, or hostname.

You must create at least one listener in the `Gateway`.

Port and Protocol

Each listener is configured with a port number and a protocol. Supported protocols are: HTTP, HTTPS, TCP, UDP, TLS.

AllowRouteNS

By default, Routes can only attach to a Gateway in the `Same` namespace. To allow cross-namespace routing, use the `Allowed Routes Namespace` field:

- `Same`: Allow Routes in the same namespace to attach to this listener.
- `All`: Allow Routes in any namespace to attach to this listener.
- `Selector`: Allow Routes in namespaces matched by the selector to attach to this listener.

In ACP, a project is identified by labels on namespaces, for example `cpaas.io/project: <project-name>`. If you want a listener to be used only by Routes from a specific project, use `Selector` and match the project label on the target namespaces.

The listener's `Allowed Routes Namespace` setting and its protocol together determine which listeners are available in the Route Web Console when you publish a route to a listener.

For more information, see attach to gateway created in other ns.

TLS Configuration

For HTTPS and TLS protocols, you need to configure TLS settings.

TLS Modes:

TLS Mode	Description	Certificate Required
Terminate	Envoy terminates TLS and decrypts traffic before forwarding to backend services	Yes, must select a TLS certificate
Passthrough	Envoy passes through encrypted TLS traffic to backend services without decryption	No

> **NOTE**
>
> - HTTPS protocol only supports **Terminate** mode
> - TLS protocol supports both **Terminate** and **Passthrough** modes
> - TLS listeners with **Passthrough** mode support TLSRoute

- TLS listeners with **Terminate** mode support TCPRoute

Certificate Requirements:

- By default, you can only use secrets created in the same namespace
- The secret must be of type `kubernetes.io/tls` and contain `tls.crt` and `tls.key` keys
- For cross-namespace secrets, see use secret created in other ns

EnvoyProxy (Deployment Configuration)

Envoy Gateway uses the `EnvoyProxy` resource to control gateway deployment configurations. We recommend creating a dedicated `EnvoyProxy` resource for each Gateway and referencing it through the Gateway's `.spec.infrastructure.parametersRef` field.

When you create a `Gateway` from the Web Console by using a `GatewayClass` created by `EnvoyGatewayCtl`, the console automatically creates a companion `EnvoyProxy` resource with the same name and namespace.

When you create a `Gateway` by applying YAML, you are responsible for keeping the `Gateway` `.spec.infrastructure.parametersRef` and the referenced `EnvoyProxy` resource consistent.

This one-to-one mapping approach provides better isolation and more granular control over deployment configurations such as:

- Replicas
- Resource limits and requests
- Node selectors
- Service type and annotations
- Image repository

Image Repository

The image repository is pre-configured with the default value for your cluster. Do not modify it unless necessary.

For other deployment configuration methods, see deployment-mode ↗.

Create Gateway

Via Web Console

1. Navigate to `Alauda Container Platform -> Networking -> Gateway -> Gateways`
2. Click the `Create Gateway` button
3. On the `Create Gateway` page, select the `GatewayClass` created by your `EnvoyGatewayCtl`. In the recommended default example from Envoy Gateway Operator, this is `envoy-gateway-operator-cpaas-default`.

The page displays the following configuration options:

Field	Description	YAML Path
Name	Gateway name	gateway: `.metadata.name` envoyproxy: `.metadata.name`
GatewayClass	Which GatewayClass to use	gateway: `.spec.gatewayClassName`
Service Type	Service Type	envoyproxy: `.spec.provider.kubernetes.envoyService.ty`
Service Annotation	Service annotations	envoyproxy: `.spec.provider.kubernetes.envoyService.an`
Resource Limits	Deployment resource limits	envoyproxy: `.spec.provider.kubernetes.envoyDeployment.container.`
Replicas	Deployment replicas	envoyproxy: `.spec.provider.kubernetes.envoyDeployment`

Field	Description	YAML Path
Node Labels	Deployment node selectors	envoyproxy: `.spec.provider.kubernetes.envoyDeployment.nodeSelect`
Listener	Listener	gateway: `.spec.listeners`

WARNING

The Web Console form only supports GatewayClasses created by `EnvoyGatewayCtl`. For other GatewayClasses, use the YAML editor.

NOTE

When using an `EnvoyGatewayCtl`-created `GatewayClass`, the Web Console automatically creates a companion EnvoyProxy resource matching the Gateway's name and namespace.

Listener Configuration

When creating or editing a listener, you can configure the following:

Field	Description	YAML Path
Name	The name of the listener	`.spec.listeners[].name`
Port	The port number the listener listens on	`.spec.listeners[].port`
Protocol	The protocol for the listener. Options: HTTP, HTTPS, TCP, UDP, TLS	`.spec.listeners[].protocol`

Field	Description	YAML Path
Hostname	Optional. The hostname for the listener	`.spec.listeners[].hostname`
Allowed Routes Namespace	Controls which namespaces can attach routes to this listener	`.spec.listeners[].allowedRoutes.namespaces.from`

HTTPS Protocol Configuration

When selecting **HTTPS** as the protocol:

Field	Description	YAML Path
Certificates	Required. Select the Kubernetes secret containing the TLS certificate.	`.spec.listeners[].tls.certificateRefs`

NOTE

- HTTPS protocol only supports **Terminate** mode
- Certificate selection is mandatory for HTTPS listeners
- You can only use secrets created in the same namespace by default

TLS Protocol Configuration

When selecting **TLS** as the protocol:

Configuration Fields

Field	Description	YAML Path
TLS Mode	Select the TLS mode. Options: Terminate, Passthrough. Default: Terminate	`.spec.listeners[].tls.mode`
TLS Certificate	Required only when TLS Mode is Terminate. Select the secret containing the TLS certificate.	`.spec.listeners[].tls.certificateRefs`

For TLS mode details, see TLS Configuration.

Via YAML

Replace `envoy-gateway-operator-cpaas-default` with the `GatewayClass` created by your own `EnvoyGatewayCtl` if you are not using the recommended default example.

The following minimal example creates an HTTP `Gateway` and a dedicated `EnvoyProxy`.

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo
  namespace: demo
spec:
  infrastructure:
    parametersRef:
      group: gateway.envoyproxy.io
      kind: EnvoyProxy
      name: demo
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: http
      port: 80
      protocol: HTTP
      allowedRoutes:
        namespaces:
          from: Same
---
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
  namespace: demo
spec:
  provider:
    kubernetes:
      envoyService:
        type: ClusterIP
      envoyDeployment:
        replicas: 1
        container:
          imageRepository: registry.alauda.cn:60080/acp/envoyproxy/envoy
          resources:
            limits:
              cpu: '1'
              memory: 1Gi
            requests:
              cpu: '1'
              memory: 1Gi
    type: Kubernetes
```

Complete Example with Multiple Listener Types

If you need more listener types, use the following complete example:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo
  namespace: demo
spec:
  infrastructure: (1)
    parametersRef:
      group: gateway.envoyproxy.io
      kind: EnvoyProxy
      name: demo
  gatewayClassName: envoy-gateway-operator-cpaas-default (2)
  listeners: (3)
    - name: http
      port: 80
      hostname: a.com (4)
      protocol: HTTP (5)
      allowedRoutes:
        namespaces:
          from: Same (6)
    - name: https
      port: 443
      hostname: a.com
      protocol: HTTPS
      allowedRoutes:
        namespaces:
          from: Same
      tls: (7)
        mode: Terminate
        certificateRefs:
          - name: demo-tls
    - name: tls-passthrough
      port: 8443
      hostname: tls.example.com
      protocol: TLS
      allowedRoutes:
        namespaces:
          from: Same
      tls:
        mode: Passthrough
    - name: tls-terminate
      port: 9443
      hostname: secure.example.com
```

```
      protocol: TLS
      allowedRoutes:
        namespaces:
          from: Same
      tls:
        mode: Terminate
        certificateRefs:
          - name: demo-tls
    - name: tcp
      port: 8080
      protocol: TCP
      allowedRoutes:
        namespaces:
          from: Same
    - name: udp
      port: 8081
      protocol: UDP
      allowedRoutes:
        namespaces:
          from: Same
---
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo (8)
  namespace: demo
spec:
  provider:
    kubernetes:
      envoyService:
        type: ClusterIP (9)
      envoyDeployment:
        replicas: 1
        container:
          imageRepository: registry.alauda.cn:60080/acp/envoyproxy/envoy
(10)
          resources: (11)
            limits:
              cpu: '1'
              memory: 1Gi
            requests:
              cpu: '1'
              memory: 1Gi
    type: Kubernetes (12)
```

1. Reference to EnvoyProxy resource for deployment configuration
2. Replace `envoy-gateway-operator-cpaas-default` with your own `GatewayClass` if needed
3. `listeners` defines how traffic enters the gateway
4. `hostname` affects how host-based routing matches the listener
5. `protocol` determines which route kinds can attach to the listener
6. `allowedRoutes` controls which namespaces can attach routes
7. `tls` configures TLS termination or passthrough for HTTPS and TLS listeners
8. The `EnvoyProxy` name must match `.spec.infrastructure.parametersRef.name`
9. `envoyService.type` controls gateway exposure
10. Keep `imageRepository` unchanged unless necessary for your environment
11. `resources` configures the Envoy data plane resource limits and requests
12. Keep `provider.type` as `Kubernetes`

View Gateway Details

In the Gateway details page, the listener list shows the following information:

Column	Description
Name	The listener name
Protocol	The listener protocol (HTTP, HTTPS, TCP, UDP, TLS)
Port	The configured port number
NodePort	Displayed when Service Type is NodePort. Shows the assigned NodePort number for accessing the listener.

NOTE

When the Gateway Service Type is **NodePort**, the listener list displays an additional **NodePort** column. Use the NodePort value, not the service port, to access the gateway. For more details, see How to Get the Correct Port When Using NodePort.

Listener and Route Reference

Use the following rules when attaching `Route` resources to a `Gateway`.

Hostname

The hostname in a listener is a unique identifier for listeners that have the same protocol. You cannot add or update a conflicting listener in a gateway.

Hostname Intersection Rule

When a request arrives, it is matched against the **intersection** of the Listener's hostname and the Route's hostnames. Only hostnames in the intersection are used for routing traffic.

Listener Hostname	Route Hostnames	Intersection Result	Example
No hostname	No hostnames	Matches all hosts	Any incoming host header is accepted
No hostname	Has hostnames (e.g., `api.example.com`)	All Route hostnames	Only requests with `api.example.com` are matched
Has hostname (e.g., `api.example.com`)	No hostnames	All Listener hostnames	Only requests with `api.example.com` are matched
Has hostname (e.g., `api.example.com`)	Has matching exact hostname	Exact match hostname	Only requests with `api.example.com` are matched

Listener Hostname	Route Hostnames	Intersection Result	Example
Has wildcard (e.g., `*.example.com`)	Has matching hostnames	Matching specific hostnames	Requests with `api.example.com` or `web.example.com` are matched
Has hostname (e.g., `api.example.com`)	Has non-matching hostnames	**No intersection** - Route status is abnormal	Route cannot process traffic

NOTE

Wildcards (`*`) perform suffix matching. For example, `*.example.com` matches `foo.example.com` and `bar.example.com`, but not `example.com`.

WARNING

No intersection means the Route status becomes abnormal and traffic cannot be processed.

Supported Route Kinds

Each listener supports different route kinds based on its protocol:

Listener Protocol	Supported Route Kind
HTTP	HTTPRoute, GRPCRoute
HTTPS	HTTPRoute, GRPCRoute
TLS (`Passthrough` mode)	TLSRoute
TLS (`Terminate` mode)	TCPRoute

Listener Protocol	Supported Route Kind
TCP	TCPRoute
UDP	UDPRoute

When configuring routes, ensure they match the protocol of the listener they attach to. For example, you cannot attach an `HTTPRoute` to a `TCP` listener.

Next Step

After the `Gateway` is ready, continue with Configure GatewayAPI Route. If you need advanced traffic control after routes are attached, continue with Configure GatewayAPI Policy.

Configure GatewayAPI Policy

TOC

Overview

This document explains how to configure policy resources after `Gateway` and `Route` resources are ready. Policies use the Policy Attachment pattern through `.spec.targetRefs` to attach additional traffic, security, and backend behavior to supported resources.

In the recommended workflow, this document comes after Configure GatewayAPI Route.

Envoy Gateway currently provides four policy types: `SecurityPolicy`, `BackendTLSPolicy`, `ClientTrafficPolicy`, and `BackendTrafficPolicy`.

Prerequisites

Please ensure that you have completed the following before proceeding:

1. Read Configure GatewayAPI Gateway and Configure GatewayAPI Route
2. Created the target resource that the policy will attach to, such as a `Gateway`, `Route`, or `Service`

Policy Attachment Basics

Policies attach to other resources through `.spec.targetRefs`.

By default, a policy can only attach to resources in the same namespace.

For `Gateway` targets, `sectionName` can be used to target a specific listener when the policy type supports it. For `Service` targets, `sectionName` refers to the Service port name.

Policy Attachment Summary

Policy Type	Purpose	Web Console Support	Gateway	HTTPRoute	GRP
SecurityPolicy	Authentication, authorization, CORS, and other security features	API Key Auth, CORS	✅ (listener name / `ALL`)	✅	✅
BackendTLSPolicy	TLS configuration between Envoy and backend services	Supported	❌	❌	❌
ClientTrafficPolicy	Client-facing timeout and	Timeout settings	✅ (listener	❌	❌

Policy Type	Purpose	Web Console Support	Gateway	HTTPRoute	GRP
	connection behavior		name / `ALL`)		
BackendTrafficPolicy	Backend timeout and connection behavior	Timeout settings	✅ (listener name / `ALL`)	✅	✅

`sectionName` is used to target a specific listener on a `Gateway` or a specific port on a `Service`. When omitted or set to `ALL`, the policy applies to all listeners or ports.

Create Policies in Web Console

All policy types are created from the same entry:

1. Navigate to `Alauda Container Platform -> Networking -> Gateway -> Policies`
2. Select the required value in the `Policy Type` dropdown
3. Click the `Create Policy` button

The following sections focus only on the fields that are specific to each policy type.

SecurityPolicy

Configuration Via Web Console

Common Fields (shared for all policies):

Field	Description	YAML Path
Policy Type	The type of policy to create	`.kind`

Field	Description	YAML Path
Attach To	The Gateway API resources this policy applies to. Supports Gateway, HTTPRoute, and GRPCRoute. When attaching to Gateway, you can optionally specify a listener name or select ALL listeners.	`.spec.targetRefs`

SecurityPolicy Specific Fields:

Field	Description	YAML Path
Authorization Type	The authentication/authorization method to use. Supports multiple selections: API Key Authentication, CORS Configuration	`.spec.apiKeyAuth`, `.spec.cors`

API Key Authentication

Field	Description	YAML Path
Secrets	Kubernetes secrets containing the API keys for authentication	`.spec.apiKeyAuth.credentialRefs`
Extract From	Specifies where to extract the API key from (HTTP headers or query parameters)	`.spec.apiKeyAuth.extractFrom`

CORS Configuration

Field	Description	YAML Path
Allow Origins	List of allowed origins for CORS requests	`.spec.cors.allowOrigins`
Allow Methods	List of allowed HTTP methods	`.spec.cors.allowMethods`
Allow Headers	List of allowed headers in CORS requests	`.spec.cors.allowHeaders`

Field	Description	YAML Path
Expose Headers	List of headers exposed to client in response	`.spec.cors.exposeHeaders`
Max Age	Cache duration for CORS preflight response	`.spec.cors.maxAge`
Allow Credentials	Whether credentials are allowed in CORS requests	`.spec.cors.allowCredentials`

Configuration Via YAML

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: SecurityPolicy
metadata:
  name: demo-security-policy
  namespace: demo
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: demo
  apiKeyAuth:
    credentialRefs:
      - group: ""
        kind: Secret
        name: demo
        namespace: demo
    extractFrom:
      - headers:
          - authorization
  cors:
    allowOrigins:
      - "https://example.com"
    allowMethods:
      - GET
      - POST
    allowHeaders:
      - "Content-Type"
    exposeHeaders:
      - "X-Custom-Header"
    maxAge: "1h"
    allowCredentials: true
```

Reference

SecurityPolicy is used to configure authentication, authorization, and other security-related features for your Gateway and Routes. It provides a declarative way to protect your services by validating incoming requests before they reach your backend applications.

```
            Gateway            Route
               |                 |
               +-- SecurityPolicy-+
Client ---> Envoy Listener --------------------> Backend Service
```

Features

- **Authentication**: Verify the identity of clients using various methods (API Key, JWT, OIDC, Basic Auth)
- **Authorization**: Control access to resources based on validated credentials
- **CORS Configuration**: Manage Cross-Origin Resource Sharing policies

How It Works

1. Create a SecurityPolicy with your desired authentication/authorization rules
2. Attach it to a specific Gateway, HTTPRoute, or GRPCRoute
3. Envoy Gateway validates incoming requests according to the policy
4. Valid requests are forwarded to backend services; invalid requests are rejected with appropriate HTTP status codes

Notes

1. The web console currently supports configuring **API Key Authentication** and **CORS**. For other authentication methods and advanced security features, you need to use YAML configuration.
2. Each Route can only be associated with one SecurityPolicy.
3. If a SecurityPolicy references a secret with no values, all requests to the attached route will be rejected with `401 Unauthorized`.
4. In the web console, by default, the `Extract From` field is set to `header` and the `Header Name` field is set to `authorization`.
5. You can view which policies are attached to a route by navigating to the Route's topology tab in the web console.

Official Documentation

- SecurityPolicy specification ↗
- API Key Authentication ↗

BackendTLSPolicy

Configuration Via Web Console

Common Fields:

Field	Description	YAML Path
Policy Type	The type of policy to create	`.kind`
Attach To	The Service this policy applies to. You must specify the Service name and port name (`sectionName`).	`.spec.targetRefs`

BackendTLSPolicy Specific Fields:

Field	Description	YAML Path
Hostname	Required. The SNI (Server Name Indication) used when Envoy connects to the backend service	`.spec.validation.hostname`
Subject Alternative Names	Optional. Used for backend HTTPS response validation. If not specified, the hostname value is used as default.	`.spec.validation.subjectAltNames`
Validation Type	The method for validating backend TLS certificates. Options: `CACertificateRefs` (use custom CA certificates), `WellKnownCACertificates` (use system CA certificates)	`.spec.validation`

CACertificateRefs Configuration:

Field	Description	YAML Path
CA Certificate Secret	Kubernetes secret containing the CA certificate. The secret must have a `ca.crt` key containing PEM-encoded TLS certificates.	`.spec.validation.cACertificateRefs`

NOTE

When creating or selecting a CA certificate secret:

- The secret type must be suitable for CA certificates
- The key must be `ca.crt`
- You can import a certificate file, which must start with `-----BEGIN CERTIFICATE-----` and end with `-----END CERTIFICATE-----`
- When importing an invalid certificate format, an error message "must contain PEM-encoded TLS certificates" will be displayed
- When selecting an existing secret without `ca.crt` key, an error message "must have ca.crt key" will be displayed

Configuration Via YAML

```yaml
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: BackendTLSPolicy
metadata:
  name: demo-backend-tls-policy
  namespace: demo
spec:
  targetRefs:
    - group: ""
      kind: Service
      name: demo-backend
      namespace: demo
      sectionName: https-port
  validation:
    hostname: backend.example.com
    subjectAltNames:
      - backend.example.com
    cACertificateRefs:
      - group: ""
        kind: Secret
        name: backend-ca
        namespace: demo
```

Reference

BackendTLSPolicy controls the TLS configuration between Envoy Gateway and backend services. It allows you to configure:

- **SNI (Server Name Indication)**: The hostname used when establishing TLS connections to backends
- **Certificate Validation**: How to validate backend server certificates
- **CA Certificates**: Custom CA certificates for validating backend certificates

Features

- Configure TLS settings for connections to backend services
- Support for custom CA certificates or system well-known CA certificates
- SNI configuration for proper TLS handshake

Notes

1. The `sectionName` in `targetRefs` corresponds to the port name of the Service.
2. When using `WellKnownCACertificates`, the system's default CA certificates are used for validation.
3. The hostname is required and is used as the SNI value when Envoy connects to the backend.

Official Documentation

- BackendTLSPolicy specification ↗

ClientTrafficPolicy

Configuration Via Web Console

Common Fields:

Field	Description	YAML Path
Policy Type	The type of policy to create	`.kind`
Attach To	The Gateway this policy applies to. You can optionally specify a listener name or select ALL listeners.	`.spec.targetRefs`

Timeout Configuration (Options):

Field	Description	YAML Path
TCP Idle Timeout	The idle timeout for a TCP connection. Idle time is defined as a period in which there are no bytes sent or received on either the upstream or downstream connection. Default: 1 hour.	`.spec.settings.timeout.tcp.idleTimeout`
HTTP Request Received Timeout	The duration Envoy waits for the complete request reception. This timer starts upon request initiation and stops when either the last byte of the request is sent upstream or when the response begins. Default: 1 hour.	`.spec.settings.timeout.http.requestReceivedTimeout`

Field	Description	YAML Path
HTTP Idle Timeout	The idle timeout for an HTTP connection. Idle time is defined as a period in which there are no active requests in the connection. Default: unlimited.	`.spec.settings.timeout.http.idleTimeout`
HTTP Stream Idle Timeout	The stream idle timeout defines the amount of time a stream can exist without any upstream or downstream activity. Default: 5 minutes.	`.spec.settings.timeout.http.streamIdleTimeout`

Configuration Via YAML

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: ClientTrafficPolicy
metadata:
  name: demo-client-traffic-policy
  namespace: demo
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      sectionName: https
  settings:
    timeout:
      tcp:
        idleTimeout: "30m"
      http:
        requestReceivedTimeout: "60s"
        idleTimeout: "5m"
        streamIdleTimeout: "30s"
```

Reference

ClientTrafficPolicy controls the behavior of connections from clients to Envoy Gateway. It provides fine-grained control over:

```
            Gateway
               |
               +-- ClientTrafficPolicy
Client ---> Envoy Listener --------------------> Backend Service
          applies on the client-facing side ^
```

- **TCP Settings**: Connection-level timeout and keepalive settings
- **HTTP Settings**: Request/response timeouts and HTTP protocol behavior

Features

- Configure TCP connection idle timeouts
- Control HTTP request reception timeouts

- Set HTTP connection idle timeouts
- Configure HTTP stream idle timeouts

Notes

1. Timeout values are specified as duration strings (e.g., "30s", "5m", "1h").

Official Documentation

- ClientTrafficPolicy specification ↗

BackendTrafficPolicy

Configuration Via Web Console

Common Fields:

Field	Description	YAML Path
Policy Type	The type of policy to create	`.kind`
Attach To	The Gateway API resources this policy applies to. Supports Gateway, HTTPRoute, GRPCRoute, TCPRoute, UDPRoute, and TLSRoute. When attaching to Gateway, you can optionally specify a listener name or select `ALL` listeners. In YAML, this is configured through `sectionName` in `.spec.targetRefs`.	`.spec.targetRefs`

Timeout Configuration (Options):

Field	Description	YAML Path
TCP Connection Timeout	The timeout for network connection establishment,	`.spec.settings.timeout.tcp.connectionTimeout`

Field	Description	YAML Path
	including TCP and TLS handshakes. Default: 10 seconds.	
HTTP Connection Idle Timeout	The idle timeout for an HTTP connection. Idle time is defined as a period in which there are no active requests in the connection. Default: 1 hour.	`.spec.settings.timeout.http.connectionIdleTimeout`
HTTP Max Connection Duration	The maximum duration of an HTTP connection. Default: unlimited.	`.spec.settings.timeout.http.maxConnectionDuration`
HTTP Request Timeout	The time until the entire response is received from the upstream. Default: 15 seconds. Supports setting to unlimited.	`.spec.settings.timeout.http.requestTimeout`

Configuration Via YAML

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: BackendTrafficPolicy
metadata:
  name: demo-backend-traffic-policy
  namespace: demo
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: demo
  settings:
    timeout:
      tcp:
        connectionTimeout: "5s"
      http:
        connectionIdleTimeout: "30m"
        maxConnectionDuration: "1h"
        requestTimeout: "30s"
```

Reference

BackendTrafficPolicy controls the behavior of connections from Envoy Gateway to backend services. It provides fine-grained control over:

```
            Gateway / Route
                  |
                  +-- BackendTrafficPolicy
Client ---> Envoy Listener --------------------> Backend Service
                                  applies here ^
```

- **TCP Settings**: Connection establishment timeouts
- **HTTP Settings**: Connection durations, idle timeouts, and request timeouts

Features

- Configure TCP connection establishment timeouts
- Control HTTP connection idle timeouts
- Set maximum HTTP connection durations

- Configure HTTP request timeouts

Notes

1. Timeout values are specified as duration strings (e.g., "30s", "5m", "1h").
2. The `requestTimeout` field supports setting to "unlimited" to disable the timeout.

Official Documentation

- BackendTrafficPolicy specification ↗

Related Tasks

After policies are attached, continue with Tasks for Envoy Gateway for more operational examples and advanced configuration tasks.

Configure GatewayAPI Route

TOC

Related Tasks

Overview

This document explains how to configure `Route` resources after a `Gateway` is ready. A `Route` attaches to one or more gateway listeners and defines how matching traffic is forwarded to backend services.

In the recommended workflow, this document comes after Configure GatewayAPI Gateway and before Configure GatewayAPI Policy.

In addition to create and update operations, this document also introduces the extra route viewing capabilities provided by the ACP Web Console.

Prerequisites

Please ensure that you have completed the following before proceeding:

1. Read Configure GatewayAPI Gateway to understand listeners, attachment rules, and `EnvoyProxy`
2. Create a `Gateway` that your `Route` will attach to

> **NOTE**
>
> This document first introduces each route type separately, then provides YAML examples, and finally explains the common route concepts in a shared reference section.

Configuration

A `Route` attaches to one or more listeners on a `Gateway`. The listener you can select depends on the route type, the listener protocol, and the listener's allowed route namespace settings.

Configuration Via Web Console

1. Navigate to `Alauda Container Platform -> Networking -> Gateway -> Routes`
2. Click the `Create Route` button
3. Select the route type (HTTPRoute, TCPRoute, UDPRoute, GRPCRoute, or TLSRoute)

Create HTTPRoute

Field	Description	YAML Path
Publish to Listener	publish to listener	`.spec.parentRefs`
Hostnames	hostnames	`.spec.hostnames`
Matches	matches	`.spec.rules[].matches`
Filters	filters	`.spec.rules[].filters`
Backend Instance	backend	`.spec.rules[].backendRefs`
Options	options	`.spec.rules[].filters`, `.spec.rules[].timeouts`, `.spec.rules[].retry`, `.spec.rules[].sessionPersistence`

Options Configuration

The Options field allows you to configure advanced traffic management settings:

Option	Description	YAML Path
Session Affinity	session persistence	`.spec.rules[].sessionPersistence`
Timeout	timeout settings	`.spec.rules[].timeouts`
Retry	retry policy	`.spec.rules[].retry`

Create TCP/UDP Route

Field	Description	YAML Path
Publish to Listener	publish to listener	`.spec.parentRefs`
Backend Instance	backend	`.spec.rules[].backendRefs`

Create GRPCRoute

Field	Description	YAML Path
Publish to Listener	publish to listener	`.spec.parentRefs`
Hostnames	hostnames	`.spec.hostnames`
Matches	grpc matches	`.spec.rules[].matches`
Filters	grpc filters	`.spec.rules[].filters`
Backend Instance	backend	`.spec.rules[].backendRefs`

Create TLSRoute

Field	Description	YAML Path
Publish to Listener	publish to listener	`.spec.parentRefs`
Hostnames	hostnames (optional)	`.spec.hostnames`
Backend Instance	backend	`.spec.rules[].backendRefs`

Configuration Via YAML

The following minimal example creates an `HTTPRoute` that attaches to the `https` listener of the `demo` `Gateway` and forwards matching traffic to a backend `Service`.

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: demo-443
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      sectionName: https
  rules:
    - matches:
        - path:
            type: Exact
            value: /a
      backendRefs:
        - group: ''
          kind: Service
          name: echo-resty
          namespace: demo-space
          port: 80
          weight: 100
```

