**Azure CNI vs Azure CNI Overlay**

**Azure CNI (Classic / VNET-Integrated)**

* **Pod IPs come directly from the subnet you provide in your VNet.**
* This means each pod is a first-class citizen on the VNet and can directly communicate with other Azure services (e.g., databases, VMs) using private IPs.

✅ Azure CNI Overlay

* Only the node itself has an IP from the subnet/VNet.
* Avoids consuming your Azure VNet subnet IPs for every pod

[Azure VNet/Subnet] (e.g. 10.10.0.0/16)

|

|-----> Nodes get IPs from VNet (e.g., 10.10.1.4)

|

|-----> Pods get IPs from separate CIDR (Overlay range, e.g., 192.168.0.0/16)

(Not routed via Azure VNet)

 The node is connected to Azure VNet

 Pods are assigned IPs from Overlay CIDR (e.g., 192.168.x.x)

 Node uses VXLAN encapsulation to route pod-to-pod traffic

 Outbound pod traffic is NAT'ed to the node’s IP before leaving VNet

**A screenshot of a computer

AI-generated content may be incorrect.**

**💡 Example**

* You created AKS with Azure CNI (classic), subnet range 10.240.0.0/16, and each pod gets an IP like 10.240.0.5, 10.240.0.6, etc.
* In CNI Overlay, the node may have IP 10.240.0.4, but pods on the node use something like 192.168.0.x (from the overlay CIDR 100.64.0.0/16).

**2. What Happens When You Enable Virtual Nodes?**

* Virtual nodes are **based on Azure Container Instances (ACI)**.
* They allow you to burst workloads **without needing to provision new VMs**.
* **When enabled, AKS uses CNI Overlay networking by default.**

Step 1: Choose CIDR for pod IPs (overlay)

az aks create \

--name aks-overlay-demo \

--resource-group myrg \

--network-plugin azure \

--network-plugin-mode overlay \

--pod-cidr 192.168.0.0/16 \

--service-cidr 10.0.0.0/16 \

--dns-service-ip 10.0.0.10 \

--vnet-subnet-id /subscriptions/xxxx/resourceGroups/myrg/providers/Microsoft.Network/virtualNetworks/myVnet/subnets/aks-subnet \

--generate-ssh-keys

kubectl run nginx --image=nginx

kubectl get pod -o wide

kubectl exec -it nginx -- curl google.com

* Traffic will NAT via node IP to the Internet

**Step 3: Try to ping Azure SQL or another service in VNet**

* Will fail **unless NAT rules or Private Endpoints** are configured

Cannot directly allow Azure services to talk to pod IP

**Design Choices:**

* **AKS with Azure CNI** (so pods have real VNet IPs to talk to SQL and Redis)
* **Application Gateway Ingress Controller** (for WAF and HTTPS)
* **Large subnet** like /22 to avoid IP exhaustion
* **Node Pools**:
  + System: 2 Standard\_D4s\_v3
  + User: 5 autoscaled nodes Standard\_D8s\_v3
* **Private Cluster** with Azure Bastion to connect
* **Pod Disruption Budget (PDB)** set for mission-critical services
* **Rolling updates** with maxSurge: 1 and maxUnavailable: 0
* **RBAC, NSGs, Azure AD integration** for identity and access control

**🌐 Kubernetes Networking & Service Discovery — Beginner to Pro**

**🧱 1. Kubernetes Networking Model (Flat Network)**

**Kubernetes networking is the system that allows Pods to talk to each other and to the outside world.**

**Core Rules of Kubernetes Networking:**

1. Each Pod gets its own unique IP address
2. All Pods can communicate with all other Pods directly — no NAT required
3. Communication is flat: no gateways or routers between Pods
4. Containers in the same Pod share the same network namespace (they can use localhost to talk)

**We'll cover:**

1. ✅ What is a Virtual Network (VNet) in AKS
2. ✅ What is a subnet and how /16, /24 ranges work
3. ✅ What are Pod CIDR, Service CIDR, and how Azure assigns them
4. ✅ What is DNS IP and how CoreDNS works
5. ✅ How this relates to your current AKS VNet setup

**🏢 Real-Life Analogy: Office Building**

**Think of a VNet as a company office building:**

* It has many rooms (like subnets)
* Employees (like VMs or AKS nodes) sit in different rooms
* Everyone inside can talk to each other without using public internet
* Security guards (NSGs/firewalls) control who can enter or leave

**You have:**

* **AKS cluster (in one subnet)**
* **Azure SQL DB (in another subnet)**
* **You don’t want this traffic to go to the internet**

**✅ Put both in the same VNet, and they’ll talk privately without going outside.**

**🔗 PART 2: What is a Subnet?**

✅ Basic Definition:

A subnet is a smaller section inside a VNet.

