ISOVALENT
now part of cisco

Kubernetes Networking and Cilium

An Instruction Manual for the Network Engineer

Version 2024-12-05-01

Contents

Contents

About us

About Isovalent

Isovalent is the company founded by the creators and maintainers of Cilium and eBPF. We build open-source software and enterprise solutions to solve the networking, security, and observability needs of modern cloud-native infrastructure. Google (GKE, Anthos), Amazon (EKS-A), and Microsoft (AKS) have all adopted Cilium to provide networking and security for Kubernetes. Cilium is used by platform engineering teams such as Adobe, Bell Canada, ByteDance, Capital One, Datadog, IKEA, Schuberg Philis, and Sky.

About the Author

Nico Vibert is a Senior Staff Technical Marketing Engineer at Isovalent – the company behind the open-source cloud-native solution Cilium.

Prior to joining Isovalent, Nico worked in many different roles – operations and support, design and architecture, technical pre-sales – at companies such as HashiCorp, VMware, and Cisco.

In his current role, Nico focuses primarily on creating content to make networking a more approachable field and regularly speaks at events like KubeCon, VMworld, and Cisco Live.

Nico has held over 15 networking certifications, including the prestigious Cisco Certified Internetwork Expert CCIE (#22990). Nico is now the Lead Subject Matter Expert on the Cilium Certified Associate (CCA) certification.

Acknowledgements

I would like to thank everyone who reviewed this eBook: Thomas Graf, Jerry Hency, Liyi Huang, Dean Lewis, Filip Nikolic, Bill Mulligan, and Raphaël Pinson. Your feedback was extremely valuable; any remaining typos or imprecisions that may have slipped through the cracks are entirely my responsibility.

I would also like to thank Vadim Shchekoldin for the fantastic cover and PixelPoint for their help with the graphic design and editing.

Preface

The dread of Kubernetes Networking is real.

Most Kubernetes operators can confidently deploy applications onto their clusters, but connecting and securing them evokes a sense of apprehension. This fear also applies to experienced network engineers: even CCIEs would need help with the subtleties and nuances of Kubernetes networking.

I can relate: despite my CCIE and almost 20 years working in the networking industry, I found Kubernetes networking confusing. Frankly speaking, most content around Kubernetes Networking was not written with the network engineering community in mind.

In this eBook, we will fill this void.

Given how ubiquitous Kubernetes has become, network engineers will increasingly need to understand Kubernetes and know how to configure, manage, and integrate clusters with the rest of the network.

This eBook will focus on the Kubernetes networking aspects and on what's now become the de facto networking platform for Kubernetes: Cilium.

It is written with you, network engineers, in mind, using words, references and terminology you will understand.

Whether you are beginning your journey as a network administrator or are an established network architect, we hope you find this useful.

Expectations

The expected audience for this eBook is the network engineering community, but we hope it's accessible to anyone. The main technical expectations we expect the reader to be familiar with are concepts such as:

- TCP/IP
- OSI Layers
- Virtualization and containerization
- Core networking building blocks such as:
  - DNS
  - DHCP
  - Routing
  - Switching
  - Load-balancing
  - Firewalls

This eBook should be accessible to most engineers with a CCNA (Cisco Certified Network Associate)-level skillset.

Don't worry if you are not familiar with Kubernetes yet - we will explain some of the core building blocks before diving into its networking aspects. However, we expect the reader to be familiar with concepts such as virtualization and containerization and recommend the Kubernetes overview from the official Kubernetes documentation as pre-reading material.

This document will **not** explain why Kubernetes has become so popular, why it's here to stay, or how becoming a Kubernetes networking expert could boost your career.

What We Need The Network To Do

Regardless of the underlying computing abstractions - bare metal, virtual machine, container - there are common networking tenets that every platform should aspire to provide:

- We need our applications to have accessible IP addresses
- We need our applications to be able to communicate with other applications
- We need our applications to be able to access the outside world (outbound access)
- We need our applications to be accessible from the outside world (inbound access)
- We need our applications to be secured and our data protected
- We need our applications to be globally resilient and highly available
- We may need to meet regulatory goals and requirements
- We need to be able to operate and troubleshoot when applications or the infrastructure misbehave

In the "traditional" world, we would have needed a combination of networking and security appliances to address all these requirements. This includes routers, switches, DNS and DHCP servers, load balancers, firewalls, network monitoring appliances, VPN devices, etc...

Throughout this document, we will address how these guiding principles can be addressed in Kubernetes.

> For the remainder of the eBook, we will refer to past networking practices as "traditional networking". We know this is an oversimplification, but please bear with us.

Back To Basics

Let's start with the basics and the Kubernetes concepts you must understand before we start diving into networking.

One of the core premises behind the use of cloud-native technologies is the adoption of micro-services. The micro-services model provides benefits such as increased scalability, agility, and resilience compared to the monolith application architecture model.

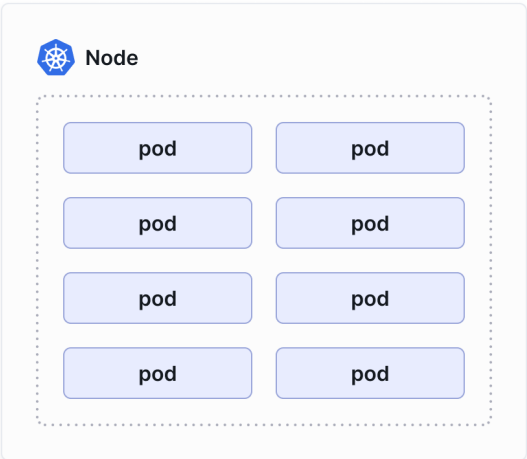
A monolith architecture consists of an entire application built as a single, tightly integrated unit. In contrast, a micro-services architecture deconstructs an application into a set of smaller, independent components - the micro-services. The applications running in micro-services can be updated independently, avoiding the development bottlenecks that can happen when developers work on the same codebase.

The micro-services could be written in different programming languages, providing developers more flexibility in adopting new frameworks than the monolith model.

This brings us to our first Kubernetes concept: **the Kubernetes pod**.

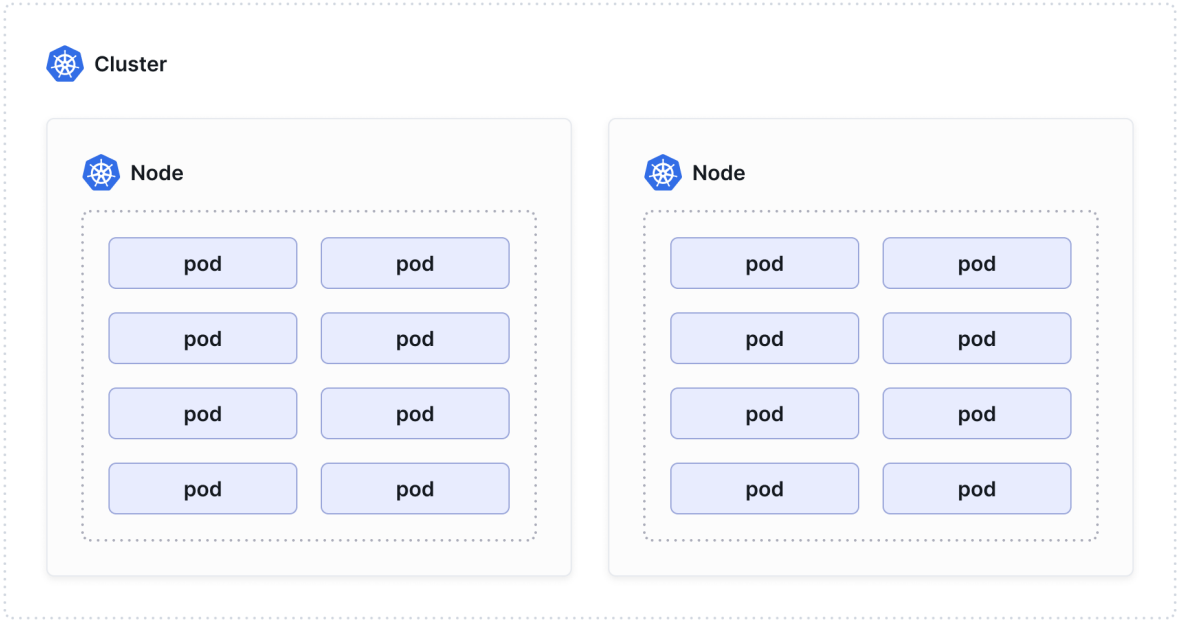
A pod consists of one or more containers. A pod can represent an entire application, a single replica of a distributed application, or an individual service in a micro-service architecture.

A pod will be running on a **Kubernetes node**.



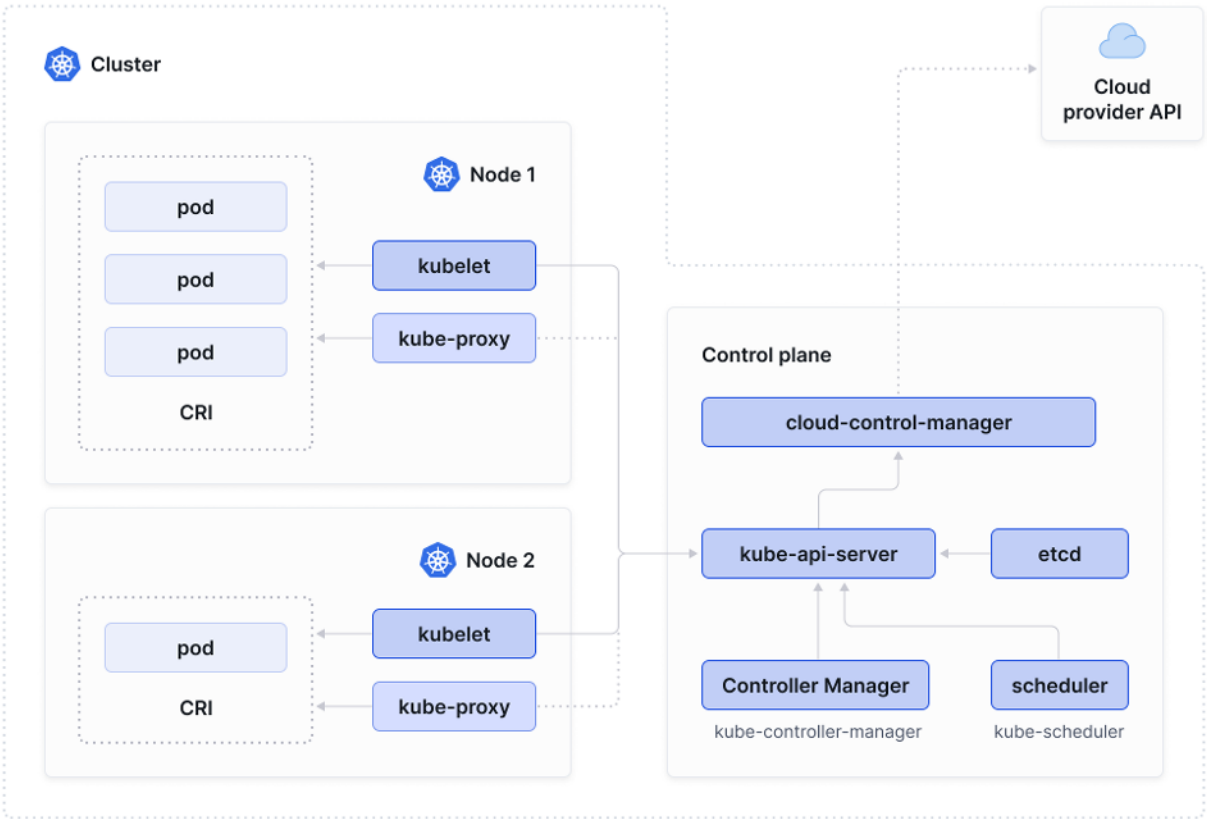
Both virtual machines (VMs) and bare metal servers can be used as nodes in Kubernetes clusters. Typically, when you deploy Kubernetes clusters in the cloud, you would be leveraging VMs as nodes. When you deploy Kubernetes on-premises, the nodes may also be bare-metal servers.

A group of nodes makes up a **Kubernetes cluster**.



Kubernetes is most well-known as a container orchestrator and scheduler platform. In other words, it decides on which node a pod will run. If you're familiar with VMware - and many network engineers operating data center networks will be - you might be familiar with VMware DRS (Dynamic Resource Scheduler). DRS dynamically balances computing resources across a cluster of VMware ESXi hosts. Similarly, Kubernetes monitors the resource usage (CPU, memory) of nodes and schedules pods onto a node based on available resources (among other factors that can be administratively controlled).

The component responsible for making global decisions about the cluster (including scheduling) is the control plane. You are probably familiar with the idea of a control plane from configuring routers: the control plane in a network determines how data packets are forwarded. Similarly, the Kubernetes control plane determines where pods should run.



Kubernetes Networking Fundamentals

Now that we understand pods, nodes, and clusters, we can introduce the Kubernetes network model.

Firstly, every pod in a cluster gets its unique cluster-wide IP address and can communicate with all other pods on any other node without using Network Address Translation (NAT). A practical benefit is that Kubernetes users do not need to create links between pods to connect them explicitly.

Once you create pods, pods should receive a unique IP address and, assuming you have not configured some network segmentation policies (we will come to them later), be able to communicate directly with each other.

Note that containers within a pod all share the same IP address and MAC address.

Kubernetes requires a Container Network Interface (CNI) plugin to implement this model.

The CNI specifies how networking should be implemented in Kubernetes. Think of it as the IETF RFCs you may have consulted over the years. A CNI plugin is an actual implementation of said specification and is commonly referred to as a "CNI" (we will follow this terminology through the rest of this document).

You must use a CNI compatible with your needs to meet your requirements. The feature set from one CNI to another greatly varies.

Just like you, as a network architect, had to go through a network vendor selection (Cisco, Juniper, Arista, etc...) at the start of a networking project, operators and architects of Kubernetes clusters must select a CNI that provides the required networking, security, and observability features.

The CNI that we will zoom in on this document - and the one that tends to win in most CNI evaluations - is the Cilium project.

Introducing Cilium

Cilium is a cloud native networking, security and observability platform. It provides most, if not all, connectivity, firewalling and network monitoring features required to operate and secure Kubernetes clusters.

In words you may be more familiar with, it provides functions such routing, switching, load balancing, firewalling, VPN, web proxy, network monitoring, IP address management, Quality of Service and much more. Throughout the document, you will learn how these components are provided.

Cilium is an open-source solution and was donated to the Cloud Native Computing Foundation (CNCF) in 2021. It has been under the neutral stewardship of the CNCF ever since.

Two years after its donation to the CNCF and after a thorough process involving due diligence reports and security audits, Cilium became and remains the first and only Graduated CNCF project in the cloud-native networking category. It has been adopted as the default CNI by most managed Kubernetes services and distributions.

Isovalent, the creator of the Cilium project, offers a hardened and enterprise edition of Cilium that comes with support and additional enterprise features. We will highlight a couple of these features later in this document, but first, let's answer a question that most network engineers will have been pondering:

Where is my CLI?

Most network engineers begin their networking apprenticeship on the CLI (Command Line Interface) of a network device. While studying for Cisco certifications such as CCNA, CCNP, and CCIE, I spent countless hours on the IOS command line. It remains the primary tool to configure and administer a network.

While the IOS CLI interacts directly with a router's operating system, interacting with Kubernetes is done by communicating with one of the components of the Kubernetes control plane: the Kubernetes API server.

