Kubernetes Networking

An In-Depth Look

Lisa Zangrando

Kubernetes Networking

Overview

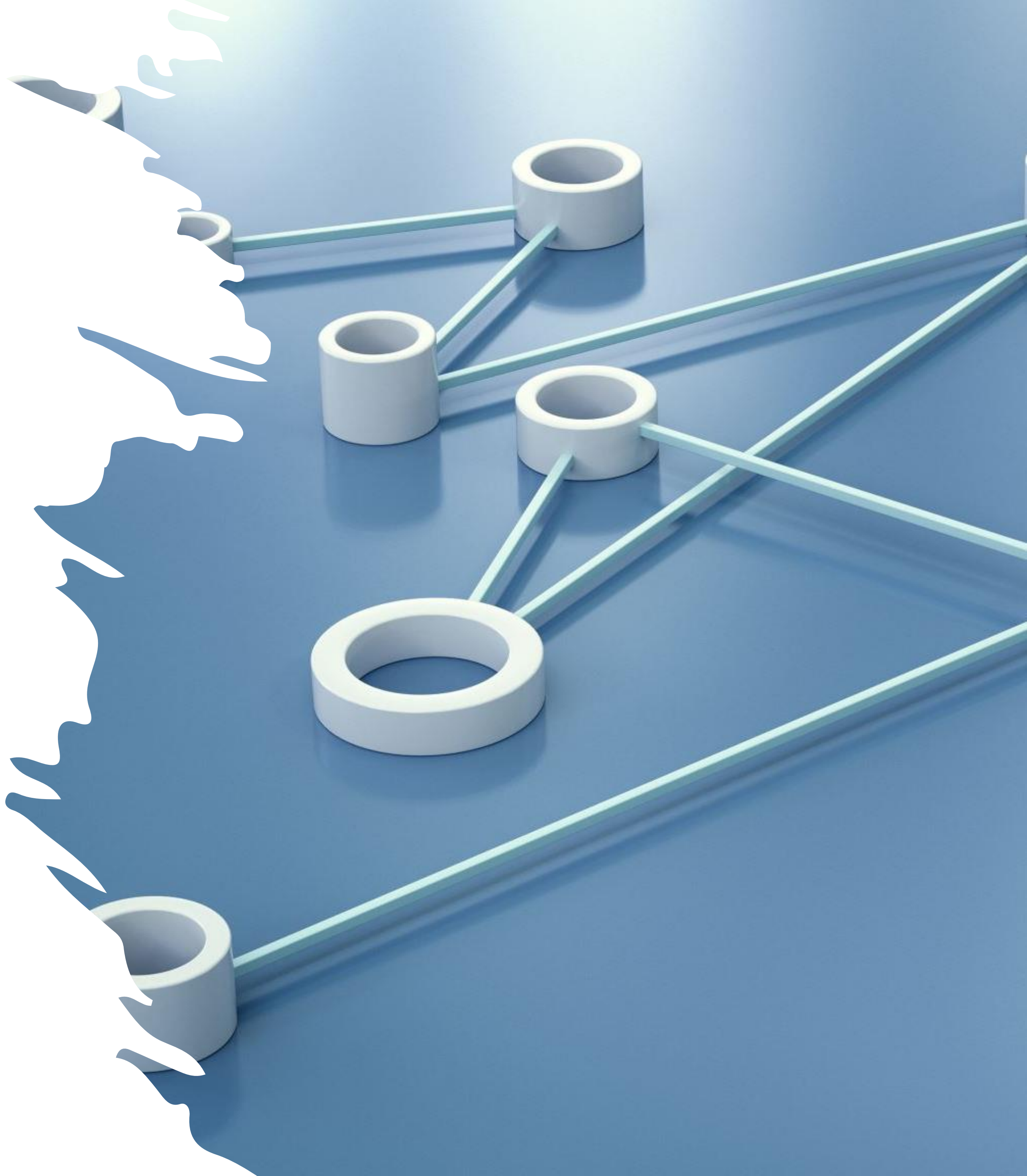
The Kubernetes networking model allows the different parts of a Kubernetes cluster, such as Nodes, Pods, Services, and outside traffic, to communicate with each other.

Why understanding it matters

- Properly configure your environment.
- Enable **complex networking scenarios**.

Key concepts covered

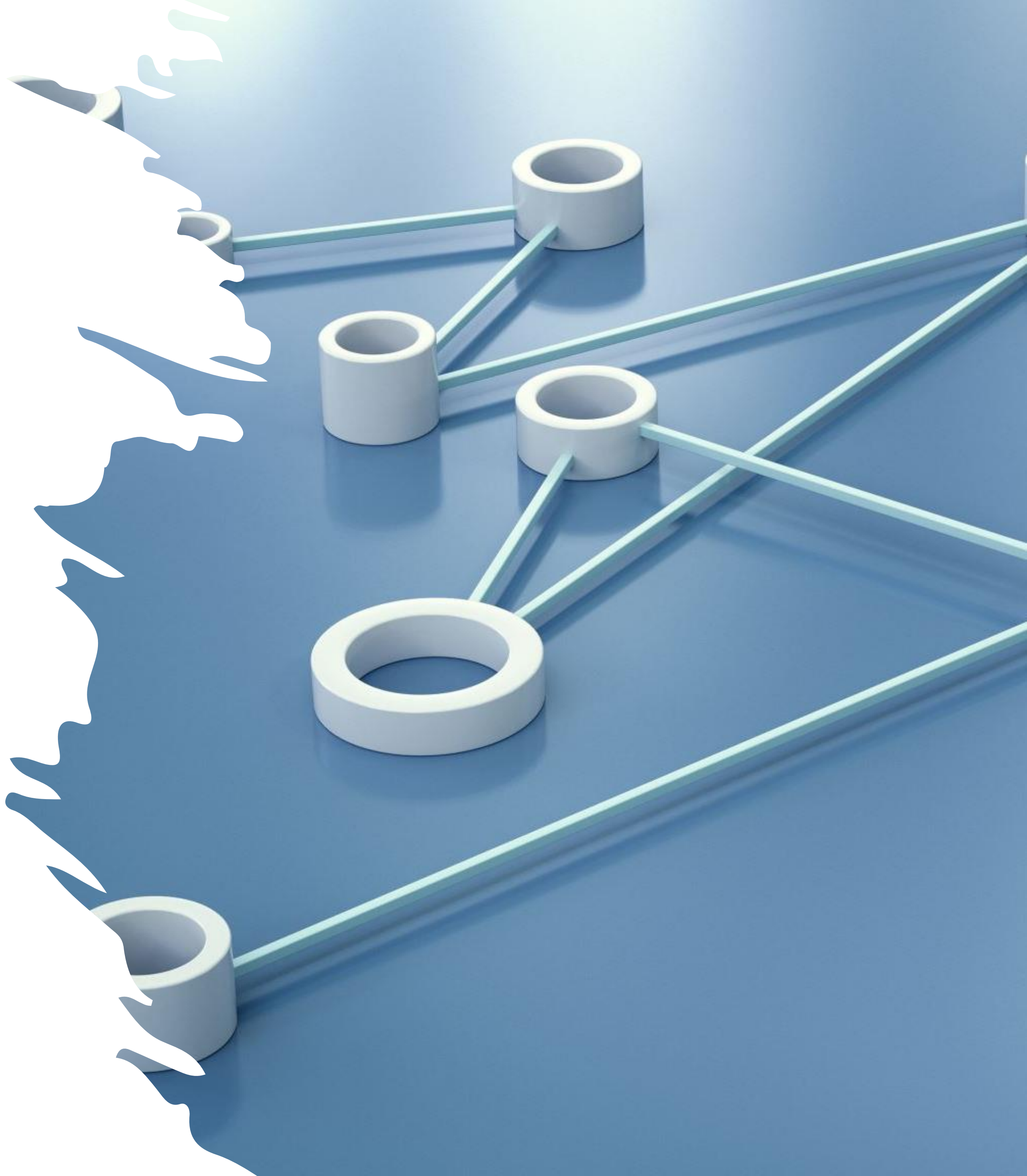
- **Networking Model**
- **Cluster communication types**:
  - Container-to-Container
  - Pod-to-Pod
  - Pod-to-Service
  - Internet-to-Service