If you need more route types and advanced HTTPRoute options, use the following complete example:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: demo-443
  namespace: demo
spec:
  hostnames: 1
    - example.com
  parentRefs: 2
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      sectionName: https
  rules: 3
    - backendRefs: 4
        - group: ''
          kind: Service
          name: echo-resty
          namespace: demo-space
          port: 80
          weight: 100
      filters: [] 5
      matches: 6
        - path:
            type: Exact
            value: /a
      timeouts: 7
        request: '30s'
        backendRequest: '10s'
      retry: 8
        codes:
          - 503
        attempts: 3
        backoff: '100ms'
      sessionPersistence: 9
        type: Cookie
        sessionName: a
---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TCPRoute
metadata:
  name: tcp
  namespace: demo-space
```

```
spec:
  parentRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      sectionName: tcp
  rules:
    - backendRefs:
        - group: ''
          kind: Service
          name: echo-resty
          port: 80
          weight: 100
---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: UDPRoute
metadata:
  name: udp
  namespace: demo
spec:
  parentRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      namespace: demo
      sectionName: udp
  rules:
    - backendRefs:
        - group: ''
          kind: Service
          name: echo-resty
          namespace: demo
          port: 53
          weight: 100
---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: GRPCRoute
metadata:
  name: grpc
  namespace: demo
spec:
  hostnames:
    - grpc.example.com
  parentRefs:
```

```
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      sectionName: https
  rules:
    - matches:
        - method:
            type: service
            value: myservice
      filters:
        - type: RequestHeaderModifier
          requestHeaderModifier:
            set:
              - name: x-custom-header
                value: custom-value
      backendRefs:
        - group: ''
          kind: Service
          name: grpc-service
          port: 50051
---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TLSRoute
metadata:
  name: tls
  namespace: demo
spec:
  hostnames:
    - tls.example.com
  parentRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      sectionName: tls
  rules:
    - backendRefs:
        - group: ''
          kind: Service
          name: tls-backend
          port: 443
```

1 Hostnames

2 Publish to listener

3. Rules
4. Backend
5. Filters
6. Matches
7. Options
8. Timeout
9. Retry
10. Session Persistence

Route Field Reference

Each route is a CR defined by the `GatewayAPI` specification. For detailed information about the fields and configuration options for each route type, please refer to the official documentation:

- HTTPRoute specification ↗
- TCPRoute specification ↗
- UDPRoute specification ↗
- GRPCRoute specification ↗
- TLSRoute specification ↗

Publish to Listener

In Web Console

In the web console, you can select multiple listeners to publish the route to. The available listener candidates are filtered based on the following criteria:

- **User permissions**: You must have access to the gateway's namespace (the project must include this namespace).
- **Route namespace allowlist**: The gateway listener's allowed route namespaces must include the route's namespace.
- **Route kind matching**: The route's kind (HTTPRoute, GRPCRoute, etc.) must match the listener's allowed route kinds.

For more complex cross-namespace scenarios, please refer to attaching to a gateway created in another namespace.

In YAML

- The `sectionName` is the listener name.
- Routes can only be attached to listeners that support their specific kind.
- By default, routes can only be attached to listeners where the `Gateway` is in the same namespace.

For cross-namespace attachment, please refer to attaching to a gateway created in another namespace.

Backend

Defines the target service(s) where matching requests should be forwarded.

Each service can have a `weight` field to specify the proportion of traffic to be routed to that service.

Hostnames

The `hostnames` field is supported by `HTTPRoute`, `GRPCRoute`, and `TLSRoute`. `TCPRoute` and `UDPRoute` do not use this field.

`hostnames` is a string array. It follows the Hostname Intersection Rules.

Rules

Each route can contain multiple rules. Each rule consists of the following components:

Matches

Defines the conditions that must be met for a request to be routed by this rule.

A rule can have multiple matches:

- Each match consists of multiple conditions (e.g., path, headers, query parameters, method)
- Conditions **within** a match use **AND** logic (all must be satisfied)

- Matches **between** each other use **OR** logic (any match can satisfy the rule)

Example: If Match-1 requires `path=/api` AND `header=v1`, and Match-2 requires `query=test`, then a request is routed if it matches either `(path=/api AND header=v1)` OR `(query=test)`.

The match structure is common across route types, but the supported match conditions depend on the route type. For example, `HTTPRoute` and `GRPCRoute` support different match condition sets.

Filters

Specifies transformations or modifications to apply to requests or responses.

The filter concept is common across route types, but the supported filter types depend on the route type.

HTTPRoute Reference

The following match conditions, filter types, and advanced options are used by `HTTPRoute`.

Match Condition Types

Object	Method	Value Types	Description	Value Requirements
Path				
	`Exact`	path (string)	Matches the URL path exactly and with case sensitivity. This means that an exact path match on /abc will only match requests to /abc, **NOT**	Must start with `/`, no consecutive `//`.

Object	Method	Value Types	Description	Value Requirements
			/abc/, /Abc, or /abcd.	
	`PathPrefix`	path (string)	Matches based on a URL path prefix split by /. Matching is case-sensitive and done on **a path element by element basis**. For example, the paths /abc, /abc/, and /abc/def would all match the prefix /abc, but the path /abcd would not.	Must start with `/`, no consecutive `//`.
	`RegularExpression`	path (string)	Regex Engine: RE2.	for example, `/api/v1/.*`.
Header				
	`Exact`	name (header key) + value	Exact header value match.	
	`RegularExpression`	name (header key) + value	Regex Engine: RE2.	

Object	Method	Value Types	Description	Value Requirements
QueryParam				
	`Exact`	name (param key) + value	Exact query parameter value match.	Parameter value: 1-1024 characters
	`RegularExpression`	name (param key) + value	Regex Engine: RE2.	
Method	-	method name	HTTP method match.	`GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `CONNECT`, `OPTIONS`, `TRACE`, `PATCH`

Match Condition References

Condition Type	Official Documentation
Path	HTTPPathMatch ↗
Headers	HTTPHeaderMatch ↗
QueryParams	HTTPQueryParamMatch ↗
Method	HTTPMethod ↗

Filter Types

Type	Method	Value Types	Description
RequestHeaderModifier	`Set`	name (string) + value (string)	Overwrites request header with given name and value
	`Add`	name (string) + value (string)	Adds header to request, appending to existing values
	`Remove`	[]string	Removes specified headers from request (case-insensitive)
ResponseHeaderModifier	`Set`	name (string) + value (string)	Overwrites response header with given name and value
	`Add`	name (string) + value (string)	Adds header to response, appending to existing values
	`Remove`	[]string	Removes specified headers from response

Type	Method	Value Types	Description
			(case-insensitive)
RequestRedirect	`Scheme`	string	Scheme for Location header (http/https)
	`Hostname`	PreciseHostname	Hostname for Location header
	`ReplaceFullPath`	string	Replace entire request path
	`ReplacePrefixMatch`	string	Replace matched path prefix
	`Port`	PortNumber	Port for Location header
	`StatusCode`	int	HTTP redirect status code
URLRewrite	`Hostname`	PreciseHostname	Hostname to rewrite in request

Type	Method	Value Types	Description
	`ReplaceFullPath`	string	Replace entire request path
	`ReplacePrefixMatch`	string	Replace matched path prefix
CORS	`AllowOrigins`	[]string	List of allowed origins for CORS requests
	`AllowMethods`	[]HTTPMethod	List of allowed HTTP methods
	`AllowHeaders`	[]string	List of allowed headers in CORS requests
	`ExposeHeaders`	[]string	List of headers exposed to client in response
	`MaxAge`	Duration	Cache duration for CORS preflight response

Type	Method	Value Types	Description
	`AllowCredentials`	bool	Whether credentials are allowed in CORS requests

Notes:

- `RequestRedirect` and `URLRewrite` cannot be used together on the same rule
- `ReplacePrefixMatch` is only compatible with a `PathPrefix` HTTPRouteMatch
- Header names are case-insensitive per RFC 7230
- Multiple values for same header must use RFC 7230 comma-separated format

Filter References

Filter Type	Official Documentation
RequestHeaderModifier	HTTPHeaderFilter ↗
ResponseHeaderModifier	HTTPHeaderFilter ↗
RequestRedirect	HTTPRequestRedirectFilter ↗
URLRewrite	HTTPURLRewriteFilter ↗
CORS	HTTPCORSFilter ↗
RequestMirror	HTTPRequestMirrorFilter ↗
HTTPExternalAuthFilter	HTTPExternalAuthFilter ↗

Options

The Options section provides advanced traffic management capabilities for `HTTPRoute`, including timeout, retry, and session persistence settings.

Timeouts

Field	Description	YAML Path
Request Timeout	The maximum duration for the gateway to complete an HTTP response after receiving the full request from the client. Options: Default (uses default timeout, typically 15 seconds), Unlimited (sets to "0s" to remove timeout), Custom.	`.spec.rules[].timeouts.request`
Backend Request Timeout	The maximum duration for a single gateway-to-backend call, from when the gateway starts sending the request to when the full backend response is received. Options: Default (uses implementation-specific default), Unlimited (sets to "0s"), Custom.	`.spec.rules[].timeouts.backendRequest`

NOTE

- The Request Timeout starts counting after the entire client request has been received and covers the complete transaction, which may include multiple backend calls if retries occur.
- Backend Request Timeout must be ≤ Request Timeout when specified.
- When selecting "Default", the field is set to nil (uses implementation default).
- When selecting "Unlimited", the field is set to "0s" (maximum possible value).

Field	Specification
`.spec.rules[].timeouts`	HTTPRouteTimeouts ↗

Retry

Field	Description	YAML Path
Status Codes	HTTP status codes that trigger retry (e.g., 503, 502). Value range: 400-599.	`.spec.rules[].retry.codes`
Attempts	Number of retry attempts.	`.spec.rules[].retry.attempts`
Backoff	Wait time before retry (e.g., "100ms", "1s").	`.spec.rules[].retry.backoff`

NOTE

- By default, retries are **disabled**. If the retry field is not configured or is left empty, the gateway will NOT retry any failed requests.
- You must explicitly configure both retry attempts and retry conditions to enable retry functionality.
- When configuring retry in the web console, if you remove all retry configuration items, the field is set to nil.

Field	Specification
`.spec.rules[].retry`	HTTPRouteRetry ↗

Session Persistence

Configures session affinity settings to ensure requests from the same client are routed to the same backend.

Field	Description	YAML Path
Type	Session persistence type. Options: Cookie, Header.	`.spec.rules[].sessionPersistence.type`

Field	Description	YAML Path
Session Name	The name of the cookie or header used for session tracking.	`.spec.rules[].sessionPersistence.sessionName`

Field	Specification
`.spec.rules[].sessionPersistence`	SessionPersistence ↗

GRPCRoute Match and Filter Reference

The following match conditions and filter types are used by `GRPCRoute`.

GRPCRoute Matches

`GRPCRoute` supports the following match types:

Object	Method	Value Types	Description
Method	-	type (service/method) + value	Matches gRPC method. Type can be `service` (matches service name) or `method` (matches method name).
Headers	`Exact`	name (header key) + value	Exact header value match.
	`RegularExpression`	name (header key) + value	Regex Engine: RE2.

GRPCRoute Filters

`GRPCRoute` only supports the `RequestHeaderModifier` filter:

Type	Method	Value Types	Description	Value Requirements
RequestHeaderModifier	`Set`	name (string) + value (string)	Overwrites request header with given name and value	Max 16 items, value: 1-4096 chars
	`Add`	name (string) + value (string)	Adds header to request, appending to existing values	Max 16 items, value: 1-4096 chars
	`Remove`	[]string	Removes specified headers from request (case-insensitive)	Max 16 items

NOTE

`GRPCRoute` does not support Options such as timeout, retry, or session persistence.

TLSRoute Reference

The following behavior is specific to `TLSRoute`.

NOTE

- `TLSRoute` hostnames are optional. If the listener has a hostname but the `TLSRoute` does not, the `TLSRoute` automatically inherits the listener's hostname.
- `TLSRoute` can only attach to listeners with `TLS` protocol in `Passthrough` mode.

View

Topology

The following features are additional viewing capabilities provided by the ACP Web Console.

The topology tab provides a visual representation of the route and its associated resources. It displays all policies attached to the route, along with their dependent resources such as secrets referenced by `SecurityPolicy`.

This feature is currently only available for `HTTPRoute`.

Next Step

After routes are attached to listeners, continue with Configure GatewayAPI Policy if you need advanced traffic or security policies. For additional operational examples, see Tasks for Envoy Gateway.

Related Tasks

- How to attach to listener created in other namespace

Configure ALB

WARNING

ALB has been deprecated. Please use the ingress-nginx-operator or envoy-gateway instead.

TOC

ALB

ALB is a custom resource that represents a load balancer. The alb-operator, which is embedded by default in all clusters, watches for create/update/delete operations on ALB resources and creates corresponding deployments and services in response.

For each ALB, a corresponding Deployment watches all Frontends and Rules attached to that ALB and routes requests to backends based on those configurations.

Prerequisites

The high availability of the **Load Balancer** requires a VIP. Please refer to Configure VIP.

Configure ALB

There are three parts to an ALB configuration.

```yaml
# test-alb.yaml
apiVersion: crd.alauda.io/v2beta1
kind: ALB2
metadata:
  name: alb-demo
  namespace: cpaas-system
spec:
  address: 192.168.66.215
  config:
    vip:
      enableLbSvc: false
      lbSvcAnnotations: {}
    networkMode: host
    nodeSelector:
      cpu-model.node.kubevirt.io/Nehalem: 'true'
    replicas: 1
    resources:
      alb:
        limits:
          cpu: 200m
          memory: 256Mi
        requests:
          cpu: 200m
          memory: 256Mi
      limits:
        cpu: 200m
        memory: 256Mi
      requests:
        cpu: 200m
        memory: 256Mi
    projects:
      - ALL_ALL
  type: nginx
```

Resource-Related Configuration

resource related field describes the deployment configuration for the alb.

Field	Type	Description
`.spec.config.nodeSelector`	map[string]string	the node selector for the alb
`.spec.config.replicas`	int,optional default 3	the number of replicas for the alb
`.spec.config.resources.limits`	k8s container-resource,optional	limit of nginx container of alb
`.spec.config.resources.requests`	k8s container-resource,optional	request of nginx container of alb
`.spec.config.resources.alb.limits`	k8s container-resource,optional	limit of alb container of alb
`.spec.config.resources.alb.requests`	k8s container-resource,optional	request of alb container of alb
`.spec.config.antiAffinityKey`	string,optional default local	k8s antiAffinityKey

Networking Configuration

Networking fields describe how to access the ALB. For example, in `host` mode, alb will use hostnetwork, and you can access the ALB via the node IP.

Field	Type	Description
`.spec.config.networkMode`	string: `host` or `container`, optional, default `host`	In `container` mode, the operator creates a LoadBalancer Service and uses its address as the ALB address.
`.spec.address`	string,required	you could manually specify the address of alb

Field	Type	Description
`.spec.config.vip.enableLbSvc`	bool, optional	Automatically true in `container` mode.
`.spec.config.vip.lbSvcAnnotations`	map[string]string, optional	Extra annotations for the LoadBalancer Service.

project configuration

Field	Type
`.spec.config.projects`	[]string,required
`.spec.config.portProjects`	string,optional
`.spec.config.enablePortProject`	bool,optional

Adding an ALB to a project means:

1. In the web UI, only users in the given project can find and configure this ALB.
2. This ALB will handle ingress resources belonging to this project. Please refer to ingress-sync.
3. In the web UI, rules created in project X cannot be found or configured under project Y.

If you enable port project and assign a port range to a project, this means:

1. You cannot create ports that do not belong to the port range assigned to the project.

tweak configuration

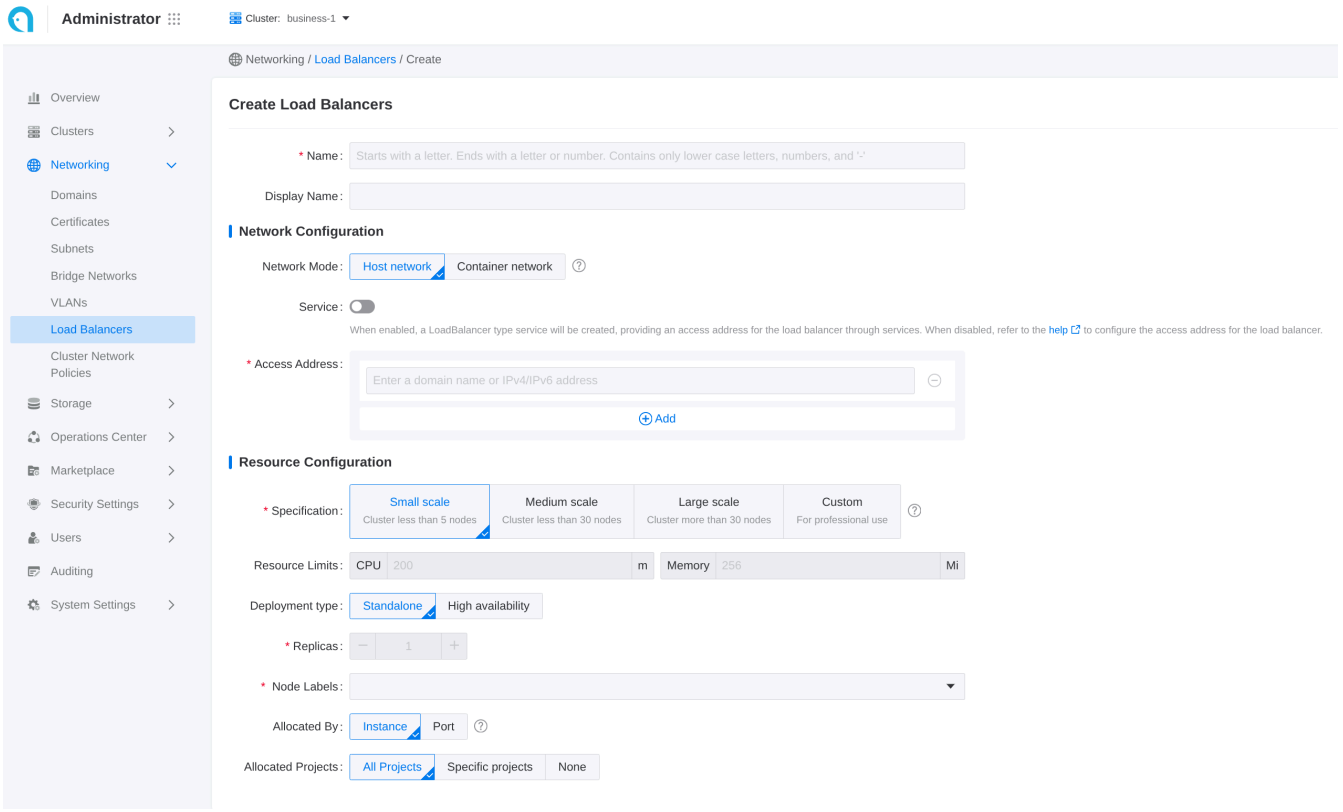
there are some global config which can be tweaked in alb cr.

- Bind NIC
- Ingress sync

Operation On ALB

Creating

Using the web console.



Some common configuration is exposed in the web UI. Follow these steps to create a load balancer:

1. Navigate to **Administrator**.
2. In the left sidebar, click on **Network Management** > **Load Balancer**.
3. Click on **Create Load Balancer**.

Each input item in the web UI corresponds to a field of the CR:

Parameter	Description
Assigned Address	`.spec.address`
Allocated By	`Instance` means project mode, and you could select project below, port means port-project mode, you could assign port-range after create alb

Using the CLI.

```
kubectl apply -f test-alb.yaml -n cpaas-system
```

Update

Using the web console

> **NOTE**
>
> Updating the load balancer will cause a service interruption for 3 to 5 minutes. Please choose an appropriate time for this operation!

1. Enter **Administrator**.
2. In the left navigation bar, click **Network Management** > **Load Balancer**.
3. Click ⋮ > **Update**.
4. Update the network and resource configuration as needed.
   - Please set specifications reasonably according to business needs. You can also refer to the relevant How to properly allocate CPU and memory resources for guidance.
   - **Internal routing** only supports updating from **Disabled** state to **Enabled** state.
5. Click **Update**.

Delete

Using the web console

> **NOTE**
>
> After deleting the load balancer, the associated ports and rules will also be deleted and cannot be restored.

1. Enter **Administrator**.
2. In the left navigation bar, click **Network Management** > **Load Balancer**.
3. Click ⋮ > **Delete**, and confirm.

Using the CLI

```
kubectl delete alb2 alb-demo -n cpaas-system
```

Frontend

Frontend is a custom resource that defines the listener port and protocol for an ALB. Supported protocols: L7 (http|https|grpc|grpcs) and L4 (tcp|udp).

- In L4 Proxy use frontend to configure backend service directly.
- In L7 Proxy use frontend to configure listener ports, and use rule to configure backend service.

If you need to add an HTTPS listener port, you should also contact the administrator to assign a TLS certificate to the current project for encryption.

Prerequisites

Create a ALB first.

Configure Frontend

```yaml
# alb-frontend-demo.yaml
apiVersion: crd.alauda.io/v1
kind: Frontend
metadata:
  labels:
    alb2.cpaas.io/name: alb-demo (1)
  name: alb-demo-00080 (2)
  namespace: cpaas-system
spec:
  port: 80 (3)
  protocol: http (4)
  certificate_name: '' (5)
  backendProtocol: 'http' (6)
  serviceGroup: (7)
    session_affinity_policy: '' (8)
    services:
      - name: hello-world
        namespace: default
        port: 80
        weight: 100
```

(1) alb label: Required, indicate the ALB instance to which this `Frontend` belongs to.

(2) frontend name: Format as `$alb_name-$port`.

(3) port: which port which listen on.

(4) protocol: what protocol this port uses.

- L7 protocol https|http|grpcs|grpc and L4 protocol tcp|udp.
- When selecting HTTPS, a certificate must be added; adding a certificate is optional for the gRPC protocol.
- When selecting the gRPC protocol, the backend protocol defaults to gRPC, which does not support session persistence.If a certificate is set for the gRPC protocol, the load balancer will unload the gRPC certificate and forward the unencrypted gRPC traffic to the backend service.
- If using a Google GKE cluster, a load balancer of the same **container network type** cannot have both TCP and UDP listener protocols simultaneously.

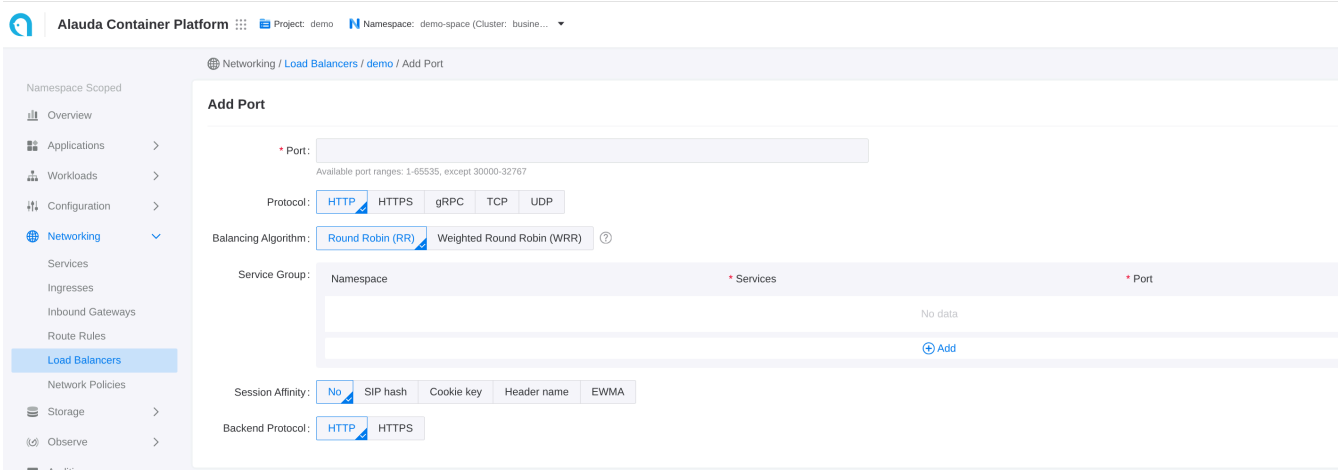
(5) certificate_name: for grpcs and https protocol which the default cert used, Format as `$secret_ns/$secret_name`.

6. backendProtocol: what protocol the backend service uses.
7. Default `serviceGroup`:
   - L4 proxy: required. ALB forwards traffic to the default service group directly.
   - L7 proxy: optional. ALB first matches Rules on this Frontend; if none match, it falls back to the default `serviceGroup`.
8. session_affinity_policy

Operation On Frontend

Creating

using the web console



1. Go to **Container Platform**.
2. In the left navigation bar, click **Network** > **Load Balancing**.
3. Click the name of the load balancer to enter the details page.
4. Click **Add Port**.

Each input item on the web UI corresponds to a field of the CR

Parameter	Description
Session Affinity	`.spec.serviceGroup.session_affinity_policy`

using the CLI

```
kubectl apply -f alb-frontend-demo.yaml -n cpaas-system
```

Subsequent Actions

For traffic from HTTP, gRPC, and HTTPS ports, in addition to the default internal routing group, you can set more varied back-end service matching rules. The load balancer will initially match the corresponding backend service according to the set rules; if the rule match fails, it will then match the backend services corresponding to the aforementioned internal routing group.

Related Operations

You can click the ⋮ icon on the right side of the list page or click **Actions** in the upper right corner of the details page to update the default route or delete the listener port as needed.

> **NOTE**
>
> If the resource allocation method of the load balancer is **Port**, only administrators can delete the related listener ports in the **Administrator** view.

Rule

Rule is a Custom Resource(CR) that defines how incoming requests are matched and processed by the ALB.

Ingresses handled by ALB can be auto-translated to rules.

Prerequisites

- Configure ALB
- Configure Frontend

Here is a demo rule to give you a quick first impression of how to use rules.

NOTE

rule must be attached to a frontend and alb via label.