It's typically done using the kubectl CLI - the Kubernetes equivalent of the IOS CLI. The kubectl CLI interacts with the Kubernetes API server to manage and control our Kubernetes cluster from the command line.

When we run a command such as the following, kubectl will communicate with the Kubernetes API server and return an output. This command shows the pods in our cluster in the `default` namespace (we'll explain namespaces later on):

```
$ kubectl get pods
NAME                                    READY           STATUS          RESTARTS        AGE
pod                                     1/1             Running         0               12s
pod-2                                   1/1             Running         0               47s
```

This more advanced command shows the IPs of our pods alongside the nodes they are running on:

```
$ kubectl get pods -o wide
NAME        READY   STATUS    RESTARTS   AGE   IP           NODE          NOMINATED NODE   READINESS GATES
pod         1/1     Running   0          63s   10.0.1.179   kind-worker   <none>           <none>
pod-2       1/1     Running   0          98s   10.0.1.241   kind-worker   <none>           <none>
```

How do I configure Kubernetes and Cilium?

Network engineers have spent decades using `conf t` (`configure terminal`) to enter the configuration mode on a Cisco IOS device. After configuring our router, we would save the configuration to permanent memory by running commands such as `write memory` or `copy running-config startup-config`.

You won't configure individual nodes because Kubernetes is a distributed system with a centralized control plane. Instead, the cluster configuration is done centrally and stored in a data store called etcd, one of the components of the Kubernetes control plane. Just like you would back up your IOS configurations to a TFTP server, Kubernetes engineers should have a backup strategy in place in case etcd fails.

Configuring Kubernetes will require you to apply a configuration to the cluster, typically using the `kubectl` CLI. You will see some files in the YAML format throughout this eBook: they are manifests that represent your desired state. When you run a command such as `kubectl apply -f manifest.yaml`, the `kubectl` tool will push the configuration to the Kubernetes API server. The Kubernetes control plane will then create the corresponding object.

As the Kubernetes documentation explains, a Kubernetes object is a "record of intent". When you create an object, you're effectively telling the Kubernetes system what your cluster's workload should look like; this is your cluster's desired state.

> Note: If you're familiar with some of the software-defined networking concepts, then terms like "intent" and "desired state" should feel familiar.

Here is a sample manifest for a Pod named `tiefighter`. The Pod consists of a single container based on the `docker.io/cilium/json-mock` image.

```
apiVersion: v1
kind: Pod
metadata:
  name: tiefighter
  labels:
    org: empire
    class: tiefighter
    app.kubernetes.io/name: tiefighter
spec:
  containers:
  - name: spaceship
    image: docker.io/cilium/json-mock
```

Cilium's configuration is stored in two separate entities. The main Cilium configuration is stored in a ConfigMap, while the Cilium policies (for network security, BGP, etc...) leverage Custom Resource Definitions (CRDs). Let's explain what both entail.

ConfigMaps are used within Kubernetes to store non-confidential information. A ConfigMap consists of key/value pairs. Cilium's configuration is stored in a ConfigMap called `cilium-config`. With the following lengthy command, you can check the Cilium configuration and filter whether IPv6 has been enabled.

```
$ kubectl get -n kube-system configmap cilium-config -o yaml | yd .data.enable-ipv6
true
```

To make things simpler, users can also use the Cilium CLI to install and manage Cilium. The Cilium CLI is a binary that helps users install, manage, and configure Cilium. The following Cilium command displays the Cilium configuration, with a filter on the IPv6 setting:

```
$ cilium config view | grep enable-ipv6
enable-ipv6                       true
```

This CLI tool will interface with the Kubernetes API as necessary, similar to that of kubectl.

Cilium - and many other Kubernetes tools - also use Custom Resource Definitions (CRDs) to extend Kubernetes. These CRDs allow users to define and manage additional resources beyond the standard Kubernetes objects. Examples of Cilium CRDs include CiliumNetworkPolicies and CiliumBGPPeeringPolicies. We will review some of these CRDs later on.

Where is my DHCP Server?

In traditional networking, a DHCP server allocates IP addresses to a server as it comes online (unless static addressing is used).

While the nodes would typically receive their IP address over DHCP, we do not use it for pods. Instead, we refer to **IPAM** (IP Address Management), a term you might be familiar with if you've used platforms like Infoblox for your network.

In Kubernetes, the CNI is often responsible for assigning an IP address to your pod. Remember, though, that while a pod frequently consists of multiple containers, a pod only receives a single IP, which is shared by all containers in the pod.

The IP assigned to a pod comes from a subnet referred to as "PodCIDR". You might remember the concept of CIDR - Classless Inter-Domain Routing - from doing subnet calculations. In Kubernetes, just think of a PodCIDR as the subnet from which the pod receives an IP address.

Cilium supports multiple IPAM modes to address Kubernetes users' different requirements. Let's review three IPAM modes:

- In Kubernetes-host scope mode (or per-Node PodCIDR), a single large cluster-wide prefix is assigned to a cluster, and Kubernetes carves out a subnet of that prefix for every node. Cilium would then assign IP addresses from each subnet. However, if you were to run out of IPs in your prefix, you would not be able to expand the cluster pool.
- In Cluster scope mode, Cilium is responsible for assigning CIDRs to each node, and as a bonus, it lets you add more prefixes to your cluster if you begin to run out of IPs. Just remember that pods on the same node would receive IP addresses from the same range in the first two modes.
- The Multi-pool mode is the most flexible one. It supports allocating PodCIDRs from multiple different IPAM pools, depending on properties of the workload defined by the user, e.g., annotations. Pods on the same node can receive IP addresses from various ranges. In addition, PodCIDRs can be dynamically added to a node as and when needed.

IPAM Mode	CIDR configuration	Multiple CIDRs per cluster	Multiple CIDRs per node	Dynamic CIDR allocation
Kubernetes Host Scope	Kubernetes	✗	✗	✗
Cluster Scope	Cilium	✓	✗	✗
Multi-Pool Scope	Cilium	✓	✓	✓

What is a Kubernetes namespace?

Kubernetes brings the concept of a namespace which serves to isolate groups of resources within a cluster. The segmentation is minimal: using multiple namespaces does not stop pods in a namespace from communicating with pods in other namespaces.

The following command shows the active namespaces in our cluster:

```
$ kubectl get namespace
NAME                   STATUS   AGE
default                Active   11h
endor                  Active   35s
kube-node-lease        Active   11h
kube-public            Active   11h
kube-system            Active   11h
local-path-storage     Active   11h
```

Operators would typically segregate an application or a tenant in its own namespace and then apply a quota. With a quota, you can limit the resources (CPU/Memory/Storage) used by pods in a specific namespace - quotas are to compute and storage resources what Quality of Service (QoS) is to networking (note: we will later discuss how to enforce networking quality of service in Kubernetes).

Note that pods in different namespaces on the same node may pick up IP addresses from the same range. As you can see from the command below, a pod in the `default` namespace and one in the `endor` namespace receive IP addresses from the same PodCIDR and should be able to communicate with each other (unless network segmentation has been implemented).

```
$ kubectl get pods -n default -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP            NODE          NOMINATED NODE   READINESS GATES
nginx   1/1     Running   0          7s    10.244.1.76   kind-worker   <none>           <none>

$ kubectl get pods -n endor -o wide
NAME            READY   STATUS    RESTARTS   AGE   IP           NODE           NOMINATED NODE   READINESS GATES
deathstar-7848  1/1     Running   0          85s   10.244.1.62  kind-worker2   <none>           <none>
```

This is an essential concept in Kubernetes: IP addresses are irrelevant. While we used to remember the IP addresses of our favorite servers in networks of the past, they no longer represent an identity in the cloud-native space.

Where is my DNS Server?

Now that we know how pods get an IP, let's look at how they get a DNS record. DNS is not part of the CNI's role - a Kubernetes cluster comes with a built-in DNS server (typically, coreDNS) that automatically assigns names based on the namespace and cluster name.

For example, a pod would get a name such as `10-244-1-234.default.pod.cluster.local`, assuming its IP is `10.244.1.234`. We will cover services later in the load balancing section, but know that a service name will have a format such as `my-service.my-namespace.svc.my-cluster.local`.

By default, pods are assigned a DNS policy called `ClusterFirst`. When using this policy, any DNS query that does not match the configured cluster domain suffix is forwarded to an upstream nameserver by the DNS server. DNS policies can be set on a per-Pod basis.

> If you used to blame DNS for most networking issues, you might be somehow reassured that DNS is often the culprit in Kubernetes, too.

How Do Pods Talk To Each Other?

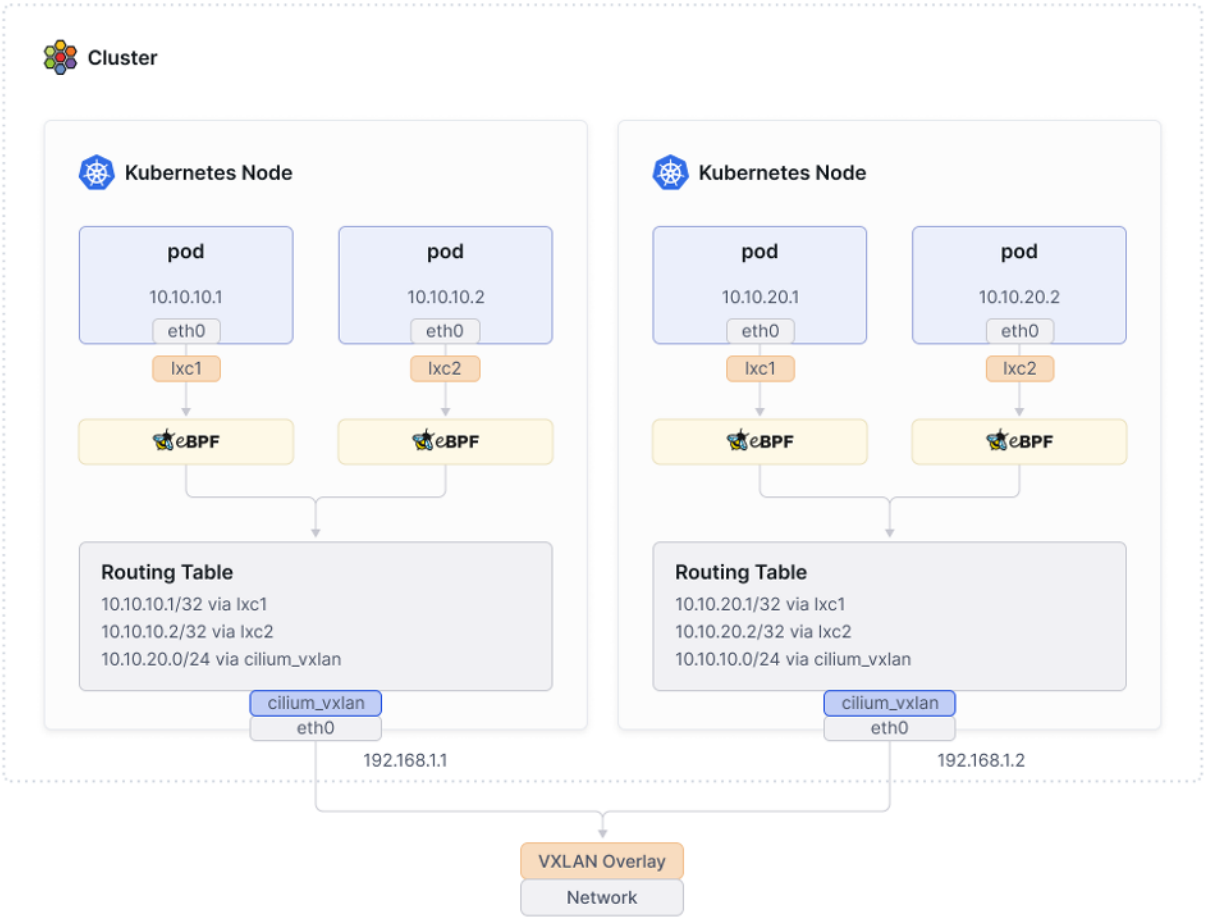
As a network engineer, you need to understand how endpoints communicate - clients and servers, bare metal servers and virtual machines, wireless clients and access points.

You would have had to learn various layers of network abstractions (VLAN, VXLAN, GENEVE, GRE, MPLS) and multiple networking protocols (BGP, OSPF, EIGRP, RIP...).

Kubernetes and Cilium are no different. You will actually be able to re-use a lot of what you've already learned.

Cilium provides two methods to connect pods on different nodes in the same cluster: encapsulation/overlay routing and native/direct routing.

The default one and the one you are most likely to come across is the "encapsulation" mode, where a network overlay (typically VXLAN, but GENEVE is also an option) is created in the cluster, and a group of tunnels is created between all the nodes.



Encapsulation Routing Mode

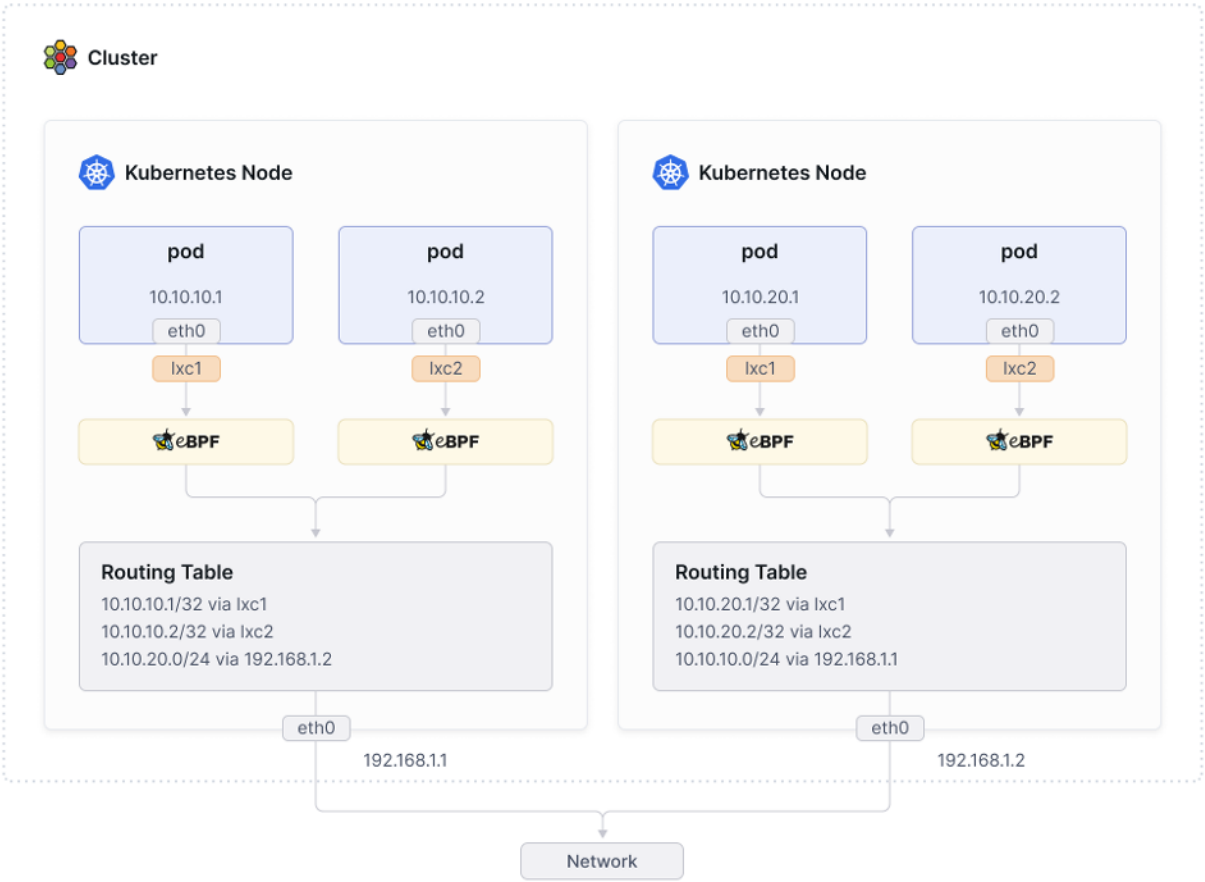
This provides a set of benefits:

- **No reliance on the underlying network** - the network that connects the nodes does not need to be made aware of the Pod CIDRs. As long as the nodes can communicate, the pods should be able to communicate.
- **Overcoming addressing space challenges** - given that there's no dependence on the underlying network, there is potentially a much larger set of IP addresses available to pods.
- **Auto-configuration and simplicity** - new nodes joining the cluster will automatically be incorporated into the overlay.