Kubernetes Networking model

- The Kubernetes networking model is designed around the following key principles:
    - Every pod gets its own IP address
    - Containers within a pod share the pod IP address and can communicate freely with each other
    - Pods can communicate with all other pods in the cluster using pod IP addresses (without NAT)
    - Isolation (restricting what each pod can communicate with) is defined using network policies
    - Plugin-based flexibility and customization.

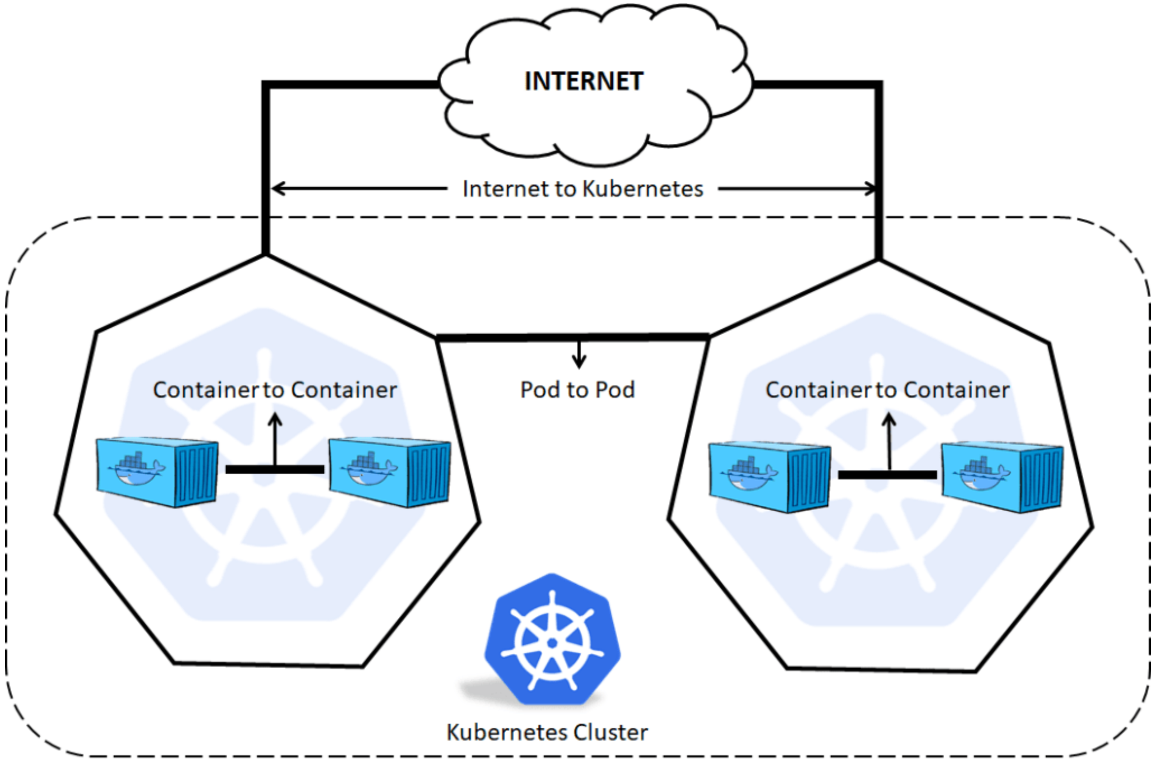
- This style of network is referred to as a "**flat network**"
    - From a pod's view, the cluster is a single network plane



Kubernetes Networking model

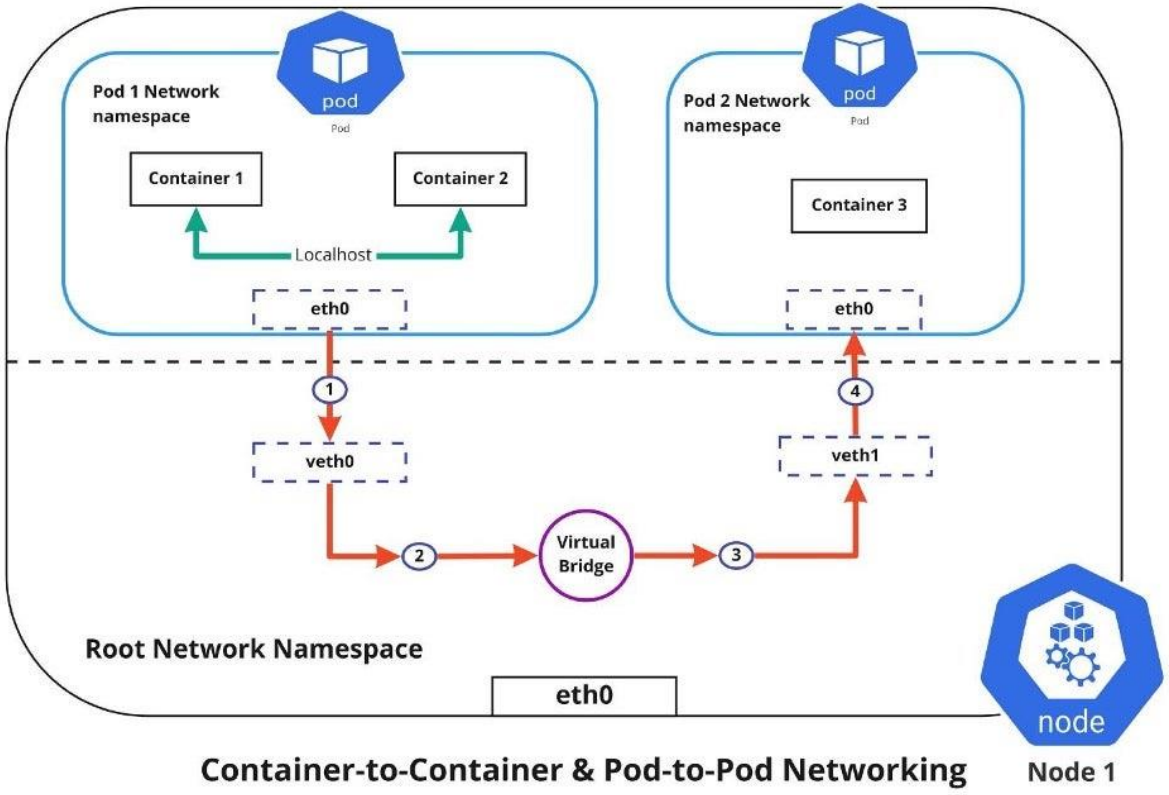
- Given these constraints, Kubernetes networking can be broken into four distinct problems to solve:
  - **Container-to-Container Networking**: how containers within the same Pod communicate.
  - **Pod-to-Pod Networking**: how Pods communicate with each other across nodes.
  - **Pod-to-Service Networking**: how Pods interact with Services, including load balancing and discovery.
  - **Internet-to-Service Networking**: how external traffic reaches cluster Services.

