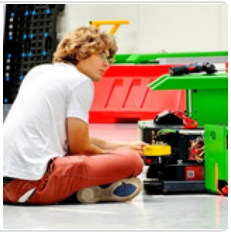
Kubernetes Networking: Pod Creation, Inter-Pod & Node Communication, CNI, Calico, and Kyverno Compliance

Check GitHub for helpful DevOps tools:

Michael Robotics
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics

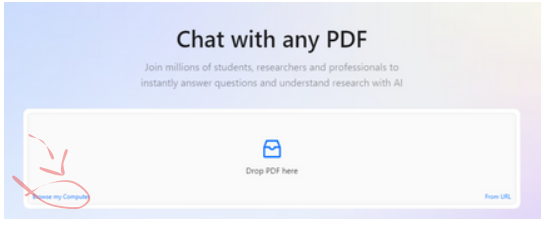


Ask Personal AI Document assistant to learn interactively (FASTER)!

1. Download PDF
2. Go to website
3. Browse file
4. Chat with Document

1 https://github.com/MichaelRobotics/DevOpsTools/blob/main/KubernetesNetwork.pdf

Click there to go to ChatPdf website 2



3

Ask questions about document! 4

Complety new to Linux and Networking?

Essential for this PDF is a thorough knowledge of networking. I highly recommend the HTB platform's networking module, which offers extensive information to help build a comprehensive understanding.

HTB - Your Cyber Performance Center

We provide a human-first platform creating and maintaining high performing cybersecurity individuals and organizations.

https://www.hackthebox.com/

What is Kubernetes?

Kubernetes is an open-source platform that automates the deployment, scaling, and management of containerized applications. It helps manage clusters of nodes running containers, ensuring efficient and reliable operation.

How Kubernetes clusters are made?

Kubernetes clusters consist of a control plane and multiple worker nodes. The control plane manages cluster operations, while worker nodes run the actual container workloads.

Why and When use Kubernetes

Kubernetes is ideal for deploying scalable, resilient, and automated containerized applications. It is used when managing multiple containers across different environments is necessary.

Example: Running a microservices-based e-commerce platform that scales up during peak hours.

System Requirements

- RAM: 2 GB per node (1 GB can work for testing but may lead to limited performance)
- 10 GB free storage
- Ubuntu

Kubernetes: Main components & packages

- **kube-apiserver:** Central management component that exposes the Kubernetes API; acts as the front-end for the cluster.
- **etcd:** Distributed key-value store for storing all cluster data, ensuring data consistency across nodes.
- **kube-scheduler:** Assigns pods to available nodes based on resource requirements and policies.
- **kube-controller-manager:** Manages core controllers that handle various functions like node status, replication, and endpoints.
- **kubelet:** Agent that runs on each node, responsible for managing pods and their containers.
- **kube-proxy:** Manages networking on each node, ensuring communication between pods and services within the cluster.

Kubernetes Network: Container deployment process

1) Container creation Flow

Deploying a Kubernetes (K8s) container involves a series of interactions between various components of the Kubernetes architecture.

API Server → User request to create a pod.

Scheduler → Selects a node for the pod.

Kubelet → Monitors pod status and initiates container creation.

CRI → Kubelet interacts with the container runtime.

containerd → Manages container image and lifecycle.

Shim → Provides management and monitoring of the container.

runc → Creates and runs the container.

CNI → Sets up networking for the container.

API Server Call

- Request Handling: The API server receives and processes the deployment request (via kubectl or CI/CD). Desired state is stored in etcd for later access.

Scheduler

- The scheduler assigns a node to the new pod based on resource requirements and policies. The pod is bound to the chosen node by updating its spec in etcd.

Kubelet

- The kubelet fetches the pod spec from the API server and monitors it. It initiates container setup by interacting with the CRI.

Container Runtime Interface (CRI)

- Lifecycle Management: The kubelet uses CRI to send commands to create, start, and manage the container.

containerd

- containerd pulls the image, creates the container, and handles its lifecycle.

Shim

- The shim enables monitoring and stream handling for the container, even if containerd restarts.

runc

- runc uses Linux namespaces and cgroups to isolate and run the container process.

