eBook:

Introduction to Kubernetes Networking and Security

A step-by-step guide covering everything you need to know to confidently approach Kubernetes networking, starting with basic networking concepts, all the way through to advanced Kubernetes networking with eBPF.

TIGERA

Table of Contents

Introduction

Kubernetes has become the de facto standard for managing containerized workloads and services. By facilitating declarative configuration and automation, it massively simplifies the operational complexities of reliably orchestrating compute, storage, and networking. The Kubernetes networking model is a core enabler of that simplification, defining a common set of behaviors that makes communication between microservices easy and portable across any environment.

This guide arms the reader with the knowledge they need to understand the essentials of how Kubernetes networking works, how it helps simplify communication between microservices, and how to secure your cluster networking following today's best practices.

Starting from an introduction to basic networking concepts, the book builds knowledge in easily digestible increments across the breadth of Kubernetes networking, leading up to some of the latest advances in networking extensions and implementations.

About the Author:

Alex Pollitt

Alex Pollitt, is CTO and co-founder of Tigera, the team that created Project Calico, the leading open source networking and network security solution for Kubernetes. With a background in developing carrier grade networking software, he was an early contributor to the Kubernetes community and helped influence early Kubernetes networking ideas that are today taken for granted.
Twitter: @lxpollitt.

Networking

You can get up and running with Calico by following any of the Calico install guides without needing to be a networking expert. Calico hides the complexities for you. However, if you would like to learn more about networking so you can better understand what is happening under the covers, this guide provides a short introduction to some of the key fundamental networking concepts for anyone who is not already familiar with them.

In this chapter you will learn:

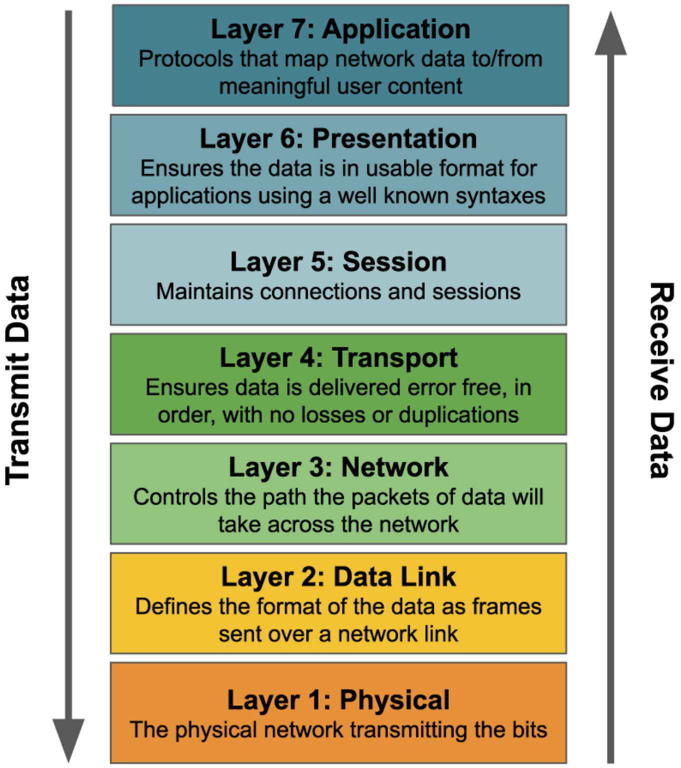
- The terms used to described different layers of the network.
- The anatomy of a network packet.
- What MTU is and why it makes a difference.
- How IP addressing, subnets, and IP routing works.
- What an overlay network is.
- What DNS and NAT are.

Network layers

The process of sending and receiving data over a network is commonly categorized into 7 layers (referred to as the OSI model). They layers are typically abbreviated as L1 - L7. You can think of data as passing through each of these layers in turn as it is sent or received from an application, which each layer being responsible for a particular part of the processing required to send the data over the network.

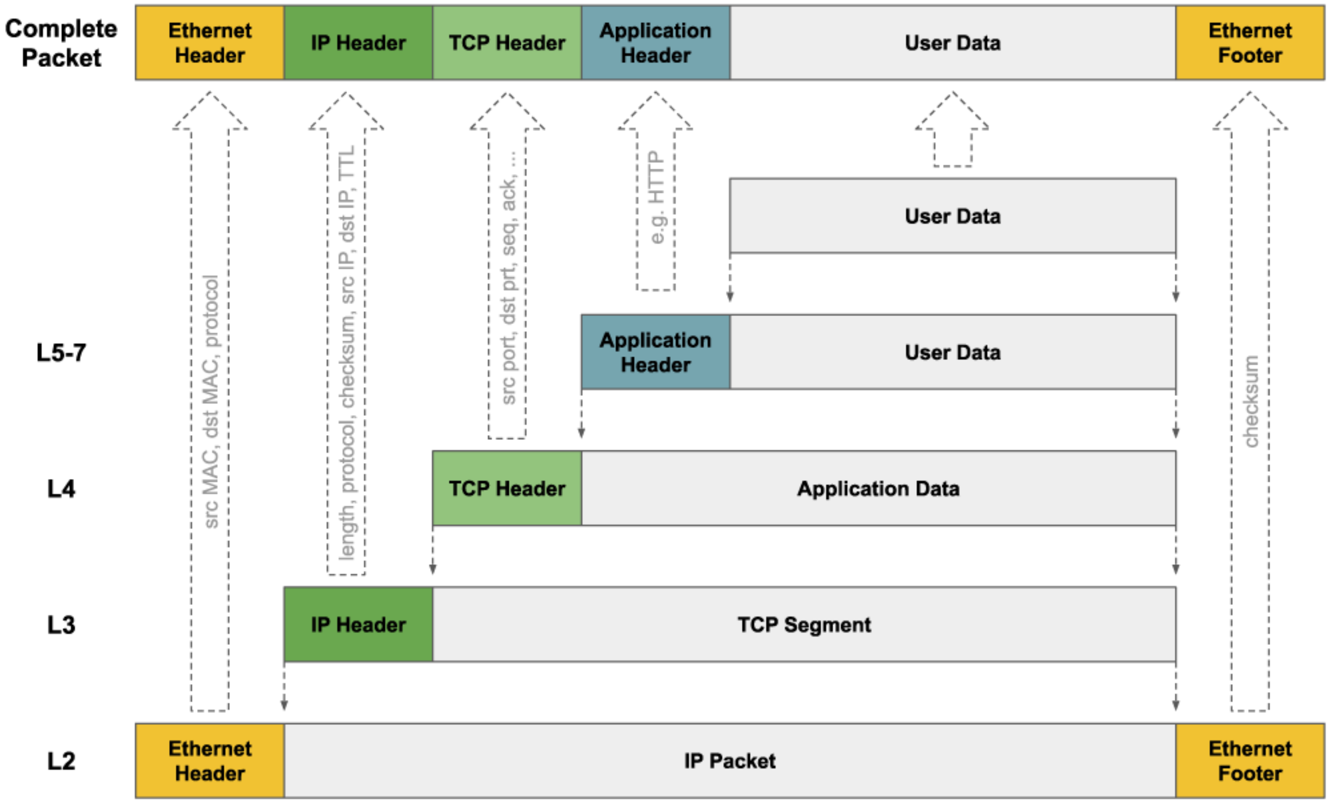
In a modern enterprise or public cloud network, the layers commonly map as follows:

- L5-7: all the protocols most application developers are familiar with. e.g. HTTP, FTP, SSH, SSL, DNS.
- L4: TCP or UDP, including source and destination ports.
- L3: IP packets and IP routing.
- L2: Ethernet packets and Ethernet switching.