- And to solve them, Kubernetes employs several key networking components and resources:
  - Network namespaces, iptables, CNI plugins, Services…



Container-to-Container networking

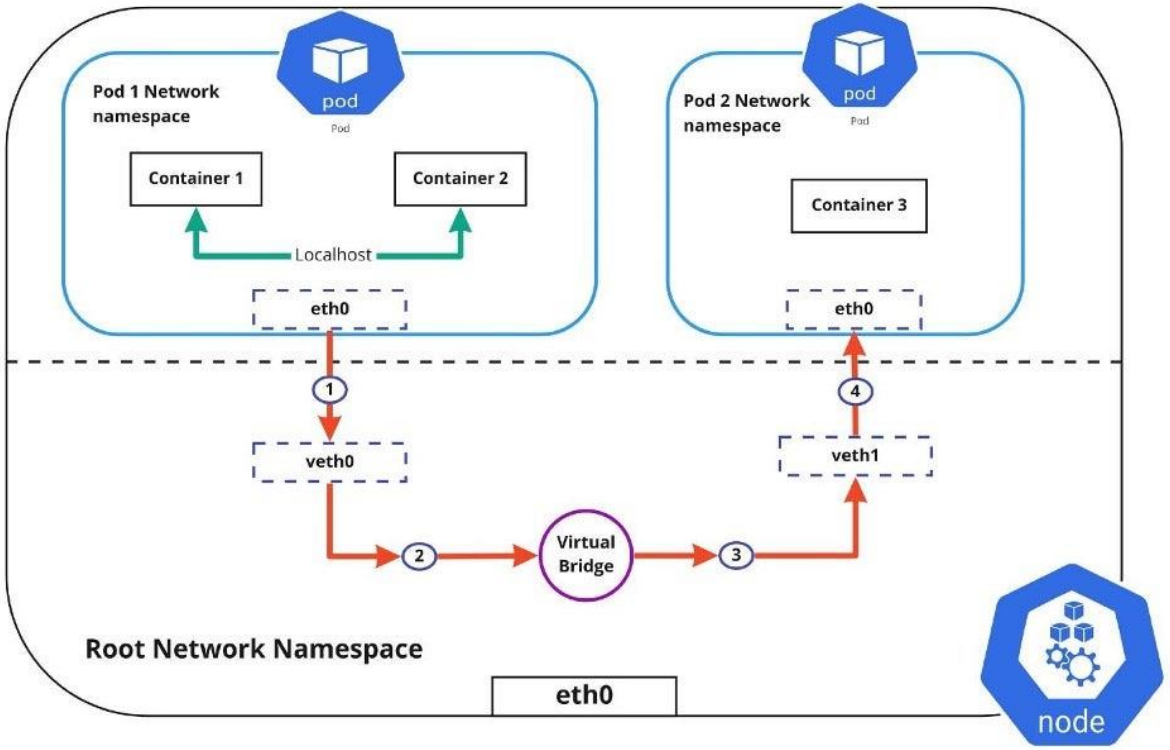
- Pod is modelled as a group of containers
- How containers within the same Pod communicate?
- Occurs through the **Pod (Linux) Network Namespace**
  - logical networking stack with its own logical router, firewall, and other network devices.
  - It allows for separate network interfaces and routing tables isolated from the rest of the system.
  - Container within the Pods will communicate with each other via **localhost** within the same Pod Network namespace



Container-to-Container & Pod-to-Pod Networking

Pod-to-Pod networking
(same node)

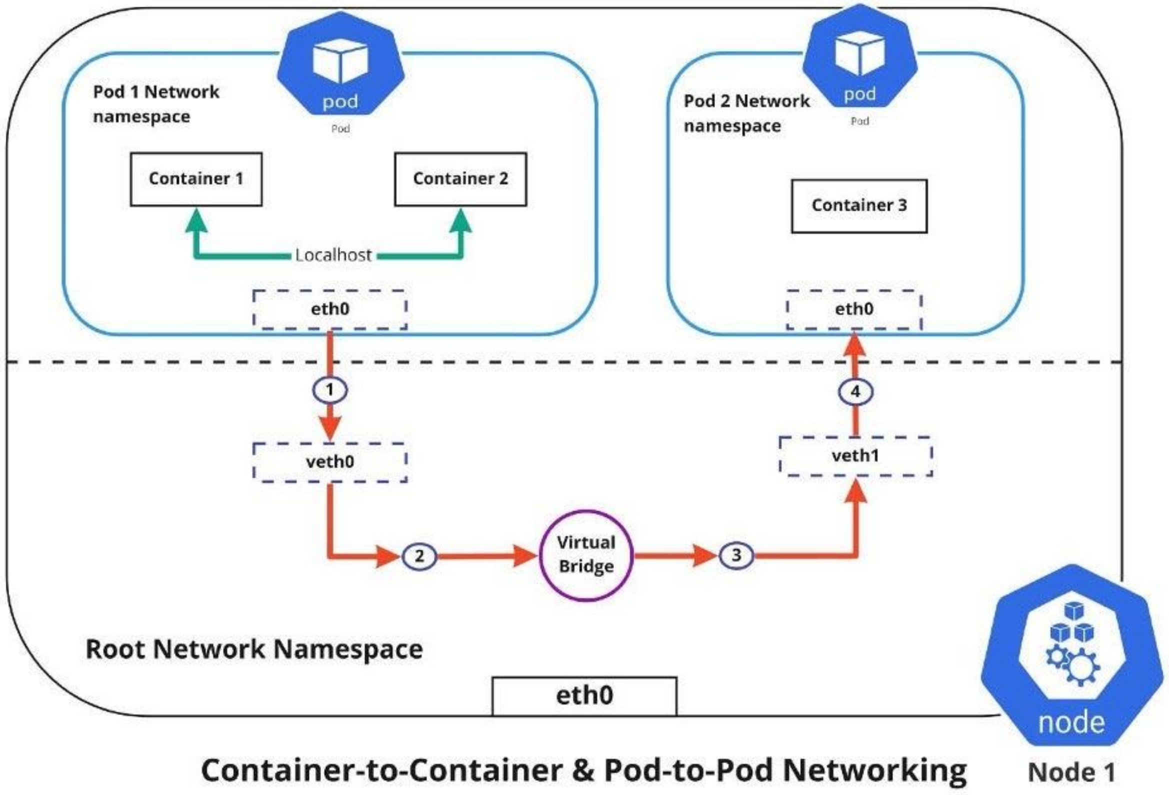
- Pods network namespaces are connected via **virtual ethernet devices** (veth pairs) to root network namespace within the node
- A **virtual network bridge** allows traffic between these interfaces, with communication using **ARP** (Address Resolution Protocol)
- Operates at **Layer 2 (Data Link)** using **MAC addresses** for packet forwarding.
- When a packet arrives:
    1. The bridge checks the destination **MAC address**.
    2. If the destination is local (on the same node), it forwards the packet to the appropriate veth interface.
    3. If the destination is not local, it sends the packet to the **default route** (gateway).