```yaml
apiVersion: crd.alauda.io/v1
kind: Rule
metadata:
  labels:
    alb2.cpaas.io/frontend: alb-demo-00080 (1)
    alb2.cpaas.io/name: alb-demo (2)
  name: alb-demo-00080-test
  namespace: cpaas-system
spec:
  backendProtocol: 'https' (3)
  certificate_name: 'a/b' (4)
  dslx: (5)
    - type: URL
      values:
        - - STARTS_WITH
          - /
  priority: 4 (6)
  serviceGroup: (7)
    services:
      - name: hello-world
        namespace: default
        port: 80
        weight: 100
```

(1) Required, indicate the `Frontend` to which this rule belongs.

(2) Required, indicate the ALB to which this rule belongs.

(3) backendProtocol

(4) certificate_name

(5) dslx

(6) The lower the number, the higher the priority.

(7) serviceGroup

match request with dslx and priority

dslx

DSLX is a domain-specific language used to describe the matching criteria. For example, the rule below matches a request that satisfies **all** the following criteria:

- url starts with /app-a **or** /app-b
- method is post
- url param's group is vip
- host is *.app.com
- header's location is east-1 or east-2
- has a cookie name is uid
- source IPs come from 1.1.1.1-1.1.1.100

```
dslx:
  - type: METHOD
    values:
      - - EQ
        - POST
  - type: URL
    values:
      - - STARTS_WITH
        - /app-a
      - - STARTS_WITH
        - /app-b
  - type: PARAM
    key: group
    values:
      - - EQ
        - vip
  - type: HOST
    values:
      - - ENDS_WITH
        - .app.com
  - type: HEADER
    key: LOCATION
    values:
      - - IN
        - east-1
        - east-2
  - type: COOKIE
    key: uid
    values:
      - - EXIST
  - type: SRC_IP
    values:
      - - RANGE
        - '1.1.1.1'
        - '1.1.1.100'
```

priority

Priority is an integer ranging from 0 to 10, where lower values indicate higher priority. To configure the priority of a rule in ingress, you can use the following annotation format:

```
# alb.cpaas.io/ingress-rule-priority-$rule_index-$path_index
alb.cpaas.io/ingress-rule-priority-0-0: '10'
```

For rules, simply set the priority directly in `.spec.priority` using an integer value.

Action

After a request matches a rule, you can apply the following actions to the request

Feature	Description	Link
Timeout	Configures the timeout settings for requests.	timeout
Redirect	Redirects incoming requests to a specified URL.	redirect
CORS	Enables Cross-Origin Resource Sharing (CORS) for the application.	cors
Header Modification	Allows modification of request or response headers.	header modification
URL Rewrite	Rewrites the URL of incoming requests before forwarding them.	url-rewrite
WAF	Integrates Web Application Firewall (WAF) for enhanced security.	waf
OTEL	Enables OpenTelemetry (OTEL) for distributed tracing and monitoring.	otel
Keepalive	Enables or disables the keepalive feature for the application.	keepalive

Backend

backend protocol

By default, the backend protocol is set to HTTP. If you want to use TLS re-encryption, you can configure it as HTTPS.

Service Group and Session Affinity Policy

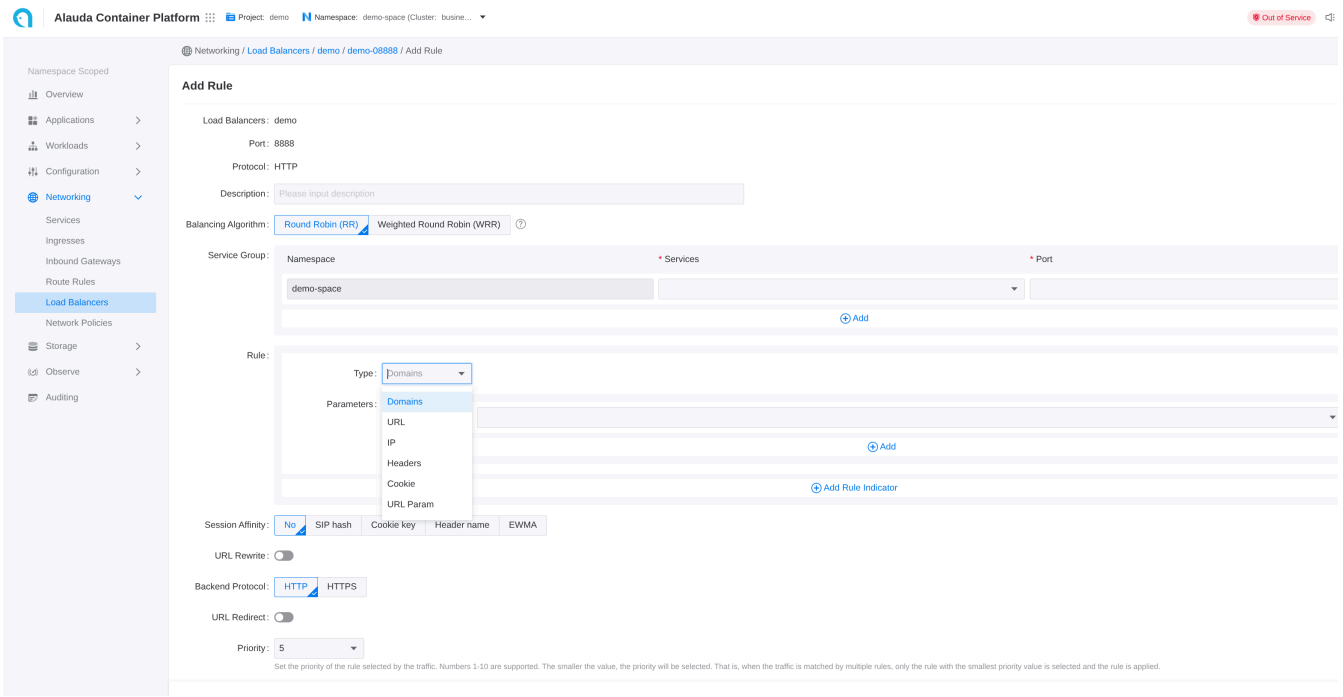
You can configure one or more services within a rule.

By default, the ALB uses a round-robin (RR) algorithm to distribute requests among backend services. However, you can assign weights to individual services or choose a different load-balancing algorithm.

For more details, refer to Balance Algorithm.

Operation On Rule

Using web console



1. Go to **Container Platform**.
2. Click on **Network** > **Load Balancing** in the left navigation bar.
3. Click on the name of the load balancer.
4. Select the listener port name.
5. Click **Add Rule**.
6. Refer to the following descriptions to configure the relevant parameters.

7. Click **Add**.

Each input item on the webui corresponds to a field of the CR

using the CLI

```
kubectl apply -f alb-rule-demo.yaml -n cpaas-system
```

Https

If the frontend protocol (ft) is HTTPS or GRPCS, the rule can also be configured to use HTTPS.

You can specify the certificate either in the rule or in the ingress to match the certificate for that specific port.

Termination is supported, and re-encryption is possible if the backend protocol is HTTPS. However, you **cannot** specify a certificate for communication with the backend service.

Certificate Annotation in Ingress

Certificates can be referenced across namespaces via annotation.

```
alb.networking.cpaas.io/tls: qq.com=cpaas-system/dex.tls,qq1.com=cpaas-sy
stem/dex1.tls
```

Certificate in Rule

In `.spec.certificate_name`, the format is `$secret_namespace/$secret_name`

TLS Mode

Edge Mode

In edge mode, the client communicates with the ALB using HTTPS, and ALB communicates with backend services using HTTP protocol. To achieve this:

1. create ft use https protocol
2. create rule with backend protocol http, and specify cert via `.spec.certificate_name`

Re-encrypt Mode

In re-encrypt mode, the client communicates with the ALB using HTTPS, and ALB communicates with backend services using HTTPS protocol. To achieve this:

1. create ft use https protocol
2. create rule with backend protocol https, and specify cert via `.spec.certificate_name`

Ingress

Ingress sync

Each ALB creates an IngressClass with the same name and handles ingresses within the same project.

When an ingress namespace has a label like `cpaas.io/project: demo`, it indicates that the ingress belongs to the `demo` project.

ALBs that have the project name `demo` in their `.spec.config.projects` configuration will automatically translate these ingresses into rules.

> **NOTE**
>
> ALB listens to ingress and automatically creates a `Frontend` or Rule. `source` field is defined as follows:
>
> 1. `spec.source.type` currently only supports `ingress`.
> 2. `spec.source.name` is ingress name.
> 3. `spec.source.namespace` is ingress namespace.

SSL strategy

For ingresses that do not have certificates configured, ALB provides a strategy to use a default certificate.

You can configure the ALB custom resource with the following settings:

- `.spec.config.defaultSSLStrategy`: Defines the SSL strategy for ingresses without certificates
- `.spec.config.defaultSSLCert`: Sets the default certificate in the format `$secret_ns/$secret_name`

Available SSL strategies:

- **Never**: Do not create rules on HTTPS ports (default behavior)
- **Always**: Create rules on HTTPS ports using the default certificate

Logs and Monitoring

By combining logs and monitoring data, you can quickly identify and resolve load balancer issues.

Viewing Logs

1. Go to **Administrator**.
2. In the left navigation bar, click on **Network Management** > **Load Balancer**.
3. Click on ***Load Balancer Name***.
4. In the **Logs** tab, view the logs of the load balancer's runtime from the container's perspective.

Monitoring Metrics

> **NOTE**
>
> The cluster where the load balancer is located must deploy monitoring services.

1. Go to **Administrator**.
2. In the left navigation bar, click on **Network Management** > **Load Balancer**.
3. Click on ***Load Balancer Name***.
4. In the **Monitoring** tab, view the metric trend information of the load balancer from the node's perspective.

   - **Usage Rate**: The real-time usage of CPU and memory by the load balancer on the current node.
   - **Throughput**: The overall incoming and outgoing traffic of the load balancer instance.

For more detailed information about monitoring metrics please refer to ALB Monitoring.

Configure NodeLocal DNSCache

TOC

Overview

NodeLocal DNSCache is a cluster plugin that improves cluster DNS performance by running a DNS caching proxy on cluster nodes. This plugin reduces DNS query latency and improves cluster stability by caching DNS responses locally on each node, minimizing the load on the central DNS service.

Key Features

- **Local DNS Caching**: Caches DNS responses locally on each node to reduce query latency
- **Improved Performance**: Significantly reduces DNS lookup times for applications

Important Notes

WARNING

Deployment Considerations:

1. **Kube-OVN Underlay Mode**: The plugin does not support deployment in Kube-OVN Underlay mode. If deployed, it may cause DNS query failures.
2. **Kubelet Restart**: Deploying this plugin will cause the kubelet to restart.
3. **Pod Restart Required**: After the plugin is successfully deployed, it will not affect running Pods, but will only take effect on newly created Pods. When the CNI is Kube-OVN, you need to manually add the parameter "--node-local-dns-ip=(IP address of the local DNS cache server)" to the kube-ovn-controller.
4. **NetworkPolicy Configuration**: If NetworkPolicy is configured in the cluster, you need to additionally allow both from and to directions for the node CIDR and nodeLocalDNSIP in the networkPolicy to ensure proper communication.
5. **MicroOS Cluster Upgrade**: For MicroOS clusters, upgrades are performed by rebuilding the cluster, which causes kubelet configuration changes to be lost. To make the NodeLocal DNS configuration persistent across upgrades, you need to add the `--cluster-dns` parameter to `kubeletExtraArgs` in the following three places of the cluster template:

   - `KubeadmControlPlane` → `initConfiguration` → `nodeRegistration` → `kubeletExtraArgs`
   - `KubeadmControlPlane` → `joinConfiguration` → `nodeRegistration` → `kubeletExtraArgs`
   - `KubeadmConfigTemplate` → `template` → `spec` → `joinConfiguration` → `nodeRegistration` → `kubeletExtraArgs`

   Add the following parameter to each of the above `kubeletExtraArgs` sections:

```
cluster-dns: "<NodeLocal_DNS_IP>"  # e.g., 169.254.20.10
```

WARNING

4.2.x Upgrade Notes

When upgrading this plugin from versions below 4.2.0 (excluding 4.2.0 itself) to 4.2.x, the following steps are required due to ResourcePatch compatibility issues:

Before Upgrade:

- Record the `--node-local-dns-ip` parameter value from the kube-ovn-controller ResourcePatch configuration
- Delete the ResourcePatch for the `deploy/kube-ovn-controller` resource

After Upgrade:

- Manually add the recorded `--node-local-dns-ip` parameter back to the kube-ovn-controller configuration

Note: This compatibility issue has been resolved in version 4.3 and above, so manual intervention is not required for upgrades to 4.3+.

Installation

Install via Marketplace

1. Navigate to **Administrator** > **Marketplace** > **Cluster Plugins**.
2. Search for "**Alauda Build of NodeLocal DNSCache**" in the plugin list.
3. Click **Install** to open the installation configuration page.
4. Configure the required parameters:

Parameter	Description	Example Value
IP	The IP address of the node local DNS cache server. For IPv4, it is recommended to use an address within the 169.254.0.0/16 range, preferably 169.254.20.10. For IPv6, it is recommended to use an address within the fd00::/8 range, preferably fd00::10.	169.254.20.10

5. Review the deployment notes and ensure your environment meets the requirements.
6. Click **Install** to complete the installation.
7. Wait for the plugin status to change to "**Ready**".

How It Works

Architecture

```
Pod → NodeLocal DNSCache → [Cache Hit] → Pod
            ↓
    [Cache Miss] → CoreDNS → Response → Cache & Pod
```

Configuration

Network Policy Configuration

> **Important**: If your cluster has NetworkPolicy enabled, you must configure proper rules to allow DNS traffic to the NodeLocal DNSCache. Without these rules, pods may not be able to resolve DNS queries.

When using NetworkPolicy, ensure the following DNS traffic is allowed:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns-cache
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - ipBlock:
            cidr: 169.254.20.10/32 # NodeLocal DNS IP address
      ports:
        - protocol: UDP
          port: 53
        - protocol: TCP
          port: 53
  egress:
    - to:
        - ipBlock:
            cidr: 169.254.20.10/32 # NodeLocal DNS IP address
      ports:
        - protocol: UDP
          port: 53
        - protocol: TCP
          port: 53
```

Configure CoreDNS

TOC

Overview

CoreDNS is the default DNS service for Kubernetes clusters. This guide describes how to configure host aliases, node selectors, and tolerations for CoreDNS.

Configuration

1. Navigate to **Administrator** > **Marketplace** > **Cluster Plugins**.
2. Search for "**Alauda Build of CoreDNS**" and click **Update**.
3. Configure the following parameters:

Host Alias

Configure custom DNS resolution entries.

Parameter	Description
IP	The IP address to be resolved
Domains	Domain names (separated by spaces) **Example**: `example.com test.example.com`

Node Selectors

Specify which nodes CoreDNS pods should run on.

Parameter	Description
Label Key	The label key to match on nodes
Label Value	The label value to match on nodes

Node Tolerations

Allow CoreDNS pods to be scheduled on nodes with taints.

Parameter	Description
Key	The taint key to tolerate
Value	The taint value (optional)
Type	`NoSchedule`, `PreferNoSchedule`, or `NoExecute`

4. Click **Update** to apply the configuration.

How To

Tasks for Ingress-Nginx

Tasks for Envoy Gateway

Tasks for Envoy Gateway

Overview

Prerequisites

Advanced Tasks

Related Documentation

More Configuration

Soft Data Center LB Solution (Alpha)

Soft Data Center LB Solution (Alpha)

Prerequisites

Procedure

Verification

Kube OVN

Understanding Kube-OVN CNI

Upstream OVN/OVS Components

Core Controller and Agent

Monitoring, Operation and Maintenance To

Preparing Kube-OVN Underlay

Usage Instructions

Terminology Explanation

Environment Requirements

Configuration Example

Kube-OVN

Overview

Prerequisites

Configuration S

Configuring Kube-OVN Network to Support Pod Multi-Network Interfaces

(Alpha)

Installing Multus CNI

Creating Subnets

Creating Pod with Multiple Network Interfaces

Verifying Dual Network Interface Creation

Additional Features

Configure Centralized Gateway

Using Label Selectors to Specify Gateway

Configure IPPool

Instructions

Precautions

Configure M

Default MTU Be

Customizing M

Automatic Interconnection of Underlay and O

weight: 13

Procedure

Isolation Between Underlay Subnets with u2oInterconnection

Configure Endpoint Health Checker

Configure Endpoint Health Checker

Overview

Key Features

alb

Tasks for ALB

Task: Migrate from OCP Route to GatewayAPI Route

Task: Migrate from OCP Route to GatewayAPI Route

Tasks for Ingress-Nginx

TOC

Prerequisites

Install ingress-nginx

Max Connections

Max-Worker-Connections ↗

Request Timeout

Configure request timeout ↗

Session Affinity (Sticky Sessions)

Configure sticky sessions ↗

Header Modification

action	link
set header in request	proxy-set-header ↗
remove header in request	set a empty header in request
set header in response	configuration-snippets ↗ with more-set-header ↗ directive

action	link
remove header in response	hide-headers ↗

URL Rewrite

rewrite ↗

HSTS (HTTP Strict Transport Security)

configure HSTS ↗

Rate Limiting

config rate limiting ↗

WAF

modsecurity ↗

Forward-header control

x-forwarded-prefix-header ↗

HTTPS

TLS re-encrypt and verify backend certificate

verify backend https certificate ↗

TLS edge termination

backend protocol ↗

Passthrough

ssl-passthrough ↗

Default Certificate

use the following yaml to deploy an ingress-nginx with default certificate

```yaml
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
spec:
  controller:
    extraArgs:
      default-ssl-certificate: $DEFAULT_CERT_NAMESPACE/$DEFAULT_CERT_NAME
```

please refer to default-ssl-certificate ↗

Add Pod Annotation in IngressNginx

Add pod annotation

Preserve Source IP

When traffic passes through load balancers or proxies, the original client IP address can be lost due to NAT (Network Address Translation). Preserving the source IP is important for:

- Access control and security policies
- Accurate logging and analytics
- Rate limiting per client

- Geolocation-based routing

Via HAProxy Proxy Protocol

How it works

The PROXY protocol ↗ is a network protocol for preserving client connection information when proxying TCP connections. It works by prepending a header to the TCP connection that contains the original source IP and port.

Traffic flow:

1. Client connects to HAProxy load balancer
2. HAProxy prepends PROXY protocol header with original client IP to the connection
3. Ingress-Nginx receives the connection and parses the PROXY protocol header
4. Ingress-Nginx extracts the real client IP from the header
5. Backend applications receive the correct client IP in `X-Forwarded-For` and `X-Real-IP` headers

Advantages:

- Works with any load balancer that supports PROXY protocol (HAProxy, AWS NLB, etc.)
- Preserves source IP across multiple proxy layers
- No impact on routing or node selection

Considerations:

- Both the load balancer and Ingress-Nginx must be configured to use PROXY protocol
- All traffic to Ingress-Nginx must use PROXY protocol once enabled (mixing PROXY and non-PROXY traffic will cause connection failures)

How to configure

Configure your HAProxy load balancer to send PROXY protocol headers, then deploy an ingress-nginx with proxy-protocol support enabled:

```yaml
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
  namespace: ingress-nginx-operator
spec:
  controller:
    config:
      use-proxy-protocol: "true" # enable proxy-protocol support
```

```
frontend tcp_front_80
    bind *:80
    mode tcp
    default_backend ingress_tcp_80

frontend tcp_front_443
    bind *:443
    mode tcp
    default_backend ingress_tcp_443

backend ingress_tcp_80
    mode tcp
    balance roundrobin
    server node1 192.168.133.46:80 check send-proxy-v2

backend ingress_tcp_443
    mode tcp
    balance roundrobin
    server node1 192.168.133.46:443 check send-proxy-v2
```

For more details, see PROXY protocol documentation ↗.

Note: HAProxy can use TCP mode to forward traffic without handling TLS certificates. Since the PROXY protocol works at the TCP layer, you can let Ingress-Nginx handle HTTPS termination and certificate management directly, eliminating the need to configure certificates in HAProxy.

Via MetalLB with externalTrafficPolicy=Local

How it works

When using a Kubernetes Service with `type: LoadBalancer`, the default behavior (`externalTrafficPolicy: Cluster`) performs source NAT, which replaces the client IP with the node's IP. Setting `externalTrafficPolicy: Local` preserves the source IP by:

1. **Direct routing**: Traffic is only routed to pods on the same node that received the traffic
2. **No SNAT**: The kube-proxy does not perform source NAT, preserving the original client IP
3. **Health checks**: Only nodes with healthy local pods are included in the load balancer pool

Traffic flow:

1. Client connects to MetalLB virtual IP
2. MetalLB routes traffic directly to a node with Ingress-Nginx pods
3. Traffic goes directly to the local Ingress-Nginx pod without SNAT
4. Ingress-Nginx sees the real client IP
5. Backend applications receive the correct client IP in headers

Advantages:

- Simple configuration, no additional protocol required
- Native Kubernetes feature
- Lower latency (no extra proxy hop)

Considerations:

- **Uneven load distribution**: Traffic can only go to nodes with local pods, potentially causing imbalanced load
- **Pod scheduling**: Ingress-Nginx pods must be scheduled on nodes that MetalLB can route to (use nodeSelector to ensure alignment)
- **Health check behavior**: If all local pods are unhealthy, the node is removed from load balancing entirely

How to configure

Deploy an ingress-nginx with `externalTrafficPolicy: Local` and ensure pod placement aligns with MetalLB configuration:

```yaml
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
  namespace: ingress-nginx-operator
spec:
  controller:
    service:
      type: LoadBalancer                    # Use MetalLB to provision a LoadBalancer service
      externalTrafficPolicy: Local          # Preserve source IP by routing traffic only to local pods
      annotations:
        metallb.universe.tf/address-pool: demo-pool  # Specify the MetalLB IP address pool to use
    nodeSelector:                           # Schedule pods only on nodes matching these labels. This selector must match the MetalLB address pool's node selector
      ingress-nginx: "true"
```

Important: The `nodeSelector` must match the nodes in your MetalLB address pool configuration to ensure Ingress-Nginx pods are scheduled on nodes that can receive traffic from MetalLB.

For more details, see externalTrafficPolicy documentation ↗.

Tasks for Envoy Gateway

TOC

Overview

This document extends the main configuration path (operator → gateway → route → policy) and introduces advanced tasks beyond the standard resource configurations covered in the previous documents.

When applying configuration changes in the Gateway API, there are three primary approaches available:

1. **Modify standard Gateway API resources**: Directly edit `Gateway`, `HTTPRoute`, `TCPRoute`, `UDPRoute`, and other core resources.
2. **Attach policies via PolicyAttachment**: Use `SecurityPolicy`, `ClientTrafficPolicy`, `BackendTrafficPolicy`, and other policy resources to extend Gateway and Route behavior.
3. **Configure global settings**: Modify the `EnvoyGatewayCtl` to change `envoy-gateway` instance behavior or other global settings that affect all gateways.

This document focuses on advanced tasks and special scenarios that are not part of the main configuration path, including cross-namespace routing, observability setup, deployment customization, and troubleshooting.

Prerequisites

1. Configure EnvoyGatewayCtl
2. Configure Gateway
3. Configure Route
4. Configure GatewayAPI Policy

Advanced Tasks

OpenTelemetry (OTel)

Please follow instructions in OpenTelemetry Integration ↗, but use `EnvoyGatewayCtl` to modify the `envoy-gateway-config`.

How To Attach to Listener Created In Other Namespace

In the Gateway's listener configuration, you need to specify which namespaces are allowed to attach Routes to it.

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: example-gateway
spec:
  listeners:
    - name: http-80
      protocol: HTTP
      port: 80
      allowedRoutes:
        namespaces:
          from: All # without limit
    - name: http-81
      protocol: HTTP
      port: 81
      allowedRoutes:
        namespaces:
          from: Same # only allow routes in the same namespace
    - name: http-82
      protocol: HTTP
      port: 82
      allowedRoutes:
        namespaces:
          from: Selector
          selector:
            matchLabels:
              team: frontend # only allow routes in the namespace with la
bel team=frontend
```

Please refer to Cross-Namespace routing ↗ for more details.

How To Use a Certificate Created In Another Namespace

To use a certificate created in another namespace, create a `ReferenceGrant` in the namespace where the certificate is stored. Please follow instructions in cross-namespace-certificate-references ↗ and referencegrant ↗.

> **NOTE**
>
> You cannot specify individual `secret` resources; you must allow the entire namespace

How To Use SSL Passthrough

Please follow instructions in

- tls ↗
- tls-passthrough ↗

How To Change the Minimum TLS Version

Please follow instructions in customize-gateway-tls-parameters ↗

```
cat <<EOF | kubectl apply -f -
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: ClientTrafficPolicy
metadata:
  name: enforce-tls-13
  namespace: default
spec:
  targetRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: eg
  tls:
    minVersion: "1.3"
EOF
```

The `.spec.tls` field in `ClientTrafficPolicy` is clienttlssettings↗. If you need to customize cipher suites in addition to the minimum TLS version, refer to the same upstream task and API reference.

How To Specify NodePort When Using NodePort Service

When using a NodePort service, Kubernetes assigns a NodePort value to each service port. When accessing the service through a node IP, use the assigned NodePort instead of the service port.

There are two approaches:

Manually retrieve the NodePort assignment by following get nodeport from svc port.

Manually specify the NodePort in the `EnvoyProxy` configuration instead of letting Kubernetes automatically assign it.

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
spec:
  ipFamily: DualStack
  provider:
    kubernetes:
      envoyDeployment:
        container:
          imageRepository: registry.alauda.cn:60080/acp/envoyproxy/envoy
      envoyService:
        patch: 1
          type: StrategicMerge
          value:
            spec:
              ports:
                - nodeport: 31888
                  port: 80
        type: NodePort
    type: Kubernetes
```

1 Use patch field to patch the generated service resource to specify the NodePort

NOTE

NodePort can only be within a specific range, typically `30000-32767`. If you want the Gateway listener port and NodePort to be consistent, your listener port must also be within the NodePort range.

How To Specify a VIP When Using MetalLB

When using MetalLB as the LoadBalancer provider, you can specify a static VIP for the Gateway service through service annotations.

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
  namespace: demo
spec:
  provider:
    type: Kubernetes
    kubernetes:
      envoyService:
        type: LoadBalancer
        annotations: 1
          metallb.universe.tf/address-pool: production 2
          metallb.universe.tf/loadBalancerIPs: VIP_IP 3
```

1. Add MetalLB annotations in the `envoyService.annotations` field
2. Specify the address pool name to allocate IP from
3. Or specify a specific IP address (must be within the address pool range)

Available Annotations:

Annotation	Description
`metallb.universe.tf/address-pool`	Select the address pool to allocate IP from

Annotation	Description
`metallb.universe.tf/loadBalancerIPs`	Specify a specific IP address (supports multiple IPs, comma-separated)

NOTE

- The specified IP must be within a configured MetalLB address pool
- Make sure MetalLB is properly installed and configured before specifying VIPs
- For MetalLB configuration, see Configure MetalLB

How To Add Pod Annotations In Envoy Gateway

Add pod annotation

How To Set NodeSelector And Tolerations For `envoy-gateway-operator`

Update the `Subscription` resource.

```
# example of nodeSelector and tolerations
kubectl patch subscription envoy-gateway-operator  -n envoy-gateway-opera
tor   --type='merge' -p '
{
  "spec": {
    "config": {
      "nodeSelector": {
        "node-role.kubernetes.io/infra": ""
      },
      "tolerations": [
        {
          "effect": "NoSchedule",
          "key": "node-role.kubernetes.io/infra",
          "operator": "Equal",
          "value": "reserved"
        }
      ]
    }
  }
}'
```

How To Set NodeSelector And Tolerations For `envoy-gateway`

Update the `EnvoyGatewayCtl` resource.