Transmit Data
Layer 7: Application
Protocols that map network data to/from meaningful user content
Layer 6: Presentation
Ensures the data is in usable format for applications using a well known syntaxes
Layer 5: Session
Maintains connections and sessions
Layer 4: Transport
Ensures data is delivered error free, in order, with no losses or duplications
Layer 3: Network
Controls the path the packets of data will take across the network
Layer 2: Data Link
Defines the format of the data as frames sent over a network link
Layer 1: Physical
The physical network transmitting the bits
Receive Data

Anatomy of a network packet

When sending data over the network, each layer in the network stack adds its own header containing the control/metadata the layer needs in order to process the packet as it traverses the network, passing the resulting packet on to the next layer of the stack. In this way the complete packet is produced, which includes all the control/metadata required by every layer of the stack, without any layer understanding the data or needing to process the control/metadata of adjacent network layers.



IP addressing, subnets and IP routing

The L3 network layer introduces IP addresses and typically marks the boundary between the part of networking that application developers care about, and the part of networking that network engineers care about. In particular application developers typically regard IP addresses as the source and destination of the network traffic, but have much less of a need to understand L3 routing or anything lower in the network stack, which is more the domain of network engineers.

There are two variants of IP addresses: IPv4 and IPv6.

- IPv4 addresses are 32 bits long and the most commonly used. They are typically represented as 4 bytes in decimal (each 0-255) separated by dots. e.g. `192.168.27.64` . There are several ranges of IP addresses that are reserved as “private”, that can only be used within local private networks, are not routable across the internet. These can be reused by enterprises as often as they want to. In contrast “public” IP addresses are globally unique across the whole of the internet. As the number of network devices and networks connected to the internet has grown, public IPv4 addresses are now in short supply.

- IPv6 addresses are 128 bits long and designed to overcome the shortage of IPv4 address space. They are typically represented by 8 groups of 4 digit hexadecimal numbers. e.g. `1203:8fe0:fe80:b897:8990:8a7c:99bf:323d` . Due to the 128 bit length, there's no shortage of IPv6 addresses. However, many enterprises have been slow to adopt IPv6, so for now at least, IPv4 remains the default for many enterprise and data center networks.

Groups of IP addresses are typically represented using **CIDR notation** that consists of an IP address and number of significant bits on the IP address separated by a `/` . For example, `192.168.27.0/24` represents the group of 256 IP addresses from `192.168.27.0` to `192.168.27.255` .

A group of IP addresses within a single L2 network is referred to as a subnet. Within a subnet, packets can be sent between any pair of devices as a single network hop, based solely on the L2 header (and footer).

To send packets beyond a single subnet requires L3 routing, with each L3 network device (router) being responsible for making decisions on the path to send the packet based on L3 routing rules. Each network device acting as a router has routes that determine where a packet for a particular CIDR should be sent next. So for example, in a Linux system, a route of `10.48.0.128/26 via 10.0.0.12 dev eth0` indicates that packets with destination IP address in `10.48.0.128/26` should be routed to a next network hop of `10.0.0.12` over the `eth0` interface.

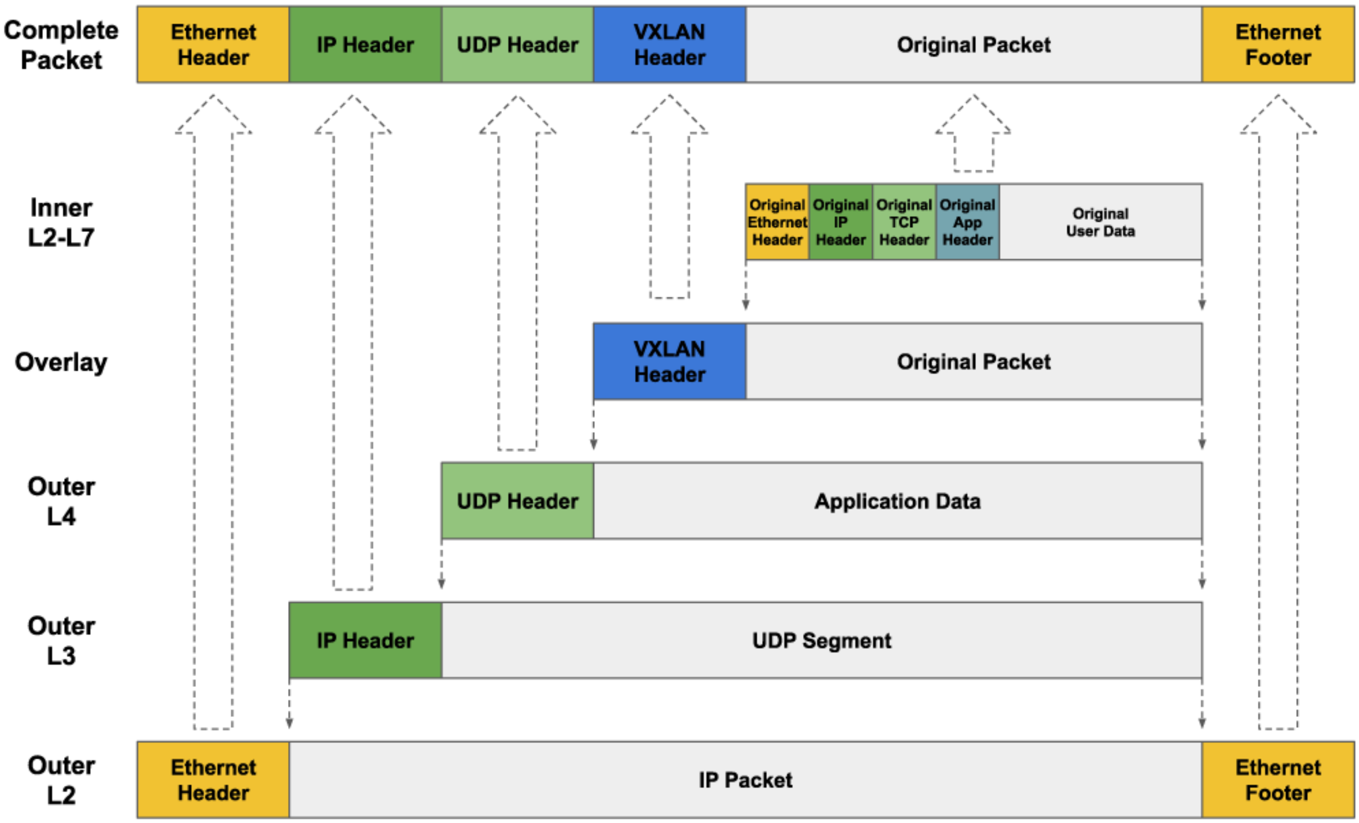
Routes can be configured statically by an administrator, or programmed dynamically using routing protocols. When using routing protocols each network device typically needs to be configured to tell it which other network devices it should be exchanging routes with. The routing protocol then handles programming the right routes across the whole of the network as devices are added or removed, or network links come in or out of service.

