O'REILLY®

Cilium
Up and Running

Cloud Native Networking, Security, and Observability

Early Release
RAW & UNEDITED

Compliments of
ISOVALENT
now part of CISCO

Nico Vibert, Filip Nikolic
& James Laverack

Cisco

ISOVALENT
now part of cisco

Enterprise Kubernetes Networking Built and Supported by the Creators of Cilium

Optimize your Kubernetes environment and deliver reliability at scale with Isovalent Networking for Kubernetes. Based on Cilium it offers industry-leading CNI, seamless connectivity, deep security controls, and full visibility into your infrastructure.

One Platform for All Your Enterprise Requirements
One Platform for Security, Networking, and Observability

Deliver consistent networking to all your applications - no more point solutions or translating policies across providers.
Eliminate the need for sidecar proxies and provide high-performance, scalable networking.

Simplify operations while improving visibility and control over service-to-service communications.

The Isovalent Enterprise Platform offers:

- SIEM Export & Threat Detection
- Fault Analytics & Troubleshooting
- Network & API Level Tracing & Metrics
- Multi-Cloud / Multi-Cluster Networking
- Load Balancing
- Service Mesh
- Micro Segmentation
- Transparent Encryption

Learn more at:

isovalent.com/product

Cilium: Up and Running

Cloud Native Networking, Security, and Observability

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Nico Vibert, Filip Nikolic, and James Laverack

Cilium: Up and Running

by Nico Vibert, Filip Nikolic, and James Laverack

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Megan Laddusaw

Development Editor: Gary O'Brien

Production Editor: Christopher Faucher

Interior Designer: David Futato

Cover Designer: Karen Montgomery

March 2026: First Edition

Revision History for the Early Release

- 2025-05-14: First Release
- 2025-06-16: Second Release
- 2025-07-28: Third Release
- 2025-10-14: Fourth Release

See http://oreilly.com/catalog/errata.csp?isbn=9798341622999 for release details.

979-8-341-62295-1

Brief Table of Contents (*Not Yet Final*)

Chapter 1. Inside Cilium

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form —the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo is available at *https://github.com/isovalent/cilium-up-and-running*.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *gobrien@oreilly.com*.

Cilium is a powerful eBPF based networking and security solution for Kubernetes and beyond. Understanding its architecture is crucial before you install it and get up and running. This chapter provides an overview of Cilium's core components and how they interact and work together to address the use cases described in Chapter 1. It will provide you with a clear overview of how functionalities are distributed among the various components of Cilium.

By the end of this chapter you will have the knowledge to identify which components are involved in different scenarios, empowering you to approach later sections with confidence.

Cilium at a Glance

You read in Chapter 1 about the cloud native networking, security and observability use cases Cilium addresses. Cilium's versatility can

be attributed to its modular architecture: it is made up of several internal components, each providing different functionalities.

- Cilium Agent
- Cilium CNI Plugin
- Cilium Operator
- Cilium CLI
- Hubble, with its sub-systems (Hubble Server, Hubble Relay, Hubble CLI and Hubble UI)
- eBPF programs
- Envoy Proxy
- DNS Proxy

Here is a brief overview of how all these building blocks fit together.

The Cilium agent is the core component of Cilium: it installs the Cilium CNI plugin, loads the eBPF programs used to forward, filter and observe traffic and monitors the custom resource definitions (CRDs) created by the Cilium operator. When a pod sends traffic, an eBPF program decides whether to forward or drop the traffic and may send it to a proxy for further processing (Envoy for HTTP-related traffic and DNS for DNS-related requests). The Cilium CLI is a binary that primarily assists with managing Cilium. Meanwhile, the Hubble server observes and enriches traffic data, with Hubble relay aggregating it across all cluster nodes. The Hubble UI and CLI provide options to visualize network activity in a user interface and a terminal, respectively.

Cilium Agent

The Cilium agent, often referred to simply as “Cilium” or the “Cilium daemon,” is the core component of Cilium. It is responsible for

orchestrating several critical tasks across the Kubernetes cluster, such as installing the CNI plugin and loading eBPF programs. To accomplish this, it runs as a DaemonSet on every node with elevated privileges. The Cilium agent plays a pivotal role in ensuring the efficient operation of both the network and security within the cluster, as shown in Figure 2-1. Its primary responsibilities include:

Installing the CNI Plugin

The agent manages the Container Network Interface (CNI) plugin, ensuring that Pod networking is correctly configured and providing seamless network connectivity.

Loading eBPF Programs and Maps

The agent loads and manages eBPF (extended Berkeley Packet Filter) programs and maps on each node. These programs enforce network policies, redirect traffic, perform load balancing, and various other fundamental functions.

Reconciling Kubernetes State

The Cilium agent continuously monitors the desired Kubernetes state by communicating with the kube-apiserver. It checks for changes in Pods, NetworkPolicies and other objects. The agent then updates eBPF maps as needed to reflect these changes.

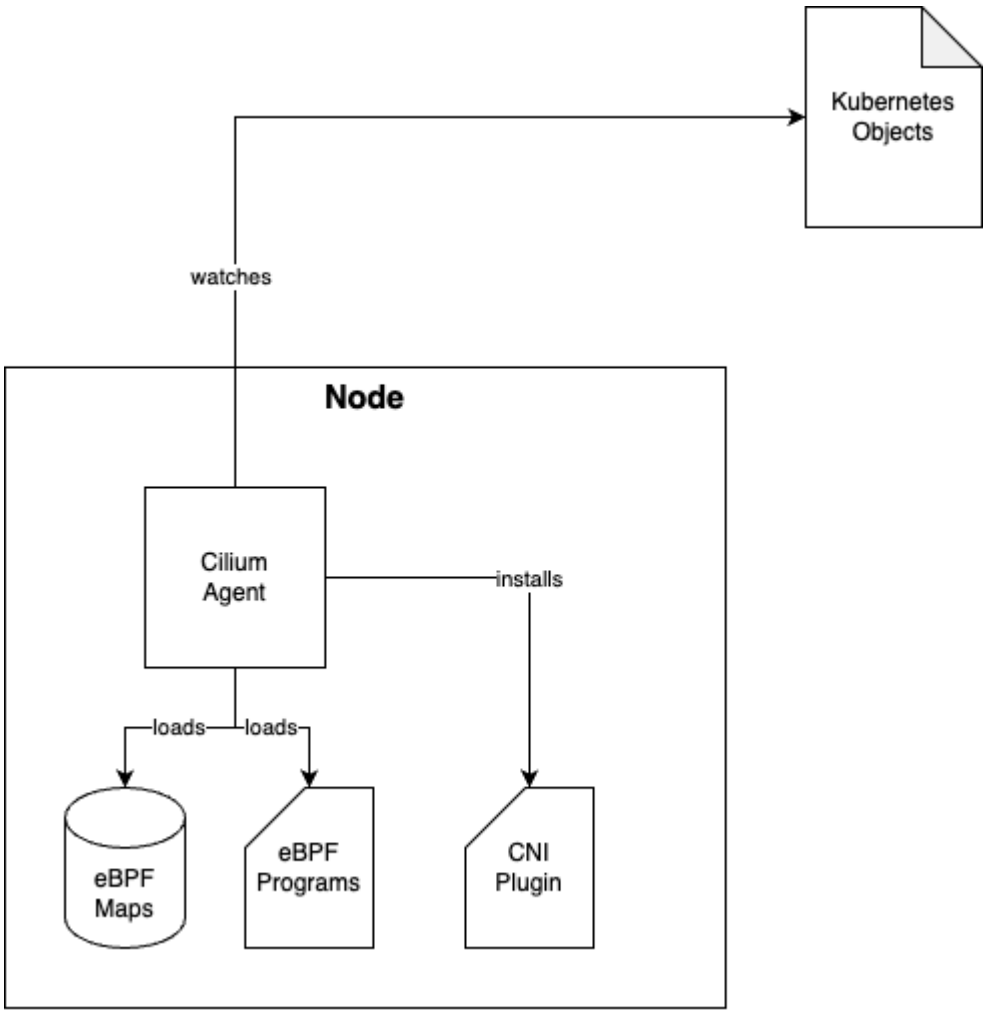


Figure 1-1. The Cilium agent installs the CNI plugin, loads the required eBPF programs and eBPF maps into the kernel and continuously watches over various Kubernetes objects, such as Pods and NetworkPolicies. It then updates the eBPF map accordingly.

For debugging and troubleshooting, the Cilium agent provides a suite of tools when executing into the Cilium agent Pod. The most notable is cilium-dbg. This tool helps gather detailed diagnostic information on the node and should not be confused with the Cilium CLI, which is used for managing Cilium at a higher level across the entire cluster.

NOTE

Previously, `cilium-dbg` was named `cilium`, which caused confusion since it was unclear whether people were referring to the "Cilium CLI" or the Cilium agent debugging tool. Keep this in mind when reviewing older issues or documentation.

Cilium CNI Plugin

Cilium is often referred to as a CNI (Container Network Interface), but its CNI functionality is just one component of a larger system working together to manage networking in Kubernetes environments.To better understand the role of Cilium CNI let's quickly talk about CNIs in general.

The Container Network Interface (CNI) is a CNCF project that specifies the relationship between a Container Runtime, such as containerd or CRI-O, responsible for container creation, and a CNI plugin tasked with configuring network interfaces within the container upon execution. Ultimately, the CNI plugin performs the substantive tasks of configuring the network, while CNI primarily denotes the interaction framework. However, it's common practice to simply refer to the CNI plugin as "CNI".

Picture a scenario where a user initiates the creation of a Pod and submits the request to the kube-apiserver (through, for example, applying a manifest with `kubectl`). Following the scheduler's determination of the node where the Pod should be deployed, the kube-apiserver lets the corresponding kubelet know. The kubelet, rather than creating containers directly, delegates this task to a container runtime. The container runtime's responsibility encompasses container creation, including the establishment of a network namespace. Once this setup is complete, the container runtime calls upon a CNI plugin to generate and configure virtual ethernet devices and necessary routes.

This flow is depicted in Figure 2-2.

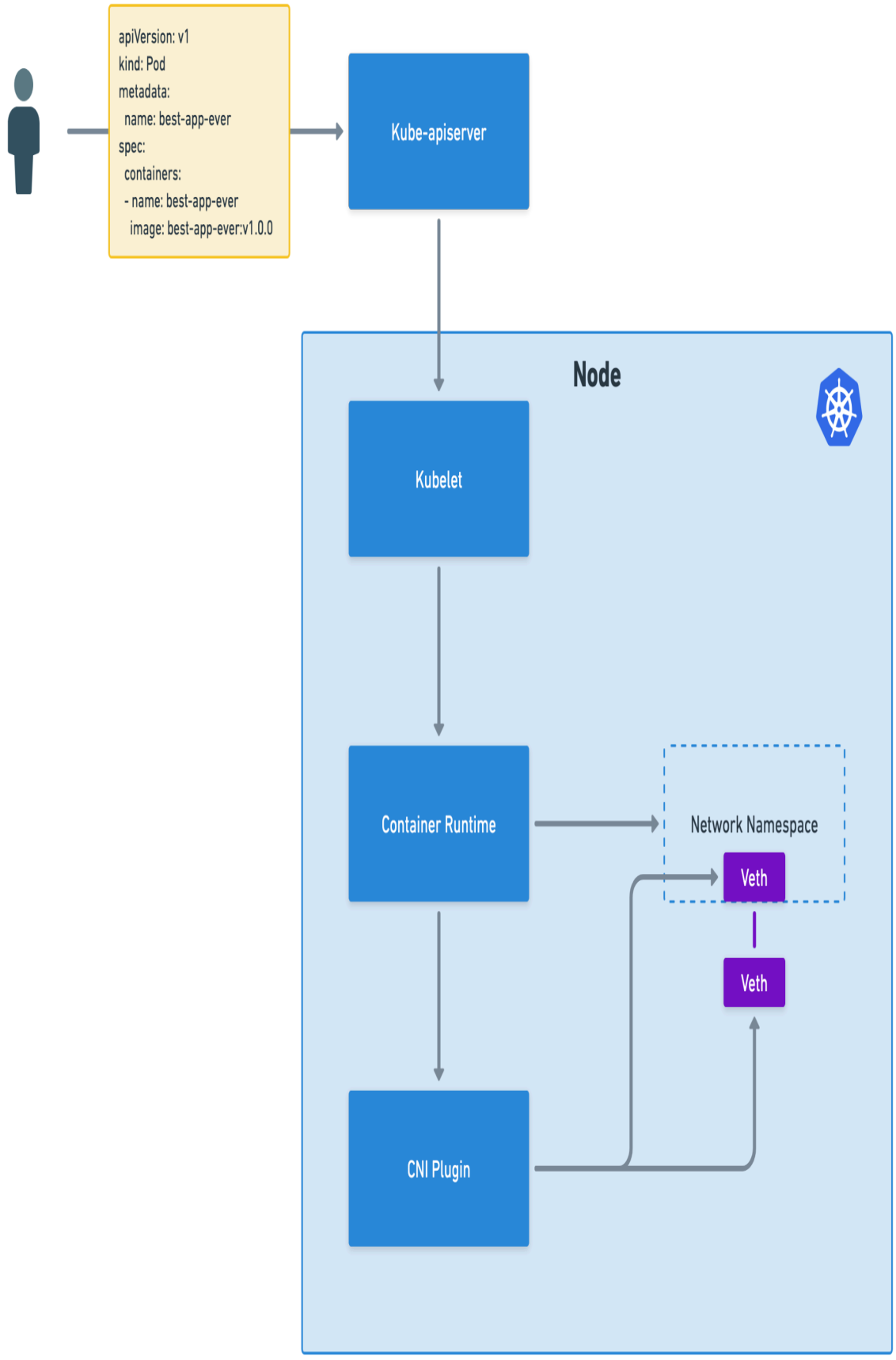

apiVersion: v1
kind: Pod
metadata:
name: best-app-ever
spec:
containers:
- name: best-app-ever
image: best-app-ever:v1.0.0
Kube-apiserver
Node
Kubelet
Container Runtime
Network Namespace
Veth
Veth
CNI Plugin

Figure 1-2. [Caption to come]

> **NOTE**
>
> For more details on the process of a CNI, visit https://github.com/f1ko/demystifying-cni or https://github.com/containernetworking/cni

The Cilium CNI consists of two key files located on every node:

- */etc/cni/net.d/05-cilium.conflist*
- */opt/cni/bin/cilium-cni*

The Cilium agent creates these files, which are then read and executed by the container runtime whenever a Pod is created, deleted, or during periodic status checks.

> **NOTE**
>
> Please note that CNIs typically do not handle traffic forwarding or load balancing. By default, kube-proxy serves as the default network proxy in Kubernetes which utilizes technologies like iptables, IPVS, and more recently nftables, to direct incoming network traffic to the relevant Pods within the cluster. However, Cilium offers a superior alternative by loading eBPF programs into the kernel, achieving the same tasks with significantly better performance, especially at scale. For more information on this topic, see Chapter 5.

Cilium Operator

While the Cilium agent operates at the node level, the Cilium operator functions at the cluster level and runs as a Deployment within the Kubernetes cluster. Certain features and environments may modify the operator's responsibilities, but two key tasks stand

out: managing Custom Resource Definitions (CRDs) and handling IP address management (IPAM).

Unlike many other projects, when Cilium is installed using `helm install`, it does not automatically install CRDs. This approach allows the Cilium operator to manage CRDs across versions, ensuring backward compatibility where needed.

> **NOTE**
>
> This means that Cilium custom resources, such as `CiliumNetworkPolicy`, cannot be applied during the initial installation of Cilium. Before applying these resources, you must wait for the Cilium operator to start up and successfully register the CRDs.

Beyond CRD registration, the Cilium operator also manages IP address allocation (IPAM) as shown in Figure 2-3. By default, it creates a `CiliumNode` object for each node (after registering the CRDs) and populates it with relevant information, such as the podCIDRs. Since the operator operates at the cluster level, it is aware of the IP ranges used by other nodes, therefore ensuring no overlaps. The Cilium agent then reads the data in the `CiliumNode` object to perform its duties.

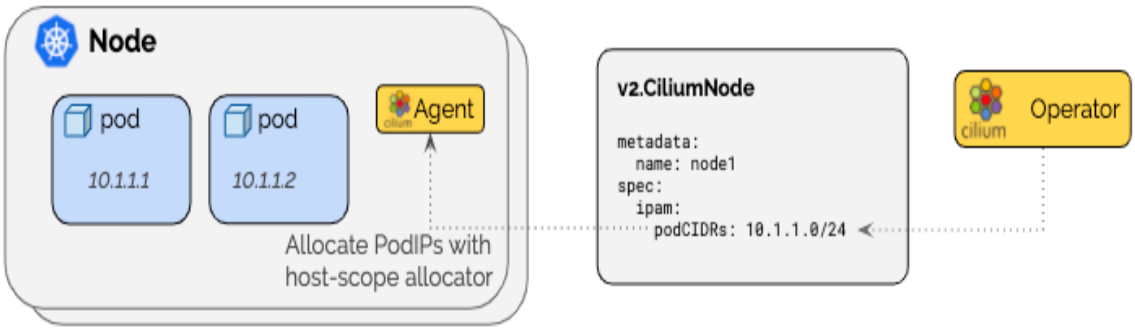


Figure 1-3. [Caption to come] (Source: Cilium documentation)

Cilium CLI

To facilitate the management and configuration of Cilium, a separate command-line binary named Cilium CLI (Command Line Interface) can be installed on your machine.

The Cilium CLI binary enables you to do the following:

- Monitor the status and health of Cilium
- Run a battery of connectivity tests
- Execute operational commands, such as rotating encryption keys and collecting sysdumps
- Install, uninstall Cilium
- Upgrade and downgrade Cilium
- Enable and disable Cilium features

The most commonly used Cilium CLI command is `cilium status`. You can use it to easily see the health of the Cilium components described in this section.

```
$ cilium status
    /¯¯\
 /¯¯\__/¯¯\    Cilium:             OK
 \__/¯¯\__/    Operator:           OK
 /¯¯\__/¯¯\    Envoy DaemonSet:    OK
 \__/¯¯\__/    Hubble Relay:       disabled
    \__/       ClusterMesh:        disabled

DaemonSet              cilium             Desired: 4,
Ready: 4/4, Available: 4/4
DaemonSet              cilium-envoy       Desired: 4,
Ready: 4/4, Available: 4/4
Deployment             cilium-operator    Desired: 1,
Ready: 1/1, Available: 1/1
Containers:            cilium             Running: 4
                       cilium-envoy       Running: 4
                       cilium-operator    Running: 1
```

```
Cluster Pods:          3/3 managed by Cilium
Helm chart version:    1.17.1
Image versions         cilium
quay.io/cilium/cilium:v1.17.1@sha256:8969bfd9c87cbea91e4066
5f8ebe327268c99d844ca26d7d12165de07f702866: 4
                       cilium-envoy
quay.io/cilium/cilium-envoy:v1.31.5-1739264036-
958bef243c6c66fcfd73ca319f2eb49fff1eb2ae@sha256:fc708bd3697
3d306412b2e50c924cd8333de67e0167802c9b48506f9d772f521: 4
                       cilium-operator
quay.io/cilium/operator-
generic:v1.17.1@sha256:628becaeb3e4742a1c36c489772109237589
1b58bae2bfcae48bbf4420aaee97: 1
```

You will use Cilium CLI throughout the book — you might want to download and install it on your local machine if you intend to follow the examples. The Cilium CLI uses information from the *kubeconfig* to access the cluster via the Kubernetes API. The Cilium CLI is available for Linux, macOS, and Windows machines.

> **NOTE**
>
> Note that a Cilium CLI client is installed by default inside the Cilium agent container, which goes by the name cilium-dbg. The "in-agent" CLI interacts with the Cilium agent running on the same node and provides more verbose information about the Cilium configuration and state, down to the eBPF maps. In this book, references to Cilium CLI will imply the "client CLI", unless we specify otherwise.

We will provide more information about the "in-agent CLI" in Chapter 16: Operations.

Hubble

Hubble, a sub-system of Cilium, provides network observability. Hubble builds on top of Cilium and eBPF to provide deep network

visibility information of Kubernetes applications and services by tapping into Cilium's eBPF datapath. Hubble is an optional component of Cilium and is disabled by default.

Hubble is made up of several components (Figure 2-4). The Hubble server captures data locally, the relay aggregates it across the cluster while the UI and the CLI provide two options to interact and consume the data aggregated by the Relay.

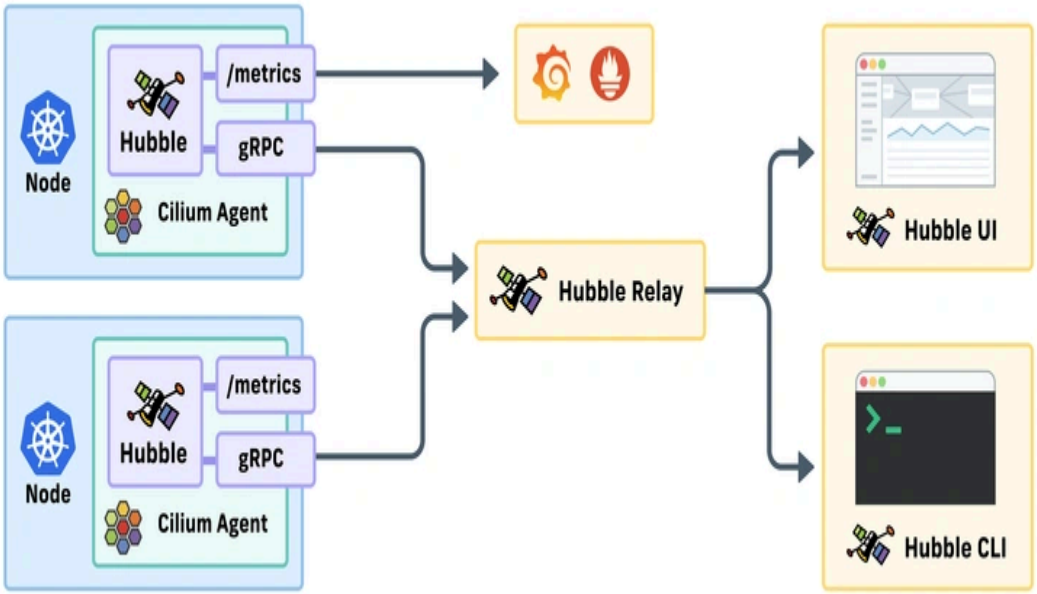


Figure 1-4. [Caption to come] (Source: Isovalent)

Hubble Server

The Hubble server runs on each node and retrieves the eBPF-based visibility from Cilium. The server is embedded into the Cilium agent and captures *flow logs*, which are then gathered across all cluster nodes by the Hubble relay to provide a richer cluster-wide view of network traffic. The flow logs can then be displayed in the Hubble

UI or interpreted in the Hubble CLI for monitoring, troubleshooting and alerting.

A flow log includes a wealth of information about:

- a timestamp
- a source pod, along with its namespace, port, and Cilium identity
- the direction of the flow (->, <-, or at times <> if the direction could not be determined)
- a destination pod, along with its namespace, port, and Cilium identity
- a trace observation point (e.g. to-endpoint, to-stack, to-overlay)
- a policy verdict (e.g. FORWARDED or DROPPED)
- a layer 4 protocol (e.g. UDP, TCP), optionally with flags

Don't worry if you are not familiar with concepts such as policy verdict and identities, we will cover them in Chapter 12 (Network Policy).

Here is an example of a flow log for an ICMPv6 packet sent from a pod named `pod-worker` on a node named `kind-worker` to a pod named `pod-worker2` located on node `kind-worker2`.

```
$ hubble observe --ipv6 --from-pod pod-worker
Sep  7 15:11:18.288: default/pod-worker (ID:3211) ->
default/pod-worker2 (ID:50760) to-overlay FORWARDED (ICMPv6
EchoRequest)
Sep  7 15:11:18.289: default/pod-worker (ID:3211) ->
default/pod-worker2 (ID:50760) to-endpoint FORWARDED
(ICMPv6 EchoRequest)
```

You will explore how to use, interpret, and display flow logs in more detail in Chapter 15.

In order for clients to consume flow logs, the Hubble server exposes a gRPC API over a local UNIX domain socket or via an API, for external systems such as Hubble Relay. The Hubble Server also exposes Prometheus metrics, which can be exported and visualized on dashboards such as Grafana.

Hubble Relay

The Hubble Relay system provides a cluster-wide API for querying Hubble flow data, which can be accessed directly or via the Hubble CLI and UI.

The Hubble Relay consumes the gRPC API from each Hubble Server to provide multi-node support. Hubble Relay connects to every node in a cluster and maintains a persistent connection with every Hubble server across the cluster. Before the Hubble Relay was released, Hubble could only operate on a per-node basis and building a cluster-wide view required a tedious iteration across multiple Hubble instances in the cluster. Hubble Relay aggregates data from all Hubble servers and serves it on one query endpoint.

> **WARNING**
>
> Given that Hubble Relay collects cluster-wide flows you should treat it as a sensitive component. Connections between Hubble instances and Hubble Relay are secured using mutual TLS by default and users can choose to use tools such as cert-manager to generate the certificates automatically.

When the Hubble feature is enabled, Hubble Relay is automatically deployed on the Kubernetes cluster through a Kubernetes Deployment.

Hubble CLI

The Hubble CLI is a binary you can use to visualize and filter network flow logs collected from the Hubble Relay.

Just like Cilium CLI, Hubble CLI is installed and managed separately from Cilium and Hubble. Platform engineers would typically install Hubble CLI on their machine and run Hubble CLI for forensics and troubleshooting purposes. Hubble CLI connects to the Hubble Relay component to provide a cluster-wide view of traffic.

```
$ hubble observe --to-fqdn api.github.com
Aug  3 15:12:13.929: tenant-jobs/crawler-5c645d68f4-
qchk8:47180 (ID:21067) -> api.github.com:80 (world) policy-
verdict:all EGRESS ALLOWED (TCP Flags: SYN)
```

Note that Hubble CLI can be used *within* the Cilium agent to display flow logs captured by Hubble. In that case, the flow logs reported would only be the logs observed by the local Cilium agent, rather than the cluster-wide logs aggregated by Hubble Relay.

The Hubble CLI binary is installed by default in Cilium agent pods. With the following kube-ctl command, you can observe traffic locally to view the last observed flow on the node where the Cilium agent is deployed:

```
$ kubectl exec -n kube-system -it cilium-gb945 -c cilium-
agent -- hubble observe --last 1
Feb 20 12:18:35.872: 10.244.2.7:58762 (host) ->
kube-system/hubble-relay-d9495cdc-wfd6l:4222 (ID:3873) to-
endpoint FORWARDED (TCP Flags: ACK)
```

The Hubble CLI is available for Linux, macOS, and Windows machines.

Hubble UI

The other method to consume the data collected by Hubble is through the Hubble User Interface. The graphical UI utilizes the aggregated data from Hubble Relay to provide a graphical service dependency and connectivity map.

> **NOTE**
>
> Hubble UI is not installed by default when you install Hubble. It must be installed separately.

Hubble UI can also visualize all flows across the cluster. The information displayed is namespace-based. By selecting the `kube-system` namespace, you can visualize the interactions between Hubble UI and Hubble Relay itself.

eBPF Programs & eBPF Maps

eBPF (extended Berkeley Packet Filter) is a revolutionary technology that enables users to extend the behavior of the Linux kernel without the need to submit patches, wait for review, or rebuild and update the kernel. Instead, eBPF allows for the creation of custom code that can be directly injected into the kernel, providing immediate changes to its behavior—without requiring a reboot.

One of the primary reasons eBPF has gained widespread adoption is its ability to provide safe execution within the kernel. Furthermore, it enables users to write highly efficient programs that execute at precise points in the kernel's workflow. By attaching programs to the most relevant hooks in the kernel, users can act at the earliest possible moment. This improves performance by avoiding unnecessary work later in the kernel's execution cycle.

Two key concepts are fundamental to understanding eBPF:

eBPF programs

Stateless, event-driven pieces of code triggered by specific events within the kernel. These programs are attached to "hook points" in the kernel and when those points are reached, the corresponding eBPF program is executed. Therefore, the programs can operate in real-time, providing immediate response to kernel events.

eBPF maps

Serve as in-kernel data storage. These maps allow eBPF programs (and other processes) to store and retrieve data, enabling dynamic decision-making. Just as a stateless microservice may need to query a database for information before making a decision, eBPF programs can perform lookups in eBPF maps to guide their actions based on the stored data.

In the context of Cilium, eBPF programs are used to implement critical networking and security functions (Figure 2-5). For example, eBPF programs can be attached to the kernel's network-relevant hook points to inspect network packets and determine if they should be allowed or dropped by Network Policies. To help with the decision, the eBPF program performs a lookup in an eBPF map containing information about allowed sources and destinations. The Cilium agent continually updates these eBPF maps to reflect the desired state as specified by NetworkPolicies.

NOTE

Not all network or security decisions in Cilium are made using eBPF. Although eBPF has the potential to handle more complex decisions, such as layer 7 policies, this is not the way Cilium is implemented. At the time of writing, layer 7 decisions are handled in userspace, typically by an Envoy or DNS proxy.

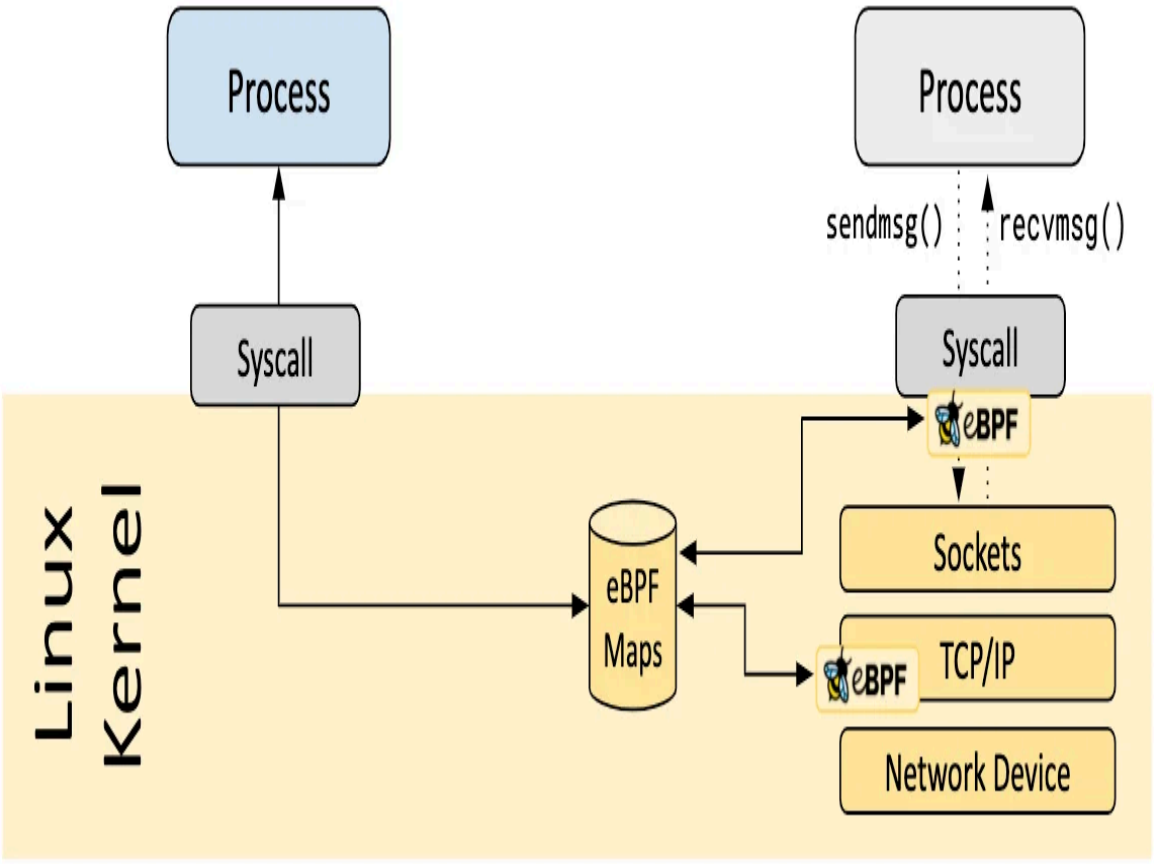


Figure 1-5. [Caption to come] (Source: [eBPF documentation](#))

> **NOTE**
>
> While eBPF was initially developed for the Linux kernel, its potential has led to efforts to port eBPF to other kernels, such as the Windows kernel, broadening its use across different platforms.

Envoy Proxy

Envoy is a high-performance open source proxy and a key project within the CNCF, widely recognized for its scalability and robust feature set. In Cilium, Envoy plays a crucial role in managing and controlling HTTP traffic, enforcing HTTP-based policies and supporting sophisticated traffic management decisions.

Envoy implements important features such as HTTP-based CiliumNetworkPolicies, Ingress & Gateway API, and more. These features allow for fine-grained control over HTTP traffic, ensuring that routing decisions, security policies, and other configurations can be applied based on specific HTTP headers, methods, paths, and additional data contained within the HTTP payload.

To accomplish this, HTTP traffic is routed through the Envoy proxy where layer 7 interactions are needed.

To minimize performance penalties Envoy runs on each node through its own dedicated DaemonSet, as you can see in the following code:

```
$ kubectl get -n kube-system daemonset/cilium-envoy
NAME          DESIRED CURRENT READY UP-TO-DATE AVAILABLE
NODE SELECTOR           AGE
cilium-envoy 3       3       3     3           3
kubernetes.io/os=linux 28h
```

Sending traffic through Envoy ensures that security and traffic management can be handled in a highly dynamic environment, particularly for cloud-native applications. By offloading HTTP-related decisions to Envoy, Cilium is able to leverage Envoy's extensive capabilities for managing HTTP traffic, all while continuing to rely on eBPF-based networking decisions for lower layers of the stack.

> **WARNING**
>
> Since HTTP functionality relies on the Envoy proxy, applications will be impacted if the Envoy proxy becomes unavailable.

DNS Proxy

The DNS Proxy in Cilium enables network security decisions to be made based on fully qualified domain names (FQDNs), adding an

important layer of granularity to security enforcement. When using FQDN-based network policies, the DNS proxy listens to and captures DNS requests and responses, allowing it to learn which domain names resolve to which IP addresses. Once this mapping is established, the network policies can allow or deny traffic based on domain names To achieve this, it's essential that DNS traffic is routed through the DNS proxy.

The DNS proxy is especially beneficial in dynamic, cloud-native environments where IP addresses are often ephemeral and subject to change. In these scenarios, services might scale or change their IP addresses frequently. The DNS proxy ensures that security policies remain effective in such environments by enforcing rules based on domain names rather than static IPs. For example, blocking traffic to a specific domain becomes possible, regardless of the IP addresses that may shift over time, providing a more flexible approach to network security.

The DNS proxy also enhances Hubble flow data with DNS service identities, enabling you to monitor and troubleshoot DNS activities and identify anomalous behaviour. Once enabled, Hubble will report the names of services in its interface.

Although the DNS proxy is considered a separate component in Cilium, it operates on every node within the cluster as part of the Cilium agent Pod. This approach minimizes performance overhead by ensuring that DNS traffic is routed locally on the same node, reducing the need for inter-node communication.

WARNING

Since FQDN functionality relies on the DNS proxy, which is part of the Cilium agent, applications will be impacted if the Cilium agent becomes unavailable.

You will learn more about the DNS proxy in Chapter 12 (Network Policy).