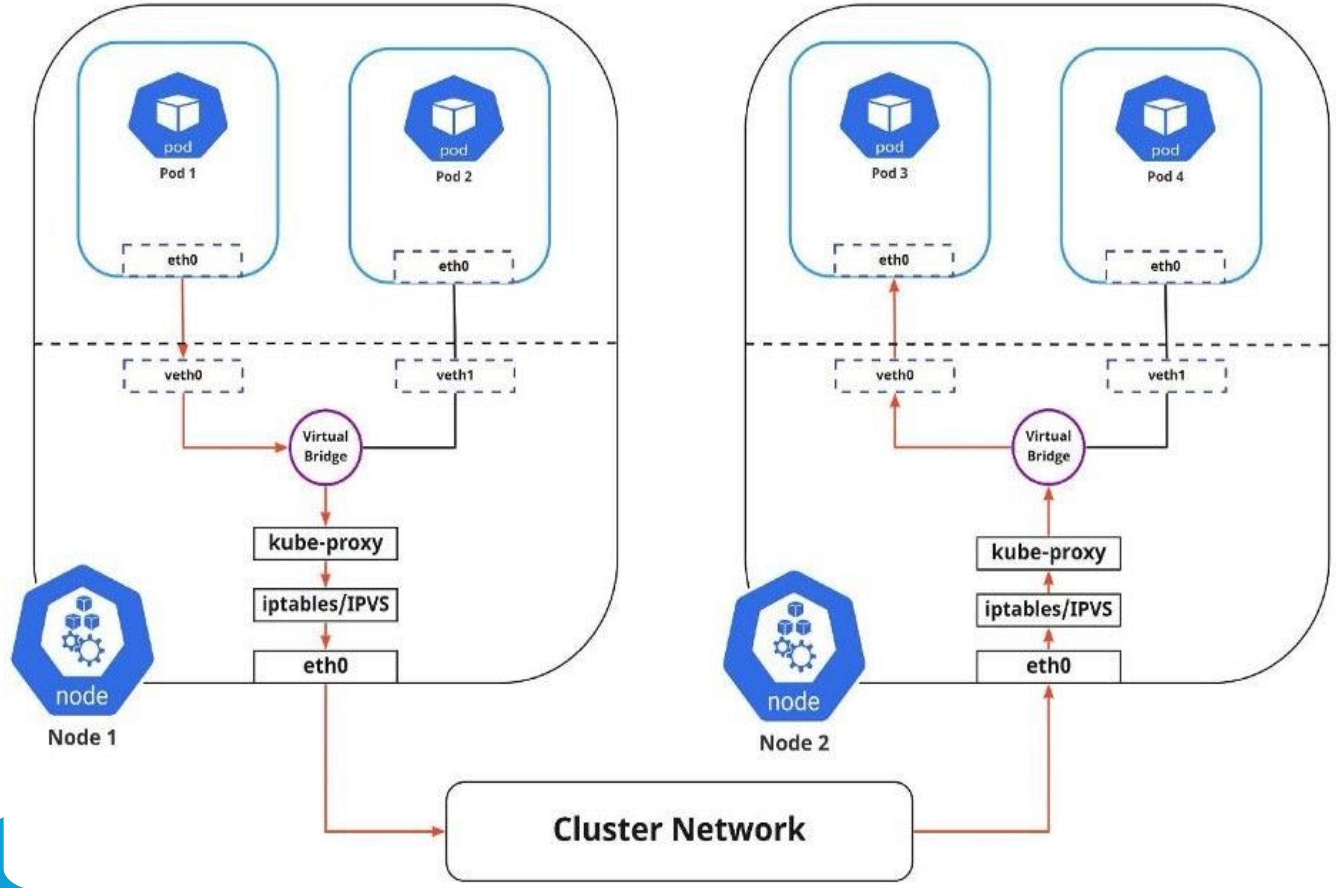
Container-to-Container & Pod-to-Pod Networking
Node 1

Pod-to-Pod networking
(same node)

- If data is sent from Pod 1 to Pod 2, the flow of events would like this (refer to diagram)
    1. Pod 1 traffic flows through eth0 to the root network namespaces virtual interface veth0.
    2. Then traffic goes via veth0 to the virtual bridge which is connected to veth1.
    3. Traffic goes via the virtual bridge to veth1.
    4. Finally, traffic reaches eth0 interface of Pod 2 via veth1.