```
# in default $NAME=cpaas-default and $NS=envoy-gateway-operator
kubectl patch envoygatewayctl $NAME -n $NS --type='merge' -p '
{
  "spec": {
    "deployment": {
      "pod": {
        "nodeSelector": {
          "node-role.kubernetes.io/infra": ""
        },
        "tolerations": [
          {
            "effect": "NoSchedule",
            "key": "node-role.kubernetes.io/infra",
            "operator": "Equal",
            "value": "reserved"
          }
        ]
      }
    }
  }
}'
```

How To Set NodeSelector And Tolerations For `envoy-proxy`

Update the `EnvoyProxy` resource.

```
kubectl patch envoyproxy $NAME -n $NS --type='merge' -p '
{
  "spec": {
    "provider": {
      "kubernetes": {
        "envoyDeployment": {
          "pod": {
            "nodeSelector": {
              "node-role.kubernetes.io/infra": ""
            },
            "tolerations": [
              {
                "effect": "NoSchedule",
                "key": "node-role.kubernetes.io/infra",
                "operator": "Equal",
                "value": "reserved"
              }
            ]
          }
        }
      }
    }
  }
}'
```

How To Use `hostNetwork` In `envoy-proxy`

Using `hostNetwork: true` allows the Envoy proxy pods to use the host network namespace directly. This can be useful for:

- achieving better network performance
- accessing the gateway through the node IP directly

Considerations:

- Pods using hostNetwork will bind directly to the host's network interfaces
- Port conflicts may occur if multiple pods try to use the same port on the same node
- Security isolation is reduced as pods share the host's network namespace

- You should use `nodeSelector` or `affinity` rules to control pod placement and avoid port conflicts

There are two approaches to configure `hostNetwork`, depending on whether you want to use privileged ports (< 1024) directly:

Approach 1: Using Port Offset (Default, Recommended)

This is the default and recommended approach. Envoy Gateway automatically adds an offset of 10000 to privileged ports to avoid requiring special permissions.

Pros:

- No special permissions or capabilities required
- More secure because it runs as a non-root user without additional privileges
- Simpler configuration
- Works out of the box

Cons:

- Clients must use offset ports (10080, 10443)

Configuration:

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
  namespace: demo
spec:
  provider:
    type: Kubernetes
    kubernetes:
      envoyDeployment:
        patch:
          type: StrategicMerge
          value:
            spec:
              template:
                spec:
                  hostNetwork: true                    # Enable host netw
ork mode
                  dnsPolicy: ClusterFirstWithHostNet   # Required for pro
per DNS resolution
        pod:
          nodeSelector:                                # Recommended: con
trol pod placement to avoid conflict
            kubernetes.io/hostname: "demo"
```

Access:

- Port 80 → Access via `http://<node-ip>:10080`
- Port 443 → Access via `https://<node-ip>:10443`

Approach 2: Using Privileged Ports Directly with useListenerPortAsContainerPort

This approach allows Envoy to bind to privileged ports (< 1024) directly, such as port 80 and 443.

Pros:

- Can use standard ports (80 and 443) directly
- Better compatibility with clients expecting standard ports

Cons:

- Requires NET_BIND_SERVICE capability
- Slightly reduced security compared to port offset approach
- More complex configuration

Configuration:

```
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
  namespace: demo
spec:
  provider:
    type: Kubernetes
    kubernetes:
      useListenerPortAsContainerPort: true  # Disable port offset
      envoyDeployment:
        patch:
          type: StrategicMerge
          value:
            spec:
              template:
                spec:
                  hostNetwork: true
                  dnsPolicy: ClusterFirstWithHostNet
                  containers:
                  - name: envoy
                    command:
                      - /usr/local/bin/envoy-with-cap  # use envoy with f
ilecap
                    securityContext:
                      capabilities:
                        add:
                        - NET_BIND_SERVICE  # Required for binding to pri
vileged ports
        pod:
          nodeSelector:                                # Recommended: con
trol pod placement to avoid conflict
            kubernetes.io/hostname: "demo"
```

Access:

- Port 80 → Access via `http://<node-ip>:80`
- Port 443 → Access via `https://<node-ip>:443`

How To Access a LoadBalancer VIP from Within the Cluster

By default, Envoy Gateway creates LoadBalancer services with `externalTrafficPolicy: Local`. This policy preserves client source IP addresses but has an important limitation: requests from cluster nodes without Envoy Gateway pods will fail because traffic is not forwarded to other nodes.

Solution 1: Use Service ClusterIP (Recommended for in-cluster access)

For applications running inside the cluster, use the service ClusterIP instead of the LoadBalancer VIP. This avoids the routing limitation entirely.

Solution 2: Change to Cluster Traffic Policy

If you need to access the LoadBalancer VIP from any cluster node, change `externalTrafficPolicy` to `Cluster`:

```
kubectl patch envoyproxy $GATEWAY_NAME -n $GATEWAY_NS --type='json' -p='[
  {"op": "replace", "path": "/spec/provider/kubernetes/envoyService/exter
nalTrafficPolicy", "value": "Cluster"}
]'
```

Related Documentation

- Configure EnvoyGatewayCtl
- Configure Gateway
- Configure Route
- Configure GatewayAPI Policy

More Configuration

Please refer to EnvoyGateway Tasks ↗

Soft Data Center LB Solution (Alpha)

Deploy a pure software data center load balancer (LB) by creating a highly available load balancer outside the cluster, providing load balancing capabilities for multiple ALBs to ensure stable business operations. It supports configuration for IPv4 only, IPv6 only, or both IPv4 and IPv6 dual stack.

TOC

Prerequisites

1. Prepare two or more host nodes as LB. It is recommended to install Ubuntu 22.04 operating system on LB nodes to reduce the time for LB to forward traffic to abnormal backend nodes.
2. Pre-install the following software on all host nodes of the external LB (this chapter takes two external LB host nodes as an example):
   - `ipvsadm`
   - `container-runtime such as containerd`

3. Ensure that the `container-runtime` starts on boot for each host.
4. Ensure that the clock of each host node is synchronized.
5. Prepare the image for Keepalived, used to start the external LB service; the platform already contains this image. The image address is in the following format: `<image repository address>/tkestack/keepalived:<version suffix>`. The version suffix may vary slightly among different versions. You can obtain the image repository address and version suffix as follows. This document uses `build-harbor.alauda.cn/tkestack/keepalived:v3.16.0-beta.3.g598ce923` as an example.

   - In the global cluster, execute `kubectl get prdb base -o json | jq .spec.registry.address` to get the **image repository address** parameter.
   - In the directory where the installation package is extracted, execute `cat ./installer/res/artifacts.json |grep keepalived -C 2|grep tag|awk '{print $2}'|awk -F '"' '{print $2}'` to get the **version suffix**.

Procedure

Note: The following operations must be executed once on each external LB host node, and the `hostname` of the host nodes must not be duplicated.

1. Add the following configuration information to the file `/etc/modules-load.d/alive.kmod.conf`.

```
ip_vs
ip_vs_rr
ip_vs_wrr
ip_vs_sh
nf_conntrack_ipv4
nf_conntrack
ip6t_MASQUERADE
nf_nat_masquerade_ipv6
ip6table_nat
nf_conntrack_ipv6
nf_defrag_ipv6
nf_nat_ipv6
ip6_tables
```

2. Add the following configuration information to the file `/etc/sysctl.d/alive.sysctl.conf`.

```
net.ipv4.ip_forward = 1
net.ipv4.conf.all.arp_accept = 1
net.ipv4.vs.conntrack = 1
net.ipv4.vs.conn_reuse_mode = 0
net.ipv4.vs.expire_nodest_conn = 1
net.ipv4.vs.expire_quiescent_template = 1
net.ipv6.conf.all.forwarding=1
```

3. Restart using the `reboot` command.

4. Create a folder for the Keepalived configuration file.

```
mkdir -p /etc/keepalived
mkdir -p /etc/keepalived/kubecfg
```

5. Modify the configuration items according to the comments in the following file and save them in the `/etc/keepalived/` folder, naming the file `alive.yaml`.

```
instances:
  - vip: # Multiple VIPs can be configured
      vip: 192.168.128.118 # VIPs must be different
      id: 20 # Each VIP's ID must be unique, optional
      interface: "eth0"
      check_interval: 1 # optional, default 1: interval to execute chec
k script
      check_timeout: 3  # optional, default 3: check script timeout per
iod
      name: "vip-1" # Identifier for this instance, can only contain al
phanumeric characters and hyphens, cannot start with a hyphen
      peer: [ "192.168.128.116", "192.168.128.75" ] # Keepalived node I
P, actual generated keepalived.conf will remove all IPs on the interfac
e.
      kube_lock:
        kubecfgs: # The kube-config list used by kube-lock will sequent
ially attempt these kubecfgs for leader election in Keepalived
          - "/live/cfg/kubecfg/kubecfg01.conf"
          - "/live/cfg/kubecfg/kubecfg02.conf"
          - "/live/cfg/kubecfg/kubecfg03.conf"
    ipvs: # Configuration for option IPVS
      ips: [ "192.168.143.192", "192.168.138.100","192.168.129.100" ] #
IPVS backend, change k8s master node IP to ALB node's node IP
      ports: # Configure health check logic for each port on the VIP
        - port: 80 # The port on the virtual server must match the real
server's port
          virtual_server_config: |
            delay_loop 10  # Interval for performing health checks on t
he real server
            lb_algo rr
            lb_kind NAT
            protocol TCP
          raw_check: |
            TCP_CHECK {
                connect_timeout 10
                connect_port 1936
            }
  - vip:
      vip: 2004::192:168:128:118
      id: 102
      interface: "eth0"
      peer: [ "2004::192:168:128:75","2004::192:168:128:116" ]
      kube_lock:
```

```
        kubecfgs: # The kube-config list used by kube-lock will sequent
ially attempt these kubecfgs for leader election in Keepalived
          - "/live/cfg/kubecfg/kubecfg01.conf"
          - "/live/cfg/kubecfg/kubecfg02.conf"
          - "/live/cfg/kubecfg/kubecfg03.conf"
    ipvs:
      ips: [ "2004::192:168:143:192","2004::192:168:138:100","2004::19
2:168:129:100" ]
      ports:
        - port: 80
          virtual_server_config: |
            delay_loop 10
            lb_algo rr
            lb_kind NAT
            protocol TCP
          raw_check: |
            TCP_CHECK {
                connect_timeout 1
                connect_port 1936
            }
```

6. Execute the following command in the business cluster to check the certificate expiration date in the configuration file, ensuring that the certificate is still valid. The LB functionality will become unavailable after the certificate expires, requiring contact with the platform administrator for a certificate update.

```
openssl x509 -in <(cat /etc/kubernetes/admin.conf | grep client-certifi
cate-data | awk '{print $NF}' | base64 -d ) -noout -dates
```

7. Copy the `/etc/kubernetes/admin.conf` file from the three Master nodes in the Kubernetes cluster to the `/etc/keepalived/kubecfg` folder on the external LB nodes, naming them with an index, e.g., `kubecfg01.conf`, and modify the `apiserver` node addresses in these three files to the actual node addresses of the Kubernetes cluster. **Note**: After the platform certificate is updated, this step needs to be executed again, overwriting the original files.

8. Check the validity of the certificates.

    1. Copy `/usr/bin/kubectl` from the Master node of the business cluster to the LB node.
    2. Execute `chmod +x /usr/bin/kubectl` to grant execution permissions.
    3. Execute the following commands to confirm certificate validity.

```
kubectl --kubeconfig=/etc/keepalived/kubecfg/kubecfg01.conf get node
kubectl --kubeconfig=/etc/keepalived/kubecfg/kubecfg02.conf get node
kubectl --kubeconfig=/etc/keepalived/kubecfg/kubecfg03.conf get node
```

If the following results are returned, the certificate is valid.

```
kubectl --kubeconfig=/etc/keepalived/kubecfg/kubecfg01.conf get node
## Output
NAME              STATUS   ROLES                  AGE     VERSION
192.168.129.100   Ready    <none>                 7d22h   v1.25.6
192.168.134.167   Ready    control-plane,master   7d22h   v1.25.6
192.168.138.100   Ready    <none>                 7d22h   v1.25.6
192.168.143.116   Ready    control-plane,master   7d22h   v1.25.6
192.168.143.192   Ready    <none>                 7d22h   v1.25.6
192.168.143.79    Ready    control-plane,master   7d22h   v1.25.6

kubectl --kubeconfig=/etc/keepalived/kubecfg/kubecfg02.conf get node
## Output
NAME              STATUS   ROLES                  AGE     VERSION
192.168.129.100   Ready    <none>                 7d22h   v1.25.6
192.168.134.167   Ready    control-plane,master   7d22h   v1.25.6
192.168.138.100   Ready    <none>                 7d22h   v1.25.6
192.168.143.116   Ready    control-plane,master   7d22h   v1.25.6
192.168.143.192   Ready    <none>                 7d22h   v1.25.6
192.168.143.79    Ready    control-plane,master   7d22h   v1.25.6

kubectl --kubeconfig=/etc/keepalived/kubecfg/kubecfg03.conf get node
## Output
NAME              STATUS   ROLES                  AGE     VERSION
192.168.129.100   Ready    <none>                 7d22h   v1.25.6
192.168.134.167   Ready    control-plane,master   7d22h   v1.25.6
192.168.138.100   Ready    <none>                 7d22h   v1.25.6
192.168.143.116   Ready    control-plane,master   7d22h   v1.25.6
192.168.143.192   Ready    <none>                 7d22h   v1.25.6
192.168.143.79    Ready    control-plane,master   7d22h   v1.25.6
```

9. Upload the Keepalived image to the external LB node and run Keepalived using nerdctl.

```
nerdctl run -dt --restart=always --privileged --network=host -v /etc/ke
epalived:/live/cfg build-harbor.alauda.cn/tkestack/keepalived:v3.16.0-b
eta.3.g598ce923
```

10. Run the following command on the node accessing `keepalived`: `sysctl -w net.ipv4.conf.all.arp_accept=1`.

Verification

1. Run the command `ipvsadm -ln` to view the IPVS rules, and you will see IPv4 and IPv6 rules applicable to the business cluster ALBs.

```
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight        ActiveConn InAc
tConn
TCP  192.168.128.118:80 rr
  -> 192.168.129.100:80           Masq    1      0          0
  -> 192.168.138.100:80           Masq    1      0          0
  -> 192.168.143.192:80           Masq    1      0          0
TCP  [2004::192:168:128:118]:80 rr
  -> [2004::192:168:129:100]:80   Masq    1      0          0
  -> [2004::192:168:138:100]:80   Masq    1      0          0
  -> [2004::192:168:143:192]:80   Masq    1      0          0
```

2. Shut down the LB node where the VIP is located and test whether the VIP of both IPv4 and IPv6 can successfully migrate to another node, typically within 20 seconds.
3. Use the `curl` command on a non-LB node to test if communication with the VIP is normal.

```
curl 192.168.128.118

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

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

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

```
curl -6 [2004::192:168:128:118]:80 -g

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

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

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

Kube OVN

Understanding Kube-OVN CNI

- Upstream OVN/OVS Components
- Core Controller and Agent
- Monitoring, Operation and Maintenance To

Preparing Kube-OVN Underlay

- Usage Instructions
- Terminology Explanation
- Environment Requirements
- Configuration Example

Kube-OVN

- Overview
- Prerequisites
- Configuration S

Configuring Kube-OVN Network to Support Pod Multi-Network Interfaces (Alpha)

- Installing Multus CNI
- Creating Subnets
- Creating Pod with Multiple Network Interfaces
- Verifying Dual Network Interface Creation
- Additional Features

Configure Centralized Gateway

Configure IPPool

- Instructions

Using Label Selectors to Specify Gateway

Precautions

Configure M

Default MTU Be

Automatic Interconnection of Underlay and O

weight: 13

Procedure

Isolation Between Underlay Subnets with u2oInterconnection

Understanding Kube-OVN CNI

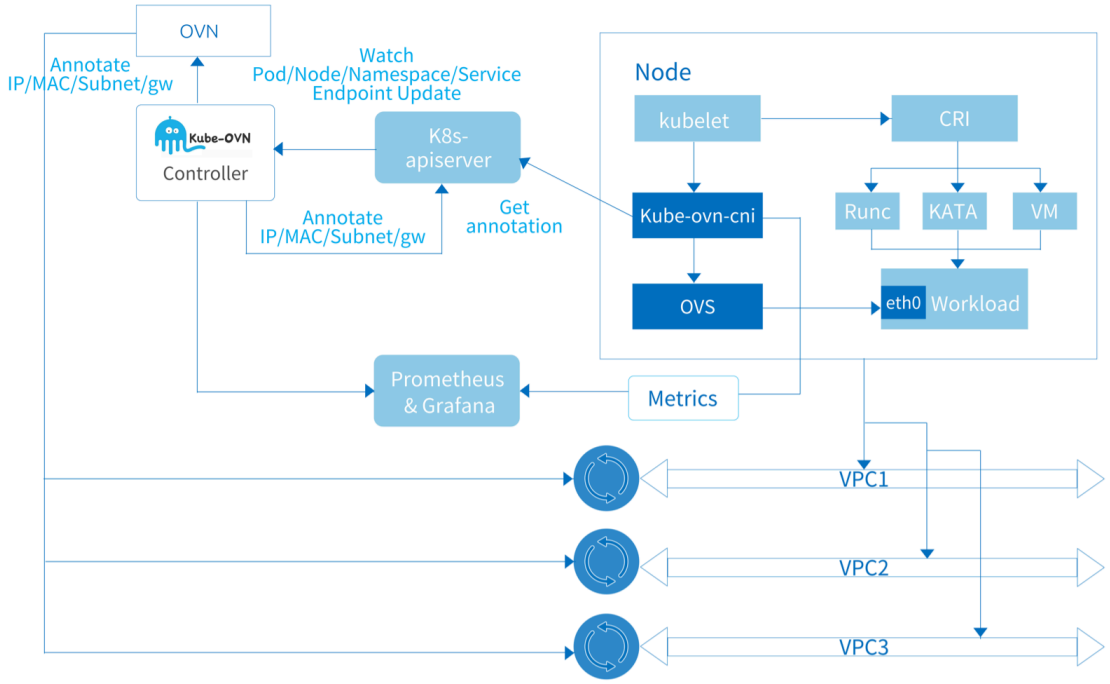
This document describes the general architecture of Kube-OVN, the functionality of each component and how they interact with each other.

Overall, Kube-OVN serves as a bridge between Kubernetes and OVN, combining proven SDN with Cloud Native. This means that Kube-OVN not only implements network specifications under Kubernetes, such as CNI, Service and Networkpolicy, but also brings a large number of SDN domain capabilities to cloud-native, such as logical switches, logical routers, VPCs, gateways, QoS, ACLs and traffic mirroring.

Kube-OVN also maintains a good openness to integrate with many technology solutions, such as Cilium, Submariner, Prometheus, KubeVirt, etc.

The components of Kube-OVN can be broadly divided into three categories.

- Upstream OVN/OVS components.
- Core Controller and Agent.
- Monitoring, operation and maintenance tools and extension components.



TOC

Upstream OVN/OVS Components

This type of component comes from the OVN/OVS community with specific modifications for Kube-OVN usage scenarios. OVN/OVS itself is a mature SDN system for managing virtual machines and containers, and we strongly recommend that users interested in the Kube-OVN implementation read ovn-architecture(7)↗ first to understand what OVN is and how to integrate with it. Kube-OVN uses the northbound interface of OVN to create and coordinate virtual networks and map the network concepts into Kubernetes.

All OVN/OVS-related components have been packaged into images and are ready to run in Kubernetes.

ovn-central

The `ovn-central` Deployment runs the control plane components of OVN, including `ovn-nb`, `ovn-sb`, and `ovn-northd`.

- `ovn-nb`: Saves the virtual network configuration and provides an API for virtual network management. `kube-ovn-controller` will mainly interact with `ovn-nb` to configure the virtual network.
- `ovn-sb`: Holds the logical flow table generated from the logical network of `ovn-nb`, as well as the actual physical network state of each node.
- `ovn-northd`: translates the virtual network of `ovn-nb` into a logical flow table in `ovn-sb`.

Multiple instances of `ovn-central` will synchronize data via the Raft protocol to ensure high availability.

ovs-ovn

`ovs-ovn` runs as a DaemonSet on each node, with `openvswitch`, `ovsdb`, and `ovn-controller` running inside the Pod. These components act as agents for `ovn-central` to translate logical flow tables into real network configurations.

Core Controller and Agent

This part is the core component of Kube-OVN, serving as a bridge between OVN and Kubernetes, bridging the two systems and translating network concepts between them. Most of the core functions are implemented in these components.

kube-ovn-controller

This component performs the translation of all resources within Kubernetes to OVN resources and acts as the control plane for the entire Kube-OVN system. The `kube-ovn-controller` listens for events on all resources related to network functionality and updates the logical network within the OVN based on resource changes. The main resources listened including:

Pod, Service, Endpoint, Node, NetworkPolicy, VPC, Subnet, Vlan, ProviderNetwork.

Taking the Pod event as an example, `kube-ovn-controller` listens to the Pod creation event, allocates the address via the built-in in-memory IPAM function, and calls `ovn-central` to create logical ports, static routes and possible ACL rules. Next, `kube-ovn-controller` writes the assigned address and subnet information such as CIDR, gateway, route, etc. to the annotation of the Pod. This annotation is then read by `kube-ovn-cni` and used to configure the local network.

kube-ovn-cni

This component runs on each node as a DaemonSet, implements the CNI interface, and operates the local OVS to configure the local network.

This DaemonSet copies the `kube-ovn` binary to each machine as a tool for interaction between `kubelet` and `kube-ovn-cni`. This binary sends the corresponding CNI request to `kube-ovn-cni` for further operation. The binary will be copied to the `/opt/cni/bin` directory by default.

`kube-ovn-cni` will configure the specific network to perform the appropriate traffic operations, and the main tasks including:

1. Config `ovn-controller` and `vswitchd`.
2. Handle CNI Add/Del requests:
    1. Create or delete veth pair and bind or unbind to OVS ports.

    2. Configure OVS ports
    3. Update host iptables/ipset/route rules.
3. Dynamically update the network QoS.
4. Create and configure the `ovn0` NIC to connect the container network and the host network.
5. Configure the host NIC to implement Vlan/Underlay/EIP.
6. Dynamically config inter-cluster gateways.

Monitoring, Operation and Maintenance Tools and Extension Components

These components provide monitoring, diagnostics, operations tools, and external interface to extend the core network capabilities of Kube-OVN and simplify daily operations and maintenance.

kube-ovn-speaker

This component is a DaemonSet running on a specific labeled nodes that publish routes to the external, allowing external access to the container directly through the Pod IP.

kube-ovn-pinger

This component is a DaemonSet running on each node to collect OVS status information, node network quality, network latency, etc.

kube-ovn-monitor

This component collects OVN status information and the monitoring metrics.

kubectl-ko

This component is a kubectl plugin, which can quickly run common operations.

Preparing Kube-OVN Underlay Physical Network

The container network under Kube-OVN Underlay transport mode relies on physical network support. Before deploying the Kube-OVN Underlay network, please collaborate with the network administrator to plan and complete the relevant configurations of the physical network in advance, ensuring network connectivity.

TOC

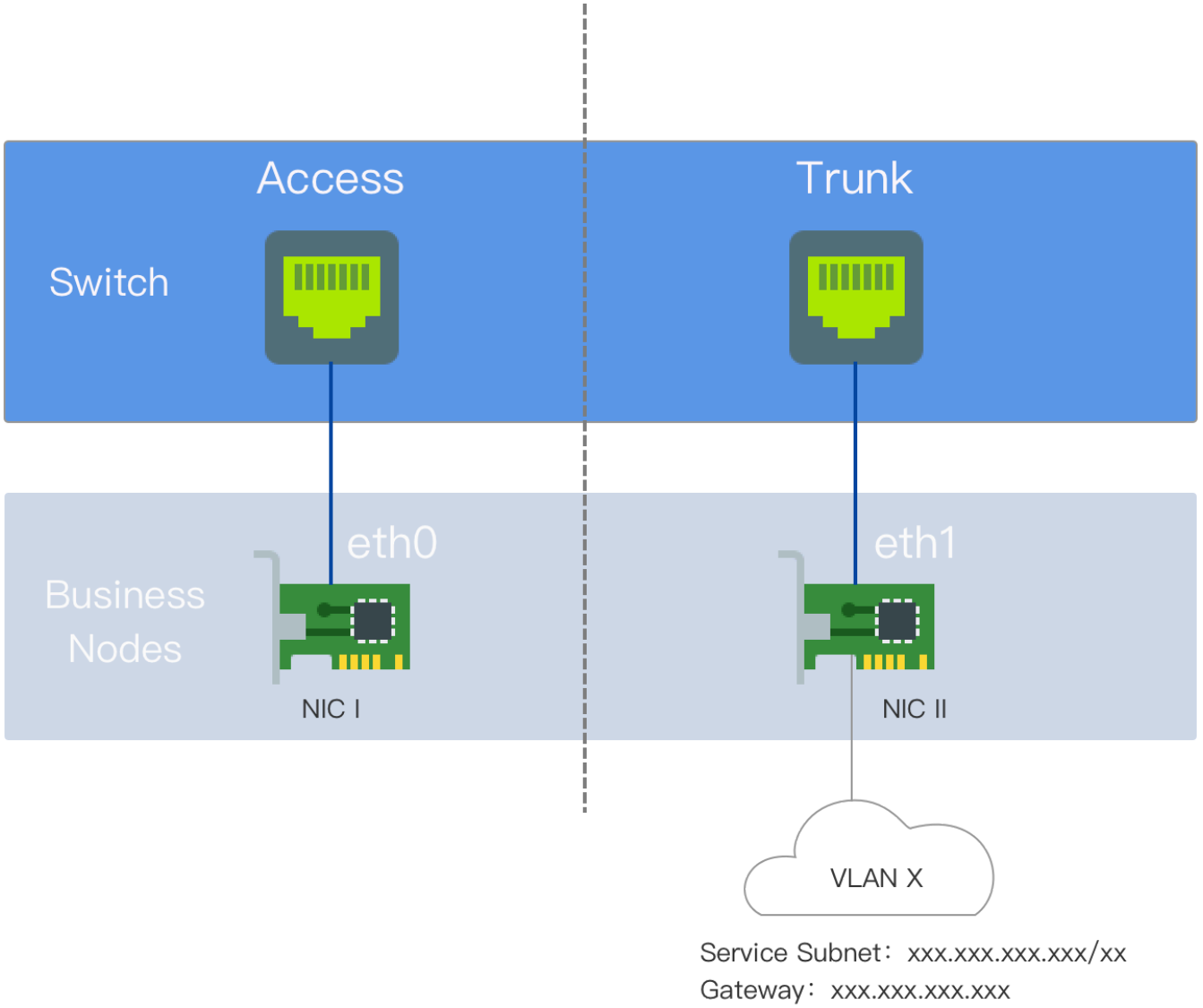
Usage Instructions

Kube-OVN Underlay requires deployment with multiple network interface cards (NICs), and the Underlay subnet must exclusively use one NIC. No other types of traffic, such as SSH,

should be on that NIC; they should utilize other NICs.

