



Network Policies

Network Policies

If you want to control traffic flow at the IP address or port level (OSI layer 3 or 4), NetworkPolicies allow you to specify rules for traffic flow within your cluster, and also between Pods and the outside world. Your cluster must use a network plugin that supports NetworkPolicy enforcement.

If you want to control traffic flow at the IP address or port level for TCP, UDP, and SCTP protocols, then you might consider using Kubernetes NetworkPolicies for particular applications in your cluster. NetworkPolicies are an application-centric construct which allow you to specify how a pod is allowed to communicate with various network "entities" (we use the word "entity" here to avoid overloading the more common terms such as "endpoints" and "services", which have specific Kubernetes connotations) over the network. NetworkPolicies apply to a connection with a pod on one or both ends, and are not relevant to other connections.

The entities that a Pod can communicate with are identified through a combination of the following three identifiers:

1. Other pods that are allowed (exception: a pod cannot block access to itself)
2. Namespaces that are allowed
3. IP blocks (exception: traffic to and from the node where a Pod is running is always allowed, regardless of the IP address of the Pod or the node)

When defining a pod- or namespace-based NetworkPolicy, you use a selector to specify what traffic is allowed to and from the Pod(s) that match the selector.

Meanwhile, when IP-based NetworkPolicies are created, we define policies based on IP blocks (CIDR ranges).

Prerequisites

Network policies are implemented by the [network plugin](#). To use network policies, you must be using a networking solution which supports NetworkPolicy. Creating a NetworkPolicy resource without a controller that implements it will have no effect.

The two sorts of pod isolation

There are two sorts of isolation for a pod: isolation for egress, and isolation for ingress. They concern what connections may be established. "Isolation" here is not absolute, rather it means "some restrictions apply". The alternative, "non-isolated for \$direction", means that no restrictions apply in the stated direction. The two sorts of isolation (or not) are declared independently, and are both relevant for a connection from one pod to another.

By default, a pod is non-isolated for egress; all outbound connections are allowed. A pod is isolated for egress if there is any NetworkPolicy that both selects the pod and has "Egress" in its `policyTypes`; we say that such a policy applies to the pod for egress. When a pod is isolated for egress, the only allowed connections from the pod are those allowed by the `egress` list of some NetworkPolicy that applies to the pod for egress. Reply traffic for those allowed connections will also be implicitly allowed. The effects of those `egress` lists combine additively.

By default, a pod is non-isolated for ingress; all inbound connections are allowed. A pod is isolated for ingress if there is any NetworkPolicy that both selects the pod and has "Ingress" in its `policyTypes`; we say that such a policy applies to the pod for ingress. When a pod is isolated for ingress, the only allowed connections into the pod are those from the pod's node and those allowed by the `ingress` list of some NetworkPolicy that applies to the pod for ingress. Reply traffic for those allowed connections will also be implicitly allowed. The effects of those `ingress` lists combine additively.

Network policies do not conflict; they are additive. If any policy or policies apply to a given pod for a given direction, the connections allowed in that direction from that pod is the union of what the applicable policies allow. Thus, order of evaluation does not affect the policy result.

For a connection from a source pod to a destination pod to be allowed, both the egress policy on the source pod and the ingress policy on the destination pod need to allow the connection. If either side does not allow the connection, it will not happen.

The NetworkPolicy resource

See the [NetworkPolicy](#) reference for a full definition of the resource.

An example NetworkPolicy might look like this:

[service/networking/networkpolicy.yaml](#)



```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - ipBlock:
            cidr: 172.17.0.0/16
            except:
              - 172.17.1.0/24
        - namespaceSelector:
            matchLabels:
              project: myproject
        - podSelector:
            matchLabels:
              role: frontend
      ports:
        - protocol: TCP
          port: 6379
  egress:
    - to:
        - ipBlock:
            cidr: 10.0.0.0/24
      ports:
        - protocol: TCP
          port: 5978
```

Note:

POSTing this to the API server for your cluster will have no effect unless your chosen networking solution supports network policy.