The only drawback is common to any platforms using network overlays - like Cisco ACI or VMware NSX; the overlay implies adding an encapsulation header. Fifty bytes per network packet for VXLAN for every 1500 bytes (assuming no jumbo frames) is marginal, but some organizations need their network to provide optimal network performance. For them, the native routing mode might be more appropriate.

In native routing, the packets sent by the pods would be routed as if a local process had emitted the packet. As a result, the network connecting the nodes must be capable of routing the IP addresses allocated from the PodCIDRs. The Linux kernel on each node must be aware of how to forward packets to pods - in other words, they need to know how to route the packets!

If all nodes share a single flat L2 network, Cilium offers an option that tells every node about each PodCIDR. If not, the nodes will need to rely on an external router with that knowledge or will need to run an internal BGP daemon.



Native Routing Mode

Kubernetes Metadata

In our traditional world of networking, keeping track of assets connected to the network and the network itself is an ongoing challenge. Ideally, enterprise organizations might have up-to-date network diagrams or use IPAM tools like Infoblox but we know many folks still rely on Excel spreadsheets.

Understanding which device is connected to the network, which role the networked entity serves, and which application it is supporting is tremendously useful for IT operators but creating and managing the information and metadata relevant to each networked device is time consuming.

In Kubernetes, we can assign metadata to our objects. While this doesn't constitute documentation, it can be used by operators to find out additional information about our applications. It can also be used for load balancing and security purposes. Let's review the two methods to assign metadata:

What's a Label?

Kubernetes labels are ways to organize and group your resources. Of course, you would also find labels on your servers and cables in a well-maintained data center with a documented and clear color scheme and naming convention. A glance at the label should inform you where the cable goes or which workload it is running.

Kubernetes labels are similar - they're a well-structured method to indicate to Kubernetes what a pod is and how to distinguish it from other objects.

The following pod manifest has multiple labels to describe the application, the tier, and the environment where the pod is being deployed.

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
  labels:
    app: myapp
    tier: frontend
    environment: production
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

Unlike traditional networking, labels play an essential role in load balancing and security. We will review this shortly.

What's an Annotation?

While labels are a well-organized way to assign data, I refer to Kubernetes annotations as institutional knowledge about our resources. As mentioned, we, network engineers, all have used spreadsheets and Post-it notes to document our network - annotations play a similar role: they're unstructured information about pods that other humans or tools may want to consult or use.

Kubernetes annotations often provide documentation and context about Kubernetes objects like pods and services. As we will see in the Ingress routing and Quality of Service sections later, they can also be interpreted by external systems to add additional functionality to Kubernetes.

```
apiVersion: v1
kind: Pod
metadata:
  name: annotation-example
  annotations:
    example: “isovalent.com”
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

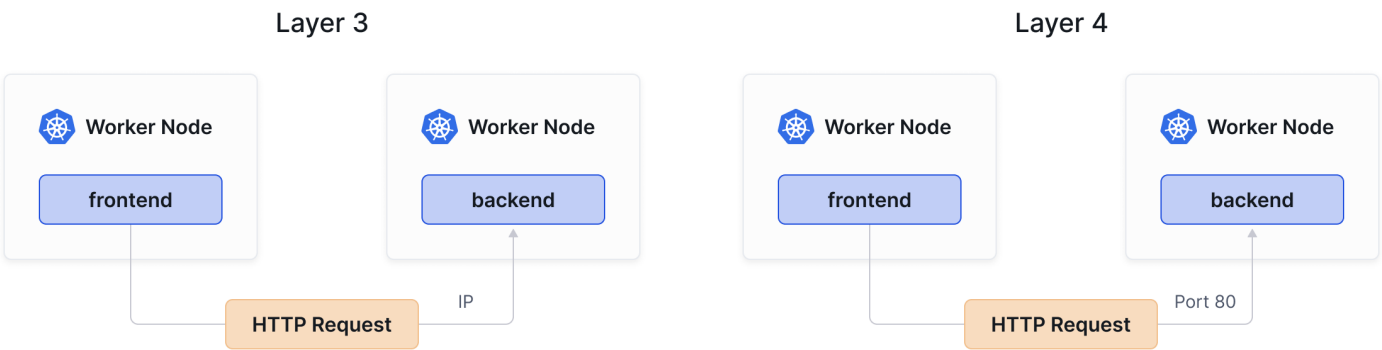
How do I secure my cluster?

As you can imagine, Kubernetes Security is very complex and covers multiple areas such as data encryption at rest and in transit, protection of the control plane and workloads, auditing, Identity and Access Management - the list goes on. Some highly recommended books have been published and dedicated to Kubernetes Security. Let's focus on one aspect of security you would probably be interested in and answer the question:

> What is the equivalent of firewall rules in Kubernetes?
>
> The answer is Network Policies.

Kubernetes Network Policies are implemented by the network plugin/CNI. Not all CNIs support network policies; some provide more advanced and granular network policies than most (we will review the benefits of using Cilium Network Policies shortly).

Standard Kubernetes Network Policies let you specify how a pod can communicate with various entities. They rely on Layer 3 and 4 elements and let you filter based on IP addresses or ports/protocols (such as TCP:80 for HTTP or TCP:22 for SSH).



We'll now review a sample Network Policy.

It should look familiar to most network engineers, but you might find some configuration aspects new.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

In the networking world, you would be familiar with an extended Access List (ACL) for example:

```
!
interface ethernet0
 ip access-group 102 in
!
access-list 102 deny tcp any any eq 23
access-list 102 permit ip any any
```

It denies all TCP traffic over port 23 (Telnet) and allows anything else.

Note that the ACL applies to traffic entering (`in`) the Ethernet0 interface - in other words, it's our selector: we select the traffic to which this particular filter applies.

In Network Policies, we use a `podSelector` to determine who the filters will apply to. An empty one would select all the pods in the namespace. A standard method is to use labels to identify the pods the policy applies to.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
spec:
  podSelector:
    matchLabels:
      role: db
```

In the ACL example above, we used the keyword `in` to indicate the rules would apply to traffic entering the interface. In Kubernetes Network Policies, we would use the `ingress` block:

```
ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
```

Consider the example above - you can see how we are allowing traffic from the `172.17.0.0/16` block, except the `172.17.1.0/24` range.

In a Cisco ACL, the equivalent would be something like this:

```
ip access-list extended K8s_Net_Policy
  deny ip 172.17.1.0 0.0.0.255 any
  permit ip 172.17.0.0 0.0.255.255 any
```

But we can also combine it with selectors - we can allow traffic only it comes from particular blocks, from a specific namespace, and from a specific pod:

```
ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
```

Why would we do this when IP addresses were all we needed in Cisco IOS ACL? As mentioned earlier, IP addresses are primarily irrelevant in Kubernetes. Pods are started and destroyed frequently. One of the advantages of Kubernetes is its ability to scale up and down an application by creating or deleting multiple replicas of a pod across the cluster. The IP addresses of pods are unpredictable. Traditional IP-based firewalling approaches are going to be mostly ineffective.

What's identity-based security?

Cilium's Network Policies are an advanced method to secure clusters. For the reasons highlighted above, Cilium's approach to security is not based on IP addresses but on identities. Identities are derived from labels and other metadata and are assigned to endpoints (Cilium refers to Kubernetes pods as endpoints).

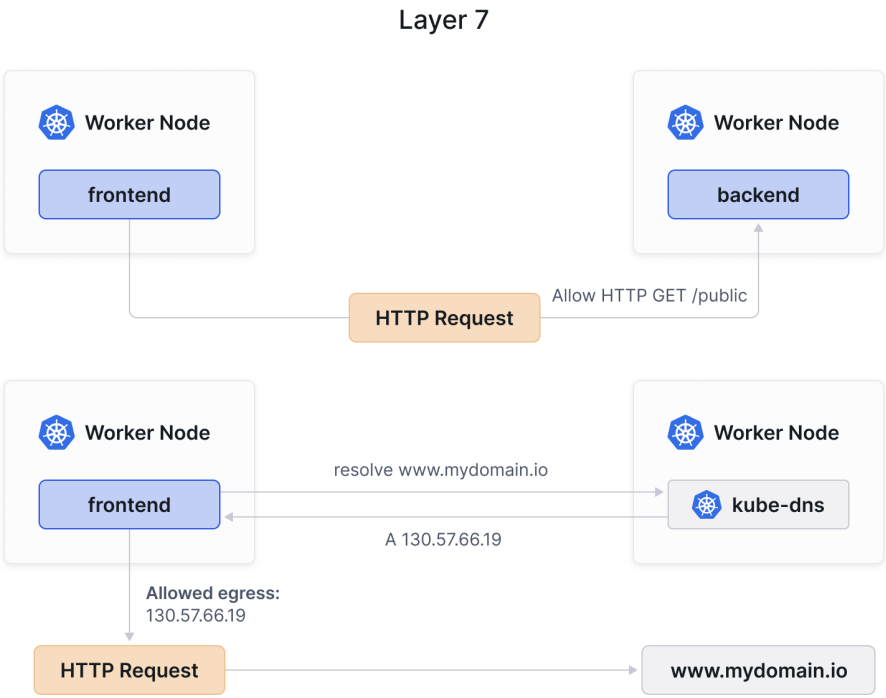
Instead of creating a rule allowing traffic from the `frontend` pod's IP address to the `backend` pod's IP address, you would create a rule to allow an endpoint with the Kubernetes label `role=frontend` to connect to any endpoint with the Kubernetes label `role=backend`.

Whenever replica pods are created with such labels, the network policy will automatically apply to such pods, meaning you will not have to update the existing network policy with the IP addresses of the new pods.

Another benefit provided by Cilium Network Policies is Layer 7 filtering.

Where's my Layer 7 firewall?

Cilium Network Policies also provide Layer 7 application visibility and control. You will be familiar with this concept if you use any web application firewalls (WAFs). Cilium Network Policies let you filter based on application (Layer 7) parameters, like HTTP path, method, or domain names.



Let's review an example:

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "rule1"
spec:
  description: "L7 policy to restrict access to specific HTTP call"
  endpointSelector:
    matchLabels:
      org: empire
      class: deathstar
  ingress:
  - fromEndpoints:
    - matchLabels:
        org: empire
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
      rules:
        http:
        - method: "POST"
          path: "/v1/request-landing"
```

While Kubernetes Network Policies use `podSelector`, Cilium Network Policies use `endpointSelector` to determine which endpoints this applies to. The principle remains the same - the policy above will only apply to pods with the labels `org: empire` and `class: deathstar`.

The policy will only permit HTTP traffic from pods with the `org: empire` label with the HTTP POST method and to the path `/v1/request-landing`.

> While the CiliumNetworkPolicies offer advanced Layer 7 filtering, they do not replace a web-facing firewall or an advanced IDS/IPS. You will learn in a later section how to integrate Kubernetes with your firewalls.

Where's my Load Balancer?

As a network engineer, you probably have deployed and used load balancers from the likes of F5 or Citrix. Load-balancing is, of course, an essential aspect of Kubernetes.

A load balancer helps you achieve resilience and scale by making an application accessible through a single IP (often referred to as a "virtual IP" of a "virtual server") and by distributing the traffic across multiple servers (which we sometimes refer to as "real servers") while monitoring the health of servers through health checks.

In the traditional world, you have to manually add the IPs of the "real servers" fronted by the virtual server. As mentioned earlier, this model wouldn't work in Kubernetes - IPs are meaningless, and the churn of pods is so high that a manual approach would cancel out Kubernetes' auto-scaling benefits.

Kubernetes Services are primarily used for load balancing. Think of it as your virtual server, fronting your pods.

Here is a sample Service configuration:

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

Note how we are using selectors again - remember when we talked about labels earlier and how we suggested they can help us identify a group of applications? Instead of manually adding the IPs of all the `echoserver` pods to our pool of virtual servers, all pods with the `app:echoserver` label will be automatically included, and traffic will be distributed across all pods.

Your data center probably has internal load balancers (to load balance between internal clients and internal servers) and external-facing load balancers (to proxy and load balance traffic from external clients to internal servers).

In Kubernetes, we use different types of Services to distinguish between internal and external use cases.

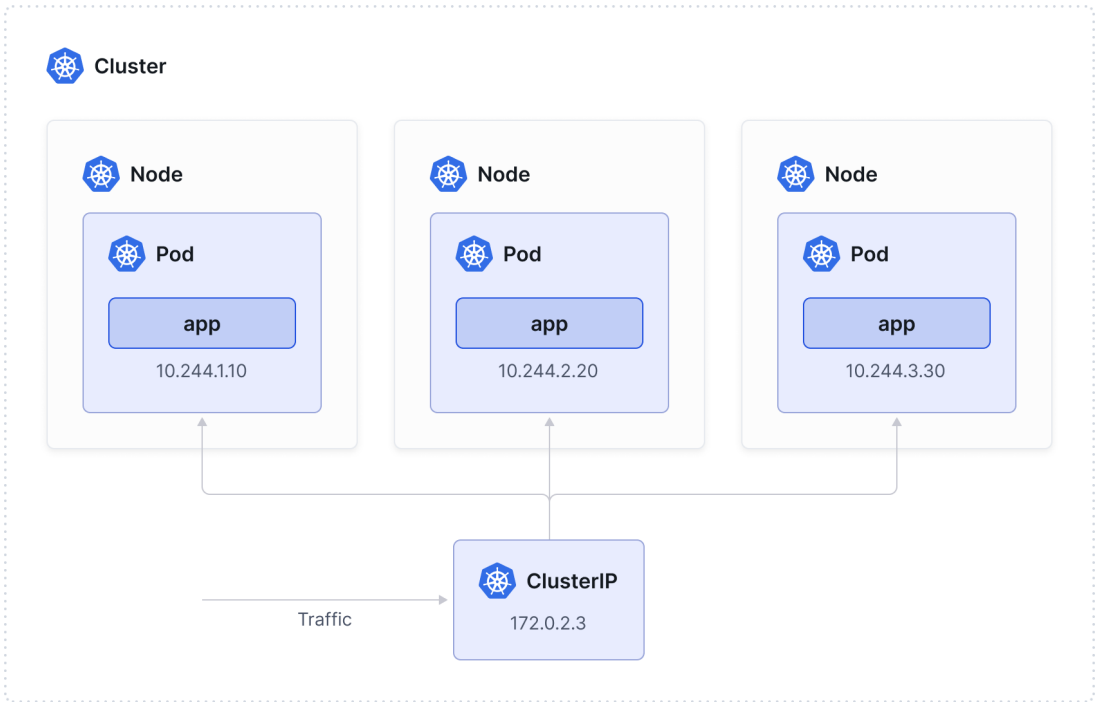
How do I load balance traffic within the cluster?

ClusterIP Services are used to expose the Service only within the cluster. As you can see in the `kubectl` command output below, the ClusterIP service receives a cluster-scoped virtual IP address.

