Kubernetes Networking & Service Mesh

Designing Secure, Scalable, and Observable Network Architectures for Cloud-Native Systems

Preface

Kubernetes has fundamentally transformed how we build, deploy, and operate software at scale. Yet for all the attention given to container orchestration, pod scheduling, and declarative workload management, there is one domain that remains persistently challenging—and persistently misunderstood: **networking**.

In a Kubernetes cluster, every pod must communicate. Every service must be discoverable. Every byte of traffic must be routed, secured, and observed. The networking layer is not a peripheral concern; it is the connective tissue that holds cloud-native systems together. When it works, it is invisible. When it fails, everything fails.

This book exists because Kubernetes networking deserves more than a single chapter in a general-purpose guide. It deserves deep, focused, and practical treatment.

Why This Book

Over the past several years, I've watched teams adopt Kubernetes with enthusiasm —only to hit a wall when confronted with the realities of network policies, ingress configuration, CNI plugin selection, and the decision of whether to introduce a service mesh. The documentation is scattered. The mental models are incomplete. The gap between "my pods are running" and "my system is production-ready, secure, and observable" is vast, and it is almost entirely a networking gap.

Kubernetes Networking & Service Mesh was written to close that gap. Whether you are a platform engineer hardening a multi-tenant cluster, a developer trying to understand why your service can't reach its dependency, or a networking professional transitioning into the cloud-native world, this book meets you where you are and takes you further than you expected to go.

What You Will Learn

The book is organized into a deliberate progression across **sixteen chapters and five appendices**, moving from foundational concepts to production-grade architectures:

- **Chapters 1–4** establish the core Kubernetes networking model—how pods communicate, how CNI plugins implement that model, how Kubernetes Services abstract network endpoints, and how ingress controllers manage external traffic.
- **Chapters 5–8** deepen the focus on security and reliability within Kubernetes, covering network policies, pod-to-pod encryption, load balancing strategies, and the critical role of DNS and service discovery.
- **Chapters 9–13** introduce the service mesh paradigm, exploring traffic management, mutual TLS, authorization policies, and observability—all within the context of Kubernetes workloads.
- **Chapters 14–16** address the realities of running these systems in production, including performance tuning, scaling considerations, and a forward-looking chapter for professionals evolving from traditional networking roles into cloud-native architecture.
- **Appendices A–E** provide immediately usable references: cheat sheets, network policy templates, service mesh configuration examples, troubleshooting checklists, and a roadmap for continued learning.

Every concept is grounded in Kubernetes. Every example assumes a Kubernetes environment. This is not a book about networking in the abstract—it is a book about networking *as Kubernetes demands it*.

How to Read This Book

You can read sequentially for a comprehensive education, or jump directly to the chapters that address your immediate challenges. The appendices are designed to live next to your terminal. Use them often.

Acknowledgments

No technical book is written in isolation. I owe a debt of gratitude to the Kubernetes community—the contributors, maintainers, and practitioners who have built and documented an extraordinary ecosystem. I am equally grateful to the early reviewers and technical editors whose sharp eyes and honest feedback made every chapter stronger. To my family and colleagues who endured my late nights and endless whiteboard diagrams of packet flows: thank you.

A Final Word Before We Begin

Kubernetes networking is not easy. But it is *learnable*, and once learned, it becomes one of the most powerful levers you have for building systems that are secure, scalable, and truly observable. That understanding starts here.

Let's get into it.

Dorian Thorne

Table of Contents

Chapter 1: Kubernetes Networking Model Explained

Networking is the backbone of any distributed system, and Kubernetes is no exception. When you deploy applications on Kubernetes, every Pod, every Service, and every node must communicate seamlessly. Yet the networking model that makes this possible is one of the most misunderstood aspects of the entire platform. This chapter takes you on a thorough journey through the Kubernetes networking model, starting from the fundamental principles and building up to the practical realities of how packets traverse a cluster. By the end of this chapter, you will have a solid mental model of how Kubernetes networking works, why it was designed the way it was, and how the various components interact to deliver reliable connectivity.

