OpenShift Container Platform 4.18

Networking overview

Understanding fundamental networking concepts and general tasks in OpenShift Container Platform

Last Updated: 2026-04-01

OpenShift Container Platform 4.18 Networking overview

Understanding fundamental networking concepts and general tasks in OpenShift Container Platform

Legal Notice

Abstract

This document provides an introduction to core networking concepts, basic architecture, and general networking tasks within OpenShift Container Platform.

Table of Contents

CHAPTER 1. UNDERSTANDING NETWORKING

To build resilient and secure applications in OpenShift Container Platform, configure the networking infrastructure for your cluster. Defining reliable pod-to-pod communication and traffic routing rules ensures that every application component functions correctly within the environment.

1.1. NETWORKING IN OPENSHIFT CONTAINER PLATFORM

OpenShift Container Platform ensures seamless communication between various components within the cluster and between external clients and the cluster. Networking relies on the following core concepts and components:

- Pod-to-pod communication
- Services
- DNS
- Ingress
- Network controls
- Load balancing

1.1.1. Common practices for networking services

In OpenShift Container Platform, services create a single IP address for clients to use, even if multiple pods are providing that service. This abstraction enables seamless scaling, fault tolerance, and rolling upgrades without affecting clients.

Network security policies manage traffic within the cluster. Network controls empower namespace administrators to define ingress and egress rules for their pods. By using network administration policies, cluster administrators can establish namespace policies, override namespace policies, or set default policies when none are defined.

Egress firewall configurations control outbound traffic from pods. These configuration settings ensure that only authorized communication occurs. The ingress node firewall protects nodes by controlling incoming traffic. Additionally, the Universal Data Network manages data traffic across the cluster.

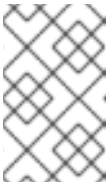
1.1.2. Networking features

OpenShift Container Platform offers several networking features and enhancements. These features and enhancements are listed as follows:

- Ingress Operator and Route API: OpenShift Container Platform includes an Ingress Operator that implements the Ingress Controller API. This component enables external access to cluster services by deploying and managing HAProxy-based Ingress Controllers that support advanced routing configurations and load balancing. OpenShift Container Platform uses the Route API to translate upstream Ingress objects to route objects. Routes are specific to networking in OpenShift Container Platform, but you can also use third-party Ingress Controllers.
- Enhanced security: OpenShift Container Platform provides advanced network security features, such as the egress firewall and the ingress node firewall.
    - Egress firewall: The egress firewall controls and restricts outbound traffic from pods within the cluster. You can set rules to limit which external hosts or IP ranges with which pods can

communicate.

- Ingress node firewall: The ingress node firewall is managed by the Ingress Firewall Operator and provides firewall rules at the node level. You can protect your nodes from threats by configuring this firewall on specific nodes within the cluster to filter incoming traffic before it reaches these nodes.



NOTE

OpenShift Container Platform also implements services, such as Network Policy, Admin Network Policy, and Security Context Constraints (SCC) to secure communication between pods and enforce access controls.

- Role-based access control (RBAC): OpenShift Container Platform extends Kubernetes RBAC to provide more granular control over who can access and manage network resources. RBAC helps maintain security and compliance within the cluster.
- Multi-tenancy support: OpenShift Container Platform offers multi-tenancy support to enable multiple users and teams to share the same cluster while keeping their resources isolated and secure.
- Hybrid and multi-cloud capabilities: OpenShift Container Platform is designed to work seamlessly across on-premise, cloud, and multi-cloud environments. This flexibility allows organizations to deploy and manage containerized applications across different infrastructures.
- Observability and monitoring: OpenShift Container Platform provides integrated observability and monitoring tools that help manage and troubleshoot network issues. These tools include role-based access to network metrics and logs.
- User-defined networks (UDN): UDNs allow administrators to customize network configurations. UDNs provide enhanced network isolation and IP address management.
- Egress IP: Egress IP allows you to assign a fixed source IP address for all egress traffic originating from pods within a namespace. Egress IP can improve security and access control by ensuring consistent source IP addresses for external services. For example, if a pod needs to access an external database that only allows traffic from specific IP adresses, you can configure an egress IP for that pod to meet the access requirements.
- Egress router: An egress router is a pod that acts as a bridge between the cluster and external systems. Egress routers allow traffic from pods to be routed through a specific IP address that is not used for any other purpose. With egress routers, you can enforce access controls or route traffic through a specific gateway.

1.2. NETWORKING WITH NODES, CLIENTS, AND CLUSTERS

A node is a machine in the cluster that can run either control-plane components, workload components, or both. A node is either a physical server or a virtual machine. A cluster is a collection of nodes that run containerized applications. Clients are the tools and users that interact with the cluster.