```
$ kubectl get service echoserver
NAME         TYPE        CLUSTER-IP        EXTERNAL-IP   PORT(S)   AGE
echoserver   ClusterIP   172.0.2.3         <none>        80/TCP    19m
```

Note that while the CNI is responsible for assigning IPs to pods, the Kubernetes control plane is responsible for assigning virtual IPs to Services.

ClusterIP is the default Service behavior if you don't specify the `type` in your Service definition. Once the ClusterIP Service is deployed, traffic will automatically be distributed across the pods in the cluster that match the selector (which is based on the labels of a pod).



How do I load balance traffic entering the cluster?

This might be one of the most complex aspects of Kubernetes networking and, to do it justice, we would probably need to double the size of this eBook. To keep it brief, let's focus on the core Kubernetes Services types that are designed to expose a Service into an external IP address: NodePort and LoadBalancer.

NodePort Service

A NodePort Service lets users access your internal services by opening the same port on all nodes (with that port typically in the `30000-32767` range). The nodes will act as NAT gateways and forward traffic to the pods.

To make the node port available, Kubernetes sets up a cluster IP address, the same as if you had requested a ClusterIP Service. The difference is that NodePort also exposes a port on every node towards this IP.

Take a look at this output of a NodePort Service:

```
$ kubectl describe service node-port
Name:                     node-port
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=echoserver
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.96.142.182
IPs:                      10.96.142.182
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  30618/TCP
Endpoints:                10.244.2.3:80,10.244.3.241:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

The node-port Service is deployed in the default namespace and uses the same selector as above (`app=echoserver`).

> This particular Service is IPv4-only but note that Kubernetes and Kubernetes Services also support Dual Stack and IPv6-only (there's a section on IPv6 later in this eBook).

All nodes in the cluster will be listening on port `30618` for this particular service.

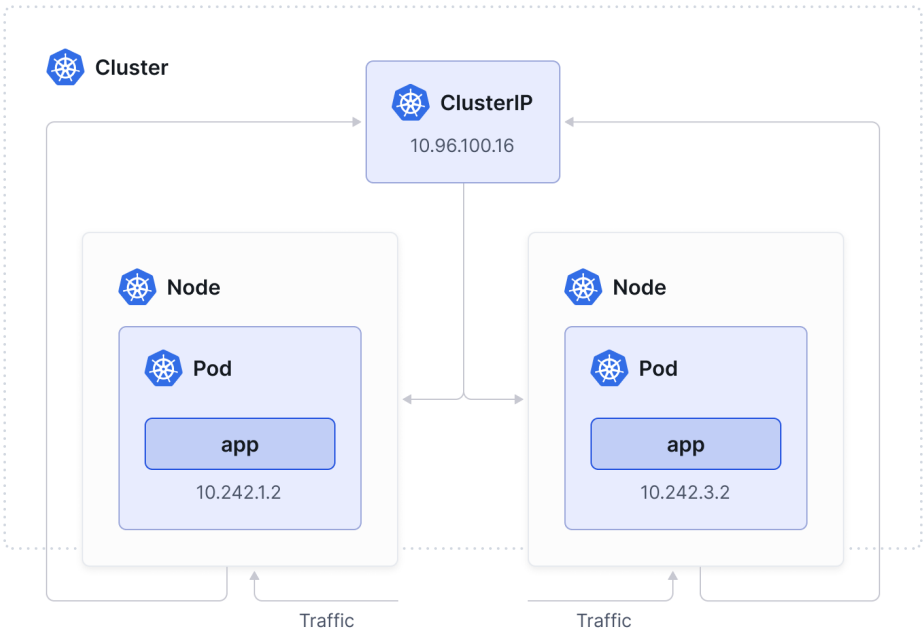
Traffic arriving on the node on this port will be NATed to the healthy pods listening on the `targetPort` specified (`80`).

Let's review a connectivity test from my external client into the cluster:

```
$ kubectl get nodes -o custom-
columns=NAME:.metadata.name,IP:.status.addresses[0].address
NAME                 IP
kind-control-plane   172.18.0.2
kind-worker          172.18.0.5
kind-worker2         172.18.0.4
kind-worker3         172.18.0.3

$ curl -w "%{http_code}\n" -o /dev/null -s http://172.18.0.2:30618
200
$ curl -w "%{http_code}\n" -o /dev/null -s http://172.18.0.3:30618
200
$ curl -w "%{http_code}\n" -o /dev/null -s http://172.18.0.4:30618
200
$ curl -w "%{http_code}\n" -o /dev/null -s http://172.18.0.5:30618
200
```

The first `kubectl` command returns the IP addresses of the nodes. The subsequent `curl` commands are executed to http://$NODE_IP:$NODE_PORT and only return the HTTP return code (with `200` highlighting a successful connection). The external client was able to access the internal service successfully.



NodePort is simple but has limitations: users often run into issues with the constrained `30000-32767` port range, and NodePort does not natively offer load-balancing capabilities. Instead (or rather, alongside it), Kubernetes engineers tend to rely on services of the type LoadBalancer.

LoadBalancer Service

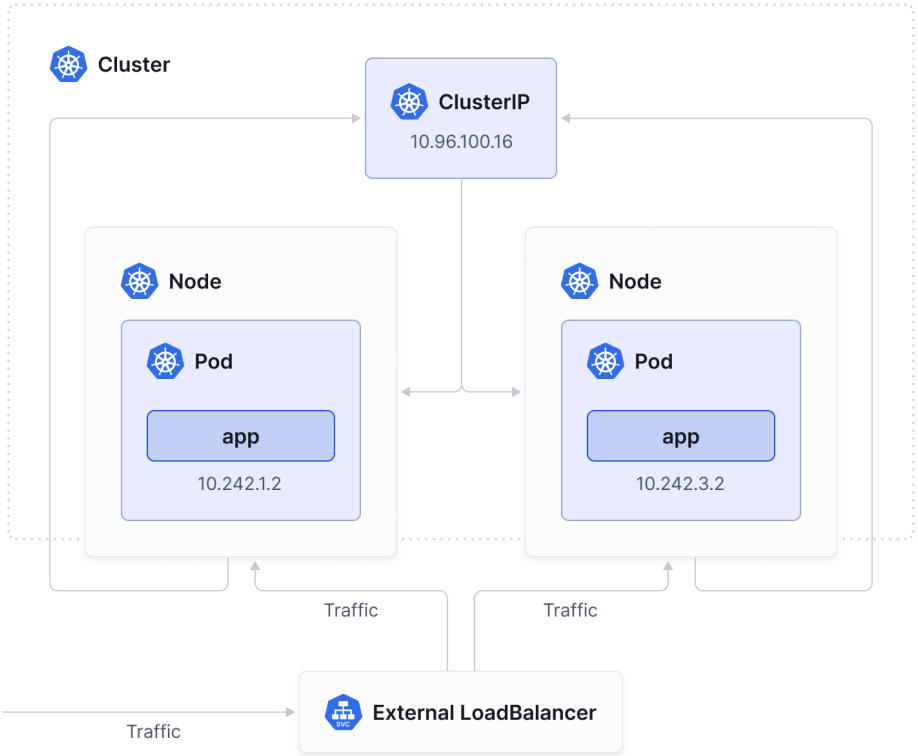
The LoadBalancer Service builds on top of the two previously mentioned Services.

When you create a LoadBalancer service, Kubernetes will communicate with the underlying provider's API to create an external load balancer (like AWS ELB, or Google Cloud Load Balancer), which is outside of the Kubernetes cluster.

The cloud provider will also allocate an external IP address and/or DNS name, which will be used by external clients to reach your internal service. The load balancer will then forward and distribute incoming traffic across the nodes.

To implement a LoadBalancer Service, Kubernetes typically starts off by making changes that are equivalent to you requesting a NodePort Service by exposing the same ports across all cluster nodes.

The load balancer will then forward and distribute incoming traffic across the nodes (helping you overcome the inherent NodePort limitations).



But the load balancer remains external to the cluster. In other words, deploying a LoadBalancer Service in Kubernetes is like sending an email to your hosting provider that says:

> I would like to expose an internal application running on these servers. Could you please assign a public IP for this application and configure your external load balancer accordingly?

When using managed Kubernetes services, the cloud providers will deploy and configure that load balancing for you.

If you're managing your own clusters on-premises, you can use Cilium instead to act as a load balancer network provider. Cilium can assign an external IP address (with the Load Balancer IP Address Management [LB-IPAM] feature) and then make this IP address accessible by BGP or ARP as you will learn in upcoming sections.

If you're interested in how load-balancing actually works in Kubernetes, the section "How do Load Balancing and NAT work under the hood?" will provide you with more information.

Where's my web proxy?

As a network engineer, you may have used web proxies, Layer 7 load balancers, and SSL VPN gateways - devices that let you load balance, route, and control HTTP/HTTPS traffic as it enters your network.

In Kubernetes, this role is often referred to as Ingress and was initially supported by the Kubernetes Ingress API and, most recently, by the Kubernetes Gateway API.

Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

In order for the Ingress resource to work, the cluster must have an Ingress Controller running.

One of the benefits of Cilium is that it is the only CNI that can also act as an ingress controller. This means you don't need to install another tool to support the Ingress functionality. Let's review a sample Cilium Ingress configuration.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: basic-ingress
  namespace: default
spec:
  ingressClassName: cilium
  rules:
  - http:
      paths:
      - backend:
          service:
            name: details
            port:
              number: 9080
        path: /details
        pathType: Prefix
      - backend:
          service:
            name: productpage
            port:
              number: 9080
        path: /
        pathType: Prefix
```

By applying a manifest such as the one above, the Cilium ingress controller would create a Service of type LoadBalancer.

The route above would direct HTTP requests for the path `/details` to the `details` Service, and `/` to the `productpage` Service. Assuming $HTTP_INGRESS is the IP address assigned to the Service, you can access the `details` Service from outside the cluster, as shown in the output below:

```
$ curl --fail -s http://"$HTTP_INGRESS"/details/1 | jq
{
  "id": 1,
  "author": "William Shakespeare",
  "year": 1595,
  "type": "paperback",
  "pages": 200,
  "publisher": "PublisherA",
  "language": "English",
  "ISBN-10": "1234567890",
  "ISBN-13": "123-1234567890"
}
```

You can see from this non-exhaustive list on the Kubernetes Ingress Controllers page that many implementations of the Ingress API exist. The challenge with Ingress API was that every Ingress implementation became different from one to another. It made changing Ingresses very difficult: migrating from one Ingress to another would be like copying your Juniper configuration onto your Cisco router and expecting good results.

To address this shortcoming, the Ingress API is being superseded by Gateway API. Gateway API is portable and inter-compatible, meaning that you can easily migrate from one Gateway API implementation to another.

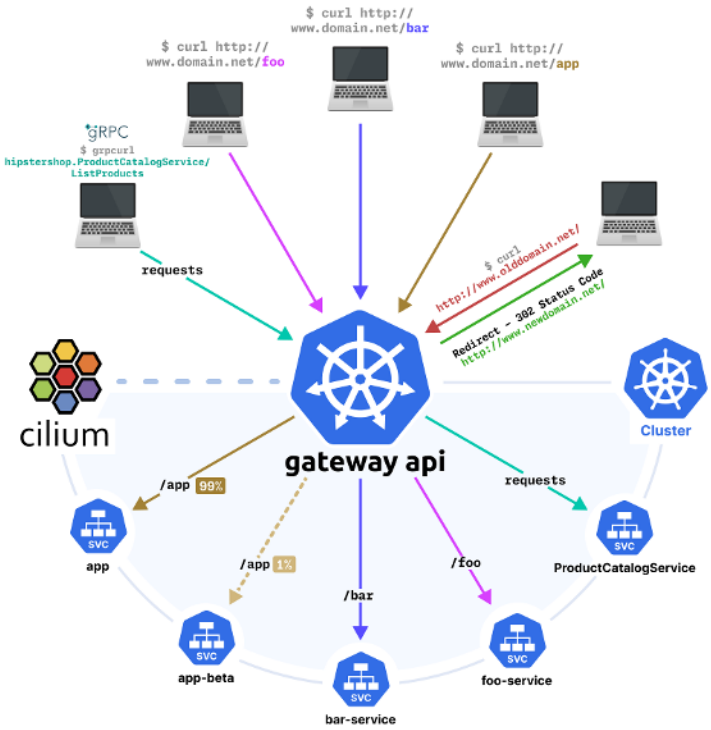
Again, Cilium natively supports Gateway API; removing the need to install another tool for ingress routing.

The equivalent Gateway API configuration to the one above would be the one below:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: my-gateway
spec:
  gatewayClassName: cilium
  listeners:
  - protocol: HTTP
    port: 80
    name: web-gw
    allowedRoutes:
      namespaces:
        from: Same
---
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: http-app-1
spec:
  parentRefs:
  - name: my-gateway
    namespace: default
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /details
    backendRefs:
    - name: details
      port: 9080
  - matches:
    - headers:
      - type: Exact
        name: magic
        value: foo
      queryParams:
      - type: Exact
        name: great
        value: example
      path:
        type: PathPrefix
        value: /
      method: GET
    backendRefs:
    - name: productpage
      port: 9080
```

One of the other advantages of using Gateway API is that it expands beyond just the HTTP path-based routing use case that Ingress API was based upon.

Gateway API supports a multitude of routing capabilities, including traffic splitting, URL redirection, path rewrites, mirroring, HTTP request/response manipulation, TLS termination and passthrough, and other capabilities that were only possible in Ingress through custom annotations.



Gateway API Use Cases

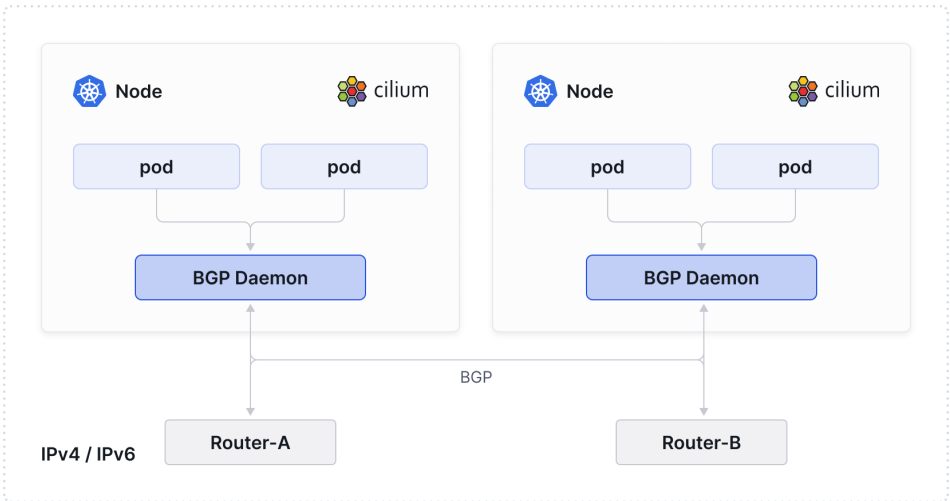
How can I connect my cluster to existing networks?