Understanding Kubernetes networking is not optional knowledge for anyone operating or developing on the platform. Whether you are a developer deploying microservices, a platform engineer building internal developer platforms, or a network engineer integrating Kubernetes into your existing infrastructure, the networking model affects everything you do. Misunderstanding it leads to misconfigurations, security vulnerabilities, and hours of painful debugging. Understanding it well, on the other hand, gives you the confidence to design robust, scalable, and secure architectures.

The Foundational Principles of Kubernetes Networking

Before diving into the mechanics, it is essential to understand the design philosophy that guides Kubernetes networking. The Kubernetes networking model was built on a set of deliberate principles that distinguish it from traditional container networking approaches, such as those used in early Docker deployments where port mapping and network address translation were the norm.

Kubernetes establishes three fundamental networking requirements that every cluster implementation must satisfy:

First Requirement: Every Pod gets its own unique IP address. In Kubernetes, each Pod is assigned a unique IP address within the cluster. This is not a shared IP with port-based multiplexing. Each Pod operates as if it were a standalone host on the network, with its own network namespace and its own IP. This design decision was intentional. It eliminates the complexity of port mapping and makes it straightforward for applications to discover and communicate with each other.

Second Requirement: Pods on any node can communicate with Pods on any other node without NAT. This is perhaps the most important principle. Regardless of which node a Pod is running on, it must be able to reach any other Pod in the cluster using that Pod's IP address directly. There is no network address translation in between. The IP address that a Pod sees as its own is the same IP address that every other Pod in the cluster sees when communicating with it. This flat networking model dramatically simplifies application design because applications do not need to be aware of the underlying infrastructure topology.

Third Requirement: Agents on a node can communicate with all Pods on that node. System daemons and agents, such as the kubelet, must be able to communicate with Pods running on the same node. This ensures that node-level services can interact with workloads without restriction.

These three requirements together create what is often called a "flat network" model. Every Pod can reach every other Pod, and the IP addresses are consistent and routable across the cluster. This is a significant departure from the Docker default networking model, where containers on different hosts could not communicate without explicit port forwarding or overlay configurations.

The following table summarizes the key differences between the traditional Docker networking approach and the Kubernetes networking model:

Aspect	**Traditional Docker Networking**	**Kubernetes Networking Model**
IP Assignment	Containers share the host IP; ports are mapped	Each Pod receives a unique cluster-wide IP
Cross-Host Communication	Requires explicit port mapping or overlay setup	Pods communicate directly without NAT
Port Conflicts	Applications must avoid port conflicts on the host	Each Pod has its own port namespace
Service Discovery	Manual or requires external tooling	Built into the platform via DNS and Services
Network Complexity	High due to port translation layers	Simplified flat network model
Application Awareness	Applications may need to know mapped ports	Applications use standard ports freely

This flat networking model is not implemented by Kubernetes itself. Instead, Kubernetes defines the requirements and delegates the actual implementation to a Container Network Interface (CNI) plugin. This is a critical architectural decision that we will explore in detail later in this chapter.

Pod Networking in Depth

To truly understand Kubernetes networking, you need to understand what happens at the Pod level. A Pod is the smallest deployable unit in Kubernetes, and it can contain one or more containers. All containers within a single Pod share the same network namespace. This means they share the same IP address, the same set of network interfaces, and the same port space.

When a Pod is created, Kubernetes (through the container runtime and the CNI plugin) creates a new network namespace for that Pod. A virtual ethernet pair (commonly called a veth pair) is created: one end is placed inside the Pod's network namespace, and the other end is attached to a network bridge or virtual switch on the host node. The Pod's end of the veth pair is typically named `eth0` inside the Pod, and it is assigned the Pod's unique IP address.