Mandatory Fields: As with all other Kubernetes config, a NetworkPolicy needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see [Configure a Pod to Use a ConfigMap](#), and [Object Management](#).

spec: NetworkPolicy `spec` has all the information needed to define a particular network policy in the given namespace.

podSelector: Each NetworkPolicy includes a `podSelector` which selects the grouping of pods to which the policy applies. The example policy selects pods with the label "role=db". An empty `podSelector` selects all pods in the namespace.

policyTypes: Each NetworkPolicy includes a `policyTypes` list which may include either `Ingress`, `Egress`, or both. The `policyTypes` field indicates whether or not the given policy applies to ingress traffic to selected pod, egress traffic from selected pods, or both. If no `policyTypes` are specified on a NetworkPolicy then by default `Ingress` will always be set and `Egress` will be set if the NetworkPolicy has any egress rules.

ingress: Each NetworkPolicy may include a list of allowed `ingress` rules. Each rule allows traffic which matches both the `from` and `ports` sections. The example policy contains a single rule, which matches traffic on a single port, from one of three sources, the first specified via an `ipBlock`, the second via a `namespaceSelector` and the third via a `podSelector`.

egress: Each NetworkPolicy may include a list of allowed `egress` rules. Each rule allows traffic which matches both the `to` and `ports` sections. The example policy contains a single rule, which matches traffic on a single port to any destination in `10.0.0.0/24`.

So, the example NetworkPolicy:

1. isolates `role=db` pods in the `default` namespace for both ingress and egress traffic (if they weren't already isolated)
2. (Ingress rules) allows connections to all pods in the `default` namespace with the label `role=db` on TCP port 6379 from:
 - any pod in the `default` namespace with the label `role=frontend`
 - any pod in a namespace with the label `project=myproject`
 - IP addresses in the ranges `172.17.0.0 - 172.17.0.255` and `172.17.2.0 - 172.17.255.255` (ie, all of `172.17.0.0/16` except `172.17.1.0/24`)
3. (Egress rules) allows connections from any pod in the `default` namespace with the label `role=db` to CIDR `10.0.0.0/24` on TCP port 5978

See the [Declare Network Policy](#) walkthrough for further examples.

Behavior of to and from selectors

There are four kinds of selectors that can be specified in an `ingress` `from` section or `egress` `to` section:

podSelector: This selects particular Pods in the same namespace as the NetworkPolicy which should be allowed as ingress sources or egress destinations.

namespaceSelector: This selects particular namespaces for which all Pods should be allowed as ingress sources or egress destinations.

namespaceSelector and podSelector: A single `to / from` entry that specifies both `namespaceSelector` and `podSelector` selects particular Pods within particular namespaces. Be careful to use correct YAML syntax. For example:

```
...
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        user: alice
    podSelector:
      matchLabels:
        role: client
...

```

This policy contains a single `from` element allowing connections from Pods with the label `role=client` in namespaces with the label `user=alice`. But the following policy is different:

```
...
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        user: alice

```

```
- podSelector:
  matchLabels:
    role: client
...
```

It contains two elements in the `from` array, and allows connections from Pods in the local Namespace with the label `role=client` , or from any Pod in any namespace with the label `user=alice` .

When in doubt, use `kubectl describe` to see how Kubernetes has interpreted the policy.

ipBlock: This selects particular IP CIDR ranges to allow as ingress sources or egress destinations. These should be cluster-external IPs, since Pod IPs are ephemeral and unpredictable.

Cluster ingress and egress mechanisms often require rewriting the source or destination IP of packets. In cases where this happens, it is not defined whether this happens before or after NetworkPolicy processing, and the behavior may be different for different combinations of network plugin, cloud provider, `Service` implementation, etc.

In the case of ingress, this means that in some cases you may be able to filter incoming packets based on the actual original source IP, while in other cases, the "source IP" that the NetworkPolicy acts on may be the IP of a `LoadBalancer` or of the Pod's node, etc.

For egress, this means that connections from pods to `Service` IPs that get rewritten to cluster-external IPs may or may not be subject to `ipBlock` -based policies.

Default policies

By default, if no policies exist in a namespace, then all ingress and egress traffic is allowed to and from pods in that namespace. The following examples let you change the default behavior in that namespace.

Default deny all ingress traffic

You can create a "default" ingress isolation policy for a namespace by creating a NetworkPolicy that selects all pods but does not allow any ingress traffic to those pods.



```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

This ensures that even pods that aren't selected by any other NetworkPolicy will still be isolated for ingress. This policy does not affect isolation for egress from any pod.

Allow all ingress traffic

If you want to allow all incoming connections to all pods in a namespace, you can create a policy that explicitly allows that.

[service/networking/network-policy-allow-all-ingress.yaml](#) 

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-ingress
spec:
  podSelector: {}
  ingress:
  - {}
  policyTypes:
  - Ingress
```

With this policy in place, no additional policy or policies can cause any incoming connection to those pods to be denied. This policy has no effect on isolation for egress from any pod.

Default deny all egress traffic

You can create a "default" egress isolation policy for a namespace by creating a NetworkPolicy that selects all pods but does not allow any egress traffic from those pods.