It divides your VNet into logical zones.

| **Subnet Name** | **Purpose** |
| --- | --- |
| aks-subnet | Hosts AKS nodes and Pod IPs |
| appgateway-subnet | Hosts Azure Application Gateway |
| database-subnet | Hosts Azure SQL / DB resources |

**PART 3: What is CIDR and /16, /24, /12?**

**✅ CIDR = Classless Inter-Domain Routing**

**CIDR defines how many IPs are available in a subnet.**

| **CIDR Notation** | **IP Range Size** | **Example** |
| --- | --- | --- |
| **/24** | **256 IPs** | **10.0.0.0 → 10.0.0.255** |
| **/16** | **65,536 IPs** | **10.0.0.0 → 10.0.255.255** |
| **/12** | **1 million+ IPs** | **10.0.0.0 → 10.15.255.255** |

**🐳 PART 4: How Does This Connect to AKS?**

**AKS needs:**

* **A subnet to host the VMs (nodes)**
* **A pool of IPs for Pods**
* **A way for Services and Ingress Controllers to expose workloads**

**How do two apps in the cluster communicate?**

✅ Inside Kubernetes Cluster (Pod-to-Pod Communication):

* Each Pod gets a unique IP address from the Pod CIDR range.
* Kubernetes uses a flat network model:
  + All Pods can reach each other across namespaces and nodes without NAT.

No need to expose the app externally to reach it within the cluster.

**✅ Pod-to-Pod communication using direct IP addresses inside an AKS (or general Kubernetes) cluster.**

**✅ 2️⃣ Deploy a Backend Pod That Responds to HTTP**

# backend-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: backend

namespace: demo-ip

labels:

app: backend

spec:

containers:

- name: backend

image: hashicorp/http-echo

args:

- "-text=Hello from Backend via IP"

ports:

- containerPort: 5678

kubectl apply -f backend-pod.yaml

**✅ 3️⃣ Get Backend Pod IP**

After the backend pod is running:

bash

CopyEdit

kubectl get pod -n demo-ip -o wide

**✅ 4️⃣ Deploy a Frontend Pod to Curl the Backend Pod**

Now let’s deploy a curl-based Pod that will attempt to connect to the backend **using its Pod IP**:

# frontend-curl.yaml

apiVersion: v1

kind: Pod

metadata:

name: frontend

namespace: demo-ip

spec:

containers:

- name: curl

image: curlimages/curl

command: [ "sleep", "3600" ]

**kubectl apply -f frontend-curl.yaml**

**kubectl exec -n demo-ip -it frontend – sh**

**curl http://<backend\_pod\_ip>:5678**

**Virtual network/subnet address space**

Kubectl get nodes -o wide

10.22.0.4

Check the Virtual Network



Check Subnet range

Under Virtual Network screen , 🡪settings 🡺subnets

A white screen with blue text

AI-generated content may be incorrect.

To check connected devices

A screenshot of a computer

AI-generated content may be incorrect.

**POD CIDR**

Represent the address range from where the pods will assign the IP address.

But this address range should not be used in any other network environment that AKS use or connect to like AKS subnet. From AKS Vnet.

This range can’t changed after cluster creation

Kubectl get pod -o wide

10.244.1.17

10.244.1.18

A screenshot of a computer

AI-generated content may be incorrect.

**sipcalc 10.244.0.0./16 to find the range**

**network range : 10.244.0.1 to 10.244.255.254**

**Service CIDR**

Represent the address range that internal services will assign the IP address from .

Like the IP addresses assigned to ClusterIP,LoadBalancer

A screenshot of a computer

AI-generated content may be incorrect.

**LoadBalancer**

**By default ,AKS create public loadbalancer named as Kubernetes ,**

**If you create service as loadbalancer , Kubernetes will provide the public IP.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Frontend of the LoadBalancer**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**🚪 What is Pod Isolation?**

Pod isolation means controlling which Pods can communicate with each other.

In a default Kubernetes setup:

* All pods can talk to all other pods, across all namespaces.
* This is not secure by default (no zero-trust).
* Pod isolation is achieved using Network Policies.