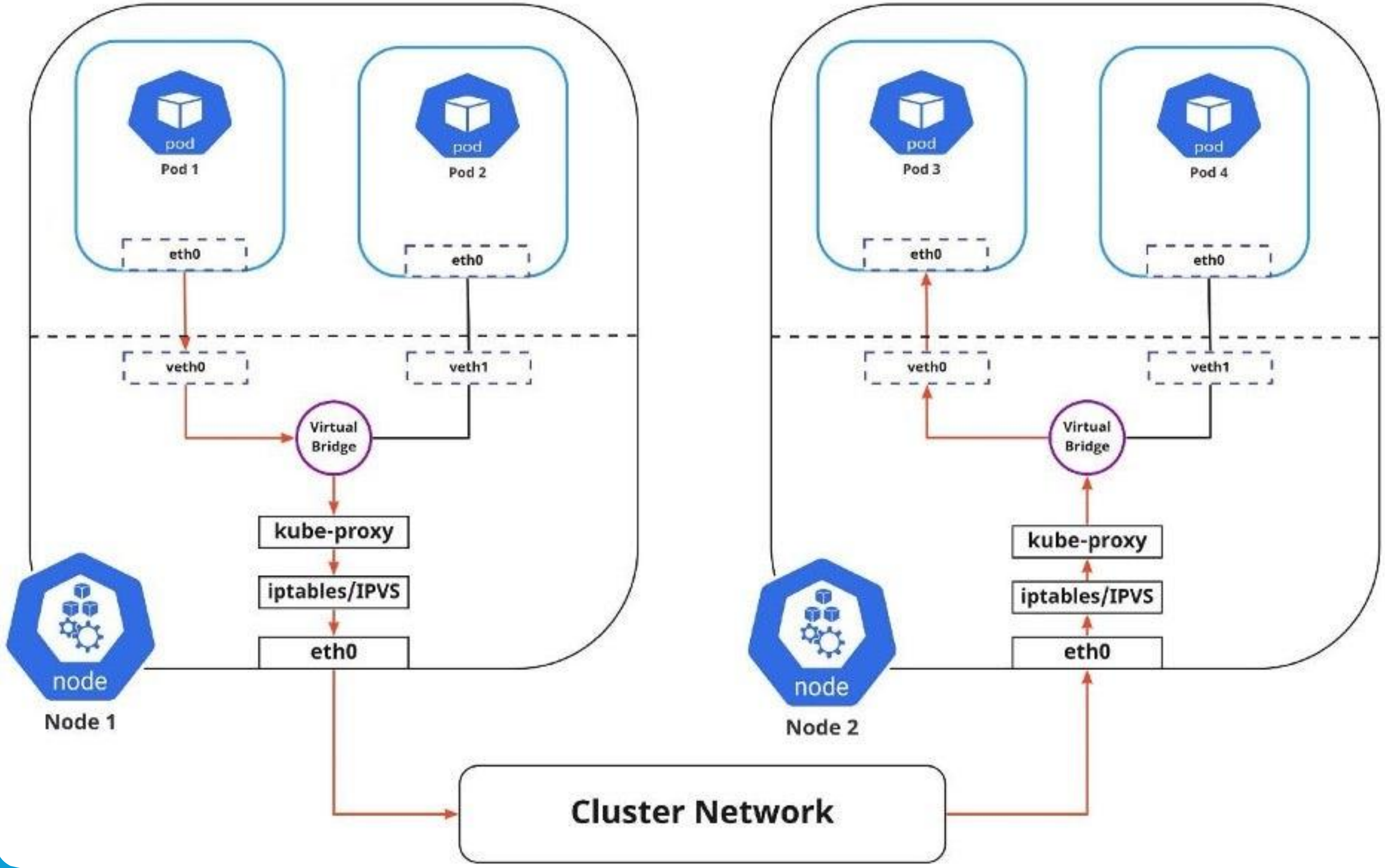


Pod-to-Pod networking
(different nodes)



- How do Pods communicate across Nodes?
- Each Pod has a unique IP within the cluster assigned by the CNI plugin.
- When a Pod sends traffic to another Pod on a different node:
  - the traffic exits the Pod through its veth interface.
  - the virtual bridge forwards the traffic to the default route if the destination is not local.
  - the default route sends the packet to node B, using one of two methods implemented by the CNI plugin: overlay and underlay network
- On the destination node:
  - The packet enters the node's root network namespace.
  - It is forwarded to the destination Pod via the virtual bridge and veth interface.

Pod-to-Pod networking
(different nodes)



Container Network Interface
(CNI plugin)

- The **Container Network Interface (CNI)** is a specification maintained by the **Cloud Native Computing Foundation (CNCF)** that standardizes the configuration of network interfaces for Linux containers.
- In Kubernetes, a **CNI plugin** is a software component implementing the CNI specification, enabling seamless communication between **Pods, Nodes, and external network components**.
- Key features:
  - configures network interfaces for Linux containers.
  - allocates networking resources, such as IP addresses.
  - enforces network policies for traffic control.
  - manages routing between Pods and external networks using two approaches: **overlay networks** and **underlay networks**.

Overlay vs Underlay Network
(CNI plugin)

- **Overlay Network**
  - Uses a **virtual network layer** on top of the existing physical network.
  - Encapsulates Pod traffic (e.g., VXLAN, IP-in-IP) so that Pods can communicate across nodes without modifying the underlying infrastructure.
  - More flexible but can introduce additional overhead.

- **Underlay Network**
  - Directly integrates Pods with the **physical network infrastructure**.
  - Assigns **routable IP addresses** to Pods, making them first-class citizens in the network.
  - Provides lower latency and better performance but requires more advanced networking configurations.

Common CNI plugins

- **Calico:** focuses on security and network policies using BGP for routing.
- **Flannel:** simplifies networking by creating an overlay network using VXLAN.
- **Weave Net:** provides a simple and fast overlay network for Kubernetes.
- **Cilium:** advanced networking with eBPF-based security policies and observability.
- **Canal:** combines Flannel for networking and Calico for network policies.
- **Kube-Router:** integrated networking, firewall, and routing for Kubernetes clusters.
- **Multus:** allows Pods to attach to multiple network interfaces.
- **Amazon VPC CNI:** optimized for AWS, enabling Pods to use VPC-native networking.
- **Azure CNI:** integrates with Azure virtual networks for Kubernetes workloads.
- **Google Cloud CNI:** provides seamless networking for Pods in GKE.
- **Antrea:** implements Open vSwitch for Kubernetes networking.

Pod-to-Service networking

- **Pods are Dynamic!**
    - **Scale up or down** in response to changes in demand.
    - **Recreated** automatically after a crash or node failure.
    - **IP addresses change** with these events, which can complicate networking.
- Kubernetes solution: the **Service** abstraction:
    - Provides **stable network access** to a set of Pods, shielding clients from the dynamic changes of Pods.
    - Assigns a **long-term virtual IP** to the frontend, ensuring reliable communication with backend Pods.
    - **Load-balances traffic** directed to the virtual IP, distributing it evenly among the backend Pods.
    - Clients connect with the static virtual IP of the Service.

Defining a Service

- This example creates a Nginx **Pod** and exposes it via a **Service**. The Service forwards traffic to any Pod with the label app: nginx, ensuring dynamic routing as Pods are added or removed.
- Explanation of Service attributes
  - **selector**: Matches the label of the target Pods, ensuring that traffic is dynamically routed to the correct set of Pods.
  - **type**: Determines how the Service is exposed:
    - **ClusterIP**: The Service is accessible only within the cluster.
    - **NodePort**: The Service is accessible externally on each node's IP and a specific port.
    - **LoadBalancer**: Integrates with cloud providers to create an external load balancer.
    - **ExternalName**: Maps the Service to an external DNS name.
  - **port**: The port on which the Service is accessible within the cluster.
  - **targetPort**: The port on the Pod where traffic should be forwarded, ensuring requests reach the correct application process.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx  # Label used by the Service selector
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80  # The port on which the container listens