Before use, ensure that the node server has at least a **dual-NIC** environment, and it is recommended that the NIC speed is **at least 10 Gbps or higher** (e.g., 10 Gbps, 25 Gbps, 40 Gbps).

- NIC One: The NIC with the default route, configured with an IP address, interconnected with the external switch interface, which is set to Access mode.
- NIC Two: The NIC without the default route and not configured with an IP address, interconnected with the external switch interface, which is set to Trunk mode. The Underlay subnet exclusively uses NIC Two.



Terminology Explanation

VLAN (Virtual Local Area Network) is a technology that logically divides a local area network into multiple segments (or smaller LANs) to facilitate data exchange for virtual workgroups.

The emergence of VLAN technology allows administrators to logically segment different users within the same physical local area network into distinct broadcast domains based on actual application needs. Each VLAN comprises a group of computer workstations with similar requirements and possesses the same properties as a physically formed LAN. Since VLANs are logically divided rather than physically, workstations within the same VLAN are not confined to the same physical area; they can exist across different physical LAN segments.

The main advantages of VLANs include:

- Port Segmentation. Even on the same switch, ports in different VLANs cannot communicate with each other. A physical switch can function as multiple logical switches. This is commonly used to control mutual access between different departments and sites in a network.
- Network Security. Different VLANs cannot communicate directly, eliminating the insecurity of broadcast information. Broadcast and unicast traffic within a VLAN will not be forwarded to other VLANs, helping control traffic, reduce equipment investments, simplify network management, and improve network security.
- Flexible Management. When changing a user's network affiliation, there's no need to replace ports or cables; it merely requires a software configuration change.

Environment Requirements

In Underlay mode, Kube-OVN bridges a physical NIC to OVS and sends packets directly to the external through that physical NIC. The L2/L3 forwarding capability relies on the underlying network devices. The corresponding gateway, VLAN, and security policies need to be pre-configured on the underlying network devices.

- **Network Configuration Requirements**
  - Kube-OVN checks the gateway's connectivity via ICMP protocol when starting containers; the underlying gateway must respond to ICMP requests.
  - For service access traffic, Pods will first send packets to the gateway, which must have the ability to forward packets back to the local subnet.

  - When the switch or bridge has Hairpin functionality enabled, **Hairpin must be disabled**. If using a VMware virtual machine environment, set **Net.ReversePathFwdCheckPromisc** on the VMware host to **1**, and Hairpin does not need to be disabled.
  - The bridging NIC **cannot** be a **Linux Bridge**.
  - NIC bonding modes support Mode 0 (balance-rr), Mode 1 (active-backup), Mode 4 (802.3ad), Mode 6 (balance-alb), with a recommendation to use 0 or 1. Other bonding modes have not been tested; please use them with caution.
- **IaaS (Virtualization) Layer Configuration Requirements**
  - For OpenStack VM environments, the **PortSecurity** for the corresponding network port needs to be disabled.
  - For VMware's vSwitch network, **MAC Address Changes**, **Forged Transmits**, and **Promiscuous Mode Operation** must all be set to **Accept**.
  - For public clouds such as AWS, GCE, and Alibaba Cloud, Underlay mode networks cannot be supported due to their lack of user-defined MAC address capabilities.

Configuration Example

The nodes in this example are dual-NIC physical machines. NIC One is the NIC with the default route; NIC Two is the NIC without the default route and is not configured with an IP address, exclusively used for the Underlay subnet. NIC Two is interconnected with the external switch.

- On the switch side, the interface connected to NIC Two should be configured in Trunk mode, allowing the corresponding VLANs to pass through.
- Configure the gateway address of the cluster subnet on the corresponding vlan-interface interface. If dual-stack is needed, the IPv6 gateway address can also be configured simultaneously.
- If the gateway is behind a firewall, access from node nodes to the cluster-cidr network must be permitted.
- No configuration is needed for server NICs.

Switch Configuration

Configure the VLAN Interface:

```
#
interface Vlan-interface74
  ip address 192.168.74.254 255.255.255.0   //IPv4 gateway address
  ipv6 address 2074::192:168:74:254/64  //IPv6 gateway address
#
```

Configure the interface connected to NIC Two:

```
#
interface Ten-GigabitEthernet1/0/19
  port link mode bridge
  port link-type trunk  // Configure the interface to Trunk mode
  undo port trunk permit vlan 1
  port trunk permit vlan 74  // Allow the corresponding VLAN to pass thro
ugh
#
```

Check Network Connectivity

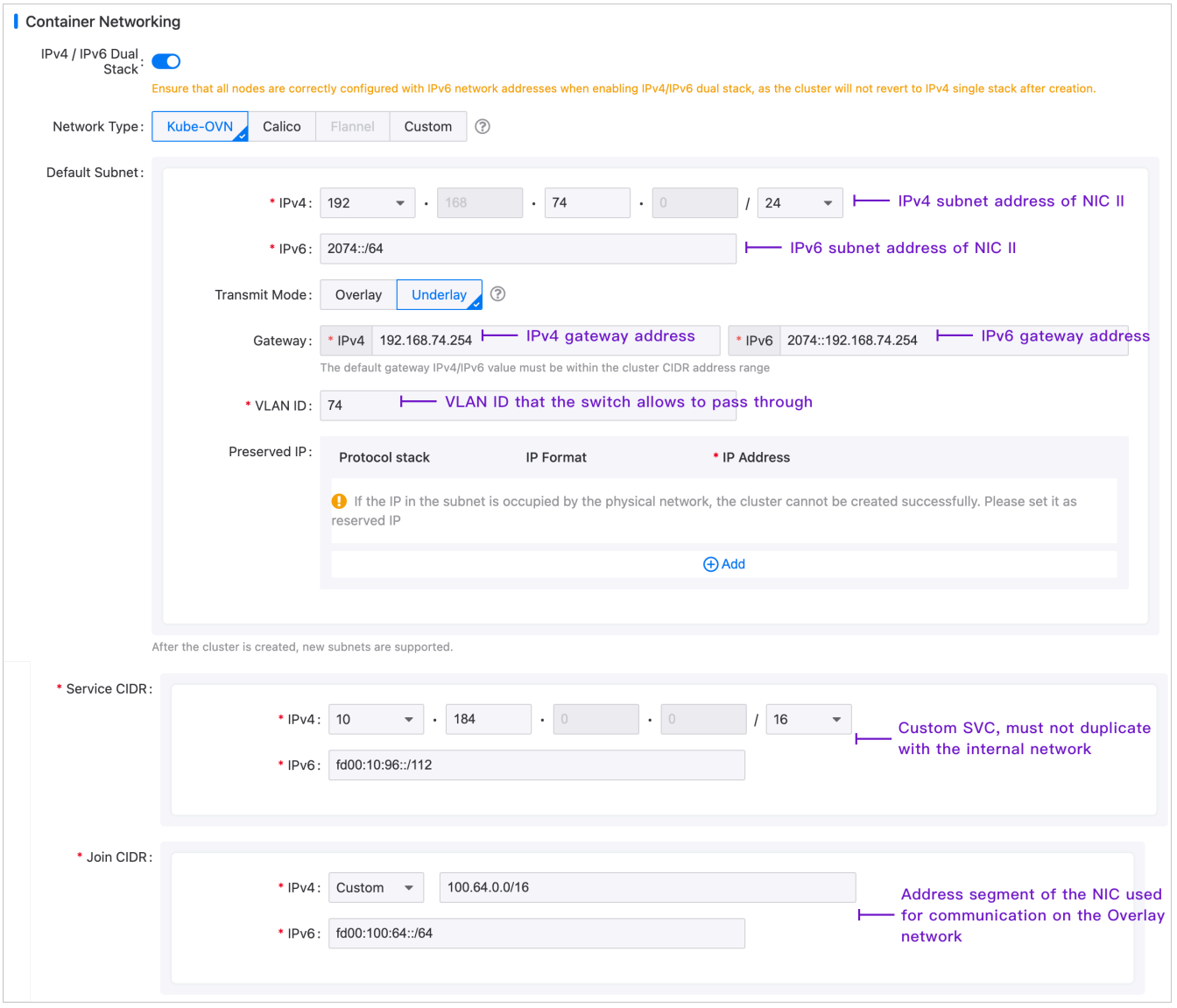
Test if NIC Two can communicate with the gateway address:

```
ip link add ens224.74 link ens224 type vlan id 74  // The NIC name is ens
224, and the VLAN ID is 74
ip link set ens224.74 up
ip addr add 192.168.74.200/24 dev ens224.74  // Select a test address wit
hin the Underlay subnet, here it's 192.168.74.200/24
ping 192.168.74.254  // If able to ping the gateway, it confirms that the
physical environment meets deployment requirements
ip addr del 192.168.74.200/24 dev ens224.74  // Delete the test address a
fter testing
ip link del ens224.74  // Delete the sub-interface after testing
```

Platform Configuration

In the left navigation bar, click **Cluster Management > Cluster**, then click **Create Cluster**. For specific configuration procedures, please refer to the Create Cluster document, with container network configuration shown in the image below.

Note: The Join subnet has no practical significance in the Underlay environment and primarily serves to create an Overlay subnet later, providing the IP address range necessary for communication between nodes and container groups.



Kube-OVN Underlay + MetalLB LoadBalancer Service Configuration

TOC

Overview

This solution addresses the integration of MetalLB L2 mode with Kube-OVN Underlay networking. It allows users to utilize Underlay subnet IPs as MetalLB LoadBalancer Service

VIPs, directly forwarding traffic to backend business Pods.

> ⚠️ **Critical**: The LoadBalancer VIP and the backend Pod IPs **must be in the same Underlay subnet**.

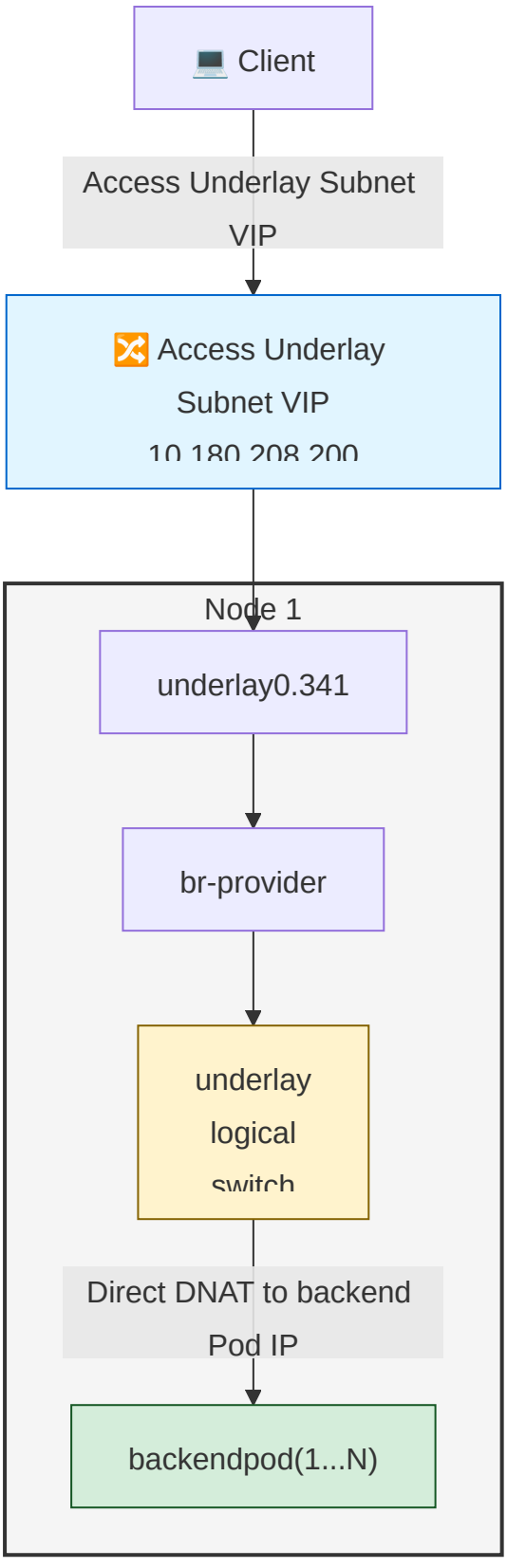
Prerequisites

Environment Requirements

- **ACP version**: >= 4.2.2
- **IP support**: IPv4 only (IPv6 is currently not supported)

Traffic Flow

Traffic Diagram:



Configuration Steps

1. Configure ProviderNetwork with VLAN Sub-interfaces

Important: VLAN sub-interfaces must be used.

Configure Kube-OVN Underlay network to automatically create VLAN sub-interfaces:

```yaml
apiVersion: kubeovn.io/v1
kind: ProviderNetwork
metadata:
  name: provider
spec:
  defaultInterface: underlay0.341
  autoCreateVlanSubinterfaces: true  # Automatically creates VLAN sub-int
erfaces (e.g., underlay0.341) if only parent interface (underlay0) exists

---
apiVersion: kubeovn.io/v1
kind: Vlan
metadata:
  name: ovn-vlan
spec:
  id: 0    # Use 0 because autoCreateVlanSubinterfaces creates the VLAN s
ub-interface (underlay0.341) which handles VLAN tagging, not Kube-OVN dir
ectly
  provider: provider
status:
  subnets:
  - ovn-default
```

> ⚠️ **Warning**: When modifying the `ProviderNetwork` or `Vlan` resources individually, the Underlay network connectivity will be interrupted. Network connectivity will only be restored after both resources are fully configured and in sync. Plan configuration changes during maintenance windows to minimize service disruption.

2. Configure Kube-OVN Controller Parameters

Configure the Kube-OVN controller with the required parameters for LoadBalancer functionality:

Using Web Console:

1. Navigate to **Administrator** > **Marketplace** > **Cluster Plugins**, then search for `ovn` to locate **Alauda Container Platform Networking for Kube-OVN**
2. In the plugin row, click the action menu (vertical ⋮) and select **Update** to open the configuration dialog
3. Configure the following settings:
   - **Skip CT for Dst LPort IPs**: **No**
   - **Enable OVN LB Local**: **Yes**

3. Configure Underlay Subnet External Address Feature

Edit the Underlay subnet to reserve an IP range for LoadBalancer usage:

> **Important**: External address pool IPs must be within the Underlay subnet.

Modify the Underlay subnet parameter `spec.enableExternalLBAddress: true`:

```
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: underlay-subnet
spec:
  enableExternalLBAddress: true       # Indicates this subnet has IP rang
e for LB service VIP
  excludeIps:
  - 10.180.208.200..10.180.208.220    # Reserve IP range for external add
ress pool
```

4. Create MetalLB External Address Pool

```yaml
# underlay-ippool.yaml
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: acp-underlay-pool
  namespace: metallb-system
spec:
  addresses:
    - 10.180.208.200-10.180.208.220  # Underlay subnet IP range
  avoidBuggyIPs: true
  autoAssign: true
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: acp-underlay-pool
  namespace: metallb-system
spec:
  ipAddressPools:
    - acp-underlay-pool
  interfaces:
    - br-provider  # Optional: Interface for ARP broadcasting; use bridge interface (br-*) instead of physical interface
  nodeSelectors: []
```

Deploy the address pool:

```bash
kubectl apply -f underlay-ippool.yaml
```

5. Create Sample Application and LoadBalancer Service

```yaml
# application-with-loadbalancer.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-app
  labels:
    app: backend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: backend
          image: nginx:1.25
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: backend-lb-service
  # To use specific IPPool: add annotation `metallb.io/address-pool: acp-underlay-pool`
  # To use fixed IP: set `spec.loadBalancerIP: 10.180.208.201`
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local  # **IMPORTANT**: Required for preserving source IP and enabling direct Pod routing
  selector:
    app: backend
  ports:
    - port: 80
      targetPort: 80
```

Deploy the application:

```
kubectl apply -f application-with-loadbalancer.yaml
```

6. Verify Configuration

```
# Check service status
kubectl get svc backend-lb-service -o wide

# Test external access
curl http://10.180.208.200
```

7. Migrating Existing Services

For existing services using the old address pool (node subnet only), you can migrate them to the new Underlay address pool:

```
# Add annotation to migrate existing service
kubectl annotate service <existing-service-name> metallb.io/address-pool=
acp-underlay-pool --overwrite

# Verify the service has been assigned a new IP from the Underlay pool
kubectl get svc <existing-service-name> -o wide
```

For **new services**, add the annotation directly:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend-lb-service
  annotations:
    metallb.io/address-pool: acp-underlay-pool  # Use the Underlay addres
s pool
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  selector:
    app: backend
  ports:
    - port: 80
      targetPort: 80
```

```
# Check service status
kubectl get svc backend-lb-service -o wide

# Test external access
curl http://10.180.208.200
```

Cluster Interconnection (Alpha)

It supports configuration of cluster interconnection between clusters whose network mode is the same as Kube-OVN, so that Pods in the clusters can access each other. Cluster Interconnect Controller is an extension component provided by Kube-OVN, which is responsible for collecting network information between different clusters and connecting the networks of multiple clusters by issuing routes.

TOC

Prerequisites

- The subnet CIDRs of different clusters cannot overlap each other.
- There needs to be a set of machines that can be accessed over IP by each cluster's kube-ovn-controller to deploy controllers that interconnect across clusters.
- A set of machines that can be accessed by kube-ovn-controller per cluster via IP for cross-cluster interconnections needs to exist for each cluster to be used as gateway nodes afterward.
- This feature is only available for the default VPC, user-defined VPCs cannot use the interconnect feature.

Multi-node Kube-OVN connectivity controller was built

There are three deployment methods available: Deploy deployment (supported in platform v3.16.0 and later versions), Podman deployment, and Containerd deployment.

Deploy Deployment

Note: This deployment method is supported in platform v3.16.0 and later versions.

Operation Steps

1. Execute the following command on the cluster Master node to obtain the `install-ic-server.sh` installation script from the kube-ovn-controller Pod.

   ```
   kubectl -n kube-system cp $(kubectl get pods -n kube-system -l app=kube-ovn-controller -o custom-columns=NAME:.metadata.name --no-headers | head -1):/kube-ovn/install-ic-server.sh ./install-ic-server.sh
   ```

2. Open the script file in the current directory and modify the parameters as follows.

```
REGISTRY="kubeovn"
VERSION=""
```

Modified parameter configurations are as follows:

```
REGISTRY="<Kube-OVN image repository address>"   ## For example: REGIST
RY="registry.alauda.cn:60080/acp/"
VERSION="<Kube-OVN version>"   ## For example: VERSION="v1.9.25"
```

3. Save the script file and execute it using the following command.

```
sh install-ic-server.sh
```

Podman and Containerd Deployment

1. Select **three or more nodes in any cluster** to deploy the Interconnected Controller. In this example, three nodes are prepared.
2. Choose any node as the Leader and execute the following commands according to the different deployment methods.
   Note: Before configuration, please check if there is an ovn directory under `/etc`. If not, use the command `mkdir /etc/ovn` to create one.

   - **Commands for container deployment Note**: Execute the command `podman images | grep ovn` to obtain the Kube-OVN image address.

     - Command for the Leader node:

```
podman run \
--name=ovn-ic-db \
-d \
--env "ENABLE_OVN_LEADER_CHECK=false" \
--network=host \
--restart=always \
--privileged=true \
-v /etc/ovn/:/etc/ovn \
-v /var/run/ovn:/var/run/ovn \
-v /var/log/ovn:/var/log/ovn \
-e LOCAL_IP="<IP address of the current node>" \   ## For example:
-e LOCAL_IP="192.168.39.37"
-e NODE_IPS="<IP addresses of all nodes, separated by commas>" \
## For example: -e NODE_IPS="192.168.39.22,192.168.39.24,192.168.3
9.37"
<image repository address> bash start-ic-db.sh    ## For example:
192.168.39.10:60080/acp/kube-ovn:v1.8.8 bash start-ic-db.sh
```

- Commands for the other two nodes:

```
podman run \
--name=ovn-ic-db \
-d \
--env "ENABLE_OVN_LEADER_CHECK=false" \
--network=host \
--restart=always \
--privileged=true \
-v /etc/ovn/:/etc/ovn \
-v /var/run/ovn:/var/run/ovn \
-v /var/log/ovn:/var/log/ovn \
-e LOCAL_IP="<IP address of the current node>" \   ## For example:
-e LOCAL_IP="192.168.39.24"
-e LEADER_IP="<IP address of the Leader node>" \  ## For example:
-e LEADER_IP="192.168.39.37"
-e NODE_IPS="<IP addresses of all nodes, separated by commas>" \
## For example: -e NODE_IPS="192.168.39.22,192.168.39.24,192.168.3
9.37"
<image repository address> bash start-ic-db.sh   ## For example: 1
92.168.39.10:60080/acp/kube-ovn:v1.8.8  bash start-ic-db.sh
```

- **Commands for Containerd deployment**

Note: Execute the command `crictl images | grep ovn` to obtain the Kube-OVN image address.

- Command for the Leader node:

```
ctr -n k8s.io run \
-d \
--env "ENABLE_OVN_LEADER_CHECK=false" \
--net-host \
--privileged \
--mount="type=bind,src=/etc/ovn/,dst=/etc/ovn,options=rbind:rw" \
--mount="type=bind,src=/var/run/ovn,dst=/var/run/ovn,options=rbin
d:rw" \
--mount="type=bind,src=/var/log/ovn,dst=/var/log/ovn,options=rbin
d:rw" \
--env="NODE_IPS=<IP addresses of all nodes, separated by commas>"
\   ## For example: --env="NODE_IPS="192.168.178.97,192.168.181.9
3,192.168.177.192""
--env="LOCAL_IP=<IP address of the current node>" \   ## For examp
le: --env="LOCAL_IP="192.168.178.97""
<image repository address> ovn-ic-db bash start-ic-db.sh   ## For
example: registry.alauda.cn:60080/acp/kube-ovn:v1.9.25 ovn-ic-db b
ash start-ic-db.sh
```

- Commands for the other two nodes:

```
ctr -n k8s.io run \
-d \
--env "ENABLE_OVN_LEADER_CHECK=false" \
--net-host \
--privileged \
--mount="type=bind,src=/etc/ovn/,dst=/etc/ovn,options=rbind:rw" \
--mount="type=bind,src=/var/run/ovn,dst=/var/run/ovn,options=rbin
d:rw" \
--mount="type=bind,src=/var/log/ovn,dst=/var/log/ovn,options=rbin
d:rw" \
--env="NODE_IPS=<IP addresses of all nodes, separated by commas>"
\   ## For example: --env="NODE_IPS="192.168.178.97,192.168.181.9
3,192.168.177.192"" \
--env="LOCAL_IP=<IP address of the current node>" \   ## For examp
le: --env="LOCAL_IP="192.168.181.93""
--env="LEADER_IP=<IP address of the Leader node>" \   ## For examp
le: --env="LEADER_IP="192.168.178.97""
<image repository address> ovn-ic-db bash start-ic-db.sh   ## For
example: registry.alauda.cn:60080/acp/kube-ovn:v1.9.25 ovn-ic-db b
ash start-ic-db.sh
```

Deploy the cluster interconnection controller in the Global cluster

In any control node of global, replace the following parameters according to the comments and execute the following command to create the ConfigMap resource.

Note: To ensure the correct operation, the ConfigMap named ovn-ic on global is not allowed to be modified. If any parameter needs to be changed, please delete the ConfigMap and reconfigure it correctly before applying the ConfigMap.

```
cat << EOF |kubectl apply -f -
apiVersion: v1
kind: ConfigMap
metadata:
  name: ovn-ic
  namespace: cpaas-system
data:
  ic-db-host: "192.168.39.22,192.168.39.24,192.168.39.37"   # Address of the node where the cluster interconnect controller is located, in this case, the local IP of the three nodes where the controller is deployed
  ic-nb-port: "6645"            # Cluster Interconnect Controller nb port, default 6645
  ic-sb-port: "6646"            # Cluster Interconnect Controller sb port, default 6646
EOF
```

Join the cluster interconnect

Add a cluster whose network mode is Kube-OVN to the cluster interconnect.

Prerequisites

The **created subnets**, **ovn-default**, and **join subnets** in a cluster do not conflict with any cluster segment in the cluster interconnection group.

Procedure of operation

1. In the left navigation bar, click **Clusters** > **Cluster of clusters**.
2. Click the name of the ***cluster*** to be added to the cluster interconnect.
3. In the upper right corner, click **Options** > **Cluster Interconnect**.
4. Click **Join the cluster interconnect**.
5. Select a gateway node for the cluster.
6. Click **Join**.

Relevant operations

Update the gateway node information of the interconnected cluster

Update information about cluster gateway nodes that have joined a cluster interconnect group.

Procedure of operation

1. In the left navigation bar, click **Clusters** > **Cluster of clusters**.
2. Click ***Cluster name*** for the gateway node information to be updated.
3. In the upper-right corner, click **Operations** > **Cluster Interconnect**.
4. Click **Update Gateway Node** for the cluster whose gateway node information you want to update.
5. Reselect the gateway node for the cluster.
6. Click **Update**.

Exit cluster interconnection

A cluster that has joined a cluster interconnection group exits cluster interconnection, and when it does, it disconnects the cluster Pod from the external cluster Pod.

Procedure of operation

1. In the left navigation bar, click **Clusters** > **Cluster of clusters**.
2. Click the name of the ***cluster*** that you want to decommission.
3. In the upper-right corner, click **Options** > **Cluster Interconnect**.
4. Click **Exit cluster interconnection** for the cluster you want to exit.
5. Enter the cluster name correctly.
6. Click **Exit**.

Cleaning up Interconnected Cluster Residue

When a cluster is deleted without leaving the interconnected cluster, some residual data may remain on the controller. When you attempt to use these nodes to create a cluster again and

join the interconnected cluster, failures may occur. You can check the detailed error information in the `/var/log/ovn/ovn-ic.log` log of the controller (kube-ovn-controller). Some error messages may include:

```
transaction error: {"details":"Transaction causes multiple rows in xxxxx
x"}
```

Operational Steps

1. Exit the interconnected cluster for the cluster to be joined.
2. Execute the cleanup script in the container or pod.
   You can execute the cleanup script directly in either the ovn-ic-db container or the ovn-ic-controller pod. Choose one of the following methods:
   Method 1: Execute in ovn-ic-db container

   - Enter the ovn-ic-db container and perform the cleanup operation with the following commands.

     ```
     ctr -n k8s.io task exec -t --exec-id ovn-ic-db ovn-ic-db /bin/bash
     ```

     Then execute one of the following cleanup commands:

     - Execute the cleanup operation with the name of the original cluster. Replace *<cluster-name>* with the **name of the original cluster**:

       ```
       ./clean-ic-az-db.sh <cluster-name>
       ```

     - Execute the cleanup operation with the name of any node in the original cluster. Replace *<node-name>* with the **name of any node in the original cluster**:

       ```
       ./clean-ic-az-db.sh <node-name>
       ```

   Method 2: Execute in ovn-ic-controller pod

   - Enter the ovn-ic-controller pod and perform the cleanup operation with the following commands.