There are multiple ways to connect your cluster to your existing network, which might depend on the routing mode selected ("native" or "encapsulation"), how the network is configured, which level of resilience and traffic engineering is expected, etc.

Network engineers might be pleased to know one of the options to connect Kubernetes clusters to the rest of the network is through our beloved Border Gateway Protocol (BGP).

BGP support in Kubernetes comes in different forms. While it is possible to install and manage your own BGP daemons in your cluster to connect your cluster to the rest of the network, this presents additional complexity. The more elegant method is to use Cilium's built-in support for BGP.

Cilium lets you build a BGP peering session over IPv4 or IPv6 and advertises the LoadBalancer Service IPs or the PodCIDR ranges to the rest of the network. Peering sessions are typically established with a Top-of-Rack (ToR) device.



Cilium supports the most commonly used BGP features - MD5-based session authentication, customization of BGP timers, communities, local preferences, graceful restart, etc...

Configuring it is simple - simply apply a CiliumBGPPeeringPolicy like the following:

```
---
apiVersion: "cilium.io/v2alpha1"
kind: CiliumBGPClusterConfig
metadata:
  name: bgp-tor
spec:
  nodeSelector:
    matchLabels:
      kubernetes.io/hostname: bgp-node
  - name: "instance-65001"
    localASN: 65001
    peers:
    - name: "peer-65000-tor"
      peerASN: 65000
      peerAddress: "172.0.0.1"
      peerConfigRef:
        name: "peer-config"
---
apiVersion: "cilium.io/v2alpha1"
kind: CiliumBGPPeerConfig
metadata:
  name: peer-config
spec:
  families:
    - afi: ipv4
      safi: unicast
      advertisements:
        matchLabels:
          advertise: "pod-cidr"
  timers:
    connectRetryTimeSeconds: 12
    holdTimeSeconds: 9
    keepAliveTimeSeconds: 3
  ebgpMultihop: 10
  gracefulRestart:
    enabled: true
    restartTimeSeconds: 20
  authSecretRef: bgp-auth-password
---
apiVersion: "cilium.io/v2alpha1"
kind: CiliumBGPAdvertisement
metadata:
  name: pod-cidr
  labels:
    advertise: pod-cidr
spec:
  advertisements:
    - advertisementType: "PodCIDR"
```

Assuming your peer is a Cisco device, it may have a BGP configuration such as:

```
router bgp 65000
 neighbor 172.0.0.2 remote-as 65001
 neighbor 172.0.0.2 connect-retry-time 12
 neighbor 172.0.0.2 timers 9 3
 neighbor 172.0.0.2 ebgp-multihop 10
 neighbor 172.0.0.2 graceful-restart
 neighbor 172.0.0.2 timers graceful-restart 20
 neighbor 172.0.0.2 password bgp-auth-password
```

Once the peering session is established and routes advertised and exchanged, the rest of the network can access applications hosted in the Kubernetes cluster.

You can even use the Cilium CLI to verify that peering sessions have been established and that routes are being advertised. Here is another example:

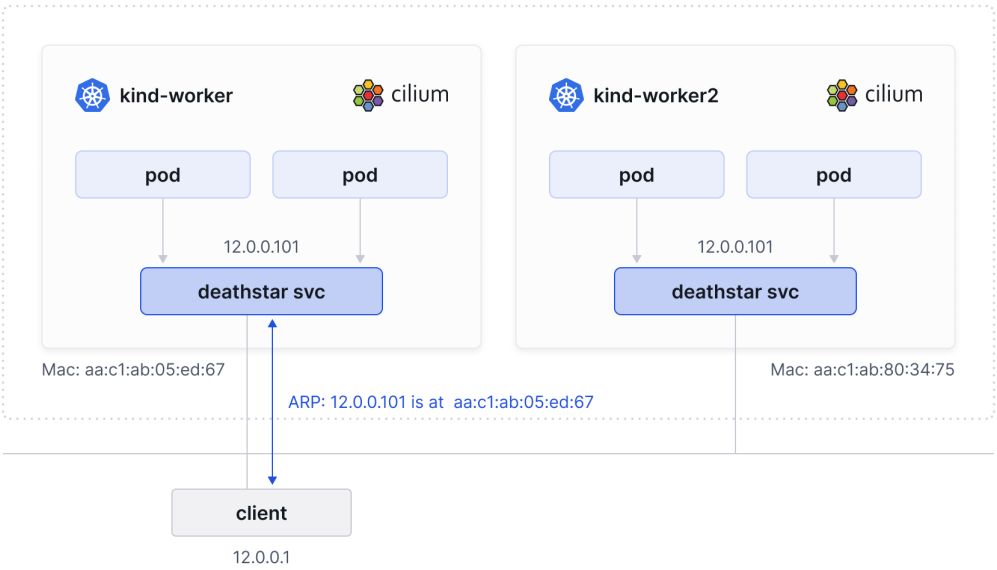
```
$ cilium bgp peers
Node                Local AS   Peer AS   Peer Address     Session State   Uptime     Family         Received   Advertised
kind-control-plane  65001      65000     fd00:10:0:1::1   established     1h43m54s   ipv4/unicast   3          1
                                                                                      ipv6/unicast   3          1
kind-worker         65002      65000     fd00:10:0:2::1   established     1h43m51s   ipv4/unicast   3          1
                                                                                      ipv6/unicast   3          1
kind-worker2        65003      65000     fd00:10:0:3::1   established     1h43m52s   ipv4/unicast   3          1
                                                                                      ipv6/unicast   3          1
```

I don't have any BGP-capable device in my existing network. Is there another way for me to access my Kubernetes services?

If you would rather not use BGP or if you don't have any BGP-capable device in your network, you can advertise the networks locally, over Layer 2, using another protocol known to all network engineers: Address Resolution Protocol (ARP).

If you have local clients on the network that need access to a Service, they need to know which MAC address to send their traffic to. Cilium leverages ARP to announce the MAC address associated with a particular IP.

Let's walk through an example. In the image below, our client is on the same network as the Kubernetes LoadBalancer Service. In this case, you don't need BGP; the client simply needs to understand which MAC address is associated with the IP address of our service so that it knows to which host it needs to send the traffic.



Cilium can reply to ARP requests from clients for LoadBalancer IPs and enable clients to access their services. In this instance, the client (with an IP of `12.0.0.1`) wants to access a Service at `12.0.0.101` and sends an ARP request as a broadcast asking, "Who has 12.0.0.101? Tell 12.0.0.1".

```
No. -   Time -      Source -          Destination -     Protocol -   Length - Info -
1       0.000000    aa:c1:ab:24:a1:a0                   ARP          48       Who has 12.0.0.101? Tell 12.0.0.1
2       0.000050    aa:c1:ab:05:ed:67                   ARP          48       12.0.0.101 is at aa:c1:ab:05:ed:67
3       0.000191    1a:be:5a:19:b4:8b                   ARP          48       Who has 10.244.2.11? Tell 10.244.2.19
4       0.000197    3a:b4:7e:ba:bc:61                   ARP          48       10.244.2.11 is at 3a:b4:7e:ba:bc:61

[+] Frame 1: 48 bytes on wire (384 bits), 48 bytes captured (384 bits)
[+] Linux cooked capture v2
[-] Address Resolution Protocol (request)
     Hardware type: Ethernet (1)
     Protocol type: IPv4 (0x0800)
     Hardware size: 6
     Protocol size: 4
     Opcode: request (1)
     Sender MAC address: aa:c1:ab:24:a1:a0 (aa:c1:ab:24:a1:a0)
     Sender IP address: 12.0.0.1
     Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
     Target IP address: 12.0.0.101
```

The Cilium node will reply to the client with its MAC address ("12.0.0.101 is at aa:c1:ab:05:ed:67") and the client will be able to access the service.

```
1       0.000000    aa:c1:ab:24:a1:a0                   ARP          48       Who has 12.0.0.101? Tell 12.0.0.1
2       0.000050    aa:c1:ab:05:ed:67                   ARP          48       12.0.0.101 is at aa:c1:ab:05:ed:67
3       0.000191    1a:be:5a:19:b4:8b                   ARP          48       Who has 10.244.2.11? Tell 10.244.2.19
4       0.000197    3a:b4:7e:ba:bc:61                   ARP          48       10.244.2.11 is at 3a:b4:7e:ba:bc:61

[+] Frame 2: 48 bytes on wire (384 bits), 48 bytes captured (384 bits)
[+] Linux cooked capture v2
[-] Address Resolution Protocol (reply)
     Hardware type: Ethernet (1)
     Protocol type: IPv4 (0x0800)
     Hardware size: 6
     Protocol size: 4
     Opcode: reply (2)
     Sender MAC address: aa:c1:ab:05:ed:67 (aa:c1:ab:05:ed:67)
     Sender IP address: 12.0.0.101
     Target MAC address: aa:c1:ab:24:a1:a0 (aa:c1:ab:24:a1:a0)
     Target IP address: 12.0.0.1
```

The feature is often referred to as “Layer 2 (L2) Announcements” and is typically for very small environments or home labs as BGP’s flexibility and scalability characteristics are better suited for data center networks.

How do I connect my cluster with an external firewall?

The applications running on your Kubernetes cluster will eventually need to access external resources, including data on the Internet. Of course, you will want to ensure your applications are protected and that your Internet or external-facing firewall controls Internet-bound traffic from the cluster.

However, the challenge when connecting your traditional firewall (from vendors such as Palo Alto Networks, CheckPoint, or Cisco) is that they typically have little to no awareness of Kubernetes objects like namespace or labels. When a packet leaves the cluster, its source IP is typically masqueraded (or source NATed) to that of the node it was on.

Your firewall will see that traffic coming in but will have no idea where the traffic originates - if it’s from a particular pod or the node itself. To address this, CNIs like Cilium support a feature called **Egress Gateway**.

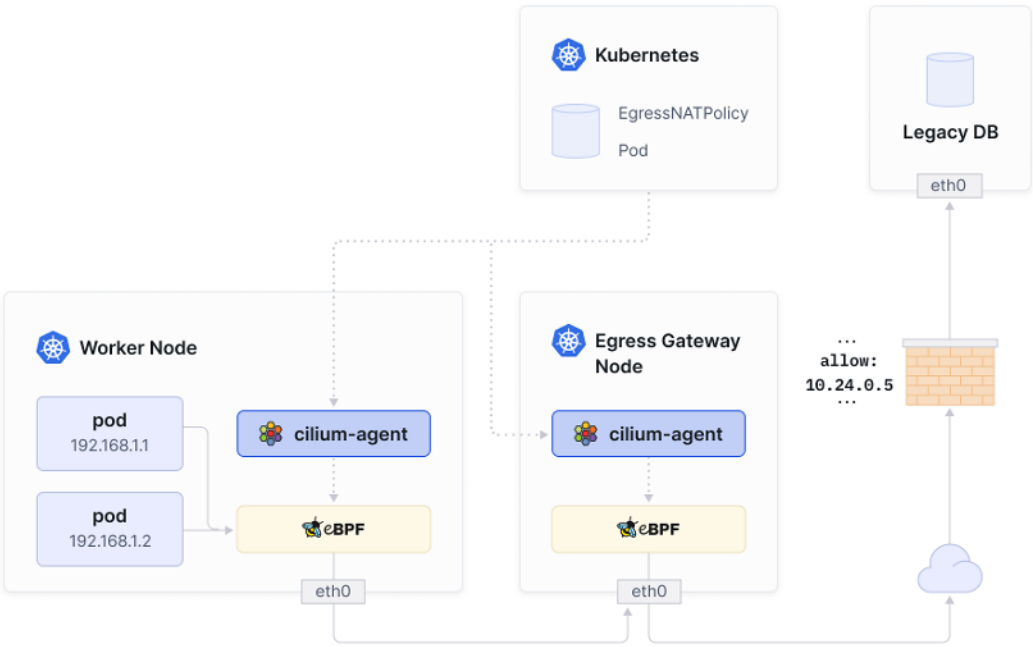
Egress Gateway is essentially Kubernetes-Aware Source-NAT: for example, for pods in a particular namespace, we can force the traffic to exit via a certain interface and to be NATed with a specific IP, enabling the firewall to apply a rule accordingly.

Let's take a look at a sample `CiliumEgressGatewayPolicy`:

```
apiVersion: cilium.io/v2
kind: CiliumEgressGatewayPolicy
metadata:
  name: egress-sample
spec:
  selectors:
  - podSelector:
      matchLabels:
        org: empire
        io.kubernetes.pod.namespace: default
 destinationCIDRs:
  - "0.0.0.0/0"
  egressGateway:
    nodeSelector:
      matchLabels:
        node.kubernetes.io/name: a-specific-node
    egressIP: 10.168.60.100
```

What the policy above achieves is the following: when pods with the `org: empire` label and located in the `default` namespace send traffic outside the cluster, the traffic will exit via a node with a specific label, and the packets will be source-NAT with `10.168.60.100`, which should be an IP associated with a network interface on this particular node.

You now know traffic originating from your `org:empire` pods will have the source IP `10.168.60.100`, and you can configure the appropriate security rules in your firewalls based on that IP.



The only drawback with this model is that the Egress node becomes a single point of failure. For this reason, Isovalent's enterprise edition of Cilium includes Egress Gateway High Availability (HA), which supports multiple egress nodes. The nodes acting as egress gateways will then load-balance traffic in a round-robin fashion and provide fallback nodes in case one or more egress nodes fail.

How do internal Load Balancing and NAT work under the hood?

In the traditional networking world, performance is dictated by many factors - the configuration and the choice of network protocols, the network interface cards, the underlying hardware, etc...

In Kubernetes, all these factors still matter (the software has to run on some hardware, after all), but one decision engineers must make as their Kubernetes environment grows is whether to use kube-proxy.

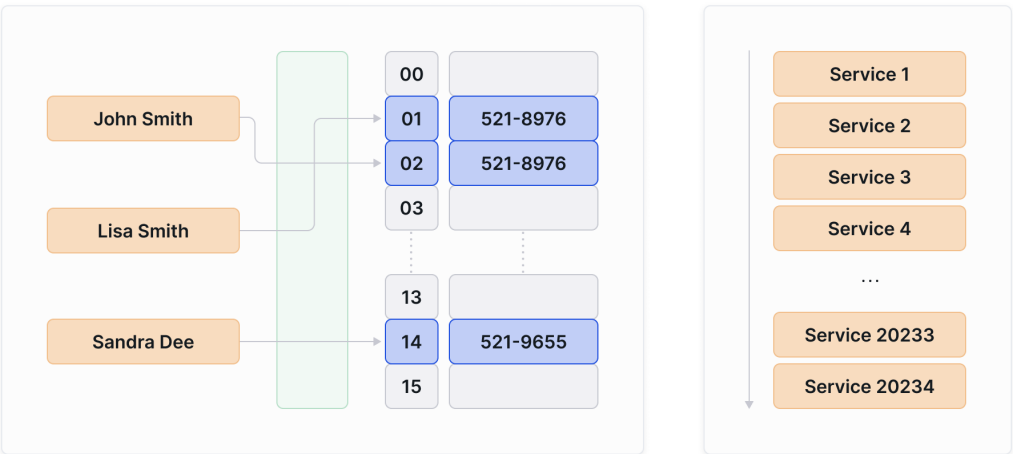
By default, kube-proxy is installed on each node, ensuring network traffic flows from services to the appropriate pods. When a new service or endpoint is created, it abstracts the underlying pod IPs, allowing other services to communicate using the service's virtual IP address.

