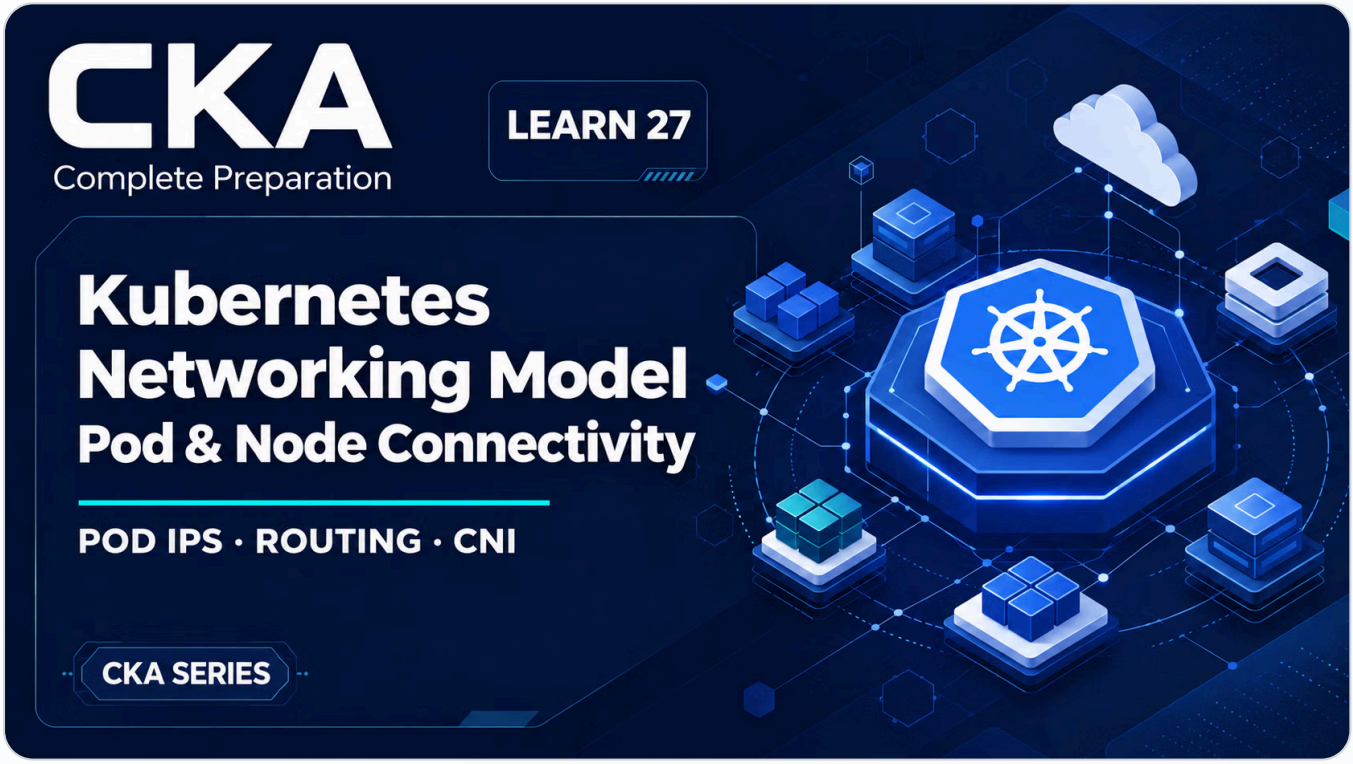
Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · OCTOBER 8, 2026

The Kubernetes Networking Model: Pod-to-Pod and Node-to-Pod Connectivity

Build a durable mental model for Pod IPs, network namespaces, same-node and cross-node packet paths, CNI responsibilities, and node-to-Pod connectivity.



This is Learn post 27 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. You already know that a Container Network Interface (CNI) implementation configures Pod networking. This lesson focuses on the outcome that implementation must produce: Pods behave as addressable network endpoints across the cluster.