1.2.1. What is a node?

Nodes are the physical or virtual machines that run containerized applications. Nodes host the pods and provide resources, such as memory and storage for running the applications. Nodes enable communication between pods. Each pod is assigned an IP address. Pods within the same node can communicate with each other using these IP addresses. Nodes facilitate service discovery by allowing

pods to discover and communicate with services within the cluster. Nodes help distribute network traffic among pods to ensure efficient load balancing and high availability of applications. Nodes provide a bridge between the internal cluster network and external networks to allowing external clients to access services running on the cluster.

1.2.2. Understanding clusters

A cluster is a collection of nodes that work together to run containerized applications. These nodes include control plane nodes and compute nodes.

1.2.3. Understanding external clients

An external client is any entity outside the cluster that interacts with the services and applications running within the cluster. External can include end users, external services, and external devices. End users are people who access a web application hosted in the cluster through their browsers or mobile devices. External services are other software systems or applications that interact with the services in the cluster, often through APIs. External devices are any hardware outside the cluster network that needs to communicate with the cluster services, such as the Internet of Things (IoT) devices.

1.3. NETWORKING CONCEPTS AND COMPONENTS

Networking in OpenShift Container Platform uses several key components and concepts.

- Pods and services are the smallest deployable units in Kubernetes, and services provide stable IP addresses and DNS names for sets of pods. Each pod in a cluster is assigned a unique IP address. Pods use IP addresses to communicate directly with other pods, regardless of which node they are on. The pod IP addresses will change when pods are destroyed and created. Services are also assigned unique IP addresses. A service is associated with the pods that can provide the service. When accessed, the service IP address provides a stable way to access pods by sending traffic to one of the pods that backs the service.

- Route and Ingress APIs define rules that route HTTP, HTTPS, and TLS traffic to services within the cluster. OpenShift Container Platform provides both Route and Ingress APIs as part of the default installation, but you can add third-party Ingress Controllers to the cluster.

- The Container Network Interface (CNI) plugin manages the pod network to enable pod-to-pod communication.

- The Cluster Network Operator (CNO) CNO manages the networking plugin components of a cluster. Using the CNO, you can set the network configuration, such as the pod network CIDR and service network CIDR.

- DNS operators manage DNS services within the cluster to ensure that services are reachable by their DNS names.

- Network controls define how pods are allowed to communicate with each other and with other network endpoints. These policies help secure the cluster by controlling traffic flow and enforcing rules for pod communication.

- Load balancing distributes network traffic across multiple servers to ensure reliability and performance.

- Service discovery is a mechanism for services to find and communicate with each other within the cluster.

- The Ingress Operator uses OpenShift Container Platform Route to manage the router and enable external access to cluster services.

Additional resources

- About network policy

1.4. HOW PODS COMMUNICATE

Pods use IP addresses to communicate and a Dynamic Name System (DNS) to discover IP addresses for pods or services. Clusters use various policy types that control what communication is allowed. Pods communicate in two ways: pod-to-pod and service-to-pod.

1.4.1. Pod-to-pod communication

Pod-to-pod communication is the ability of pods to communicate with each other within the cluster. This is crucial for the functioning of microservices and distributed applications.

Each pod in a cluster is assigned a unique IP address that they use to communicate directly with other pods. Pod-to-pod communication is useful for intra-cluster communication where pods need to exchange data or perform tasks collaboratively. For example, Pod A can send requests directly to Pod B using Pod B's IP address. Pods can communicate over a flat network without Network Address Translation (NAT). This allows for seamless communication between pods across different nodes.

1.4.1.1. Example: Controlling pod-to-pod communication

In a microservices-based application with multiple pods, a frontend pod needs to communicate with the a backend pod to retrieve data. By using pod-to-pod communication, either directly or through services, these pods can efficiently exchange information.