> **NOTE**
>
> Another optional component of Cilium is the Cluster Mesh API server.
>
> Cilium Cluster Mesh provides connectivity and service load balancing across multiple clusters and leverages a component called the Cluster Mesh API server. You will learn more about it in Chapter 8.

Putting it all together

In this chapter we have explored the individual components of Cilium. Let's dive into how they interact with each other to get a better understanding of the big picture (Figure 2-6).

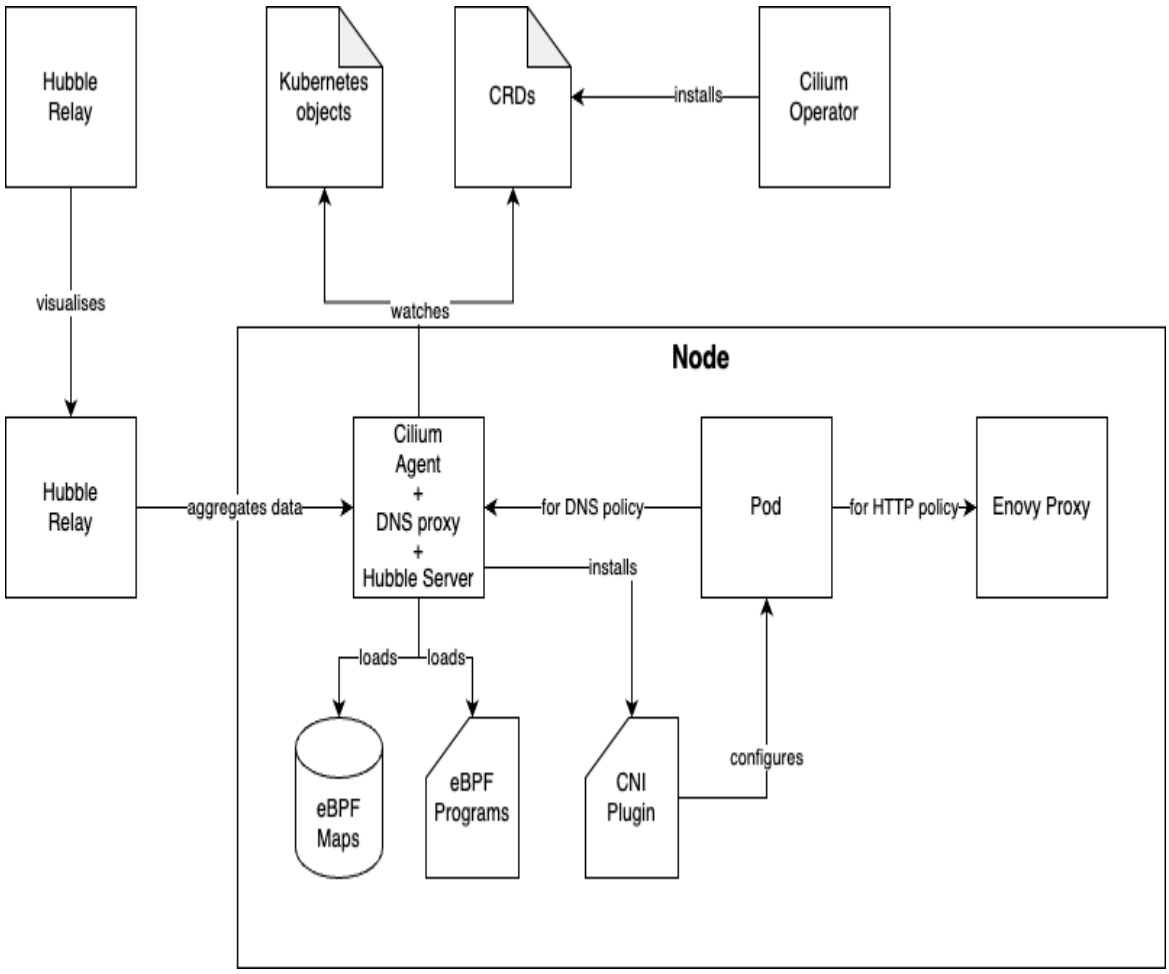


Figure 1-6. [Caption to come]

The Cilium Operator is responsible for creating various CRDs. These CRDs, along with other Kubernetes objects, are monitored by the Cilium agent. The Cilium agent uses this information to load and configure eBPF maps and programs as needed. Additionally, the agent installs the CNI plugin, which is responsible for setting up networking for each new Pod.

When a Pod sends traffic it reaches a hook point where an eBPF program is triggered. This program evaluates the traffic and decides whether to allow or drop it. If layer 7 or FQDN policies are in use, the eBPF program forwards the traffic to either the DNS proxy for DNS-related requests or the Envoy proxy for HTTP-related traffic.

The Envoy proxy processes the traffic and applies its own logic to decide whether to allow or drop it. For DNS traffic, the DNS proxy captures DNS responses, maps IP addresses to FQDNs and provides this information to the Cilium agent. The agent then updates the eBPF maps to allow traffic to the IPs associated with specific FQDNs.

Meanwhile, the Hubble server observes all traffic on the node. It also uses data from the DNS proxy to enrich the traffic observations, providing not just IP addresses but also their corresponding domain names. Hubble Relay aggregates traffic data from all the Hubble servers running across nodes, enabling a centralized view of network traffic within the cluster. Finally, the Hubble UI queries Hubble relay and visualizes the aggregated traffic data, offering an intuitive interface for exploring and analyzing network activity within the cluster.

With that, we've established a solid understanding of the various components, allowing us to explore each one in greater depth in the following chapters.

Chapter 2. Getting Started with Cilium

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo is available at *https://github.com/isovalent/cilium-up-and-running*.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *gobrien@oreilly.com*.

Now that you're familiar with the core components of Cilium, let's go ahead and deploy it. The best way to become familiar with Cilium is to deploy in a Kubernetes cluster and try out some of the core capabilities. In this section, you will learn how to:

- Install Cilium in a Kubernetes cluster
- Deploy an application,
- Access, secure and monitor the application.

In later chapters, we will dive further into various areas of Cilium, but a practical hands-on experience with Cilium will give you strong foundational knowledge for the rest of the book. To get started, you will first need access to a Kubernetes cluster. We'll create one locally.

NOTE

If you don't have local compute resources available to run a Kubernetes cluster, feel free to use the free online hands-on labs provided by Isovalent, the creators of Cilium. Head over to isovalent.com/labs to begin your evaluation of Cilium.

Setting Up Your Environment

Throughout this book, you will find code snippets and scripts to install, test and configure Cilium. This implies that you have access to a Kubernetes cluster. Do not try any of the scripts in a production cluster: they may cause some disruption or even worse, a complete cluster outage.

Cilium works with any distribution conformant with "vanilla" Kubernetes and is often even packaged and included by default with many Kubernetes distributions and services.

Not everyone has the financial or computing resources to afford a Kubernetes cluster. Therefore, the environment we will recommend for the vast majority of the exercises in the book is based on Kubernetes in Docker (kind), a popular tool used to run Kubernetes clusters locally in Docker containers.

We recommend kind over alternatives because:

- It works across multiple platforms (Linux, macOS and Windows)
- It is a CNCF-certified conformant Kubernetes installer
- It's lightweight compared to some of the alternatives.

That said, if you prefer another lightweight Kubernetes solution, check the documentation to make sure it's compatible with Cilium.

NOTE

During the creation of this book, we tested all scripts across a variety of Kubernetes environments, including:

- Minikube
- K3S
- Azure Kubernetes Services (Bring your own CNI)
- Google Kubernetes Engine
- Amazon Elastic Kubernetes Services

The majority of features work seamlessly but you can find some considerations and caveats in the book's GitHub repository.

Kubernetes-in-Docker (kind)

This chapter assumes you have Docker or another container runtime installed, as `kind` requires it to launch Kubernetes clusters. If you need help installing Docker, refer to the official documentation for your operating system.

If you don't have `kind` installed yet, let's do that now. For those of you on MacOS machines, you can simply use Homebrew to install it:

```
$ brew install kind
```

The instructions for other platforms can be found on the kind website.

Installing the Cilium CLI

To manage and inspect Cilium deployments, we will use the Cilium CLI. It should be available in your operating system's package manager; for example, you can run the following Homebrew command to install it on MacOs:

```
$ brew install cilium-cli
```

You can also download and install the appropriate tarball from the Cilium CLI GitHub repository (*https://github.com/cilium/cilium-cli/*). Detailed instructions can also be found on the Cilium Docs.

Once installed, verify it has been installed correctly:

```
$ cilium version --client
cilium-cli: v0.18.0 compiled with go1.24.0 on darwin/arm64
cilium image (default): v1.17.0
cilium image (stable): v1.17.3
```

Installing the Hubble CLI

To observe network traffic in the cluster, we will use Hubble and its binary client, the Hubble CLI. Likewise, you can also install it with a package manager like Homebrew:

```
$ brew install hubble
```

Alternatively, download and install the tarball from the Hubble GitHub repository (*https://github.com/cilium/hubble*). You can also find detailed installation instructions on the Cilium docs (*https://docs.cilium.io/en/stable/observability/hubble/setup/#hubble-cli-install*).

Once installed, verify the Hubble CLI has been installed correctly by running the following command.

```
$ hubble version
hubble v1.16.4@HEAD-4b765dc compiled with go1.23.3 on
darwin/arm64
```

Installing Helm

We will use the Kubernetes package manager Helm throughout the book. If you don't have it installed on your machine yet, install it via your package manager (e.g. `brew install helm`) or refer to the official Helm installation docs.

Deploying a Kind Cluster

The following YAML manifest, available in the book's GitHub repository, includes the specification of the cluster.

```
kind: Cluster                                  # Declares a
Kind cluster
apiVersion: kind.x-k8s.io/v1alpha4             # Kind API
version
nodes:
- role: control-plane                          # One control-
plane node
- role: worker                                 # Two worker
nodes
- role: worker                                 # Two worker
nodes
networking:
  disableDefaultCNI: true          # Disables kindnetd prior
to Cilium install
```

The most notable aspect of the configuration is the `networking.disableDefaultCNI: true` setting. It specifies that the cluster must be deployed without kind's builtin CNI (kindnetd).

As you recall from Chapter 1, a CNI plugin is required in order for our cluster to host and connect applications. We intentionally disable kind's default built-in CNI so that we can install Cilium instead in the next section.

NOTE

You can find all the YAML manifests you will use in this chapter to deploy and test Cilium in the chapter03 directory of the book's GitHub repository *https://github.com/isovalent/cilium-up-and-running* . Navigate to this folder before proceeding.

Let's now deploy the kind cluster, with the following command:

```
$ kind create cluster --config kind-cluster-config.yaml
Creating cluster "kind" ...
 ✓ Ensuring node image (kindest/node:v1.32.0) ؼ
Preparing nodes F桟 F桟
 ✓ Writing configuration F젊 ✓ Starting control-plane b鰊
 Installing StorageClass ᖁ  Joining worker nodes ㅂ젊Set
kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Not sure what to do next? ㎝堠Check out
https://kind.sigs.k8s.io/docs/user/quick-start/
```

With the cluster now deployed, let's use `kubectl` to check our cluster.

```
$ kubectl get nodes
NAME                 STATUS     ROLES           AGE
VERSION
kind-control-plane   NotReady   control-plane   46m
v1.32.0
kind-worker          NotReady   <none>          46m
v1.32.0
kind-worker2         NotReady   <none>          46m
v1.32.0
```

The preceding terminal output shows that the four nodes (one control-plane and three workers) have been deployed but they are

in `NotReady` status. This indicates the nodes are unhealthy and will not accept workloads. You may have already guessed the reason but let's confirm why by inspecting the status of one of our worker nodes.

```
$ kubectl get nodes kind-worker -o yaml
[...]
- lastHeartbeatTime: "2025-02-21T18:34:29Z"
  lastTransitionTime: "2025-02-21T17:47:48Z"
  message: 'container runtime network not ready:
NetworkReady=false reason:NetworkPluginNotReady
message:Network plugin returns error: cni plugin not
initialized'
  reason: KubeletNotReady
  status: "False"
  type: Ready
```

As you can see, since no network plugin has been deployed, the Kubelet reports it is not ready to run pods. We'll resolve that by installing the Cilium CNI plugin in the next section.

Installing Cilium

Cilium supports multiple installation methods. The two most common are:

- Using the **Cilium CLI** (`cilium install`) for quick setup and environment auto-detection
- Using **Helm** directly, which provides explicit control and aligns better with GitOps and long-term operations

Let's review both options.

Installing Cilium Using the Cilium CLI

The Cilium CLI simplifies the initial experience by inspecting your current Kubernetes context and generating a set of recommended Helm values based on your environment.

To preview the configuration without performing the installation:

```
$ cilium install --dry-run-helm-values
```

On your kind cluster, this should return:

```
cluster:
  name: kind-kind
ipam:
  mode: kubernetes
operator:
  replicas: 1
routingMode: tunnel
tunnelProtocol: vxlan
```

In this configuration:

- The Cilium operator will be deployed with a single replica
- IPAM mode is set to Kubernetes-host Scope mode
- Routing mode is set to encapsulation, with VXLAN tunnels used to encapsulate traffic between nodes

You will explore IPAM and routing modes in detail in Chapters 4 and 5. For now, note that:

- Cilium will form VXLAN tunnels between all nodes to enable cross-node pod communication
- It will allocate pod IPs from the subnets assigned to each node by Kubernetes

To verify the pod CIDRs Kubernetes has assigned:

```
$ kubectl get nodes kind-worker -o yaml | yq .spec.podCIDR
10.244.1.0/24

$ kubectl get nodes kind-worker2 -o yaml | yq .spec.podCIDR
10.244.2.0/24
```

When you deploy the sample app shortly, you will observe that Cilium will assign IPs from these ranges.

Running `cilium install` now would install Cilium using the auto-generated settings:

```
$ cilium install
🔮 Auto-detected Kubernetes kind: kind
ℹ️  Using Cilium version 1.17.4
🔮 Auto-detected cluster name: kind-kind
🔮 Auto-detected kube-proxy has been installed
```

Installing Cilium Using Helm (Recommended)

Helm is already widely used to manage other Kubernetes applications, such as Prometheus, Grafana, cert-manager, etc.. Using it for Cilium as well ensures a consistent deployment workflow and makes it easier to integrate with CI/CD and GitOps practices.

To begin, save the auto-generated values into `helm-values.yaml` file:

```
$ cilium install --dry-run-helm-values >> helm-values.yaml
```

Then install Cilium with Helm:

```
$ helm install cilium cilium/cilium -n kube-system --values
helm-values.yaml
```

Sample output:

```
NAME: cilium
LAST DEPLOYED: Fri Mar 21 09:39:20 2025
NAMESPACE: kube-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
You have successfully installed Cilium with Hubble.

Your release version is 1.17.1.

For any further help, visit
https://docs.cilium.io/en/v1.17/gettinghelp
```

Throughout the rest of the book, you'll find Helm values files in the folder corresponding to each chapter. Use them with the Helm install command shown earlier to deploy Cilium with the appropriate settings.

NOTE

Instead of using a values file, you can pass values inline:

```
$ helm install cilium cilium/cilium -n kube-system \
  --set cluster.name=kind-kind \
  --set ipam.mode=kubernetes \
  --set operator.replicas=1 \
  --set routingMode=tunnel \
  --set tunnelProtocol=vxlan
```

The `cilium install` command also accepts these Helm-style `--set` flags for customization:

```
$ cilium install \
  --set cluster.name=kind-kind \
  --set ipam.mode=kubernetes \
  --set operator.replicas=1 \
  --set routingMode=tunnel \
  --set tunnelProtocol=vxlan
```

By default, both `cilium install` and Helm will install the latest stable version of Cilium. However, you can explicitly specify a version to ensure consistency across environments or to avoid unexpected changes. This is especially useful in production, where you may want to pin versions and manage upgrades intentionally.

To install a specific version, use the following flag:

```
$ helm install cilium cilium/cilium -n kube-system --
version 1.17.4 --values helm-values.yaml
```

After a couple of minutes, Cilium should be successfully installed. Verify the installation with the `cilium status` command.

```
$ cilium status
    /¯¯\
```

```
 /¯¯\__/¯¯\    Cilium:             OK
 \__/¯¯\__/    Operator:           OK
 /¯¯\__/¯¯\    Envoy DaemonSet:    OK
 \__/¯¯\__/    Hubble Relay:       disabled
    \__/       ClusterMesh:        disabled

DaemonSet              cilium                   Desired: 3,
Ready: 3/3, Available: 3/3
DaemonSet              cilium-envoy             Desired: 3,
Ready: 3/3, Available: 3/3
Deployment             cilium-operator          Desired: 1,
Ready: 1/1, Available: 1/1
Containers:            cilium                   Running: 3
                       cilium-envoy             Running: 3
                       cilium-operator          Running: 1
                       clustermesh-apiserver
                       hubble-relay
Cluster Pods:          3/3 managed by Cilium
[...]
```

TIP

Note that you can watch the progress of the installation by typing `cilium status –wait` and observe the progression of the installation.

In the output you can see how the components we describe in Chapter 2 are deployed. As the Cilium Agent and Cilium Envoy components are both deployed through a DaemonSet, there is an instance of each deployed on every cluster node. As you might have seen in the earlier `cilium install --dry-run-helm-values`, a single Cilium Operator was deployed as part of the deployment.

You can verify with the following `kubectl` commands:

```
$ kubectl get -n kube-system daemonset/cilium
NAME     DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE
NODE SELECTOR            AGE
cilium   3         3         3       3            3
```

```
kubernetes.io/os=linux   41h
$ kubectl get -n kube-system daemonset/cilium-envoy
NAME           DESIRED   CURRENT   READY   UP-TO-DATE
AVAILABLE   NODE SELECTOR            AGE
cilium-envoy   3         3         3       3            3
kubernetes.io/os=linux   41h
$ kubectl get -n kube-system deployment/cilium-operator
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
cilium-operator   1/1     1            1           41h
```

Deploying a Sample Application

Let's now deploy a sample application. The following example is a common first sample application deployed in Kubernetes: the lightweight `NGINX` web server app. We'll use the *nginx-deployment.yaml* manifest file from the book's GitHub repository:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector:
        kubernetes.io/hostname: kind-worker2
      containers:
        - name: nginx
          image: nginx
```

"Pinning" the nginx-server pod to a specific worker node (`kind-worker2`) isn't a best practice in production but it's useful here to

illustrate inter-node traffic, which Cilium handles through VXLAN tunneling in the selected configuration mode.

Let's deploy the manifest:

```
$ kubectl apply -f nginx-deployment.yaml
deployment.apps/nginx-deployment created
```

Next, let's verify that the pod was scheduled on the expected node and that it was assigned an IP address by running the command:

```
$ kubectl get pods -o wide
NAME                                READY   STATUS
RESTARTS   AGE    IP             NODE
nginx-deployment-979f5455f-tjd2w   1/1     Running   0
7s     10.244.2.127   kind-worker2
```

Let's now deploy a pod on a different worker node (kind-worker)using the following manifest. We are using a networking utility called [netshoot](), which we'll use throughout the book because it includes many helpful networking tools. Given that it's a debugging utility, it's meant to run once and exit unless we specify to keep it running (which is what we will do, by instructing it to run `sleep infinity`).

```
apiVersion: v1
kind: Pod
metadata:
  name: netshoot-client
  labels:
    app: netshoot-client
spec:
  nodeSelector:
    kubernetes.io/hostname: kind-worker
  containers:
    - name: netshoot
      image: nicolaka/netshoot
      command: ["sleep", "infinity"]
```

Let's deploy it and verify it is scheduled on the kind-worker node:

```
$ kubectl apply -f netshoot-client-pod.yaml
pod/netshoot-client created
```

Next, verify that the pod was scheduled and deployed:

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

The output should tell you the pod was deployed to the kind-worker node and that Cillium assigned it an IP address (10.244.1.67) from the kind-worker PodCIDR.

Verify you can connect from netshoot-client (located on kind-worker) to the nginx-server (located on kind-worker2).

```
$ kubectl exec -t pod/netshoot-client -- curl -s
http://10.244.3.167
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
[OUTPUT TRUNCATED]
```

To verify HTTP connectivity, we really only need the HTTP status code. In our case, we expect a `200` to show a successful access. The following curl command will only output the HTTP status code.

```
$ kubectl exec -t pod/netshoot-client -- \
    curl -s -o /dev/null \
    -w "%{http_code}\n" \
    http://10.244.3.167
200
```

We've successfully deployed two pods across different nodes in the cluster, confirmed IP address assignments, and verified successful

inter-node connectivity, using Cilium's VXLAN-based tunneling. You will dive deeper into VXLAN and the datapath in Chapter 5.

While a successful connection from a client to the IP of a destination pod is a good first step, you might already know it's not necessarily the preferred approach in Kubernetes. Kubernetes provides the Service abstraction as a deterministic method to accessing applications.

Upon creation, Kubernetes will assign a Service a virtual IP address. When a client connects to the Service's virtual IP, traffic will be forwarded to one of the Service's endpoints.

> **WARNING**
>
> Kubernetes uses Endpoints and EndpointSlices to track the backends of a Service. These objects are used for load-balancing decisions for each Service. EndpointSlices, in particular, offer a more scalable way to track backends across large clusters. By contrast, Cilium uses CiliumEndpoints (CEPs) and CiliumEndpointSlices (CESs) to support network routing and policy enforcement.

To better understand how Kubernetes Services work, let's create one using the following manifest (*nginx-service.yaml*). This Service listens on TCP port 80 and load-balances traffic randomly towards pods labeled `app:nginx`.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
```

```
    targetPort: 80
  type: ClusterIP
```

Let's deploy it:

```
$ kubectl apply -f nginx-service.yaml
service/nginx-service created
```

Now that you have deployed the service, let's inspect it:

```
$ kubectl get svc nginx-service -o yaml
apiVersion: v1
kind: Service
metadata:
[...]
  name: nginx-service
  namespace: default
spec:
  clusterIP: 10.96.242.74
  clusterIPs:
  - 10.96.242.74
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  sessionAffinity: None
  type: ClusterIP
```

The preceding output highlights a few key aspects:

- The Service type is ClusterIP, which means it's only accessible from within the cluster

- It was assigned a cluster-internal IP (10.96.242.74) in the 10.96.0.0/16 CIDR range (the default IPv4 Service Subnet in kind)

You can confirm that the Service is working by sending a HTTP request to its ClusterIP:

```
$ kubectl exec netshoot-client -- curl http://10.96.242.74
<h1>Welcome to nginx!</h1>
```

> **NOTE**
>
> ClusterIP is one of several Kubernetes Services, alongside NodePort and LoadBalancer. Chapter 5 will provide a refresher on Service types. While ClusterIP is used for internal communication, most of your applications will need to be accessible from outside the cluster. Chapter 7 will introduce Ingress and Gateway API for handling external traffic, and Chapter 10 will explain the networking options available to expose your applications.

Although the client sends traffic to the Service IP, the actual response comes from one of the backend Pods created by the deployment. Kubernetes uses the EndpointSlice API to track which pods back a given Service. Each EndpointSlice lists the IPs of the pods that match the Service's label selector.

```
$ kubectl get endpointslices.discovery.k8s.io nginx-
service-66q4l
NAME                  ADDRESSTYPE   PORTS   ENDPOINTS
AGE
nginx-service-66q4l   IPv4          80      10.244.1.189
2m13s
```

If you scale the Deployment, Kubernetes automatically updates the endpoints.

```
$ kubectl scale deployment nginx-deployment --replicas=2
deployment.apps/nginx-deployment scaled
$ kubectl get pods
NAME                                READY   STATUS
RESTARTS   AGE
netshoot-client                     1/1     Running   0
7m15s
nginx-deployment-979f5455f-9pm7d    1/1     Running   0
4m29s
nginx-deployment-979f5455f-xw8nz    1/1     Running   0
13s
$ kubectl get endpointslices.discovery.k8s.io nginx-
service-66q4l
NAME                  ADDRESSTYPE   PORTS   ENDPOINTS
AGE
nginx-service-66q4l   IPv4          80
10.244.1.189,10.244.1.89   14m
```

After scaling, you'll see both pods running:

```
$ kubectl get pods
NAME                                READY   STATUS
RESTARTS   AGE
netshoot-client                     1/1     Running   0
7m15s
nginx-deployment-979f5455f-9pm7d    1/1     Running   0
4m29s
nginx-deployment-979f5455f-xw8nz    1/1     Running   0
13s
$ kubectl get endpointslices.discovery.k8s.io nginx-
service-66q4l
NAME                  ADDRESSTYPE   PORTS   ENDPOINTS
AGE
nginx-service-66q4l   IPv4          80
10.244.1.189,10.244.1.89   14m
```

And the EndpointSlices object now reflects both IPs:

```
$ kubectl get endpointslices.discovery.k8s.io nginx-
service-66q4l
NAME                  ADDRESSTYPE   PORTS   ENDPOINTS
```

```
AGE
nginx-service-66q4l   IPv4          80
10.244.1.189,10.244.1.89   14m
```

Behind the scenes, these endpoint IPs are used by Kubernetes to direct traffic to Pods. By default, clusters like kind use kube-proxy, a DaemonSet that runs on every node and programs iptables (with IPVS and nftables as newer alternatives) rules to implement Service routing.

You can confirm that kube-proxy is running:

```
$ kubectl get -n kube-system daemonsets/kube-proxy
NAME         DESIRED   CURRENT   READY   UP-TO-DATE
AVAILABLE   NODE SELECTOR            AGE
kube-proxy   3         3         3       3            3
kubernetes.io/os=linux   42m
```

In Chapter 6, you'll learn how Cilium replaces kube-proxy entirely using eBPF, eliminating the need for iptables, IPVS, or nftables altogether.

Another benefit of leveraging a Service is the automatic service DNS naming generation.

You can verify successful DNS resolution by accessing the nginx-service name from the client:

```
$ kubectl exec netshoot-client -- curl http://nginx-
service.default.svc.cluster.local
<h1>Welcome to nginx!</h1>
```

Given that the client and the nginx servers reside in the same namespace, you only need the service name:

```
$ kubectl exec netshoot-client -- curl http://nginx-service
<h1>Welcome to nginx!</h1>
```

> **NOTE**
>
> Cilium is not responsible for DNS activities, however, as we saw briefly in Chapter 2, it sometimes leverages a component named DNS Proxy when users define domain-based network policies. You will learn more about it in Chapter 12.

Securing the Application with Cilium Network Policies

With the application deployed and reachable, let's apply a network policy to restrict access-allowing only trusted systems to access it.

We will create a network policy that only allows access from the `netshoot-client` to the `nginx-server` pod, blocking the traffic from unauthorized systems.

First, to help us demonstrate network policies, we'll create another client pod with the following manifest (`unauthorized-client.yaml`).

```
apiVersion: v1
kind: Pod
metadata:
  name: unauthorized-client
spec:
  containers:
    - name: netshoot
      image: nicolaka/netshoot
      command: ["sleep", "infinity"]
```

Deploy it and verify it has access to the `nginx-server` prior to deploying network policies.

```
$ kubectl apply -f unauthorized-client.yaml
pod/unauthorized-client created
```

```
$ kubectl exec unauthorized-client -- curl http://nginx-
service
<h1>Welcome to nginx!</h1>
```

You might already be familiar with Kubernetes Network Policies. Cilium Network Policies (CNP) build upon them and provide more granular controls. Chapter 12 will cover the concepts of identity, endpoints, labels and how to construct CNPs. For now just know that the approach to define CNPs is through the use of Kubernetes labels.

Now, take a look at the following network policy (`policy.yaml`).

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: ch03-policy
  namespace: default
spec:
  endpointSelector:
    matchLabels:
      app: nginx
  ingress:
    - fromEndpoints:
        - matchLabels:
            app: netshoot-client
      toPorts:
        - ports:
            - port: "80"
```

We'll cover the anatomy of a Cilium Network Policy in detail in Chapter 12 but, let's do a quick review:

- `endpointSelector` selects the pods the policy applies to (e.g., `app:nginx`).
- `ingress` defines in which traffic direction the policy applies to. In this instance, we are defining a rule that will authorize

traffic *to* the pods with the `app:nginx` label (egress would instead apply to traffic leaving the pods).

- `fromEndpoints` determines which source pods are allowed to connect (those with the `app:netshoot-client` label)
- `toPorts` specifies which TCP/UDP ports are permitted.

By default, all egress and ingress traffic is allowed for all endpoints. When an endpoint is selected by a network policy, it transitions to a default-deny state, where only explicitly allowed traffic is permitted.

Given that our rule has an ingress section, the endpoint goes into default deny-mode for ingress.

In other words, when traffic enters endpoints with the `app:nginx` labels, only traffic from clients with the `app:netshoot-client` to the port 80 will be allowed. Any other traffic to these particular endpoints will be denied.

NOTE

Isovalent offers a free SaaS visualization Network Policy Editor tool on *https://editor.cilium.io* to help engineers create and manage network policies. You can even upload policies like the one above to see how they would take effect (see FIGURE 3-1).

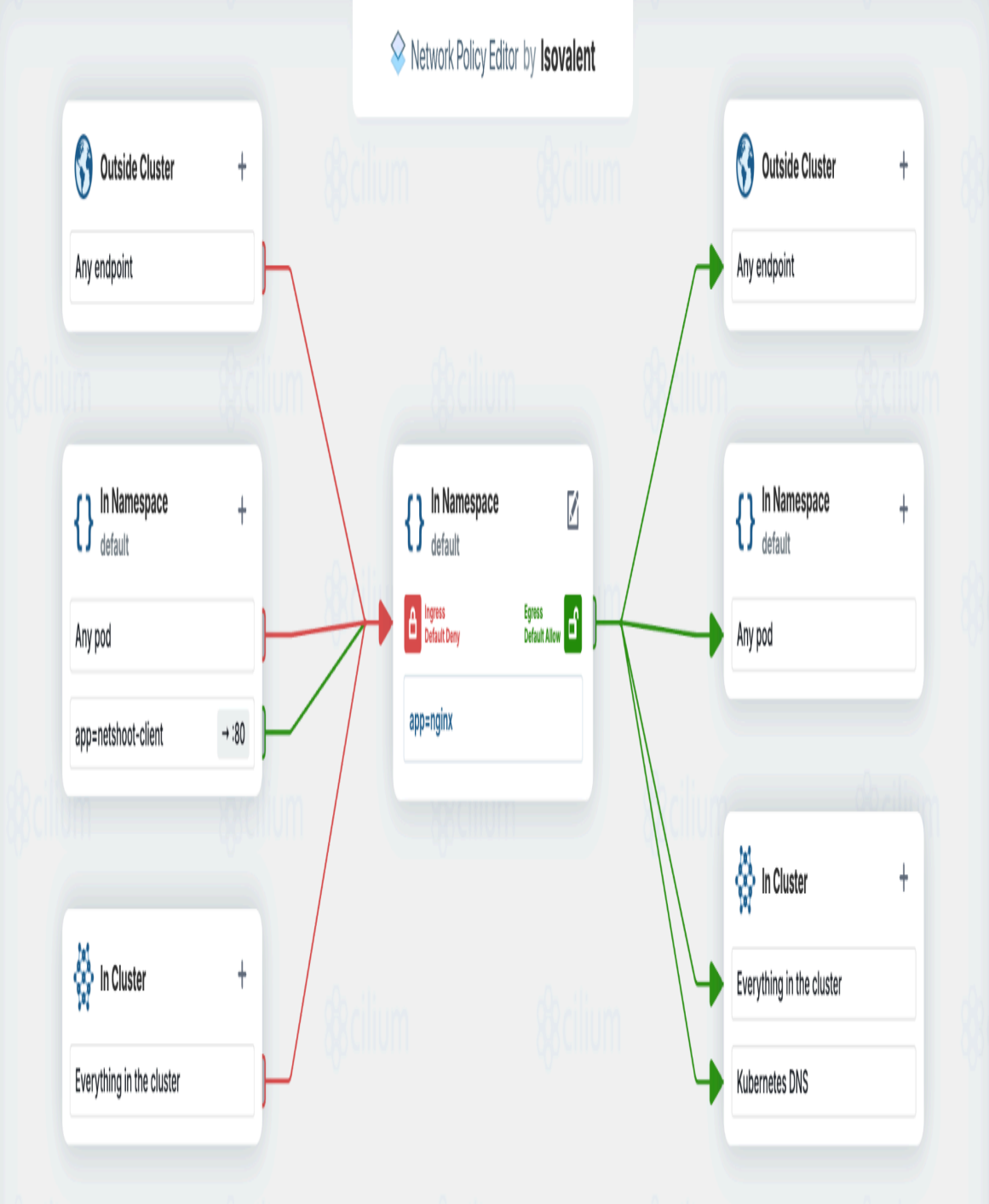


Figure 2-1. Network Policy Editor

Let's check that our `netshoot-client` has the right label and that `unauthorized-client` has not.

```
$ kubectl get pods --show-labels
NAME                     READY   STATUS    RESTARTS   AGE
LABELS
netshoot-client          1/1     Running   0          2d20h
app=netshoot-client
nginx-server             1/1     Running   0          3d3h
app=nginx
unauthorized-client      1/1     Running   0          80m
<none>
```

You can deploy the policy with the following command.

```
$ kubectl apply -f policy.yaml
ciliumnetworkpolicy.cilium.io/policy created
```

Verify that the unauthorized-client pod can no longer access the nginx server, while the netshoot-client still can.

```
$ kubectl exec unauthorized-client -- curl --max-time 3 -s
http://nginx-service
command terminated with exit code 28
$ kubectl exec netshoot-client -- curl --max-time 3 -s
http://nginx-service
<h1>Welcome to nginx!</h1>
```

Unlike standard Kubernetes network policies, Cilium Network Policies can also apply at a greater depth, up to layer 7. It is particularly useful to secure API calls, such as REST API or gRPC. With layer 7 network policies, you can restrict access based on HTTP methods, headers, and paths. You can verify now by augmenting the previous network policy and limit the HTTP call to only the `/index.html` file.

Traffic matching layer 7 rules is directed to Envoy, which performs HTTP inspection before forwarding or denying the request. We introduced Envoy in Chapter 2 and you will learn more about how traffic is re-directed to Envoy in Chapter 15 (Life of a Packet).

By default, the NGINX server we deployed also includes an error page named 50x.html. The current Cilium Network Policy lets you access it, with no restrictions.

```
$ kubectl exec -t netshoot-client -- curl http://nginx-
service/50x.html
<html>
<head>
<title>Error</title>
</head>
<body>
<h1>An error occurred.</h1>
<p>Sorry, the page you are looking for is currently
unavailable.<br/>
Please try again later.</p>
<p>If you are the system administrator of this resource
then you should check
the error log for details.</p>
<p><em>Faithfully yours, nginx.</em></p>
</body>
```

You can refine the previous policy by only allowing access to the *index.html* page by adding layer 7 rules such as the following:

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: policy
  namespace: default
spec:
  endpointSelector:
    matchLabels:
      app: nginx
  ingress:
    - fromEndpoints:
        - matchLabels:
            app: netshoot-client
      toPorts:
        - ports:
            - port: "80"
              protocol: TCP
```

```
rules: <1>
  http: <1>
  - method: "GET" <1>
    path: "/index.html" <1>
```

<1>This ensures that, in addition to the previous rules and conditions we specified earlier, only access with the HTTP GET Method to the *index.html* page is permitted.