[service/networking/network-policy-default-deny-egress.yaml](#) 

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-egress
spec:
  podSelector: {}
  policyTypes:
  - Egress
```

This ensures that even pods that aren't selected by any other NetworkPolicy will not be allowed egress traffic. This policy does not change the ingress isolation behavior of any pod.

Caution:

A default deny-all egress policy also blocks DNS traffic. If your workloads need DNS resolution, you must add a separate NetworkPolicy that allows egress to your cluster's DNS service.

Allow all egress traffic

If you want to allow all connections from all pods in a namespace, you can create a policy that explicitly allows all outgoing connections from pods in that namespace.

[service/networking/network-policy-allow-all-egress.yaml](#) 

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-egress
spec:
  podSelector: {}
```

egress:

- {}

policyTypes:

- Egress

With this policy in place, no additional policy or policies can cause any outgoing connection from those pods to be denied. This policy has no effect on isolation for ingress to any pod.

Default deny all ingress and all egress traffic

You can create a "default" policy for a namespace which prevents all ingress AND egress traffic by creating the following NetworkPolicy in that namespace.

[service/networking/network-policy-default-deny-all.yaml](#)



```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
```

This ensures that even pods that aren't selected by any other NetworkPolicy will not be allowed ingress or egress traffic.

Network traffic filtering

NetworkPolicy is defined for [layer 4](#) connections (TCP, UDP, and optionally SCTP). For all the other protocols, the behaviour may vary across network plugins.

Note:

You must be using a CNI plugin that supports SCTP protocol NetworkPolicies.

When a `deny all` network policy is defined, it is only guaranteed to deny TCP, UDP and SCTP connections. For other protocols, such as ARP or ICMP, the behaviour is undefined. The same applies to allow rules: when a specific pod is allowed as ingress source or egress destination, it is undefined what happens with (for example) ICMP packets. Protocols such as ICMP may be allowed by some network plugins and denied by others.

Targeting a range of ports

 **FEATURE STATE:** Kubernetes v1.25 [stable]

When writing a NetworkPolicy, you can target a range of ports instead of a single port.

This is achievable with the usage of the `endPort` field, as the following example:

[service/networking/networkpolicy-multiport-egress.yaml](#)



```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: multi-port-egress
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Egress
  egress:
    - to:
        - ipBlock:
            cidr: 10.0.0.0/24
      ports:
        - protocol: TCP
          port: 32000
          endPort: 32768
```

The above rule allows any Pod with label `role=db` on the namespace `default` to communicate with any IP within the range `10.0.0.0/24` over TCP, provided that the target port is between the range 32000 and 32768.

The following restrictions apply when using this field:

- The `endPort` field must be equal to or greater than the `port` field.
- `endPort` can only be defined if `port` is also defined.
- Both ports must be numeric.

Note:

Your cluster must be using a CNI plugin that supports the `endPort` field in NetworkPolicy specifications. If your [network plugin](#) does not support the `endPort` field and you specify a NetworkPolicy with that, the policy will be applied only for the single `port` field.

Targeting multiple namespaces by label

In this scenario, your `Egress` NetworkPolicy targets more than one namespace using their label names. For this to work, you need to label the target namespaces. For example:

```
kubectl label namespace frontend namespace=frontend
kubectl label namespace backend namespace=backend
```