Container Network Interface (CNI)

- The CNI plugin configures network interfaces and IPs, enabling container communication within the cluster and beyond.

Kubernetes Network: CNI and Calico

1) Essence of CNI

The Container Network Interface (CNI) is a standardized framework for networking containers. CNI addresses the need for:

Flexibility and Portability

CNI's modular design supports diverse networking setups, from basic local configurations to advanced multi-node networks, all through a uniform API. It creates consistent configuration standard between runtimes and plugins.

Efficient Management

With commands like ADD, DEL, and CHECK, CNI streamlines network lifecycle management, allowing smooth attachment/detachment of containers and resource cleanup as needed.

There are multiple CNI plugins available:

Flannel - Simple overlay network for direct pod communication; ideal for basic setups.

Calico - Advanced traffic control and security; suited for scalable, secure clusters.

Cilium - eBPF-based for deep network visibility; perfect for secure microservices.

To write your own, check this video:

Kubernetes Networking

CLOUD NATIVE COMPUTING FOUNDATION

How to Write a CNI Plugin From Scratch - Eran Yanay, Twistlock

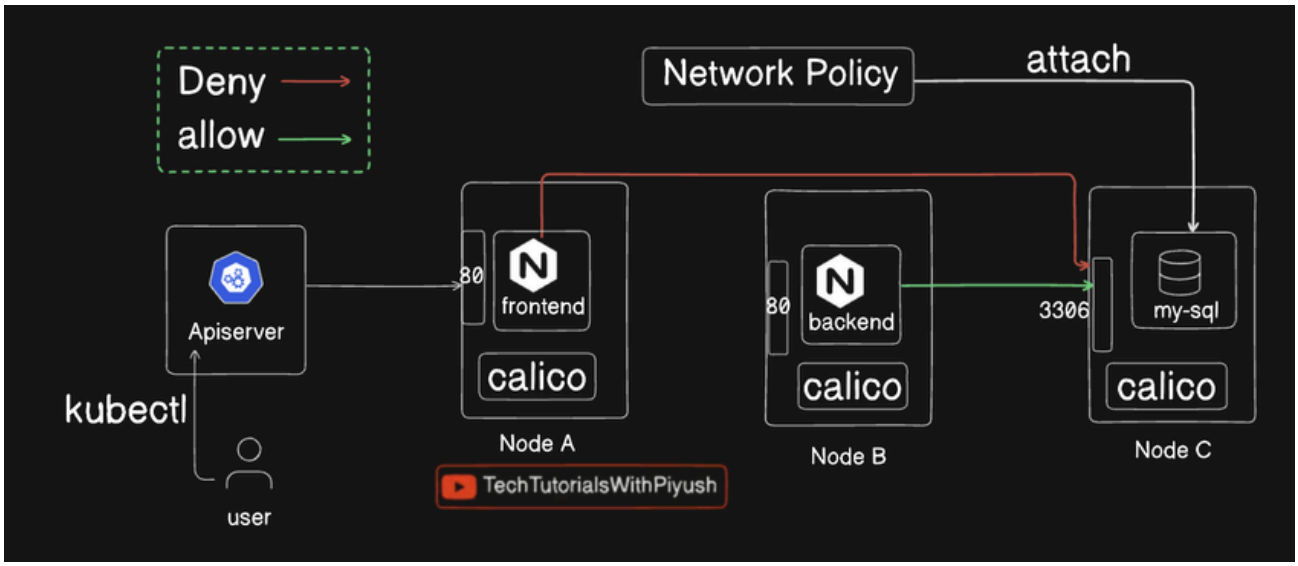
https://www.youtube.com/watch?v=zmYxdtFzK6s&t=497s

2) Calico Network Policies

By default, all Kubernetes pods can communicate freely. However, it's best practice to restrict access between services. For example, the backend should access the database, while the frontend should only interact with the backend. This can be achieved using Network Policies.

What is network Policy?

Network policies allow you to control inbound and outbound traffic to and from the cluster. For instance, you can create a deny-all policy to block all incoming traffic or an allow policy that permits access to specific services or pods on designated ports.