Deploy the updated network policy (it will overwrite the one previously configured):

```
$ kubectl apply -f policy-with-l7.yaml
ciliumnetworkpolicy.cilium.io/policy configured
```

Verify access to /50x.html and */index.html*. The former is now blocked while the latter is still allowed.

```
$ kubectl exec netshoot-client -- curl -s http://nginx-
service/50x.html
Access denied
$ kubectl exec netshoot-client -- curl http://nginx-
service/index.html
<title>Welcome to nginx!</title>
```

You might have noticed that this time, you received an "Access Denied" error, rather than a timeout as you saw earlier when you used a Layer 3/4 policy. While enforced L3/L4 rules lead to dropped packets, enforced layer 7 rules applied by Envoy return deny codes for the application protocol. In this case, we're parsing HTTP traffic so it returns `403` for a denied request.

Now that we have set up a security policy to control who can access our applications, let's observe its effect on actual flows.

Monitoring the Application with Cilium Hubble

Hubble is the observability subsystem of Cilium. You cannot use Hubble without Cilium because it leverages the data collected through eBPF to provide deep visibility into network traffic and policy enforcement.

By default, Hubble runs on the individual node on which the Cilium agent runs. This confines the network insights to the traffic observed by the local Cilium agent.

To enable Hubble to start collecting and aggregating flow logs across the entire cluster, use the following command.

```
$ helm upgrade cilium cilium/cilium --version 1.17.1 \
   --namespace kube-system \
   --reuse-values \
   --set hubble.relay.enabled=true
```

NOTE

Alternatively, you can use the Cilium CLI command `cilium hubble enable` to activate the Hubble function instead.

Once enabled, verify that the Hubble Relay status is `OK` with `cilium status`:

```
$ cilium status
    /¯¯\
 /¯¯\__/¯¯\    Cilium:             OK
 \__/¯¯\__/    Operator:           OK
 /¯¯\__/¯¯\    Envoy DaemonSet:    OK
 \__/¯¯\__/    Hubble Relay:       OK
    \__/       ClusterMesh:        disabled

DaemonSet              cilium             Desired: 3,
Ready: 3/3, Available: 3/3
DaemonSet              cilium-envoy       Desired: 3,
```

```
Ready: 3/3, Available: 3/3
Deployment             cilium-operator    Desired: 1,
Ready: 1/1, Available: 1/1
Deployment             hubble-relay       Desired: 1,
Ready: 1/1, Available: 1/1
Containers:            cilium             Running: 3
                       cilium-envoy       Running: 3
                       cilium-operator    Running: 1
                       hubble-relay       Running: 1
[...]
```

To access the logs collected from Hubble, you can use the port-forward feature of Kubernetes. With the following command, you can connect the Hubble client to the local port `4245` and access the Hubble Relay service in your Kubernetes cluster. The `&` lets you run Hubble in the background rather than having to keep the terminal open.

```
$ cilium hubble port-forward&
[1] 71325
ℹ Hubble Relay is available at 127.0.0.1:4245
$
```

NOTE

The equivalent kubectl command would be `$ kubectl -n kube-system port-forward svc/hubble-relay 4245:80 &`

```
$ kubectl -n kube-system port-forward svc/hubble-relay
4245:80 &
```

You can now connect to the Hubble Server, using the Hubble UI or CLI. To start with, let's use the Hubble CLI.

First, verify the status of Hubble using `hubble status`. It provides a real-time snapshot of Hubble's health, including metrics such as

the rate of flows observed per second and the volume of flows currently stored locally in its ring buffer.

```
$ hubble status
Healthcheck (via localhost:4245): Ok
Current/Max Flows: 16,380/16,380 (100.00%)
Flows/s: 9.43
Connected Nodes: 3/3
```

And list all the nodes that Hubble Relay is connected to, with the following command:

```
$ hubble list nodes
NAME                              STATUS      AGE
FLOWS/S   CURRENT/MAX-FLOWS
kind-kind/kind-control-plane   Connected   29h35m53s   1.05
4095/4095 (100.00%)
kind-kind/kind-worker          Connected   29h35m53s   1.06
4095/4095 (100.00%)
kind-kind/kind-worker2         Connected   29h35m53s   4.32
4095/4095 (100.00%)
```

Next, use `hubble observe` to list all the flows collected by Hubble.

```
$ hubble observe
Mar 14 11:22:41.654: default/netshoot-client:50976
(ID:18901) -> kube-system/coredns-668d6bf9bc-qdkn6:53
(ID:23236) to-overlay FORWARDED (UDP)
Mar 14 11:22:41.655: default/netshoot-client:50976
(ID:18901) <- kube-system/coredns-668d6bf9bc-qdkn6:53
(ID:23236) to-endpoint FORWARDED (UDP)

Mar 14 11:22:41.656: default/netshoot-client:36998
(ID:18901) -> default/nginx-deployment-979f5455f-xw8nz:80
(ID:4437) policy-verdict:L3-L4 INGRESS ALLOWED (TCP Flags:
SYN)
Mar 14 11:22:41.657: default/netshoot-client:36998
(ID:18901) -> default/nginx-deployment-979f5455f-xw8nz:80
(ID:4437) http-request FORWARDED (HTTP/1.1 GET
```

```
http://nginx-service/index.html)
Mar 14 11:22:41.658: default/netshoot-client:36998
(ID:18901) <- default/nginx-deployment-979f5455f-xw8nz:80
(ID:4437) http-response FORWARDED (HTTP/1.1 200 1ms (GET
http://nginx-service/index.html))
```

Like `tcpdump`, the Hubble CLI can be extremely verbose!

The output shows traffic from the netshoot client pod, including a DNS query to CoreDNS (UDP/53) in a different namespace, followed by a successful HTTP request to the NGINX service:

You can see, with the arrow direction (`<-` and `->`), the flow of the traffic, including which namespace (`default` and `kube-system`) it originated from and where it was sent. The logs also show the identity of each pod and whether traffic was forwarded, dropped or denied by a network policy.

The first two lines show a DNS query to CoreDNS. Notice the `to-overlay` flag indicating the path the packet has followed: as the DNS server was located on a different node than the client, the traffic was sent via the VXLAN tunnel.

> **NOTE**
>
> Incidentally, sending DNS traffic outside of the node is inefficient as it adds latency and creates unnecessary inter-node traffic. In Chapter 8 you will learn how Cilium can optimize DNS by intercepting these requests locally using a feature called Local Redirect Policy.

To control the output of Hubble, let's use filters to narrow down the output. Let's only list the flows originating from netshoot-client.

```
$ hubble observe --from-pod netshoot-client
Jul  2 10:39:11.325: default/netshoot-client:42214
(ID:7613) -> default/nginx-deployment-979f5455f-qcbjh:80
(ID:42054) to-overlay FORWARDED (TCP Flags: SYN)
```

```
Jul  2 10:39:11.325: default/netshoot-client:42214
(ID:7613) -> default/nginx-deployment-979f5455f-qcbjh:80
(ID:42054) policy-verdict:L3-L4 INGRESS ALLOWED (TCP Flags:
SYN)
Jul  2 10:39:11.325: default/netshoot-client:42214
(ID:7613) -> default/nginx-deployment-979f5455f-qcbjh:80
(ID:42054) to-proxy FORWARDED (TCP Flags: SYN)
Jul  2 10:39:11.325: default/netshoot-client:42214
(ID:7613) -> default/nginx-deployment-979f5455f-qcbjh:80
(ID:42054) to-overlay FORWARDED (TCP Flags: ACK)
Jul  2 10:39:11.325: default/netshoot-client:42214
(ID:7613) -> default/nginx-deployment-979f5455f-qcbjh:80
(ID:42054) to-proxy FORWARDED (TCP Flags: ACK)
Jul  2 10:39:11.326: default/netshoot-client:42214
(ID:7613) -> default/nginx-deployment-979f5455f-qcbjh:80
(ID:42054) to-overlay FORWARDED (TCP Flags: ACK, PSH)
Jul  2 10:39:11.326: default/netshoot-client:42214
(ID:7613) -> default/nginx-deployment-979f5455f-qcbjh:80
(ID:42054) to-proxy FORWARDED (TCP Flags: ACK, PSH)
Jul  2 10:39:11.326: default/netshoot-client:42214
(ID:7613) -> default/nginx-deployment-979f5455f-qcbjh:80
(ID:42054) http-request FORWARDED (HTTP/1.1 GET
http://nginx-service/index.html)
Jul  2 10:39:11.327: default/netshoot-client:42214
(ID:7613) -> default/nginx-deployment-979f5455f-qcbjh:80
(ID:42054) to-overlay FORWARDED (TCP Flags: ACK, FIN)
Jul  2 10:39:11.327: default/netshoot-client:42214
(ID:7613) -> default/nginx-deployment-979f5455f-qcbjh:80
(ID:42054) to-proxy FORWARDED (TCP Flags: ACK, FIN)
```

The output was taken shortly after executing the `kubectl exec netshoot-client -- curl nginx-service/index.html` command. The `to-proxy` confirms that traffic was redirected to the Envoy proxy because of the layer 7 policy in place.

You can combine multiple filters together, for example, to only show traffic from netshoot-client to a HTTP path “/index.html”:

```
$ hubble observe --from-pod netshoot-client --http-path
"/index.html"
Jul  2 10:41:55.108: default/netshoot-client:59110
```

```
(ID:7613) -> default/nginx-deployment-979f5455f-qcbjh:80
(ID:42054) http-request FORWARDED (HTTP/1.1 GET
http://nginx-service/index.html)
```

Another useful Hubble CLI filter, `--label`, lets you select flows based on labels:

```
$ hubble observe --label app=nginx
[...]
Mar 14 11:28:32.002: 10.244.1.218:51932 (ID:18901) ->
default/nginx-deployment-979f5455f-9pm7d:80 (ID:4437) to-
endpoint FORWARDED (TCP Flags: ACK, FIN)
Mar 14 11:28:32.002: 10.244.1.218:51932 (host) <-
default/nginx-deployment-979f5455f-9pm7d:80 (ID:4437) to-
stack FORWARDED (TCP Flags: ACK, FIN)
```

You can also filter based on the network policy verdict and only check the flows where a network policy decision was made. You can find the full list of available filters with `hubble observe -help`.

```
$ hubble observe -t policy-verdict
Mar 21 11:18:51.917: default/netshoot-client:34898
(ID:11661) -> default/nginx-deployment-979f5455f-bnxh7:80
(ID:5834) policy-verdict:L3-L4 INGRESS ALLOWED (TCP Flags:
SYN)
Mar 21 12:12:27.525: default/netshoot-client-worker2:48768
(ID:11661) -> default/nginx-deployment-979f5455f-bnxh7:80
(ID:5834) policy-verdict:L3-L4 INGRESS ALLOWED (TCP Flags:
SYN)
Mar 21 12:15:40.016: default/unauthorized-client:41212
(ID:8087) <> default/nginx-deployment-979f5455f-bnxh7:80
(ID:5834) policy-verdict:none INGRESS DENIED (TCP Flags:
SYN)
```

Hubble CLI provides insight into what's happening in your cluster but for those of you who prefer a more visual representation of the network traffic, you can also leverage Hubble UI. In Chapter 2 we mentioned how the Hubble UI can also consume the flows aggregated from the Hubble Relay.

Use the following Helm commands to enable the Hubble UI:

```
$ helm upgrade cilium cilium/cilium --version 1.17.1 \
   --namespace kube-system \
   --reuse-values \
   --set hubble.relay.enabled=true \
   --set hubble.ui.enabled=true
```

NOTE

If you prefer to use the Cilium CLI to enable the UI (via `cilium hubble enable --ui`), you must first temporarily disable Hubble using `cilium hubble disable`. This is necessary because the Hubble UI cannot be enabled once Hubble is already running.

Once you enable Hubble UI, you will see a Hubble UI Deployment being created and a Hubble UI pod made of a backend container and a frontend UI.

```
$ kubectl -n kube-system get deployment/hubble-ui
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
hubble-ui   1/1     1            1           2m22s

$ kubectl get pod hubble-ui-76d4965bb6-m8s78 -n kube-system
-o yaml
 containerStatuses:
  - containerID:
[...]
    name: backend
  - containerID:
[...]
    name: frontend
```

To access the Hubble UI, we will once again use the Kubernetes port-forward functionality. Note that the Hubble UI can also be accessed over Ingress, as you will learn in Chapter 14.

```
$ cilium hubble ui
ℹ  Opening "http://localhost:12000" in your browser...
```

A browser should automatically open. If it doesn't, simply open *http://localhost:12000* in your browser. You will see the Hubble UI landing page (Figure 3-2):

Idle

Welcome!

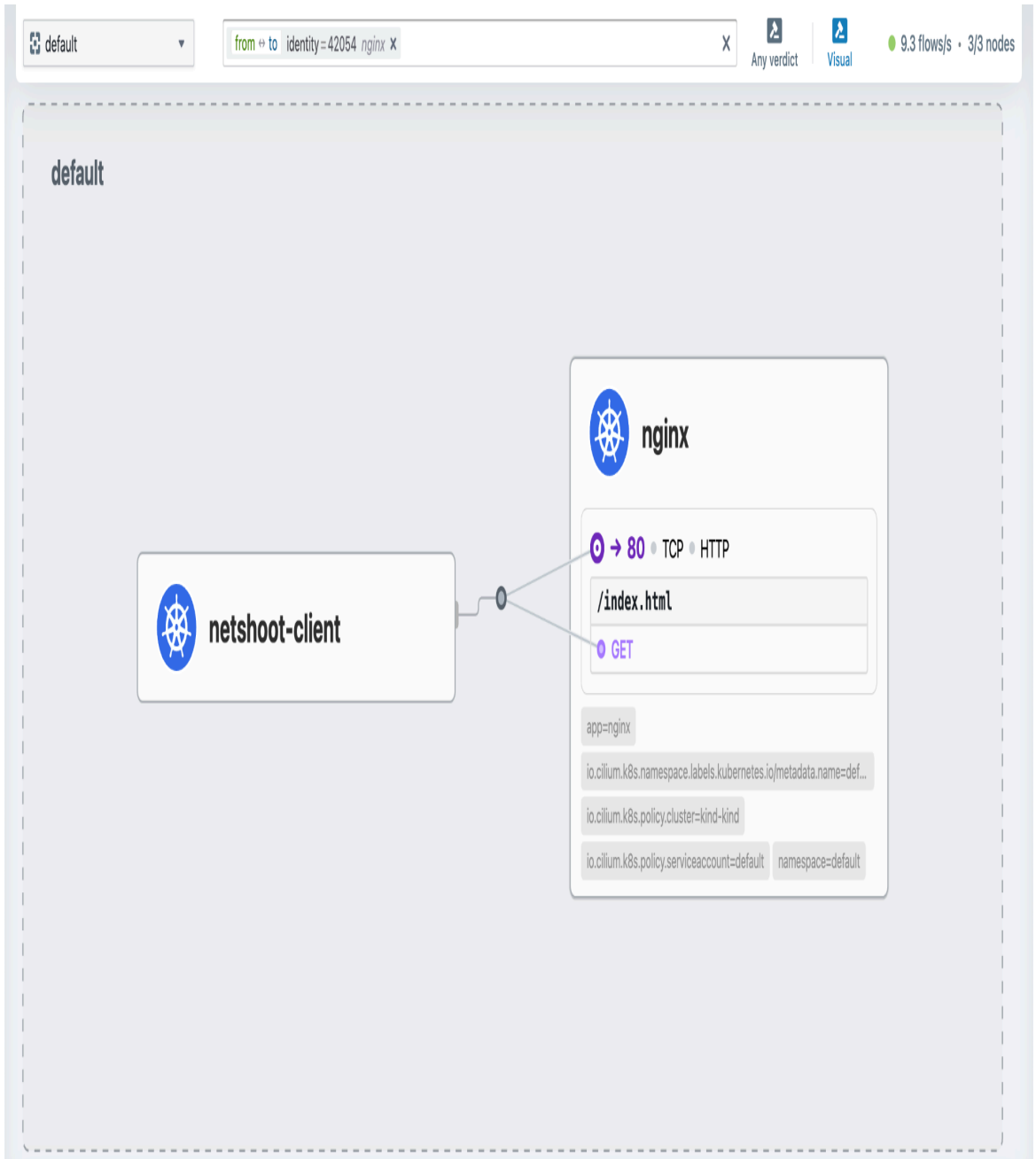
To begin select one of the namespaces:

Choose namespace ▾

Figure 2-2. Hubble UI Landing Page

Just like the Hubble CLI, the Hubble UI can display flows filtered by namespace. Select the default namespace, and you'll see the same flows you observed earlier - now shown as a visual service map that illustrates communication between services.

Click on any of the pods to see the equivalent representation to `hubble observe –pod`. For example, you can see the flows to and from `netshoot-client` in Figure 3-3.



Columns

Source Pod	Destination Pod	Destination Port	L7 info	Traffic Direction
–	nginx-deployment-979f5455f-nk4w5	80	–	egress
–	nginx-deployment-979f5455f-nk4w5	80	–	egress
netshoot-client	nginx-deployment-979f5455f-nk4w5	80	–	egress
netshoot-client	nginx-deployment-979f5455f-nk4w5	80	–	egress
–	nginx-deployment-979f5455f-nk4w5	80	–	egress
netshoot-client	nginx-deployment-979f5455f-nk4w5	80	→ GET /index.html 0ms	ingress
–	nginx-deployment-979f5455f-nk4w5	80	–	egress
–	nginx-deployment-979f5455f-nk4w5	80	–	egress

Figure 2-3. Hubble UI Service Map

You might also notice an `L7 Info` column in the user interface, populated with details such as HTTP path, method and latency. This is a side effect - and added benefit - of the Layer 7-based Cilium Network Policy we enforced earlier. Layer 7 rules not only provide more granular control - they also enhance observability. As traffic is intercepted by Envoy, it enriches the network flow information with application-layer context.

Chapter 14 will dive deeper into Hubble.

Conclusion

We hope this chapter has demonstrated that getting started with Cilium is straightforward. You've deployed Cilium in a Kubernetes cluster and taken your first steps toward using it to secure, observe, and manage application traffic.

In the next chapter, we'll take a closer look at how Cilium handles IP address management and establishes network connectivity for pods. You'll learn how to choose the right IPAM mode for your environment and understand the implications of using IPv4, IPv6, or both.

Chapter 3. IP Address Management

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo is available at *https://github.com/isovalent/cilium-up-and-running*.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *gobrien@oreilly.com*.

Before deploying Cilium in a Kubernetes cluster, platform engineers need to make two fundamental networking decisions: how pod IP addresses are allocated, and whether to use IPv4, IPv6, or both. Selecting the right IP Address Management (IPAM) mode is essential, as changing it later may disrupt connectivity for running workloads or may not be supported at all.

Broader organizational requirements typically drive the choice between IPv4, IPv6 or dual-stack, but it also demands careful consideration by platform teams during initial cluster design.

This chapter will help you make an informed decision. We will start by examining the different IPAM modes supported by Cilium, highlighting the pros and cons of each one. We will then explore the

dual-stack and IPv6 deployment models. Every aspect will come with sample manifests for you to try in your own test environments.

IPv4 and IPv6 Support in Cilium

Before we explain how IP addresses are assigned and how pods communicate with each other, we should start with a preamble on IPv6 support.

Kubernetes supports IPv4 single-stack, IPv6 single-stack, and dual-stack. However, this support depends on your chosen CNI to implement the required functionality.

IPv4/IPv6 single-stack networking is when each pod and service can be assigned either a single IPv4 or IPv6 address. With IPv4/IPv6 dual-stack networking, each pod and service can be assigned both an IPv4 and an IPv6 address. IPv4/IPv6 dual-stack on your Kubernetes cluster provides the following features:

- Dual-stack Pod networking (a single IPv4 and IPv6 address assignment per Pod)
- IPv4 and IPv6 enabled Services.
- Pod off-cluster egress routing (e.g., the Internet) via IPv4 and IPv6 interfaces.

When Cilium was first introduced, it supported only IPv6. This was a deliberate and forward-looking decision by the development team, who saw IPv6 as a natural fit for Kubernetes. Its expansive address space aligned well with the scale and dynamic nature of container workloads, where traditional IPv4 ranges can become limiting.

However, it soon became clear that most users - and the broader cloud-native ecosystem - were not yet prepared for IPv6-only clusters. Many tools, services, and operational practices were still built around IPv4. In response, the Cilium team added IPv4 support

early in the project's lifecycle, making it possible to run clusters in dual-stack or IPv4-only mode.

IPv6 has remained a core part of Cilium's design. In some cases, new features were even implemented with IPv6 support before IPv4. While most of the examples in this book use IPv4 (reflecting its continued prevalence in Kubernetes environments), we include IPv6 examples where relevant and note any important caveats or considerations along the way.

IP Address Management (IPAM)

Assigning an IP address to entities is one of Cilium's core responsibilities. In non-containerized environments, you would have relied on Dynamic Host Configuration Protocol (DHCP) or SLAAC to dynamically assign IP addresses to machines. In Kubernetes, we refer to the process of assigning IP addresses to pods as IP Address Management (IPAM).

The CNI plugin is often responsible for assigning an IP address to your pod, and Cilium offers several options, which we will discuss in this section.

Pod IP Address Allocation

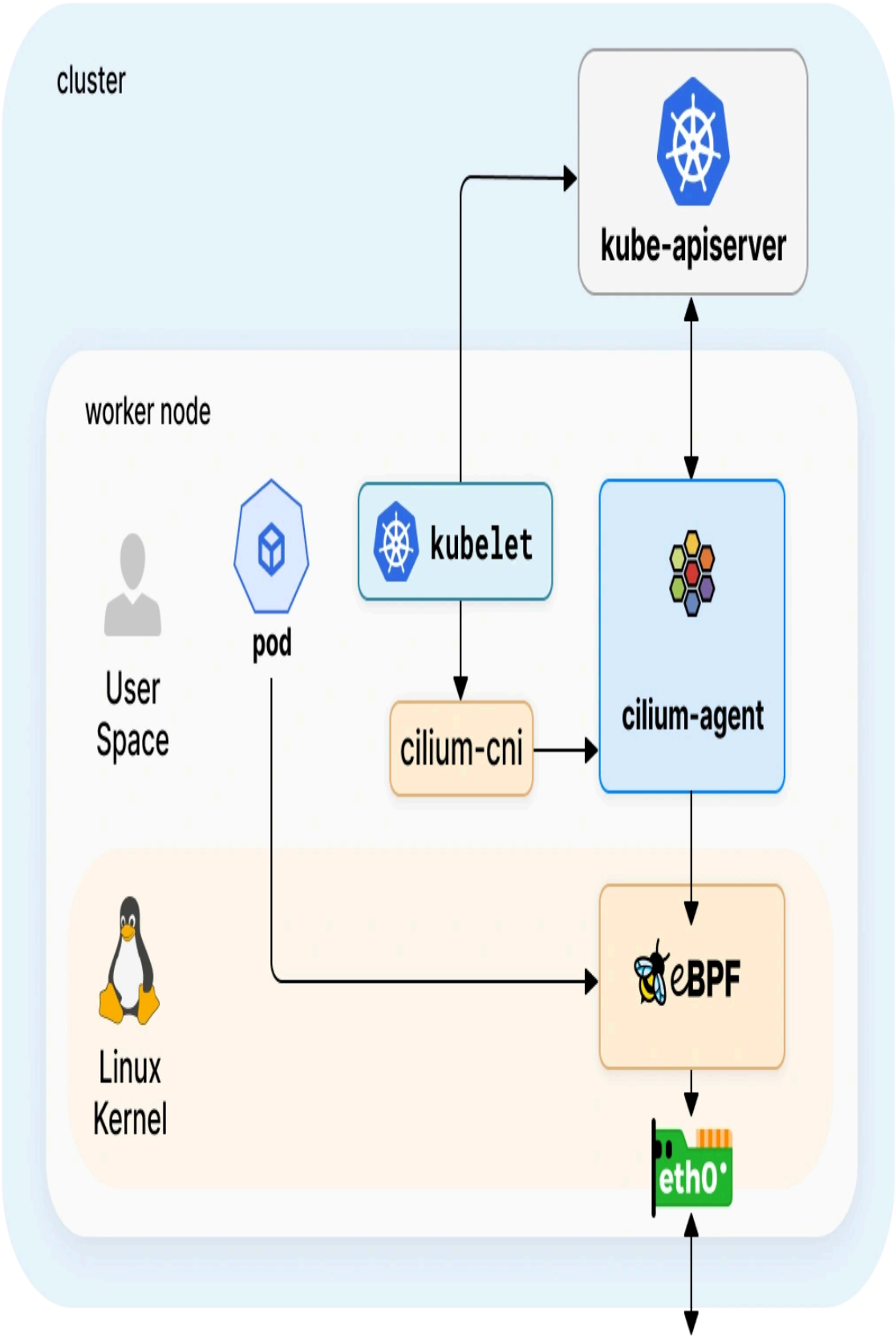

cluster
kube-apiserver
worker node
pod
kubelet
User
Space
cilium-cni
cilium-agent
eBPF
Linux
Kernel
eth0

Figure 3-1. Pod IP Address Allocation

Figure 4-1 illustrates the pod IP address allocation process. When a new pod is added in Kubernetes, the Kubernetes Scheduler assigns it to a node. The Kubelet running on that node will be notified and begin creating the pod's containers.

The Kubelet uses the CNI for networking. First, the Kubelet checks the CNI configuration on the node located in */etc/cni/net.d/*. When Cilium is installed on the cluster, it creates a configuration file in */etc/cni/net.d/05-cilium.conf* which instructs the Kubelet on how to configure pod networking.

Each pod receives an IP address from a subnet referred to as PodCIDR. CIDR (classless inter-domain routing) defines IP address blocks using subnet prefixes. In Kubernetes, each node is assigned one or more PodCIDRs, and pods on that node receive their IPs from those ranges via the IPAM process.

Cilium supports multiple IPAM modes. Some are specific to a cloud provider (such as AWS and GKE), while others can be used across any Kubernetes environment. We won't cover them all in this book, but we will cover the ones we see the most commonly used:

Kubernetes Host Scope

: Uses PodCIDRs assigned to each node by Kubernetes. Simple but inflexible.

Cluster Scope

: Cilium manages PodCIDR allocation via the operator. Default mode.

Multi-Pool

: Allows pods to be assigned IPs from multiple IP pools based on namespaces or annotations. Ideal for multi-tenant setups.

ENI IPAM

Exclusive to AWS EKS environments. Allocates pod IPs from AWS Elastic Network Interfaces.

> **NOTE**
>
> This chapter focuses on pod IP address management. Other components of Kubernetes require an IP address. Nodes typically receive theirs via DHCP or SLAAC. Kubernetes services receive theirs either from the Kubernetes API server or, in the case of LoadBalancer services, via a cloud provider or a dedicated system. We'll explore service networking in more detail in Chapter 6.

Kubernetes-host Scope IPAM Mode

We'll begin with Kubernetes Host Scope, the simplest IPAM mode to understand and deploy. In this mode, Kubernetes allocates a cluster-wide prefix and assigns each node a subnet from it. Cilium then assigns pod IPs from these subnets, relying on the PodCIDRs set by the Kubernetes Controller Manager.

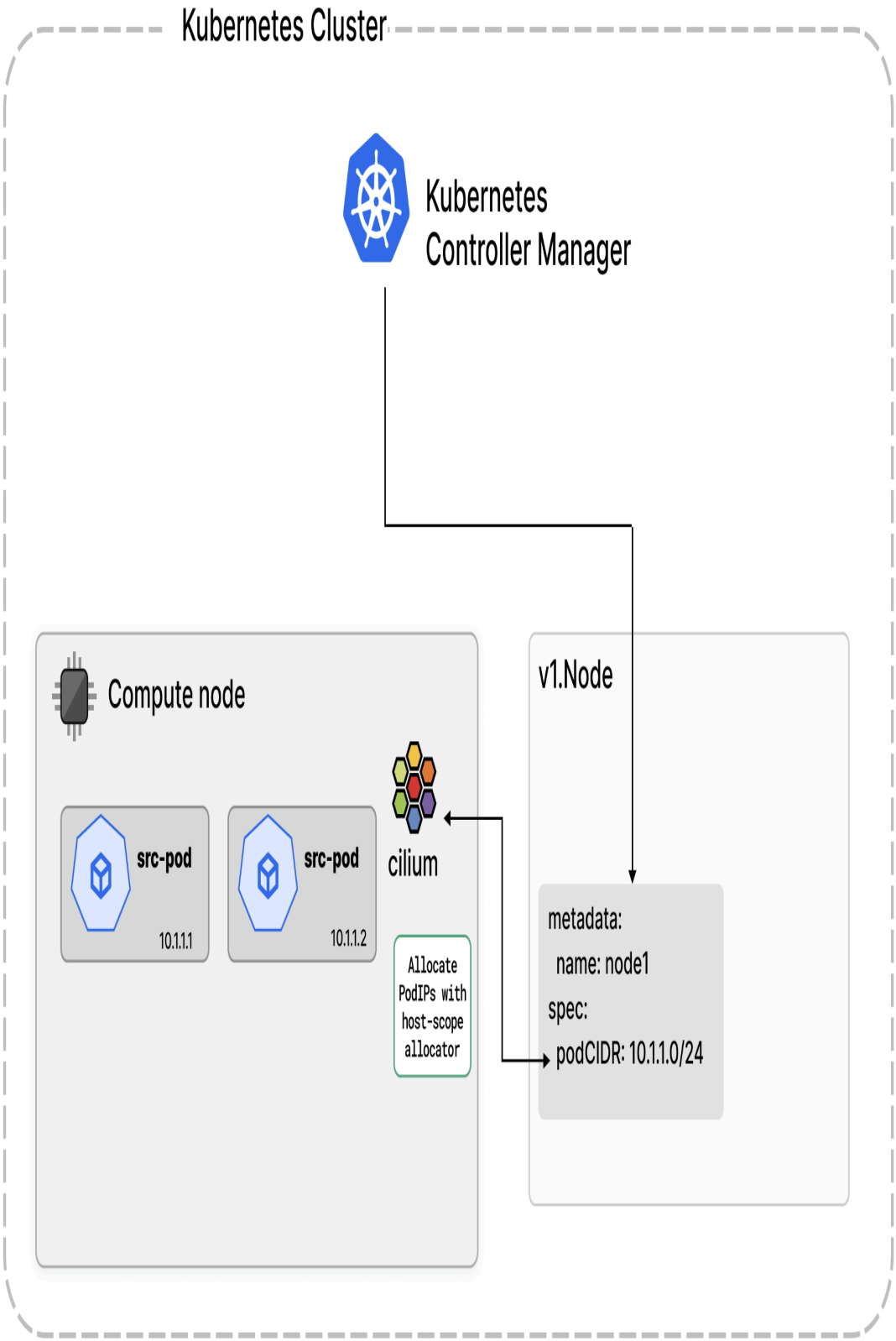

Kubernetes Cluster
Kubernetes
Controller Manager
Compute node
src-pod
10.1.1.1
src-pod
10.1.1.2
cilium
Allocate PodIPs with host-scope allocator
v1.Node
metadata:
name: node1
spec:
podCIDR: 10.1.1.0/24

Figure 3-2. Kubernetes-Host Scope Mode

Let's try it. Start by creating a kind cluster. We will use the same kind configuration we used in Chapter 2.

> **NOTE**
>
> You can find all the YAML manifests you will use in this chapter in the *chapter04* directory of the book's GitHub repository.

```
$ kind create cluster --config=chapter04/kind-cluster-
config.yaml
```

Once you have created the cluster, install Cilium with the following Helm values (cilium-ipam-kubernetes.yaml).

```
cluster:
  name: kind-kind
ipam:
  mode: kubernetes
operator:
  replicas: 1
routingMode: tunnel
tunnelProtocol: vxlan
```

Let's use Helm to deploy Cilium.

```
$ helm install cilium cilium/cilium -n kube-system --values
cilium-ipam-kubernetes.yaml
```

By default, kind uses "10.244.0.0/16" as the cluster-wide prefix. Kubernetes assigns /24 subnets to the various nodes in the cluster. Verify in your own cluster:

```
$ kubectl get ciliumnode kind-worker -o
jsonpath='{.spec.ipam}'
```

```
{"podCIDRs":["10.244.1.0/24"]}
```

> **NOTE**
>
> The assigned prefixes may differ in your environment as subnet allocation is done randomly.

Let's now deploy a pod.

```
$ kubectl apply -f netshoot-client-pod.yaml
pod/netshoot-client created
```

You can verify with `kubectl` that the pod receives an IP address from the node's subnet.

The `kubectl get ciliumnode kind-worker -o jsonpath='{.spec.ipam}'` command indicates the PodCIDR assigned to the node and `kubectl get pod netshoot-client -o wide` displays the pod's IP address.

While Host Scope is straightforward, it comes with limitations:

- You cannot configure the size of the PodCIDRs allocated to each node.
- You cannot add or remove CIDRs to the cluster or individual nodes.

This makes it difficult to grow the pool of IP addresses available to your cluster as it grows.

Let's take a closer look at a more flexible IPAM mode.

Cluster Scope IPAM Mode

In Cluster Scope mode, Cilium (and not Kubernetes) allocates PodCIDRs to nodes. As this mode doesn't require a specific

configuration of the Kubernetes cluster, it is the default IPAM option in Cilium.

In Chapter 2 we explained that one of the Cilium Operator's responsibilities is IPAM. In Cluster Scope mode, the operator assigns subnets to each node, by updating the CiliumNode custom resources (as illustrated in Figure 4-3).

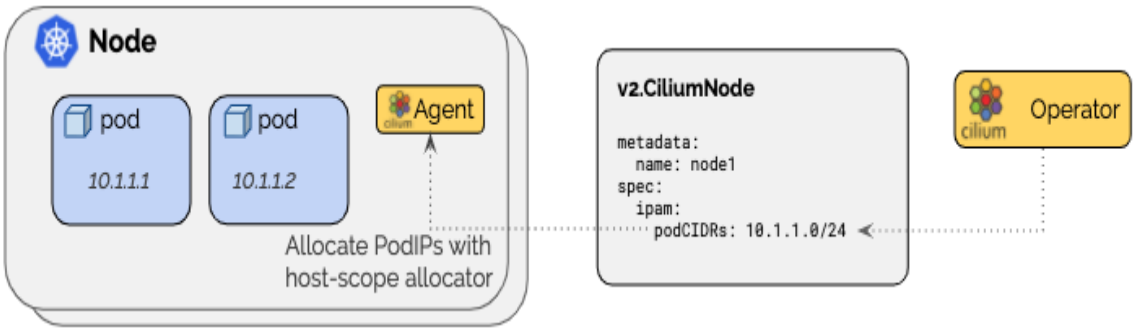


Figure 3-3. Cluster Scope IPAM Mode

First, let's re-create a cluster. If you had deployed a cluster previously, delete it first.