One common routing protocol used in large enterprise and data center networks is **BGP**. BGP is one of the main protocols that powers the internet, so scales incredibly well, and is very widely supported by modern routers.

Overlay networks

An overlay network allows network devices to communicate across an underlying network (referred to as the underlay) without the underlay network having any knowledge of the devices connected to the overlay network. From the point of view of the devices connected to the overlay network, it looks just like a normal network. There are many different kinds of overlay networks that use different protocols to make this happen, but in general they share the same common characteristic of taking a network packet, referred to as the inner packet, and encapsulating it inside an outer network packet. In this way the underlay sees the outer packets without needing to understand how to handle the inner packets.

How the overlay knows where to send packets varies by overlay type and the protocols they use. Similarly exactly how the packet is wrapped varies between different overlay types. In the case of VXLAN for example, the inner packet is wrapped and sent as UDP in the outer packet.



Overlay networks have the advantage of having minimal dependencies on the underlying network infrastructure, but have the downsides of:

- having a small performance impact compared to non-overlay networking, which you might want to avoid if running network intensive workloads
- workloads on the overlay are not easily addressable from the rest of the network. so NAT gateways or load balancers are required to bridge between the overlay and the underlay network for any ingress to, or egress from, the overlay.

Calico networking options are exceptionally flexible, so in general you can choose whether you prefer Calico to provide an overlay network, or non-overlay network. You can read more about this in the Calico **determine best networking option** guide.

DNS

While the underlying network packet flow across a the network is determined using IP addresses, users and applications typically want to use well known names to identify network destinations that remain consistent over time, even if the underlying IP addresses change. For example, to map `google.com` to `216.58.210.46` . This translation from name to IP address is handled by **DNS**. DNS runs on top of the base networking described so far. Each device connected to a network is typically configured with the IP addresses one or more DNS servers. When an application wants to connect to a domain name, a DNS message is sent to the DNS server, which then responds with information about which IP address(es) the domain name maps to. The application can then initiate its connection to the chosen IP address.

NAT

Network Address Translation (NAT) is the process of mapping an IP address in a packet to a different IP address as the packet passes through the device performing the NAT. Depending on the use case, NAT can apply to the source or destination IP address, or to both addresses.

One common use case for NAT is to allow devices with private IP address to talk to devices with public IP address across the internet. For example, if a device with a private IP address attempts to connect to a public IP address, then the router at the border of the private network will typically use SNAT (Source Network Address Translation) to map the private source IP address of tha packet to the router's own public IP address before forwarding it on to the internet. The router then maps response packets coming in the opposite direction back to the original private IP address, so packets flow end-to-end in both directions, with neither source or destination being aware the mapping is happening. The same technique is commonly used to allow devices connected to an overlay network to connect with devices outside of the overlay network.

Another common use case for NAT for load balancing. In this case the load balancer performs DNAT (Destination Network Address Translation) to change the destination IP address of the incoming connection to the IP address of the chosen device it is load balancing to. The load balancer then reverses this NAT on response packets so neither source or destination device is aware the mapping is happening.

MTU

The Maximum Transmission Unit (MTU) of a network link is the maximum size of packet that can be sent across that network link. It is common for all links in a network to be configured with the same MTU to reduce the need to fragment packets as they traverse the network, which can significantly lower the performance of the network. In addition, TCP tries to learn path MTUs, and adjust packet sizes for each network path based on the smallest MTU of any of the links in the network path. When an application tries to send more data than can fit in a single packet, TCP will fragment the data into multiple TCP segments, so the MTU is not exceeded.

Most networks have links with an MTU of 1,500 bytes, but some networks support MTUs of 9,000 bytes. In a Linux system, larger MTU sizes can result in lower CPU being used by the Linux networking stack when sending large amounts of data, because it has to process fewer packets for the same amount of data. Depending on the network interface hardware being used, some of this overhead may be offloaded to the network interface hardware, so the impact of small vs large MTU sizes varies from device to device.

Kubernetes Networking

Kubernetes defines a network model that helps provide simplicity and consistency across a range of networking environments and network implementations. The Kubernetes network model provides the foundation for understanding how containers, pods, and services within Kubernetes communicate with each other. This guide explains the key concepts and how they fit together.

In this chapter you will learn:

- The fundamental network behaviors the Kubernetes network model defines.
- How Kubernetes works with a variety of different network implementations.
- What Kubernetes Services are.
- How DNS works within Kubernetes.
- What “NAT outgoing” is and when you would want to use it.
- What “dual stack” is.

The Kubernetes network model

The Kubernetes network model specifies:

- Every pod gets its own IP address
- Containers within a pod share the pod IP address and can communicate freely with each other
- Pods can communicate with all other pods in the cluster using pod IP addresses (without NAT)
- Isolation (restricting what each pod can communicate with) is defined using network policies

As a result, pods can be treated much like VMs or hosts (they all have unique IP addresses), and the containers within pods very much like processes running within a VM or host (they run in the same network namespace and share an IP address). This model makes it easier for applications to be migrated from VMs and hosts to pods managed by Kubernetes. In addition, because isolation is defined using network policies rather than the structure of the network, the network remains simple to understand. This style of network is sometimes referred to as a “flat network”.

Note that, although very rarely needed, Kubernetes does also support the ability to map host ports through to pods, or to run pods directly within the host network namespace sharing the host’s IP address.

Kubernetes network implementations

Kubernetes built in network support, kubenet, can provide some basic network connectivity. However, it is more common to use third party network implementations which plug into Kubernetes using the CNI (Container Network Interface) API.

There are lots of different kinds of CNI plugins, but the two main ones are:

- network plugins, which are responsible for connecting pod to the network
- IPAM (IP Address Management) plugins, which are responsible for allocating pod IP addresses.

Calico provides both network and IPAM plugins, but can also integrate and work seamlessly with some other CNI plugins, including AWS, Azure, and Google network CNI plugins, and the host local IPAM plugin. This flexibility allows you to choose the best networking options for your specific needs and deployment environment. You can read more about this in the Calico **determine best networking option** guide.

Kubernetes Services

Kubernetes **Services** provide a way of abstracting access to a group of pods as a network service. The group of pods is usually defined using a **label selector**. Within the cluster the network service is usually represented as virtual IP address, and kube-proxy load balances connections to the virtual IP across the group of pods backing the service. The virtual IP is discoverable through Kubernetes DNS. The DNS name and virtual IP address remain constant for the life time of the service, even though the pods backing the service may be created or destroyed, and the number of pods backing the service may change over time.