Despite what its name might suggest, kube-proxy is not an actual proxy - rather, it watches the Kubernetes control plane to then ensure that rules on the underlying operating system packet filtering are up-to-date. It manages network rules necessary for routing traffic, including Network Address Translation (NAT) when required, to ensure packets reach their intended destinations. The problem is that kube-proxy was originally based on a 25-year-old technology (iptables), which wasn't designed for the churn of Kubernetes.

The best comparison I can come up with is this: imagine walking into a data center today and installing a Cisco PIX to secure your AI workloads.

Yikes.

The main challenge with iptables' performance is that it relies on a sequential algorithm. Iptables goes through each rule individually in the table to match rules against observed traffic. This means that the time taken scales linearly with each added rule, and performance strain quickly becomes apparent as more services and endpoints are added. A packet has to traverse each rule to find a match, introducing latency and instability.



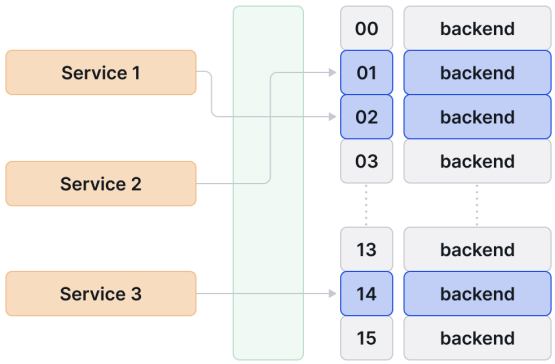
While Kubernetes users can now use kube-proxy in the better-performing ipvs mode, it is still reliant on netfilter. The alternative is to use Cilium's eBPF-based kube-proxy replacement.

eBPF is a revolutionary Linux technology that you may have come across. You don't have to know about eBPF to install and operate Cilium - after all, most of us network engineers operate networks without understanding how ASICs are programmed - but a brief primer doesn't hurt.

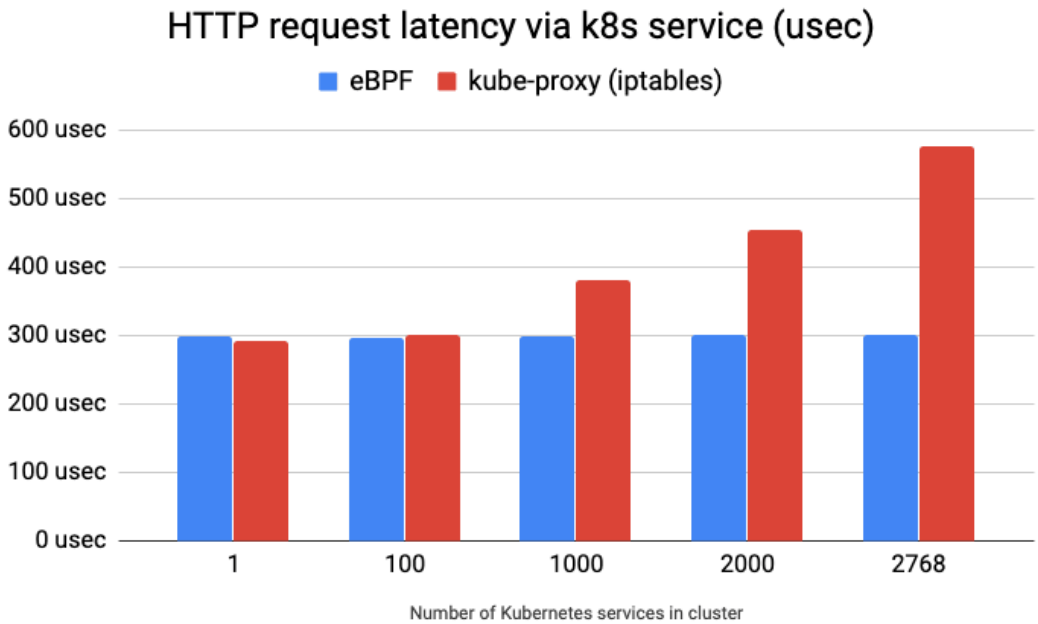
eBPF is best known for the ability to safely run virtual networking and security programs within the Linux kernel. It resembles the virtual network function (VNF) concept in Telco networks or service insertion for VMware NSX or Cisco ACI users.

Here is another comparison: our beloved Cisco Catalyst 6500 switches were initially used purely for switching packets. Eventually we started inserting dedicated line cards in the chassis to support routing, security, and load-balancing functionalities. You can think of eBPF-based tools like Cilium as service modules you can insert into your Linux kernel.

Cilium provides a kube-proxy replacement that relies on efficient eBPF hash tables. eBPF maps are the mechanism in the Linux kernel used by eBPF programs to store and retrieve data efficiently.



Cilium's eBPF kube-proxy replacement benefits are particularly evident at scale. While iptables struggles with latency as the number of microservices grows, you can see the efficiency of eBPF in the performance benchmarking graphic below.



How do I manage and troubleshoot my network?

In the traditional networking world, network engineers have relied on trusted tools and systems to troubleshoot. Here is a non-exhaustive list of tools you may have in your toolbox and their equivalent for Kubernetes and Cilium users:

Use case	Traditional Networking	Kubernetes Networking
check TCP/IP connectivity	ping/traceroute	ping
check HTTP connectivity	curl	curl
check the status of the network	IOS CLI	kubectl or cilium CLI
capture logs from network	IOS CLI and Syslog	kubectl logs
capture traffic patterns and bandwidth usage	NetFlow/sFlow	Hubble
analyse network traffic	tcpdump/Wireshark	tcpdump/Wireshark/Hubble
generate traffic for performance testing	iperf	iperf

In your network, you may have run some of the connectivity tooling commands from your routers and devices or straight from your terminal. In your Kubernetes cluster, you may need to deploy a monitoring pod for troubleshooting purposes. I recommend the Kubernetes networking troubleshooting container netshoot as it includes tools such as tcpdump and iperf amongst many others.

```
$ kubectl run -it debug-pod --image nicolaka/netshoot
If you don't see a command prompt, try pressing enter.

                    dP            dP                           dP
                    88            88                           88
88d888b. .d8888b. d8888P .d8888b. 88d888b. .d8888b. .d8888b. d8888P
88'  `88 88ooood8   88   Y8ooooo. 88'  `88 88'  `88 88'  `88   88
88    88 88.  ...   88         88 88    88 88.  .88 88.  .88   88
dP    dP `88888P'   dP   `88888P' dP    dP `88888P' `88888P'   dP

Welcome to Netshoot! (github.com/nicolaka/netshoot)
Version: 0.12
 debug-pod  ~
 debug-pod  ~  tcpdump
tcpdump: verbose output suppressed, use -v[v]... for full protocol
decode
listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144
bytes
13:32:10.717686 IP6 fe80::80a8:fff:fed9:c3f2 > ff02::2: ICMP6, router
solicitation, length 16
13:32:10.766010 ARP, Request who-has 10.1.2.205 tell debug-pod, length
28
13:32:10.766022 ARP, Reply 10.1.2.205 is-at fa:fa:8e:ba:b2:2b (oui
Unknown), length 28
```

Even better, you can use `kubectl debug` to deploy an ephemeral container in an existing pod to debug networking issues (remember they share the same networking stack):

```
$ kubectl debug backend_pod -it --image=nicolaka/netshoot  -- tcpdump
-i eth0
11:30:15.778113 IP 10.244.2.149.59950 > infra-backend-
v2-664bffc4-8hbll.3000: Flags [F.], seq 235, ack 627, win 503, options
[nop,nop,TS val 4109474445 ecr 389226307], length 0
11:30:15.778233 IP infra-backend-v2-664bffc4-8hbll.3000 >
10.244.2.149.59950: Flags [F.], seq 627, ack 236, win 501, options
[nop,nop,TS val 389286309 ecr 4109474445], length 0
```

How can I monitor and visualize network traffic?

In your traditional network, you may use Netflow/sFlow to collect and export network flows in the following format:

Source IP	Destination IP	Packets	Bytes
192.0.2.2	198.51.100.5	20	500
203.0.113.129	198.51.100.3	50	1250
192.0.2.6	203.0.113.68	10	1000

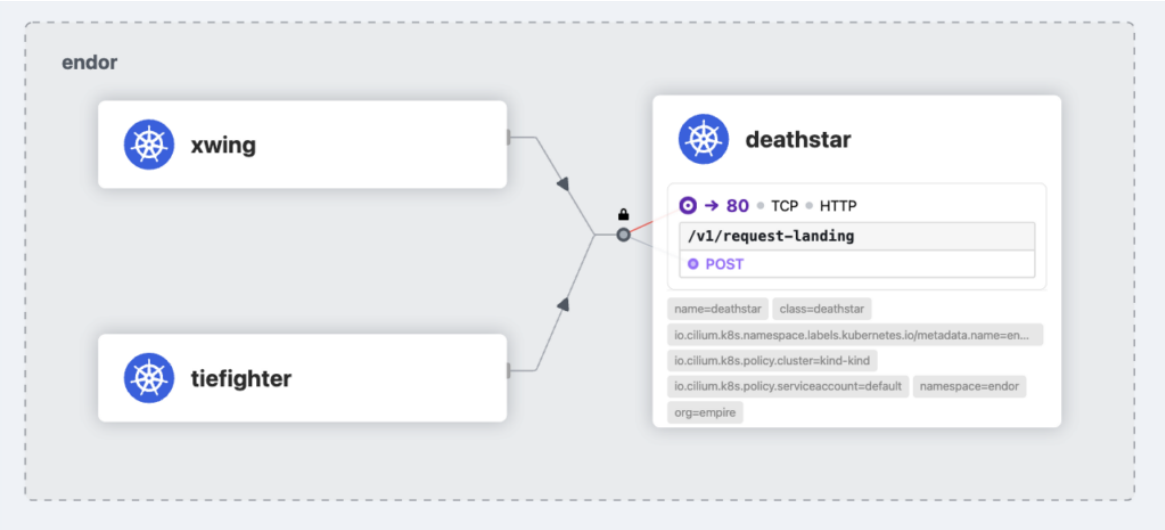
Your NetFlow collector might then be able to represent the flows in a graphical interface and build a visual representation of who's talking to whom. To do the same in Kubernetes, you'd typically use **Hubble**.

Hubble is the network observability tool that comes with Cilium - think of it as a combination of NetFlow and Wireshark. By using eBPF, Hubble can hook into the network and extract networking data. Hubble comes with its own CLI and UI.

You can use the CLI to observe all the flows in the cluster, with the option to use filters to, for example, only see traffic from the `tiefighter` pod in the `endor` namespace.

```
$ hubble observe --from-pod endor/tiefighter
Feb 23 13:59:22.041: endor/tiefighter-76d85c5887-hl5hc:56038 (ID:11220) -> endor/
deathstar-7848d6c4d5-5bdqn:80 (ID:16163) to-endpoint FORWARDED (TCP Flags: ACK, FIN)
```

The Hubble UI lets you visualize traffic in a Kubernetes cluster as a service map. The example below shows three pod identities in the `endor` namespace:



Both `xwing` and `tiefighter` pods are trying to access the Deathstar. Note the granularity of the flow information you get from Hubble. You can see that the traffic was HTTP over the `/v1/request-landing` HTTP path, using the POST method.

While the open source edition of Cilium includes a standard version of Hubble, the enterprise edition of Cilium includes an advanced enterprise edition of Hubble, with multi-tenant self-service access, historical flow and analytics data, and a built-in network policy editor (described in more details at the end of the next section).

How do I start securing my cluster?

Clusters are typically deployed without network policies: as is the case in traditional networking, we are all often guilty of bolting security on later.

Deploying network policies without disrupting the applications running on the cluster presents a couple of challenges for operators: 1) How do I deploy network policies in a transparent manner without breaking the cluster? and 2) How do I even know which security rules to enforce?

The first challenge can be addressed by selecting the Cilium Network Policy enforcement mode. In the `default` enforcement mode, endpoints start without restriction but as soon as a rule matches, traffic has to be explicitly allowed or traffic will be dropped. In the `never` enforcement mode, all traffic is permitted. This is an audit-only mode you can run to see what the policy might catch before enforcing it.

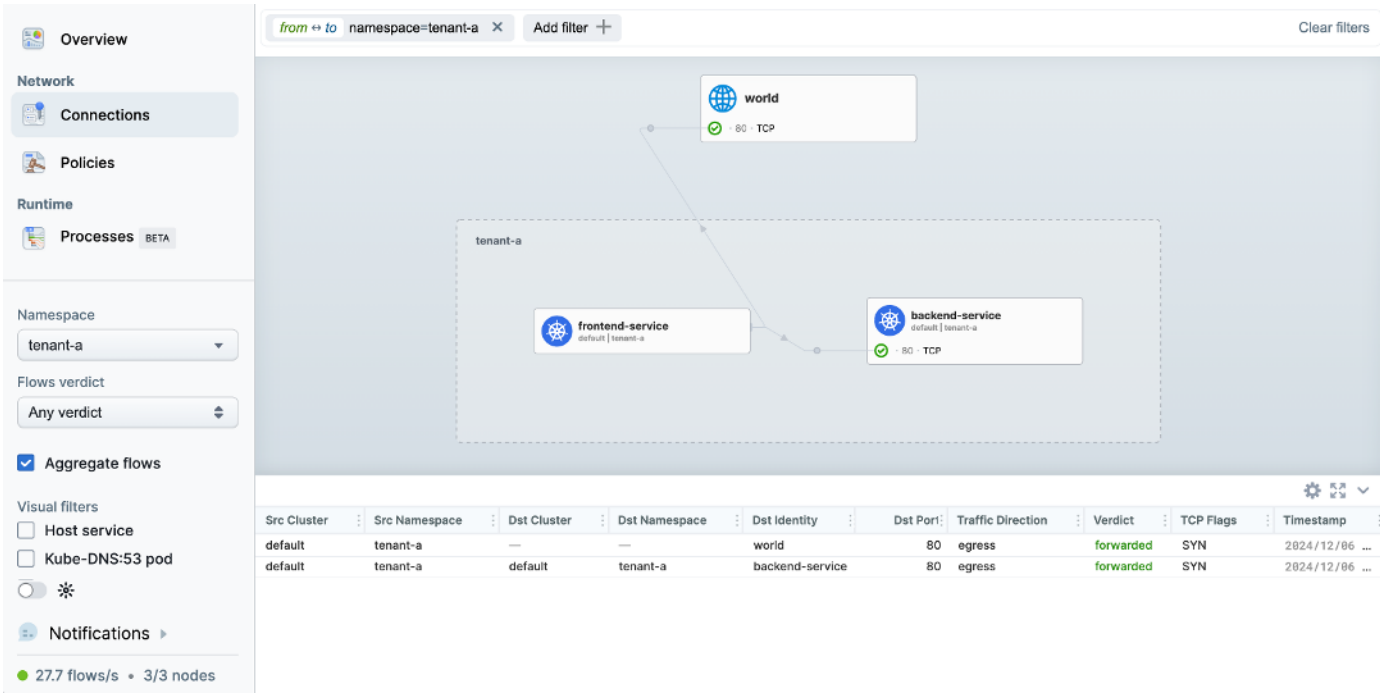
> The more security-savvy network engineers that want strict, or zero trust, network security from the start could also set the policy enforcement mode to `always` in which Cilium will block all traffic unless there is a Network Policy explicitly allowing it.

The second challenge can be tackled in various manners. You can rely on the developers to tell you how the micro-services are inter-connected, over which ports they are communicating, which API they're using, etc... The problem is that it can be difficult to understand the developer's intent. Let's face it - most developers don't always know how micro-services are interconnected!