To control and secure pod-to-pod communication, you can define network controls. These controls enforce security and compliance requirements by specifying how pods interact with each other based on labels and selectors.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-some-pods
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: app
  ingress:
    - from:
        - podSelector:
            matchLabels:
              role: backend
      ports:
        - protocol: TCP
          port: 80
```

1.4.2. Service-to-pod communication

Service-to-pod communication ensures that services can reliably route traffic to the appropriate pods. Services are objects that define a logical set of pods and provide a stable endpoint, such as IP addresses and DNS names. Pod IP addresses can change. Services abstract pod IP addresses to provide a consistent way to access the application components even as IP addresses change.

Key concepts of service-to-pod communication include:

- Endpoints: Endpoints define the IP addresses and ports of the pods that are associated with a service.
- Selectors: Selectors use labels, such as key-value pairs, to define the criteria for selecting a set of objects that a service should target.
- Services: Services provide a stable IP address and DNS name for a set of pods. This abstraction allows other components to communicate with the service rather than individual pods.
- Service discovery: DNS makes services discoverable. When a service is created, it is assigned a DNS name. Other pods discover this DNS name and use it to communicate with the service.
- Service Types: Service types control how services are exposed within or outside the cluster.
    - ClusterIP exposes the service on an internal cluster IP. It is the default service type and makes the service only reachable from within the cluster.
    - NodePort allows external traffic to access the service by exposing the service on each node's IP at a static port.
    - LoadBalancer uses a cloud provider's load balancer to expose the service externally.

Services use selectors to identify the pods that should receive the traffic. The selectors match labels on the pods to determine which pods are part of the service. Example: A service with the selector **app: myapp** will route traffic to all pods with the label **app: myapp**.

Endpoints are dynamically updated to reflect the current IP addresses of the pods that match the service selector. {product-name} maintains these endpoints and ensures that the service routes traffic to the correct pods.

The communication flow refers to the sequence of steps and interactions that occur when a service in Kubernetes routes traffic to the appropriate pods. The typical communication flow for service-to-pod communication is as follows:

- Service creation: When you create a service, you define the service type, the port on which the service listens, and the selector labels.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

- DNS resolution: Each pod has a DNS name that other pods can use to communicate with the service. For example, if the service is named **my-service** in the **my-app** namespace, its DNS name is **my-service.my-app.svc.cluster.local**.
- Traffic routing: When a pod sends a request to the service's DNS name, OpenShift Container Platform resolves the name to the service's ClusterIP. The service then uses the endpoints to route the traffic to one of the pods that match its selector.
- Load balancing: Services also provide basic load balancing. They distribute incoming traffic across all the pods that match the selector. This ensures that no single pod is overwhelmed with too much traffic.

1.4.2.1. Example: Controlling service-to-pod communication

A cluster is running a microservices-based application with two components: a front-end and a backend. The front-end needs to communicate with the backend to fetch data.

Procedure

1. Create a backend service.

```
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

2. Configure backend pods.

```
apiVersion: v1
kind: Pod
metadata:
  name: backend-pod
  labels:
    app: backend
spec:
  containers:
    - name: backend-container
      image: my-backend-image
      ports:
        - containerPort: 8080
```

3. Establish front-end communication.
   The front-end pods can now use the DNS name **backend.default.svc.cluster.local** to communicate with the backend service. The service ensures that the traffic is routed to one of the backend pods.

Service-to-pod communication abstracts the complexity of managing pod IPs and ensures reliable and efficient communication within the cluster.

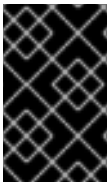
1.5. SUPPORTED LOAD BALANCERS

Load balancing distributes incoming network traffic across multiple servers to maintain the health and efficiency of your clusters by ensuring that no single server bears too much load. Load balancers are devices that perform load balancing. They act as intermediaries between clients and servers to manage and direct traffic based on predefined rules.

OpenShift Container Platform supports the following types of load balancers:

- Classic Load Balancer (CLB)
- Elastic Load Balancing (ELB)
- Network Load Balancer (NLB)
- Application Load Balancer (ALB)

ELB is the default load-balancer type for AWS routers. CLB is the default for self-managed environments. NLB is the default for Red Hat OpenShift Service on AWS (ROSA).



IMPORTANT

Use ALB in front of an application but not in front of a router. Using an ALB requires the AWS Load Balancer Operator add-on. This operator is not supported for all Amazon Web Services (AWS) regions or for all OpenShift Container Platform profiles.

1.5.1. Configuring Load balancers

You can define your default load-balancer type during cluster installation. After installation, you can configure your ingress controller to behave in a specific way that is not covered by the global platform configuration that you defined at cluster installation.

1.5.1.1. Define the default load balancer type

When installing the cluster, you can specify the type of load balancer that you want to use. The type of load balancer you choose at cluster installation gets applied to the entire cluster.

This example shows how to define the default load-balancer type for a cluster deployed on AWS.You can apply the procedure on other supported platforms.

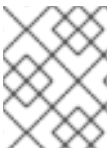
```
apiVersion: v1
kind: Network
metadata:
  name: cluster
platform:
 aws: 1
  lbType: classic 2
```

1 The **platform** key represents the platform on which you have deployed your cluster. This example uses **aws**.

2 The **lbType** key represents the load balancer type. This example uses the Classic Load Balancer, **classic**.

1.5.1.2. Specify load balancer behavior for an Ingress Controller

After you install a cluster, you can configure your Ingress Controller to specify how services are exposed to external networks, so that you can better control the settings and behavior of a load balancer.