> **WARNING**
>
> We recommend you delete the cluster because changing the IPAM mode in a live environment may cause persistent connectivity disruption for existing workloads.

```
$ kind delete clusters kind
$ kind create cluster --config=kind-cluster-config.yaml
```

Install Cilium with the following Helm values (cilium-ipam-cluster-scope.yaml).

```
cluster:
  name: kind-kind
ipam:
  mode: cluster-pool
```

```
operator:
  replicas: 1
routingMode: tunnel
tunnelProtocol: vxlan
```

> **NOTE**
>
> "Cluster Scope" is sometimes referred to as "Cluster-Pool" as you can see from the Helm value.

```
$ helm install cilium cilium/cilium -n kube-system --values
cilium-ipam-cluster-scope.yaml
```

By default, Cilium uses the cluster-wide 10.0.0.0/8 prefix and assigns /24 subnets to CiliumNode resources. Verify in your cluster.

```
$ cilium config view | grep cluster-pool
cluster-pool-ipv4-cidr                    10.0.0.0/8
cluster-pool-ipv4-mask-size               24
ipam                                      cluster-pool
```

You can also list the CIDRs associated with each node.

```
$ kubectl get ciliumnode -o jsonpath='{range .items[*]}
{.metadata.name} {.spec.ipam.podCIDRs[]}{"\n"}{end}' |
column -t
kind-control-plane  10.0.0.0/24
kind-worker         10.0.1.0/24
kind-worker2        10.0.2.0/24
```

Deploy a sample pod.

```
$ kubectl apply -f netshoot-client-pod.yaml
pod/netshoot-client created
```

Check the verbose output of the `cilium-dbg status` on the Cilium agent on the node hosting the pod.

```
$ kubectl -n kube-system exec -ti cilium-jhhpp -- cilium-
dbg status --verbose
[...]
Allocated addresses:
  10.0.1.235 (default/netshoot-client)
  10.0.1.133 (router)
  10.0.1.238 (health)
```

One advantage cluster scope IPAM has over Kubernetes IPAM is that it can allocate multiple CIDRs to the cluster, instead of a single one. This mode also lets you specify the subnet mask size, unlike in Kubernetes-host scope mode. Cluster scope, therefore, can provide more flexibility, albeit it doesn't necessarily overcome IP address exhaustions.

Let's simulate such a scenario.

Once again, starting with a new cluster is preferable, so we will delete the previously created one and start with a new one.

Use the following values for Cilium's starting configuration (cilium-ipam-cluster-scope-small.yaml). In this example, Cilium will assign two small CIDRs (/28) to the cluster and each node will receive a /29 subnet from one of the CIDRs. As two IPs are reserved for Cilium Host and Cilium Health interfaces, each node will get 6 usage addresses.

```
cluster:
  name: kind-kind
ipam:
  mode: cluster-pool
  operator:
    clusterPoolIPv4MaskSize: 29
    clusterPoolIPv4PodCIDRList:
      - 10.0.42.0/28
      - 10.0.84.0/28
operator:
  replicas: 1
routingMode: tunnel
tunnelProtocol: vxlan
```

Let's install Cilium in this mode and observe what happens.

```
$ helm install cilium cilium/cilium -n kube-system --values
cilium-ipam-cluster-scope-small.yaml
```

Check the subnets assigned to your nodes.

```
$ kubectl get ciliumnode -o jsonpath='{range .items[*]}
{.metadata.name} {.spec.ipam.podCIDRs[*]}{"\n"}{end}' |
column -t
kind-control-plane  10.0.42.8/29
kind-worker         10.0.42.0/29
kind-worker2        10.0.84.8/29
kind-worker3        10.0.84.0/29
```

As a /28 prefix only allows two /29 allocations, the third and fourth nodes receive their subnets from the second block.

Let's now create a deployment with 10 pods.

```
$ kubectl apply -f nginx-deployment-10-replicas.yaml
deployment.apps/nginx-deployment created
```

When you run `kubectl get pods -o wide`, you'll notice that a number of pods remain stuck in *ContainerCreating* stage — meaning they are not running properly. When you run `cilium-dbg status` on any of the Cilium agents, you'll see that all available IP addresses have been allocated.

```
$ kubectl exec -it ds/cilium -n kube-system -c cilium-agent
-- cilium-dbg status
IPAM:                   IPv4: 6/6 allocated from
10.0.42.0/29,
```

While this mode provides greater flexibility than the Kubernetes Host Scope mode it comes with some limitations:

- You cannot dynamically add PodCIDRs to a running cluster

- You have limited control over how pods receive IP addresses.

To address these limitations, let’s take a look at Multi-Pool IPAM in the next section.

Multi-Pool IPAM Mode

Multi-Pool is the most flexible IPAM mode in Cilium. It enables you to allocate PodCIDRs from multiple distinct pools, depending on user-defined workload properties, such as annotations or namespaces. Pods on the same node can receive IP addresses from different ranges, and new CIDRs can be dynamically added to a node as and when needed.

This flexibility makes Multi-Pool the recommended mode, especially for multi-tenant clusters or when granular IP addressing control is preferred. The IP allocation process used in Multi-Pool IPAM mode is described in Figure 4-4.

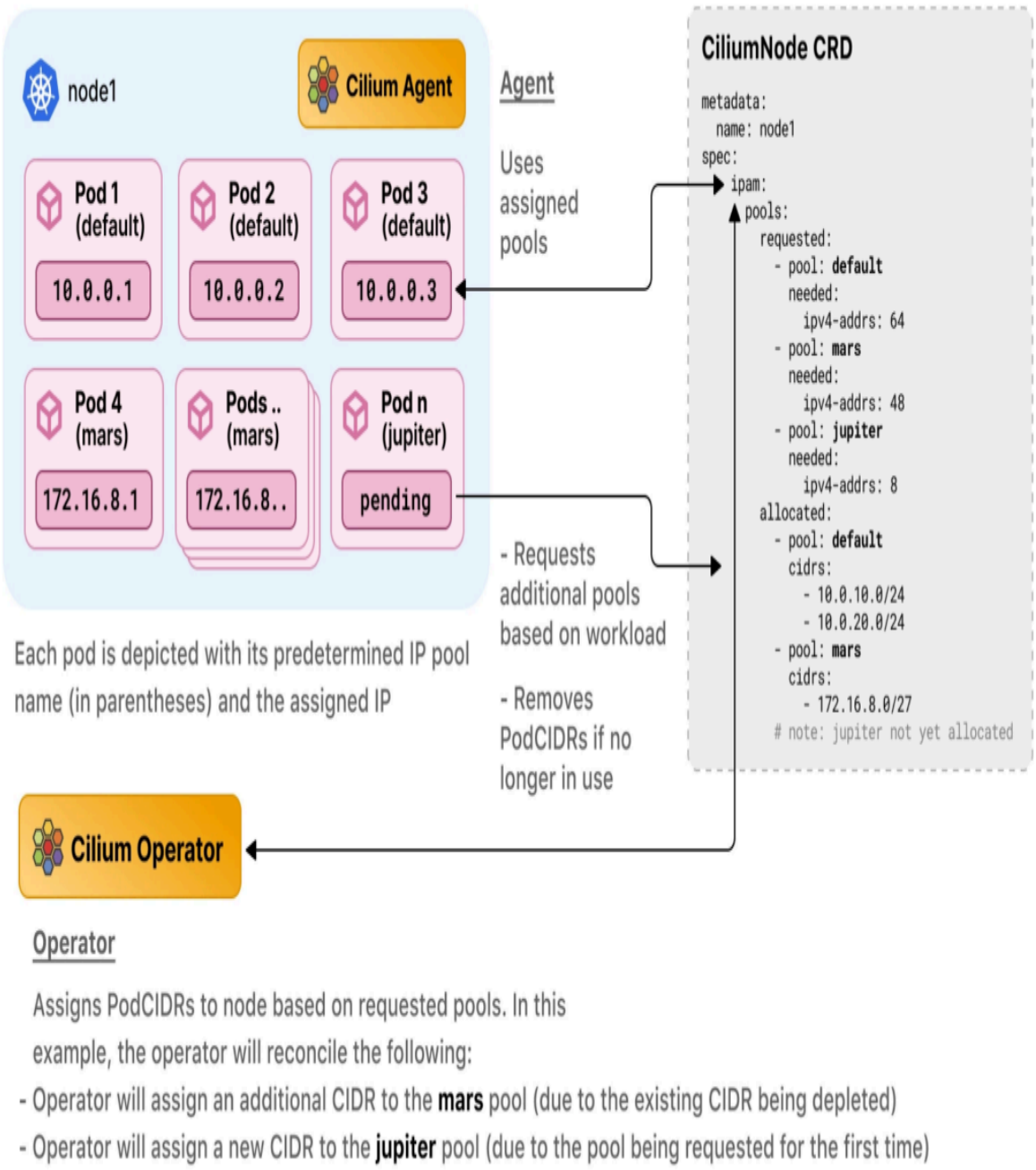


Figure 3-4. Multi-Pool IPAM Mode

Let's deploy Cilium in Multi-Pool mode. First, create a new cluster (delete the existing one if needed). Use the values below for Cilium's starting configuration (`cilium-ipam-multi-pool.yaml`).

```
ipam:
  mode: multi-pool
  operator:
    autoCreateCiliumPodIPPools:
      default:
        ipv4:
          cidrs: ["10.10.0.0/16"]
          maskSize: 27

routingMode: native
endpointRoutes:
  enabled: true

autoDirectNodeRoutes: true
ipv4NativeRoutingCIDR: 10.0.0.0/8
```

In Chapter 2, we introduced the encapsulation routing mode (using VXLAN). Multi-pool is not compatible with this mode so instead, you will need to use native routing which we will explain in more detail in Chapter 5.

Let's install Cilium.

```
$ helm install cilium cilium/cilium -n kube-system --values
cilium-ipam-multi-pool.yaml
```

The `autoCreateCiliumPodIPPools` option configures Cilium to create a default pool of IP addresses at startup and to assign nodes prefixes based on the `ipam.operator.autoCreateCiliumPodIPPools.ipv4.maskSize` value (in our case, 27).

Verify that a default pool has been created.

```
$ kubectl get ciliumpodippools.cilium.io default -o yaml
apiVersion: cilium.io/v2alpha1
kind: CiliumPodIPPool
spec:
  ipv4:
    cidrs:
```

```
    - 10.10.0.0/16
    maskSize: 27
```

Let's also verify that the kind-worker node received a /27 subnet from this pool.

```
$ kubectl get ciliumnode kind-worker -o yaml | yq
.spec.ipam
pools:
  allocated:
    - cidrs:
        - 10.10.0.64/27
      pool: default
  requested:
    - needed:
        ipv4-addrs: 16
      pool: default
```

Deploy a pod on this specific node using the `netshoot-client-pod.yaml` manifest we used previously.

```
$ kubectl apply -f netshoot-client-pod.yaml
pod/netshoot-client created
```

Once more, when you verify its IP address, the pod should receive an IP address from the allocated subnet (e.g., 10.10.0.93).

```
$ kubectl get pods -o wide
NAME              READY   STATUS    RESTARTS   AGE   IP
NODE
netshoot-client   1/1     Running   0          24s
10.10.0.93   kind-worker
```

So far, Multi-Pool functions similarly to Cluster Scope IPAM. One advantage of Multi-Pool over other IPAM modes is that it gives you more control over how IP addresses are assigned to pods. You can force a pod – or all pods in a particular namespace – to receive IPs from a particular range.

This is useful for multi-tenant environments. For example, consider two tenants: *ACME Corp.* and *Foobar Inc*. Namespaces are often used in Kubernetes to differentiate tenants.

While previous IPAM modes are not namespace-aware, Multi-Pool IPAM lets you assign pods in a namespace IPs from a distinct pool. This capability is particularly useful when traffic exits the cluster, as it helps engineers understand from which tenant the traffic originated.

> **NOTE**
>
> In Chapter 11, you will learn more about what to consider when traffic leaves the cluster, including another method to map outbound traffic to a particular IP or interface: Egress Gateway.

Let's create two namespaces for our two tenants.

```
$ kubectl apply -f namespaces.yaml
namespace/acme-corp created
namespace/foobar-inc created
```

Create two separate pools.

```
$ cat foobar-pool.yaml
apiVersion: cilium.io/v2alpha1
kind: CiliumPodIPPool
metadata:
  name: foobar-pool
spec:
  ipv4:
    cidrs:
    - 10.30.0.0/16
    maskSize: 27
$ cat acme-pool.yaml
apiVersion: cilium.io/v2alpha1
kind: CiliumPodIPPool
```

```
metadata:
  name: acme-pool
spec:
  ipv4:
    cidrs:
    - 10.20.0.0/16
    maskSize: 27
```

Apply the manifests.

```
$ kubectl apply -f foobar-pool.yaml
ciliumpodippool.cilium.io/foobar-pool created
$ kubectl apply -f acme-pool.yaml
ciliumpodippool.cilium.io/acme-pool created
```

To ensure that all workloads in a particular namespace receive an IP address from a particular IP pool, you can annotate the namespace with `ipam.cilium.io/ip-pool=namespace_name`.

```
$ kubectl annotate namespace acme-corp ipam.cilium.io/ip-
pool=acme-pool
namespace/acme-corp annotated
$ kubectl annotate namespace foobar-inc ipam.cilium.io/ip-
pool=foobar-pool
namespace/foobar-inc annotated
```

When deploying workloads, they will be allocated an IP address from the appropriate pool.

```
$ kubectl apply -f nginx-deployment-10-replicas.yaml -n
acme-corp
deployment.apps/nginx-deployment created

$ kubectl apply -f nginx-deployment-10-replicas.yaml -n
foobar-inc
deployment.apps/nginx-deployment created
```

Here is a trimmed version of the output.

```
$ kubectl get pods -o wide -n acme-corp
NAME                                READY   STATUS    IP
NODE
nginx-deployment-979f5455f-5tkws   1/1     Running
10.20.0.25   kind-worker2
nginx-deployment-979f5455f-6dw7n   1/1     Running
10.20.0.6    kind-worker2
nginx-deployment-979f5455f-j42nl   1/1     Running
10.20.0.18   kind-worker2
nginx-deployment-979f5455f-ksvd5   1/1     Running
10.20.0.20   kind-worker2
nginx-deployment-979f5455f-l4gwg   1/1     Running
10.20.0.7    kind-worker2
nginx-deployment-979f5455f-lblmh   1/1     Running
10.20.0.12   kind-worker2
nginx-deployment-979f5455f-lbltm   1/1     Running
10.20.0.26   kind-worker2
nginx-deployment-979f5455f-lnwjt   1/1     Running
10.20.0.15   kind-worker2
nginx-deployment-979f5455f-vv4nc   1/1     Running
10.20.0.27   kind-worker2
nginx-deployment-979f5455f-xg5h9   1/1     Running
10.20.0.29   kind-worker2

$ kubectl get pods -o wide -n foobar-inc
NAME                                READY   STATUS    IP
NODE
nginx-deployment-979f5455f-2p7mc   1/1     Running
10.30.0.28   kind-worker2
nginx-deployment-979f5455f-82g98   1/1     Running
10.30.0.13   kind-worker2
nginx-deployment-979f5455f-cdnz7   1/1     Running
10.30.0.29   kind-worker2
nginx-deployment-979f5455f-hxxtt   1/1     Running
10.30.0.25   kind-worker2
nginx-deployment-979f5455f-jklr2   1/1     Running
10.30.0.14   kind-worker2
nginx-deployment-979f5455f-kndrs   1/1     Running
10.30.0.27   kind-worker2
nginx-deployment-979f5455f-pbdtt   1/1     Running
10.30.0.6    kind-worker2
nginx-deployment-979f5455f-pcqsm   1/1     Running
10.30.0.20   kind-worker2
```

```
nginx-deployment-979f5455f-qmfwg   1/1     Running
10.30.0.4    kind-worker2
nginx-deployment-979f5455f-vxdx6   1/1     Running
10.30.0.30   kind-worker2
```

Check the Cilium Node to see which PodCIDRs were assigned to the nodes.

```
$ kubectl get ciliumnode kind-worker2 -o yaml | yq
.spec.ipam
pools:
  allocated:
    - cidrs:
        - 10.20.0.0/27
      pool: acme-pool
    - cidrs:
        - 10.10.0.96/27
      pool: default
    - cidrs:
        - 10.30.0.0/27
      pool: foobar-pool
  requested:
    - needed:
        ipv4-addrs: 10
      pool: acme-pool
    - needed:
        ipv4-addrs: 16
      pool: default
    - needed:
        ipv4-addrs: 10
      pool: foobar-pool
```

If you need more IPs, Cilium dynamically assigns additional subnets to the node. For example, scale up the ACME deployment from 10 replicas to 40.

```
$ kubectl scale -n acme-corp deployment nginx-deployment --
replicas=40
deployment.apps/nginx-deployment scaled
```

Note the `needed` field in the following output. As the number of required IPs significantly increased, another /27 subnet from the cluster-wide `10.20.0.0/16` was allocated to the node.

```
$ kubectl get ciliumnode kind-worker2 -o yaml | yq
.spec.ipam
pools:
  allocated:
    - cidrs:
        - 10.20.0.0/27
        - 10.20.0.32/27
      pool: acme-pool
    - cidrs:
        - 10.10.0.96/27
      pool: default
    - cidrs:
        - 10.30.0.0/27
      pool: foobar-pool
  requested:
    - needed:
        ipv4-addrs: 40
      pool: acme-pool
    - needed:
        ipv4-addrs: 16
      pool: default
    - needed:
        ipv4-addrs: 10
      pool: foobar-pool
```

NOTE

In some environments (e.g., telecoms or financial), pods may have multiple interfaces (for example, for out-of-band management and for signaling or data). The Isovalent Enterprise Platform extends Multi-Pool with a feature called Multi-Network, which lets you connect a pod to multiple networks.

CRD-Backed IPAM Mode

Cilium supports a lesser-known IPAM mode: CRD-backed IPAM, which delegates IP address management to an external operator. This mode gives you significant flexibility, but can be complex. When using CRD-backed mode, Cilium is not automatically programmed to route traffic to the assigned CIDRs, so administrators must assign IP ranges to nodes and ensure proper routing between them.

CRD-backed IPAM is often used in managed Kubernetes environments, where the cloud provider handles IP address management and routing outside of Cilium.

The final IPAM mode we will review is specific to Amazon Elastic Kubernetes Service and is widely adopted by AWS users, therefore deserving a detailed look through in this book.

ENI Mode

In ENI IPAM mode, Cilium integrates tightly with AWS networking by assigning pod IP addresses from secondary IPs on Elastic Network Interfaces (ENI). At startup, Cilium contacts the EC2 metadata ENI to retrieve instance ID, type, and VPC information and populates each Cilium Node custom resource with the info (as illustrated in Figure 4-5).

Cilium then allocates pod IPs directly from the pool of secondary addresses associated with those ENIs. This provides a clear benefit: pods receive IPs that are directly routable within the AWS VPC. No NAT is required, which simplifies networking and improves observability for operators.

> **WARNING**
>
> Note that ENI Mode is not currently supported with IPv6.

ENI mode is only relevant when running Cilium on Amazon EKS, where ENIs and their IPs are managed by AWS.

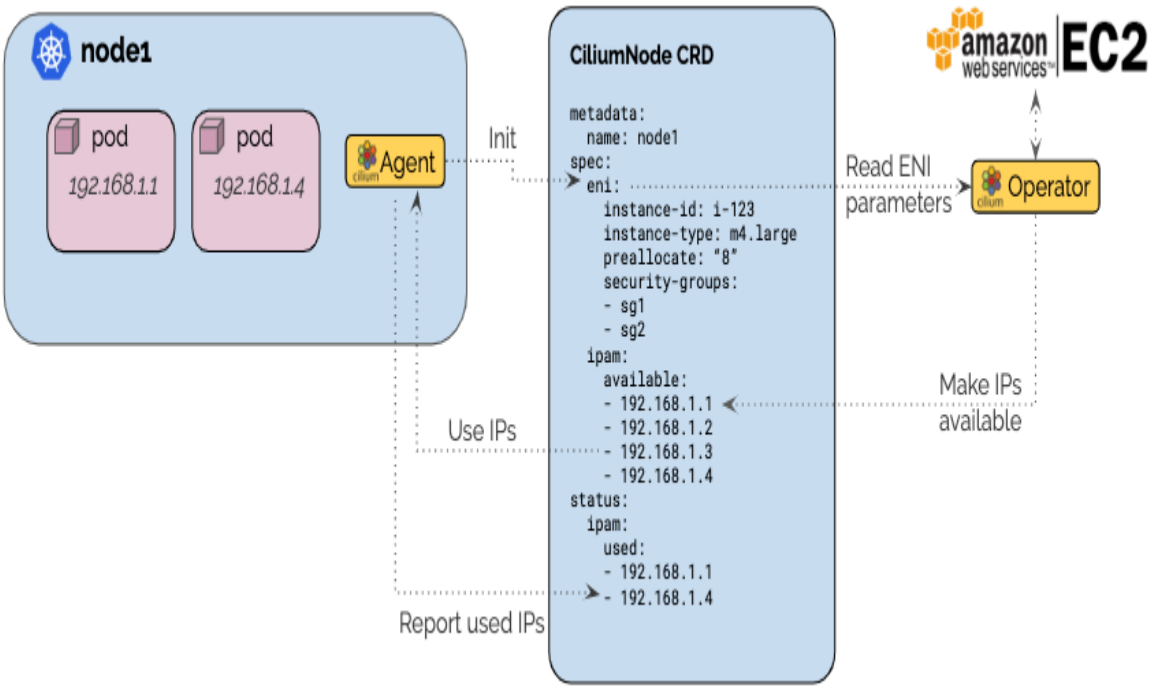


Figure 3-5. ENI Mode

Let's explain it by walking through the example.

> **WARNING**
>
> Unlike the previous examples where we were using kind, deploying a cluster on EKS comes with a cost. Proceed carefully before deploying a managed cloud-hosted cluster.

To deploy an EKS cluster, we recommend the eksctl tool. Follow the instructions in the documentation to install and configure it.

The eksctl configuration below will create a cluster with 2 nodes, in the `eu-west-1` region (update it to the region of your choice).

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
```

```
metadata:
  name: cuar
  region: eu-west-1

managedNodeGroups:
- name: ng-1
  desiredCapacity: 2
  privateNetworking: true
  taints:
   - key: "node.cilium.io/agent-not-ready"
     value: "true"
     effect: "NoExecute"
```

NOTE

In certain cloud environments (like EKS or GKE), another CNI plugin might already be installed, which may cause Cilium to fail to take control of CNI. To address this, Cilium supports a taint-based mechanism that delays pod scheduling until the agent is ready, helping ensure pods are managed by Cilium rather than the existing CNI plugin. Read more about it in the Cilium documentation.

Deploy the cluster

```
$ eksctl create cluster -f ./eks-config.yaml
```

After 15-20 minutes, the cluster should be ready, and you can now install Cilium.

```
$ helm install cilium cilium/cilium -n kube-system --values
helm-eni-values.yaml
```

Let’s deploy five pods with a sample deployment. All pods should receive an IP address as you would expect.

```
$ kubectl apply -f echo-deployment.yaml
deployment.apps/echoserver created

$ kubectl get pods -o wide
NAME                          READY   STATUS    IP
echoserver-79974b75cd-fwd9r   1/1     Running
192.168.168.230
echoserver-79974b75cd-kn9fz   1/1     Running
192.168.156.61
echoserver-79974b75cd-p5q9b   1/1     Running
192.168.131.53
echoserver-79974b75cd-wgdzf   1/1     Running
192.168.170.2
echoserver-79974b75cd-whjvc   1/1     Running
192.168.142.65
```

The pod IPs are actually taken from the EC2 instance's network interface and listed as a secondary private IPs, as you can see on the AWS Console, under EC2 > Instances > instance_name > Networking (Figure 4-6).

▼ Networking details Info

Public IPv4 address
-

Private IPv4 addresses
192.168.137.85
192.168.140.216

VPC ID
vpc-0dd8897be063cbd0b (eksctl-nvi-cluster/VPC)

Public IPv4 DNS
-

Private IP DNS name (IPv4 only)
ip-192-168-137-85.eu-west-1.compute.internal

Subnet ID
subnet-036eb318bd26ee489 (eksctl-nvi-cluster/SubnetPrivateEUWEST1C)

IPv6 addresses
-

Secondary private IPv4 addresses
192.168.159.141
192.168.156.61
192.168.149.241
192.168.142.65
192.168.139.147
192.168.158.35
192.168.135.116
192.168.131.53
192.168.157.103
192.168.157.2
192.168.151.105
192.168.150.0
192.168.142.5

Availability zone
eu-west-1c

Carrier IP addresses (ephemeral)
-

Outpost ID
-

Use RBN as guest OS hostname
Disabled

Answer RBN DNS hostname IPv4
Disabled

Figure 3-6. AWS Console

You can also see the IPs being populated on the Cilium Node resource.

```
$ kubectl get cn ip-192-168-132-54.eu-west-
1.compute.internal -o yaml | yq .status.eni
enis:
  eni-02f39382678db84b5:
    addresses:
      - 192.168.159.141
      - 192.168.156.61
      - 192.168.149.241
      - 192.168.142.65
      - 192.168.139.147
      - 192.168.158.35
      - 192.168.135.116
      - 192.168.131.53
      - 192.168.157.103
[...]
```

> **NOTE**
>
> When using ENI mode, you should be aware of certain scaling limitations. In ENI mode, the number of IPs that can be allocated per node is limited by the instance's type ENI and secondary IP capacity (e.g., a m5.large supports only 3 ENIs with 10 IPs each). This restricts the number of pods a node can run (with a m5.large only able to support 30 pods - far below Kubernetes' 110 pods per node limit).
>
> Cilium supports AWS Prefix Delegation, a feature that assigns entire CIDR blocks (/28) to each ENI. With this feature, pod density per node increases significantly, without having to scale the cluster horizontally or vertically.

IPv6 Clusters

The other essential decision you will need to make regarding IP addressing is which version of IP to use: IPv4, IPv6, or both. While

this choice often stems from broader organizational directives, it has direct implications for your Kubernetes networking architecture.

Cilium offers robust IPv6 support across all IPAM modes described in this chapter, with the exception of ENI mode. The vast majority of Cilium features are IPv6-ready, although a few still have limitations: encapsulation routing mode (covered in Chapter 5) and Egress Gateway (covered in Chapter 11) currently do not support IPv6.

In this section, we will explore two common deployment models:

- *Dual Stack (IPv4 and IPv6)*, where pods and services receive both an IPv4 and IPv6 address
- *Single Stack (IPv6 only)*, where pods and services only receive an IPv6 address.

Choosing between IPv4, Dual Stack, or IPv6-only should be based on a combination of technical readiness and organizational strategy. You should consider the following aspects:

Organizational IPv6 strategy: Is your company actively working towards IPv6 adoption or still primarily reliant on IPv4? Some enterprises mandate IPv6 for all new workloads; in some regions, governments and public sector institutions go even further, requiring IPv6 compliance for all digital infrastructure and services.

Operational readiness: Do your teams have the tooling and expertise to monitor and troubleshoot IPv6 traffic? Logging, observability, and security tooling must be IPv6-aware.

Ecosystem limitations: While Kubernetes and Cilium support IPv6 well, other tools and services may not. For instance, as of writing, GitHub-hosted services are not accessible via IPv6. This means IPv6-only environments may need to rely on mechanisms like NAT64/DNS64[1] to reach IPv4-only endpoints, adding complexity.

As a general guide:

- IPv4-only is the default and safest option for most organizations. It ensures broad compatibility albeit does not help you future-proof your stack.
- Dual Stack is often a transitional strategy. It allows IPv6 experimentation while maintaining full IPv4 compatibility. However, it introduces operational complexity, especially in IP management and observability.
- IPv6-only is viable in greenfield deployments where external dependencies are well understood and NAT64/DNS64 can be reliably deployed. It demands a mature operational model and close awareness of third-party IPv6 support.

With those considerations in mind, let's look at how to configure and operate Cilium in both Dual Stack and IPv6-only modes.

Dual Stack (IPv4/IPv6)

In Dual Stack configuration, each pod is allocated both an IPv4 and an IPv6 address. This enables communications with both IPv6 systems and legacy apps that only use IPv4.

To test it, we will use the following kind configuration (kind-cluster-dual-stack.yaml).

Use the following for Linux. Notice that the ipFamily configuration is set to `dual` to enable dual stack functionality, supporting both IPv4 and IPv6 protocols.

```
---
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
  - role: worker
  - role: worker
  - role: worker
networking:
```

```
  disableDefaultCNI: true
  ipFamily: dual
```

> **NOTE**
>
> This configuration works on Linux machines with IPv6 support enabled. When using a Mac, add the `apiServerAddress` set to `127.0.0.1` under `networking` to work around the lack of IPv6 forwarding in Docker. Read more on the kind docs for platform-specific guidance.

Deploy the cluster.

```
$ kind create cluster --config=kind-cluster-dual-stack.yaml
```

The nodes receive both an IPv4 and IPv6 address. Verify with the following command.

```
$ kubectl get node kind-worker -o
jsonpath='{.status.addresses}' | jq
[
  {
    "address": "172.18.0.4",
    "type": "InternalIP"
  },
  {
    "address": "fc00:f853:ccd:e793::4",
    "type": "InternalIP"
  },
[...]
]
```

With Cilium, IPv6 is disabled by default and has to be explicitly specified in the Helm values. Use the following command to deploy Cilium in dual stack mode.

```
$ helm install cilium cilium/cilium \
  --version 1.17.1 \
```

```
  --namespace kube-system \
  --set kubeProxyReplacement=true \
  --set k8sServiceHost=kind-control-plane \
  --set k8sServicePort=6443 \
  --set ipv6.enabled=true \
  --set ipam.mode=kubernetes
```

Run the following command to see the PodCIDRs from which IPv4 and IPv6 addresses will be allocated to your pods.

```
$ kubectl describe nodes | grep PodCIDRs
PodCIDRs:
10.244.0.0/24,fd00:10:244::/64
PodCIDRs:
10.244.1.0/24,fd00:10:244:1::/64
PodCIDRs:
10.244.3.0/24,fd00:10:244:3::/64
PodCIDRs:
10.244.2.0/24,fd00:10:244:2::/64
```

Once again, let's deploy a couple of sample pods and pin them to specific nodes to validate inter-node connectivity.

```
$ kubectl apply -f pod1.yaml -f pod2.yaml
pod/pod-worker created
pod/pod-worker2 created

$ kubectl get pods -o wide
NAME          READY   AGE     IP
pod-worker    1/1     2m31s   10.244.1.117
pod-worker2   1/1     2m31s   10.244.3.29
```

As we're using the `kubernetes` IPAM mode, IP addresses are assigned from the PodCIDR assigned to each node.

```
$ kubectl describe pod pod-worker | grep -A 2 IPs
IPs:
  IP:  10.244.1.117
  IP:  fd00:10:244:1::8204
$ kubectl describe pod pod-worker2 | grep -A 2 IPs
```

```
IPs:
  IP:  10.244.3.29
  IP:  fd00:10:244:3::833f
```

Let's verify that pods can communicate. We'll test a ping from pod-worker to pod-worker's IPv6 address. Because the pods were pinned to different nodes, it should show successful IPv6 connectivity between pods on different nodes.

```
$ IPv6=$(kubectl get pod pod-worker2 -o
jsonpath='{.status.podIPs[1].ip}')
$ echo $IPv6
fd00:10:244:3::833f
$ kubectl exec -it pod-worker -- ping6 -c 5 $IPv6
PING fd00:10:244:3::833f (fd00:10:244:3::833f) 56 data
bytes
64 bytes from fd00:10:244:3::833f: icmp_seq=1 ttl=63
time=0.330 ms
64 bytes from fd00:10:244:3::833f: icmp_seq=2 ttl=63
time=0.156 ms
64 bytes from fd00:10:244:3::833f: icmp_seq=3 ttl=63
time=0.133 ms
64 bytes from fd00:10:244:3::833f: icmp_seq=4 ttl=63
time=0.140 ms
64 bytes from fd00:10:244:3::833f: icmp_seq=5 ttl=63
time=0.148 ms

--- fd00:10:244:3::833f ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time
4087ms
rtt min/avg/max/mdev = 0.133/0.181/0.330/0.074 ms
```

Let's now deploy a Service. Note that we specify PreferDualStack in the Service configuration.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: echoserver
spec:
```

```
  replicas: 5
  selector:
    matchLabels:
      app: echoserver
  template:
    metadata:
      labels:
        app: echoserver
    spec:
      containers:
        - image: ealen/echo-server:latest
          imagePullPolicy: IfNotPresent
          name: echoserver
          ports:
            - containerPort: 80
          env:
            - name: PORT
              value: "80"
---
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  ipFamilyPolicy: PreferDualStack
  ipFamilies:
    - IPv6
    - IPv4
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
  type: ClusterIP
  selector:
    app: echoserver
```

Deploy the manifest.

```
$ kubectl apply -f echo-kube-dual-stack.yaml
deployment.apps/echoserver created
service/echoserver created
```

Verify that both IPv4 and IPv6 addresses are assigned.

```
# kubectl describe svc echoserver
Name:                     echoserver
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=echoserver
Type:                     ClusterIP
IP Family Policy:         PreferDualStack
IP Families:              IPv6,IPv4
IP:                       fd00:10:96::d4d5
IPs:                      fd00:10:96::d4d5,10.96.181.159
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
Endpoints:
10.244.1.206:80,10.244.3.235:80,10.244.2.55:80 + 7 more...
[...]
```

Let's verify access to the service from the pod-worker client:

```
$ ServiceIPv6=$(kubectl get svc echoserver -o
jsonpath='{.spec.clusterIP}')
$ echo $ServiceIPv6
fd00:10:96::d4d5
$ kubectl exec -i -t pod-worker -- curl -6
http://[$ServiceIPv6]/ | jq
{
  "host": {
    "hostname": "[fd00:10:96::d4d5]",
    "ip": "fd00:10:244:1::8204",
    "ips": []
  },
  "http": {
    "method": "GET",
    "baseUrl": "",
    "originalUrl": "/",
    "protocol": "http"
  },
```

While we're at it, and even if it's not Cilium's role, you can verify that the Kubernetes cluster DNS addon created an AAAA record for this service.

```
$ kubectl exec -i -t pod-worker -- nslookup -q=AAAA
echoserver.default
Server:         10.96.0.10
Address:        10.96.0.10#53

Name:   echoserver.default.svc.cluster.local
Address: fd00:10:96::d4d5
```

Single Stack (IPv6 only)

Let's now explore an IPv6-only cluster with Cilium. At the time of writing, IPv6 does not support tunnel (encapsulation) mode, and therefore IPv6 can only be installed in native mode (you'll learn more about native mode in Chapter 5).

Let's start once again with the kind cluster configuration. When using IPv6, set ipfamily to `ipv6` and specify the pod and service subnets.

```
---
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
  - role: worker
  - role: worker
networking:
  ipFamily: ipv6
  disableDefaultCNI: true
  podSubnet: "fd00:10:244::/48"
  serviceSubnet: "fd00:10:96::/112"
```

Deploy the cluster with `kind create cluster --config=kind-ipv6-only.yaml` and check that the nodes only

receive an IPv6 address.

```
$ kubectl get nodes -o wide
NAME                 STATUS   ROLES           AGE   VERSION
INTERNAL-IP
kind-control-plane   Ready    control-plane   29m   v1.31.0
fc00:f853:ccd:e793::4
kind-worker          Ready    <none>          29m   v1.31.0
fc00:f853:ccd:e793::2
kind-worker2         Ready    <none>          29m   v1.31.0
fc00:f853:ccd:e793::3
```

Only IPv6 PodCIDRs are assigned to the node.

```
$ kubectl describe node kind-worker
[...]
PodCIDR:                      fd00:10:244:2::/64
PodCIDRs:                     fd00:10:244:2::/64
[...]
```

As IPv4 is enabled by default in Cilium, make sure to disable it during the Cilium installation. As mentioned earlier, as encapsulation mode is not supported, we'll use native routing mode and autoDirectNodeRoutes instead (these features will be explained in Chapter 5).

```
$ helm install cilium cilium/cilium \
  --version 1.17.1 \
  --namespace kube-system \
  --set ipv6.enabled=true \
  --set ipv4.enabled=false \
  --set ipam.mode=kubernetes \
  --set routingMode=native \
  --set autoDirectNodeRoutes=true \
  --set ipv6NativeRoutingCIDR=fd00:10:244::/48
```

Let's deploy the two pods once again.

```
$ kubectl apply -f pod1.yaml -f pod2.yaml
pod/pod-worker created
pod/pod-worker2 created
```

They have only been assigned an IPv6 address.

```
$ kubectl get pods -o wide
NAME          READY   AGE   IP
pod-worker    1/1     10m   fd00:10:244:2::4647
pod-worker2   1/1     10m   fd00:10:244:1::e1c
```

Let's also consider the service - notice a slight change on the ipFamilyPolicy and ipFamilies settings required to deploy IPv6-only services.