**🧩 Two Types of Isolation**

| **Type** | **Controls...** | **Enabled via** |
| --- | --- | --- |
| 🔐 Ingress | Who can talk to a pod | ingress in a NetworkPolicy |
| 🚫 Egress | Who a pod can talk to | egress in a NetworkPolicy |

**🧪 How Does Pod Isolation Work?**

Pod isolation is not automatic.

A pod becomes "isolated" only when a matching NetworkPolicy exists that selects the pod — and defines allowed traffic.

Let’s see how

**🚦 What is a Network Policy?**

Think of it like a **firewall for Pods** — it controls **who can talk to whom** inside your Kubernetes cluster.

**🎯 Why Do You Need Network Policies?**

Without network policy:

* ✅ All Pods can talk to all other Pods.
* ❌ There’s **no isolation** between frontend, backend, database, etc.
* ❌ Security risk if one pod is hacked — it can reach everything!

With network policy:

* 🔐 You can allow only **specific Pods or Namespaces** to talk.
* 🔒 You can **block egress** to the internet or IP ranges.
* 👮 You define “who can talk to whom” inside your cluster.

| **Mode** | **Enforced by** | **Supports Ingress** | **Supports Egress** | **Supports IPBlock** | **Use Case** |
| --- | --- | --- | --- | --- | --- |
| None | ❌ No enforcement | ❌ No restrictions | ❌ No restrictions | ❌ No rules | Dev/test clusters |
| Azure | ✅ Azure NSG | ✅ Yes | ❌ No | ❌ No | Basic secure apps |
| Calico | ✅ Calico agent | ✅ Yes | ✅ Yes | ✅ Yes | Secure enterprise apps |

**🔍 Let’s Break It Down One-by-One**

**🛑 1. Network Policy: None**

This means **no rules**, **no restrictions**, and **no enforcement**.

🔓 Example:

* Your frontend, backend, and database Pods can **freely communicate** with each other.
* Even if you write a NetworkPolicy YAML — it will be ignored!

🧪 Use case:

* For quick testing, lab clusters, or where security is not a concern.

📌 Azure default when you don't specify --network-policy.

**🛡️ 2. Network Policy: Azure**

Enforced using **Azure NSGs** (Network Security Groups) under the hood.

✅ Supports:

* **Ingress** (Who can send traffic to a Pod)

❌ Does NOT support:

* **Egress rules** (You cannot restrict outgoing connections)
* **IPBlocks** (CIDRs)

📦 Azure mode is good when:

* You want **basic intra-cluster security**
* You want to block **Pod-to-Pod** communication except for allowed ones

**🧱 3. Network Policy: Calico**

Uses **Calico plugin** to enforce rules.

✅ Supports:

* Ingress
* Egress
* IP CIDR (IPBlock)
* Namespace selectors

This is **full-featured**, open-source, and **CNCF compliant**.

**🔹 1. Flat Networking Model**

* Every **Pod gets its own IP address**.
* All Pods are in the **same flat network**—no NAT is needed.
* This makes it easy to treat a Pod like a mini server.

📦 **Key rule**:

All Pods can talk to all other Pods in the cluster **by default**, across nodes and namespaces.

**🔹 2. Pod-to-Pod Communication**

* Kubernetes uses **CNI plugins** like:
  + **Azure CNI** (AKS default): Pods get IP from Azure VNet.
  + **Kubenet** (legacy): Pods get internal IPs NATed via nodes.

📍Example:  
If Pod A has IP 10.240.0.12, and Pod B has 10.240.0.13, they can directly curl each other.

**🔹 4. Ingress and Egress**

* **Ingress** = traffic coming **into the cluster** from outside.
  + Handled via LoadBalancers, Ingress Controllers (like NGINX, AGIC)
* **Egress** = traffic going **out to internet or other networks**
  + Can be unrestricted, or controlled via Network Policy or firewall.

**🔐 Kubernetes Network Policies — for Isolation**

Now, let’s talk about **security and isolation**.

🔒 Without policies:  
Everything can talk to everything = 🧨 not secure!

**✅ Why Use Network Policies?**

* Prevent Pod A from accessing Pod B unless allowed
* Lock down database access to backend only
* Prevent apps from making outbound internet calls
* Ensure frontend can’t talk to another team’s namespace

**🧱 How Network Policies Work**

They work **like firewalls** at the Pod level.

* Based on:
  + Pod labels
  + Namespace selectors
  + IP blocks

If you **apply any network policy** to a Pod, **Kubernetes denies all traffic by default**—and only allows what’s explicitly defined.