Let us look at this practically. If you exec into a running Pod and inspect its network interfaces, you will see something like this:

```
kubectl exec -it my-pod -- ip addr show
```

The output will typically show:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
3: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP
    link/ether 02:42:0a:f4:00:05 brd ff:ff:ff:ff:ff:ff
    inet 10.244.0.5/24 scope global eth0
       valid_lft forever preferred_lft forever
```

Here you can see the loopback interface and the `eth0` interface with the Pod's assigned IP address (10.244.0.5 in this example). The `@if7` notation indicates that this is one end of a veth pair, with the other end being interface index 7 on the host.

When multiple containers run within the same Pod, they communicate with each other over `localhost`. For example, if one container runs a web server on port 8080 and another container in the same Pod needs to access it, it simply connects to `127.0.0.1:8080`. This is because they share the same network namespace. This shared namespace is actually created by a special "pause" container (sometimes called the infrastructure container) that is started before any of the application containers. The pause container's sole purpose is to hold the network namespace open so that application containers can join it.

The following table describes the networking behavior within and between Pods:

Communication Path	**Mechanism**	**NAT Required**	**Example**
Container to container within the same Pod	Localhost (127.0.0.1)	No	App container calling sidecar on localhost:9090
Pod to Pod on the same node	Virtual bridge or direct routing	No	Pod A (10.244.0.5) to Pod B (10.244.0.6)
Pod to Pod on different nodes	Overlay network or direct routing via CNI	No	Pod on Node 1 (10.244.0.5) to Pod on Node 2 (10.244.1.3)
Pod to external service	Node's network interface with SNAT	Yes (typically)	Pod accessing an external API endpoint
External client to Pod	NodePort, LoadBalancer, or Ingress	Yes (typically)	Browser accessing a web application

Node-Level Networking and How Pods Communicate Across Nodes

Each node in a Kubernetes cluster is assigned a subnet (a CIDR block) from which Pod IP addresses are allocated. For example, Node 1 might be assigned the range 10.244.0.0/24, Node 2 might get 10.244.1.0/24, and so on. When a Pod is scheduled on a particular node, it receives an IP from that node's allocated range.

Communication between Pods on the same node is relatively straightforward. The veth pairs from each Pod connect to a common bridge (such as `cbr0` or `cni0`), and traffic between Pods is routed through this bridge. The bridge acts like a virtual switch, forwarding frames between the connected veth interfaces based on MAC addresses.

Cross-node communication is where things become more interesting and where the CNI plugin plays its crucial role. When Pod A on Node 1 wants to send a packet to Pod B on Node 2, the packet must leave Node 1's network namespace, traverse the physical (or virtual) network between the nodes, and arrive at Node 2 where it is delivered to Pod B. The CNI plugin is responsible for setting up the networking infrastructure that makes this possible.

There are generally two approaches to cross-node Pod networking:

Overlay Networking: In this approach, the CNI plugin encapsulates Pod-to-Pod traffic in an outer packet that uses the node IP addresses. Technologies like VXLAN, Geneve, or IP-in-IP are commonly used. The original packet (with Pod IP addresses) is wrapped inside a new packet (with node IP addresses) for transit across the physical network. When the packet arrives at the destination node, it is decapsulated and delivered to the target Pod. This approach works well in environments where you cannot modify the underlying network routing, such as many cloud environments or legacy data centers. The trade-off is a small overhead due to encapsulation.

Direct Routing: In this approach, the underlying network infrastructure is configured to route Pod subnet traffic directly to the appropriate nodes. This can be achieved through BGP (Border Gateway Protocol) peering, static routes, or cloud provider route tables. For example, the network knows that traffic destined for 10.244.1.0/24 should be sent to Node 2's IP address. This approach avoids encapsulation overhead and can provide better performance, but it requires more integration with the underlying network.

To inspect the networking configuration on a node, you can use standard Linux networking commands:

```
# View the network bridges on a node
ip link show type bridge

# View the routing table on a node
ip route show

# View the veth pairs
ip link show type veth

# View the iptables rules (used for Service networking)
iptables -t nat -L -n -v
```

Understanding these node-level networking constructs is essential for troubleshooting. When a Pod cannot communicate with another Pod, the problem often lies in the routing table, the bridge configuration, or the overlay tunnel between nodes.

The Container Network Interface (CNI)

The Container Network Interface is a specification and a set of libraries for configuring network interfaces in Linux containers. Kubernetes uses CNI as its standard for network plugin integration. When the kubelet needs to set up networking for a new Pod, it calls the configured CNI plugin, which handles all the necessary network plumbing.

The CNI plugin is responsible for:

1. Allocating an IP address to the Pod from the node's assigned CIDR range (IPAM, or IP Address Management).
2. Creating the veth pair and placing one end in the Pod's network namespace.
3. Configuring the bridge or routing rules on the host.
4. Setting up any overlay tunnels or routing entries needed for cross-node communication.
5. Cleaning up network resources when a Pod is deleted.

CNI configuration is typically stored in `/etc/cni/net.d/` on each node, and the CNI plugin binaries are located in `/opt/cni/bin/`. A typical CNI configuration file looks like this:

```json
{
  "cniVersion": "0.4.0",
  "name": "my-network",
  "type": "bridge",
  "bridge": "cni0",
  "isGateway": true,
  "ipMasq": true,
  "ipam": {
    "type": "host-local",
    "subnet": "10.244.0.0/24",
    "routes": [
      { "dst": "0.0.0.0/0" }
    ]
  }
}
```

This configuration tells the CNI plugin to create a bridge named `cni0`, assign IP addresses from the 10.244.0.0/24 subnet using the host-local IPAM plugin, and set up IP masquerading for outbound traffic.

Several popular CNI plugins are available for Kubernetes, each with different characteristics:

CNI Plugin	**Networking Approach**	**Key Features**	**Best Suited For**
Flannel	Overlay (VXLAN) or host-gw	Simple setup, minimal configuration	Small to medium clusters, learning environments
Calico	Direct routing (BGP) or overlay (VXLAN/IP-in-IP)	Network policies, high performance, flexible	Production clusters requiring network policies
Cilium	eBPF-based dataplane	Advanced observability, security, service mesh capabilities	Large-scale production, security-focused deployments
Weave Net	Overlay (custom protocol)	Automatic mesh, encryption support	Multi-cloud or hybrid deployments
AWS VPC CNI	Native VPC networking	Pods get VPC IP addresses, high performance	Amazon EKS clusters
Azure CNI	Native Azure VNet networking	Pods get VNet IP addresses	Azure AKS clusters
Antrea	Open vSwitch based	VMware integration, rich feature set	VMware Tanzu environments

Choosing the right CNI plugin is one of the most important decisions when setting up a Kubernetes cluster. The choice affects performance, security capabilities, observability, and operational complexity.

Note: You can verify which CNI plugin is installed on your cluster by examining the DaemonSet or Deployment in the `kube-system` namespace. For example:

```
kubectl get pods -n kube-system | grep -E "calico|flannel|cilium|
weave"
```

You can also check the CNI configuration directly on a node:

```
ls /etc/cni/net.d/
cat /etc/cni/net.d/10-flannel.conflist
```

Cluster DNS and Service Discovery

While Pod IP addresses provide direct connectivity, they are ephemeral. Pods are created and destroyed frequently, and their IP addresses change each time. Kubernetes solves this problem through Services, which provide stable virtual IP addresses (called ClusterIPs) that front a set of Pods. We will explore Services in depth in later chapters, but it is important to understand their role in the networking model at this stage.

Every Kubernetes cluster runs a DNS server (typically CoreDNS) as a cluster add-on. This DNS server automatically creates DNS records for Services and, optionally, for Pods. When a Pod wants to communicate with a Service, it can use the Service's DNS name rather than tracking individual Pod IP addresses.

For example, if you have a Service named `backend` in the `production` namespace, any Pod in the cluster can reach it using the DNS name `backend.production.svc.cluster.local`. Pods within the same namespace can use the shorter form `backend`.

You can verify DNS resolution from within a Pod:

```
kubectl exec -it my-pod -- nslookup
backend.production.svc.cluster.local
```

The output will show the ClusterIP assigned to the Service:

```
Server:     10.96.0.10
Address:    10.96.0.10#53

Name:   backend.production.svc.cluster.local
Address: 10.96.45.123
```

The DNS server address (10.96.0.10 in this example) is configured in each Pod's `/etc/resolv.conf` file by the kubelet. You can inspect it:

```
kubectl exec -it my-pod -- cat /etc/resolv.conf


nameserver 10.96.0.10
search production.svc.cluster.local svc.cluster.local
cluster.local
options ndots:5
```

The `search` domains allow short DNS names to be resolved. The `ndots:5` option tells the resolver to append search domains to any name with fewer than 5 dots before trying the name as an absolute query. This is a common source of confusion and can impact DNS performance in clusters with high query volumes.

Note: The `ndots:5` setting means that a query for `api.example.com` (which has only 2 dots) will first try `api.example.com.production.svc.cluster.local`, then `api.example.com.svc.cluster.local`, then `api.example.com.cluster.local`, and finally `api.example.com` as an absolute name. This can result in multiple unnecessary DNS queries for external domain names. You can optimize this by setting `ndots:2` in the Pod spec's `dnsConfig` field or by using fully qualified domain names (with a trailing dot) in your application configurations.

Practical Exercise: Exploring the Kubernetes Networking Model

This exercise will help you observe the networking model in action on a real Kubernetes cluster. You will create Pods on different nodes and verify that the networking principles discussed in this chapter hold true.

Step 1: Create two Pods and ensure they are scheduled on different nodes.

Create a file named `networking-lab.yaml`:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-alpha
  labels:
    app: networking-lab
spec:
  containers:
  - name: nettools
    image: nicolaka/netshoot
    command: ["sleep", "3600"]
  nodeName: ""  # Will be auto-scheduled
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-beta
  labels:
    app: networking-lab
spec:
  containers:
  - name: nettools
    image: nicolaka/netshoot
    command: ["sleep", "3600"]
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchLabels:
            app: networking-lab
        topologyKey: kubernetes.io/hostname
```

Apply the manifest:

```
kubectl apply -f networking-lab.yaml
```

Step 2: Verify the Pods are running on different nodes and note their IP addresses.

```
kubectl get pods -o wide
```

You should see output similar to:

```
NAME         READY    STATUS     RESTARTS    AGE    IP              NODE
pod-alpha    1/1      Running    0           30s    10.244.0.12
node-1
pod-beta     1/1      Running    0           30s    10.244.1.8
node-2
```

Step 3: Test Pod-to-Pod communication across nodes.

```
# From pod-alpha, ping pod-beta using its IP address
kubectl exec pod-alpha -- ping -c 3 10.244.1.8
```

You should see successful ping responses, confirming that cross-node Pod communication works without NAT.

Step 4: Verify that the source IP is preserved (no NAT).

```
# Start a TCP listener on pod-beta
kubectl exec pod-beta -- nc -l -p 8080 &

# Connect from pod-alpha and check the source IP seen by pod-beta
kubectl exec pod-alpha -- sh -c "echo hello | nc 10.244.1.8 8080"
```

Step 5: Inspect the network configuration inside a Pod.

```
kubectl exec pod-alpha -- ip addr show
kubectl exec pod-alpha -- ip route show
kubectl exec pod-alpha -- cat /etc/resolv.conf
```

Step 6: Clean up.

```
kubectl delete -f networking-lab.yaml
```

This exercise demonstrates the core principles of the Kubernetes networking model: unique Pod IPs, direct cross-node communication without NAT, and DNS-based service discovery.