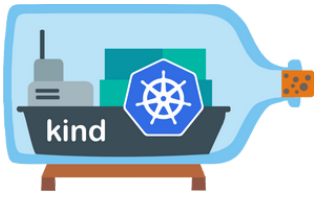


Install Kind. Follow steps from website:

Quick Start

This guide covers getting started with the kind command.

https://kind.sigs.k8s.io/docs/user/quick-start



Create kind cluster:

```
kind create cluster --name=test --config=kind.yaml
```

In the kind.yaml file, disable the default Kind CNI (as it doesn't support Network Policies) and use Calico instead.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 30001
    hostPort: 30001
- role: worker
- role: worker
networking:
  disableDefaultCNI: true
  podSubnet: 192.168.0.0/16
```

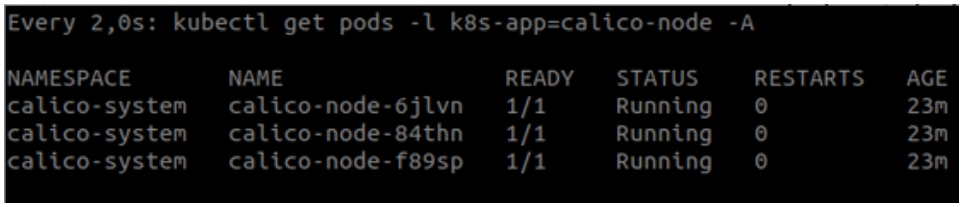
Install Calico by using the following command.

```
kubectl apply -f https://raw.githubusercontent.com/projectcalico/calico/v3.29.0/manifests/calico.yaml
```

You can verify Calico installation in your cluster by issuing the following command.

```
watch kubectl get pods -l k8s-app=calico-node -A
```

Wait untill Status will change to running.



This YAML configuration file defines a simple multi-tier application in Kubernetes. It consists of several Pods and Services that work together to set up a frontend, backend, and database. Let's go through each section in detail.

apply deployment from manifest located on my Github:

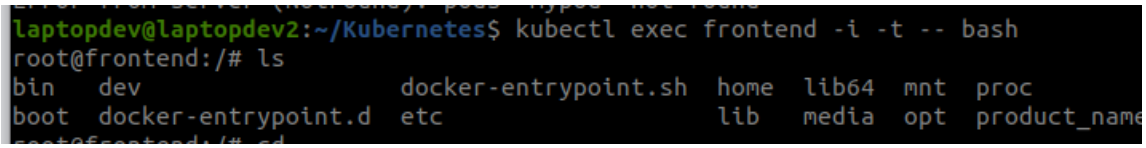
```
kubectl apply -f
https://raw.githubusercontent.com/MichaelRobotics/Kubernetes/main/Calico/manifest.yaml
```

Check full manifest file on my Github:

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
  labels:
    role: frontend
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - name: http
      containerPort: 80
      protocol: TCP
---
[...]
```

To verify that each pod can communicate with others by default, log into each pod and use curl to test connectivity. Start by logging into the frontend pod:

```
kubectl exec frontend -i -t -- bash
```



Curl backend service:

```
curl backend
```

Install telnet to check database availability:

```
apt-get update & apt-get install telnet
```

Then try to connect to database:

```
telnet db 3306
```

Everythin works as predicted - We can connect to each pod from fronted pod. Now implement Network Policies.

This NetworkPolicy permits only the pods labeled role: backend to connect to the mysql pods on port 3306. Any other pod or traffic attempting to connect to the mysql pods is denied by default

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-test
spec:
  podSelector:
    matchLabels:
      name: mysql
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: backend
    ports:
    - port: 3306
```

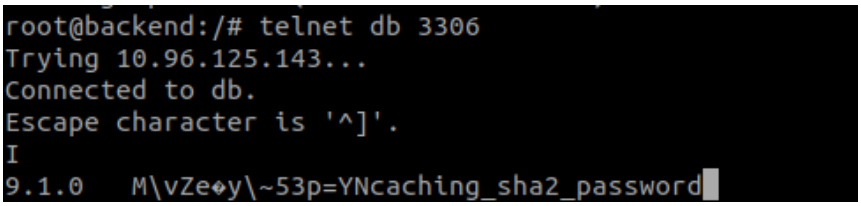
Save this manifest as netpolicy.yaml and apply:

```
kubectl apply -f netpolicy.yaml
```

Now log into frontend pod and try to connect to database:

As you can see, frontend pod cannot connect to database.

Now check what happens if I try to connect to db from backend pod. Exec into backend pod and install telnet then try to connect to backend:



Backend can acces db. Network policies are setup properly.

Great thanks to Piyush who make all materials and tutorial available!

Day 32/40 - Kubernetes Networking Explained

Container Network Interface (CNI)

https://www.youtube.com/watch?v=EkAzMGldC5M&list=PLI4APkPHzsUUOkOv3i62U

Kubernetes Network: Governance & Compliance Kyverno

1) Kyverno introduction

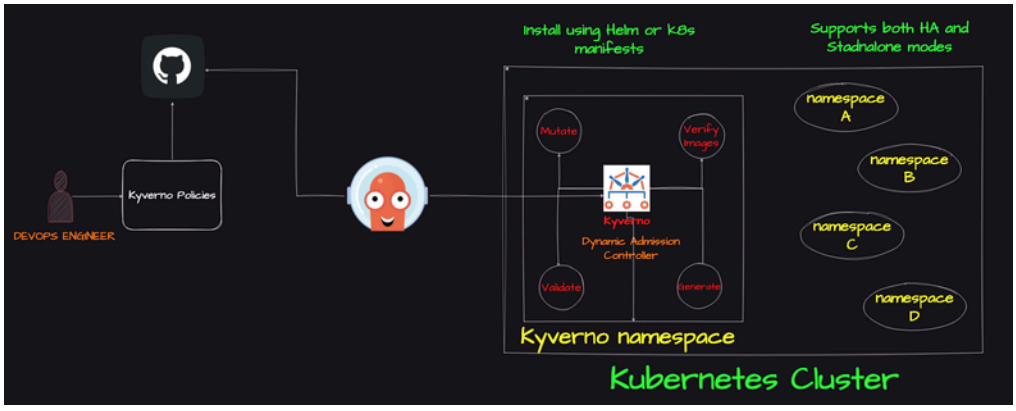
What is Kyverno?

Kyverno allows you to define policies as Kubernetes resources to secure, manage, and validate configurations in your Kubernetes cluster. Its policy engine supports four primary types of policies:

- **Generate policies**: Automatically create default configurations for new resources.
- **Validate policies**: Enforce standards by rejecting resources that don't meet security requirements.
- **Mutate policies**: Automatically adjust configurations to meet best practices.
- **Verify Images**: Ensure that container images meet specific integrity standards.

On a very high level, A DevOps Engineer will write the required Kyverno Policy custom resource and commits it to a Git repository. Argo CD which is pre configured with auto-sync to watch for resources in the git repo, deploys the Kyverno Policies on to the Kubernetes cluster.

Image representation, created by Greatest YT Mentor Abhisej Veermala:



2) Kyverno use case example (manual setup without GitOps) - Validate Policy example

Create kind cluster created in CNI step, or create new kind cluster

Then install Kyverno:

```
kubectl create -f https://github.com/kyverno/kyverno/releases/download/v1.8.5/install.yaml
```

Apply Kyverno Policy located on my GitHub, which will make restrainsts.

```
kubectl apply -f https://raw.githubusercontent.com/MichaelRobotics/Kubernetes/main/Kyverno/enforce-pod-requests-limits.yml
```

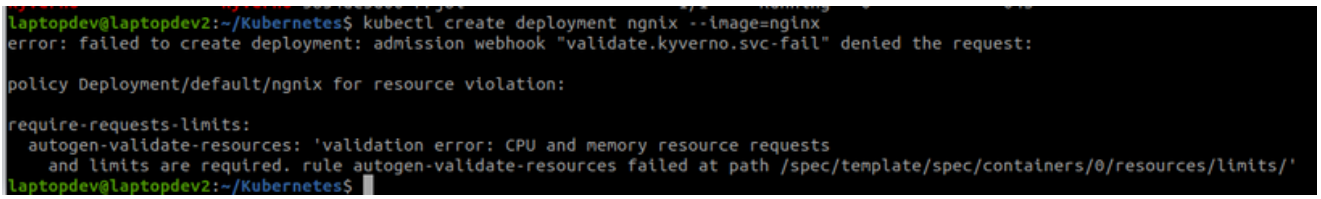
Check full manifest file on my Github:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
[...]
spec:
  validationFailureAction: enforce
  background: true
  rules:
  - name: validate-resources
    match:
      any:
      - resources:
          kinds:
          - Pod
    validate:
      message: "requests and limits are required."
      pattern:
        spec:
          containers:
          - resources:
              requests:
                memory: "?*"
                cpu: "?*"
              limits:
                memory: "?*"
```