Kubernetes Services can also define how a service accessed from outside of the cluster, for example using

- a node port, where the service can be accessed via a specific port on every node
- or a load balancer, whether a network load balancer provides a virtual IP address that the service can be accessed via from outside the cluster.

Note that when using Calico in on-prem deployments you can also **advertise service IP addresses**, allowing services to be conveniently accessed without going via a node port or load balancer.

Kubernetes DNS

Each Kubernetes cluster provides a DNS service. Every pod and every service is discoverable through the Kubernetes DNS service.

For example:

- Service: `my-svc.my-namespace.svc.cluster-domain.example`
- Pod: `pod-ip-address.my-namespace.pod.cluster-domain.example`
- Pod created by a deployment exposed as a service: `pod-ip-address.deployment-name.my-namespace.svc.cluster-domain.example` .

The DNS service is implemented as Kubernetes Service that maps to one or more DNS server pods (usually CoreDNS), that are scheduled just like any other pod. Pods in the cluster are configured to use the DNS service, with a DNS search list that includes the pod's own namespace and the cluster's default domain.

This means that if there is a service named `foo` in Kubernetes namespace `bar` , then pods in the same namespace can access the service as `foo` , and pods in other namespaces can access the service as `foo.bar`

Kubernetes supports a rich set of options for controlling DNS in different scenarios. You can read more about these in the Kubernetes guide **DNS for Services and Pods**.

NAT outgoing

The Kubernetes network model specifies that pods must be able to communicate with each other directly using pod IP addresses. But it does not mandate that pod IP addresses are routable beyond the boundaries of the cluster. Many Kubernetes network implementations use **overlay networks**. Typically for these deployments, when a pod initiates a connection to an IP address outside of the cluster, the node hosting the pod will SNAT (Source Network Address Translation) map the source address of the packet from the pod IP to the node IP. This enables the connection to be routed across the rest of the network to the destination (because the node IP is routable). Return packets on the connection are automatically mapped back by the node replacing the node IP with the pod IP before forwarding the packet to the pod.

When using Calico, depending on your environment, you can generally choose whether you prefer to run an overlay network, or prefer to have fully routable pod IPs. You can read more about this in the Calico **determine best networking option** guide. Calico also allows you to **configure outgoing NAT** for specific IP address ranges if more granularity is desired.

Dual stack

If you want to use a mix of IPv4 and IPv6 then you can enable Kubernetes **dual-stack** mode. When enabled, all pods will be assigned both an IPv4 and IPv6 address, and Kubernetes Services can specify whether they should be exposed as IPv4 or IPv6 addresses.

Above and beyond

- **The Kubernetes Network Model**
- **Video: Everything you need to know about Kubernetes pod networking on AWS**
- **Video: Everything you need to know about Kubernetes networking on Azure**
- **Video: Everything you need to know about Kubernetes networking on Google Cloud**

Network Policy

Kubernetes and Calico provide network policy APIs to help you secure your workloads.

In this chapter you will learn:

- What network policy is and why it is important.
- The differences between Kubernetes and Calico network policies and when you might want to use each.
- Some best practices for using network policy.

What is network policy?

Network policy is the primary tool for securing a Kubernetes network. It allows you to easily restrict the network traffic in your cluster so only the traffic that you want to flow is allowed.

To understand the significance of network policy, let's briefly explore how network security was typically achieved prior to network policy. Historically in enterprise networks, network security was provided by designing a physical topology of network devices (switches, routers, firewalls) and their associated configuration. The physical topology defined the security boundaries of the network. In the first phase of virtualization, the same network and network device constructs were virtualized in the cloud, and the same techniques for creating specific network topologies of (virtual) network devices were used to provide network security. Adding new applications or services often required additional network design to update the network topology and network device configuration to provide the desired security.

In contrast, the **Kubernetes network model** defines a "flat" network in which every pod can communicate with all other pods in the cluster using pod IP addresses. This approach massively simplifies network design and allows new workloads to be scheduled dynamically anywhere in the cluster with no dependencies on the network design.

In this model, rather than network security being defined by network topology boundaries, it is defined using network policies that are independent of the network topology. Network policies are further abstracted from the network by using label selectors as their primary mechanism for defining which workloads can talk to which workloads, rather than IP addresses or IP address ranges.

Why is network policy important?

In an age where attackers are becoming more and more sophisticated, network security as a line of defense is more important than ever.

While you can (and should) use firewalls to restrict traffic at the perimeters of your network (commonly referred to as north-south traffic), their ability to police Kubernetes traffic is often limited to a granularity of the cluster as a whole, rather than to specific groups of pods, due to the dynamic nature of pod scheduling and pod IP addresses. In addition, the goal of most attackers once they gain a small foothold inside the perimeter is to move laterally (commonly referred to as east-west) to gain access to higher value targets, which perimeter based firewalls can't police against.

Network policy on the other hand is designed for the dynamic nature of Kubernetes by following the standard Kubernetes paradigm of using label selectors to define groups of pods, rather than IP addresses. And because network policy is enforced within the cluster itself it can police both north-south and east-west traffic.

Network policy represents an important evolution of network security, not just because it handles the dynamic nature of modern microservices, but because it empowers dev and devops engineers to easily define network security themselves, rather than needing to learn low-level networking details or raise tickets with a separate team responsible for managing firewalls. Network policy makes it easy to define intent, such as "only this microservice gets to connect to the database", write that intent as code (typically in YAML files), and integrate authoring of network policies into git workflows and CI/CD processes.

Note:
Calico and Calico Enterprise offer capabilities that can help perimeter firewalls integrate more tightly with Kubernetes. However, this does not remove the need or value of network policies within the cluster itself.)

Kubernetes network policy

Kubernetes network policies are defined using the Kubernetes **NetworkPolicy** resource.

The main features of Kubernetes network policies are:

- Policies are namespace scoped (i.e. you create them within the context of a specific namespace just like, for example, pods)
- Policies are applied to pods using label selectors
- Policy rules can specify the traffic that is allowed to/from other pods, namespaces, or CIDRs
- Policy rules can specify protocols (TCP, UDP, SCTP), named ports or port numbers

Kubernetes itself does not enforce network policies, and instead delegates their enforcement to network plugins. Most network plugins implement the mainline elements of Kubernetes network policies, though not all implement every feature of the specification. (Calico does implement every feature, and was the original reference implementation of Kubernetes network policies.)

To learn more about Kubernetes network policies, read the **Get started with Kubernetes network policy** guide.

Calico network policy

In addition to enforcing Kubernetes network policy, Calico supports its own namespaced **NetworkPolicy** and non-namespaced **GlobalNetworkPolicy** resources, which provide additional features beyond those supported by Kubernetes network policy. This includes support for:

- policy ordering/priority
- deny and log actions in rules
- more flexible match criteria for applying policies and in policy rules, including matching on Kubernetes ServiceAccounts, and (if using Istio & Envoy) cryptographic identity and layer 5-7 match criteria such as HTTP & gRPC URLs.
- ability to reference non-Kubernetes workloads in polices, including matching on **NetworkSets** in policy rules

While Kubernetes network policy applies only to pods, Calico network policy can be applied to multiple types of endpoints including pods, VMs, and host interfaces.