Common Networking Troubleshooting Approaches

When networking issues arise in a Kubernetes cluster, having a systematic approach to diagnosis is invaluable. The following table outlines common symptoms, their likely causes, and the commands you can use to investigate:

Symptom	Likely Cause	Investigation Commands
Pod cannot reach another Pod on the same node	Bridge misconfiguration or CNI issue	`ip link show type bridge`, `brctl show`, check CNI logs
Pod cannot reach Pods on other nodes	Overlay tunnel or routing issue	`ip route show`, check CNI pod logs, verify node-to-node connectivity
DNS resolution fails inside Pods	CoreDNS not running or misconfigured	`kubectl get pods -n kube-system -l k8s-app=kube-dns`, `kubectl logs` on CoreDNS pods
Service ClusterIP not reachable	kube-proxy misconfiguration or iptables issues	`iptables -t nat -L -n`, `ipvsadm -Ln` (if using IPVS mode), check kube-proxy logs

Intermittent connectivity	MTU mismatch in overlay network	Check MTU settings on Pod interfaces and overlay tunnels, compare with node MTU
Pod can reach cluster resources but not external internet	Missing IP masquerade rules or network policy blocking egress	`iptables -t nat -L POSTROUTING -n`, check NetworkPolicy resources

Note: The `nicolaka/netshoot` container image used in the exercise above is an excellent troubleshooting tool. It includes utilities like `ping`, `traceroute`, `dig`, `nslookup`, `curl`, `tcpdump`, `iperf`, `netstat`, and many others. Keeping a troubleshooting Pod available in your cluster can save significant time when diagnosing network issues.

```
# Quick troubleshooting Pod
kubectl run netshoot --rm -it --image=nicolaka/netshoot -- /bin/
bash
```

Summary and Looking Ahead

This chapter has laid the groundwork for understanding Kubernetes networking. You have learned that the Kubernetes networking model is built on the principle of a flat network where every Pod gets a unique IP address and can communicate directly with any other Pod without network address translation. You have seen how this is implemented at the Pod level through network namespaces and veth pairs, at the node level through bridges and routing, and across nodes through CNI plugins that provide either overlay networking or direct routing.

You have also explored how DNS and service discovery fit into this model, providing stable names for ephemeral workloads. And you have gained practical experience inspecting and verifying the networking model on a live cluster.

In the chapters that follow, we will build on this foundation. We will explore Kubernetes Services in detail, examining how ClusterIP, NodePort, and LoadBalancer services work under the hood. We will dive into Ingress controllers and how external traffic enters the cluster. We will examine Network Policies for securing Pod-to-Pod communication. And ultimately, we will explore service mesh architectures that add sophisticated traffic management, security, and observability capabilities on top of this networking foundation.

Every concept in the chapters ahead depends on the networking model described here. Take the time to internalize these principles, experiment with the exercises, and build your intuition for how packets flow through a Kubernetes cluster. That intuition will serve you well as we tackle increasingly complex networking scenarios.

Chapter 2: Container Network Interfaces (CNI)

Every Kubernetes cluster, no matter how small or large, depends on a functioning network to connect its Pods, Services, and external clients. At the heart of this networking layer lies a specification that many administrators interact with daily yet rarely examine in depth: the Container Network Interface, or CNI. This chapter takes you on a thorough journey through the CNI specification, its architecture, the way Kubernetes leverages it, and the practical considerations you must weigh when choosing and configuring a CNI plugin for production workloads. By the end of this chapter, you will not only understand how CNI works under the hood but also possess the hands-on knowledge to install, configure, troubleshoot, and compare the most widely adopted CNI plugins in the Kubernetes ecosystem.

The Origin and Purpose of CNI

Before Kubernetes existed in its current form, container runtimes such as Docker handled networking internally. Each runtime implemented its own networking model, which created fragmentation. If you moved from one orchestrator to another, or even from one container runtime to another, your networking configuration and assumptions would break. The Cloud Native Computing Foundation recognized this problem early and supported the development of a standard interface that any container runtime or orchestrator could use to configure network connectivity for containers.