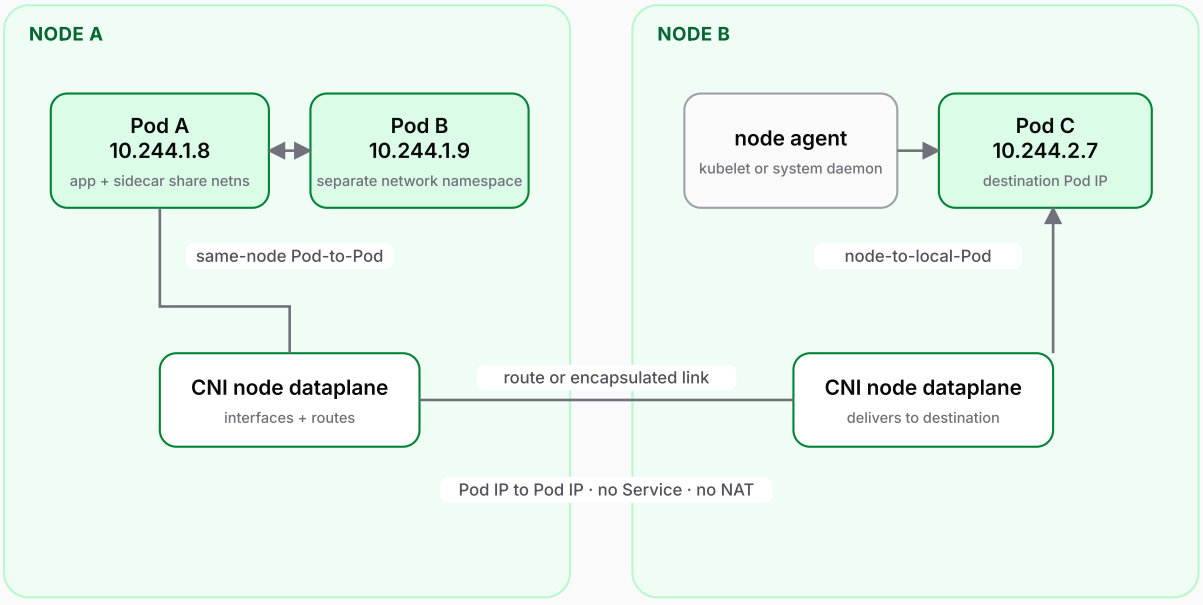
The model matters because almost every later networking resource builds on it. Services, DNS, NetworkPolicy, Ingress, and Gateway API become easier to reason about once a packet's real source and destination are clear.

What you'll learn

- Explain the cluster-wide Pod IP model and the network namespace shared by containers in one Pod.
- Follow same-Pod, same-node, cross-node, and node-to-Pod traffic at an administrator-useful level.
- Separate the Kubernetes networking contract from implementation choices such as routes, bridges, and overlays.
- Observe Pod and Node addresses, inspect a Pod's interfaces and routes, and verify direct Pod-to-Pod connectivity.
- Recognize host-networked Pods and avoid confusing Pod IPs with Node or Service IPs.

The mental model: every Pod is a small networked host

> A normal Pod owns one network namespace and one or more cluster-wide Pod IPs. Containers inside that Pod share the namespace. The Pod network makes those Pod IPs directly reachable from other Pods, regardless of node placement.



One Pod address model across nodes

Kubernetes specifies the connectivity result. The network implementation chooses how each node forwards or encapsulates the packet.

This model removes host-port coordination from ordinary Pod communication. Two Pods on different nodes can both listen on TCP port 8080 because each has its own address. A client sends to the destination Pod IP and port, not to a node port chosen for that container.

Keep the three address spaces separate

address roles · text

```
Address     Represents                    Stability
----------  ----------------------------  ----------------------------
Node IP     A machine or virtual machine  Usually infrastructure-owned
Pod IP      One Pod network namespace     Changes when a Pod is replaced
Service IP  A virtual frontend for backends  Stable for the Service lifetime
```

This lesson uses Node and Pod IPs. Service IP behavior belongs to post 28. For now, remember that a Pod IP identifies one current Pod instance; it is not an application identity and should not be stored as a durable endpoint.

```bash
kubectl get nodes \
  -o custom-columns='NODE:.metadata.name,NODE-IP:.status.addresses[?(@.type=="InternalIP")].address,POD-CIDR:.spec.podCIDR'

kubectl get pods --all-namespaces -o wide
```

The first view correlates each node with its InternalIP and, when the implementation populates it, its assigned Pod CIDR. The second correlates every Pod IP with its node. A blank podCIDR does not prove networking is broken; some network implementations manage address allocation without using Node spec.podCIDR.