```
kubectl -n kube-system exec -ti $(kubectl get pods -n kube-system -l app=ovn-ic-controller -o custom-columns=NAME:.metadata.name --no-headers) -- /bin/bash
```

Then execute one of the following cleanup commands:

- Execute the cleanup operation with the name of the original cluster. Replace <*cluster-name*> with the **name of the original cluster**:

  ```
  ./clean-ic-az-db.sh <cluster-name>
  ```

- Execute the cleanup operation with the name of any node in the original cluster. Replace <*node-name*> with the **name of any node in the original cluster**:

  ```
  ./clean-ic-az-db.sh <node-name>
  ```

Uninstalling the Interconnected Cluster

Note: Step 1 to Step 3 need to be performed on all **business clusters that have joined the interconnected cluster**.

Operational Steps

1. Exit the interconnected cluster. There are two specific exit methods, choose one according to your needs.

   - Delete the ConfigMap named ovn-ic-config in the business cluster. Use the following command.

     ```
     kubectl -n kube-system delete cm ovn-ic-config
     ```

   - Exit the interconnected cluster through platform operations.

2. Enter the Leader Pod of ovn-central with the following command.

```
kubectl -n kube-system exec -ti $(kubectl get pods -n kube-system -lovn
-nb-leader=true -o custom-columns=NAME:.metadata.name --no-headers) --
/bin/bash
```

3. Clean up the ts logical switch with the following command.

```
ovn-nbctl ls-del ts
```

4. Log in to the node where the controller is deployed and delete the controller.

   - Podman command:

   ```
   podman stop ovn-ic-db
   podman rm ovn-ic-db
   ```

   - Containerd command:

   ```
   ctr -n k8s.io task kill ovn-ic-db
   ctr -n k8s.io containers rm ovn-ic-db
   ```

5. Delete the ConfigMap named ovn-ic in the global cluster with the following command.

```
kubectl delete cm ovn-ic -n cpaas-system
```

Configure Cluster Gateway High Availability

To configure the cluster gateway to be highly available after joining the cluster interconnection, you can perform the following steps:

1. Log in to the cluster that needs to be transformed into a High Availability Gateway and execute the following command to change the `enable-ic` field to `false`.
   Note: Changing the `enable-ic` field to `false` will disrupt the cluster interconnect until it is set to `true` again.

```
kubectl edit cm ovn-ic-config -n kube-system
```

2. Modify the gateway node configuration by updating the `gw-nodes` field and separating the gateway nodes with English commas; also change the `enable-ic` field to `true`.

```
kubectl edit cm ovn-ic-config -n kube-system

# Configuration example
apiVersion: v1
data:
  auto-route: "true"
  az-name: az1
  enable-ic: "true"
  gw-nodes: 192.168.188.234,192.168.189.54
  ic-db-host: 192.168.178.97
  ic-nb-port: "6645"
  ic-sb-port: "6646"
kind: ConfigMap
metadata:
  creationTimestamp: "2023-06-13T08:01:16Z"
  name: ovn-ic-config
  namespace: kube-system
  resourceVersion: "99671"
  uid: 6163790a-ad9d-4d07-ba82-195b11244983
```

3. Go to the Pod in cluster ovn-central and execute the `ovn-nbctl lrp-get-gateway-chassis {current cluster name}-ts` command to verify that the configuration is in effect.

```
ovn-nbctl lrp-get-gateway-chassis az1-ts

# Return to the display example. In this case, the values of 100 and 99 are the priority, and the larger the value, the higher the priority of the corresponding gateway node to be used.
az1-ts-71292a21-131d-492a-9f0c-0611af458950 100
az1-ts-1de7ee15-f372-4ab9-8c85-e54d61ea18f1 99
```

Configure Egress Gateway

TOC

Overview

Egress Gateway, also called VPC Egress Gateway, provides stable outbound addresses for Pods in an overlay network. It routes selected workloads through dedicated gateway Pods before the traffic leaves the cluster.

Use Egress Gateway when you need:

- Stable source IP addresses for specific workloads
- Workload-level egress control instead of subnet-level control
- Higher throughput through horizontal scaling
- Faster failover for outbound traffic

Main capabilities:

- Active-Active high availability through ECMP, with horizontal throughput scaling
- Fast failover through BFD, typically in less than 1 second
- Support for IPv4, IPv6, and dual-stack environments
- Fine-grained traffic matching through NamespaceSelector and PodSelector
- Flexible scheduling through node selectors and tolerations

Current limitations:

- Multi-replica deployments require multiple egress IPs
- Source NAT mapping records are not retained

Egress Gateway vs. Centralized Gateway

Use this section to choose the gateway model that best fits your scenario. For centralized mode details, refer to Configure Centralized Gateway.

Dimension	Egress Gateway	Centralized Gateway
Granularity	Fine-grained control by `selectors` and `policies`	Applied at subnet level (`gatewayType:`

Dimension	Egress Gateway	Centralized Gateway
	(Namespace/Pod/Subnet/IPBlock).	`centralized`).
Egress Address Source	Uses dedicated gateway Pods with external subnet IPs.	Uses designated gateway nodes; with `natOutgoing: true`, egress uses node IPs.
Data Path	Traffic is routed to VPC Egress Gateway Pods, then SNATed to the external network.	Traffic is routed to designated nodes (`ovn0`) and then forwarded by host routing/NAT rules.
HA and Failover	Active-Active ECMP; supports BFD fast failover (sub-second).	Supports ECMP and primary-backup; failover is generally slower than BFD-based Egress Gateway.
Scheduling Control	Supports node scheduling controls for gateway workloads (`nodeSelector`, `tolerations`).	Gateway nodes are selected by `gatewayNode` or `gatewayNodeSelectors`.
Typical Use Cases	Tenant-level or workload-level egress isolation, fine-grained policy control, higher performance and faster failover.	Simpler subnet-level fixed-source egress, auditing, IP allowlists, and firewall source-IP management.

Selection guidance:

- Choose **Egress Gateway** when you need workload-level policy control, scalable throughput, and fast failover.
- Choose **Centralized Gateway** when subnet-level fixed egress and simple operations are sufficient.

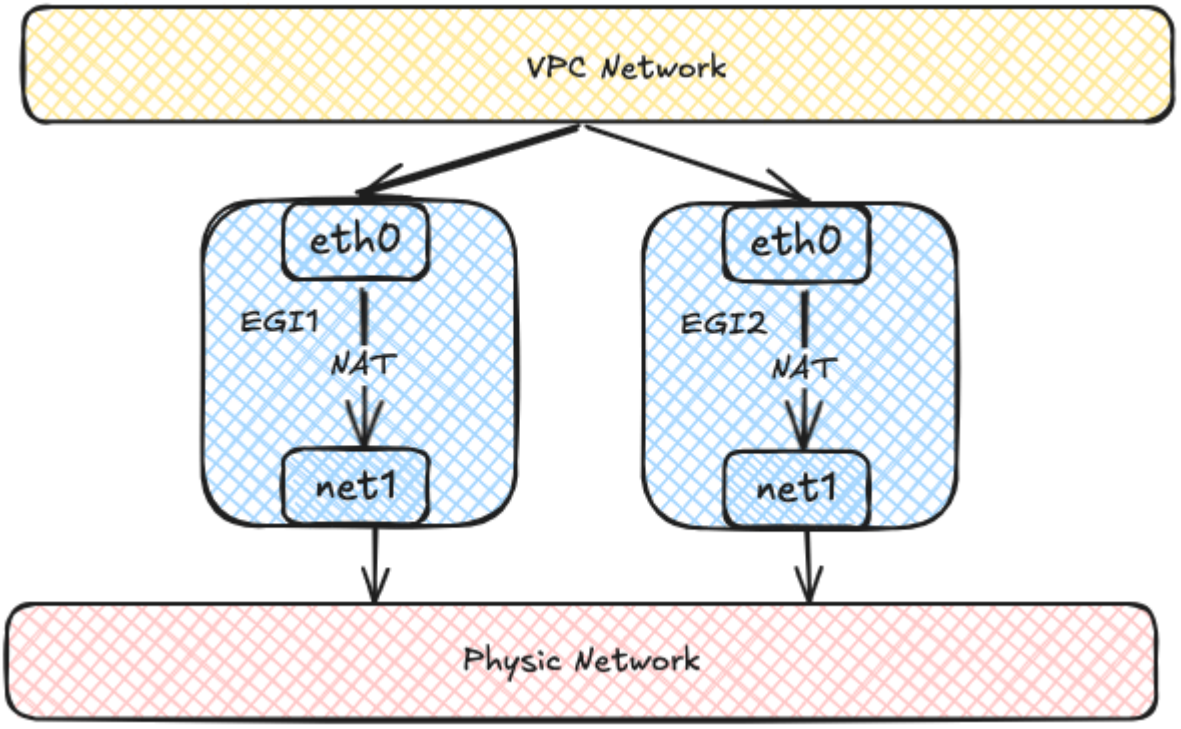
How Egress Gateway Works

An Egress Gateway runs as one or more Pods. Each Pod uses two network interfaces:

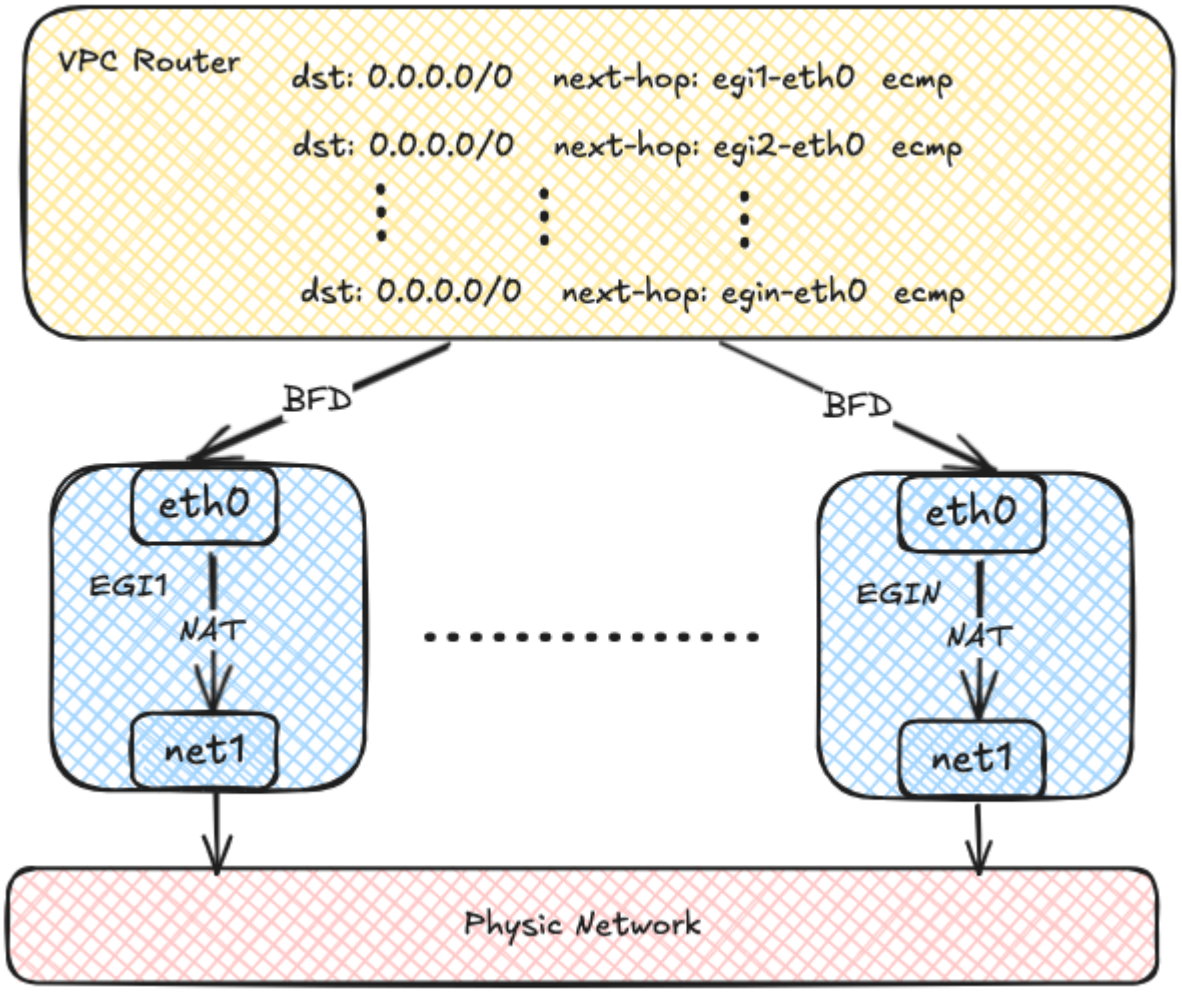
- One interface connects to the overlay network inside the cluster.

- The other interface connects to the external underlay network.

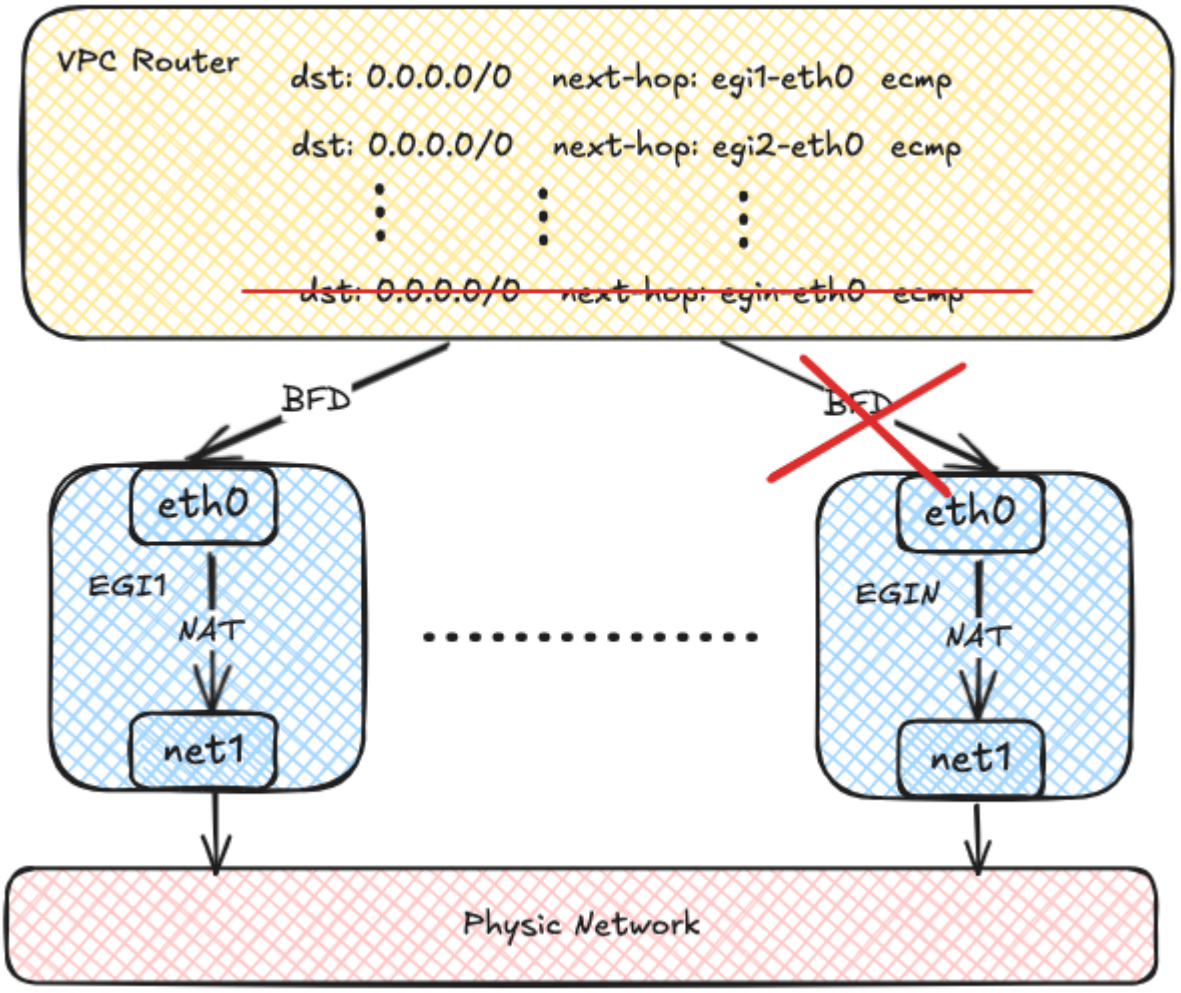
Traffic from selected Pods is first forwarded to the gateway Pods and then sent to the external network through the underlay interface.



Each Egress Gateway instance registers its address in the OVN routing table. When a Pod in the overlay network accesses the external network, OVN uses source-address hashing to distribute traffic across multiple gateway instances. This provides load balancing and lets throughput scale horizontally as more instances are added.



OVN can use BFD to probe multiple gateway instances. If one instance fails, OVN marks the corresponding route as unavailable and quickly redirects traffic to a healthy instance.



Before You Begin

Before you start, make sure the following prerequisites are met:

- *Multus CNI Plugin* **MUST** already be installed on the cluster.
- The external underlay network, VLAN, and bridge network are already planned and available.
- You have identified which workloads should use the gateway and whether they require SNAT.

To install *Multus CNI Plugin*, refer to Deploying the Multus CNI Plugin.

Configuration Workflow

Configure Egress Gateway in the following order:

1. Prepare the external network attachment, including the subnet and Network Attachment Definition.
2. Create a VPC Egress Gateway resource and specify the external subnet and policies.
3. Verify resource status, routes, and traffic forwarding.
4. Optionally scale out replicas and enable BFD-based fast failover.

Step 1: Prepare the External Network Attachment

Egress Gateway uses multiple interfaces to connect to both the internal and external networks. Before creating the gateway, prepare the following resources:

- An external subnet
- A Network Attachment Definition (NAD) for that subnet

The following example uses a Kube-OVN underlay subnet as the external network.

> **NOTE**
>
> This example assumes that the bridge network and VLAN resource `external-vlan` have already been created for the underlay network.

```yaml
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  name: underlay-ext
  namespace: default
spec:
  config: |-
    {
      "cniVersion": "0.3.0",
      "type": "kube-ovn", 1
      "server_socket": "/run/openvswitch/kube-ovn-daemon.sock", 2
      "provider": "underlay-ext.default.ovn" 3
    }
---
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: underlay-ext
spec:
  protocol: IPv4
  provider: underlay-ext.default.ovn 4
  cidrBlock: 172.17.0.0/16 5
  gateway: 172.17.0.1 6
  vlan: external-vlan 7
  excludeIps: 8
    - 172.17.0.11..172.17.0.20
```

1 Use the Kube-OVN CNI plugin for the secondary network.

2 Kube-OVN daemon socket used by the CNI plugin.

3 Provider name in the format `<network attachment definition name>.<namespace>.ovn`.

4 The provider used by the subnet. This value MUST match the provider in the NetworkAttachmentDefinition.

5 CIDR of the external underlay network.

6 Gateway of the external underlay network.

7 VLAN resource used by the underlay subnet.

8 IP range excluded from automatic allocation. For details, refer to Example Subnet custom resource (CR) with Kube-OVN Underlay Network.

TIP

Before using an underlay subnet, make sure the physical network, VLAN, and bridge network are prepared correctly. For environment planning details, refer to Preparing Kube-OVN Underlay Physical Network.

Step 2: Create a VPC Egress Gateway

Create a VPC Egress Gateway resource and define which workloads should use it.

```yaml
apiVersion: kubeovn.io/v1
kind: VpcEgressGateway
metadata:
  name: gateway1
  namespace: default (1)
spec:
  replicas: 1 (2)
  internalSubnet: ovn-default (3)
  externalSubnet: underlay-ext (4)
  externalIPs: (5)
    - 172.17.0.11
    - 172.17.0.12
  resources: (6)
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 200m
      memory: 256Mi
      ephemeral-storage: 2Gi
  nodeSelector: (7)
    - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
            - node1
            - node2
  tolerations: (8)
    - key: node-role.kubernetes.io/control-plane
      operator: Exists
      effect: NoSchedule
  selectors: (9)
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: ns1
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: ns2
      podSelector:
        matchLabels:
          app: myapp
  policies: (10)
    - snat: true (11)
```

```
      subnets: 12
        - subnet1
    - snat: false
      ipBlocks: 13
        - 10.18.0.0/16
```

1 Namespace where the VPC Egress Gateway instances are created.

2 Number of VPC Egress Gateway instances.

3 Internal subnet that connects to the internal network. The subnet MUST be an overlay subnet in the same VPC and have enough free IPs for the gateway instances. If not specified, the gateway Pods will use the default internal subnet of the VPC.

4 External subnet that connects to the external network.

5 External IPs used by the gateway Pods on the underlay network. Each gateway instance is allocated one IP from this list. These IPs MUST be within the CIDR of the external subnet and should be included in the `excludeIps` range of the subnet. It's recommended to reserve *.spec.replicas + 1* IPs so that a gateway Pod can still obtain an IP in edge cases.

6 Resource requests and limits for each VPC Egress Gateway instance. If not specified, the default resource requests and limits defined in the VPC Egress Gateway controller will be applied.

7 Node selectors used for scheduling the VPC Egress Gateway instances.

8 Tolerations used for scheduling the VPC Egress Gateway instances.

9 Namespace selectors and Pod selectors used to select Pods that access the external network via the VPC Egress Gateway.

10 Policies for the VPC Egress Gateway, including SNAT and subnets/ipBlocks to be applied.

11 Whether to enable SNAT for the policy.

12 Subnets to which the policy applies.

13 IP blocks to which the policy applies.

This example creates a VPC Egress Gateway named *gateway1* in the `default` namespace. Traffic that matches the selectors and policies is forwarded through the external subnet *underlay-ext*. In this example, that includes:

- Pods in the *ns1* namespace
- Pods in the *ns2* namespace with the label `app: myapp`

- Traffic related to the *subnet1* subnet
- Traffic related to the CIDR *10.18.0.0/16*

> **NOTE**
>
> Pods matching *.spec.selectors* are always SNATed by the gateway.

Step 3: Validate the Gateway

After creating the gateway, confirm that it is ready and forwarding traffic as expected.

1. Check the resource status

Start with the basic resource status:

```
$ kubectl get veg gateway1
NAME       VPC           REPLICAS   BFD ENABLED   EXTERNAL SUBNET   PHASE
READY   AGE
gateway1   ovn-cluster   1          false         underlay-ext      Compl
eted   true    13s
```

Then check the detailed gateway information:

```
kubectl get veg gateway1 -o wide
NAME       VPC           REPLICAS   BFD ENABLED   EXTERNAL SUBNET   PHASE
READY   INTERNAL IPS     EXTERNAL IPS      WORKING NODES   AGE
gateway1   ovn-cluster   1          false         underlay-ext      Compl
eted   true    ["10.16.0.12"]   ["172.17.0.11"]   ["node1"]       82s
```

Finally, verify that the gateway workloads are running:

```
$ kubectl get deployment -n default -l ovn.kubernetes.io/vpc-egress-gatew
ay=gateway1
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
gateway1   1/1     1            1           4m40s

$ kubectl get pod -n default -l ovn.kubernetes.io/vpc-egress-gateway=gate
way1 -o wide
NAME                        READY   STATUS    RESTARTS   AGE     IP
NODE    NOMINATED NODE   READINESS GATES
gateway1-b9f8b4448-76lhm   1/1     Running   0          4m48s   10.16.0.1
2   node1   <none>           <none>
```

2. Inspect networking inside the gateway Pod

Inspect IP addresses, routing entries, and iptables rules inside the gateway Pod:

```
$ kubectl exec -n default gateway1-b9f8b4448-76lhm -c gateway -- ip addre
ss show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: net1@if13: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue st
ate UP group default qlen 1000
    link/ether 62:d8:71:90:7b:86 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.11/16 brd 172.17.255.255 scope global net1
       valid_lft forever preferred_lft forever
    inet6 fe80::60d8:71ff:fe90:7b86/64 scope link
       valid_lft forever preferred_lft forever
17: eth0@if18: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc noqueue s
tate UP group default
    link/ether 36:7c:6b:c7:82:6b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.16.0.12/16 brd 10.16.255.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::347c:6bff:fec7:826b/64 scope link
       valid_lft forever preferred_lft forever

$ kubectl exec -n default gateway1-b9f8b4448-76lhm -c gateway -- ip rule
show
0:      from all lookup local
1001:   from all iif eth0 lookup default
1002:   from all iif net1 lookup 1000
1003:   from 10.16.0.12 iif lo lookup 1000
1004:   from 172.17.0.11 iif lo lookup default
32766:  from all lookup main
32767:  from all lookup default

$ kubectl exec -n default gateway1-b9f8b4448-76lhm -c gateway -- ip route
show
default via 172.17.0.1 dev net1
10.16.0.0/16 dev eth0 proto kernel scope link src 10.16.0.12
10.17.0.0/16 via 10.16.0.1 dev eth0
10.18.0.0/16 via 10.16.0.1 dev eth0
172.17.0.0/16 dev net1 proto kernel scope link src 172.17.0.11

$ kubectl exec -n default gateway1-b9f8b4448-76lhm -c gateway -- ip route
```

```
show table 1000
default via 10.16.0.1 dev eth0

$ kubectl exec -n default gateway1-b9f8b4448-76lhm -c gateway -- iptables
-t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N VEG-MASQUERADE
-A PREROUTING -i eth0 -j MARK --set-xmark 0x4000/0x4000
-A POSTROUTING -d 10.18.0.0/16 -j RETURN
-A POSTROUTING -s 10.18.0.0/16 -j RETURN
-A POSTROUTING -j VEG-MASQUERADE
-A VEG-MASQUERADE -j MARK --set-xmark 0x0/0xffffffff
-A VEG-MASQUERADE -j MASQUERADE --random-fully
```

3. Confirm traffic forwarding

Capture packets in the gateway Pod to confirm that traffic is forwarded through the gateway:

```
$ kubectl exec -n default gateway1-b9f8b4448-76lhm -c gateway -- tcpdump
-i any -nnve icmp and host 172.17.0.1
tcpdump: data link type LINUX_SLL2
tcpdump: listening on any, link-type LINUX_SLL2 (Linux cooked v2), snapsh
ot length 262144 bytes
06:50:58.936528 eth0  In  ifindex 17 92:26:b8:9e:f2:1c ethertype IPv4 (0x
0800), length 104: (tos 0x0, ttl 63, id 30481, offset 0, flags [DF], prot
o ICMP (1), length 84)
    10.17.0.9 > 172.17.0.1: ICMP echo request, id 37989, seq 0, length 64
06:50:58.936574 net1  Out ifindex 2 62:d8:71:90:7b:86 ethertype IPv4 (0x0
800), length 104: (tos 0x0, ttl 62, id 30481, offset 0, flags [DF], proto
ICMP (1), length 84)
    172.17.0.11 > 172.17.0.1: ICMP echo request, id 39449, seq 0, length
64
06:50:58.936613 net1  In  ifindex 2 02:42:39:79:7f:08 ethertype IPv4 (0x0
800), length 104: (tos 0x0, ttl 64, id 26701, offset 0, flags [none], pro
to ICMP (1), length 84)
    172.17.0.1 > 172.17.0.11: ICMP echo reply, id 39449, seq 0, length 64
06:50:58.936621 eth0  Out ifindex 17 36:7c:6b:c7:82:6b ethertype IPv4 (0x
0800), length 104: (tos 0x0, ttl 63, id 26701, offset 0, flags [none], pr
oto ICMP (1), length 84)
    172.17.0.1 > 10.17.0.9: ICMP echo reply, id 37989, seq 0, length 64
```

4. Confirm OVN routing policies

OVN Logical Router policies are created automatically for the selected traffic:

```
$ kubectl ko nbctl lr-policy-list ovn-cluster
Routing Policies
     31000                             ip4.dst == 10.16.0.0/16   allow
     31000                             ip4.dst == 10.17.0.0/16   allow
     31000                           ip4.dst == 100.64.0.0/16   allow
     30000                              ip4.dst == 172.18.0.2  reroute  1
00.64.0.4
     30000                              ip4.dst == 172.18.0.3  reroute  1
00.64.0.3
     30000                              ip4.dst == 172.18.0.4  reroute  1
00.64.0.2
     29100                    ip4.src == $VEG.8ca38ae7da18.ipv4  reroute  1
0.16.0.12 1
     29100                     ip4.src == $VEG.8ca38ae7da18_ip4  reroute  1
0.16.0.12 2
     29000 ip4.src == $ovn.default.kube.ovn.control.plane_ip4  reroute  1
00.64.0.3
     29000       ip4.src == $ovn.default.kube.ovn.worker2_ip4  reroute  1
00.64.0.2
     29000        ip4.src == $ovn.default.kube.ovn.worker_ip4  reroute  1
00.64.0.4
     29000     ip4.src == $subnet1.kube.ovn.control.plane_ip4  reroute  1
00.64.0.3
     29000           ip4.src == $subnet1.kube.ovn.worker2_ip4  reroute  1
00.64.0.2
     29000            ip4.src == $subnet1.kube.ovn.worker_ip4  reroute  1
00.64.0.4
```

1 Logical Router Policy used by the VPC Egress Gateway to forward traffic matched by *.spec.policies*.

2 Logical Router Policy used by the VPC Egress Gateway to forward traffic matched by *.spec.selectors*.

Optional: Enable Multi-Replica Load Balancing

> **NOTE**
>
> Before scaling out replicas, make sure to prepare enough external IPs in the external subnet and specify them in *.spec.externalIPs*.

To enable ECMP load balancing and scale throughput horizontally, increase *.spec.replicas*:

```
$ kubectl scale veg -n default gateway1 --replicas=2
vpcegressgateway.kubeovn.io/gateway1 scaled

$ kubectl get veg -n default gateway1
NAME       VPC           REPLICAS   BFD ENABLED   EXTERNAL SUBNET   PHASE
READY   AGE
gateway1   ovn-cluster   2          false         underlay-ext      Compl
eted   true    39m

$ kubectl get pod -n default -l ovn.kubernetes.io/vpc-egress-gateway=gate
way1 -o wide
NAME                        READY   STATUS    RESTARTS   AGE   IP
NODE    NOMINATED NODE   READINESS GATES
gateway1-b9f8b4448-76lhm   1/1     Running   0          40m   10.16.0.12
node1   <none>           <none>
gateway1-b9f8b4448-zd4dl   1/1     Running   0          64s   10.16.0.13
node2   <none>           <none>

$ kubectl ko nbctl lr-policy-list ovn-cluster
Routing Policies
     31000                            ip4.dst == 10.16.0.0/16    allow
     31000                            ip4.dst == 10.17.0.0/16    allow
     31000                           ip4.dst == 100.64.0.0/16    allow
     30000                              ip4.dst == 172.18.0.2  reroute  1
00.64.0.4
     30000                              ip4.dst == 172.18.0.3  reroute  1
00.64.0.3
     30000                              ip4.dst == 172.18.0.4  reroute  1
00.64.0.2
     29100                  ip4.src == $VEG.8ca38ae7da18.ipv4  reroute  1
0.16.0.12, 10.16.0.13
     29100                   ip4.src == $VEG.8ca38ae7da18_ip4  reroute  1
0.16.0.12, 10.16.0.13
     29000 ip4.src == $ovn.default.kube.ovn.control.plane_ip4  reroute  1
00.64.0.3
     29000       ip4.src == $ovn.default.kube.ovn.worker2_ip4  reroute  1
00.64.0.2
     29000        ip4.src == $ovn.default.kube.ovn.worker_ip4  reroute  1
00.64.0.4
     29000     ip4.src == $subnet1.kube.ovn.control.plane_ip4  reroute  1
00.64.0.3
     29000           ip4.src == $subnet1.kube.ovn.worker2_ip4  reroute  1
00.64.0.2
```

```
      29000            ip4.src == $subnet1.kube.ovn.worker_ip4  reroute  1
00.64.0.4
```

Optional: Enable BFD-based High Availability

BFD-based failover depends on the VPC BFD LRP. Enable it in the following order.

1. Enable a BFD Port on the VPC

First, enable a BFD Port on the VPC:

```yaml
apiVersion: kubeovn.io/v1
kind: Vpc
metadata:
  name: ovn-cluster
spec:
  bfdPort:
    enabled: true (1)
    ip: 10.255.255.255 (2)
    nodeSelector: (3)
      matchLabels:
        kubernetes.io/os: linux
```

(1) Whether to enable the BFD Port.

(2) IP address of the BFD Port, which MUST be a valid IP address that does not conflict with ANY other IPs/Subnets.

(3) Node selector used to select the nodes where the BFD Port runs in Active-Backup mode.

> **TIP**
>
> The Vpc resource *ovn-cluster* exists by default. You can edit it directly to enable the BFD Port.

After the BFD Port is enabled, a dedicated BFD LRP is automatically created on the OVN Logical Router:

```
$ kubectl ko nbctl show ovn-cluster
router 0c1d1e8f-4c86-4d96-88b2-c4171c7ff824 (ovn-cluster)
    port bfd@ovn-cluster 1
        mac: "8e:51:4b:16:3c:90"
        networks: ["10.255.255.255"]
    port ovn-cluster-join
        mac: "d2:21:17:71:77:70"
        networks: ["100.64.0.1/16"]
    port ovn-cluster-ovn-default
        mac: "d6:a3:f5:31:cd:89"
        networks: ["10.16.0.1/16"]
    port ovn-cluster-subnet1
        mac: "4a:09:aa:96:bb:f5"
        networks: ["10.17.0.1/16"]
```

1 BFD Port created on the OVN Logical Router.

2. Enable BFD on the VPC Egress Gateway

Then enable BFD on the VPC Egress Gateway by setting *.spec.bfd.enabled* to *true*:

```yaml
apiVersion: kubeovn.io/v1
kind: VpcEgressGateway
metadata:
  name: gateway2
  namespace: default
spec:
  vpc: ovn-cluster (1)
  replicas: 2
  internalSubnet: ovn-default (2)
  externalSubnet: underlay-ext (3)
  externalIPs: (4)
    - 172.17.0.11
    - 172.17.0.12
    - 172.17.0.13
  bfd:
    enabled: true (5)
    minRX: 100 (6)
    minTX: 100 (7)
    multiplier: 5 (8)
  policies:
    - snat: true
      ipBlocks:
        - 10.18.0.0/16
```

(1) VPC to which the Egress Gateway belongs.

(2) Internal subnet to which the Egress Gateway instances are connected.

(3) External subnet to which the Egress Gateway instances are connected.

(4) External IPs assigned to the Egress Gateway instances.

(5) Whether to enable BFD for the Egress Gateway.

(6) Minimum receive interval for BFD, in milliseconds.

(7) Minimum transmit interval for BFD, in milliseconds.

(8) Multiplier for BFD, which determines the number of missed packets before declaring a failure.

This example creates a VPC Egress Gateway named *gateway2* with two replicas and BFD enabled. If one instance fails, the BFD session goes down, OVN marks the route as unavailable, and redirects traffic to the healthy instance.

Failover detection time depends on the BFD settings. Use the following formula: *break time* = *(multiplier + 1) * max(minRX, minTX)*. With this sample configuration, failover detection is approximately 500-600 ms.

> **NOTE**
>
> Existing connections may be interrupted during failover and require reconnection. New connections can still be established normally.

3. Verify BFD status

Check the VPC Egress Gateway status:

```
$ kubectl get veg -n default gateway2 -o wide
NAME       VPC           REPLICAS   BFD ENABLED   EXTERNAL SUBNET   PHASE
READY   INTERNAL IPS                EXTERNAL IPS                   WOR
KING NODES       AGE
gateway2   ovn-cluster   2          true          underlay-ext      Compl
eted   true    ["10.16.0.12","10.16.0.13"]   ["172.17.0.11","172.17.0.1
2"]   ["node1","node2"]   58s

$ kubectl get pod -n default -l ovn.kubernetes.io/vpc-egress-gateway=gate
way2 -o wide
NAME                        READY   STATUS    RESTARTS   AGE     IP
NODE    NOMINATED NODE   READINESS GATES
gateway2-fcc6b8b87-8lgvx   1/1     Running   0          2m18s   10.16.0.1
3   node2   <none>           <none>
gateway2-fcc6b8b87-wmww6   1/1     Running   0          2m18s   10.16.0.1
2   node1   <none>           <none>

$ kubectl ko nbctl lr-policy-list ovn-cluster
Routing Policies
     31000                            ip4.dst == 10.16.0.0/16    allow
     31000                            ip4.dst == 10.17.0.0/16    allow
     31000                           ip4.dst == 100.64.0.0/16    allow
     30000                              ip4.dst == 172.18.0.2  reroute  1
00.64.0.4
     30000                              ip4.dst == 172.18.0.3  reroute  1
00.64.0.3
     30000                              ip4.dst == 172.18.0.4  reroute  1
00.64.0.2
     29100                    ip4.src == $VEG.8ca38ae7da18.ipv4  reroute  1
0.16.0.12, 10.16.0.13  bfd
     29100                     ip4.src == $VEG.8ca38ae7da18_ip4  reroute  1
0.16.0.12, 10.16.0.13  bfd
     29090                    ip4.src == $VEG.8ca38ae7da18.ipv4     drop
     29090                     ip4.src == $VEG.8ca38ae7da18_ip4     drop
     29000 ip4.src == $ovn.default.kube.ovn.control.plane_ip4  reroute  1
00.64.0.3
     29000       ip4.src == $ovn.default.kube.ovn.worker2_ip4  reroute  1
00.64.0.2
     29000        ip4.src == $ovn.default.kube.ovn.worker_ip4  reroute  1
00.64.0.4
     29000     ip4.src == $subnet1.kube.ovn.control.plane_ip4  reroute  1
00.64.0.3
     29000           ip4.src == $subnet1.kube.ovn.worker2_ip4  reroute  1
```

```
00.64.0.2
     29000            ip4.src == $subnet1.kube.ovn.worker_ip4  reroute  1
00.64.0.4

$ kubectl ko nbctl list bfd
_uuid               : 223ede10-9169-4c7d-9524-a546e24bfab5
detect_mult         : 5
dst_ip              : "10.16.0.12"
external_ids        : {af="4", vendor=kube-ovn, vpc-egress-gateway="defau
lt/gateway2"}
logical_port        : "bfd@ovn-cluster"
min_rx              : 100
min_tx              : 100
options             : {}
status              : up

_uuid               : b050c75e-2462-470b-b89c-7bd38889b758
detect_mult         : 5
dst_ip              : "10.16.0.13"
external_ids        : {af="4", vendor=kube-ovn, vpc-egress-gateway="defau
lt/gateway2"}
logical_port        : "bfd@ovn-cluster"
min_rx              : 100
min_tx              : 100
options             : {}
status              : up
```

Then check the BFD sessions:

```
$ kubectl exec -n default gateway2-fcc6b8b87-8lgvx -c bfdd -- bfdd-contro
l status
There are 1 sessions:
Session 1
 id=1 local=10.16.0.13 (p) remote=10.255.255.255 state=Up

$ kubectl exec -n default gateway2-fcc6b8b87-wmww6 -c bfdd -- bfdd-contro
l status
There are 1 sessions:
Session 1
 id=1 local=10.16.0.12 (p) remote=10.255.255.255 state=Up
```

NOTE

If all gateway instances are down, egress traffic handled by the VPC Egress Gateway is dropped.

Operations That May Interrupt Traffic

The following operations may briefly interrupt egress traffic because they delete or recreate gateway instances:

1. Changing the number of replicas
2. Changing configuration such as internal or external IPs, node selectors, or BFD settings
3. Upgrading or downgrading Kube-OVN if *.spec.image* is not specified
4. Manually deleting an Egress Gateway Pod

Additional Resources

- RFC 5880 - Bidirectional Forwarding Detection (BFD)↗

Configuring Kube-OVN Network to Support Pod Multi-Network Interfaces (Alpha)

By using Multus CNI, you can add multiple network interfaces with different networks to Pods. Use Kube-OVN network's Subnet and IP CRDs for advanced IP management, implementing subnet management, IP reservation, random allocation, fixed allocation, and other features.

TOC

Installing Multus CNI

Deploying the Multus CNI Plugin

1. Go to **Administrator**.
2. In the left navigation bar, click **Marketplace** > **Cluster Plugins**.
3. In the search bar, type "multus" to find the Multus CNI plugin.
4. Locate the **"Alauda Container Platform Networking for Multus"** plugin in the list.
5. Click the three dots (⋮) next to the plugin entry and select **Install**.
6. The plugin will be deployed to your cluster. You can monitor the installation status in the **State** column.

> **NOTE**
>
> The Multus CNI plugin serves as middleware between other CNI plugins and Kubernetes, enabling Pods to have multiple network interfaces.

Creating Subnets

Create an attachnet subnet according to the following example: `network-attachment-definition.yml`.

> **NOTE**
>
> The provider format in config is `<NAME>.<NAMESPACE>.ovn`, where `<NAME>` and `<NAMESPACE>` are the name and namespace of this NetworkAttachmentDefinition CR respectively.

```
apiVersion: 'k8s.cni.cncf.io/v1'
kind: NetworkAttachmentDefinition
metadata:
  name: attachnet
  namespace: default
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "kube-ovn",
    "server_socket": "/run/openvswitch/kube-ovn-daemon.sock",
    "provider": "attachnet.default.ovn"
    }'
```

After creation, apply the resource:

```
kubectl apply -f network-attachment-definition.yml
```

Use the following example to create the Kube-OVN subnet for the second network interface: `subnet.yml`.

NOTE

- `spec.provider` must be consistent with the provider in NetworkAttachmentDefinition.
- If you need to use an Underlay subnet, set the `spec.vlan` of the subnet to the VLAN CR name you want to use. Configure other subnet parameters as needed.

```
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: subnet1
spec:
  cidrBlock: 172.170.0.0/16
  provider: attachnet.default.ovn
```

After creation, apply the resource:

```
kubectl apply -f subnet.yml
```

Creating Pod with Multiple Network Interfaces

Create a pod according to the following example.

NOTE

- The `metadata.annotations` must contain a key-value pair `k8s.v1.cni.cncf.io/networks=default/attachnet`, where the value format is `<NAMESPACE>/<NAME>`, and `<NAMESPACE>` and `<NAME>` are the namespace and name of the NetworkAttachmentDefinition CR respectively.
- If the Pod needs three network interfaces, configure the value of `k8s.v1.cni.cncf.io/networks` as `default/attachnet,default/attachnet2`.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  annotations:
    k8s.v1.cni.cncf.io/networks: default/attachnet
spec:
  containers:
    - name: web
      image: nginx:latest
      ports:
        - containerPort: 80
```

After the Pod is created successfully, use the command `kubectl exec pod1 -- ip a` to view the Pod's IP addresses.

Verifying Dual Network Interface Creation

Use the following command to verify that the dual network interfaces have been created successfully:

```
kubectl exec pod1 -- ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen
1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
151: eth0@if152: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1400 qdisc
noqueue state UP
    link/ether a6:3c:d8:ae:83:06 brd ff:ff:ff:ff:ff:ff
    inet 10.3.0.8/16 brd 10.3.255.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::a43c:d8ff:feae:8306/64 scope link
       valid_lft forever preferred_lft forever
153: net1@if154: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1400 qdisc
noqueue state UP
    link/ether 0a:36:08:01:dc:df brd ff:ff:ff:ff:ff:ff
    inet 172.170.0.3/16 brd 172.170.255.255 scope global net1
       valid_lft forever preferred_lft forever
    inet6 fe80::836:8ff:fe01:dcdf/64 scope link
       valid_lft forever preferred_lft forever
```

Additional Features

Fixed IP

- **Primary Network Interface (First Interface)**: If you need to fix the IP of the primary network interface, the method is the same as using a fixed IP with a single network interface. Add the annotation `ovn.kubernetes.io/ip_address=<IP>` to the Pod.
- **Secondary Network Interface (Second Interface or Other Interfaces)**: The basic method is similar to the primary network interface, with the difference that the `ovn` in the Annotation Key is replaced with the corresponding NetworkAttachmentDefinition provider. Example: `attachnet.default.ovn.kubernetes.io/ip_address=172.170.0.101`.

Additional Routes

Starting from version 1.8.0, Kube-OVN supports configuring additional routes for secondary network interfaces. When using this feature, add the `routers` field to the config in NetworkAttachmentDefinition and fill in the routes you need to configure. Example:

```
apiVersion: 'k8s.cni.cncf.io/v1'
kind: NetworkAttachmentDefinition
metadata:
  name: attachnet
  namespace: default
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "kube-ovn",
    "server_socket": "/run/openvswitch/kube-ovn-daemon.sock",
    "provider": "attachnet.default.ovn",
    "routes": [{
      "dst": "19.10.0.0/16"
    }, {
      "dst": "19.20.0.0/16",
      "gw": "19.10.0.1"
    }]
  }'
```

Configure Centralized Gateway

Centralized Gateway allows Pods within a subnet to access the external network using fixed IPs. This is particularly useful for security operations such as network auditing, IP whitelisting, and firewall rule management, where you need to identify and control traffic from specific source IPs. In centralized gateway mode, all outbound traffic from Pods is routed through designated gateway nodes, enabling centralized network policy enforcement and monitoring.

> **NOTE**
>
> Pods under a centralized subnet cannot be accessed through `hostport` or a NodePort type Service with `externalTrafficPolicy: Local`.

If you want traffic within the Subnet to access the external network using a fixed IP for security operations such as auditing and whitelisting, you can set the gateway type in the Subnet to centralized. In centralized gateway mode, packets from Pods accessing the external network are first routed to the `ovn0` NIC of a specific nodes, and then outbound through the host's routing rules. When `natOutgoing` is `true`, the Pod will use the IP of a specific nodes when accessing the external network.

The centralized gateway example is as follows, where the `gatewayType` field is `centralized` and `gatewayNode` is the NodeName of the particular machine in Kubernetes.

```yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: centralized
spec:
  protocol: IPv4
  cidrBlock: 10.166.0.0/16
  default: false
  excludeIps:
  - 10.166.0.1
  gateway: 10.166.0.1
  gatewayType: centralized
  gatewayNode: "node1,node2"
  natOutgoing: true
```

- If a centralized gateway wants to specify a specific NIC of a machine for outbound networking, `gatewayNode` format can be changed to `kube-ovn-worker:172.18.0.2,kube-ovn-control-plane:172.18.0.3`.
- The spec field `enableEcmp` has been added to the subnet crd definition since Kube-OVN v1.12.0 to migrate the ECMP switch to the subnet level. You can set whether to enable ECMP mode based on different subnets. The `enable-ecmp` parameter in the `kube-ovn-controller` deployment is no longer used. After the previous version is upgraded to v1.12.0, the subnet switch will automatically inherit the value of the original global switch parameter.

NOTE

In centralized gateway ECMP mode, kube-ovn-controller actively probes node status through ping, detecting failures within 5s and completing failover within 5s-10s, during which some traffic may fail.

In centralized gateway primary-backup mode, failover is based on Node Ready status, and it may take several minutes to complete failover in case of power outage.

TOC

Using Label Selectors to Specify Gateway Nodes

In addition to specifying node names directly, you can use `gatewayNodeSelectors` to dynamically select gateway nodes using label selectors. This approach is more flexible, especially useful when node names are not fixed or when you need to select gateways based on labels dynamically.

NOTE

- If `gatewayNode` is not empty, it takes precedence and `gatewayNodeSelectors` is ignored.
- Multiple selectors are evaluated with OR logic - a node matching any selector becomes a gateway node.
- When node labels change, the system automatically updates the gateway node list.

```yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: centralized-selector
spec:
  protocol: IPv4
  cidrBlock: 10.166.0.0/16
  default: false
  excludeIps:
  - 10.166.0.1
  gateway: 10.166.0.1
  gatewayType: centralized
  gatewayNodeSelectors:
    - matchLabels:
        role: gateway
    - matchExpressions:
        - key: node-type
          operator: In
          values: ["gateway", "egress"]
  natOutgoing: true
```

Configure IPPool

IPPool is a more granular IPAM management unit than Subnet. You can subdivide the subnet segment into multiple units through IPPool, and each unit is bound to one or more namespaces.

TOC

Instructions

Create IPPool

Below is an example:

```yaml
apiVersion: kubeovn.io/v1
kind: IPPool
metadata:
  name: pool-1
spec:
  subnet: ovn-default (1)
  ips: (2)
  - "10.16.0.201"
  - "10.16.0.210/30"
  - "10.16.0.220..10.16.0.230"
  namespaces: (3)
  - ns-1
```

(1) Subnet to which the IP pool belongs.

(2) IP ranges. Supported formats: <*IP*>, <*CIDR*> and <*IP1*>..<*IP2*>. Both IPv4 and IPv6 are supported.

(3) Optional namespaces the IP pool is bound to. Pods in a bound namespace will only get IPs from the bound pool(s), not other ranges in the subnet(s).

Use IPPool

To assign IPs randomly from the IP pool, simply bind the IP pool to the desired namespace(s). When Pods in the bound namespace are created, their IPs will be allocated from the corresponding IP pool(s).

> **NOTE**
>
> Before binding an IP pool to a namespace, make sure only the subnet to which the IP pool belongs is bound to the namespace.

You can also assign an IP pool to Pods through annotation:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-1
  annotations:
    ovn.kubernetes.io/ip_pool: pool-1
spec:
  containers:
  - name: web
    image: nginx:latest
```

For workloads, use annotation in the Pod template of the Deployment, StatefulSet, etc.:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
      annotations:
        ovn.kubernetes.io/ip_pool: pool-1
    spec:
      containers:
      - name: web
        image: nginx:latest
```

Precautions

1. To ensure compatibility with function *Fixed Addresses*, name of the IP pool cannot be an IP address.
2. IP addresses out of the subnet range are allowed, while these IPs will not be effective.

3. Different IP pools belonging to the same subnet cannot have overlapping IP ranges.
4. The `.spec.ips` field can be updated whenever necessary. Any changes will take effect immediately.
5. An IP pool will inherit the reserved IP of the subnet. When randomly assigning an IP address from an IP pool, the reserved IP within the IP pool range will be skipped.
6. When randomly assigning an IP address from a subnet, IP ranges of all IP pools in the subnet will be excluded.
7. Multiple IP pools can be bound to the same namespace.

Configure MTU

In Kube-OVN, the MTU (Maximum Transmission Unit) setting is crucial for ensuring optimal network performance and avoiding packet fragmentation. The MTU defines the maximum size of a packet that can be transmitted over the network.

By default, Kube-OVN detects the MTU of the underlying physical network interface and sets the MTU for the virtual network interfaces accordingly. However, there are scenarios where you might need to customize the MTU settings for your Kube-OVN networking components.

TOC

Default MTU Behavior

Kube-OVN automatically detects the MTU of the physical network interface on the host machine, and sets the MTU for the Pod and OVS interfaces accordingly.

In overlay networks, Kube-OVN reduces the MTU to accommodate the VXLAN or Geneve encapsulation overhead. Here is how the MTU is calculated:

Encapsulation Type	Tunnel IP Version	MTU Calculation
Geneve	IPv4	Physical Network Interface MTU - 100
	IPv6	Physical Network Interface MTU - 120
VXLAN	IPv4	Physical Network Interface MTU - 50
	IPv6	Physical Network Interface MTU - 70

In underlay networks, the MTU is set to match the physical network interface MTU.

Customizing MTU Settings

MTU settings can be customized globally for overlay networks or on a per-subnet basis.

WARNING

Adjusting MTU settings incorrectly can lead to network performance issues, including packet loss and fragmentation. Ensure that the MTU settings are compatible with your underlying network infrastructure before making changes.

Critical Consideration When Increasing MTU

When increasing MTU from a smaller to a larger value, you must restart all Pods to ensure the new MTU takes effect uniformly across the cluster.

Why is this necessary?

OVS internal ports (such as `ovn0`) automatically adopt the **smallest MTU** among all interfaces connected to the `br-int` bridge as their own MTU. This behavior creates a potential issue in the following scenario:

1. You configure a larger MTU value for new Pods
2. Some existing Pods still use the original smaller MTU
3. The `ovn0` interface retains the smaller MTU due to the minimum MTU rule

4. Traffic from Pods with the larger MTU gets dropped when packets exceed the `ovn0` MTU

Solution: After increasing MTU settings, recreate all Pods to ensure consistent MTU configuration across the entire network path.

> **NOTE**
>
> MTU configuration changes will only take effect for newly created Pods. Existing Pods will retain their original MTU settings until they are recreated. It's recommended to plan for Pod restarts when changing MTU settings.