The Container Network Interface specification emerged as that standard. CNI defines a simple contract between the container runtime and a network plugin. The runtime calls the plugin when a container is created or destroyed, and the plugin is responsible for attaching or detaching the container to or from the network. This separation of concerns is powerful: Kubernetes does not need to know the details of how IP addresses are assigned, how routes are configured, or how overlay networks are constructed. It simply calls the CNI plugin and trusts it to do the right thing.

The CNI specification itself is deliberately minimal. It defines a set of operations, a configuration format, and a set of expected behaviors. This minimalism is intentional because it allows plugin authors tremendous flexibility in how they implement networking. Whether a plugin uses VXLAN tunnels, BGP peering, eBPF programs, or simple Linux bridges, it can conform to the CNI specification as long as it fulfills the basic contract.

Understanding the CNI Specification

The CNI specification revolves around a few core concepts that every Kubernetes administrator should internalize. At its most fundamental level, CNI is a set of Go libraries and a specification for how executables (plugins) are invoked by a container runtime.

When the kubelet on a Kubernetes node needs to set up networking for a new Pod, it does not do so directly. Instead, it delegates this task to the container runtime (such as containerd or CRI-O), which in turn invokes the configured CNI plugin. The plugin receives information about the container through environment variables and a JSON configuration passed via standard input.

The specification defines the following primary operations:

Operation	Description	When It Is Called
ADD	Attaches a container to a network, assigns an IP address, and sets up necessary routes and interfaces	When a new Pod sandbox is created on the node
DEL	Removes a container from a network, releases the IP address, and cleans up interfaces and routes	When a Pod sandbox is being destroyed on the node
CHECK	Verifies that the container networking is still correctly configured and functional	Periodically or on demand to validate network health
VERSION	Reports the CNI specification versions that the plugin supports	During initialization to ensure compatibility

The ADD operation is the most critical. When called, the plugin must create a virtual ethernet (veth) pair, place one end inside the container network namespace, assign an IP address from the configured range, set up default routes inside the container, and optionally configure the host side of the connection. The plugin then returns a JSON result to the runtime that includes the assigned IP address, the gateway, and any DNS configuration.

The configuration file for a CNI plugin is typically stored in `/etc/cni/net.d/` on each node. The kubelet reads the configuration files from this directory and uses them to determine which plugin to invoke. A simple configuration file might look like this:

```
{
  "cniVersion": "1.0.0",
  "name": "my-cluster-network",
  "type": "bridge",
  "bridge": "cni0",
  "isGateway": true,
  "ipMasq": true,
  "ipam": {
    "type": "host-local",
    "subnet": "10.244.1.0/24",
    "routes": [
      { "dst": "0.0.0.0/0" }
    ]
  }
}
```

In this configuration, the `type` field tells the runtime which plugin binary to execute. The binary must be located in the CNI binary directory, which is typically `/opt/cni/bin/`. The `ipam` block specifies how IP address management is handled. In this case, the `host-local` IPAM plugin manages a local subnet and assigns addresses from it.

Note: The CNI plugin binaries are standalone executables. They are not long-running daemons. Each time a Pod is created or destroyed, the binary is invoked, performs its work, and exits. Some CNI solutions do run additional daemons for coordination and route distribution, but the CNI plugin itself is always an executable that follows the invoke-and-exit pattern.

How Kubernetes Invokes CNI Plugins

Understanding the exact sequence of events when a Pod is scheduled to a node helps demystify the networking layer. Let us walk through the process step by step.

First, the Kubernetes scheduler assigns a Pod to a specific node. The kubelet on that node receives the Pod specification and begins the process of creating the Pod sandbox. The sandbox is a pause container that holds the network namespace for all containers in the Pod.

Second, the kubelet instructs the container runtime (for example, containerd) to create the sandbox. The runtime creates a new network namespace for the sandbox.