Pod, Service, and Node ranges must not overlap. Because routing decisions choose the most specific matching destination, overlapping ranges can send packets toward the wrong interface or gateway. This is why kubeadm cluster planning asks you to choose a non-overlapping Pod network CIDR before installing the add-on.

One Pod means one shared network namespace

A network namespace provides an isolated view of interfaces, addresses, routes, and ports. All regular containers in a Pod join the same namespace, so they see the same Pod IP and communicate through localhost. They must also avoid binding the same address and port combination.

```bash
kubectl exec --namespace net-model client -- sh -c '
  echo "Interfaces:"
  ip addr show eth0
  echo "Routes:"
  ip route
'
```

Run this after creating the demonstration Pods later in the article. The interface shows the address assigned to the Pod; the routing table shows how destinations leave its namespace. Interface names and gateway details vary by CNI implementation, so learn the questions the command answers rather than memorizing one output.

The containerPort field does not create an interface, open a firewall, publish a host port, or make the Pod reachable. It records the port an application intends to use and can be referenced by name. Reachability comes from the Pod network and the listening process.

Kubernetes defines the contract; CNI supplies the path

When kubelet and the container runtime prepare a Pod sandbox, the node's network implementation connects that sandbox to the Pod network. On Linux, this is commonly coordinated through CNI. The implementation assigns an address, creates or attaches interfaces, and installs the state needed for packets to enter and leave the Pod namespace.

- Kubernetes-visible result: the Pod reports its assigned address in status.podIP or status.podIPs.
- Required behavior: Pods can reach other Pods directly across nodes, except where intentional traffic controls apply.
- Implementation choice: native routes, virtual switches, encapsulation, cloud routing, eBPF programs, or a combination of them.

```bash
kubectl get pods --namespace kube-system -o wide

kubectl get pod --namespace net-model client \
  -o jsonpath='{range .status.podIPs[*]}{.ip}{"\n"}{end}'
```

The kube-system view often reveals the network add-on's node agents, but names are product-specific. The JSONPath query reads every recorded Pod IP, which also works for dual-stack Pods. Inspecting status tells you the result; it does not by itself reveal how the dataplane implements the route.

Four packet paths to remember

Container to container inside one Pod

The packet stays in the shared network namespace and uses localhost. No CNI traversal between the containers is needed because there is only one Pod network stack.

Pod to Pod on the same node

The source Pod routes the packet out of its namespace. The node-local dataplane forwards it toward the destination Pod's interface. The exact bridge, virtual Ethernet pair, or eBPF path is implementation-specific, but the application still addresses the destination Pod IP directly.

Pod to Pod on different nodes

The source node must identify a path toward the node that owns the destination Pod address. A routed implementation may send the Pod packet across the underlying network. An overlay may wrap that packet inside another node-to-node packet, transport it, then remove the wrapper on the destination node. Either way, the inner communication remains Pod IP to Pod IP.

Node to Pod

Node agents such as kubelet and system daemons must be able to reach Pods on their node. A normal Linux Pod does not share the node's network namespace, but the node dataplane provides a path to its Pod IP. Whether a node can directly reach every remote Pod IP, and whether an administrator's laptop can reach any Pod IP, depends on the network implementation and surrounding infrastructure; do not infer either from Pod-to-Pod success.

Worked demonstration: observe direct Pod addressing

Create one web Pod and one long-running client Pod. No Service is created: the purpose is to observe the base Pod network directly.

```bash
kubectl create namespace net-model

kubectl run web-a \
  --namespace net-model \
  --image=nginx:1.27-alpine \
  --labels=app=web

kubectl run client \
  --namespace net-model \
  --image=busybox:1.36.1 \
  --restart=Never \
  --command -- sleep 3600

kubectl wait --namespace net-model \
  --for=condition=Ready pod/web-a pod/client \
  --timeout=90s
```

Now place the addresses beside the scheduling result. This tells you whether the following packet stays on one node or crosses the node network.

```bash
kubectl get pods --namespace net-model -o wide
```

```text
example output from a two-node cluster · text

NAME     READY   STATUS    IP           NODE
client   1/1     Running   10.244.1.8   worker-a
web-a    1/1     Running   10.244.2.7   worker-b
```