Global MTU Configuration

To set a global MTU for overlay networks, you can modify the command parameter of the `kube-ovn-cni` DaemonSet. For example, to set the global MTU to 1400, you would update the DaemonSet as follows:

```
kubectl -n kube-system edit ds kube-ovn-cni
```

Then, add or update the `--mtu=1400` parameter to the container's arguments:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
    kubernetes.io/description: |
      This daemon set launches the kube-ovn cni daemon.
  name: kube-ovn-cni
  namespace: kube-system
spec:
  selector:
    matchLabels:
      app: kube-ovn-cni
  template:
    metadata:
      labels:
        app: kube-ovn-cni
        component: network
        type: infra
    spec:
      containers:
      - name: cni-server
        args:
        - --mtu=1400
```

NOTE

The global MTU setting only affects overlay networks. For underlay networks, the MTU remains the same as the physical network interface MTU by default.

Per-Subnet MTU Configuration

You can also set the MTU for individual subnets by specifying the `mtu` field in the subnet configuration. For example, to set the MTU for a specific subnet to 1450, you can use the following command:

```
kubectl patch subnet my-subnet --type merge -p '{"spec": {"mtu": 1450}}'
```

This setting will override the global MTU setting for that particular subnet.

TOC

weight: 13

Automatic Interconnection of Underlay and Overlay Subnets

If a cluster has both Underlay and Overlay subnets, by default, Pods under the Overlay subnet can access Pods' IPs in the Underlay subnet through a gateway using NAT. However, Pods in the Underlay subnet need to configure node routing to access Pods in the Overlay subnet.

To achieve automatic interconnection between Underlay and Overlay subnets, you can manually modify the YAML file of the Underlay subnet. Once configured, Kube-OVN will also use an additional Underlay IP to connect the Underlay subnet and the ovn-cluster logical router, setting the corresponding routing rules to enable interconnection.

Procedure

1. Go to **Administrator**.
2. In the left navigation bar, click on **Cluster Management** > **Resource Management**.
3. Enter **Subnet** to filter resource objects.
4. Click on ⋮ > **Update** next to the Underlay subnet to be modified.
5. Modify the YAML file, adding the field `u2oInterconnection: true` in the `Spec`.
6. Click **Update**.

Note: Existing compute components in the Underlay subnet need to be recreated for the changes to take effect.

Isolation Between Underlay Subnets with u2oInterconnection Enabled

When multiple Underlay subnets have `u2oInterconnection: true` enabled, traffic between them no longer goes through the physical gateway but is routed directly via the internal OVN network.

If you need to isolate two Underlay subnets while both have `u2oInterconnection` enabled, you must first configure the kube-ovn-controller parameter, then configure the subnet isolation.

Step 1: Configure kube-ovn-controller

Modify the kube-ovn-controller Deployment to disable connection tracking skip for destination logical port IPs:

```
kubectl edit deployment kube-ovn-controller -n kube-system
```

Add or modify the following argument:

```
spec:
  template:
    spec:
      containers:
      - name: kube-ovn-controller
        args:
        - --ls-ct-skip-dst-lport-ips=false
```

CAUTION

`--ls-ct-skip-dst-lport-ips` controls whether to skip connection tracking (conntrack) for traffic destined to logical port IPs. The default value is `true`, which skips conntrack to improve performance. Setting it to `false` does not affect functionality but may slightly impact performance.

However, for Underlay subnets with ACL-based isolation, you **must** set it to `false`. Otherwise, gateway-to-Pod traffic will fail (e.g., ping requests reach the Pod but replies are dropped), because ACL isolation uses `allow-related` which requires conntrack state; without it, replies cannot be identified as "related" and get dropped.

Step 2: Configure Subnet Isolation

Configure the subnet with the following parameters:

```
spec:
  u2oInterconnection: true
  acls:
  - action: drop
    direction: to-lport  # Ingress direction (traffic entering the logica
l port)
    match: ip4.src == 172.20.0.0/16
    priority: 1002
  - action: drop
    direction: to-lport  # Ingress direction
    match: ip4.src == 192.50.0.0/16
    priority: 1002
```

ACL Parameters:

Parameter	Description
`action`	The action to take: `allow`, `drop`, or `allow-related`
`direction`	Traffic direction: `to-lport` (ingress) or `from-lport` (egress)
`match`	OVN match expression using L2-L4 fields and boolean operators
`priority`	Rule priority (higher values are evaluated first; recommended range: 1002-1899)

NOTE

- The `acls` field provides priority-based rule evaluation, offering more flexibility than standard Kubernetes NetworkPolicy.
- When using `to-lport` direction, `ip4.src` refers to the source IP of incoming traffic.
- **Recommended priority range**: `1002` to `1899` to avoid conflicts with system default ACL rules.

Configure Endpoint Health Checker

TOC

Overview

The Endpoint Health Checker is a cluster plugin designed to monitor and manage the health status of service endpoints in k8s cluster. It automatically removes unhealthy endpoints from service to ensure traffic is only routed to healthy instances, improving overall service reliability and availability.

Key Features

- **Automatic Health Monitoring**: Continuously monitors the health status of service endpoints in k8s cluster
- **Load Balancer Integration**: Automatically removes unhealthy endpoints from service
- **Service Availability**: Ensures traffic is only directed to healthy, available endpoints
- **Rapid Failover**: Reduces endpoint switching time from 40s to 10s during node power outages

Installation

Install via Marketplace

1. Navigate to **Administrator** > **Marketplace** > **Cluster Plugins**.
2. Search for "**Alauda Container Platform Endpoint Health Checker**" in the plugin list.
3. Click **Install** to open the installation configuration page.
4. In the deployment configuration dialog, you can optionally configure the following parameters:

Parameter	Description
Node Selectors	Configure label selectors to specify which nodes the Endpoint Health Checker components should run on. Click **Add** to add multiple label key-value pairs.

Parameter	Description
Node Tolerations	Configure tolerations to allow Endpoint Health Checker components to be scheduled on nodes with specific taints. Click **Add** to add multiple tolerations with Key, Value, and Type.

5. Click **Install** to deploy the plugin.
6. Wait for the plugin status to change to "**Ready**".

How It Works

Health Check Mechanism

The Endpoint Health Checker is a dedicated health monitoring component that ensures only healthy endpoints receive traffic. It operates by monitoring service endpoints and automatically managing their availability status.

Core Functionality

The Endpoint Health Checker works by:

1. **Service Discovery**: Identifies services and pods configured for health monitoring in the cluster.
2. **Pod Health Monitoring**: Monitors the readiness and liveness probe status of pods backing the service endpoints
3. **Active Health Checks**: Performs active health assessments using configurable criteria:
   - **TCP connectivity checks**: Establishes TCP connections to verify port accessibility
4. **Endpoint Management**: Automatically removes unhealthy endpoints from service endpoint lists to prevent traffic routing to failed instances

Health Check Process

The health checking process involves:

- **Probe Integration**: Leverages Kubernetes readiness and liveness probe results as initial health indicators
- **Network Connectivity**: Sends TCP packets to target endpoint ports to verify accessibility
- **Response Validation**: Evaluates response status, timing, and content to determine endpoint health
- **Automatic Failover**: Removes unresponsive or failed endpoints from service endpoint lists

Performance Improvement

- **Previous Method**: Relied on kubelet heartbeat detection with up to 40 seconds delay
- **Current Method**: Active endpoint health checking with 10 second detection and switching time
- **Improvement**: Significantly improves service availability during node failures in ALB + MetalLB environments

How To Activate

Health checking can be activated through two methods:

Pod-level annotation (Recommended)

For ALB

set `alb.cpaas.io/pod-annotations` annotation of `ALB2`

```
apiVersion: crd.alauda.io/v2
kind: ALB2
metadata:
  annotations:
    alb.cpaas.io/pod-annotations: '{"endpoint-health-checker.io/enable
d":"true"}'
  name: demo-alb
spec:
  config:
    loadbalancerName: demo-alb
    nodeSelector:
      ingress: 'true'
    replicas: 1
  type: nginx
```

For IngressNginx

1. Install ingress-nginx
2. Set `podAnnotations` in `.spec.controller.podAnnotations` of `IngressNginx`.

```
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
  namespace: ingress-nginx-operator
spec:
  controller:
    replicaCount: 1
    podAnnotations:
      endpoint-health-checker.io/enabled: 'true'
```

For EnvoyGateway

1. Install envoy-gateway-operator
2. Set `annotations` in `.spec.provider.kubernetes.envoyDeployment.patch.value.spec.template.metadata.annotations` of `EnvoyProxy`.

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo
spec:
  infrastructure:
    parametersRef:
      group: gateway.envoyproxy.io
      kind: EnvoyProxy
      name: demo
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: http
      port: 80
      protocol: HTTP
---
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
spec:
  provider:
    kubernetes:
      envoyDeployment:
        replicas: 1
        patch:
          type: StrategicMerge
          value:
            spec:
              template:
                metadata:
                  annotations:
                    endpoint-health-checker.io/enabled: 'true'
        container:
          imageRepository: registry.alauda.cn:60080/acp/envoyproxy/envo
y
    type: Kubernetes
```

For Custom Deployment

set `annotations` in `.spec.template.metadata.annotations` of `Deployment`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
      annotations:
        endpoint-health-checker.io/enabled: 'true'
    spec:
      containers:
        - name: container
          ports:
            - containerPort: 8080
          livenessProbe:
            tcpSocket:
              port: 8080
            initialDelaySeconds: 15
            periodSeconds: 10
          readinessProbe:
            tcpSocket:
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 5
```

Pod-level readinessGates (Legacy)

Configure readinessGates in the pod spec for older versions:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pod-legacy
  namespace: cpaas-system
spec:
  replicas: 3
  selector:
    matchLabels:
      app: pod-legacy
  template:
    metadata:
      labels:
        app: pod-legacy
    spec:
      readinessGates:
        - conditionType: 'endpointHealthCheckSuccess'
      containers:
        - name: container
          image: your-image:latest
          ports:
            - containerPort: 8080
          livenessProbe:
            tcpSocket:
              port: 8080
            initialDelaySeconds: 15
            periodSeconds: 10
          readinessProbe:
            tcpSocket:
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 5
```

> **Note**: The readinessGates configuration is from an older version. It's recommended to use the pod annotation `endpoint-health-checker.io/enabled: 'true'` for new deployments.

Uninstallation

To uninstall the Endpoint Health Checker:

1. Navigate to **Administrator** > **Marketplace** > **Cluster Plugins**.
2. Find the installed "**Endpoint Health Checker**" plugin.
3. Click the options menu and select **Uninstall**.
4. Confirm the uninstallation when prompted.

Tasks for ALB

TOC

How To Set NodeSelector And Tolerations For alb-operator

update the `deployment` resources

```
# example of nodeSelector and tolerations
kubectl patch subscription ingress-nginx-operator -n ingress-nginx-operat
or    --type='merge' -p '
{
  "spec": {
    "config": {
      "nodeSelector": {
        "node-role.kubernetes.io/infra": ""
      },
      "tolerations": [
        {
          "effect": "NoSchedule",
          "key": "node-role.kubernetes.io/infra",
          "operator": "Equal",
          "value": "reserved"
        }
      ]
    }
  }
}'
```

How To Set NodeSelector And Tolerations For alb

update the `alb` resources

```
kubectl patch alb2 $NAME -n $MS --type='merge' -p '
{
  "metadata": {
    "annotations": {
      "alb.cpaas.io/toleration": "[{\"key\":\"node-role.kubernetes.io/infra\",\"operator\":\"Equal\",\"value\":\"reserved\",\"effect\":\"NoSchedule\"}]"
    }
  },
  "spec": {
    "config": {
      "nodeSelector": {
        "node-role.kubernetes.io/infra": ""
      }
    }
  }
}'
```

Task: Migrate from OCP Route to GatewayAPI Route

TOC

Introduction

This guide provides detailed instructions for migrating from OpenShift Container Platform (OCP) Routes to Kubernetes Gateway API HTTPRoutes with Envoy Gateway. Each section covers a specific OCP Route feature and its equivalent configuration in Gateway API.

Prerequisites

1. Configure EnvoyGatewayCtl
2. Configure Gateway
3. Basic understanding of Gateway API Routes

Basic HTTP Route

OCP Route Configuration

In OCP, a basic HTTP route is created using the Route resource:

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  namespace: demo
spec:
  host: example.com 1
  to: 2
    kind: Service
    name: example-service
  port: 3
    targetPort: 8080
```

1. The hostname for the route
2. Backend service reference
3. Target port on the backend service

Gateway API Configuration

In Gateway API, the equivalent configuration uses HTTPRoute:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames: (1)
    - example.com
  parentRefs: (2)
    - name: demo-gateway
      namespace: demo
  rules:
    - backendRefs: (3)
        - name: example-service
          port: 8080
```

1. Hostnames that this route accepts (equivalent to OCP Route's `host`)
2. Reference to the Gateway listener
3. Backend service and port

Route Timeouts

OCP Route Configuration

In OCP, timeouts are configured using annotations:

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/timeout: 30s (1)
    haproxy.router.openshift.io/timeout-tunnel: 1h (2)
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

(1) General timeout for HTTP requests

(2) Timeout for tunnel connections (WebSocket, HTTP/2, etc.)

Gateway API Configuration

In Gateway API, timeouts are configured in the HTTPRoute rules:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - backendRefs:
        - name: example-service
          port: 8080
      timeouts: (1)
        request: 30s
        backendRequest: 25s
```

(1) Request timeout configuration

For more details, see Request Timeouts↗.

> **NOTE**
>
> Gateway API does not have a separate timeout specifically for tunnel connections. The `request` timeout applies to all connection types.

HTTP Strict Transport Security (HSTS)

OCP Route Configuration

In OCP, HSTS is configured using the annotation on edge-terminated or re-encrypt routes:

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/hsts_header: max-age=31536000;includeSubD
omains;preload (1)
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
  tls:
    termination: edge
```

(1) HSTS header configuration

Gateway API Configuration

In Gateway API, HSTS is configured using response header modification:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - filters: (1)
        - type: ResponseHeaderModifier
          responseHeaderModifier:
            add:
              - name: Strict-Transport-Security
                value: max-age=31536000;includeSubDomains;preload
      backendRefs:
        - name: example-service
          port: 8080
```

(1) Add HSTS header to response

For more details, see HTTP Header Modification ↗.

> **NOTE**
>
> Unlike OCP's `requiredHSTSPolicies` cluster-wide enforcement, Gateway API requires explicit configuration per route. There is no global HSTS policy enforcement mechanism in Gateway API.

Cookie-Based Session Affinity

OCP Route Configuration

In OCP, session affinity is configured using annotations:

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/balance: source (1)
    haproxy.router.openshift.io/disable_cookies: "false" (2)
    router.openshift.io/cookie_name: my-cookie (3)
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

1. Load balancing algorithm
2. Enable cookie-based session affinity
3. Cookie name for session persistence

Gateway API Configuration

In Gateway API, session persistence is configured in the HTTPRoute rules:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - backendRefs:
        - name: example-service
          port: 8080
      sessionPersistence: 1
        type: Cookie
        sessionName: my-cookie
        cookieConfig:
          lifetimeType: Permanent 2
```

1 Session persistence configuration

2 Cookie lifetime type: `Permanent` (persists across browser sessions) or `Session` (expires when browser closes)

For more details, see Session Affinity/Sticky Sessions↗.

Path-Based Routing

OCP Route Configuration

In OCP, path-based routing uses the `path` field:

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
spec:
  host: example.com
  path: /api (1)
  to:
    kind: Service
    name: example-service
```

(1) Path prefix for matching requests

Gateway API Configuration

In Gateway API, path matching is configured in the HTTPRoute rules:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - matches: (1)
        - path:
            type: PathPrefix
            value: /api
      backendRefs:
        - name: example-service
          port: 8080
```

(1) Path matching configuration

Gateway API supports multiple match types:

- `PathPrefix`: Matches the path prefix (equivalent to OCP's default behavior)
- `Exact`: Matches the exact path
- `RegularExpression`: Matches using regular expressions

For more details, see HTTPRoute Matches.

Header Modification

OCP Route Configuration

In OCP, header modification uses annotations:

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/response-set-header: X-Custom-Header:valu
e (1)
    haproxy.router.openshift.io/request-set-header: X-Request-Header:valu
e (2)
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

(1) Set response headers

(2) Set request headers

Gateway API Configuration

In Gateway API, header modification is configured using filters:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - filters:
        - type: RequestHeaderModifier (1)
          requestHeaderModifier:
            add:
              - name: X-Request-Header
                value: value
        - type: ResponseHeaderModifier (2)
          responseHeaderModifier:
            add:
              - name: X-Custom-Header
                value: value
      backendRefs:
        - name: example-service
          port: 8080
```

(1) Request header modification

(2) Response header modification

For more details, see HTTP Header Modification ↗.

Connection Limits

OCP Route Configuration

In OCP, connection limits are configured using annotations:

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/pod-concurrent-connections: "100" (1)
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

(1) Maximum concurrent connections per backend pod

Gateway API Configuration

In Gateway API, connection limits are configured using ClientTrafficPolicy attached to the Gateway:

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: ClientTrafficPolicy
metadata:
  name: connection-limit-policy
  namespace: demo
spec:
  targetRefs: (1)
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo-gateway
  connection: (2)
    connectionLimit:
      value: 100
```

(1) Attach policy to Gateway

(2) Connection limit configuration

For more details, see Connection Limit↗.

NOTE

Unlike OCP's per-backend pod limit, Gateway API's connection limit is applied at the Gateway level and is distributed across Envoy proxy instances.

Rate Limiting

OCP Route Configuration

In OCP, rate limiting uses annotations:

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/rate-limit-connections: "true" 1
    haproxy.router.openshift.io/rate-limit-connections.concurrent-tcp: "1
0"
    haproxy.router.openshift.io/rate-limit-connections.rate-http: "100"
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

1 Rate limiting configuration

Gateway API Configuration

In Gateway API, rate limiting is configured using BackendTrafficPolicy:

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: BackendTrafficPolicy
metadata:
  name: rate-limit-policy
  namespace: demo
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: example-route
  rateLimit: 1
    type: Local
    local:
      rules:
        - limit:
            requests: 100
            unit: Second
```

1 Rate limiting configuration

For more details, see Rate Limiting ↗.

> **NOTE**
>
> Gateway API rate limiting is more flexible than OCP Route annotations and supports both local and global rate limiting mechanisms.

IP Allowlist/Blocklist

OCP Route Configuration

In OCP, IP allowlisting uses annotations:

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/ip_allowlist: 192.168.1.0/24 10.0.0.1 1
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

1 IP allowlist configuration

Gateway API Configuration

In Gateway API, IP filtering is implemented using SecurityPolicy with authorization rules:

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: SecurityPolicy
metadata:
  name: ip-filter-policy
  namespace: demo
spec:
  targetRefs: 1
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: example-route
  authorization: 2
    defaultAction: Deny
    rules:
      - action: Allow
        principal:
          clientCIDRs: 3
            - 192.168.1.0/24
            - 10.0.0.1/32
```

1 Attach policy to HTTPRoute

2 Authorization configuration with default deny

3 IP allowlist using CIDR notation

For IP blocklist (denylist), set `defaultAction: Allow` and use `action: Deny`:

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: SecurityPolicy
metadata:
  name: ip-blocklist-policy
  namespace: demo
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: example-route
  authorization:
    defaultAction: Allow (1)
    rules:
      - action: Deny (2)
        principal:
          clientCIDRs:
            - 192.168.100.0/24
```

1 Default action allows all traffic

2 Deny traffic from specific IPs

For more details, see IP Allowlist/Denylist↗.

> **NOTE**
>
> Ensure you configure the client IP detection correctly using ClientTrafficPolicy if your Gateway is behind a load balancer or proxy.

URL Rewrite

OCP Route Configuration

In OCP, URL rewriting uses the annotation:

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/rewrite-target: /new-path (1)
spec:
  host: example.com
  path: /old-path
  to:
    kind: Service
    name: example-service
```

(1) Rewrite target path

Gateway API Configuration

In Gateway API, URL rewriting is configured using filters:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - matches:
        - path:
            type: PathPrefix
            value: /old-path
      filters:
        - type: URLRewrite (1)
          urlRewrite:
            path:
              type: ReplacePrefixMatch
              replacePrefixMatch: /new-path
      backendRefs:
        - name: example-service
          port: 8080
```

(1) URL rewrite filter

For more details, see URL Rewrite ↗.

Cross-Namespace Route Admission

OCP Route Configuration

In OCP, the route admission policy controls whether routes in different namespaces can claim the same hostname. This is configured at the cluster level through the Ingress Operator.

Gateway API Configuration

In Gateway API, cross-namespace access is controlled at the Gateway listener level:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: gateway-ns
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: http
      protocol: HTTP
      port: 80
      allowedRoutes: 1
        namespaces:
          from: All  # Allow routes from all namespaces
```

1 Configure which namespaces can attach routes

Options for `allowedRoutes.namespaces.from`:

- `Same`: Only routes in the same namespace as the Gateway
- `All`: Routes from any namespace
- `Selector`: Routes from namespaces matching a label selector

For more details, see Cross-Namespace Routing.

> **NOTE**
>
> Unlike OCP's cluster-wide admission policy, Gateway API controls this at the individual Gateway listener level, providing more granular control.

Default TLS Certificate for Ingress

OCP Route Configuration

In OCP, routes without TLS configuration can use a default certificate configured at the Ingress Controller level.

Gateway API Configuration

In Gateway API, configure a default TLS certificate on the Gateway listener:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: demo
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: https
      protocol: HTTPS
      port: 443
      tls: (1)
        mode: Terminate
        certificateRefs:
          - name: default-tls-cert
```

(1) Default TLS certificate for the listener

Any HTTPRoute attached to this listener without specific TLS configuration will use this default certificate.

TLS Re-encrypt with Custom CA

OCP Route Configuration

In OCP, re-encrypt routes terminate TLS at the router and re-encrypt to the backend with validation:

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
  tls:
    termination: reencrypt (1)
    destinationCACertificate: | (2)
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
```

(1) Re-encrypt termination mode

(2) CA certificate for validating backend

Gateway API Configuration

In Gateway API, backend TLS validation is configured using BackendTLSPolicy:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: demo
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: https
      protocol: HTTPS
      port: 443
      tls:
        mode: Terminate (1)
        certificateRefs:
          - name: frontend-tls
---
apiVersion: gateway.networking.k8s.io/v1
kind: BackendTLSPolicy
metadata:
  name: backend-tls-policy
  namespace: demo
spec:
  targetRefs: (2)
    - group: ""
      kind: Service
      name: example-service
  validation: (3)
    caCertificateRefs:
      - name: backend-ca-cert
        kind: ConfigMap
    hostname: backend.example.com
---
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  parentRefs:
    - name: demo-gateway
  hostnames:
    - example.com
  rules:
```

```
    - backendRefs:
        - name: example-service
          port: 8443
```

1. Terminate TLS at the Gateway
2. Apply policy to the backend Service
3. Backend TLS validation configuration

For more details, see Backend TLS↗.

> **NOTE**
>
> The CA certificate must be stored in a ConfigMap, not a Secret.

Edge Termination with Custom Certificate

OCP Route Configuration

In OCP, edge termination uses the route's TLS configuration:

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
  tls:
    termination: edge (1)
    certificate: | (2)
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
    key: | (3)
      -----BEGIN PRIVATE KEY-----
      ...
      -----END PRIVATE KEY-----
```

(1) Edge termination mode

(2) TLS certificate

(3) TLS private key

Gateway API Configuration

In Gateway API, TLS termination is configured on the Gateway listener:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: example-tls
  namespace: demo
type: kubernetes.io/tls (1)
data:
  tls.crt: <base64-encoded-cert>
  tls.key: <base64-encoded-key>
---
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: demo
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: https
      protocol: HTTPS (2)
      port: 443
      tls:
        mode: Terminate (3)
        certificateRefs: (4)
          - name: example-tls
---
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  parentRefs:
    - name: demo-gateway
      sectionName: https
  hostnames:
    - example.com
  rules:
    - backendRefs:
        - name: example-service
          port: 8080
```

1. TLS secret type
2. HTTPS protocol
3. Terminate TLS mode
4. Reference to TLS secret

TLS Passthrough

OCP Route Configuration

In OCP, passthrough routes do not terminate TLS:

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
  tls:
    termination: passthrough (1)
```

1. Passthrough termination mode

Gateway API Configuration

In Gateway API, TLS passthrough uses TLSRoute:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: demo
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: tls-passthrough
      protocol: TLS (1)
      port: 443
      tls:
        mode: Passthrough (2)
---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TLSRoute
metadata:
  name: example-route
  namespace: demo
spec:
  parentRefs:
    - name: demo-gateway
      sectionName: tls-passthrough
  hostnames: (3)
    - example.com
  rules:
    - backendRefs:
        - name: example-service
          port: 8443
```

(1) TLS protocol

(2) Passthrough mode

(3) SNI hostname for routing

For more details, see TLS Passthrough and TLSRoute specification ↗.

NOTE

TLSRoute uses SNI (Server Name Indication) for routing decisions without terminating TLS. This means:

- No access to HTTP headers or request content for routing decisions
- No L7 filters (header modification, URL rewrite, etc.) can be applied
- Routing is purely based on the SNI hostname

Feature Comparison Summary

Feature	OCP Route	Gateway API
Basic HTTP Routing	Route spec	HTTPRoute
Path-based Routing	`.spec.path`	`.spec.rules[].matches[].path`
Timeouts	Annotations	HTTPRoute `.spec.rules[].timeouts`
HSTS	Annotation	ResponseHeaderModifier filter
Session Affinity	Annotations	HTTPRoute `.spec.rules[].sessionPersistence`
Header Modification	Annotations	RequestHeaderModifier/ResponseHeaderModi filters

Feature	OCP Route	Gateway API
Connection Limits	Annotation	ClientTrafficPolicy
Rate Limiting	Annotations	BackendTrafficPolicy
IP Allowlist/Blocklist	Annotation	SecurityPolicy with authorization
URL Rewrite	Annotation	URLRewrite filter
Cross-Namespace	Cluster-level policy	Gateway `.spec.listeners[].allowedRoutes`
Default TLS Cert	Controller-level	Gateway listener TLS
TLS Re-encrypt	`.spec.tls.termination: reencrypt`	BackendTLSPolicy
TLS Edge	`.spec.tls.termination: edge`	Gateway listener TLS with HTTPS
TLS Passthrough	`.spec.tls.termination: passthrough`	TLSRoute with Passthrough mode

Feature	OCP Route	Gateway API

Migration Strategy

When migrating from OCP Routes to Gateway API:

1. **Start with a Gateway**: Create a Gateway resource with appropriate listeners for your use case
    - Deploy the Gateway in the same namespace or a dedicated gateway namespace
    - Configure listeners for all protocols you need (HTTP, HTTPS, TLS, TCP, UDP)
    - Ensure the Gateway service is created and has an accessible endpoint
2. **Convert Routes to HTTPRoutes**: Migrate each OCP Route to an HTTPRoute, starting with simple routes
    - Begin with non-production or low-traffic routes to minimize risk
    - Verify hostname and path matching configurations
    - Test basic connectivity before adding advanced features
3. **Apply Policies**: For features that require policies (BackendTrafficPolicy, SecurityPolicy, ClientTrafficPolicy), create and attach them after the basic route is working
    - Add one policy at a time and validate functionality
    - Monitor for any unexpected behavior or performance impact
4. **Test Incrementally**: Validate each migration step before proceeding to the next route
    - **Monitoring**: Check Gateway and Route status conditions for any errors
    - **Functional Testing**: Verify all routes work as expected with curl or automated tests
    - **Performance Testing**: Compare response times and throughput with OCP Routes
    - **Validation Checks**:
        - Verify TLS certificates are correctly applied

- Test session affinity and load balancing behavior
- Validate rate limiting and security policies
- Check header modification and URL rewrites

5. **Update DNS**: Once validated, update DNS entries to point to the new Gateway service

- **Dual-Running Period** (recommended): Point a subset of traffic to the new Gateway while keeping OCP Routes active for rollback capability
- Gradually shift traffic using weighted DNS or canary deployments
- Monitor error rates, latency, and application metrics during cutover

6. **Rollback Procedures**: If issues are discovered after migration

- Revert DNS to point back to OCP Routes
- Keep OCP Routes active for at least 24-48 hours after DNS cutover
- Document any configuration differences that caused issues
- Have a communication plan for stakeholders if rollback is needed

Related Documentation

- Configure GatewayAPI Gateway
- Configure GatewayAPI Route
- Configure GatewayAPI Policy
- Tasks for Envoy Gateway

Trouble Shooting

How to Solve Inter-node Communication Issues in ARM Environments?

When using lower kernel versions and certain domestic network cards, there may be an issue where the network card computes checksums incorrectly after enabling Checksum Offload. This can lead to communication failures between nodes in the Kube-OVN Overlay network. The specific solutions are as follows:

- **Solution 1: Upgrade the Kernel Version**. It is recommended to upgrade the kernel version to 4.19.90-25.16.v2101 or a higher version.
- **Solution 2: Disable Checksum Offload**. If it is not possible to immediately upgrade the kernel version and inter-node communication issues occur, you can disable the Checksum Offload for the physical network card using the following command.

```
ethtool -K eth0 tx off
```

Find Who Cause the Error

The `X-ALB-ERR-REASON` field in the response header of the error request will indicate the reason for the error.

The error reason might be:

```
InvalidBalancer : no balancer found for xx # it means no endpoint found f
or the service
BackendError : read xxx byte data from backend # it means the backend did
give response, the error code is not cause by alb.
InvalidUpstream : no rule match # it means the request does not match any
rule, so alb return 404.
```