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  ipFamilyPolicy: SingleStack
  ipFamilies:
  - IPv6
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
  type: ClusterIP
  selector:
    app: echoserver
```

Let's deploy the service and the associated deployment.

```
$ kubectl apply -f echo-kube-ipv6.yaml
deployment.apps/echoserver created
service/echoserver created
```

Pod-to-pod connectivity tests are successful.

```
$ IPv6=$(kubectl get pod pod-worker2 -o
jsonpath='{.status.podIPs[0].ip}')
$ echo $IPv6
fd00:10:244:1::e1c
$ kubectl exec -it pod-worker -- ping6 -c 5 $IPv6
PING fd00:10:244:1::e1c (fd00:10:244:1::e1c) 56 data bytes
64 bytes from fd00:10:244:1::e1c: icmp_seq=1 ttl=60
time=0.179 ms
64 bytes from fd00:10:244:1::e1c: icmp_seq=2 ttl=60
time=0.090 ms
64 bytes from fd00:10:244:1::e1c: icmp_seq=3 ttl=60
time=0.094 ms
^C
--- fd00:10:244:1::e1c ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time
2039ms
rtt min/avg/max/mdev = 0.090/0.121/0.179/0.041 ms
```

So are pod-to-service and DNS resolution.

```
$ ServiceIPv6=$(kubectl get svc echoserver -o
jsonpath='{.spec.clusterIP}')
$ echo $ServiceIPv6
fd00:10:96::5876
$
$ kubectl exec -i -t pod-worker -- curl -6 -o /dev/null -s
-w "%{http_code}\n" http://[$ServiceIPv6]/
200

$ kubectl exec -i -t pod-worker -- nslookup -q=AAAA
echoserver.default
Server:         fd00:10:96::a
Address:        fd00:10:96::a#53

Name:   echoserver.default.svc.cluster.local
Address: fd00:10:96::5876

$ kubectl exec -i -t pod-worker -- curl -6 -o /dev/null -s
-w "%{http_code}\n" http://echoserver.default.svc
200
```

Conclusion

This chapter laid the foundation for Cilium networking by covering IP address management (IPAM) and IPv6/dual-stack support. We explored the various IPAM modes Cilium offers, from simple host-scope setups to more flexible options like cluster-scope and multi-pool, and discussed how to deploy clusters using IPv4, IPv6, or both.

These decisions are critical; changing them later can be disruptive, unsupported, or simply impractical. Our goal is to help you make informed, durable choices that align with your environment's long-term needs.

Now that you understand how Cilium assigns IP addresses to pods, the next step is to understand how traffic actually flows between them. In the next chapter, we'll dive into how packets are forwarded within and across nodes, examine the role of encapsulation, and compare native routing with overlay tunnels. We'll even dissect packet headers to see how Cilium, and the underlying network, moves traffic across your cluster.

---

[1] NAT64 and DNS64 are mechanisms that allow IPv6-only clients to communicate with IPv4-only servers. DNS64 synthesizes AAAA records from A records, enabling IPv6 clients to resolve IPv4-only domain names. NAT64 then translates the IPv6 traffic to IPv4 at the network edge. Together, they provide access from IPv6-only environments to legacy IPv4 services.

Chapter 4. Cilium Datapath

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo is available at *https://github.com/isovalent/cilium-up-and-running*.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *gobrien@oreilly.com*.

In the previous chapter, we explored how Cilium allocates IP addresses to pods. With each pod now assigned an IP, the next logical step is to understand how traffic flows between them. This is the role of the Cilium datapath: the set of technologies Cilium uses to process, route, and forward packets within a Kubernetes cluster.

In this chapter, we'll examine how packets are delivered both within and across nodes; how different routing models work; and how encapsulation protocols like VXLAN and Geneve come into play when native routing isn't possible.

We'll begin with the simplest case (intra-node pod-to-pod communication), then move on to inter-node traffic and the differences between native and overlay modes (Figure 5-1):

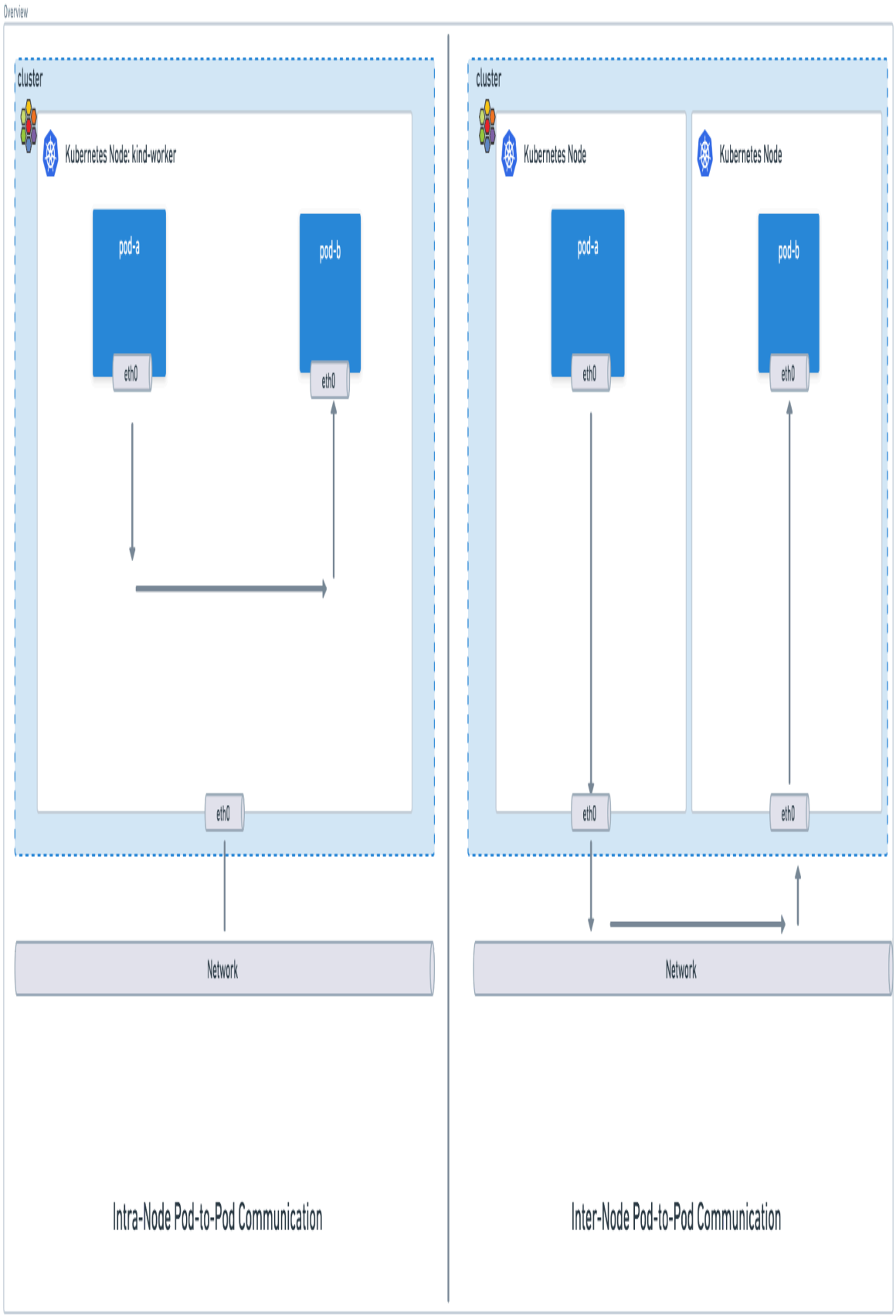

Overview
cluster
Kubernetes Node: kind-worker
pod-a
pod-b
eth0
eth0
eth0
Network
Intra-Node Pod-to-Pod Communication
cluster
Kubernetes Node
Kubernetes Node
pod-a
pod-b
eth0
eth0
eth0
eth0
Network
Inter-Node Pod-to-Pod Communication

Figure 4-1. Intra-node and Inter-node communication overview

BACKGROUND: LINUX NAMESPACES AND VETH PAIRS

This chapter does not attempt to cover Linux and container networking comprehensively. For a broader introduction, see *Networking and Kubernetes* by James Strong and Vallery Lancey (O'Reilly), especially Chapters 2 and 3. Here, we will briefly cover the essentials you need to follow how Cilium integrates with the kernel datapath.

Before diving into Cilium specifics, recall two key building blocks of container networking (Figure 5-2):

Network namespaces

In Linux, a network namespace provides an isolated copy of the network stack (interfaces, routing tables, firewall rules). Each Kubernetes pod runs in its own network namespace, giving it the illusion of having a dedicated network stack. The host itself also has a network namespace that is shared by system processes (including the Cilium agent).

Veth pairs

Virtual Ethernet devices come in pairs. Packets sent into one end of the pair immediately appear on the other end. Kubernetes uses veth pairs to connect pod namespaces to the host's network namespace. Inside the pod, you see an interface such as `eth0`; on the host, you see its peer, usually named `lxc*`.

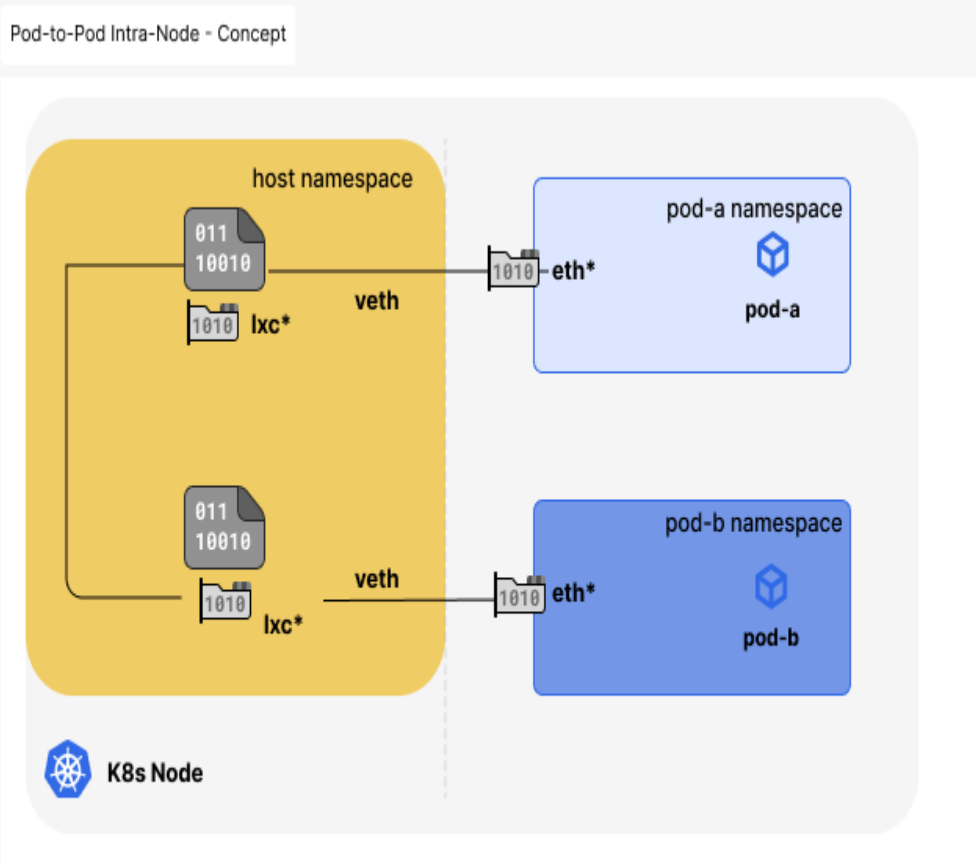


Figure 4-2. High-level view of container networking

Intra-Node Connectivity

With this background in mind, let us now look at how two pods (`pod-a` and `pod-b`) on the same node communicate. To do that, we will trace a successful ping from one pod to another, running on the same node.

As explained in the background section, each pod connects to the host through a veth pair. Cilium attaches eBPF programs to these interfaces so that packets are inspected and forwarded as soon as they leave or enter the pod. This early interception allows Cilium to

consult its eBPF maps for identity, policy, or connection tracking information and apply decisions immediately.

This is different from many traditional CNIs, where packets are typically bridged or passed through iptables chains for filtering and routing. With Cilium, the decision is made directly in the datapath at the earliest possible moment, avoiding the extra layers of processing.

Let's test it.

> **NOTE**
>
> You can find all the YAML manifests you will use in this chapter in the chapter05 directory of the book's GitHub repository.

In a generic kind cluster (`kind create cluster --config=kind-cluster-config.yaml`), with a default Cilium installation (`cilium install`), deploy a manifest such as the one below (`intra-node-example.yaml`) with kubectl. It will deploy two pods (`pod-a` and `pod-b`) on the same node (`kind-worker`).

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-a
spec:
  nodeName: kind-worker
  containers:
    - name: netshoot
      image: nicolaka/netshoot:latest
      command: ["sleep", "infinite"]
---
apiVersion: v1
kind: Pod
metadata:
```

```
  name: pod-b
spec:
  nodeName: kind-worker
  containers:
    - name: netshoot
      image: nicolaka/netshoot:latest
      command: ["sleep", "infinite"]
```

Check their IP addresses and verify the successful intra-node ping:

```
$ kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP
NODE
pod-a   1/1     Running   0          58m   10.244.1.203
kind-worker
pod-b   1/1     Running   0          58m   10.244.1.69
kind-worker

$ kubectl exec -it pod-a -- ping 10.244.1.69
PING 10.244.1.69 (10.244.1.69) 56(84) bytes of data.
64 bytes from 10.244.1.69: icmp_seq=1 ttl=63 time=0.223 ms
64 bytes from 10.244.1.69: icmp_seq=2 ttl=63 time=0.073 ms
```

Both pods are connected to the host's network namespace via their veth pairs. The host namespace acts as the common network domain where these veth pairs terminate and provides the bridge between otherwise isolated pod stacks. For two pods on the same node to exchange traffic, their packets must leave the pod namespace and pass through this shared host namespace, where Cilium's datapath logic is applied.

Enter the `pod-a` shell with `kubectl exec -it pod-a -- bash` and check its `eth0` interface details and routing table. From inside `pod-a`, the output shows that its `eth0` interface, with the index 23, is connected to a peer with index 24 (`eth0@if24`). This indicates that the other end of the veth pair exists outside the pod's namespace.

```
pod-a:~# ip address show eth0
23: eth0@if24: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 65520
qdisc noqueue state UP group default qlen 1000
    link/ether 76:43:9c:d5:bd:2a brd ff:ff:ff:ff:ff:ff
link-netnsid 0
    inet 10.244.1.203/32 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::7443:9cff:fed5:bd2a/64 scope link proto
kernel_ll
       valid_lft forever preferred_lft forever

pod-a:~# ip route
default via 10.244.1.198 dev eth0 mtu 65470
10.244.1.198 dev eth0 scope link
```

The `@if24` suffix shows the peer index. To confirm what this peer is, we look at the host's network namespace. On the host, the other end of the veth pair appears as an `lxc*` interface with the corresponding peer index (`if23` in this case):

```
$ docker exec -it kind-worker bash
root@kind-worker:/# ip address | grep -A 3 if23
24: lxcbee302eb186c@if23: <BROADCAST,MULTICAST,UP,LOWER_UP>
mtu 65520 qdisc noqueue state UP group default qlen 1000
    link/ether a2:cf:c3:e6:e4:c7 brd ff:ff:ff:ff:ff:ff
link-netns cni-d35c64e3-0ea6-5967-0698-427e7255cdc8
    inet6 fe80::a0cf:c3ff:fee6:e4c7/64 scope link
       valid_lft forever preferred_lft forever
```

This pairing (`eth0@if24` inside the `pod` ↔ `lxc*@if23` on the host) is how traffic leaves the pod's isolated namespace and enters the host's network namespace, where Cilium's eBPF programs can process it.

Note you will see a similar pairing for pod-b:

```
## on pod-b
pod-b:~# ip address show eth0
25: eth0@if26: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 65520
qdisc noqueue state UP group default qlen 1000
```

```
    link/ether 46:9c:c0:5f:ec:1c brd ff:ff:ff:ff:ff:ff
link-netnsid 0
    inet 10.244.1.69/32 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::449c:c0ff:fe5f:ec1c/64 scope link proto
kernel_ll
       valid_lft forever preferred_lft forever

## on the host

root@kind-worker:/# ip address | grep -A 3 if25
26: lxcc5b0a1b7ddfb@if25: <BROADCAST,MULTICAST,UP,LOWER_UP>
mtu 65520 qdisc noqueue state UP group default qlen 1000
    link/ether fa:a2:c6:90:65:f7 brd ff:ff:ff:ff:ff:ff
link-netns cni-f4803bdf-3eb6-a209-831e-b93beac138a3
    inet6 fe80::f8a2:c6ff:fe90:65f7/64 scope link
       valid_lft forever preferred_lft forever
```

This confirms that both pods exchange traffic through the host namespace, with their veth pairs terminating there.

This intra-node pod-to-pod connectivity is depicted in Figure 5-3:

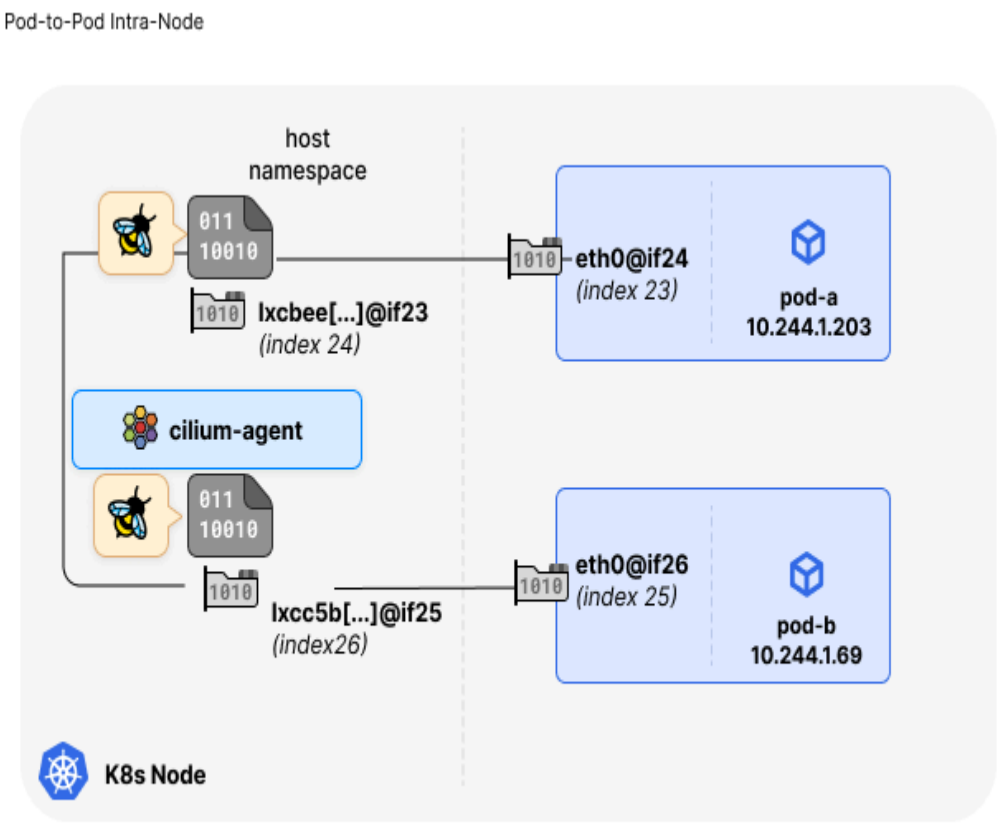


Figure 4-3. Intra-node connectivity

When pod-a sends a packet to pod-b, it uses its default gateway (`10.244.1.198` in this example). This gateway address maps to the `cilium_host` device in the host's network namespace. The `cilium_host` interface represents the node's entry and exit point for pod traffic: all packets leaving or entering pods on the node will traverse this interface.

```
root@kind-worker:/# ip address show cilium_host
13: cilium_host@cilium_net:
<BROADCAST,MULTICAST,NOARP,UP,LOWER_UP> mtu 65520 qdisc
noqueue state UP group default qlen 1000
    link/ether 2e:62:71:74:47:90 brd ff:ff:ff:ff:ff:ff
    inet 10.244.1.198/32 scope global cilium_host
       valid_lft forever preferred_lft forever
```

```
    inet6 fe80::2c62:71ff:fe74:4790/64 scope link
       valid_lft forever preferred_lft forever
```

Cilium also installs a set of interfaces during initialization, which can be confusing the first time you see them in the `ip address` output. Let's break them down:

- `Cilium_net`: a companion device paired with `cilium_host`. Together, they form the two ends of a veth pair, which is why the output shows `cilium_host@cilium_net` and `cilium_net@cilium_host`. This construction allows traffic between pods and the host to be processed by Cilium's datapath.

  ```
  root@kind-worker:/# ip address show cilium_net
  12: cilium_net@cilium_host:
  <BROADCAST,MULTICAST,NOARP,UP,LOWER_UP> mtu 65520
  qdisc noqueue state UP group default qlen 1000
      link/ether ba:6c:63:bb:3a:f8 brd
  ff:ff:ff:ff:ff:ff
      inet6 fe80::b86c:63ff:febb:3af8/64 scope link
         valid_lft forever preferred_lft forever
  ```

- `cilium_vxlan` (or `cilium_geneve`, if Geneve is enabled): the overlay interface used for tunnelling traffic to other nodes when encapsulation mode is configured. It is not used for intra-node pod-to-pod traffic; packets that stay on the same node are forwarded via the host's network namespace without encapsulation. We will cover encapsulation mode later in this chapter.

  ```
  root@kind-worker:/# ip address show cilium_vxlan
  14: cilium_vxlan: <BROADCAST,MULTICAST,UP,LOWER_UP>
  mtu 65520 qdisc noqueue state UNKNOWN group default
      link/ether c2:6e:2a:c2:91:42 brd
  ff:ff:ff:ff:ff:ff
      inet6 fe80::c06e:2aff:fec2:9142/64 scope link
         valid_lft forever preferred_lft forever
  ```

To understand how Cilium programs control traffic, we can inspect which eBPF programs are attached to the node's interfaces. This can be done with either `cilium-dbg` (introduced in Chapter 2 and first used in Chapter 4) or the kernel's `bpftool`. Here we will use `bpftool` so you can see exactly which programs are attached to which devices.

```
$ kubectl -n kube-system exec -it cilium-gq458 -c cilium-
agent -- bash
root@kind-worker:/home/cilium# bpftool net show
xdp:

tc:
eth0(11)          tcx/ingress cil_from_netdev      prog_id
617 link_id 16
cilium_net(12)    tcx/ingress cil_to_host          prog_id
609 link_id 15
cilium_host(13)   tcx/ingress cil_to_host          prog_id
607 link_id 13
cilium_host(13)   tcx/egress  cil_from_host        prog_id
601 link_id 14
cilium_vxlan(14)  tcx/ingress cil_from_overlay     prog_id
598 link_id 11
cilium_vxlan(14)  tcx/egress  cil_to_overlay       prog_id
599 link_id 12
lxc_health(16)    tcx/ingress cil_from_container   prog_id
633 link_id 17
lxc0090ec0fed69(18) tcx/ingress cil_from_container prog_id
643 link_id 18
lxcefc945dce028(20) tcx/ingress cil_from_container prog_id
666 link_id 20
lxce97d7b72b3aa(22) tcx/ingress cil_from_container prog_id
657 link_id 19
lxcbee302eb186c(24) tcx/ingress cil_from_container prog_id
1668 link_id 37
lxcc5b0a1b7ddfb(26) tcx/ingress cil_from_container prog_id
1695 link_id 38
[...]
```

The listing shows several important things:

- Host devices (`cilium_host`, `cilium_net`, `cilium_vxlan`) all have programs attached, handling traffic into and out of the node.
- Pod veth devices (`lxc*`) each have ingress programs attached. These entries are shown as `cil_from_container-*`. The attachment point is ingress on the veth, which corresponds to packets leaving the container. In other words, whenever a packet exits a pod, it first passes through this eBPF program before entering the host's network namespace.

Putting this together: packets from pod-a traverse its veth into the host-side `lxc*` device, where Cilium's ingress program (`cil_from_container-*`) runs first. The packet is then steered through the node's datapath (via `cilium_host`), and finally delivered to pod-b if the traffic stays on the same node.

You can confirm the packet path with a simple packet capture on the host-side veth for `pod-b (lxcc5b0a1b7ddfb)`, by using tcpdump on the worker node. The ICMP echo request and reply below are the ping traffic we initiated earlier from pod-a (10.244.1.203) to pod-b (10.244.1.69):

```
$ tcpdump -i lxcc5b0a1b7ddfb
tcpdump: verbose output suppressed, use -v[v]... for full
protocol decode
listening on lxcc5b0a1b7ddfb, link-type EN10MB (Ethernet),
snapshot length 262144 bytes
14:12:34.847472 IP 10.244.1.203 > 10.244.1.69: ICMP echo
request, id 51, seq 1, length 64
14:12:34.847589 IP 10.244.1.69 > 10.244.1.203: ICMP echo
reply, id 51, seq 1, length 64
```

This confirms the intra-node path: packets leave `pod-a`, enter the host's network namespace, are processed by Cilium's eBPF datapath, and are delivered to `pod-b` without encapsulation.

NOTE

From Cilium 1.16 onwards, an alternative to veth called the netkit device is supported. Introduced in Linux 6.7, netkit allows eBPF programs to run directly inside pod namespaces for improved performance. This is covered in more detail in Chapter 8 on performance networking.

Inter-Node Connectivity

While intra-node pod-to-pod communication is handled entirely on the local host, inter-node traffic must be forwarded across the physical network connecting cluster nodes. This introduces additional complexity, as each node must know which pod IP ranges are assigned to other nodes and how to reach them.

A fundamental rule of Kubernetes networking is that **every pod must be able to communicate with every other pod using its IP address, without any form of Network Address Translation (NAT)**. In other words:

- The source IP of a packet sent by a pod must be visible, unmodified, at the destination pod.
- The destination pod must be reachable directly by its pod IP, not by a translated node IP or service address.

To meet this requirement, Cilium supports two primary routing models for inter-node connectivity:

- **Native routing**, where the nodes and the underlying network are aware of all pod networks and can forward the traffic accordingly.
- **Encapsulation (overlay) routing**, where Cilium builds a mesh of tunnels between nodes and hides pod network details from the underlay.

Let's explore both models, how Cilium implements them, and the trade-offs of each.

Native Routing Mode

In native routing, packets sent by the pods are routed through the host networking stack, just like packets from any other local process. Once a packet exits the pod and enters the host's networking namespace, it is routed based on the host's routing table.

This mode requires that every Kubernetes node knows which PodCIDRs are assigned to other nodes, and that the underlying network is able to deliver packets between those ranges. For example, if the `10.10.0.32/27` subnet is allocated to **kind-worker2**, then **kind-worker1** must have a route directing packets for that range to **kind-worker2**'s node IP.

Table 5-1 shows a simple example with three nodes, their PodCIDRs, and the corresponding next hops.

Table 4-1. Native routing example

Destination Prefix	Destination Node	Next-hop
10.10.0.0/27	kind-worker	172.18.0.3
10.10.0.32/27	kind-worker2	172.18.0.4
10.10.0.64/27	kind-control-plane	172.18.0.5

The same example is illustrated in the Figure 5-4:

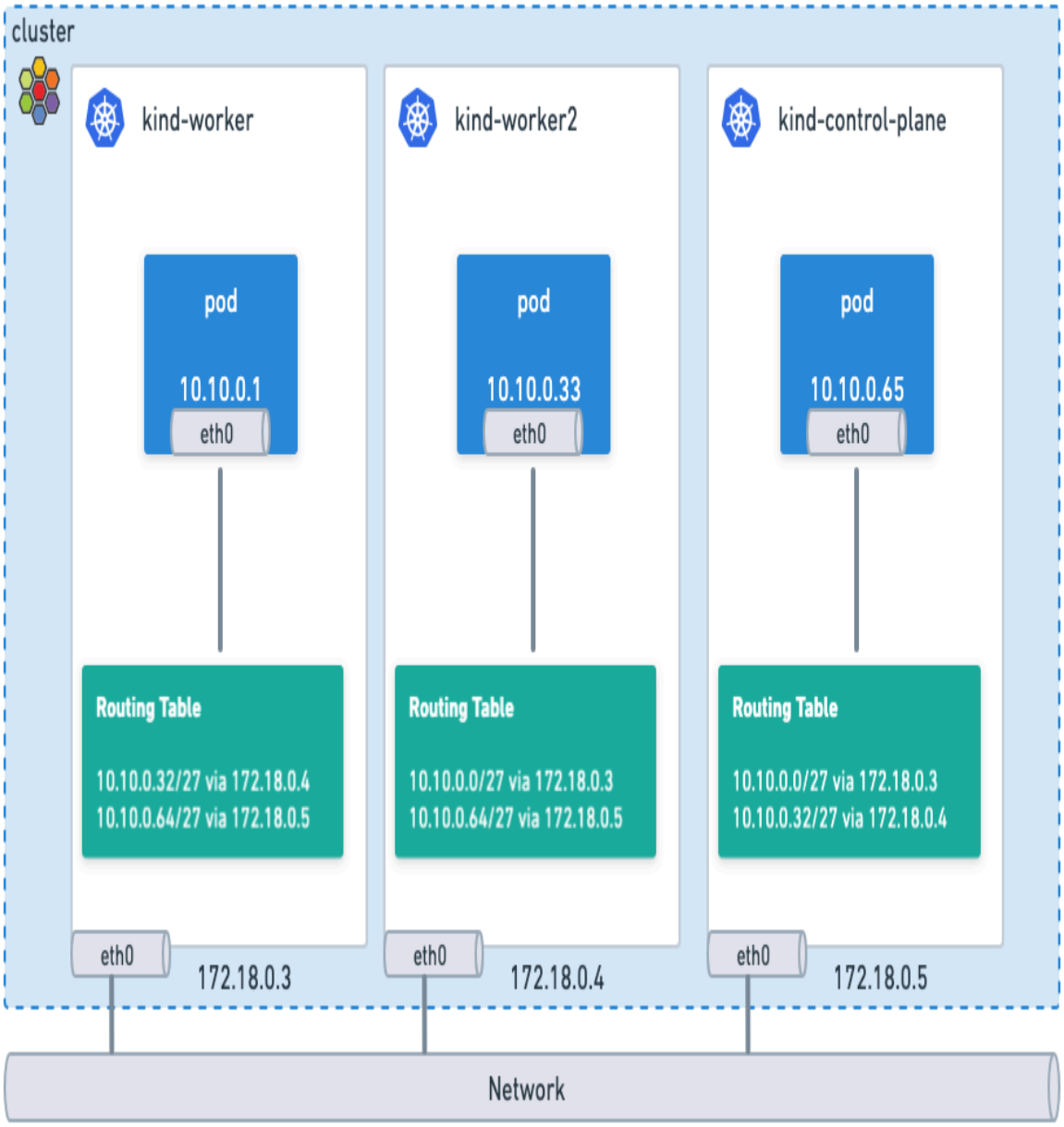


Figure 4-4. Native routing mode

As encapsulation is Cilium's default routing mode; native routing must be enabled explicitly, with the following Helm flag:

```
routingMode: native
```

The other required value when using native routing is the CIDR range that is already routable by the underlying networking, which can be defined with the `ipv4NativeRoutingCIDR` (or `ipv6NativeRoutingCIDR`) option. It indicates that packets within that range should be handed directly to the Linux kernel's routing subsystem.

For the nodes to learn about each PodCIDR, you have three main options:

- Auto Route Injection
- Static Routing
- BGP

Let's review them in detail.

Auto Route Injection

Auto route injection (also known as auto-direct node routes) is the simplest way to propagate the pod network information but requires each node to be on the same Layer 2 network.

Let's try it. First, create a generic kind cluster once again:

```
$ kind create cluster --config=kind-cluster-config.yaml
```

Istall Cilium with the following Helm values (`cilium-native-auto-node-routes.yaml`). We are actually re-using the values from the Multi-Pool example from Chapter 4.

```
ipam:
  mode: multi-pool
  operator:
    autoCreateCiliumPodIPPools:
      default:
```

```
        ipv4:
          cidrs: ["10.10.0.0/16"]
          maskSize: 27

routingMode: native
endpointRoutes:
  enabled: true

autoDirectNodeRoutes: true
ipv4NativeRoutingCIDR: 10.0.0.0/8
```

When using auto route injection (indicated by the `autoDirectNodeRoutes:true` option), Cilium updates each node's routing table to include routes for the PodCIDRs owned by other nodes.

Let's check the PodCIDRs first. You will remember from the Chapter 4's section on Multi-Pool and the previous Helm configuration that Cilium will automatically create podCIDRs on each node. Let's check:

```
$ kubectl get ciliumnodes kind-worker -o yaml | yq
.spec.ipam.pools.allocated
- cidrs:
    - 10.10.0.64/27
  pool: default
$ kubectl get ciliumnodes kind-worker2 -o yaml | yq
.spec.ipam.pools.allocated
- cidrs:
    - 10.10.0.32/27
  pool: default
$ kubectl get ciliumnodes kind-control-plane -o yaml | yq
.spec.ipam.pools.allocated
- cidrs:
    - 10.10.0.0/27
  pool: default
```

Let's inspect the routing tables on the nodes in a kind cluster with autoDirectNodeRoutes enabled. Cilium has automatically populated

each table with routes to the PodCIDRs, using the corresponding node IPs as next hops.

```
$ docker exec kind-worker ip route
[...]
10.10.0.0/27 via 172.18.0.4 dev eth0 proto kernel
10.10.0.32/27 via 172.18.0.2 dev eth0 proto kernel

$ docker exec kind-worker2 ip route
[...]
10.10.0.0/27 via 172.18.0.4 dev eth0 proto kernel
10.10.0.64/27 via 172.18.0.3 dev eth0 proto kernel

$ docker exec kind-control-plane ip route
[...]
10.10.0.32/27 via 172.18.0.2 dev eth0 proto kernel
10.10.0.64/27 via 172.18.0.3 dev eth0 proto kernel
```

Apart from the auto route Injection option, other approaches to distribute PodCIDRs include:

Static Routing

In very small clusters with a fixed set of nodes, such as a two-node lab environment or a development setup where nodes rarely change, you can configure PodCIDRs manually using `ip route` or a configuration management tool.

For instance, if the PodCIDR `10.10.0.32/27` is allocated to kind-worker2 (node IP `172.18.0.4`), then every other node in the cluster must be told how to reach that range. On kind-worker you would configure:

```
ip route add 10.10.0.32/27 via 172.18.0.4 dev
eth0
```

This ensures that any packets destined for `10.10.0.32/27` are forwarded to kind-worker2. The same principle applies in reverse: each node must have static routes installed for the PodCIDRs owned by all of the other nodes.