---

apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx  # Matches the label of the target Pod(s)
  type: ClusterIP  # ClusterIP, NodePort, LoadBalancer,
ExternalName
  ports:
  - protocol: TCP  # Communication protocol (TCP/UDP)
    port: 80       # Port exposed by the Service
    targetPort: 80 # Port on the Pod to which traffic is forwarded
```

ClusterIP Service type

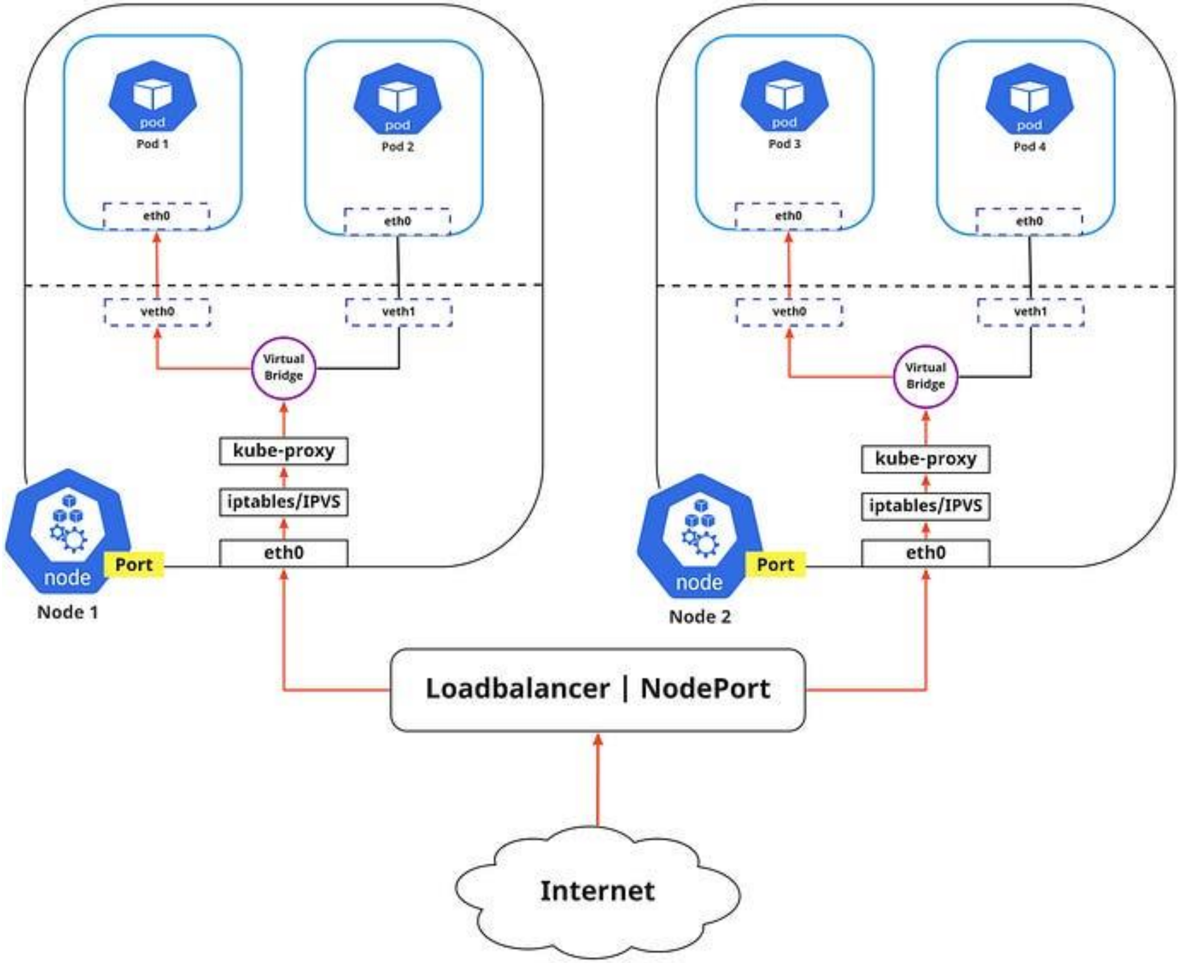
- This example illustrates how to set up a Service to route traffic to two NGINX Pods, using a **ClusterIP** Service for exposure.
- The Pods are accessible only within the cluster

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-clusterip
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

```
$ kubectl get pods -o wide
NAME        READY   STATUS    RESTARTS    AGE   IP           NODE
nginx1      1/1     Running   0           65m   10.244.1.3   k8s-node
nginx2      1/1     Running   0           65m   10.244.1.4   k8s-node

$ kubectl get svc -o wide
NAME             TYPE      CLUSTER-IP     EXTERNAL-IP PORT(S) AGE SELECTOR
nginx-clusterip ClusterIP 10.103.197.222 <none>      80/TCP  46m app=nginx

$ kubectl describe svc nginx-clusterip
...
IPs:                      10.103.197.222
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
Endpoints:                10.244.1.3:80,10.244.1.4:80

# Access the Service from within the cluster (e.g., using another Pod):

kubectl exec -it dnsutils – sh

curl http://10.103.197.222
<html>
<body>
  <h1>Nginx 1</h1>
</body>
</html>

curl http://10.103.197.222
<html>
<body>
  <h1>Nginx 2</h1>
</body>
</html>
```

NodePort Service type

- When a Service is defined with type: NodePort, Kubernetes exposes it on a static port (e.g., 30080) on all cluster nodes. This allows external users to access the application using http://<node-ip>:30080.
- The traffic flow is as follows:
  - A user sends a request to http://<node-ip>:30080.
  - The request reaches any Kubernetes node in the cluster.
  - Kubernetes forwards the request internally to the appropriate Pod running the Nginx container on targetPort: 80.
- User -> http://<node-ip>:30080 -> Kubernetes Node (listening on 30080) -> Forwards to Pod (targetPort: 80)
- This mechanism allows external access to services without requiring an external load balancer, making it useful for testing or internal access scenarios.



NodePort example

- Same example as before but using **NodePort** type to expose our nginx

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-nodeport
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30115 # port exposed on each node (optional)
```

```
$ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE   IP           NODE
nginx1    1/1     Running   0          65m   10.244.1.3   k8s-node
nginx2    1/1     Running   0          65m   10.244.1.4   k8s-node

$ kubectl get svc -o wide
NAME             TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)        SELECTOR
nginx-nodeport   NodePort   10.105.120.195   <none>        80:30155/TCP   app=nginx


$ kubectl describe svc nginx-nodeport
...
IPs:                      10.105.120.195
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  30155/TCP
Endpoints:                10.244.1.3:80,10.244.1.4:80


# Access the Service from outside the cluster using the IP of the node:
# curl http://<NODE_IP>:30155

curl http://192.168.81.87:30155
<html>
<body>
  <h1>Nginx 2</h1>
</body>
</html>
```

LoadBalancer Service type

- The **LoadBalancer** service type exposes a service to the outside world using an external load balancer resource.
  - It automatically provisions an external load balancer
  - integration with a load balancer provider is required.
- Cloud providers such as **Google Cloud (GKE)** and **Amazon Web Services (AWS)** automatically provision cloud-based load balancers when you create a LoadBalancer service.
- For **on-premises** Kubernetes clusters, **MetalLB** offers a similar load balancing functionality, allowing you to use external IPs and distribute traffic across your cluster nodes,
- **Difference from NodePort:**
  - **NodePort** exposes a service on a specific port across all nodes in the cluster. However, external clients must know the node's IP and port to access the service.
  - **LoadBalance** provides a single external IP that simplifies access, and it automatically distributes traffic to the pods.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-nodeport
spec:
  type: LoadBalancer
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