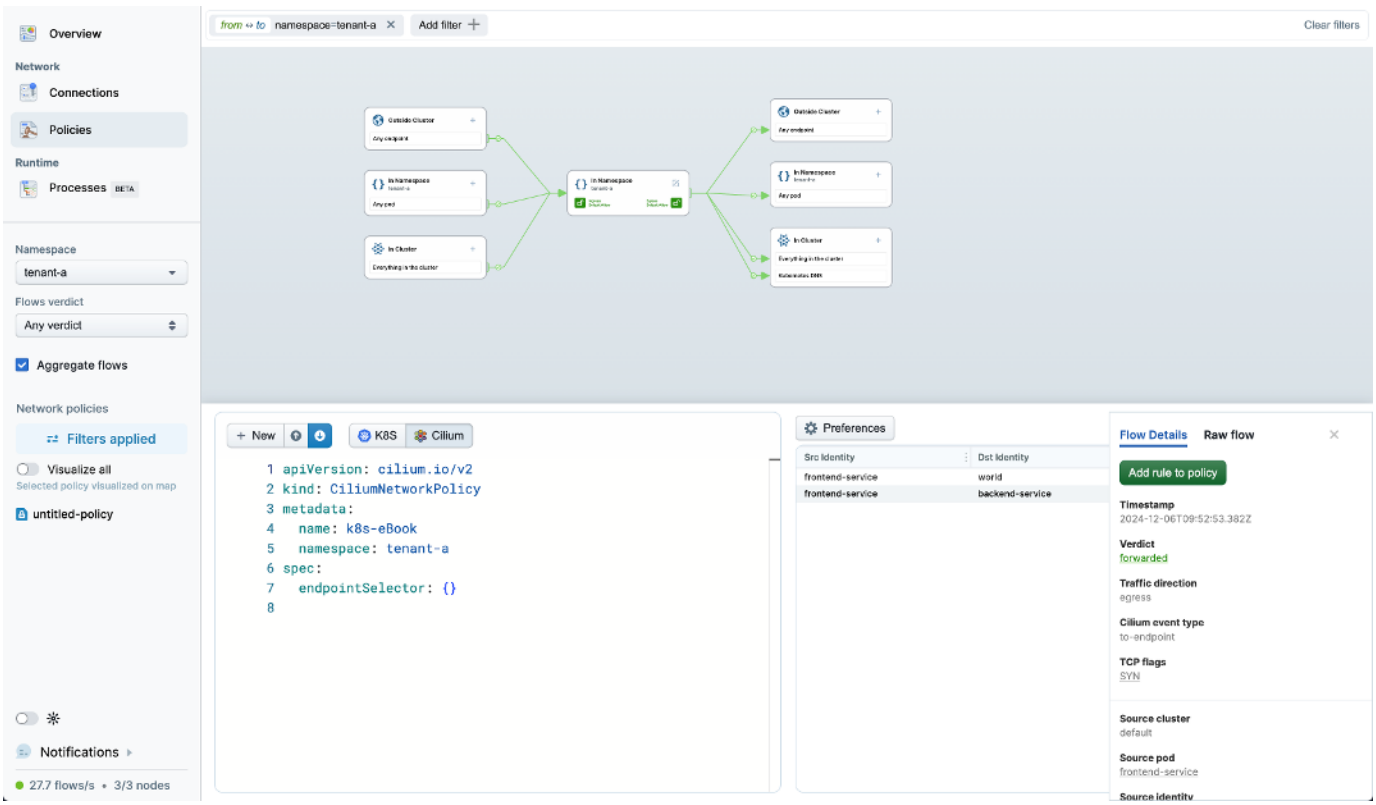
Alternatively, you could deploy Hubble, observe the traffic, and create network policies based on the traffic. This works fine although it can be time-consuming.

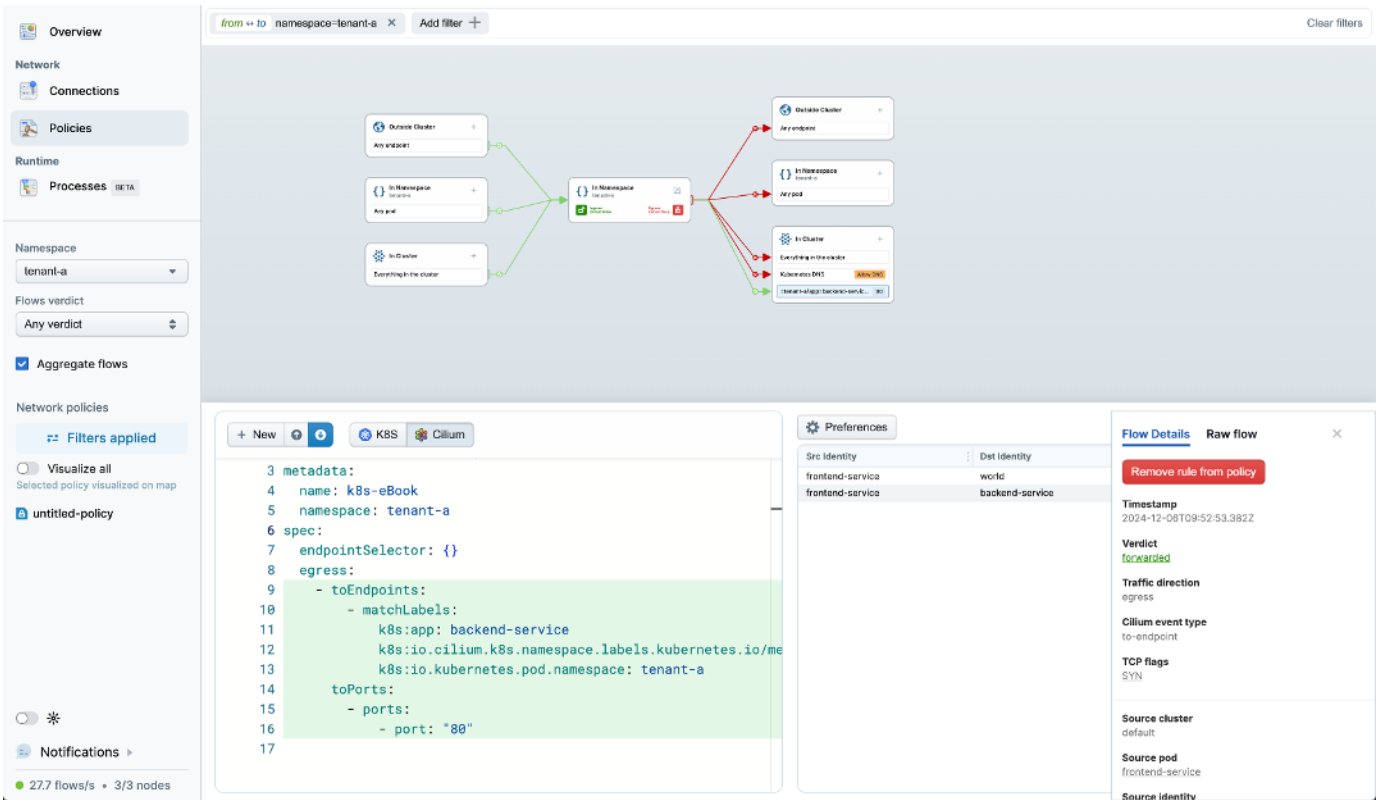
This is an area where Isovalent is here to help. Firstly, Isovalent provides a free, online Network Policy Editor designed to ease the cognitive overhead of developing network policies. It lets you visualize policies, allowing users to easily create policies before downloading them in YAML files.

Additionally, Isovalent's Enterprise Edition of Cilium includes an advanced enterprise Hubble view that is designed to greatly simplify and accelerate network policy generation.



It lists all the flows in the cluster (which can be filtered by namespace) and by clicking on a particular flow, you can automatically create a Cilium Network Policy based on this particular flow. The Network Policy can then be downloaded as a YAML file and applied to the cluster.



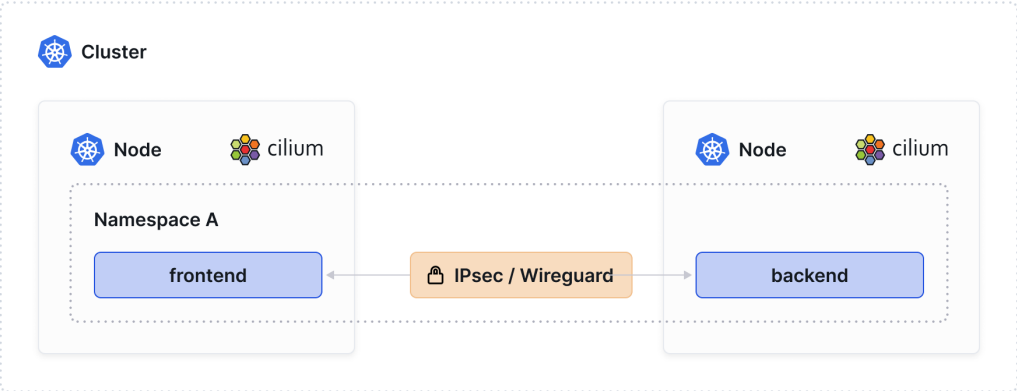


How do I encrypt the traffic in my cluster?

Encrypting data in transit is a common requirement. Whether over public networks or even within your data centers, you probably have considered how to protect your data and ensure you meet certain regulatory requirements.

Over the Internet, between data centers, or even within a network, you may have used IPsec-based VPNs to encrypt the traffic between two different sites. Those who can afford the specialized crypto hardware may have used technologies such as MACsec (defined by the IEEE 802.1AE standard) to encrypt traffic on Ethernet links.

The requirement can also exist in Kubernetes and might be even more common, given Kubernetes clusters are often deployed on shared hardware in cloud data centers. Cilium's Transparent Encryption feature set offers two options for encrypting traffic between pods on different nodes (or even between clusters): IPsec and WireGuard.



The IPsec method lets you choose the algorithm and cipher and manage the keys (just as you would do when setting IPsec between two VPN endpoints in your network), while the WireGuard option is opinionated: it leverages very robust cryptography and does not let the user choose ciphers and protocols.

Both methods ensure that traffic between pods is encrypted as long as pods are located on different nodes.

How do we connect clusters together?

As a network engineer, you have probably been tasked to design resilient and highly resilient solutions across multiple sites. You may also have had to connect both networks together, often over a Layer 3 network, and use an intersite control plane such as BGP EVPN VXLAN to distribute network information. You may have also implemented some form of load balancing across multiple sites. You would have had to consider how to enforce consistent security policy across multiple sites.

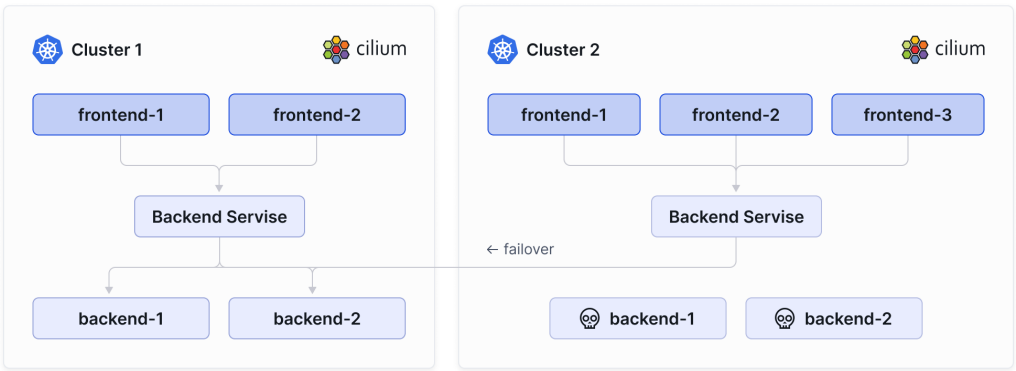
In Kubernetes, these challenges remain - in a multi-cluster topology, you have multiple clusters located across different zones for resiliency and high availability. While it might be relatively trivial to set up IP connectivity between the clusters, they would have no awareness of each other's objects, such as pods, services, namespaces, labels, and other metadata.

As we've seen, network policies and load balancing often rely on such metadata. Therefore, load-balancing across clusters or applying security policies across multiple clusters is not possible in a standard Kubernetes cluster implementation.

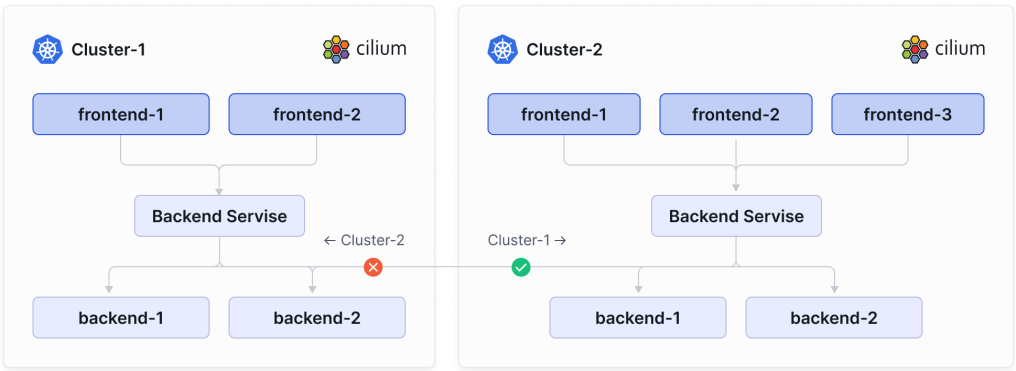
This is where you might need a multi-cluster solution like Cilium Cluster Mesh.

Cilium Cluster Mesh addresses these requirements by federating multiple Cilium instances on different clusters. Cluster Mesh automatically discovers services across all meshed clusters and load balances the traffic effectively.

One of the main use cases for using Cluster Mesh is High Availability. This use case includes operating Kubernetes clusters in multiple regions or availability zones and running the replicas of the same services in each cluster. Traffic will by default be load-balanced to all endpoints across clusters. When all endpoints in a given cluster fail, traffic will be forwarded to the remaining endpoints in other clusters.



With Cilium Cluster Mesh, you can enforce network policy to workloads spanning multiple clusters.



In the example above, the frontend pods on cluster-1 are allowed to connect to backend pods running in cluster-2. However, frontend pods on cluster-2 are not allowed to connect to backend pods running in cluster-1.

Below is an example of a Cilium Network Policy that can be applied in such a configuration.