NOTE

Changing the load balancer settings on an Ingress Controller might override the load balancer settings you specified at installation.

```
apiVersion: v1
kind: Network
metadata:
  name: cluster
endpointPublishingStrategy:
  loadBalancer: 1
    dnsManagementPolicy: Managed
    providerParameters:
      aws:
        classicLoadBalancer: 2
          connectionIdleTimeout: 0s
        type: Classic
      type: AWS
    scope: External
  type: LoadBalancerService
```

1. The `loadBalancer' field specifies the load balancer configuration settings.
2. The **classicLoadBalancer** field sets the load balancer to **classic** and includes settings specific to the CLB on AWS.

1.6. THE DOMAIN NAME SYSTEM (DNS)

The Domain Name System (DNS) is a hierarchical and decentralized naming system used to translate human-friendly domain names, such as www.example.com, into IP addresses that identify computers on a network. DNS plays a crucial role in service discovery and name resolution.

OpenShift Container Platform provides a built-in DNS to ensure that services can be reached by their DNS names. This helps maintain stable communication even if the underlying IP addresses change. When you start a pod, environment variables for service names, IP addresses, and ports are created automatically to enable the pod to communicate with other services.

1.6.1. Key DNS terms

- CoreDNS: CoreDNS is the DNS server and provides name resolution for services and pods.
- DNS names: Services are assigned DNS names based on their namespace and name. For example, a service named **my-service** in the **default** namespace would have the DNS name **my-service.default.svc.cluster.local**.
- Domain names: Domain names are the human-friendly names used to access websites and services, such as **example.com**.

- IP addresses: IP addresses are numerical labels assigned to each device connected to a computer network that uses IP for communication. An example of an IPv4 address is **192.0.2.1**. An example of an IPv6 address is **2001:0db8:85a3:0000:0000:8a2e:0370:7334**.

- DNS servers: DNS servers are specialized servers that store DNS records. These records map domain names to IP addresses. When you type a domain name into your browser, your computer contacts a DNS server to find the corresponding IP address.

- Resolution process: A DNS query is sent to a DNS resolver. The DNS resolver then contacts a series of DNS servers to find the IP address associated with the domain name. The resolver will try using the name with a series of domains, such as **<namespace>.svc.cluster.local**, **svc.cluster.local**, and **cluster.local**. This process stops at the first match. The IP address is returned to your browser and then connects to the web server using the IP address.

1.6.2. Example: DNS use case

For this example, a front-end application is running in one set of pods and a back-end service is running in another set of pods. The front-end application needs to communicate with the back-end service. You create a service for the back-end pods that gives it a stable IP address and DNS name. The front-end pods use this DNS name to access the back-end service regardless of changes to individual pod IP addresses.

By creating a service for the back-end pods, you provide a stable IP and DNS name, **backend-service.default.svc.cluster.local**, that the front-end pods can use to communicate with the back-end service. This setup would ensure that even if individual pod IP addresses change, the communication remains consistent and reliable.

The following steps demonstrate an example of how to configure front-end pods to communicate with a back-end service using DNS.

1. Create the back-end service.

    a. Deploy the back-end pods.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
  labels:
    app: backend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: backend-container
          image: your-backend-image
          ports:
            - containerPort: 8080
```

b. Define a service to expose the back-end pods.

```
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

2. Create the front-end pods.

a. Define the front-end pods.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
  labels:
    app: frontend
  spec:
    replicas: 3
    selector:
    matchLabels:
      app: frontend
   template:
   metadata:
     labels:
       app: frontend
   spec:
     containers:
       - name: frontend-container
         image: your-frontend-image
         ports:
           - containerPort: 80
```

b. Apply the pod definition to your cluster.

```
$ oc apply -f frontend-deployment.yaml
```

3. Configure the front-end to communicate with the back-end.
In your front-end application code, use the DNS name of the back-end service to send requests. For example, if your front-end application needs to fetch data from the back-end pod, your application might include the following code:

```
fetch('http://backend-service.default.svc.cluster.local/api/data')
  .then(response => response.json())
  .then(data => console.log(data));
```

1.7. NETWORK CONTROLS

Network controls define rules for how pods are allowed to communicate with each other and with other network endpoints. Network controls are implemented at the network level to ensure that only allowed traffic can flow between pods. This helps secure the cluster by restricting traffic flow and preventing unauthorized access.

- Admin network policies (ANP): ANPs are cluster-scoped custom resource definitions (CRDs). As a cluster administrator, you can use an ANP to define network policies at a cluster level. You cannot override these policies by using regular network policy objects. These policies enforce strict network security rules across the entire cluster. ANPs can specify ingress and egress rules to allow administrators to control the traffic that enters and leaves the cluster.

- Egress firewall: The egress firewall restricts egress traffic leaving the cluster. With this firewall, administrators can limit the external hosts that pods can access from within the cluster. You can configure egress firewall policies to allow or deny traffic to specific IP ranges, DNS names, or external services. This helps prevent unauthorized access to external resources and ensures that only allowed traffic can leave the cluster.

- Ingress node firewall: The ingress node firewall controls ingress traffic to the nodes in a cluster. With this firewall, administrators define rules that restrict which external hosts can initiate connections to the nodes. This helps protect the nodes from unauthorized access and ensures that only trusted traffic can reach the cluster.

1.8. ROUTES AND INGRESS

Routes and ingress are both used to expose applications to external traffic. However, they serve slightly different purposes and have different capabilities.

1.8.1. Routes

Routes are specific to OpenShift Container Platform resources that expose a service at a host name so that external clients can reach the service by name.

Routes map a host name to a service. Route name mapping allows external clients to access the service using the host name. Routes provide load balancing for the traffic directed to the service. The host name used in a route is resolved to the IP address of the router. Routes then forward the traffic to the appropriate service. Routes can also be secured using SSL/TLS to encrypt traffic between the client and the service.

1.8.2. Ingress

Ingress is a resource that provides advanced routing capabilities, including load balancing, SSL/TLS termination, and name-based virtual hosting. Here are some key points about Ingress:

- HTTP/HTTPS routing: You can use Ingress to define rules for routing HTTP and HTTPS traffic to services within the cluster.

- Load balancing: Ingress Controllers, such as NGINX or HAProxy, manage traffic routing and load balancing based on user-defined defined rules.

- SSL/TLS termination: SSL/TLS termination is the process of decrypting incoming SSL/TLS traffic before passing it to the backend services.

- Multiple domains and paths: Ingress supports routing traffic for multiple domains and paths.

1.8.3. Comparing Routes and ingress

Routes provide more flexibility and advanced features compared to ingress. This makes routes suitable for complex routing scenarios. Routes are simpler to set up and use, especially for basic external access needs. Ingress is often used for simpler, straightforward external access. Routes are used for more complex scenarios that require advanced routing and SSL/TLS termination.

1.8.4. Example: Configuring routes and ingress to expose a web application

A web application is running on your OpenShift Container Platform cluster. You want to make the application accessible to external users. The application should be accessible through a specific domain name, and the traffic should be securely encrypted using TLS. The following example shows how to configure both routes and ingress to expose your web application to external traffic securely.

1.8.4.1. Configuring Routes

1. Create a new project.

```
$ oc new-project webapp-project
```

2. Deploy the web application.

```
$ oc new-app nodejs:12~https://github.com/sclorg/nodejs-ex.git --name=webapp
```

3. Expose the service with a Route.

```
$ oc expose svc/webapp --hostname=webapp.example.com
```

4. Secure the Route with TLS.

   a. Create a TLS secret with your certificate and key.

```
$ oc create secret tls webapp-tls --cert=path/to/tls.crt --key=path/to/tls.key
```

   b. Update the route to use the TLS secret.

```
$ oc patch route/webapp -p '{"spec":{"tls":
{"termination":"edge","certificate":"path/to/tls.crt","key":"path/to/tls.key"}}}'
```

1.8.4.2. Configuring ingress

1. Create an ingress resource.
   Ensure your ingress Controller is installed and running in the cluster.

2. Create a service for the web application. If not already created, expose the application as a service.

```
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
  namespace: webapp-project
spec:
  selector:
    app: webapp
```

```
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

3. Create the ingress resource.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: webapp-ingress
  namespace: webapp-project
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
    - host: webapp.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: webapp-service
                port:
                  number: 80
```

4. Secure the ingress with TLS.

   a. Create a TLS secret with your certificate and key.

```
$ oc create secret tls webapp-tls --cert=path/to/tls.crt --key=path/to/tls.key -n webapp-project
```

   b. Update the ingress resource to use the TLS secret.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: webapp-ingress
  namespace: webapp-project
spec:
  tls: (1)
    - hosts:
        - webapp.example.com
      secretName: webapp-tls (2)
  rules:
    - host: webapp.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
```

```
      name: webapp-service
      port:
        number: 80
```

1. The **TLS** section specifies TLS settings.
2. The **secretName** field is the name of Kubernetes secret that contains the TLS certificate and key.

1.9. SECURITY AND TRAFFIC MANAGEMENT

Administrators can expose applications to external traffic and secure network connections using service types, such as **ClusterIP**, **NodePort**, and **LoadBalaner** and API resources such as **Ingress** and **Route**. The Ingress Operator and Cluster Network Operator (CNO) help configure and manage these services and resources. The Ingress Operator deploys and manages one or more Ingress Controllers. These controllers route external HTTP and HTTPS traffic to services within the cluster. A CNO deploys and manages the cluster network components, including pod networks, service networks, and DNS.

1.9.1. Exposing applications

ClusterIP exposes services on an internal IP within the cluster to make the cluster accessible only to other services within the cluster. The NodePort service type exposes the service on a static port on each node's IP. This service type allows external traffic to access the service. Load balancers are typically used in cloud or bare-metal environments that use MetalLB. This service type provisions an external load balancer that routes external traffic to the service. On bare-metal environments, MetalLB uses VIPs and ARP announcements or BGP announcements.