ExternalName Service type

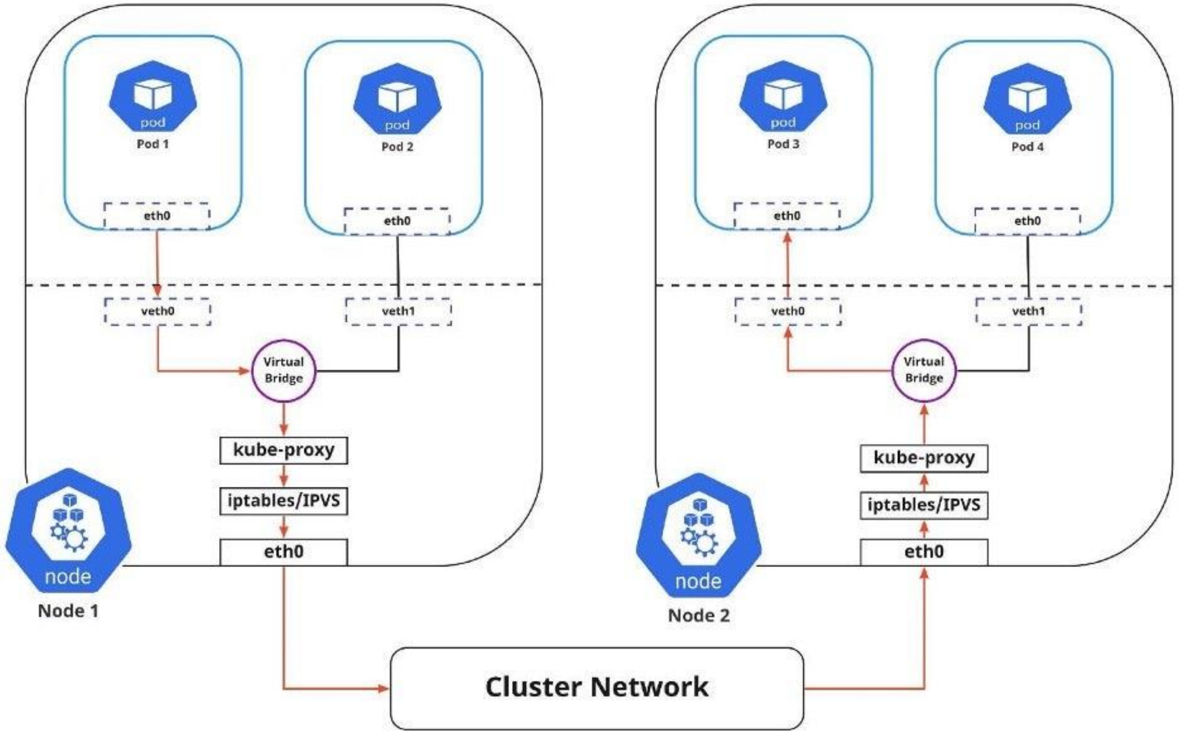
- **ExternalName** is a Kubernetes **Service** type that acts as an alias for an external DNS name instead of directing traffic to internal **Pods**.
- **How it works**
  - When a client queries the **Service**, Kubernetes responds with a **CNAME** record pointing to the external service.
  - Useful for integrating **external databases, APIs, or legacy services** without exposing internal cluster details.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: external-db
spec:
  type: ExternalName
  externalName: database.example.com
```

💡 **Pod accessing external-db will be redirected to database.example.com automatically.**

Kube-proxy

- **What is kube-proxy?**
  - A Kubernetes component running on each node.
  - Handles network traffic to Pods associated with a Service.
  - Uses **iptables** or **IPVS** to route and balance traffic.
- **How does it work?**
  - **PREROUTING:** Intercepts incoming traffic to a Service and forwards it to a Pod.
  - **POSTROUTING:** Modifies the source IP to ensure correct communication between the node and the client.
  - **Load Balancing:** Distributes traffic across available Pods
  - **Dynamic Updates:** reconfigures rules when Pods change.
- **Example of iptables rules:**
  - iptables -t nat -A PREROUTING -p tcp -d <Service-IP> --dport <Service-Port> -j DNAT --to-destination <Pod-IP>:<Pod-Port>
  - iptables -t nat -A POSTROUTING -p tcp -d <Pod-IP> --dport <Pod-Port> -j SNAT --to-source <Node-IP>



Kubernetes DNS

- Kubernetes DNS is a built-in service that enables **name resolution** within a Kubernetes cluster. It facilitates communication between pods and services using **human-readable names** instead of IP addresses.
- **DNS records** are automatically configured for all services and pods, streamlining service discovery.
- **Service Discovery:** automatically resolves service names to their Cluster IPs, enabling seamless communication.
- **Support for Namespaces:** uses Fully Qualified Domain Names (FQDNs) to uniquely identify services across namespaces.
  - service-name.namespace.svc.cluster.local
- **CoreDNS Integration**: Kubernetes uses CoreDNS as its default DNS server for efficient and scalable name resolution.

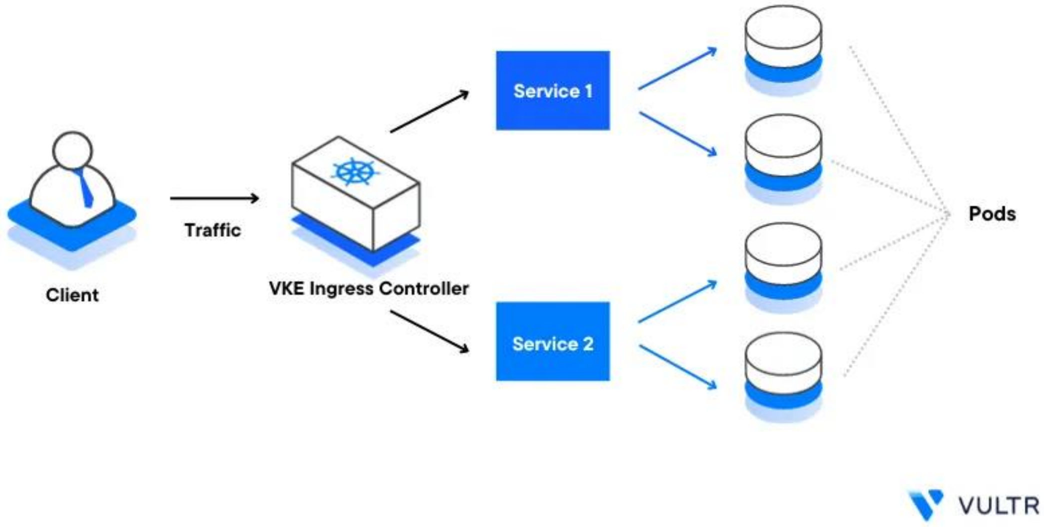
CoreDNS

Kubernetes DNS
(example)

- Assume our nginx-service is running in the default namespace, its FQAN is: **nginx-service.default.svc.cluster.local.**
- Inside a pod, use the following command to resolve the service name: **nslookup nginx-service.default.svc.cluster.local.**