While simple and explicit, this approach does not scale. As soon as nodes are added, removed, or rescheduled, static routes need to be updated manually across the cluster. This quickly becomes error-prone and operationally unmanageable outside of very small test setups.

Dynamic routing with BGP

In production environments with many nodes or frequent churn, such as deployments spanning multiple racks, the most scalable option is to distribute PodCIDRs dynamically using BGP. With BGP enabled, each node advertises its PodCIDRs to its BGP peers (which might be other nodes or upstream routers). This ensures routes are kept up to date automatically. With Cilium's BGP support (covered in Chapter 10), a `CiliumBGPPeeringPolicy` can be used to advertise pod prefixes directly.

With a `CiliumBGPPeeringPolicy` like the one that follows, nodes can dynamically advertise the PodCIDR to the underlying network (typically, by peering with Top of Rack devices).

```
apiVersion: cilium.io/v2alpha1
kind: CiliumBGPPeeringPolicy
metadata:
  name: bgp-policy
spec:
  nodeSelector:
    matchLabels:
      bgp-enabled: "true"
  virtualRouters:
    - localASN: 64512
      neighbors:
        - peerASN: 64513
```

```
      peerAddress: 172.18.0.1
exportPodCIDR: true
```

This approach allows the cluster to scale without manual configuration. Nodes that join or leave the cluster update the routing fabric automatically.

Native routing works well when the underlying network can carry PodCIDRs directly or when you can extend the routing fabric with static routes or BGP. But in many environments, especially in cloud or multi-tenant infrastructures, you do not have the ability to influence the underlay. In those cases, Cilium cannot rely on the network to forward pod IPs.

To solve this, Cilium provides *encapsulation mode*, where pod traffic is tunneled between nodes. From the underlay's perspective this is just ordinary node-to-node traffic, but the original pod packets are preserved inside the tunnel. Prefer native routing if you can - encapsulation mode does come with a small performance penalty, due to the packet encapsulation overhead.

Encapsulation Mode

Encapsulation mode is also known as tunnel or overlay mode. In this mode, all nodes form a mesh of overlay tunnels using a UDP-based encapsulation protocol such as VXLAN or Geneve. **This is Cilium's default routing mode.**

Encapsulation is most useful in environments where you cannot control the underlay routing. Instead of requiring the physical network to carry pod prefixes, Cilium hides pod addressing behind node IPs. From the underlay's perspective, all inter-pod traffic looks like node-to-node traffic between Kubernetes nodes.

Encapsulation Mode with VXLAN

Before diving into how Cilium uses it, let's briefly define VXLAN. VXLAN (Virtual eXtensible LAN) is a tunnelling protocol that encapsulates Layer 2 Ethernet frames inside UDP packets, allowing networks to extend Layer 2 segments across Layer 3 boundaries**.** It was designed to overcome the scalability limits of VLANs by supporting up to 16 million unique identifiers (VNIs), compared to the 4096 VLAN IDs supported by traditional Ethernet.

In practical terms, VXLAN provides a way to build overlay networks on top of an existing IP underlay. The underlay only needs to deliver UDP traffic between nodes, while the overlay preserves the original Ethernet and IP headers of pod-to-pod traffic.

When Cilium is installed in VXLAN mode, a `cilium_vxlan` device appears on each node:

```
root@kind-worker:/# ip address show cilium_vxlan
14: cilium_vxlan: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
65520 qdisc noqueue state UNKNOWN group default
    link/ether c2:6e:2a:c2:91:42 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::c06e:2aff:fec2:9142/64 scope link
       valid_lft forever preferred_lft forever
```

Routes to remote PodCIDRs point to this device. Packets destined for another node's PodCIDR are encapsulated and sent via the VXLAN interface, as illustrated in Figure 5-5.

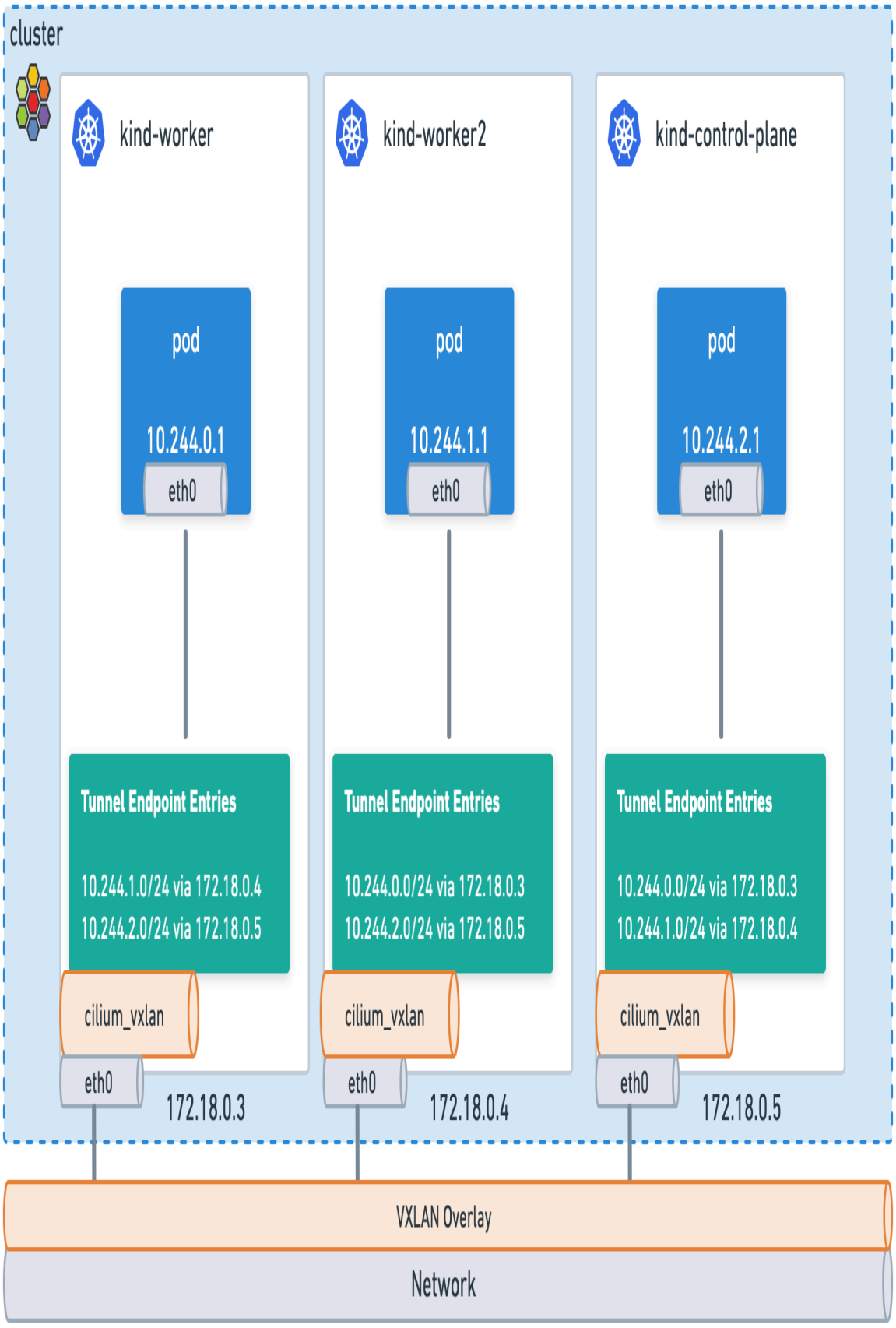

cluster
kind-worker
kind-worker2
kind-control-plane
pod
10.244.0.1
eth0
pod
10.244.1.1
eth0
pod
10.244.2.1
eth0
Tunnel Endpoint Entries
10.244.1.0/24 via 172.18.0.4
10.244.2.0/24 via 172.18.0.5
Tunnel Endpoint Entries
10.244.0.0/24 via 172.18.0.3
10.244.2.0/24 via 172.18.0.5
Tunnel Endpoint Entries
10.244.0.0/24 via 172.18.0.3
10.244.1.0/24 via 172.18.0.4
cilium_vxlan
eth0
172.18.0.3
cilium_vxlan
eth0
172.18.0.4
cilium_vxlan
eth0
172.18.0.5
VXLAN Overlay
Network

Figure 4-5. Encapsulation Routing Mode

You can verify using a generic kind cluster (`kind create cluster --config=kind-cluster-config.yaml`), with a default Cilium installation (`cilium install`),

The routing table is updated to direct PodCIDRs via the `cilium_host` device. For example:

```
$ docker exec kind-worker ip route
[...]
10.244.0.0/24 via 10.244.2.63 dev cilium_host proto kernel
src 10.244.2.63 mtu 65470
10.244.1.0/24 via 10.244.2.63 dev cilium_host proto kernel
src 10.244.2.63 mtu 65470
10.244.2.0/24 via 10.244.2.63 dev cilium_host proto kernel
src 10.244.2.63
[...]
```

Similar routes are installed on the other nodes, always pointing PodCIDRs to the local `cilium_host`.

If this looks familiar, it should. We saw a similar pattern in native routing mode with auto route injection: Cilium updated the host routing table with PodCIDRs and directed them to the appropriate node IPs. The difference here is that in VXLAN mode the routes point to the local `cilium_host` device. From there, Cilium intercepts the traffic, encapsulates it, and sends it through the `cilium_vxlan` interface to the destination node.

Figure 5-6 illustrates the packet format:

- The outer IP header holds the node IPs (for example 172.18.0.5 → 172.18.0.3).
- The outer UDP header identifies the tunnel. Cilium uses UDP port 8472 by default, although the IANA-assigned VXLAN port is 4789.

- The VXLAN header contains the VNI. In traditional VXLAN, the VNI is used to distinguish overlay networks. In Cilium, the VNI is repurposed to carry the identity of the source pod. It provides a minor performance boost as it allows the destination node to enforce policy without performing a separate identity lookup.
- The inner packet is the original pod-to-pod traffic, preserved unmodified.

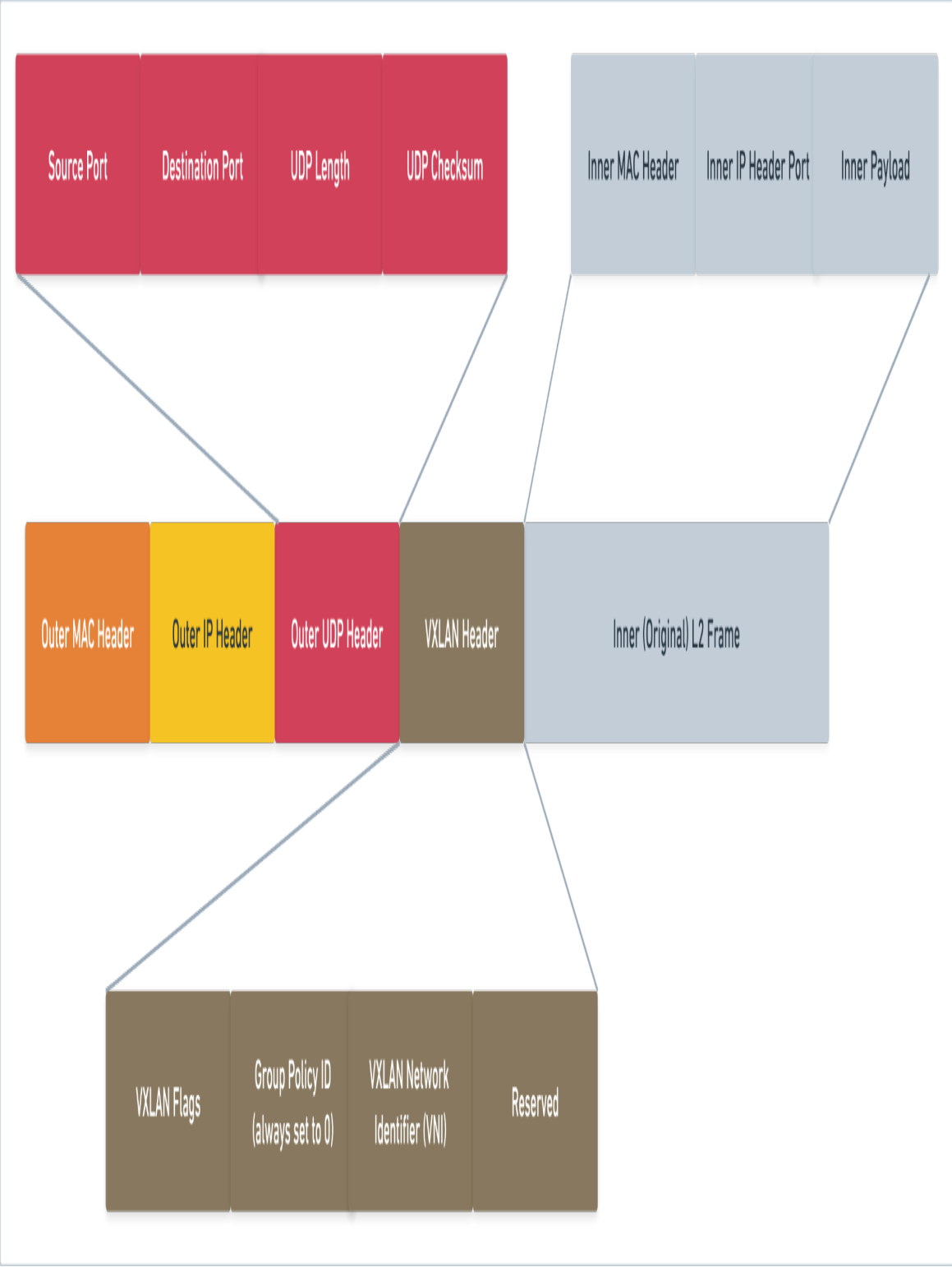


Figure 4-6. VXLAN encapsulation

You can observe and analyze this with tcpdump and Wireshark (Figure 5-7). You can find a sample `.pcap` capture (*vxlan_traffic.pcap*) in the `/pcap` directory of the GitHub repository. Note that the source and destination IPs (`172.18.0.5` and `172.18.0.3`) in the outer IP header are `kind-worker` and `kind-worker2`'s IP addresses respectively.

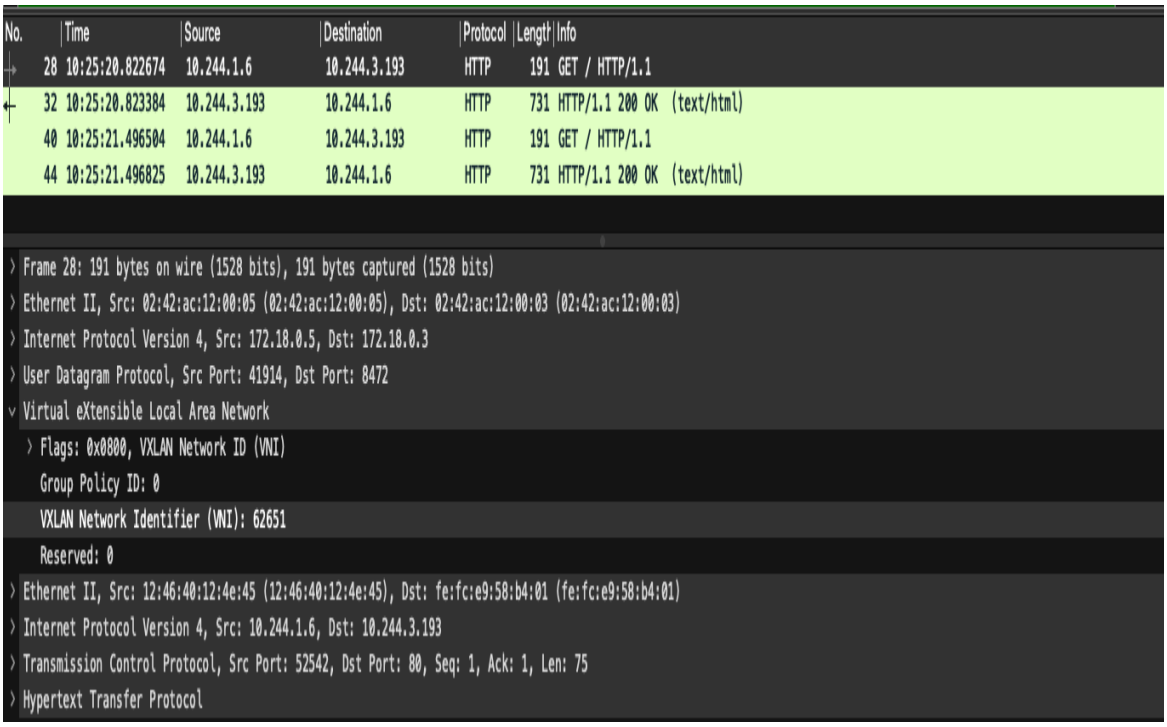


Figure 4-7. Wireshark Output (VXLAN)

From the network's point of view, this is just inter-node traffic. The nodes do not need to be on the same subnet, and they do not need to understand or route pod IPs.

Once the packet reaches the destination node, Cilium removes the VXLAN header and delivers the original packet to the receiving pod.

The key benefit of the encapsulation mode is that it abstracts the pod network from the underlying infrastructure. As long as nodes can reach each other over IP, pod-to-pod communication will work. The downside is the encapsulation overhead. Each packet includes an additional VXLAN header and outer IP/UDP headers. This adds

approximately 50 bytes per packet (depending on configuration), which reduces effective MTU and can impact performance.

For most clusters, this overhead is acceptable. But in performance-sensitive environments, native routing is typically preferred when possible.

Encapsulation Mode with Geneve

Like VXLAN, Geneve (Generic Network Virtualization Encapsulation) lets Cilium tunnel pod traffic between nodes when the underlay cannot route PodCIDRs. The effect is the same: the underlay only sees node-to-node IP traffic while the original pod packet is preserved inside the tunnel.

Unlike VXLAN, its header supports custom TLVs (Type-Length-Value) fields. TLVs allow Geneve to carry extra metadata in the tunnel. This extensibility makes Geneve a good fit for advanced use cases like Direct Server Return (DSR).

> **NOTE**
>
> We will cover DSR and other performance networking features in Chapter 8.

To test Geneve, we will use another managed Kubernetes service: Azure Kubernetes Service (AKS). This gives you the opportunity to try Cilium in a different cloud environment. The choice of AKS is deliberate because it offers a convenient and flexible networking mode: *Bring your own Container Network Interface* (BYOCNI). In this mode, the cluster is deployed without a CNI, leaving you free to install the CNI of your choice, in this case Cilium.

> **WARNING**
>
> As mentioned in Chapter 4, keep in mind that creating clusters and infrastructure in cloud environments may incur costs. Be sure to clean up any resources when you're done.

To deploy an AKS cluster, we'll use the Azure CLI ("az"). Follow the documentation to install it, sign in with `az login` and then use the following instructions to create a managed cluster in AKS (*"aks-byocni.md"*). In this example we use names prefixed with `geneve-` to make their purpose clearer.

```
az group create -l canadacentral -n geneve-rg <1>
az network vnet create \ <2>
  -g geneve-rg \
  --location canadacentral \
  --name geneve-vnet \
  --address-prefixes 192.168.8.0/22 \
  -o none

az network vnet subnet create \ <3>
  -g geneve-rg \
  --vnet-name geneve-vnet \
  --name geneve-subnet \
  --address-prefixes 192.168.10.0/24 \
  -o none

SUBNET_ID=$(az network vnet subnet show \ <4>
  --resource-group geneve-rg \
  --vnet-name geneve-vnet \
  --name geneve-subnet \
  --query id -o tsv)

az aks create \ <5>
  -l canadacentral \
  -g geneve-rg \
  -n geneve-cluster \
  --network-plugin none \
  --vnet-subnet-id "$SUBNET_ID"
```

```
az aks get-credentials --resource-group geneve-rg --name
geneve-cluster <6>
```

What this set of commands does:

1. Creates a new Azure resource group in the Canada Central region (customize to the region of your choice).
2. Sets up a virtual network (`geneve-vnet`) with a /22 address space.
3. Creates a subnet (`geneve-subnet`) within the VNet using a /24 prefix.
4. Dynamically retrieves the subnet ID.
5. Provisions an AKS cluster (`geneve-cluster`) in BYOCNI mode (indicated by the `--network-plugin none` flag)
6. Fetches the kubeconfig for the created cluster.

Once the cluster is ready, you can install Cilium in Geneve mode with the following Helm values (`cilium-geneve.yaml`).

```
aksbyocni:
  enabled: true
ipam:
  mode: cluster-pool

tunnelProtocol: geneve
```

Once the installation is completed (verify until `cilium status --wait` successfully completes), check the cluster nodes' IP addresses:

```
$ kubectl get nodes -o wide
NAME                                INTERNAL-IP
aks-nodepool1-30411986-vmss000000   192.168.10.4
```

```
aks-nodepool1-30411986-vmss000001   192.168.10.6
aks-nodepool1-30411986-vmss000002   192.168.10.5
```

Next, confirm which PodCIDRs have been allocated to each node. This is available from the CiliumNode objects.

```
$ kubectl get ciliumnodes -o yaml | grep -C 1 CIDR
    ipam:
      podCIDRs:
      - 10.0.0.0/24
--
    ipam:
      podCIDRs:
      - 10.0.1.0/24
--
    ipam:
      podCIDRs:
      - 10.0.2.0/24
```

Deploy a set of pods (`netshoot-deployment.yaml`) and verify that they get an IP address from the expected pools:

```
$ kubectl get pods -o wide
NAME                          READY   STATUS    RESTARTS
AGE   IP           NODE
pod-worker-54f8b77c97-fm22b   1/1     Running   0
63s   10.0.0.56    aks-nodepool1-39039968-vmss000000
pod-worker-54f8b77c97-kxcr9   1/1     Running   0
63s   10.0.0.170   aks-nodepool1-39039968-vmss000000
pod-worker-54f8b77c97-pbgv6   1/1     Running   0
63s   10.0.0.249   aks-nodepool1-39039968-vmss000000
```

First, let's verify inter-node connectivity between pods to confirm that Geneve encapsulation is working:

```
$ kubectl exec -it pod-worker-54f8b77c97-fm22b -- ping
10.0.0.249
PING 10.0.0.249 (10.0.0.249) 56(84) bytes of data.
64 bytes from 10.0.0.249: icmp_seq=1 ttl=63 time=0.129 ms
64 bytes from 10.0.0.249: icmp_seq=2 ttl=63 time=0.064 ms
```

```
64 bytes from 10.0.0.249: icmp_seq=3 ttl=63 time=0.066 ms
64 bytes from 10.0.0.249: icmp_seq=4 ttl=63 time=0.064 ms
^C
--- 10.0.0.249 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time
3077ms
rtt min/avg/max/mdev = 0.064/0.080/0.129/0.027 ms
```

When using native routing with auto route injection or VXLAN, we explained earlier how PodCIDR routes are installed in the node's routing table and point to the destination node's IP address. The same applies in Geneve mode, but in this case the mapping can be verified by checking the Cilium agent. For example, on the agent located on `aks-***-vmss000002`, the route to 10.0.0.0/24 (the PodCIDR on `aks-***-vmss000000`) maps to the tunnel bound to the node's IP address (192.168.10.4).

```
$ kubectl -n kube-system exec -it cilium-8z49v -- cilium-
dbg bpf tunnel list
TUNNEL      VALUE
10.0.0.0    192.168.10.4:0
10.0.2.0    192.168.10.5:0
```

Similar to VXLAN, and as illustrated in Figure 5-8, the original pod packet is wrapped inside a UDP packet for transmission between nodes.

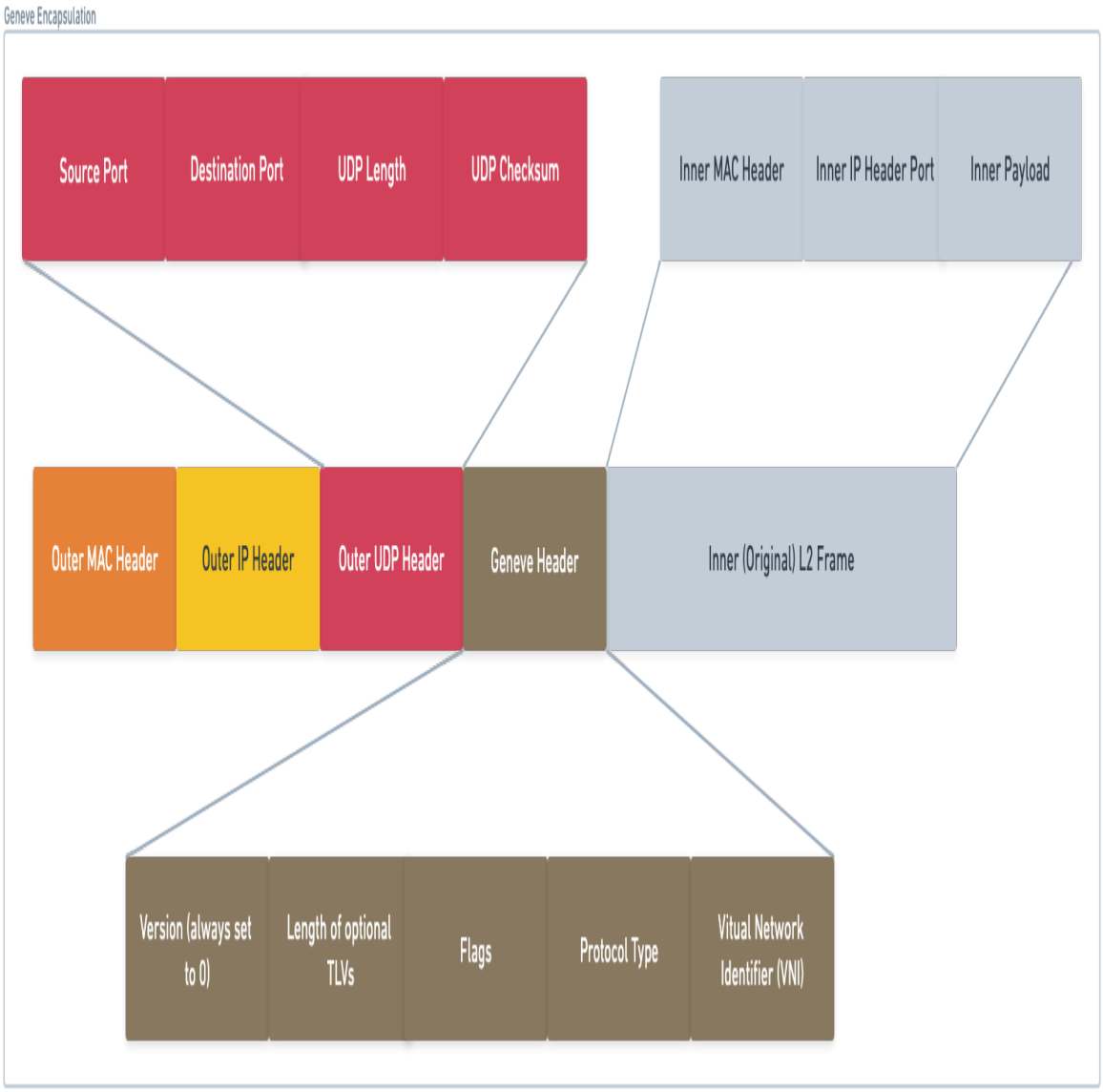


Figure 4-8. Geneve encapsulation

The main differences between VXLAN and Geneve are the UDP port and the ability to use TLVs. Cilium uses UDP port **6081** for Geneve, compared to 8472 for VXLAN.

Most of the Geneve header fields remain fixed in Cilium's implementation:

- **Version** is always 0
- **Flags** are set to 0x00

- **Protocol Type** is usually 0x6558, meaning Ethernet

Two fields are more interesting to examine:

- **Options Length**: this indicates whether any TLVs are present. In our example it is 0, meaning no TLVs are used. In Chapter 8 you will see how Geneve TLVs enable advanced features such as Direct Server Return.
- **Virtual Network Identifier (VNI)**: just like in VXLAN, the VNI in Geneve is used by Cilium to carry the identity of the source pod. This allows the destination node to enforce network policy without having to perform a separate identity lookup.

This can be verified by capturing the traffic with tcpdump and analyzing it with Wireshark once again. You'll also find a sample packet capture (*geneve.pcap*) in the `/pcap` folder on Github.

As you can see in Figure 5-9, the Generic Network Virtualization Encapsulation (Geneve) field is populated with the fields previously described. The VNI is set to 23129.

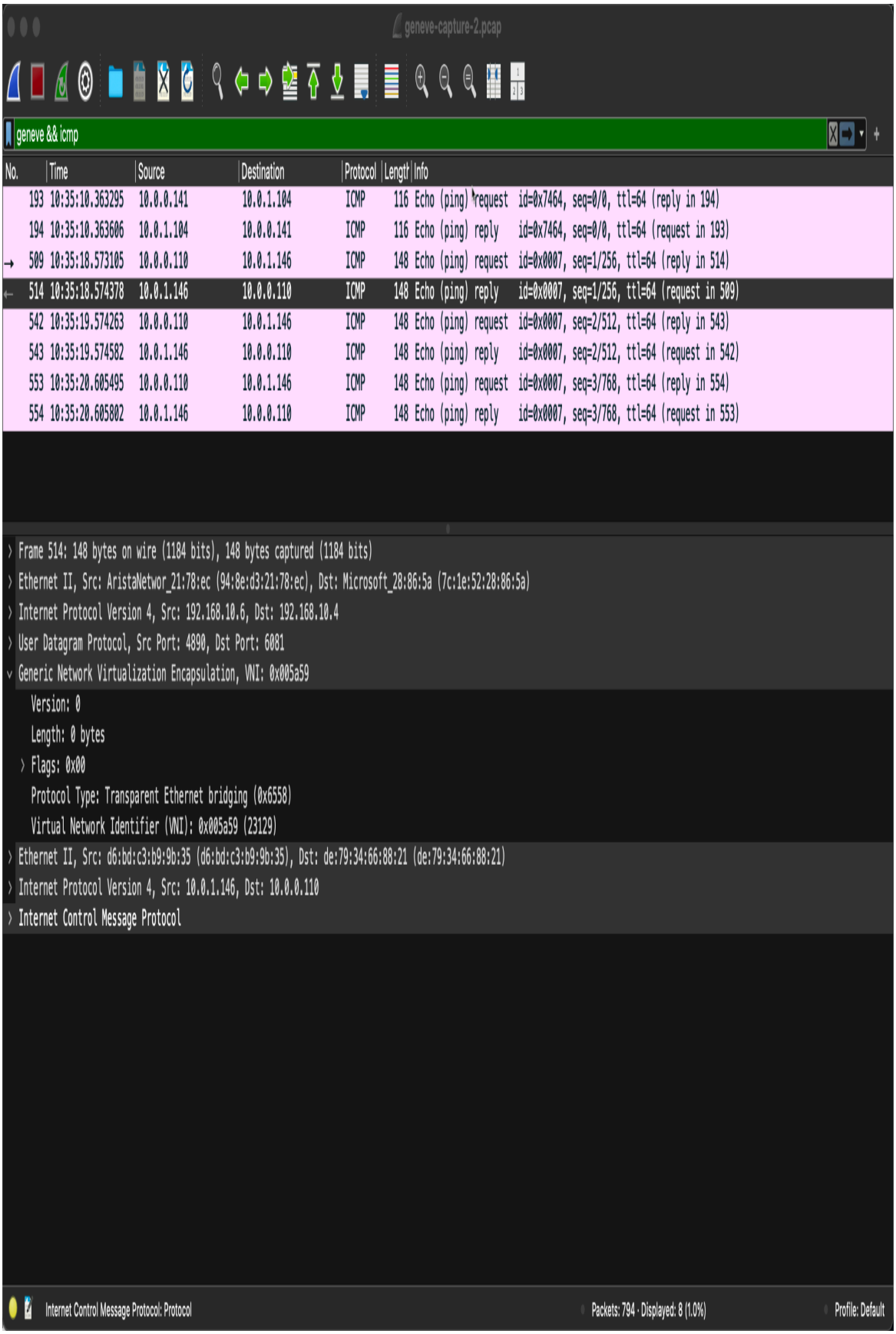

geneve-capture-2.pcap
geneve && icmp
No. Time Source Destination Protocol Length Info
193 10:35:10.363295 10.0.0.141 10.0.1.104 ICMP 116 Echo (ping) request id=0x7464, seq=0/0, ttl=64 (reply in 194)
194 10:35:10.363606 10.0.1.104 10.0.0.141 ICMP 116 Echo (ping) reply id=0x7464, seq=0/0, ttl=64 (request in 193)
509 10:35:18.573105 10.0.0.110 10.0.1.146 ICMP 148 Echo (ping) request id=0x0007, seq=1/256, ttl=64 (reply in 514)
514 10:35:18.574378 10.0.1.146 10.0.0.110 ICMP 148 Echo (ping) reply id=0x0007, seq=1/256, ttl=64 (request in 509)
542 10:35:19.574263 10.0.0.110 10.0.1.146 ICMP 148 Echo (ping) request id=0x0007, seq=2/512, ttl=64 (reply in 543)
543 10:35:19.574582 10.0.1.146 10.0.0.110 ICMP 148 Echo (ping) reply id=0x0007, seq=2/512, ttl=64 (request in 542)
553 10:35:20.605495 10.0.0.110 10.0.1.146 ICMP 148 Echo (ping) request id=0x0007, seq=3/768, ttl=64 (reply in 554)
554 10:35:20.605802 10.0.1.146 10.0.0.110 ICMP 148 Echo (ping) reply id=0x0007, seq=3/768, ttl=64 (request in 553)
Frame 514: 148 bytes on wire (1184 bits), 148 bytes captured (1184 bits)
Ethernet II, Src: AristaNetwor_21:78:ec (94:8e:d3:21:78:ec), Dst: Microsoft_28:86:5a (7c:1e:52:28:86:5a)
Internet Protocol Version 4, Src: 192.168.10.6, Dst: 192.168.10.4
User Datagram Protocol, Src Port: 4890, Dst Port: 6081
Generic Network Virtualization Encapsulation, VNI: 0x005a59
Version: 0
Length: 0 bytes
Flags: 0x00
Protocol Type: Transparent Ethernet bridging (0x6558)
Virtual Network Identifier (VNI): 0x005a59 (23129)
Ethernet II, Src: d6:bd:c3:b9:9b:35 (d6:bd:c3:b9:9b:35), Dst: de:79:34:66:88:21 (de:79:34:66:88:21)
Internet Protocol Version 4, Src: 10.0.1.146, Dst: 10.0.0.110
Internet Control Message Protocol
Internet Control Message Protocol: Protocol
Packets: 794 · Displayed: 8 (1.0%)
Profile: Default

Figure 4-9. Geneve Wireshark Capture

You can verify that this is the identity of the source pod by checking the Cilium identities with `kubectl get ciliumid`:

```
$ kubectl get ciliumid 23129 -o yaml
apiVersion: cilium.io/v2
kind: CiliumIdentity
metadata:
[...]
  labels:
    app: pod-worker
    io.cilium.k8s.policy.cluster: default
    io.cilium.k8s.policy.serviceaccount: default
    io.kubernetes.pod.namespace: default
  name: "23129"
[...]
security-labels:
  k8s:app: pod-worker
[...]
```

Conclusion

Understanding how packets move between pods (both within the same node and across different nodes) is fundamental to mastering Kubernetes networking with Cilium. In this chapter, we traced a packet from one pod to another, examined how Cilium leverages Linux primitives like veth pairs, and explored the role of Cilium in forwarding decisions.