You should use them in your clusters to prevent apps from reaching each other over the network, which will help limit the damage if one of your apps is compromised.

A diagram of a network

AI-generated content may be incorrect.

In the diagram above you can see we have a test-service which requires to accept TCP connections only from client-green pod and not from client-red. In such a situation a network policy can help you control this traffic flow as desired .

**❗ Can You Enforce Network Policies on a Cluster Created with --network-policy none?**

**❌ No, you cannot enforce network policies on a cluster with network-policy=none.**

Even if you apply a NetworkPolicy YAML, it will **not be enforced** because the cluster is **not configured with a network policy engine**.

**🔍 Why?**

When you create an AKS cluster with:

**--network-policy none**

* No network policy plugin (like **Azure** or **Calico**) is installed.
* Kubernetes will **accept** your NetworkPolicy YAML (no errors), but **will not enforce it**.
* All pods can **freely communicate**, regardless of any rules written.

✅ **The API server stores the policy, but no CNI plugin applies it.**

az aks create \

--resource-group sumi-testing \

--name aks-none-demo \

--location eastus \

--node-count 2 \

--node-vm-size Standard\_B1ms \

--enable-managed-identity \

--network-plugin azure \

--network-policy none \

--generate-ssh-keys

**🚀 PHASE 2: AKS with network-policy=azure**

**🧠 What is Azure Network Policy?**

* Built-in policy support in AKS when using Azure CNI.
* Supports only Ingress rules (Egress blocking is not supported).
* Integrated with Azure NSGs and enforces policies at the VM level.
* Useful for internal microservices isolation.

**🎯 Real-Time Scenario**

You are deploying a frontend and backend microservice.  
You want to:

1. ✅ Allow frontend to access backend
2. 🚫 Deny everything else
3. ✅ Allow backend to talk to external services (Egress is always allowed with Azure policy).

az aks create \

--resource-group sumi-testing \

--name netpol-azure-cluster \

--network-plugin azure \

--network-policy azure \

--node-vm-size Standard\_B2s \

--node-count 2 \

--enable-managed-identity \

--generate-ssh-keys

az aks get-credentials \

--resource-group netpol-azure-rg \

--name netpol-azure-cluster \

--overwrite-existing

Step 3: Deploy Frontend and Backend Pods

🧩 backend.yaml

apiVersion: v1

kind: Pod

metadata:

name: backend

labels:

app: backend

spec:

containers:

- name: backend

image: nginx

ports:

- containerPort: 80

🧩 frontend.yaml

apiVersion: v1

kind: Pod

metadata:

name: frontend

labels:

app: frontend

spec:

containers:

- name: curlbox

image: radial/busyboxplus:curl

command: ["sleep", "3600"]

kubectl apply -f backend.yaml

kubectl apply -f frontend.yaml

🔹 Step 4: Test Default Communication (Pre-Policy)

kubectl exec -it frontend -- curl <ip address backend>

✅ Output: HTML from Nginx (success)  
🧠 All pods can talk by default.

**🔹 Step 5: Create a Deny-All Ingress Policy**

**🧩 deny-all-ingress.yaml**

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: deny-all-ingress

spec:

podSelector: {}

policyTypes:

- Ingress

kubectl apply -f deny-all-ingress.yaml

kubectl exec -it frontend -- curl backend

❌ Output: Connection refused — because **ingress is denied for all pods.**

**🔹 Step 6: Allow Frontend → Backend Only**

**🧩 allow-frontend-to-backend.yaml**

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: allow-frontend-backend

spec:

podSelector:

matchLabels:

app: backend

ingress:

- from:

- podSelector:

matchLabels:

app: frontend

policyTypes:

- Ingress

**kubectl apply -f allow-frontend-to-backend.yaml**

**kubectl get pods -o wide**

**kubectl exec -it frontend -- curl <backend ip address>**

**✅ Output: Nginx page (Success)**

**🔹 Step 7: Confirm Other Pods Are Still Blocked**

**Create a third pod:**

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: attacker**

**labels:**

**app: attacker**

**spec:**

**containers:**

**- name: curlbox**

**image: radial/busyboxplus:curl**

**command: ["sleep", "3600"]**

kubectl apply -f attacker.yaml

kubectl exec -it attacker -- curl backend

❌ Connection refused — since attacker is **not allowed**.