To learn more about Calico network policies, read the **Get started with Calico network policy** guide.

Benefits of using Calico for network policy

Full Kubernetes network policy support

Unlike some other network policy implementations, Calico implements the full set of Kubernetes network policy features.

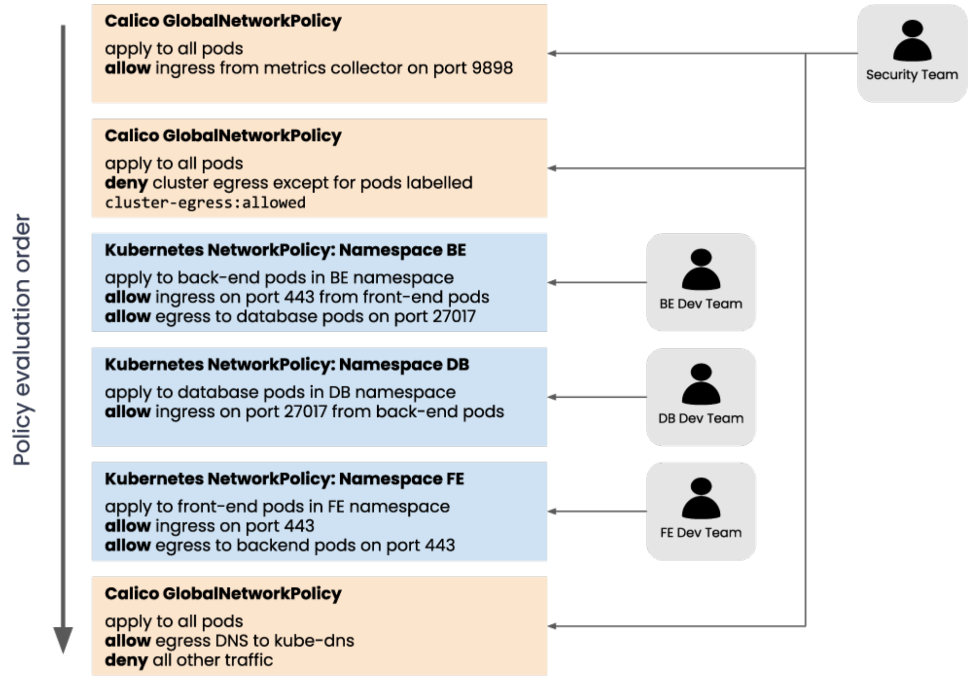
Richer network policy

Calico network policies allow even richer traffic control than Kubernetes network policies if you need it. In addition, Calico network policies allow you to create policy that applies across multiple namespaces using GlobalNetworkPolicy resources.

Mix Kubernetes and Calico network policy

Kubernetes and Calico network policies can be mixed together seamlessly. One common use case for this is to split responsibilities between security / cluster ops teams and developer / service teams. For example, giving the security / cluster ops team RBAC permissions to define Calico policies, and giving developer / service teams RBAC permissions to define Kubernetes network policies in their specific namespaces. As Calico policy rules can be ordered to be enforced either before or after Kubernetes network policies, and can include actions such as deny and log, this allows the security / cluster ops team to define basic higher-level more-general purpose rules, while empowering the developer / service teams to define their own fine-grained constraints on the apps and services they are responsible for.

For more flexible control and delegation of responsibilities between two or more teams, Calico Enterprise extends this model to provide **hierarchical policy**.


Policy evaluation order
Calico GlobalNetworkPolicy
apply to all pods
allow ingress from metrics collector on port 9898
Calico GlobalNetworkPolicy
apply to all pods
deny cluster egress except for pods labelled
cluster-egress:allowed
Kubernetes NetworkPolicy: Namespace BE
apply to back-end pods in BE namespace
allow ingress on port 443 from front-end pods
allow egress to database pods on port 27017
Kubernetes NetworkPolicy: Namespace DB
apply to database pods in DB namespace
allow ingress on port 27017 from back-end pods
Kubernetes NetworkPolicy: Namespace FE
apply to front-end pods in FE namespace
allow ingress on port 443
allow egress to backend pods on port 443
Calico GlobalNetworkPolicy
apply to all pods
allow egress DNS to kube-dns
deny all other traffic
Security Team
BE Dev Team
DB Dev Team
FE Dev Team

Ability to protect hosts and VMs

As Calico policies can be enforce on host interfaces, you can use them to protect your Kubernetes nodes (not just your pods), including for example, limiting access to node ports from outside of the cluster. To learn more, check out the Calico **policy for hosts** guides.

Integrates with Istio

When used with Istio service mesh, Calico policy engine enforces the same policy model at the host networking layer and at the service mesh layer, protecting your infrastructure from compromised workloads and protecting your workloads from compromised infrastructure. This also avoids the need for dual provisioning of security at the service mesh and infrastructure layers, or having to learn different policy models for each layer.

Extendable with Calico Enterprise

Calico Enterprise adds even richer network policy capabilities, with the ability to specify hierarchical policies, with each team have particular boundaries of trust, and FQDN / domain names in policy rules for restricting access to specific external services.

Best practices for network policies

Ingress and egress

At a minimum we recommend that every pod is protected by network policy ingress rules that restrict what is allowed to connect to the pod and on which ports. The best practice is also to define network policy egress rules that restrict the outgoing connections that are allowed by pods themselves. Ingress rules protect your pod from attacks outside of the pod. Egress rules help protect everything outside of the pod if the pod gets compromised, reducing the attack surface to make moving laterally (east-west) or to prevent an attacker from exfiltrating compromised data from your cluster (north-south).

Policy schemas

Due to the flexibility of network policy and labelling, there are often multiple different ways of labelling and writing policies that can achieve the same particular goal. One of the most common approaches is to have a small number of global policies that apply to all pods, and then a single pod specific policy that defines all the ingress and egress rules that are particular to that pod.