Create new deployment using CLI to test if pod will be created:

```
kubectl create deployment ngnix --image=nginx
```

Error should appear:



As logs indicate, pod creation was denied due to NetworkPolicy enforcement made by Kyverno.

3) Kyverno Network Policies - Generate Policy example

At first, delete enforce-pod-requests-limits policy. Then Apply Kyverno Policy located on my GitHub, which will block ingress and egress traffic in all newly created pods in new created namespaces:

```
kubectl apply -f https://raw.githubusercontent.com/MichaelRobotics/Kubernetes/main/Kyverno/add-network-policy.yml
```

Check full manifest file on my Github:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
 name: add-networkpolicy
[...]
      spec:
        # select all pods in the namespace
        podSelector: {}
        # deny all traffic
        policyTypes:
        - Ingress
        - Egress
```

Create new namespace:

```
kubectl create namespace my-app-namespace
```

Change context into new namespace:

```
kubectl config set-context --current --namespace=my-app-namespace
```

Create new deployment

```
kubectl create deployment nginx --image=nginx
```

Check pod name:

```
kubectl get pods
```

```
laptopdev@laptopdev2:~/Kubernetes$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
my-app-deployment-59984ccf6-lnxtr   1/1     Running   0          7m8s
```

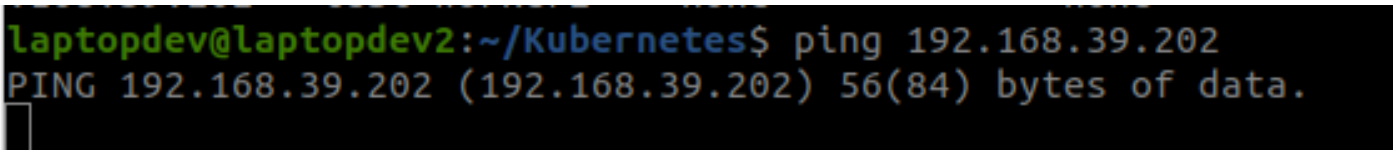
Check pod IP

```
kubectl get pod my-app-deployment-59984ccf6-lnxtr -o wide
```

```
laptopdev@laptopdev2:~/Kubernetes$ kubectl get pod my-app-deployment-59984c
cf6-lnxtr -o wide
NAME                                READY   STATUS    RESTARTS   AGE    IP
              NODE           NOMINATED NODE   READINESS GATES
my-app-deployment-59984ccf6-lnxtr   1/1     Running   0          8m6s   192
.168.39.202   test-worker2   <none>           <none>
```

ping this pod from host:

```
ping 192.168.39.202
```



As predicted , ping is send, dns reached but nothing happend.

Great thanks to Abhishek Veeramalla who make all materials and tutorial available!

Enforce Kubernetes Security with Kyverno

RealTime #kubernetes project

https://www.youtube.com/watch?v=5ihkMblumD0

Kubernetes Network: Container, Pod, Node structure and networking

1) Resources isolation in kubernetes

In Kubernetes, namespaces serve to organize and isolate resources within the cluster, providing both scoping and resource management for different types of resources such as Pods, containers, and nodes. These namespaces help maintain clean separations of concerns,

1) What is container made of

PID Namespace: Containers have their own PID numbering, isolating their processes from the host and other containers. Containers in the same Pod share the same PID namespace.

Network Namespace: Containers in a Pod share the same IP, ports, and interfaces Containers in same Pod can communicate through localhost. Containers in different Pods or nodes have separate network namespaces.

Mount Namespace: Containers have isolated filesystems, but containers in the same Pod can share volumes, with control over mounting.

UTS Namespace: Containers can have unique hostnames and domain names, appearing as separate machines.

IPC Namespace: Containers can use isolated communication resources like shared memory. Containers in the same Pod share the IPC namespace.

User Namespace: Containers can run as non-root users within their own namespace, even if running as root on the host.

We can check all namespaces that make specific container, at first check its master process:

```
lsns | grep <container name>
```

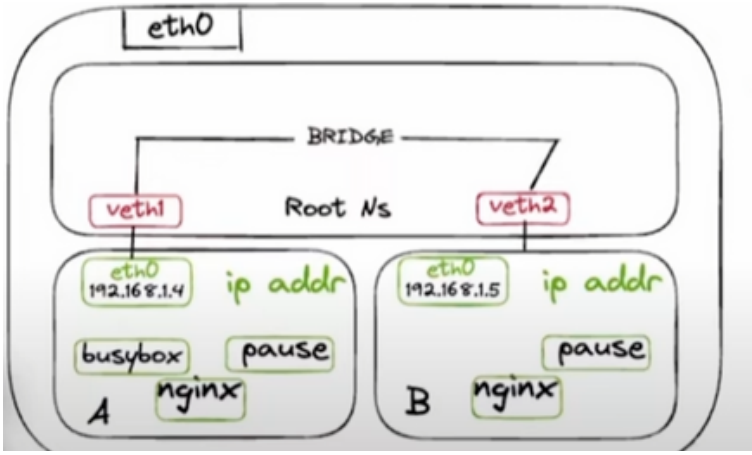
Then check its namespaces:

```
lsns -p <master_process>
```

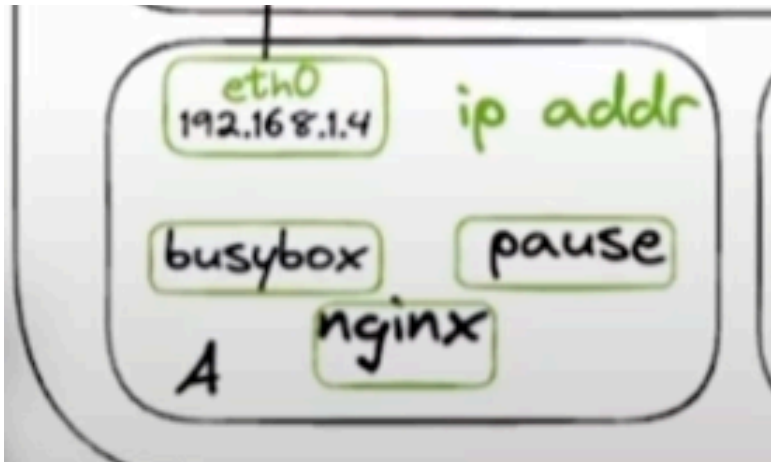
```
controlplane $ lsns | grep ngnix
controlplane $ ssh node01
Last login: Fri Nov  8 15:49:40 2024 from 10.244.6.87
node01 $ lsns | grep ngnix
node01 $ lsns | grep nginx
4026532557 mnt         2  5233 root            nginx: master process nginx -g daemon off;
4026532558 pid         2  5233 root            nginx: master process nginx -g daemon off;
node01 $ lsns -p 5233
        NS TYPE   NPROCS   PID USER  COMMAND
4026531835 cgroup    139     1 root  /sbin/init
4026531837 user      139     1 root  /sbin/init
4026532494 net         3  5175 65535 /pause
4026532554 uts         3  5175 65535 /pause
4026532555 ipc         3  5175 65535 /pause
4026532557 mnt         2  5233 root  nginx: master process nginx -g daemon off;
4026532558 pid         2  5233 root  nginx: master process nginx -g daemon off;
```

2) What makes pod and node

A Pod is the smallest unit in Kubernetes, containing one or more containers that share storage, network, and runtime settings. Containers within a Pod share a network namespace for communication via localhost. Pods live on a Node and connect to the Node's bridge network through a veth interface for communication with other Pods and external systems.



The pause container acts as the "parent container" for all other containers within a pod. It is the first container to start when the pod is created and is responsible for setting up the network namespace. This namespace is then shared by all other containers in the pod and is maintained for the entire lifetime of the pod.



To check that pod network is hold in namespace we can list network namespaces on node and show insides of this related to specific pod. SSH into node, then:

```
sudo ip netns list
```

```
node01 $ sudo ip netns list
cni-be8449ec-2fea-15a5-06ef-d92b34c08906 (id: 2)
cni-7a1f3e2c-3c20-1b2c-a16f-30d0be2caf66 (id: 1)
cni-90e6ae1d-b19a-a7d4-9398-9ed9354bddf2 (id: 0)
```

exec into namespace:

```
sudo ip netns exec cni-be8449ec-2fea-15a5-06ef-d92b34c08906 ip link
```

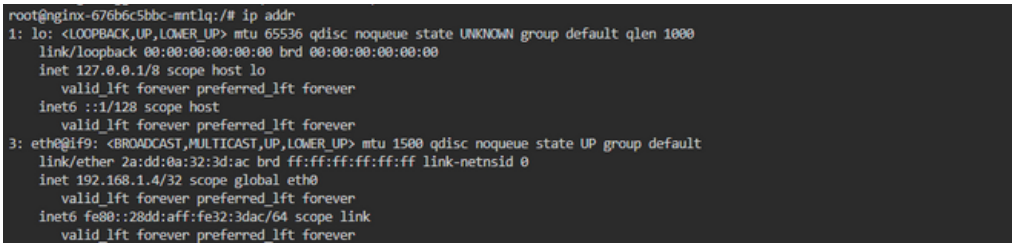
```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
3: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default
    link/ether 6e:64:ee:15:1f:68 brd ff:ff:ff:ff:ff:ff link-netnsid 0
node01 $
```

Then log into pod

```
kubectl exec -it nginx-676b6c5bbc-mntlq -- bash
```

and check ip links using **ip addr** command.

```
ip addr
```



As we can see, we can check Pod network configuration by ssh into it or directly checking associated namespace configuration.

Day 32/40 - Kubernetes Networking Explained

Container Network Interface (CNI)

https://www.youtube.com/watch?v=EkAzMGldC5M&list=PLI4APkPHzsUUOkOv3i62U

common troubleshooting

1) Kyverno Policy Blocking Resource Creation

Cause: Kyverno policy rejecting resource due to validation rules.
Solution: Review the policy with kubectl get clusterpolicy <policy-name> -o yaml and adjust as needed.

2) CNI Plugin Issues (e.g., Calico, Flannel)

Cause: Misconfigured or incompatible CNI plugin.
Solution: Verify CNI plugin status with kubectl get pods -n kube-system and consult documentation.

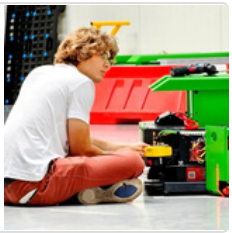
3) Pod Cannot Resolve DNS

Cause: DNS issues in the cluster or misconfigured resolv.conf.
Solution: Verify CoreDNS status with kubectl get pods -n kube-system. Test DNS resolution using kubectl exec with nslookup or dig.

6) Check my Kubernetes Troubleshooting series:

Michael Robotics
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics



Learn more about Kubernetes

Check Kubernetes and piyushsachdeva - great docs!

Setup a Multi Node Kubernetes Cluster

kubeadm is a tool to bootstrap the Kubernetes cluster

https://github.com/piyushsachdeva/CKA-2024/tree/main/Resources/Day27

Kubernetes Documentation

This section lists the different ways to set up and run Kubernetes

https://kubernetes.io/docs/setup/

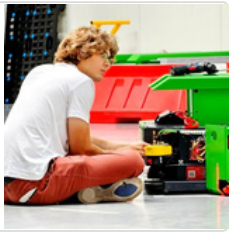
Share, comment, DM and check GitHub for scripts & playbooks created to automate process.

Check my GitHub

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics



PS.

If you need a playbook or bash script to manage KVM on a specific Linux distribution, feel free to ask me in the comments or send a direct message!