**🔍 Real-Time Use Cases**

| **Use Case** | **Azure Network Policy** |
| --- | --- |
| ✅ Frontend ↔ Backend | Allowed (via label selector) |
| 🚫 Unknown Pod ↔ Backend | Denied |
| ✅ Backend to Internet | Allowed (Egress is not enforced) |
| 🚫 DNS to kube-system | Always allowed by AKS |

**🧱 Limitations**

* ❌ No Egress enforcement
* ❌ No CIDR/IPBlock support
* ❌ No namespace-level rules

**🔍 What We’ll Prove**

In AKS with --network-policy azure:

* **Ingress rules are enforced**
* **Egress is NOT enforced** — i.e., pods can access the internet by default

**✅ Goal**

From your backend pod (or any pod), **curl an external domain** (like google.com or httpbin.org) to verify **egress works**.

**✅ 1. Create a New Pod to Act as a Backend (If not already created)**

You can reuse an existing one (backend) or create a test pod:

# test-egress.yaml

apiVersion: v1

kind: Pod

metadata:

name: egress-test

labels:

app: egress-test

spec:

containers:

- name: curl

image: radial/busyboxplus:curl

command: ["sleep", "3600"]

**kubectl apply -f test-egress.yaml**

✅ 2. Exec Into the Pod and Test Egress

kubectl exec -it egress-test -- sh

curl https://google.com

curl <https://httpbin.org/ip>

**🧠 Why Does This Work?**

Because Azure Network Policy **only enforces ingress**:

| **Feature** | **Azure Network Policy** |
| --- | --- |
| Ingress | ✅ Enforced |
| Egress | ❌ Not Enforced |

So unless you use Calico or other CNI that supports egress rules, **pods can access internet by default**.

==

**PHASE 3: Calico Mode** — the most **advanced and complete network policy implementation** in Kubernetes.

We'll now use **Calico Network Policies** on Azure Kubernetes Service (AKS), covering **real-time enterprise use cases** with **Ingress + Egress controls**.

| **Feature** | **Azure Policy** | **Calico** |
| --- | --- | --- |
| Ingress filtering | ✅ Yes | ✅ Yes |
| Egress filtering | ❌ No | ✅ Yes |
| IPBlock/CIDR rules | ❌ No | ✅ Yes |
| DNS-based egress control | ❌ No | ⚠️ Limited (use with sidecar tools) |
| Namespaced isolation | ✅ Limited | ✅ Yes |
| Label-based enforcement | ✅ Yes | ✅ Yes |

**🎯 Real-Time Enterprise Scenario :**

Let's build **Calico Network Policies from scratch** with a complete, beginner-friendly, and step-by-step guide. This includes:

az aks create \

--resource-group calico-lab-rg \

--name calico-aks-demo \

--node-count 2 \

--node-vm-size Standard\_B2s \

--enable-managed-identity \

--network-plugin azure \

--network-policy calico \

--generate-ssh-keys

A NetworkPolicy in Kubernetes is like a **firewall rule for Pods** — it controls:

* **Who can talk to your Pod (Ingress)**
* **Who your Pod can talk to (Egress)**

**🧪 Phase 1: Setup — Namespaces & Pods**

**1️⃣ Create Namespaces**

kubectl create namespace team-api

kubectl create namespace team-ui

kubectl create namespace testers

🔖 Add Namespace Labels (needed for network policy namespaceSelector)

kubectl label ns team-api name=team-api

kubectl label ns team-ui name=team-ui

kubectl label ns testers name=testers

**Create Pods in Each Namespace**

**✅ checkout.yaml (in team-api)**

apiVersion: v1

kind: Pod

metadata:

name: checkout

namespace: team-api

labels:

role: checkout

spec:

containers:

- name: web

image: nginx

ports:

- containerPort: 80

✅ frontend.yaml (in team-ui)

apiVersion: v1

kind: Pod

metadata:

name: frontend

namespace: team-ui

labels:

role: frontend

spec:

containers:

- name: curl

image: radial/busyboxplus:curl

command: ["sleep", "3600"]

✅ tester.yaml (in testers)

apiVersion: v1

kind: Pod

metadata:

name: tester

namespace: testers

labels:

role: tester

spec:

containers:

- name: curl

image: radial/busyboxplus:curl

command: ["sleep", "3600"]

kubectl apply -f checkout.yaml

kubectl apply -f frontend.yaml

kubectl apply -f tester.yaml

**🔍 Phase 2: Baseline Test — Everything Should Work**

✅ frontend → checkout

kubectl exec -n team-ui frontend -- curl -s checkout.team-api.svc.cluster.local

✅ tester → checkout

kubectl exec -n testers tester -- curl -s checkout.team-api.svc.cluster.local

✅ checkout → Internet

kubectl exec -n team-api checkout -- curl -s <https://ifconfig.me>

All these should work **before we apply any policies**.