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "ingress-to-backend"
spec:
  description: "Allow frontend in cluster-1 to contact backend in
cluster2"
  endpointSelector:
    matchLabels:
      name: backend
      io.cilium.k8s.policy.cluster: cluster-2
  ingress:
  - fromEndpoints:
    - matchLabels:
        name: frontend
        io.cilium.k8s.policy.cluster: cluster-1
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
```

Is IPv6 supported on Kubernetes?

Yes. Kubernetes has supported IPv6-only clusters since Kubernetes 1.18 and Dual Stack IPv4/IPv6 reached General Availability (GA) in Kubernetes 1.23.

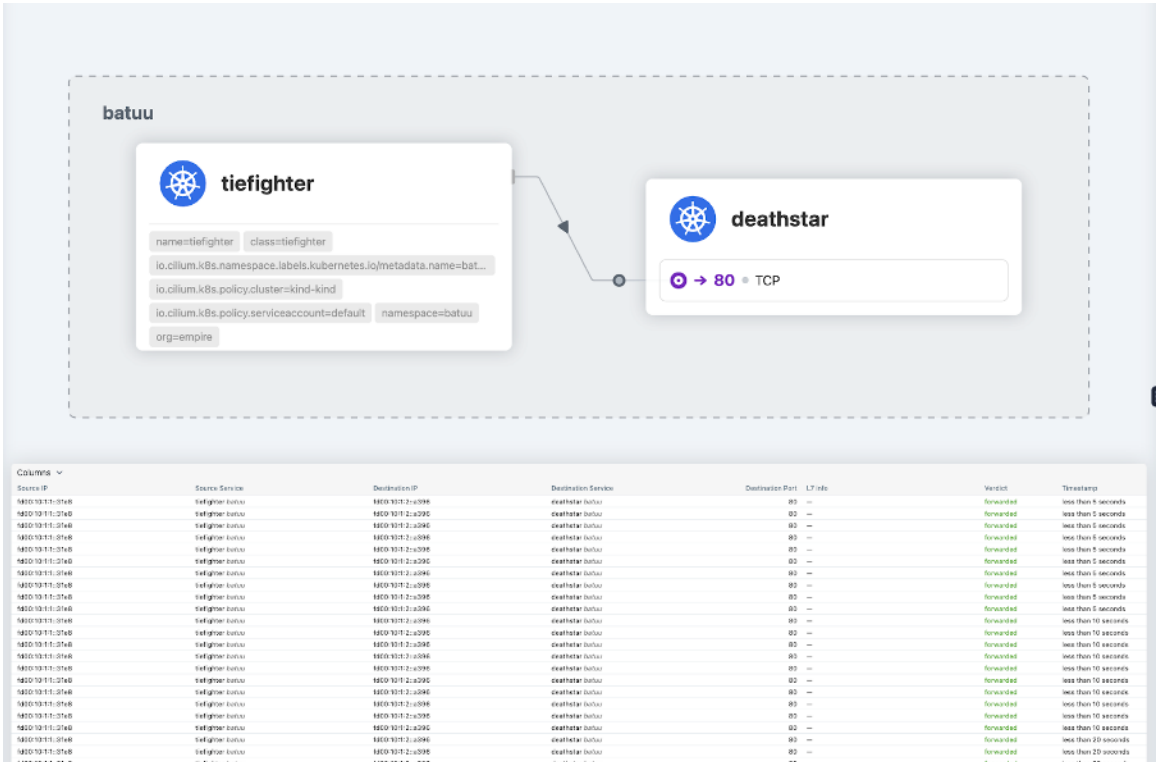
Like network policies, this depends entirely on whether the CNI plugin supports this feature: not all CNIs support IPv6-only or Dual Stack. The good news: Cilium supports IPv4-only, IPv6-only, and Dual Stack IPv4/IPv6 clusters.

However, the challenge isn't necessarily whether your pod picks up an IPv6 address at deployment; it's whether you can operate and troubleshoot an IPv6 cluster. Thankfully, Hubble - mentioned earlier - greatly simplifies IPv6 operations.

You can, for example, easily find all IPv6 traffic from a particular pod:

```
$ hubble observe --ipv6 --from-pod pod-worker -o dict --ip-
translation=false --protocol ICMPv6
  TIMESTAMP: Sep  7 15:27:27.615
     SOURCE: fd00:10:244:3::ba17
DESTINATION: fd00:10:244:1::3203
       TYPE: to-overlay
    VERDICT: FORWARDED
    SUMMARY: ICMPv6 EchoRequest
------------
  TIMESTAMP: Sep  7 15:27:27.615
     SOURCE: fd00:10:244:3::ba17
DESTINATION: fd00:10:244:1::3203
       TYPE: to-endpoint
    VERDICT: FORWARDED
    SUMMARY: ICMPv6 EchoRequest
```

You can also visualize IPv6 service connectivity, using the Hubble UI that comes with Cilium OSS:



Does the concept of Quality of Service (QoS) exist in Kubernetes?

Let's distinguish between compute QoS and networking QoS. While you're probably thinking of networking QoS, the Kubernetes docs refer to Quality of Service (QoS) for pods. Kubernetes operators can assign a QoS class, which can be used to determine whether to evict pods when node resources are exceeded. The focus of this feature is to prevent CPU and memory starvation.

As a network engineer, you would have used networking QoS to prevent bandwidth starvation. Networking QoS encompasses various requirements, such as:

- Marking: how a network device labels packets with a specific priority marker to ensure they receive preferential treatment throughout the network
- Congestion management: how a network device manages packets as they wait to exit a device
- Congestion avoidance: how a network device decides if and when to drop packets as a system becomes congested
- Shaping/Policing: how a network device delays or drops packets to ensure that the throughput does not exceed a defined rate

As the Kubernetes platform will be running on a physical network, you may want to enforce some level of quality of service within the physical network (out of scope for this document). Within Kubernetes, you should ensure a fair distribution of the node's network interface bandwidth.

Consider that all pods on a node will typically share a single network interface. If a pod consumes an unfair share of the available bandwidth, the other pods on the node might suffer.

With Cilium's Bandwidth Manager, you can mark a Pod by adding an annotation to its specification and shape traffic as it egresses the pod. See an example below:

```
---
apiVersion: v1
kind: Pod
metadata:
  annotations:
    # Limits egress bandwidth to 10Mbit/s.
    kubernetes.io/egress-bandwidth: "10M"
  labels:
    app.kubernetes.io/name: netperf-server
  name: netperf-server
spec:
  containers:
  - name: netperf
    image: cilium/netperf
    ports:
    - containerPort: 12865
```

When running a throughput performance test, you would see the egress traffic limited down to just under 10 Mbps:

```
$ kubectl apply -f netperf.yaml
pod/netperf-server created
pod/netperf-client created
$ NETPERF_SERVER_IP=$(kubectl get pod netperf-server -o
jsonpath='{.status.podIP}')
$ kubectl exec netperf-client -- \
    netperf -t TCP_MAERTS -H "${NETPERF_SERVER_IP}"
MIGRATED TCP MAERTS TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
10.108.2.172 (10.108.) port 0 AF_INET
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

87380  16384  16384    10.00       9.54
```

Which Kubernetes networking options are available in managed Kubernetes services?

With most organizations following a cloud-first approach, it is no surprise that many Kubernetes clusters are hosted in cloud providers such as AWS, Azure, and Google Cloud. Some choose to build and manage clusters themselves on platforms such as AWS IaaS service EC2. Still, most organizations tend to gravitate towards the managed Kubernetes services offerings, such as Azure Kubernetes Service (AKS), AWS Elastic Kubernetes Service (EKS) and Google Kubernetes Service (GKE). In these services, the Kubernetes control plane components are managed on your behalf - you get access to your worker nodes and consume your pool of compute resources as you see fit.

We would need a separate white paper to cover the diversity of Kubernetes networking options in cloud offerings and given how quickly it evolves, it would soon become outdated. However, the majority of the concepts highlighted in this eBook apply. Cilium is either the default networking layer or often recommended by managed service providers and Kubernetes distribution maintainers.

Take AKS and GKE, for example - GKE's datapath v2 is based on Cilium, while the preferred option for large-scale clusters on AKS is the "Azure CNI powered by Cilium" mode.

Given that these cloud-hosted Kubernetes clusters are managed services, you don't get to choose the Cilium version and the features available.

If you'd like access to all Cilium features, you can, for example, use AKS's "Bring-Your-Own-CNI" option. It lets you deploy a cluster without a CNI and install the one of your choice. While this model enables you to install and customize Cilium fully, it comes with the caveat that Microsoft support couldn't help with CNI-related issues.

Head to https://isovalent.com/partners/ for up-to-date information on how the various cloud providers and technology partners are standardizing on Cilium.

Conclusion

I hope this book has helped you understand the core building blocks of Kubernetes networking, how pods can communicate with each other and with the outside world, how you can secure the applications running in your cluster, and how you can operate and manage it.

But there are many intricacies to Kubernetes Networking and Cilium. To keep this eBook brief, we made editorial decisions that experienced Kubernetes architects might question: we decided to omit concepts and tools like headless services, networking namespaces, service meshes, multicast, etc...We also recognize we barely scratched the surface of what eBPF is capable and didn't even mention Tetragon.

We will save them for another volume.

We greatly welcome your feedback and comments. Feel free to get in touch on social media to share your thoughts. You can find me on LinkedIn at @nicolasvibert.

Learn More with the Isovalent Hands-On Labs

As useful as this eBook may have been to grasp the key concepts of Kubernetes networking and the many Cilium use cases, nothing beats practical experience. To that effect, Isovalent introduced a program of free, online, hands-on labs.

By undertaking these labs, you will learn about different features and use cases of Cilium, Hubble, Tetragon, and their Isovalent Enterprise editions. You will also earn digital badges: in late 2022, we started an accreditation program to recognize the efforts of the individuals who had completed the labs.

In the past 12 months alone, over 37,000 labs have been played by over 12,000 engineers.

The labs are constantly maintained and updated to the latest Cilium version. At the time of writing, there are 33 publicly available labs.

Here are some of the labs that cover functionalities highlighted in this eBook.

Getting Started with Cilium

In this hands-on lab we provide you a fully fledged Cilium installation and a few challenges to solve. See for yourself how Cilium works by securing a moon-sized battlestation in a "Star Wars"-inspired challenge.

Getting Started with Cilium Lab

Cilium Network Policies

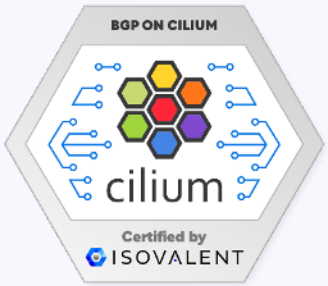
Achieving zero-trust network connectivity via Kubernetes Network Policy is complex. Enter Isovalent Enterprise for Cilium: it provides tooling to simplify and automate the creation of Network Policy.

Isovalent Enterprise for Cilium: Network Policies Lab

BGP on Cilium

In order to connect Kubernetes and the existing network infrastructure, BGP is needed. Cilium offers native support for BGP, exposing Kubernetes to the outside and all the while simplifying users' deployments.

BGP on Cilium Lab



Learn More with the Isovalent Hands-On Labs

Cilium Gateway API

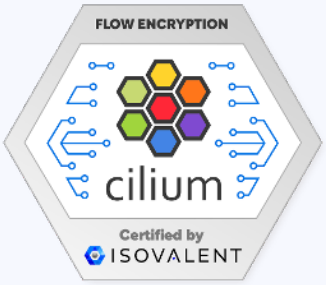
In this lab, you will learn how you can use the Cilium Gateway API functionality to route HTTP and HTTPS traffic into your Kubernetes-hosted application.

Cilium Gateway API Lab

Cilium Transparent Encryption

Cilium provides two options to encrypt traffic: IPsec and WireGuard. In this lab, you will be installing and testing both features and will get to experience how easy it is to encrypt data in transit with Cilium.

Cilium Transparent Encryption with IPSec and WireGuard Lab



Cilium Egress Gateway

Cilium's Egress Gateway feature allows you to specify which nodes should be used by a pod in order to reach the outside world.

Cilium Egress Gateway Lab

IPv6 Networking and Observability

This lab will walk you through how to deploy a IPv4/IPv6 Dual Stack Kubernetes cluster and install Cilium and Hubble to benefit from their networking and observability capabilities.

Cilium IPv6 Networking and Observability Lab

Introducing Cilium, Hubble, and Tetragon

What is Cilium?

Cilium is an open source, cloud native solution for providing, securing, and observing network connectivity between workloads. Cilium was created by Isovalent and is currently the only CNCF graduated networking plugin project.

Cilium is best known as a CNI (Container Network Interface) and provides secure and observable connectivity at the network and service mesh level inside Kubernetes and beyond. It is fueled by the revolutionary Kernel technology eBPF. Google (GKE, Anthos), Amazon (EKS-A), and Microsoft (AKS) have all adopted Cilium to provide networking and security for Kubernetes. Cilium is used by platform engineering teams including Adobe, Bell Canada, ByteDance, Capital One, Datadog, IKEA, Schuberg Philis, and Sky.

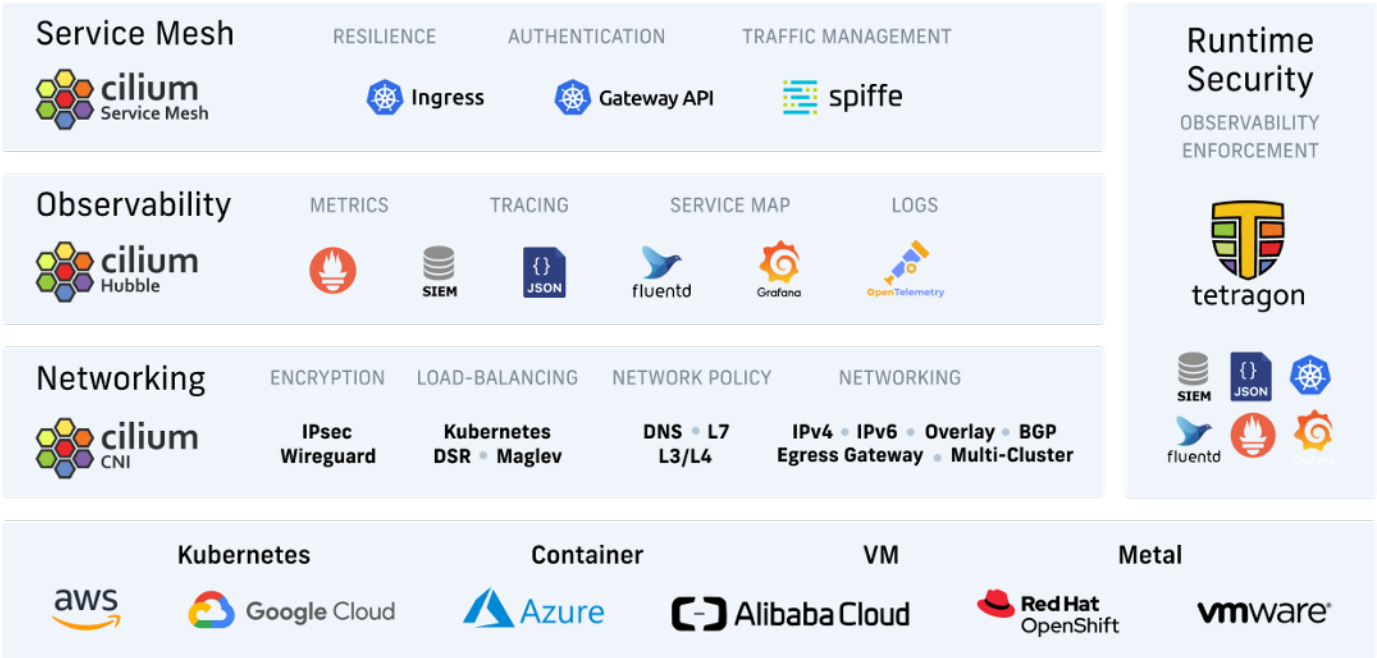


Figure 1: Overview of Cilium Suite

What is Hubble?

Hubble is a fully distributed networking and security observability platform built on top of Cilium and eBPF. It enables deep visibility into the communication and behaviour of services and networking infrastructure in a completely transparent manner.

What is Tetragon?

Tetragon is a flexible, low overhead eBPF-based security observability and runtime enforcement agent. Tetragon is Kubernetes-aware and able to detect and react to security-significant events directly in the kernel, monitoring all processes executed in the system, sensitive files opened, egress network connections made and Linux credentials and namespaces that were changed.