The exact addresses and placement will differ. In this example, the Pods are on different nodes, so a successful request proves the Pod network carries traffic across nodes. If both land on one node, the same request verifies the same-node path.

```bash
WEB_IP=$(kubectl get pod web-a \
  --namespace net-model \
  -o jsonpath='{.status.podIP}')

echo "web-a Pod IP: $WEB_IP"

kubectl exec --namespace net-model client -- \
  wget -qO- "http://$WEB_IP" | head
```

The HTML response comes directly from web-a's Pod IP. There is no Service lookup, stable virtual IP, or kube-proxy decision in this request. That separation is valuable: first prove the underlying Pod path, then add higher-level networking resources.

```bash
kubectl exec --namespace net-model client -- ip addr show eth0
kubectl exec --namespace net-model client -- ip route

kubectl delete namespace net-model
```

The final observations connect Kubernetes status to the Linux view inside the Pod. Deleting the namespace removes both demonstration Pods and their network attachments.

Observe the node-side decision

When you have authorized shell access to a node, ask the host kernel how it would reach a Pod IP before inspecting product-specific state.

run on a cluster node · bash

```
ip route get 10.244.2.7

curl --connect-timeout 3 http://10.244.2.7
```

Replace the example address with a current Pod IP. The route result identifies the chosen interface, gateway, and source address. A local Pod may use a directly connected path; a remote Pod may use a route, tunnel interface, or implementation-specific rule. If the remote curl fails while Pod-to-Pod traffic succeeds, verify whether broad node-to-remote-Pod reachability is supported before treating that difference as a fault.

hostNetwork is an explicit exception

A Pod with spec.hostNetwork: true uses the node's network namespace instead of receiving an ordinary isolated Pod network stack. Its reported IP commonly matches the node address, and its containers share the node's port space. This can be useful for node-level components, but it gives up the normal Pod address and port isolation.

host-network-pod.yaml · yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: host-observer
spec:
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
  containers:
    - name: observer
      image: busybox:1.36.1
      command: ["sleep", "3600"]
```

ClusterFirstWithHostNet preserves cluster-first DNS behavior for a host-networked Pod. Detailed DNS behavior belongs to post 29; the networking point here is that hostNetwork changes the namespace and address boundary.

An administrator's observation order

1. Placement: use kubectl get pods -o wide to pair source and destination Pod IPs with their nodes.
2. Endpoint: verify the destination process is listening on the intended Pod port.
3. Pod view: inspect the source interface and route, then send traffic directly to the current destination Pod IP.
4. Node boundary: determine whether the path is same-node or cross-node and inspect the host route when node access is appropriate.
5. Implementation: only then inspect the installed network add-on's Pods, logs, routes, tunnels, or product-specific tools.

This sequence moves from portable Kubernetes facts toward implementation details. It prevents a common mistake: studying a CNI daemon before confirming the address, node placement, port, or application process involved.

Common confusions resolved

quick distinctions · text

```
Confusion                          Correct model
---------------------------------  ----------------------------------------------------
Pod IP identifies an application   It identifies one current Pod instance
containerPort exposes a port       It documents a container port; it creates no path
Different nodes require a Service  No: the Pod network already connects Pod IPs
Pod network means kube-proxy       No: kube-proxy traditionally implements Services
CNI implies one packet path        No: implementations may route or encapsulate
Pod IP reachable from my laptop    Not guaranteed; cluster-private routes may be absent
hostNetwork Pod has normal Pod IP  No: it shares the node's network namespace
```

Return to the contract whenever an implementation looks complex: one Pod network namespace, a current Pod address, and a dataplane that carries traffic to the destination. Higher-level resources may select, name, filter, or expose that endpoint, but they do not replace this foundation.

Where this fits in the series

Post 25 introduced CNI as an extension boundary; this lesson explains the network model that boundary realizes. Post 28 adds stable Services and EndpointSlices over changing Pod endpoints. Post 29 covers DNS-based discovery, post 30 controls allowed flows with NetworkPolicy, and posts 31–32 introduce north-south HTTP routing. Those layers are intentionally not developed here.

Official documentation

Services, Load Balancing, and Networking defines the current Kubernetes network model and separates Pod networking from Services, Gateway API, and NetworkPolicy.

Cluster Networking explains the communication problems a cluster network solves, address-range planning, IP families, and network implementations.

Creating a cluster with kubeadm: Installing a Pod network add-on documents CNI add-on installation and the requirement to avoid overlap with host networks.

Pods describes the shared network namespace and localhost communication among containers in one Pod.