Ingress is an API object that manages external access to services, such as load balancing, SSL/TLS termination, and name-based virtual hosting. An Ingress Controller, such as NGINX or HAProxy, implements the Ingress API and handles traffic routing based on user-defined rules.

1.9.2. Securing connections

Ingress Controllers manage SSL/TLS termination to decrypt incoming SSL/TLS traffic before passing it to the backend services. SSL/TLS termination offloads the encryption/decryption process from the application pods. You can use TLS certificates to encrypt traffic between clients and your services. You can manage certificates with tools, such as **cert-manager**, to automate certificate distribution and renewal.

Routes pass TLS traffic to a pod if it has the SNI field. This process allows services that run TCP to be exposed using TLS and not only HTTP/HTTPS. A site administrator can manage the certificates centrally and allow application developers to read private keys even without permission.

The Route API enables encryption of router-to-pod traffic with cluster-managed certificates. This ensures external certificates are centrally managed while the internal leg remains encrypted. Application developers receive unique private keys for their applications. These keys can be mounted as a secret in the pod.

Network controls define rules for how pods can communicate with each other and other network endpoints. This enhances security by controlling traffic flow within the cluster. These controls are implemented at the network plugin level to ensure that only allowed traffic flows between pods.

Role-based access control (RBAC) manages permissions and control who can access resources within the cluster. Service accounts provide identity for pods that access the API. RBAC allows granular control over what each pod can do.

1.9.3. Example: Exposing applications and securing connections

In this example, a web application running in your cluster needs to be accessed by external users.

1. Create a service and expose the application as a service using a service type that suits your needs.

```
apiVersion: v1
kind: Service
metadata:
  name: my-web-app
spec:
  type: LoadBalancer
  selector:
    app: my-web-app
  ports:
  - port: 80
    targetPort: 8080
```

2. Define an **Ingress** resource to manage HTTP/HTTPS traffic and route it to your service.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-web-app-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - host: mywebapp.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-web-app
            port:
              number: 80
```

3. Configure TLS for your ingress to ensure secured, encrypted connections.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-web-app-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  tls:
  - hosts:
```

```
    - mywebapp.example.com
    secretName: my-tls-secret
  rules:
  - host: mywebapp.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-web-app
            port:
              number: 80
```

1.9.4. Choosing between service types and API resources

Service types and API resources offer different benefits for exposing applications and securing network connections. By leveraging the appropriate service type or API resource, you can effectively manage how your applications are exposed and ensure secure, reliable access for both internal and external clients.

OpenShift Container Platform supports the following service types and API resources:

- Service Types
  - **ClusterIP** is intended for internal-only exposure. It is easy to set up and provides a stable internal IP address for accessing services within the cluster. **ClusterIP** is suitable for communication between services within the cluster.
  - **NodePort** allows external access by exposing the service on each node's IP at a static port. It is straightforward to set up and useful for development and testing. **NodePort** is good for simple external access without the need for a load balancer from the cloud provider.
  - **LoadBalancer** automatically provisions an external load balancer to distribute traffic across multiple nodes. It is ideal for production environments where reliable, high-availability access is needed.
  - **ExternalName** maps a service to an external DNS name to allow services outside the cluster to be accessed using the service's DNS name. It is good for integrating external services or legacy systems with the cluster.
  - Headless service is a DNS name that returns the list of pod IPs without providing a stable **ClusterIP**. This is ideal for stateful applications or scenarios where direct access to individual pod IPs is needed.
- API Resources
  - **Ingress** provides control over routing HTTP and HTTPS traffic, including support for load balancing, SSL/TLS termination, and name-based virtual hosting. It is more flexible than services alone and supports multiple domains and paths. **Ingress** is ideal when complex routing is required.
  - **Route** is similar to **Ingress** but provides additional features, including TLS re-encryption and passthrough. It simplifies the process of exposing services externally. **Route** is best for when you need advanced features, such as integrated certificate management.

If you need a simple way to expose a service to external traffic, **Route** or **Ingress** might be the best choice. These resources can be managed by a namespace admin or developer. The easiest approach is to create a route, check its external DNS name, and configure your DNS to have a CNAME that points to the external DNS name.

For HTTP/HTTPS/TLS, **Route** or **Ingress** should suffice. Anything else is more complex and requires a cluster admin to ensure ports are accessible or MetalLB is configured. **LoadBalancer** services are also an option in cloud environments or appropriately configured bare-metal environments.

CHAPTER 2. ACCESSING HOSTS

To establish secure administrative access to OpenShift Container Platform instances and control plane nodes, create a bastion host.