Add the labels under `namespaceSelector` in your NetworkPolicy document. For example:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: egress-namespaces
spec:
  podSelector:
    matchLabels:
      app: myapp
  policyTypes:
    - Egress
  egress:
    - to:
        - namespaceSelector:
            matchExpressions:
              - key: namespace
                operator: In
                values: ["frontend", "backend"]
```

Note:

It is not possible to directly specify the name of the namespaces in a NetworkPolicy. You must use a namespaceSelector with matchLabels or matchExpressions to select the namespaces based on their labels.

Targeting a Namespace by its name

The Kubernetes control plane sets an immutable label `kubernetes.io/metadata.name` on all namespaces, the value of the label is the namespace name.

While NetworkPolicy cannot target a namespace by its name with some object field, you can use the standardized label to target a specific namespace.

Pod lifecycle

Note:

The following applies to clusters with a conformant networking plugin and a conformant implementation of NetworkPolicy.

When a new NetworkPolicy object is created, it may take some time for a network plugin to handle the new object. If a pod that is affected by a NetworkPolicy is created before the network plugin has completed NetworkPolicy handling, that pod may be started unprotected, and isolation rules will be applied when the NetworkPolicy handling is completed.

Once the NetworkPolicy is handled by a network plugin,

1. All newly created pods affected by a given NetworkPolicy will be isolated before they are started. Implementations of NetworkPolicy must ensure that filtering is effective throughout the Pod lifecycle, even from the very first instant that any container in that Pod is started. Because they are applied at Pod level, NetworkPolicies apply equally to init containers, sidecar containers, and regular containers.
2. Allow rules will be applied eventually after the isolation rules (or may be applied at the same time). In the worst case, a newly created pod may have no network connectivity at all when it is first started, if isolation rules were already applied, but no allow rules were applied yet.

Every created NetworkPolicy will be handled by a network plugin eventually, but there is no way to tell from the Kubernetes API when exactly that happens.

Therefore, pods must be resilient against being started up with different network connectivity than expected. If you need to make sure the pod can reach certain destinations before being started, you can use an [init container](#) to wait for those destinations to be reachable before kubelet starts the app containers.

Every NetworkPolicy will be applied to all selected pods eventually. Because the network plugin may implement NetworkPolicy in a distributed manner, it is possible that pods may see a slightly inconsistent view of network policies when the pod is first created, or when pods or policies change. For example, a newly-created pod that is supposed to be able to reach both Pod A on Node 1 and Pod B on Node 2 may find that it can reach Pod A immediately, but cannot reach Pod B until a few seconds later.

NetworkPolicy and hostNetwork pods

NetworkPolicy behaviour for `hostNetwork` pods is undefined, but it should be limited to 2 possibilities:

- The network plugin can distinguish `hostNetwork` pod traffic from all other traffic (including being able to distinguish traffic from different `hostNetwork` pods on the same node), and will apply NetworkPolicy to `hostNetwork` pods just like it does to pod-network pods.
- The network plugin cannot properly distinguish `hostNetwork` pod traffic, and so it ignores `hostNetwork` pods when matching `podSelector` and `namespaceSelector`. Traffic to/from `hostNetwork` pods is treated the same as all other traffic to/from the node IP. (This is the most common implementation.)

This applies when

1. a `hostNetwork` pod is selected by `spec.podSelector`.

```
...
spec:
  podSelector:
    matchLabels:
      role: client
...
```

2. a `hostNetwork` pod is selected by a `podSelector` or `namespaceSelector` in an `ingress` or `egress` rule.

```
...
ingress:
```

```
- from:
  - podSelector:
      matchLabels:
        role: client
...

```

At the same time, since `hostNetwork` pods have the same IP addresses as the nodes they reside on, their connections will be treated as node connections. For example, you can allow traffic from a `hostNetwork` Pod using an `ipBlock` rule.

What you can't do with network policies (at least, not yet)

As of Kubernetes 1.36, the following functionality does not exist in the NetworkPolicy API, but you might be able to implement workarounds using Operating System components (such as SELinux, OpenVSwitch, IPTables, and so on) or Layer 7 technologies (Ingress controllers, Service Mesh implementations) or admission controllers. In case you are new to network security in Kubernetes, it's worth noting that the following User Stories cannot (yet) be implemented using the NetworkPolicy API.

- Forcing internal cluster traffic to go through a common gateway (this might be best served with a service mesh or other proxy).
- Anything TLS related (use a service mesh or ingress controller for this).
- Node specific policies (you can use CIDR notation for these, but you cannot target nodes by their Kubernetes identities specifically).
- Targeting of services by name (you can, however, target pods or namespaces by their labels, which is often a viable workaround).
- Creation or management of "Policy requests" that are fulfilled by a third party.
- Default policies which are applied to all namespaces or pods (there are some third party Kubernetes distributions and projects which can do this).
- Advanced policy querying and reachability tooling.
- The ability to log network security events (for example connections that are blocked or accepted).
- The ability to explicitly deny policies (currently the model for NetworkPolicies are deny by default, with only the ability to add allow rules).
- The ability to prevent loopback or incoming host traffic (Pods cannot currently block localhost access, nor do they have the ability to block access from their resident

node).

NetworkPolicy's impact on existing connections

When the set of NetworkPolicies that applies to an existing connection changes - this could happen either due to a change in NetworkPolicies or if the relevant labels of the namespaces/pods selected by the policy (both subject and peers) are changed in the middle of an existing connection - it is implementation defined as to whether the change will take effect for that existing connection or not. Example: A policy is created that leads to denying a previously allowed connection, the underlying network plugin implementation is responsible for defining if that new policy will close the existing connections or not. It is recommended not to modify policies/pods/namespaces in ways that might affect existing connections.

What's next

- See the [Declare Network Policy](#) walkthrough for further examples.
- See more [recipes](#) for common scenarios enabled by the NetworkPolicy resource.

Feedback

Was this page helpful?

Yes

No

Last modified April 14, 2026 at 1:15 AM PST: [fix\(links\): update kubernetes/community links from master to main \(03c191bcc4\)](#)