```
$ kubectl get pods
NAME        READY   STATUS     RESTARTS    AGE   IP           NODE
nginx1      1/1     Running    0           65m   10.244.1.3   k8s-node
nginx2      1/1     Running    0           65m   10.244.1.4   k8s-node

$ kubectl get svc -o wide
NAME              TYPE      CLUSTER-IP     EXTERNAL-IP PORT(S) AGE SELECTOR
nginx-clusterip ClusterIP 10.103.197.222 <none>       80/TCP  46m app=nginx

$ kubectl describe svc nginx-clusterip
...
IPs:                         10.103.197.222
Port:                        <unset>  80/TCP
TargetPort:                  80/TCP
Endpoints:                   10.244.1.3:80,10.244.1.4:80

#Access the Service from within the cluster (e.g., using another Pod):

kubectl exec -it dnsutils – sh
# nslookup  10.103.197.222
222.197.103.10.in-addr.arpa  name = nginx-service.default.svc.cluster.local.

# curl nginx-service.default.svc.cluster.local.
<html>
<body>
  <h1>Nginx 2</h1>
</body>
</html>
```

Internet-to-Service
(Ingress)



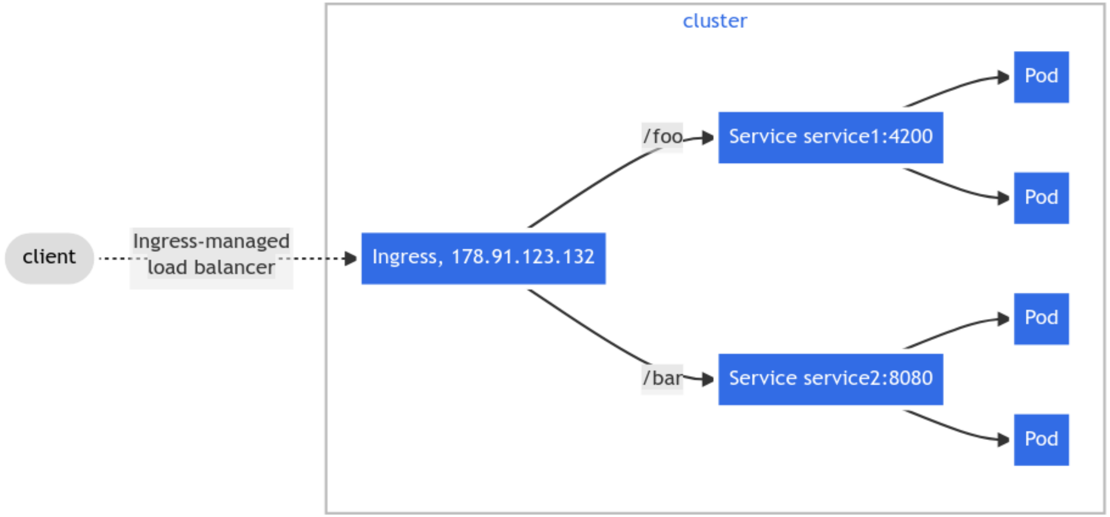
- Kubernetes Ingress is an advanced solution for managing external access to services within a cluster, centralizing traffic routing, load balancing, and secure access.

- **Ingress vs LoadBalancer Service type**

Feature	Ingress	LoadBalancer Service
OSI Layer	Layer 7 (HTTP/HTTPS)	Layer 4 (TCP/UDP)
External Exposure	Based on hostname and path	Public IP assigned by cloud provider
SSL Management	Centralized with certificates	Must be managed manually
Advanced Routing	Supported (host/path-based)	Not supported
Cost (Cloud)	More cost-effective (1 shared IP)	More expensive (1 IP per service)
Load Balancing	Application-level (reverse proxy)	Network-level

Ingress: fanout

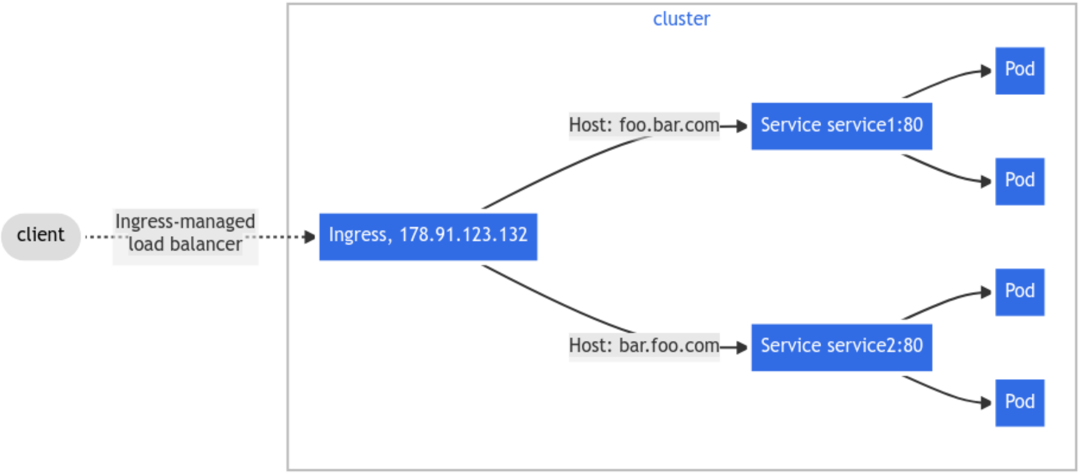
- A fanout configuration routes traffic from a single IP address to more than one Service, based on the HTTP URI being requested.




```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-fanout-example
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        pathType: Prefix
        backend:
          service:
            name: service1
            port:
              number: 4200
      - path: /bar
        pathType: Prefix
        backend:
          service:
            name: service2
            port:
              number: 8080
```

Ingress: virtual hosting

- Name-based virtual hosts support routing HTTP traffic to multiple host names at the same IP address.




```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: name-virtual-host-ingress
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service1
            port:
              number: 80
  - host: bar.foo.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service2
            port:
              number: 80
```

Thanks!

References

- https://kubernetes.io/docs/concepts/cluster-administration/networking/
- https://kubernetes.io/docs/concepts/services-networking/
- https://kubernetes.io/docs/concepts/services-networking/service/
- https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/
- https://kubernetes.io/docs/concepts/services-networking/gateway/
- https://medium.com/@extio/understanding-kubernetes-node-to-node-communication-a-deep-dive-e1d6a5ff87f3
- https://support.tools/post/kubernetes-networking-deep-dive/
- https://docs.cilium.io/en/stable/network/concepts/routing/