We looked at two main models of inter-node connectivity:

- Native routing mode, where the underlying network must be aware of PodCIDRs, and
- Encapsulation mode, where Cilium builds a tunnel mesh using VXLAN or Geneve to abstract the pod network from the infrastructure.

We explained how each approach has trade-offs in terms of performance, simplicity and control.

In the next chapter, we'll turn our attention to Kubernetes Service Networking: how they work, how Cilium handles service discovery, load balancing, and proxying.

Chapter 5. Service Networking

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo is available at *https://github.com/isovalent/cilium-up-and-running*.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *gobrien@oreilly.com*.

In Kubernetes, pod-to-pod communication enables direct connectivity between workloads but this model alone does not scale well for dynamic, distributed applications. Instead, most real-world applications interact through Services, which provide stable virtual IPs and load balancing across dynamic backend pods. This abstraction is critical for scalability, reliability and operational simplicity in distributed systems.

Service networking sits at the core of how applications discover and reach each other in a cluster. It enables decoupling of workloads, supports horizontal scaling and provides mechanisms for high availability.

In this chapter, we will examine how Kubernetes manages service communication, the traditional role of `kube-proxy`, and how

Cilium enhances and ultimately replaces it with an eBPF-based datapath.

Along the way, we will explore concepts such as headless Services, session affinity, and external access through LoadBalancer Services,

Kubernetes Service Refresher

To better understand some of the principles referred to in later sections of this book, it's useful to get a brief refresher on Kubernetes Services. This is not meant to be a comprehensive explanation, given that there are various books dedicated to this topic already (starting with O'Reilly's "Networking and Kubernetes" and "Container Networking"). Instead, it serves as a concise refresher of the concepts most relevant to service communication.

Kubernetes is built for ephemeral workloads. New pods are constantly created while old ones are removed. The IP addresses assigned to pods are not reserved: once a pod terminates, the cluster returns its IP to the pool and may reassign it to another pod. Because of this, relying solely on IPs for connectivity in Kubernetes is impractical. Instead, we typically use Services and DNS to provide stable names and a virtual IP in front of a set of ready pods.

A Service defines a consistent way for clients to reach a group of pods. It acts as a tiny virtual load balancer that targets pods with specific labels. Traffic will be load-balanced randomly between pods marked as *ready*, according to the `readinessProbe`. Each Service is assigned a unique virtual IP address (VIP) address, that exists only inside the cluster.

Kubernetes supports several types of Services for different use cases:

- ClusterIP (default): provides an in-cluster VIP so that other pods can reach the Service.

- NodePort: exposes the Service on the same static port across all nodes, which allows basic external access. Chapter 10 will explain some of the limitations with NodePort and how to address them.
- LoadBalancer: provisions an external IP by integrating with a cloud provider's load balancer or with Cilium's LoadBalancer IPAM (LB-IPAM). We will cover external reachability and announcements in Chapter 7 (Ingress and Gateway API) and Chapter 10 (BGP).
- headless: skips allocating a VIP. DNS lookups return the individual pod IPs directly, which works well with StatefulSets where each pod keeps a stable identity.

Inside the cluster, Kubernetes Services rely on `kube-proxy`. Let's take a look at this component, its role and limitations before walking through how Cilium addresses them.

Kube-Proxy Refresher

This component is deployed as a DaemonSet so it runs on every node. Its job is to program the networking rules required to forward traffic from a Service VIP to one of its backend pods. To do this, kube-proxy watches for changes to Services and their backends, recorded as Endpoints or Endpointslices, and updates the node's networking tables accordingly. When a packet is addressed to a Service IP, the kernel applies one of these rules and performs Destination Network Address Translation (DNAT) to rewrite the destination address to a selected backend pod (as illustrated in Figure 6-1).

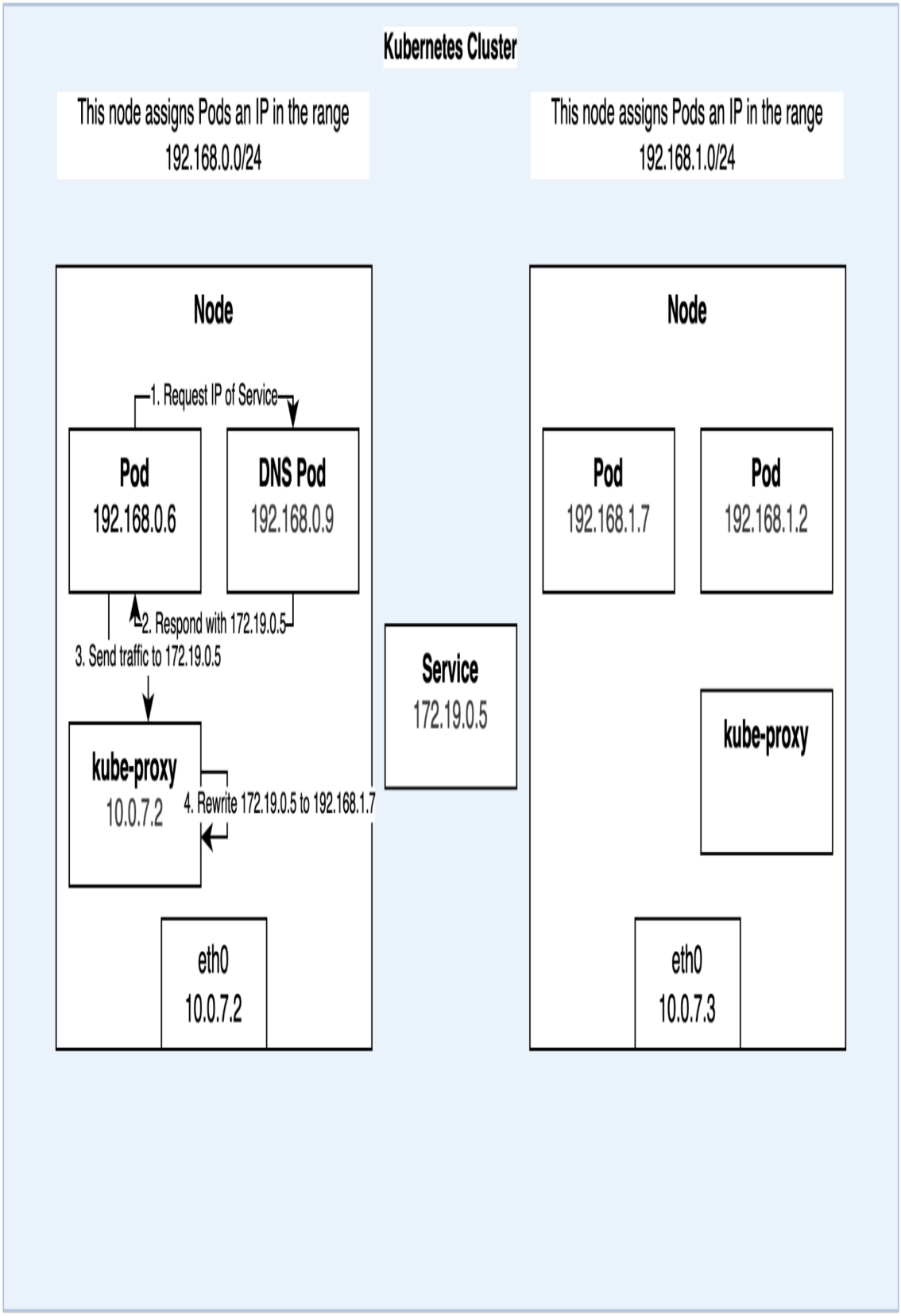

Kubernetes Cluster
This node assigns Pods an IP in the range 192.168.0.0/24
This node assigns Pods an IP in the range 192.168.1.0/24
Node
1. Request IP of Service
Pod
192.168.0.6
DNS Pod
192.168.0.9
2. Respond with 172.19.0.5
3. Send traffic to 172.19.0.5
kube-proxy
10.0.7.2
4. Rewrite 172.19.0.5 to 192.168.1.7
eth0
10.0.7.2
Service
172.19.0.5
Node
Pod
192.168.1.7
Pod
192.168.1.2
kube-proxy
eth0
10.0.7.3

Figure 5-1. Kubernetes Service Networking

`kube-proxy` supports multiple modes for programming these rules, and each mode influences how load-balancing decisions are made. In *iptables* mode, packets are matched against large chains of NAT rules, with a random selection from the available endpoints. *IPVS* introduces a richer set of algorithms such as round-robin or least connections. *nftables* provides a more modern alternative to *iptables* while offering similar functionality. Regardless of the backend, the outcome is the same: traffic sent to a single Service IP is distributed across the set of healthy pods.

For a long time, most clusters relied on the iptables backend. iptables was originally designed more than two decades ago for static firewalling, not for the highly dynamic nature of Kubernetes. Each time a pod is added or removed, the iptables rules must be rewritten, which can become extremely expensive at scale. A well-known KubeCon talk in 2017 described how updating iptables rules for 20,000 Services could take as long as five hours. Even without large updates, every packet must traverse the iptables chains one rule at a time, which is an O(n) lookup that grows linearly with the number of rules. These factors combine to increase latency, CPU usage, and operational complexity as clusters expand.

Demonstrating iptables Rule Explosion

To understand why iptables does not scale well as a backend for `kube-proxy`, we can create a large number of Services and observe how many rules get inserted into the kernel. Recall that `kube-proxy` installs NAT rules for every Service and every backend endpoint. As the number of Services grows, so does the size of the iptables chains.

Let's try on a kind cluster. This time though, let's not use Cilium yet but the built-in CNI for `kind` clusters (kindnet). Let's also use the

iptables-based kube-proxy (`kind-cluster-config-no-cilium.yaml`):

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
- role: worker
networking:
  kubeProxyMode: iptables
```

We start with a simple Deployment (httpd-deployment.yaml) and Service (httpd-service.yaml) for httpd:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpd
spec:
  selector:
    matchLabels:
      app: httpd
  template:
    metadata:
      labels:
        app: httpd
    spec:
      containers:
      - name: httpd
        image: httpd
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: httpd
spec:
  type: ClusterIP
  selector:
```

```
    app: httpd
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
```

Once this service is deployed, you can check directly on the worker nodes and the raw iptables rules yourself:

```
$ docker exec -it kind-worker iptables -t nat -L -n --line-
numbers
[...]
Chain KUBE-SERVICES (2 references)
num  target     prot opt source               destination
1    KUBE-SVC-ERIFXISQEP7F7OF4  6    --  0.0.0.0/0
10.96.0.10           /* kube-system/kube-dns:dns-tcp
cluster IP */ tcp dpt:53
2    KUBE-SVC-JD5MR3NA4I4DYORP  6    --  0.0.0.0/0
10.96.0.10           /* kube-system/kube-dns:metrics
cluster IP */ tcp dpt:9153
3    KUBE-SVC-TMH6CLKYL4UVJZSU  6    --  0.0.0.0/0
10.96.1.252          /* default/httpd cluster IP */ tcp
dpt:80
4    KUBE-SVC-NPX46M4PTMTKRN6Y  6    --  0.0.0.0/0
10.96.0.1            /* default/kubernetes:https cluster IP
*/ tcp dpt:443
5    KUBE-SVC-TCOU7JCQXEZGVUNU  17   --  0.0.0.0/0
10.96.0.10           /* kube-system/kube-dns:dns cluster IP
*/ udp dpt:53
6    KUBE-NODEPORTS  0    --  0.0.0.0/0
0.0.0.0/0            /* kubernetes service nodeports; NOTE:
this must be the last rule in this chain */ ADDRTYPE match
dst-type LOCAL
```

You will observe that, even with a single service, multiple rules have been added into the NAT tables.

Let's generate 100 services pointing to the httpd deployment, with this short `for` loop.

```
$ for i in {1..100}; do
  kubectl expose deployment httpd --port=80 --name=httpd-$i
done

service/httpd-1 exposed
service/httpd-2 exposed
[...]
service/httpd-99 exposed
service/httpd-100 exposed
```

Count the total numbers of rules created with `grep`:

```
$ docker exec kind-worker iptables-save | grep KUBE-SEP |
wc -l
1240
```

This illustrates the problem with iptables: every new Service requires more rules, each rule must be evaluated linearly, and updating rulesets becomes increasingly expensive.

This is one of the main reasons to consider replacing `kube-proxy`. Cilium's implementation, which we will examine next, uses eBPF maps in the kernel: a far more efficient way to store and update Service state.

Kube-Proxy Replacement

Cilium approaches service connectivity differently. Let's examine Cilium's eBPF-based kube-proxy replacement, commonly referred to as KPR.

Rather than relying on iptables, it uses eBPF maps in the Linux kernel to store connection tracking and load-balancer state. Both looking up an entry in a hash table and inserting a new one are approximately O(1) operations, which means they scale much, much better.[1]

When enabling KPR, Cilium loads eBPF programs into the Linux kernel on every node. These programs attach to key networking interfaces, as described in Chapter 4, and intercept packets destined for a Service. Instead of relying on long iptables chains, the program performs a lookup in an eBPF map that stores the current set of backends for that Service. It then selects one of the available pods according to the load-balancing policy, rewrites the destination address (DNAT), and forwards the packet, as illustrated in Figure 6-2.

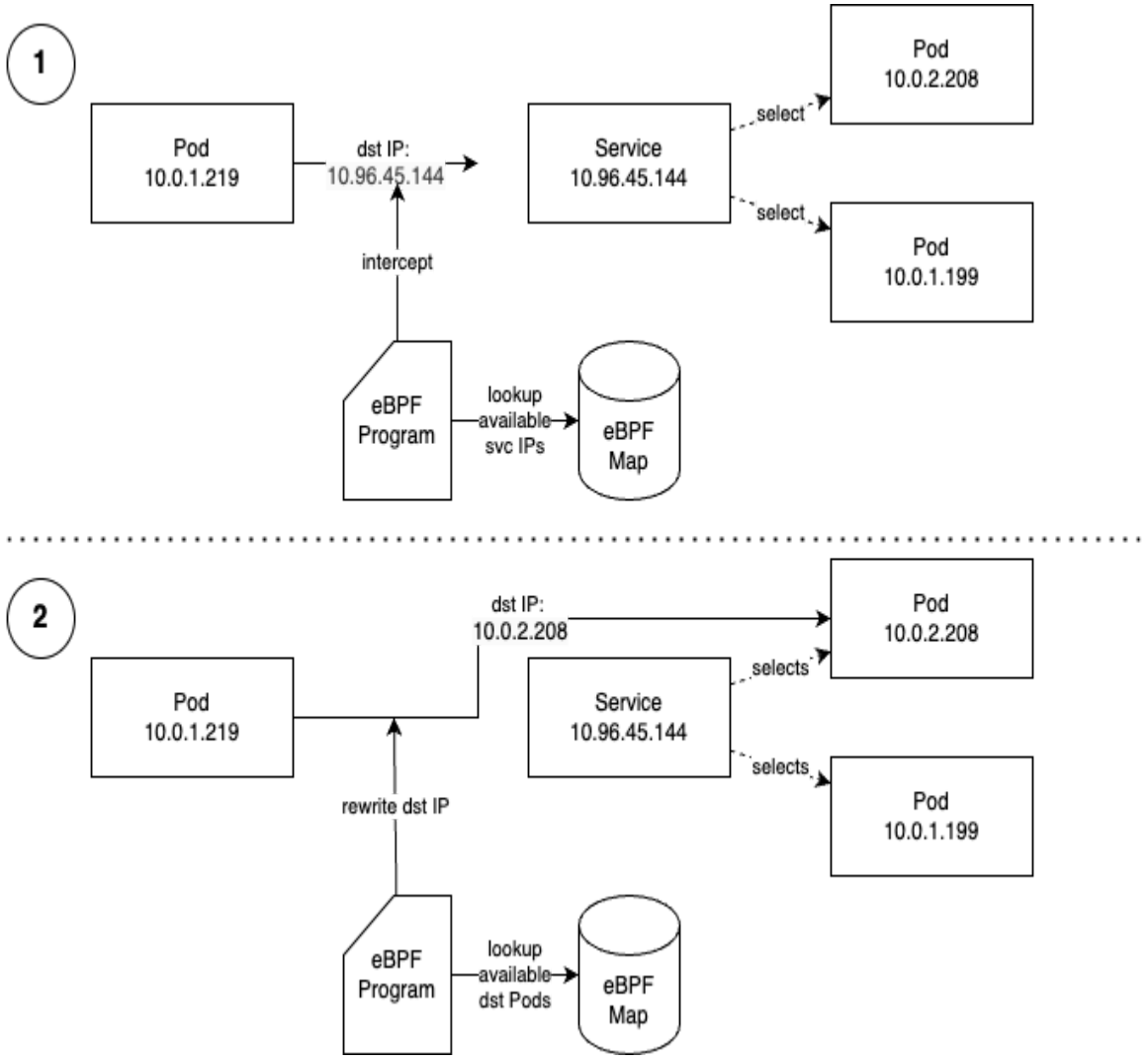


Figure 5-2. Cilium's eBPF-based Kube-Proxy Replacement

Let's demonstrate this in practice, Deploy a kind cluster, using the following kind configuration (`kind-cluster-config.yaml`). Note how we are creating a cluster without the built-in kube-proxy.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
- role: worker
networking:
  disableDefaultCNI: true
  kubeProxyMode: "none"
```

Then, install Cilium in KPR mode. When using Helm, you'll need to specify the Kubernetes API Server IP and port (`cilium-kpr-values.yaml`):

```
kubeProxyReplacement: true
k8sServiceHost: kind-control-plane
k8sServicePort: 6443
```

Let's explain briefly why these settings are needed. When kube-proxy is in place, it handles service traffic towards Kubernetes services, and that includes the Kubernetes API server itself. Pods can connect to kubernetes.default.svc and kube-proxy will DNAT the Service IP (usually 10.96.0.1 in kind) to the actual API Server endpoint.

But given that we deployed a cluster without kube-proxy, there's no component to program the rules that let the Cilium agent and operator talk to the API server through the service IP. To avoid a chicken-and-egg problem, Cilium needs to know how to contact the API server directly bypassing the service abstractions.

Which is why you need to set `k8sServiceHost` (and optionally, `k8sServicePort`) in the Helm values when enabling KPR. When

using kind, we can just use `kind-control-plane` as that hostname is resolvable from the worker containers.

One advantage of using Cilium CLI in this instance is that it can detect the Kubernetes API Server IP address and port automatically from the kubeconfig:

```
$ cilium install --set kubeProxyReplacement=true
🔮 Auto-detected Kubernetes kind: kind
ℹ️  Using Cilium version 1.17.5
🔮 Auto-detected cluster name: kind-kind
ℹ️  Detecting real Kubernetes API server addr and port on
Kind
🔮 Auto-detected kube-proxy has not been installed
ℹ️  Cilium will fully replace all functionalities of kube-
proxy
```

Let's first verify that KPR was enabled by checking the detailed configuration status with the Cilium agent's `cilium-dbg` binary (introduced in Chapter 2):

```
$ kubectl -n kube-system exec ds/cilium -- cilium-dbg
status --verbose
[...]
KubeProxyReplacement Details:
  Status:                 True ❶
  Socket LB:              Enabled
  Socket LB Tracing:      Enabled
  Socket LB Coverage:     Full
  Devices:                eth0    172.18.0.3
fc00:f853:ccd:e793::3 fe80::bc40:6fff:feb9:deac (Direct
Routing) ❷
  Mode:                   SNAT ❸
  Backend Selection:      Random ❹
  Session Affinity:       Enabled ❺
  Graceful Termination:   Enabled
  NAT46/64 Support:       Disabled
  XDP Acceleration:       Disabled
  Services: ❻
  - ClusterIP:      Enabled
```

```
  - NodePort:       Enabled (Range: 30000-32767)
  - LoadBalancer:   Enabled
  - externalIPs:    Enabled
  - HostPort:       Enabled
  Annotations:
  - service.cilium.io/node
  - service.cilium.io/src-ranges-policy
  - service.cilium.io/type
[...]
```

❶ Confirms that KPR is active.

❷ Shows the network devices where Cilium has attached its eBPF programs, with (direct) native routing enabled on eth0. Note that KPR works for both native routing and encapsulation modes.

❸ Translation mode is SNAT, which means incoming packets are source NATed with the IPs of the nodes. We will see an alternative mode (DSR) in detail in Chapter 8.

❹ Backend selection uses random choice by default. For more consistent load balancing across nodes, Cilium also supports Maglev hashing, which we cover in Chapter 8.

❺ Session affinity is enabled. This ensures that packets from the same client IP can be steered to the same backend pod. We discuss session affinity later in this chapter.

❻ Lists the Service types that are implemented by KPR: ClusterIP, NodePort, LoadBalancer, externalIPs, and HostPort.

NOTE

KPR is a highly customizable feature. It exposes a wide range of options that control how Services are implemented in the datapath, including translation modes, load-balancing algorithms, traffic policies, and acceleration features. The user experience is due to be simplified in future releases so that common configurations require fewer manual settings. Covering every option in detail is beyond the scope of this book. In this chapter, we will focus on the most important features that are relevant for day-to-day use, while pointing you to later chapters for advanced topics such as Direct Server Return, XDP acceleration, and Maglev load balancing. For full details, see the Cilium documentation.

Let's create the same httpd Deployment (httpd-deployment.yaml) and Service (httpd-service.yaml) we used in the previous section. Take note of the pod IPs, as well as the service IP:

```
$ kubectl get pods -o wide
NAME                     READY   STATUS    RESTARTS   AGE
IP           NODE
httpd-77b5fcff59-k6svw   1/1     Running   0          22s
10.0.1.199   kind-worker2
httpd-77b5fcff59-vfnxt   1/1     Running   0          22s
10.0.2.208   kind-worker
$ kubectl get service
NAME    TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
httpd   ClusterIP   10.96.45.144   <none>        80/TCP
17s
```

First, let's just check that there's no iptables rules created to handle traffic for this service, by running the same command we did in the previous step:

```
$ docker exec kind-worker iptables-save | grep KUBE-SEP |
wc -l
0
```

Instead, eBPF entries are populated with the healthy (active) backends. Let's use `cilium-dbg` to inspect current Services and their backends:

```
$ kubectl exec -n kube-system -it ds/cilium -c cilium-agent
-- cilium-dbg service list
ID   Frontend               Service Type   Backend
[...]
6    10.96.130.157:80/TCP   ClusterIP      1 =>
10.244.1.105:80/TCP (active)
```

Now, let's launch a client Pod (netshoot-client.yaml).

```
$ kubectl apply -f netshoot-client.yaml
pod/netshoot-client created
```

From the client pod (`kubectl exec -it netshoot-client -- bash`), let's send traffic to the Service IP (`10.96.45.144`) and then inspect `/proc/<pid>/net/tcp` immediately afterward.

```
$ curl 10.96.45.144 && cat /proc/$$/net/tcp
<html><body><h1>It works!</h1></body></html>
  sl  local_address rem_address   st tx_queue rx_queue tr
tm->when retrnsmt   uid  timeout inode
   0: DB01000A:D316 D002000A:0050 06 00000000:00000000
03:0000176F 00000000     0        0 0 3 0000000000000000
```

The output may look overwhelming at first, so let's walk through it step by step.

When we run `curl 10.96.45.144`, the request goes to the Service we just created. The HTML response `<html><body><h1>It works!</h1></body></html>` confirms that the request successfully reached one of the `httpd` pods. Traffic was forwarded appropriately to the right backend, even if there's no matching iptables rules.

On Linux, every process has a directory under `/proc/<pid>/` that contains details about that process, where `<pid>` refers to the process ID. The command `cat /proc/$$/net/tcp` lists the TCP connections for the current process (`$$` expands to the current process ID). In that file, the `rem_address` field shows the remote IP address. Instead of being displayed in a human-readable format, the address appears in hexadecimal.

For example, the value `D002000A` corresponds to the IPv4 address `D0.02.00.0A`. Splitting it into byte pairs gives us the four octets of the address. Converting them to decimal produces `208.2.0.10`. Because the kernel stores addresses in little-endian order (least significant byte first), we reverse the sequence to obtain the real address: `10.0.2.208`. This IP belongs to one of the `httpd` pods.

This confirms what is happening under the hood. Although we connected to the Service IP `10.96.45.144`, the eBPF program in the kernel rewrote the destination early in the networking stack, allowing us to observe the pod's actual IP as the backend. If we repeat the request, we may see another value such as `C701000A`, which translates to `10.0.1.199` - the second `httpd` pod.

```
$ curl 10.96.45.144 && cat /proc/$$/net/tcp
<html><body><h1>It works!</h1></body></html>
  sl  local_address rem_address   st tx_queue rx_queue tr
tm->when retrnsmt   uid  timeout inode
   0: DB01000A:ED76 D002000A:0050 06 00000000:00000000
03:000016D3 00000000     0        0 0 3 0000000000000000
   1: DB01000A:85B0 C701000A:0050 06 00000000:00000000
03:0000176F 00000000     0        0 0 3 0000000000000000
```

NOTE

The value `0050` is the port number in hexadecimal. Converting `0050` gives us 80, which is the default port for HTTP.

Finally, let's repeat the service scale test and verify there's still no iptables being created:

```
$ for i in {1..100}; do
  kubectl expose deployment httpd --port=80 --name=httpd-$i
done

service/httpd-1 exposed
service/httpd-2 exposed
[...]
service/httpd-99 exposed
service/httpd-100 exposed
$ docker exec kind-worker iptables-save | grep KUBE-SEP |
wc -l
0
```

Agent Availability and Datapath Resiliency

Cilium separates control plane work from datapath forwarding. The Cilium agent watches the Kubernetes API server and continuously watches for changes in pods and their readiness (Figure 6-3). It computes state, and writes Service and backend information into eBPF maps. eBPF programs in the Linux kernel read those maps to steer packets to pod backends. The programs themselves are stateless; they do not have any built-in knowledge of pods but rely entirely on the data maintained by the agent.

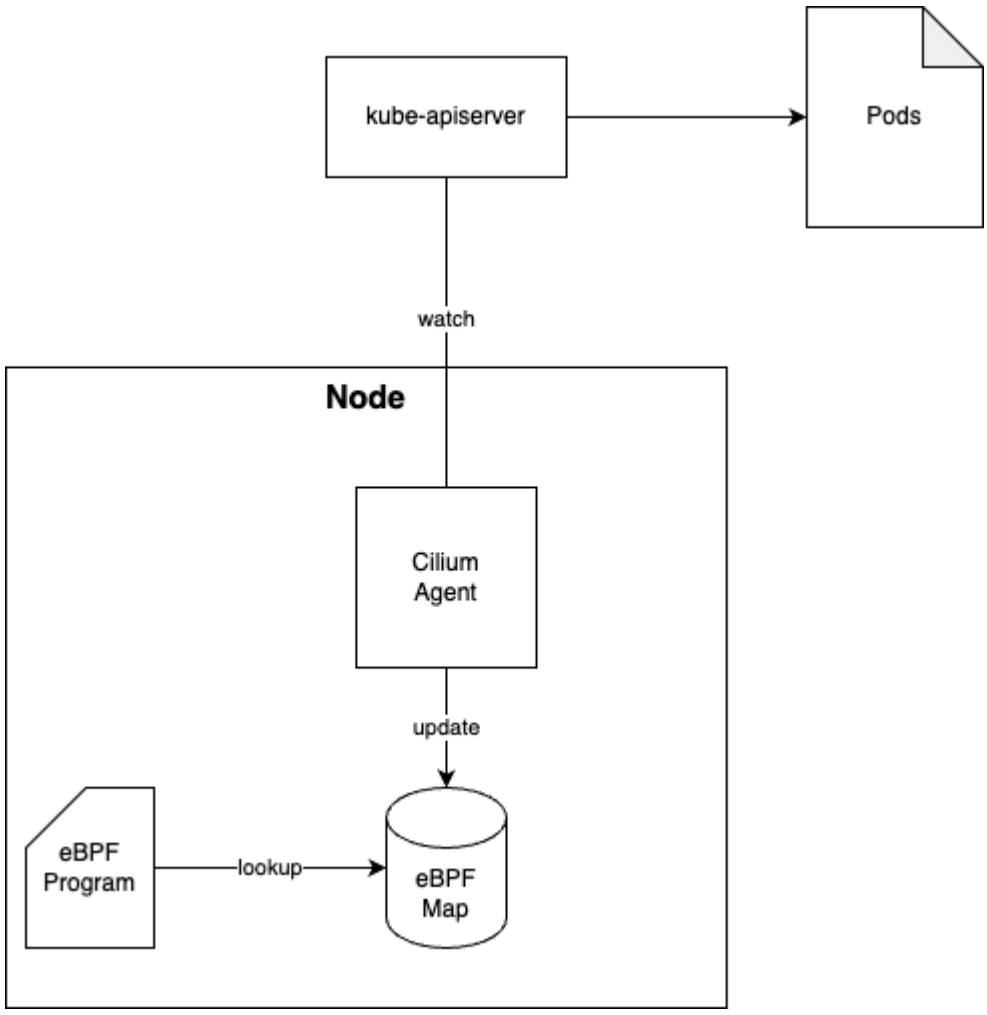


Figure 5-3. Split control and data planes

This architecture provides a significant advantage: the datapath itself continues to operate as long as the kernel state is intact. This design means that Service connectivity does not immediately break if the agent is restarted or temporarily unavailable.

This resiliency is important because rolling upgrades, node restarts, or other operational events should not disrupt applications. Since the eBPF programs and maps live in the kernel, packets continue to be forwarded while the agent restarts

Even if the Cilium agent crashes or is temporarily removed, Service connectivity can continue to function because the eBPF programs and maps remain in the kernel. To demonstrate this, we will

simulate the removal of the agent by modifying its DaemonSet. Setting the DaemonSet's `nodeSelector` to an unused label removes all Cilium agent pods from the cluster:

```
$ kubectl patch daemonset -n kube-system cilium --
type='merge' -p '{"spec":{"template":{"spec":
{"nodeSelector":{"foo":"bar"}}}}}'
daemonset.apps/cilium patched
```

As a result, there's no Cilium agent running anywhere anymore:

```
$ kubectl get daemonset -n kube-system cilium
NAME     DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE
NODE SELECTOR                     AGE
cilium   0         0         0       0            0
foo=bar,kubernetes.io/os=linux   15m
```

Despite the Cilium agent being absent, traffic to the Service still works:

```
$ kubectl exec -it netshoot-client -- curl httpd
<html><body><h1>It works!</h1></body></html>
```

This demonstrates the resiliency benefit: the datapath continues forwarding packets even though the control plane component has disappeared.

However, the design also has a limitation. While the agent is unavailable, eBPF maps are not updated to reflect changes in the cluster. If a pod crashes during this time, the map still contains its IP address, and the datapath continues to forward traffic to it (Figure 6-4).

```
$ kubectl delete pod httpd-77b5fcff59-vfnxt
pod "httpd-77b5fcff59-vfnxt" deleted
$ curl -m 2 10.96.45.144
curl: (28) Connection timed out after 2004 milliseconds
```

The timeout occurs because the datapath is still trying to send traffic to the failed backend.

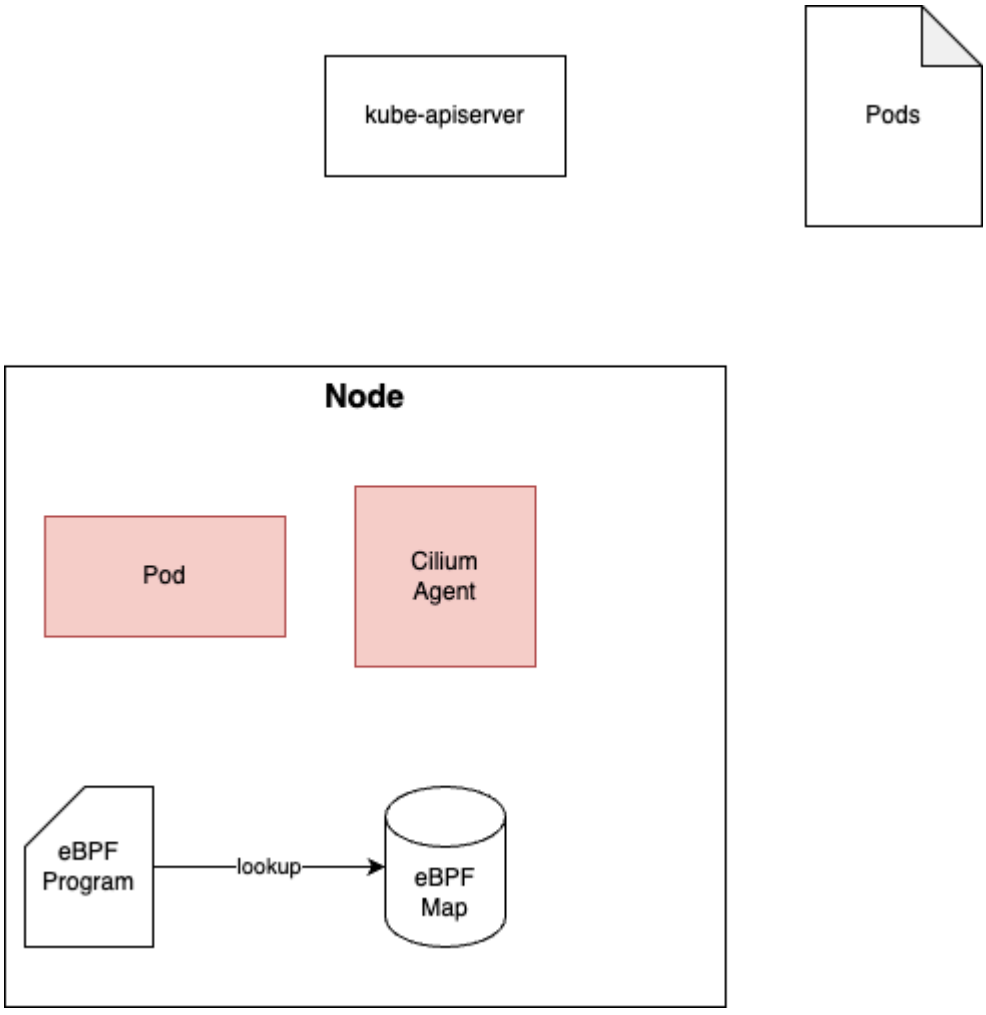


Figure 5-4. Behaviour During Agent Outagage

Only when the agent is brought back online (by applying the rightful label back on the DaemonSet's node selector) are the maps reconciled and stale entries removed:

```
$ kubectl patch daemonset -n kube-system cilium --
type='json' -p='[{"op": "remove", "path":
"/spec/template/spec/nodeSelector/foo"}]'
daemonset.apps/cilium patched
$ kubectl get ds -n kube-system cilium
NAME     DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE
NODE SELECTOR            AGE
```

```
cilium   3         3         3       3            3
kubernetes.io/os=linux   18m
```

Now everything is working as expected again. All healthy backends are now listed for our service:

```
$ kubectl get pods -o wide
NAME                     READY   STATUS    RESTARTS   AGE
IP             NODE
httpd-7ffb6c7dd7-8lxjg   1/1     Running   0          4m15s
10.244.1.233   kind-worker2
httpd-7ffb6c7dd7-m65rg   1/1     Running   0          4m15s
10.244.1.183   kind-worker2
httpd-7ffb6c7dd7-vj6r8   1/1     Running   0          5m21s
10.244.2.112   kind-worker

$ kubectl exec -it -n kube-system ds/cilium -c cilium-agent
-- cilium-dbg service list
ID   Frontend                Service Type   Backend
[...]
6    10.96.130.157:80/TCP   ClusterIP      1 =>
10.244.1.233:80/TCP (active)
                                           2 =>
10.244.1.183:80/TCP (active)
                                           3 =>
10.244.2.112:80/TCP (active)
[...]
```

In summary, the datapath continues to forward traffic without the Cilium agent, giving operators resiliency during upgrades or restarts. But until the agent resumes, eBPF maps remain stale, which can lead to timeouts if backends fail. Minimising agent downtime is therefore essential to maintain both connectivity and accuracy. This behaviour highlights one of the design principles of Cilium KPR: the data plane is independent of the control plane.

NOTE

One way to reduce agent downtime during upgrades is to pre-pull the new Cilium images on all nodes. This can be done manually or by setting `preflight.enabled=true` in the Helm chart before upgrading. More information on upgrading Cilium is provided in chapter TODO.

KPR Service Behaviour Support

So far we have seen how Cilium replaces kube-proxy, how it handles service load balancing at scale without the iptables burden, and how it continues to operate even if the agent is temporarily unavailable.

The next question is what this means for the behaviours that users expect from Services. Kubernetes supports a variety of options that influence how traffic is routed, including session affinity, external and internal traffic policies, NodePort, and LoadBalancer Services.

Cilium's kube-proxy replacement implements all of these features, which means that users can rely on the same semantics as with `kube-proxy`, while also gaining the efficiency of eBPF in the datapath. When moving from `kube-proxy` to Cilium's replacement, the same Service behaviours remain available, including the following:

Session Affinity (ClientIP)

Ensures that traffic from a client IP is directed to the same backend pod. We will cover Session Affinity in more detail in the next section.

Traffic Policies

`externalTrafficPolicy` and `internalTrafficPolicy` allow operators to specify local vs cluster affinity and whether the original client IP is preserved.

Service Traffic Distribution

Balances traffic more intelligently across backends, replacing earlier approaches such as topology-aware hints. We will cover this in detail in Chapter 8.

Session Affinity

Services managed by Cilium's KPR support *session affinity*, commonly known as sticky sessions. By default, when a client sends multiple requests to a Service, the requests may be distributed across different pods. With session affinity enabled, the same client consistently reaches the same pod for the duration of the session. This behaviour can be useful for applications that need consistent routing to a single pod, such as when storing in-memory state or maintaining a shopping cart. However, sticky sessions should be used sparingly. In most cases it is better to design applications so that they do not depend on affinity, since it can skew load distribution and reduce resiliency.

> **WARNING**
>
> Sticky sessions can provide short-term relief for applications that expect stateful behaviour, but they are best considered a workaround for workloads that are not fully cloud-native. Wherever possible, refactor applications to avoid this dependency and allow requests to be load balanced freely.

To demonstrate how sticky sessions work, let's create a simple Deployment and Service (affinity-deployment.yaml) in your KPR-enabled cluster:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: podinfo
```

```
spec:
  replicas: 5
  selector:
    matchLabels:
      app: podinfo
  template:
    metadata:
      labels:
        app: podinfo
    spec:
      containers:
      - name: podinfo
        image: stefanprodan/podinfo
---
apiVersion: v1
kind: Service
metadata:
  name: podinfo
spec:
  selector:
    app: podinfo
  ports:
  - port: 9898
```

Next, use the client pod we deployed in the previous task and from its shell (`kubectl exec -it netshoot-client -- bash`), make a couple of requests to the podinfo Service:

```
$  curl podinfo:9898
{
  "hostname": "podinfo-849bfb5c8d-mp8l7",
  "version": "6.9.0",
  "revision": "fb3b01be30a3f353b221365cd3b4f9484a0885ea",
  "color": "#34577c",
  "logo":
"https://raw.githubusercontent.com/stefanprodan/podinfo/gh-
pages/cuddle_clap.gif",
  "message": "greetings from podinfo v6.9.0",
  "goos": "linux",
  "goarch": "arm64",
  "runtime": "go1.24.3",
```

```
  "num_goroutine": "6",
  "num_cpu": "10"
}
$ curl podinfo:9898
{
  "hostname": "podinfo-849bfb5c8d-bk452",
  "version": "6.9.0",
  "revision": "fb3b01be30a3f353b221365cd3b4f9484a0885ea",
  "color": "#34577c",
  "logo":
"https://raw.githubusercontent.com/stefanprodan/podinfo/gh-
pages/cuddle_clap.gif",
  "message": "greetings from podinfo v6.9.0",
  "goos": "linux",
  "goarch": "arm64",
  "runtime": "go1.24.3",
  "num_goroutine": "6",
  "num_cpu": "10"
}
```

As shown, the `hostname` in the response changes depending on which podinfo pod we are routed to. By enabling sticky sessions, we can ensure the client always reaches the same pod for subsequent requests.