**🔒 Phase 3: Use Case 1 — Deny All in team-api**

**🔐 Policy: deny-all.yaml**

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: deny-all

namespace: team-api

spec:

podSelector: {}

policyTypes:

- Ingress

- Egress

**kubectl apply -f deny-all.yaml**

**🔬 Test After deny-all**

| **Source → Target** | **Test Command** | **Expected** |
| --- | --- | --- |
| frontend → checkout | curl | ❌ Blocked |
| tester → checkout | curl | ❌ Blocked |
| checkout → Internet | curl | ❌ Blocked |

🧩 Phase 4: Use Case 2 — Allow frontend to access checkout

🎯 Policy: allow-frontend-checkout.yaml

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: allow-frontend-checkout

namespace: team-api

spec:

podSelector:

matchLabels:

role: checkout

ingress:

- from:

- namespaceSelector:

matchLabels:

name: team-ui

- podSelector:

matchLabels:

role: frontend

policyTypes:

- Ingress

**kubectl apply -f allow-frontend-checkout.yaml**

**🔬 Test After Adding Ingress Policy**

| **Source → Target** | **Expected** |
| --- | --- |
| **frontend → checkout** | **✅ Allowed** |
| **tester → checkout** | **❌ Blocked** |
| **checkout → Internet** | **❌ Blocked** |

**🌍 Phase 5: Use Case 3 — Allow checkout → Internet (egress)**

**🌐 Policy: allow-egress-internet.yaml**

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: allow-egress-internet

namespace: team-api

spec:

podSelector:

matchLabels:

role: checkout

egress:

- to:

- ipBlock:

cidr: 0.0.0.0/0

policyTypes:

- Egress

**kubectl apply -f allow-egress-internet.yaml**

**🔬 Final Test Summary**

| **Source → Target** | **Expected** |
| --- | --- |
| **frontend → checkout** | **✅ Allowed** |
| **tester → checkout** | **❌ Blocked** |
| **checkout → Internet** | **✅ Allowed** |
| **inventory (if exists) → net** | **❌ Blocked** |

**kubectl delete ns team-api team-ui testers**

| **Use Case** | **Description** |
| --- | --- |

|  |  |
| --- | --- |
| **❌ Deny All** | **Zero-trust baseline** |

|  |  |
| --- | --- |
| **✅ Allow specific ingress** | **Allow only trusted apps to talk to pods** |

|  |  |
| --- | --- |
| **✅ Allow specific egress** | **Let selected pods talk to internet only** |

|  |  |
| --- | --- |
| **🔒 Combine rules** | **Multiple policies applied in additive logic** |

**👥 Use Case: Isolate Pod A from Pod B**

Let’s say you have two pods:

* pod-a with label app=a
* pod-b with label app=b

You want:

* Pod A can’t talk to Pod B
* Others can still talk freely

**❌ Block Pod A → Pod B**

You can write a **policy that allows traffic only from other pods**, **excluding Pod A**.

But Calico does not allow "deny from specific pod" natively via vanilla NetworkPolicy — instead, we **allow only specific pods**.

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: only-allow-from-certain-apps

namespace: my-apps

spec:

podSelector:

matchLabels:

app: b

ingress:

- from:

- podSelector:

matchLabels:

app: c # allow only pod C

policyTypes:

- Ingress

This indirectly isolates pod-b from pod-a, by **not allowing pod-a at all**.

**🎯 Pod Isolation Best Practices (Org-Level)**

| **Goal** | **Network Policy Approach** |
| --- | --- |
| ✅ Zero-trust | Apply a deny-all policy, then allow only trusted traffic |
| 🔄 Control egress | Use egress rules to prevent sensitive apps from reaching internet |
| 🔍 Visibility | Combine policies with monitoring tools like Calico Flow Logs |
| 👥 Multi-team isolation | Use namespaceSelector to isolate dev, QA, prod teams |

**🧪 Test for Pod Isolation**

**After applying a policy like deny-all, test like this:**

# Try curl from pod A to pod B

kubectl exec -n my-apps pod-a -- curl pod-b.my-apps.svc.cluster.local

# Should fail (Connection refused or timeout)

# Try DNS

kubectl exec -n my-apps pod-b -- nslookup google.com

# May fail if DNS not allowed