For example:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: front-end
  namespace: staging
spec:
  podSelector:
    matchLabels:
      app: back-end
  ingress:
    - from:
      - podSelector:
          matchLabels:
            app: front-end
    ports:
    - protocol: TCP
      port: 443
  egress:
    - to:
      - podSelector:
          matchLabels:
            app: database
    ports:
    - protocol: TCP
      port: 27017
```

Default deny

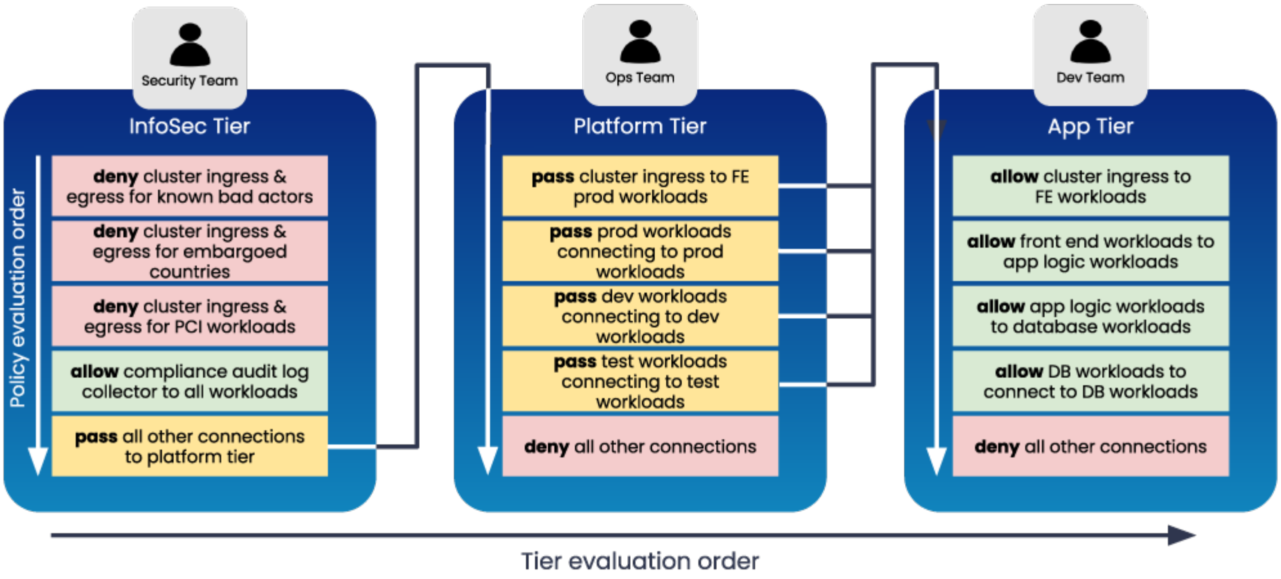
One approach to ensuring these best practices are being followed is to define default deny network policies. These ensure that if no other policy is defined that explicitly allows traffic to/from a pod, then the traffic will be denied. As a result, anytime a team deploys a new pod, they are forced to also define network policy for the pod. It can be useful to use a Calico GlobalNetworkPolicy for this (rather than needing to define a policy every time a new namespace is created) and to include some exceptions to the default deny (for example to allow pods to access DNS).

For example, you might use the following policy to default-deny all (non-system) pod traffic except for DNS queries to kube-dns/core-dns.

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: default-app-policy
spec:
  namespaceSelector: has(projectcalico.org/name) && projectcalico.org/name not in {“kube-
system”, “calico-system”}
  types:
  - Ingress
  - Egress
  egress:
    - action: Allow
      protocol: UDP
      destination:
        selector: k8s-app == “kube-dns”
        ports:
        - 53
```

Hierarchical policy

Calico Enterprise supports hierarchical network policy using policy tiers. RBAC for each tier can be defined to restrict who can interact with each tier. This can be used to delegate trust across multiple teams.


Security Team
InfoSec Tier
Policy evaluation order
deny cluster ingress & egress for known bad actors
deny cluster ingress & egress for embargoed countries
deny cluster ingress & egress for PCI workloads
allow compliance audit log collector to all workloads
pass all other connections to platform tier
Ops Team
Platform Tier
pass cluster ingress to FE prod workloads
pass prod workloads connecting to prod workloads
pass dev workloads connecting to dev workloads
pass test workloads connecting to test workloads
deny all other connections
Dev Team
App Tier
allow cluster ingress to FE workloads
allow front end workloads to app logic workloads
allow app logic workloads to database workloads
allow DB workloads to connect to DB workloads
deny all other connections
Tier evaluation order

Kubernetes Services

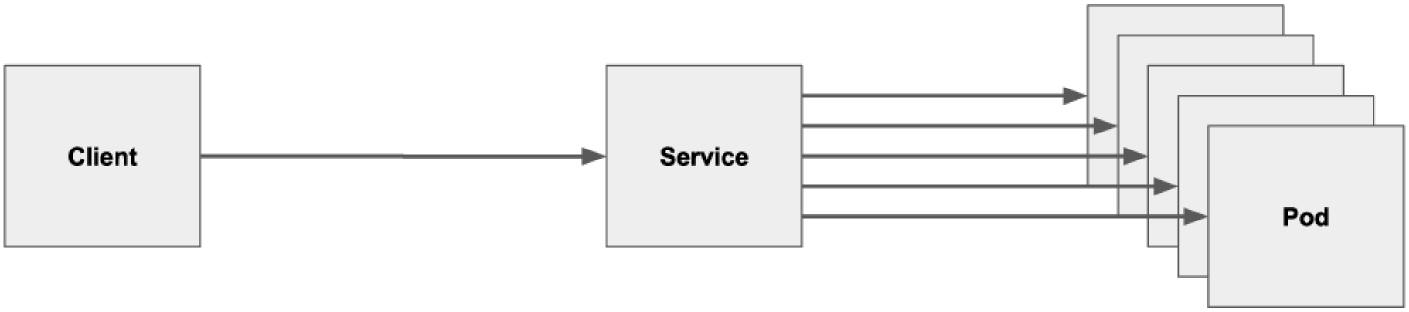
In this chapter you will learn:

- What are Kubernetes Services?
- What are the differences between the three main service types and what do you use them for?
- How do services and network policy interact?
- Some options for optimizing how services are handled.

What are Kubernetes Services?

Kubernetes **Services** provide a way of abstracting access to a group of pods as a network service. The group of pods backing each service is usually defined using a **label selector**.

When a client connects to a Kubernetes service, the connection is load balanced to one of the pods backing the service, as illustrated in this conceptual diagram:



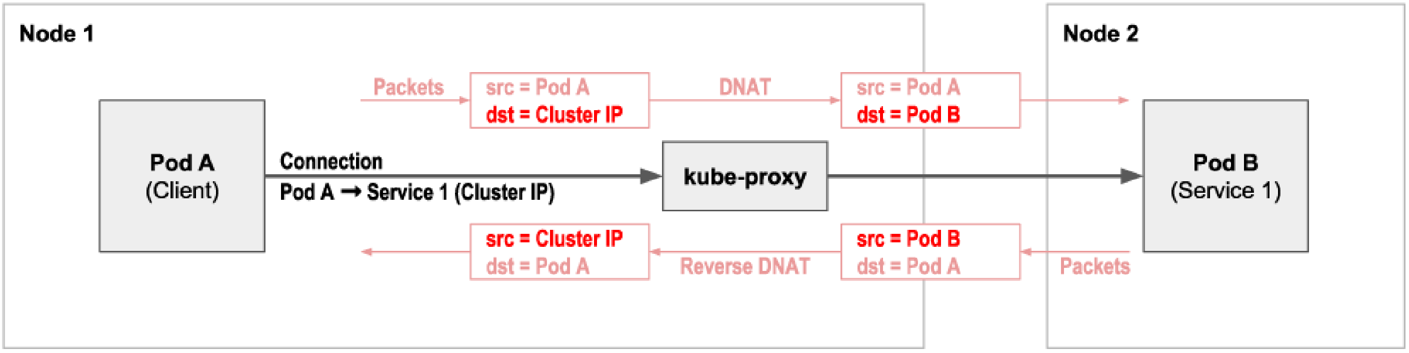
There are three main types of Kubernetes services:

- Cluster IP - which is the usual way of accessing a service from inside the cluster
- Node port - which is the most basic way of accessing a service from outside the cluster
- Load balancer - which uses an external load balancer as a more sophisticated way to access a service from outside the cluster.