Configuring a bastion host provides an entry point for Secure Shell (SSH) traffic, ensuring that your cluster remains protected while allowing for remote management.

2.1. ACCESSING HOSTS ON AMAZON WEB SERVICES IN AN INSTALLER-PROVISIONED INFRASTRUCTURE CLUSTER

To establish Secure Shell (SSH) access to OpenShift Container Platform hosts on Amazon EC2 instances that lack public IP addresses, configure a bastion host or secure gateway. Defining this access path ensures that you can safely manage and troubleshoot your private infrastructure within an installer-provisioned environment.

Procedure

1. Create a security group that allows SSH access into the virtual private cloud (VPC) that the **openshift-install** command-line interface creates.

2. Create an Amazon EC2 instance on one of the public subnets the installation program created.

3. Associate a public IP address with the Amazon EC2 instance that you created.
   Unlike with the OpenShift Container Platform installation, associate the Amazon EC2 instance you created with an SSH keypair. The operating system selection is not important for this instance, because the instanace serves as an SSH bastion to bridge the internet into the VPC of your OpenShift Container Platform cluster. The Amazon Machine Image (AMI) you use does matter. With Red Hat Enterprise Linux CoreOS (RHCOS), for example, you can provide keys through Ignition by using a similar method to the installation program.

4. After you provisioned your Amazon EC2 instance and can SSH into the instance, add the SSH key that you associated with your OpenShift Container Platform installation. This key can be different from the key for the bastion instance, but this is not a strict requirement.

   NOTE

   Use direct SSH access only for disaster recovery. When the Kubernetes API is responsive, run privileged pods instead.

5. Run **oc get nodes**, inspect the output, and choose one of the nodes that is a control plane. The hostname looks similar to **ip-10-0-1-163.ec2.internal**.

6. From the bastion SSH host that you manually deployed into Amazon EC2, SSH into that control plane host by entering the following command. Ensure that you use the same SSH key that you specified during installation:

   ```
   $ ssh -i <ssh-key-path> core@<control_plane_hostname>
   ```

CHAPTER 3. NETWORKING DASHBOARDS

To monitor and analyze network performance within your cluster, view networking metrics in the OpenShift Container Platform web console. By accessing these dashboards through **Observe → Dashboards**, you can identify traffic patterns and troubleshoot connectivity issues to ensure consistent workload availability.

Network Observability Operator

If you have the Network Observability Operator installed, you can view network traffic metrics dashboards by selecting the **Netobserv** dashboard from the **Dashboards** drop-down list. For more information about metrics available in this **Dashboard**, see Network Observability metrics dashboards.

Networking and OVN-Kubernetes dashboard

You can view both general networking metrics and OVN-Kubernetes metrics from the dashboard. To view general networking metrics, select **Networking/Linux Subsystem Stats** from the **Dashboards** drop-down list. You can view the following networking metrics from the dashboard: **Network Utilisation**, **Network Saturation**, and **Network Errors**.

To view OVN-Kubernetes metrics select **Networking/Infrastructure** from the **Dashboards** drop-down list. You can view the following OVN-Kubernetes metrics: **Networking Configuration**, **TCP Latency Probes**, **Control Plane Resources**, and **Worker Resources**.

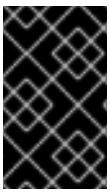
Ingress Operator dashboard

You can view networking metrics handled by the Ingress Operator from the dashboard. This includes metrics like the following:

- Incoming and outgoing bandwidth
- HTTP error rates
- HTTP server response latency
  To view these Ingress metrics, select **Networking/Ingress** from the **Dashboards** drop-down list. You can view Ingress metrics for the following categories: **Top 10 Per Route**, **Top 10 Per Namespace**, and **Top 10 Per Shard**

CHAPTER 4. CIDR RANGE DEFINITIONS

To ensure stable and accurate network routing in OpenShift Container Platform clusters that use OVN-Kubernetes, define non-overlapping Classless Inter-Domain Routing (CIDR) subnet ranges. Establishing unique ranges prevents IP address conflicts so that internal traffic reaches its intended destination without interference.



IMPORTANT

For OpenShift Container Platform 4.17 and later versions, clusters use **169.254.0.0/17** for IPv4 and **fd69::/112** for IPv6 as the default masquerade subnet. You must avoid these ranges. For upgraded clusters, there is no change to the default masquerade subnet.

The following subnet types and are mandatory for a cluster that uses OVN-Kubernetes:

- Join: Uses a join switch to connect gateway routers to distributed routers. A join switch reduces the number of IP addresses for a distributed router. For a cluster that uses the OVN-Kubernetes plugin, an IP address from a dedicated subnet is assigned to any logical port that attaches to the join switch.
- Masquerade: Prevents collisions for identical source and destination IP addresses that are sent from a node as hairpin traffic to the same node after a load balancer makes a routing decision.
- Transit: A transit switch is a type of distributed switch that spans across all nodes in the cluster. A transit switch routes traffic between different zones. For a cluster that uses the OVN-Kubernetes plugin, an IP address from a dedicated subnet is assigned to any logical port that attaches to the transit switch.