To configure session affinity, set the `sessionAffinity` field in the Service specification. There are only two possible values:

- `None` (default): requests from the same client can be distributed across different pods.
- `ClientIP`: requests from the same client IP are consistently directed to the same pod.

With `kubectl edit service podinfo`, edit the `sessionAffinity` value from `None` to `ClientIP`. The Service should now look like this

```
$ kubectl get service podinfo -o yaml
apiVersion: v1
kind: Service
```

```
metadata:
  name: podinfo
spec:
  sessionAffinity: ClientIP
  selector:
    app: podinfo
  ports:
  - port: 9898
```

Now, when we make multiple requests, they will always land on the same pod:

```
$ curl podinfo:9898
{
  "hostname": "podinfo-849bfb5c8d-4v8sc",
  "version": "6.9.0",
  "revision": "fb3b01be30a3f353b221365cd3b4f9484a0885ea",
  "color": "#34577c",
  "logo":
"https://raw.githubusercontent.com/stefanprodan/podinfo/gh-
pages/cuddle_clap.gif",
  "message": "greetings from podinfo v6.9.0",
  "goos": "linux",
  "goarch": "arm64",
  "runtime": "go1.24.3",
  "num_goroutine": "6",
  "num_cpu": "10"
}
$ curl podinfo:9898
{
  "hostname": "podinfo-849bfb5c8d-4v8sc",
  "version": "6.9.0",
  "revision": "fb3b01be30a3f353b221365cd3b4f9484a0885ea",
  "color": "#34577c",
  "logo":
"https://raw.githubusercontent.com/stefanprodan/podinfo/gh-
pages/cuddle_clap.gif",
  "message": "greetings from podinfo v6.9.0",
  "goos": "linux",
  "goarch": "arm64",
  "runtime": "go1.24.3",
  "num_goroutine": "6",
```

```
  "num_cpu": "10"
}
```

As you can see, the `hostname` remains consistent across multiple requests, indicating that we are always reaching the same Pod. By default, the sticky session lasts for three hours. This means that if no requests are made during that period, the client will be redirected to a random pod on the next request. However, that period can be adjusted. For example, to set the sticky session timeout to 10 seconds, update the Service like this:

```
apiVersion: v1
kind: Service
metadata:
  name: podinfo
spec:
  sessionAffinity: ClientIP
  sessionAffinityConfig:
    clientIP:
      timeoutSeconds: 10
  selector:
    app: podinfo
  ports:
  - port: 9898
```

Let's perform some more calls now, then stop for 10 seconds and see if anything changed:

```
$ curl podinfo:9898
{
  "hostname": "podinfo-849bfb5c8d-4v8sc",
  "version": "6.9.0",
  "revision": "fb3b01be30a3f353b221365cd3b4f9484a0885ea",
  "color": "#34577c",
  "logo":
"https://raw.githubusercontent.com/stefanprodan/podinfo/gh-
pages/cuddle_clap.gif",
  "message": "greetings from podinfo v6.9.0",
  "goos": "linux",
```

```
  "goarch": "arm64",
  "runtime": "go1.24.3",
  "num_goroutine": "6",
  "num_cpu": "10"
}
$ curl podinfo:9898
{
  "hostname": "podinfo-849bfb5c8d-4v8sc",
  "version": "6.9.0",
  "revision": "fb3b01be30a3f353b221365cd3b4f9484a0885ea",
  "color": "#34577c",
  "logo":
"https://raw.githubusercontent.com/stefanprodan/podinfo/gh-
pages/cuddle_clap.gif",
  "message": "greetings from podinfo v6.9.0",
  "goos": "linux",
  "goarch": "arm64",
  "runtime": "go1.24.3",
  "num_goroutine": "6",
  "num_cpu": "10"
}
$ sleep 10
$
$ curl podinfo:9898
{
  "hostname": "podinfo-849bfb5c8d-9qfh6",
  "version": "6.9.0",
  "revision": "fb3b01be30a3f353b221365cd3b4f9484a0885ea",
  "color": "#34577c",
  "logo":
"https://raw.githubusercontent.com/stefanprodan/podinfo/gh-
pages/cuddle_clap.gif",
  "message": "greetings from podinfo v6.9.0",
  "goos": "linux",
  "goarch": "arm64",
  "runtime": "go1.24.3",
  "num_goroutine": "6",
  "num_cpu": "10"
}#
$ curl podinfo:9898
{
  "hostname": "podinfo-849bfb5c8d-9qfh6",
  "version": "6.9.0",
```

```
  "revision": "fb3b01be30a3f353b221365cd3b4f9484a0885ea",
  "color": "#34577c",
  "logo":
"https://raw.githubusercontent.com/stefanprodan/podinfo/gh-
pages/cuddle_clap.gif",
  "message": "greetings from podinfo v6.9.0",
  "goos": "linux",
  "goarch": "arm64",
  "runtime": "go1.24.3",
  "num_goroutine": "6",
  "num_cpu": "10"
}#
```

As demonstrated, after the 10 second timeout, the client is landing on a different pod, as the sticky session has expired.

In summary, sticky sessions provide a way to keep traffic from a client directed to the same pod, which can help applications that rely on in-memory state. They should be treated as an exception rather than the norm, since they reduce load-balancing flexibility and resiliency.

Traffic Policies Support

As mentioned previously, KPR supports both standard Kubernetes traffic management policies: externalTrafficPolicy and internalTrafficPolicy. Let's briefly explain them.

InternalTrafficPolicy

By default, Services use a cluster-wide internalTrafficPolicy, which distributes traffic randomly to all ready endpoints across the cluster (Figure 6-5).

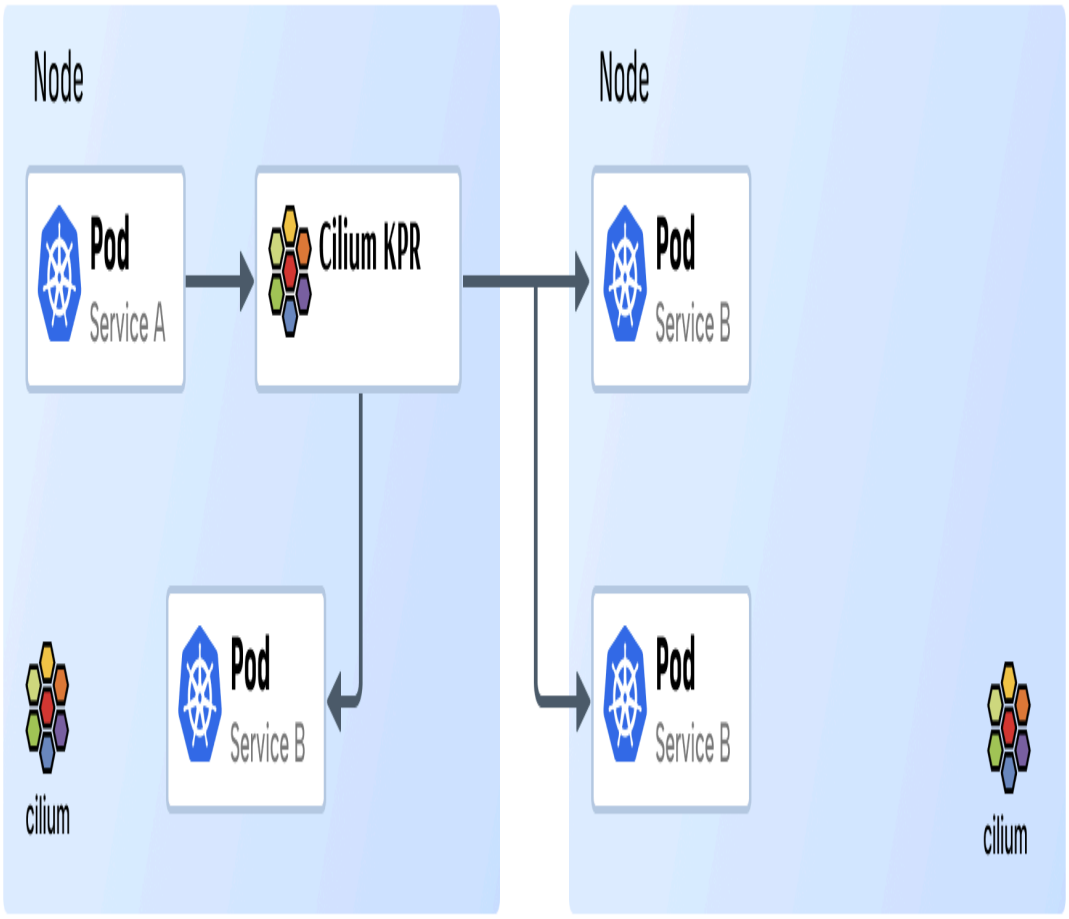


Figure 5-5. Cluster-wide Internal Traffic Policy

When set to Local (see following manifest), Cilium only forwards internal traffic to endpoints that are local to the same node (Figure 6-6):

```
apiVersion: v1
kind: Service
metadata:
  name: internal-traffic-policy-service
spec:
  selector:
    app: nginx
```

```
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  internalTrafficPolicy: Local
```

WARNING

If a node has no local endpoints for a Service configured with `internalTrafficPolicy: Local`, pods on that node will see the Service as having no backends, even though endpoints exist on other nodes.

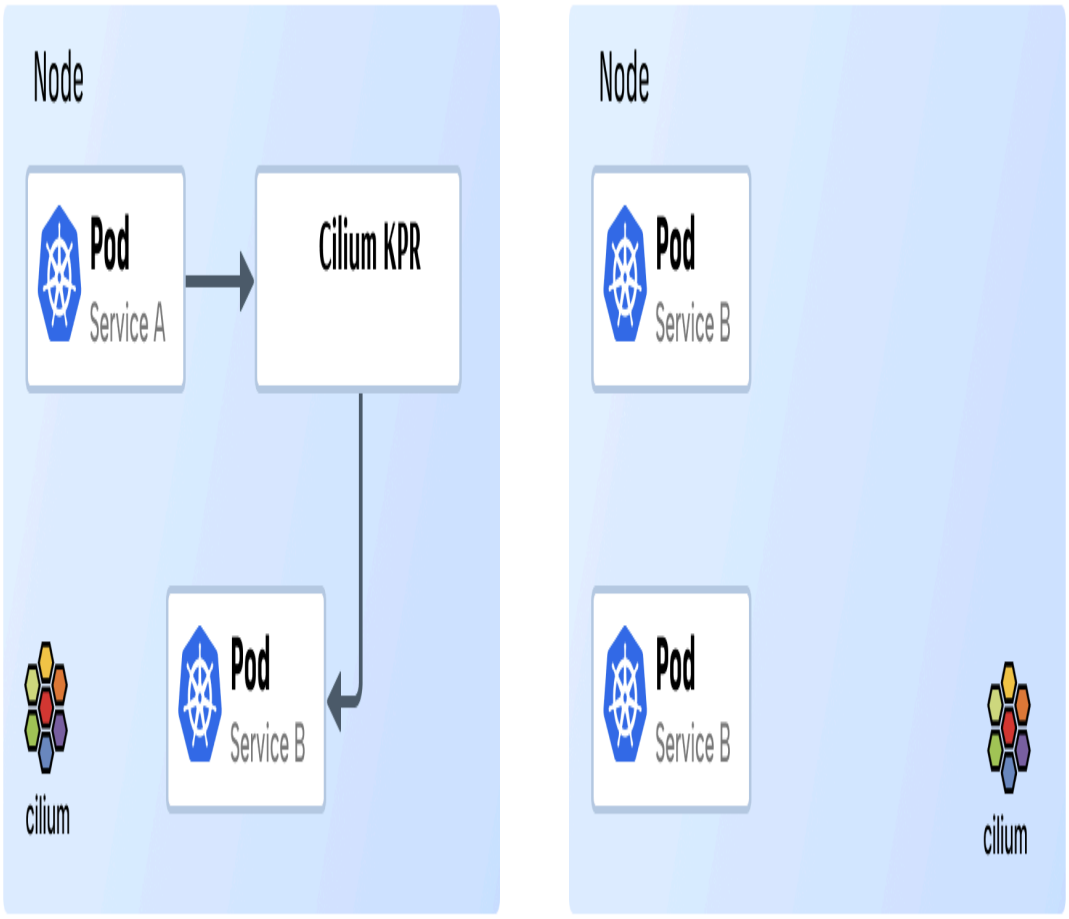


Figure 5-6. Local Internal Traffic Policy

This option is useful for cases where locality matters, such as directing traffic to a node-local logging daemon or metrics agent, and avoiding the latency or cross-node transfer costs of sending traffic elsewhere. Cilium itself leverages this feature through Hubble, which uses node-local routing for observability traffic.

We'll cover more traffic engineering features in Chapter 8.

ExternalTrafficPolicy

`externalTrafficPolicy` controls how traffic that originates from outside the cluster is forwarded and is to be used for external-facing services like NodePort and LoadBalancer Services.

By default, the `externalTrafficPolicy` is set to `Cluster`: in this mode, any node that receives the connection can forward it to any ready backend in the cluster, even if the node hasn't got a local backend. This maximises availability but comes with 2 caveats: 1) it rewrites the client source IP and the loss of the client IP is to the inbound node's IP (client IP is not preserved) and 2) it might add an additional hop as traffic wouldn't necessarily enter the node closest to the backend.

Figure 6-7 illustrates it: even if Service-A only has one backend in Node-1, on reception of traffic, both nodes would forward the traffic over to the pod. For traffic entering via Node-2, the source IP would be changed to Node-2's.

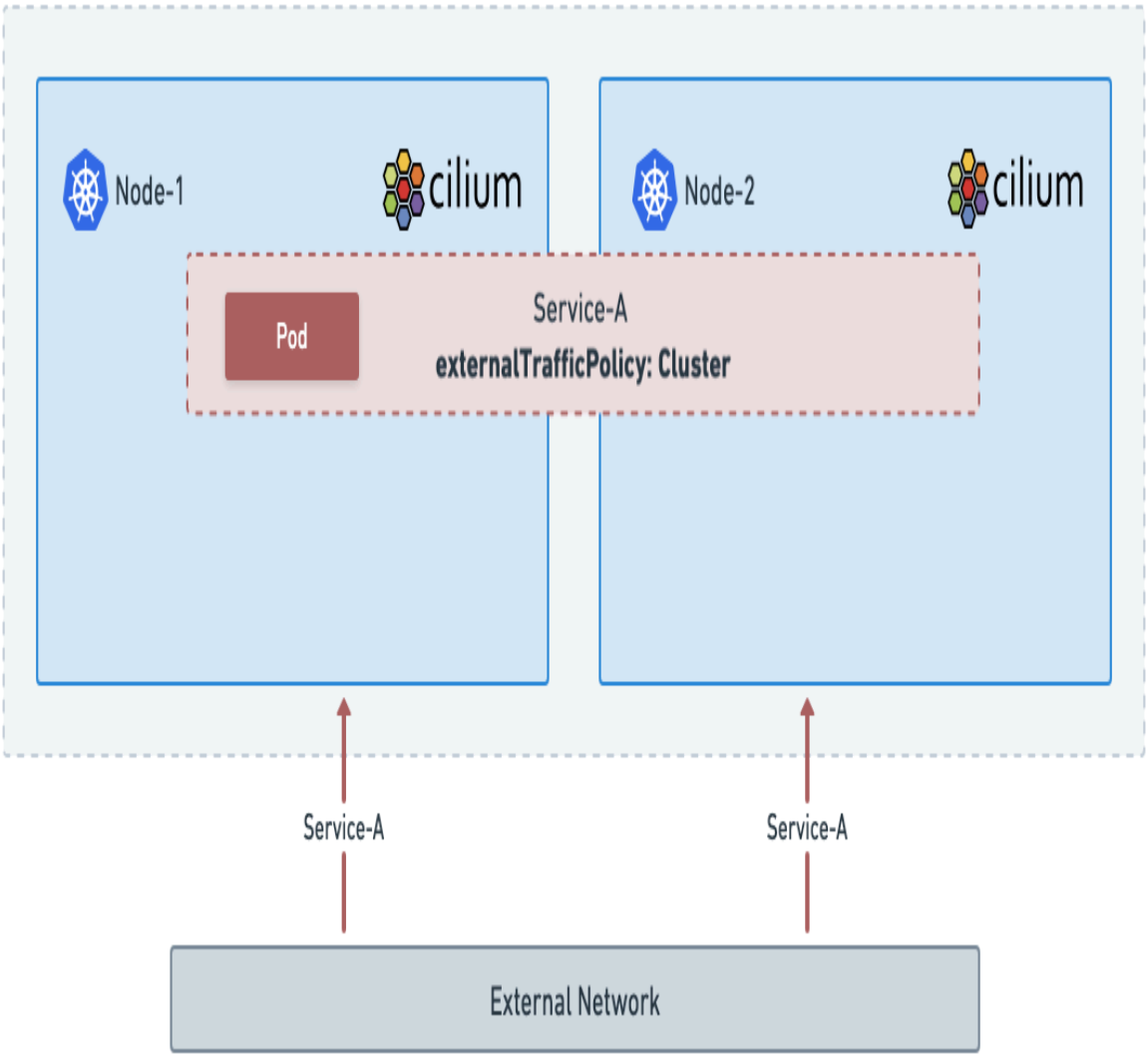


Figure 5-7. Cluster ExternalTrafficPolicy

When setting `externalTrafficPolicy` to `Local` like in the following example, nodes would only forward the connection to **local** backends on that node (Figure 6-8).

```
apiVersion: v1
kind: Service
metadata:
  name: httpd
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  selector:
```

```
    app: httpd
  ports:
  - port: 80
    targetPort: 80
```

This preserves the original client IP. Just be aware that, if the node has no local backends, it will drop external traffic for that Service, even if other nodes have healthy backends.

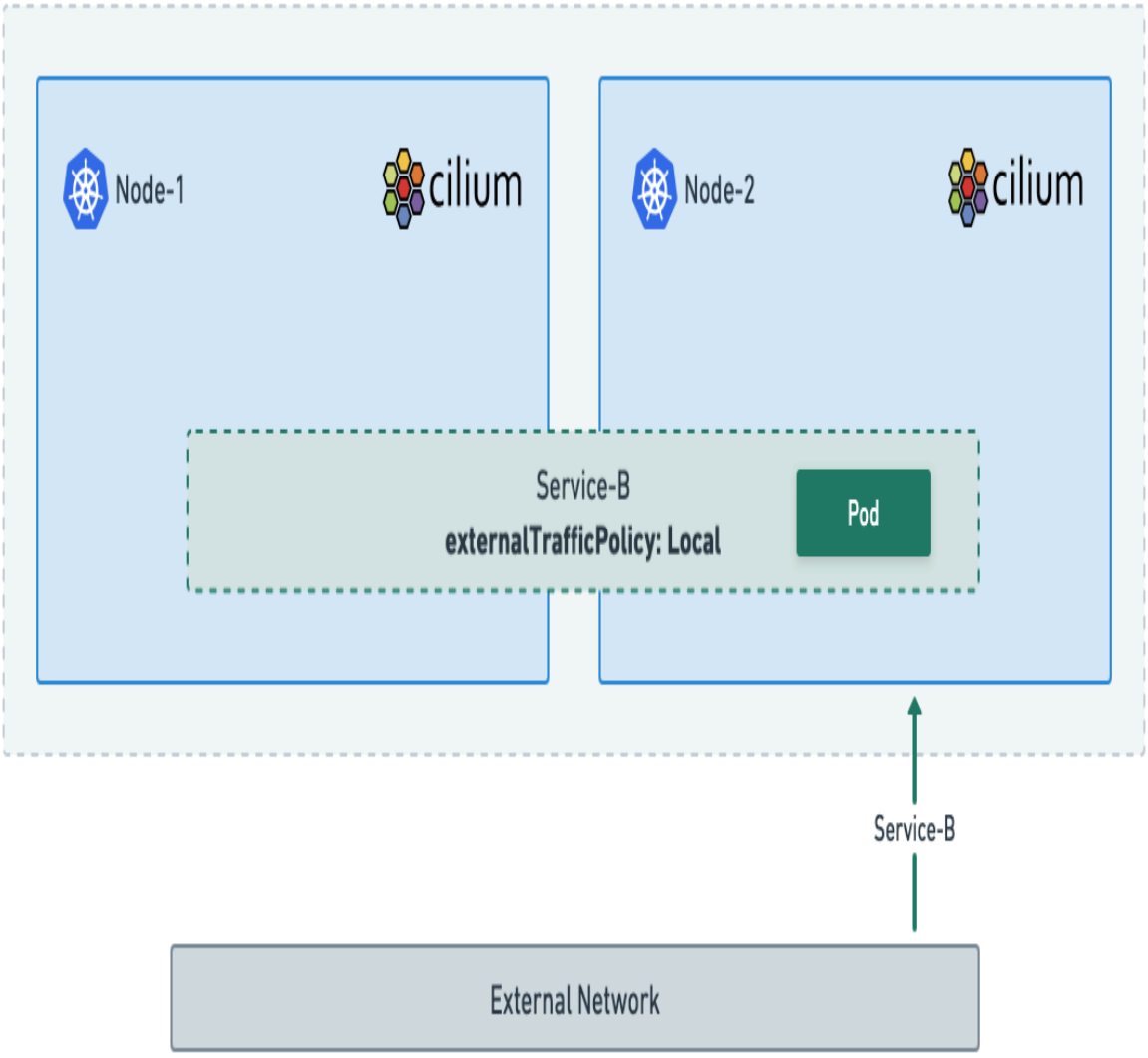


Figure 5-8. Local ExternalTrafficPolicy

KPR fully supports both `internalTrafficPolicy` and `externalTrafficPolicy`, so users can rely on the same semantics as with `kube-proxy`. When client IP preservation

matters but you also want to avoid source translation side effects, we will revisit alternatives such as Direct Server Return (DSR) in Chapter 8.

This brings us to another common requirement: exposing Services outside the cluster.

External-Facing Services

Services have different types for different purposes. By default, a Service is of type `ClusterIP`. This means it provides an in-cluster virtual IP and acts as an internal load balancer. However, a `ClusterIP` Service is not accessible from outside the cluster.

When workloads outside the cluster must access an application running inside Kubernetes, two other types of Services can be used. The first is a Service of type `NodePort`. A `NodePort` Service exposes a port on every node in the cluster. Traffic sent to that port is forwarded to the application, but while the port number is the same across nodes, each node has a different IP address. Clients must therefore be aware of all node IPs and handle load balancing themselves.

A Service of type `LoadBalancer` addresses this problem by providing a single external IP. In cloud environments such as GKE, EKS, or AKS, this process is handled automatically by the cloud-controller-manager. When a `LoadBalancer` Service is created, the controller provisions a load balancer instance in the cloud, points it at the cluster nodes, and assigns the instance's IP as the Service's external IP. The Service is then marked as ready, making it easy to consume in supported cloud environments.

LoadBalancer IP Address Management (LB

IPAM) / Service Type LoadBalancer

The situation is different in environments without a cloud-controller-manager. A `LoadBalancer` Service can still be created, but no external IP will be assigned to it. This is where Cilium's LoadBalancer IP Address Management (LB IPAM) comes in. Enabling LB IPAM allows Cilium to assign IP addresses directly to `LoadBalancer` Services, making them externally reachable even in on-premises or bare-metal environments. These IPs can then be announced using Cilium's BGP or L2 announcement features, which we will cover in Chapter 10.

> **NOTE**
>
> This section focuses on assigning external IP addresses to Services. IP address management for pods was covered in Chapter 4.

Let's test it. This feature is always enabled in Cilium but is dormant until the first IP pool is configured. If you created a cluster with Cilium earlier, you can reuse it for this example. Otherwise, create a kind cluster using the generic configuration (`kind-cluster-config.yaml`) and run `cilium install`. Kind does not come with a cloud-controller-manager - instead Cilium will be the system assigning the IP to our external LB service. Let's create one (`httpd-lb-service.yaml`):.

```
apiVersion: v1
kind: Service
metadata:
  name: httpd
spec:
  type: LoadBalancer
  selector:
    app: httpd
```

```
  ports:
  - port: 80
```

Once the Service is created, it will show a pending external IP:

```
$ kubectl get svc httpd
NAME    TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
httpd   LoadBalancer   10.96.72.137   <pending>
80:31478/TCP   20s
```

No matter how long we wait no external IP will be assigned. To get an IP assigned we must create a `CiliumLoadBalancerIPPool` object (lb-pool.yaml)::

```
apiVersion: "cilium.io/v2alpha1"
kind: CiliumLoadBalancerIPPool
metadata:
  name: "lb-pool"
spec:
  blocks:
  - cidr: "172.16.0.0/24"
```

Once this object is created the Cilium operator takes responsibility for assigning IPs from the defined pool to `LoadBalancer` Services. If we check the Service again we will see an external IP from the pool assigned:

```
$ kubectl get svc
NAME    TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
httpd   LoadBalancer   10.96.72.137   172.16.0.0
80:31478/TCP   48s
```

Note that the block defined was `172.16.0.0/24`, which ranges from `172.16.0.0` to `172.16.0.255`. By default, the Cilium operator may assign any IP within that block. In conventional networking, however, the first and last IP addresses are usually

reserved: the first is the network address and the last is the broadcast address. In many cases we want to avoid assigning these addresses to Services.

There are two ways to prevent the use of the first and last addresses:

1. **Define a custom start and stop range** that explicitly excludes the first and last addresses.
2. **Set** `allowFirstLastIPs: "No"` : this is the most scalable option and requires the least manual configuration.

We will start with the first option. Update the pool object as follows (`lb-pool-start-stop.yaml`)::

```
apiVersion: "cilium.io/v2alpha1"
kind: CiliumLoadBalancerIPPool
metadata:
  name: "lb-pool"
spec:
  blocks:
  - start: 172.16.0.1
    stop: 172.16.0.254
```

Now the Cilium operator will only assign IPs from 172.16.0.1 to 172.16.0.254. Since we updated the existing pool it will automatically reassign the external IP of the Service:

```
$ kubectl get svc
NAME    TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
httpd   LoadBalancer   10.96.72.137   172.16.0.1
80:31478/TCP   1m14s
```

When managing multiple blocks it can be tedious to define start and stop for each one manually. Instead we can use

`allowFirstLastIPs: "No"` to automatically exclude those IPs. Let's update the object again (lb-pool-no-allow-first-last.yaml)

```
apiVersion: "cilium.io/v2alpha1"
kind: CiliumLoadBalancerIPPool
metadata:
  name: "lb-pool"
spec:
  allowFirstLastIPs: "No"
  blocks:
  - cidr: "192.168.9.0/24"
  - cidr: "10.234.9.0/24"
```

By specifying `allowFirstLastIPs: "No"`, we ensure that the Cilium operator does not assign 192.168.9.0, 192.168.9.255, 10.234.9.0 and 10.234.9.255. We can verify this by checking the Service again:

```
$ kubectl get svc
NAME    TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
httpd   LoadBalancer   10.96.72.137   192.168.9.1
80:31478/TCP   2m11s
```

In some cases we may want to create multiple `CiliumLoadBalancerIPPools` for different use cases. For example, we might want to reserve a specific block of addresses for Services that are part of a production workload:

```
apiVersion: "cilium.io/v2alpha1"
kind: CiliumLoadBalancerIPPool
metadata:
  name: "production-pool"
spec:
  allowFirstLastIPs: "No"
  blocks:
  - cidr: "100.16.38.0/24"
  serviceSelector:
```

```
    matchLabels:
      environment: production
```

This example restricts the use of the `100.16.38.0/24` block to Services that carry the label `environment: production`. Note that `CiliumLoadBalancerIPPool` is a cluster-wide resource, so any Service in any namespace with the matching label can receive an IP from this pool.

If we want to further restrict the pool to a specific namespace, we can use the special selector `io.kubernetes.service.namespace`:

```
apiVersion: "cilium.io/v2alpha1"
kind: CiliumLoadBalancerIPPool
metadata:
  name: "production-pool"
spec:
  allowFirstLastIPs: "No"
  blocks:
  - cidr: "100.16.38.0/24"
  serviceSelector:
    matchLabels:
      io.kubernetes.service.namespace: payments
      environment: production
```

In this case, only Services in the `payments` namespace with the label `environment: production` will be allocated IPs from the pool.

> **WARNING**
>
> When using `serviceSelector`, avoid overlapping selectors. If multiple `CiliumLoadBalancerIPPools` match the same Service, the outcome is undefined. This can lead to Services receiving an IP from the wrong pool, which may cause incorrect or even unsafe configurations. As a best practice, either do not use selectors at all and let pools serve all Services, or ensure that each selector is unique and unambiguous.

> **NOTE**
>
> Cilium LB IPAM only assigns external IPs to Services. It does not advertise those IPs to external clients. For the addresses to be routable, LB IPAM is typically combined with Cilium's BGP or L2 announcement features, which we will cover in Chapter 10.

Summary

In this chapter we reviewed the fundamentals of Service networking in Kubernetes and examined how Cilium enhances this area by leveraging eBPF. We looked at how Services abstract and load balance traffic to pod backends, and how `kube-proxy` traditionally implements this behaviour using iptables. We then introduced Cilium's kube-proxy replacement, showing how eBPF maps in the kernel enable efficient load balancing, resiliency during agent downtime, and independence between the control plane and datapath.

We also explored Service-specific behaviours supported by KPR, including session affinity, external and internal traffic policies, and their trade-offs. Finally, we discussed how external access can be provided through `LoadBalancer` Services in environments without a cloud-controller-manager, using Cilium's LoadBalancer IPAM

feature, with BGP and L2 announcements (covered in Chapter 10) to make these addresses routable.

These features demonstrate that Cilium not only matches the functionality of `kube-proxy` but also extends it with performance, flexibility, and operational resilience.

In the next chapter, we will build on this foundation by exploring Cilium Ingress and Gateway API. These mechanisms handle external traffic entering the cluster while also providing fine-grained control over L7 traffic for advanced routing and security.

---

[1] Rice, Liz. *Learning eBPF*. Sebastopol, CA: O'Reilly Media, Inc., 2023.

About the Authors

Nico Vibert is a Senior Staff Technical Marketing Engineer at Isovalent—the company behind the open-source cloud native solution Cilium. Prior to Isovalent, Nico worked in many different roles—operations and support, design and architecture, technical pre-sales—at companies such as HashiCorp, VMware and Cisco. Nico regularly speaks at events, whether on a large scale such as VMworld, Cisco Live or at smaller forums such as VMware and AWS User Groups or virtual events such as HashiCorp HashiTalks. Outside of Isovalent, Nico's passionate about intentional diversity & inclusion initiatives and is Chief DEI Officer at the Open Technology organization OpenUK.

Filip Nikolic is a Solutions Architect at Isovalent, the creators of eBPF and Cilium. With years of hands-on experience across a variety of Cloud Native Computing Foundation (CNCF) projects, Filip is not only a seasoned engineer but also a passionate advocate for open-source innovation.

James Laverack is a software engineer and technical speaker with over a decade of industry experience specialising in cloud native software, distributed systems, and networking. Currently James is a Principal Customer Success Architect, Isovalent at Cisco and he has previously worked as a Kubernetes consultant and software engineer across a range of industries.