Cluster IP services

The default service type is `ClusterIP` . This allows a service to be accessed within the cluster via a virtual IP address, known as the service Cluster IP. The Cluster IP for a service is discoverable through Kubernetes DNS. For example, `my-svc.my-namespace.svc.cluster-domain.example` . The DNS name and Cluster IP address remain constant for the life time of the service, even though the pods backing the service may be created or destroyed, and the number of pods backing the service may change over time.

In a typical Kubernetes deployment, kube-proxy runs on every node and is responsible for intercepting connections to Cluster IP addresses and load balancing across the group of pods backing each service. As part of this process **DNAT** is used to map the destination IP address from the Cluster IP to the chosen backing pod. Response packets on the connection then have the NAT reverse on their way back to the pod that initiated the connection.

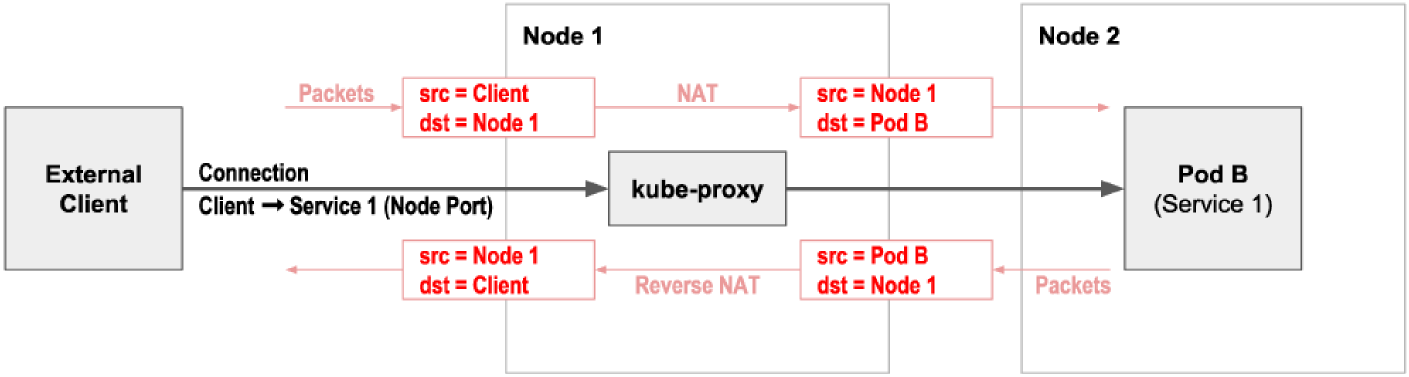


Importantly, network policy is enforced based on the pods, not the service Cluster IP. (i.e. Egress network policy is enforced for the client pod after the DNAT has changed the connection's destination IP to the chosen service backing pod. And because only the destination IP for the connection is changed, ingress network policy for the backing pod sees the original client pod as the source of the connection.)

Node port services

The most basic way to access a service from outside the cluster is to use a service of type `NodePort` . A Node Port is a port reserved on each node in the cluster through which the service can be accessed. In a typical Kubernetes deployment, kube-proxy is responsible for intercepting connections to Node Ports and load balancing them across the pods backing each service.

As part of this process NAT is used to map the destination IP address and port from the node IP and Node Port, to the chosen backing pod and service port. In addition the source IP address is mapped from the client IP to the node IP, so that response packets on the connection flow back via the original node, where the NAT can be reversed. (It's the node which performed the NAT that has the connection tracking state needed to reverse the NAT.)

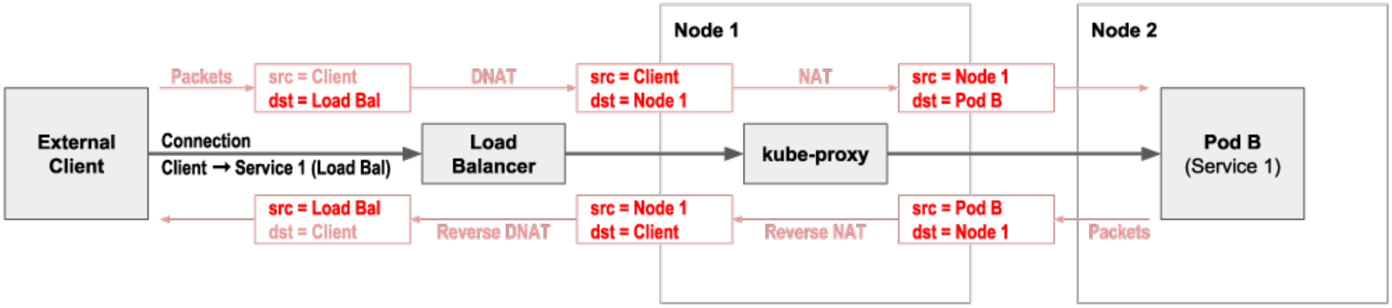


Note that because the connection source IP address is SNATed to the node IP address, ingress network policy for the service backing pod does not see the original client IP address. Typically this means that any such policy is limited to restricting the destination protocol and port, and cannot restrict based on the client / source IP. This limitation can be circumvented in some scenarios by using **externalTrafficPolicy** or by using Calico's eBPF dataplane **native service handling** (rather than kube-proxy) which preserves source IP address.

Load balancer services

Services of type `LoadBalancer` expose the service via an external network load balancer (NLB). The exact type of network load balancer depends on which public cloud provider or, if on-prem, which specific hardware load balancer integration is integrated with your cluster.

The service can be accessed from outside of the cluster via a specific IP address on the network load balancer, which by default will load balance evenly across the nodes using the service node port.