NOTE

You can change the join, masquerade, and transit CIDR ranges for your cluster as a postinstallation task.

OVN-Kubernetes, the default network provider in OpenShift Container Platform 4.14 and later versions, internally uses the following IP address subnet ranges:

- **V4JoinSubnet**: **100.64.0.0/16**
- **V6JoinSubnet**: **fd98::/64**
- **V4TransitSwitchSubnet**: **100.88.0.0/16**
- **V6TransitSwitchSubnet**: **fd97::/64**
- **defaultV4MasqueradeSubnet**: **169.254.0.0/17**
- **defaultV6MasqueradeSubnet**: **fd69::/112**

IMPORTANT

The earlier list includes join, transit, and masquerade IPv4 and IPv6 address subnets. If your cluster uses OVN-Kubernetes, do not include any of these IP address subnet ranges in any other CIDR definitions in your cluster or infrastructure.

Additional resources

- Configuring OVN-Kubernetes internal IP address subnets

4.1. MACHINE CIDR

To establish the network scope for cluster nodes in OpenShift Container Platform, specify an IP address range in the Machine Classless Inter-Domain Routing (CIDR) parameter. Defining this range ensures that all machines within the environment have valid, routable addresses for internal cluster communication.

NOTE

You cannot change Machine CIDR ranges after you create your cluster.

The default is **10.0.0.0/16**. This range must not conflict with any connected networks.

Additional resources

- Cluster Network Operator configuration

4.2. SERVICE CIDR

To allocate IP addresses for cluster services in OpenShift Container Platform, specify an IP address range in the Service Classless Inter-Domain Routing (CIDR) parameter. Defining this range ensures that internal services have a dedicated block of addresses for reliable communication without overlapping with node or pod networks.

The range must be large enough to accommodate your workload. The address block must not overlap with any external service accessed from within the cluster. The default is **172.30.0.0/16**.

4.3. POD CIDR

To allocate internal network addresses for cluster workloads in OpenShift Container Platform, specify an IP address range in the pod Classless Inter-Domain Routing (CIDR) field. Defining this range ensures that pods can communicate with each other reliably without overlapping with the node or service networks.

The pod CIDR is the same as the **clusterNetwork** CIDR and the cluster CIDR. The range must be large enough to accommodate your workload. The address block must not overlap with any external service accessed from within the cluster. The default is **10.128.0.0/14**. You can expand the range after cluster installation.

Additional resources

- Cluster Network Operator configuration
- Configuring the cluster network range

4.4. HOST PREFIX

To allocate a dedicated pool of IP addresses for pods on each node in OpenShift Container Platform, specify the subnet prefix length in the hostPrefix parameter. Defining an appropriate prefix ensures that

every machine has sufficient unique addresses to support its scheduled workloads without exhausting the cluster's network resources.

For example, if the host prefix is set to **/23**, each machine is assigned a **/23** subnet from the pod CIDR address range. The default is **/23**, allowing 510 cluster nodes and 510 pod IP addresses per node.

Consider another example where you set the **clusterNetwork.cidr** parameter to **10.128.0.0/16**, you define the complete address space for the cluster. This assigns a pool of 65,536 IP addresses to your cluster. If you then set the **hostPrefix** parameter to **/23**, you define a subnet slice to each node in the cluster, where the **/23** slice becomes a subnet of the **/16** subnet network. This assigns 512 IP addresses to each node, where 2 IP addresses get reserved for networking and broadcasting purposes. The following example calculation uses these IP address figures to determine the maximum number of nodes that you can create for your cluster:

```
65536 / 512 = 128
```

You can use the Red Hat OpenShift Network Calculator to calculate the maximum number of nodes for your cluster.

4.5. CIDR RANGES FOR HOSTED CONTROL PLANES

To successfully deploy hosted control planes on OpenShift Container Platform, define the network environment by using specific Classless Inter-Domain Routing (CIDR) subnet ranges. Establishing these nonoverlapping ranges ensures reliable communication between cluster components and prevents internal IP address conflicts.

The following Classless Inter-Domain Routing (CIDR) subnet ranges are the default settings for hosted control planes:

- **v4InternalSubnet**: 100.65.0.0/16 (OVN-Kubernetes)
- **clusterNetwork**: 10.132.0.0/14 (pod network)
- **serviceNetwork**: 172.31.0.0/16

By using one of the default subnet ranges, you can avoid CIDR overlap with the management cluster and avoid connectivity issues. However, you can use other CIDR subnet ranges if they do not overlap with the management cluster.