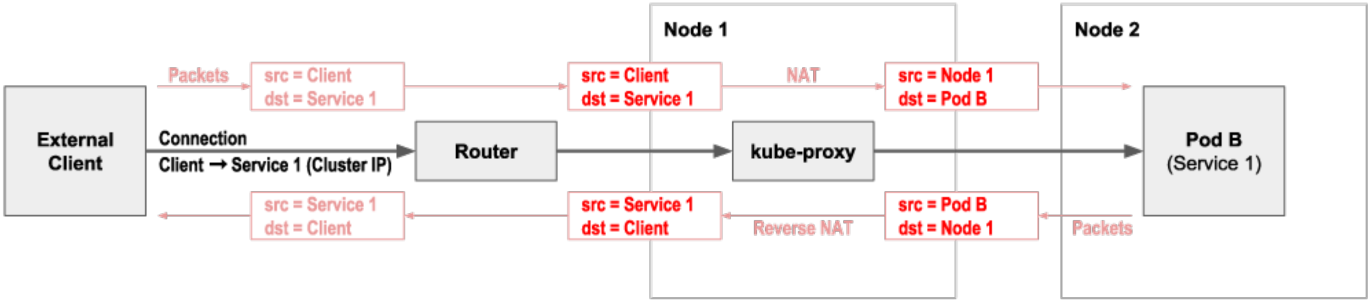


Most network load balancers preserve the client source IP address, but because the service then goes via a node port, the backing pods themselves do not see the client IP, with the same implications for network policy. As with node ports, this limitation can be circumvented in some scenarios by using **externalTrafficPolicy** or by using Calico's eBPF dataplane **native service handling** (rather than kube-proxy) which preserves source IP address.

Advertising service IPs

One alternative to using node ports or network load balancers is to advertise service IP addresses over BGP. This requires the cluster to be running on an underlying network that supports BGP, which typically means an on-prem deployment with standard Top of Rack routers.

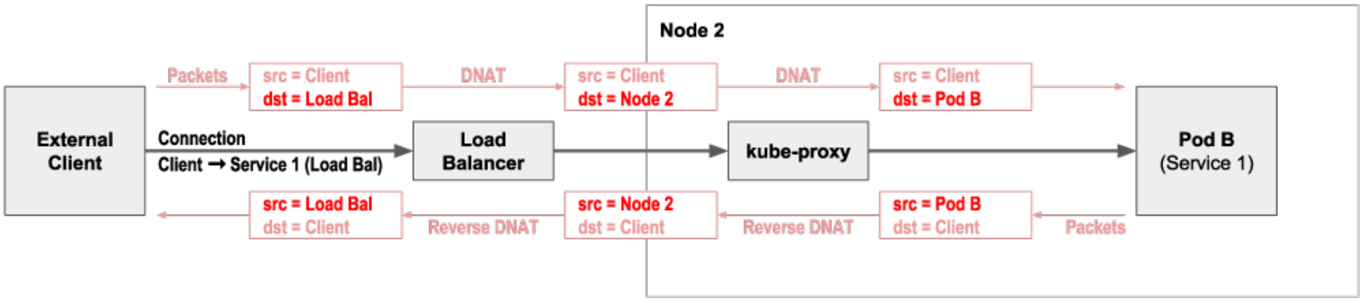
Calico supports advertising service Cluster IPs, or External IPs for services configured with one. If you are not using Calico as your network plugin then MetalLB provides similar capabilities that work with a variety of different network plugins.

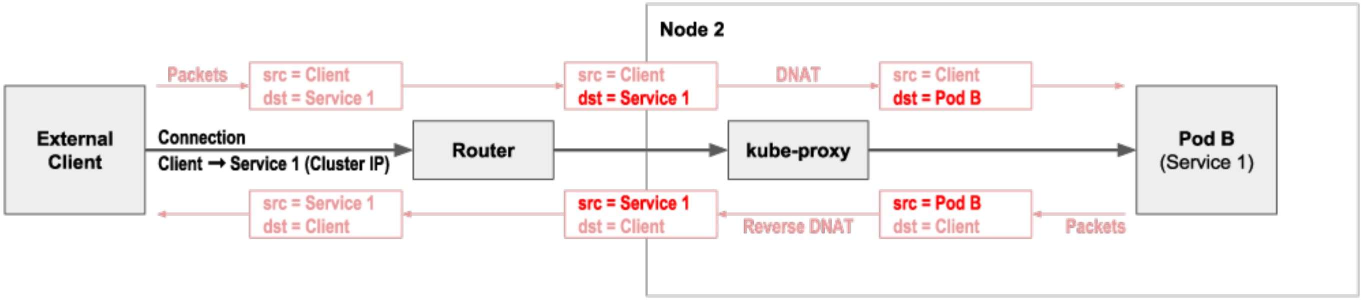


externalTrafficPolicy:local

By default, whether using service type `NodePort` or `LoadBalancer` or advertising service IP addresses over BGP, accessing a service from outside the cluster load balances evenly across all the pods backing the service, independent of which node the pods are on. This behavior can be changed by configuring the service with `externalTrafficPolicy:local` which specifies that connections should only be load balanced to pods backing the service on the local node.

When combined with services of type `LoadBalancer` or with Calico service IP address advertising, traffic is only directed to nodes that host at least one pod backing the service. This reduces the potential extra network hop between nodes, and perhaps more importantly, to maintain the source IP address all the way to the pod, so network policy can restrict specific external clients if desired.

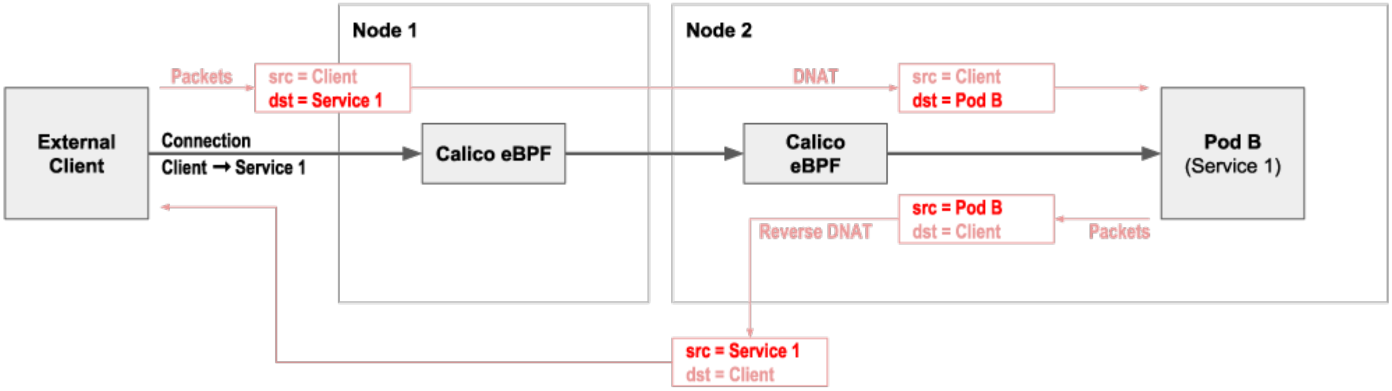




Note that in the case of services of type `LoadBalancer` , not all Load Balancers support this mode. And in the case of service IP advertisement, the evenness of the load balancing becomes topology dependent. In this case, pod anti-affinity rules can be used to ensure even distribution of backing pods across your topology, but this does add some complexity to deploying the service.

Calico eBPF native service handling

As an alternative to using Kubernetes standard kube-proxy, Calico's **eBPF dataplane** supports native service handling. This preserves source IP to simplify network policy, offers DSR (Direct Server Return) to reduce the number of network hops for return traffic, and provides even load balancing independent of topology, with reduced CPU and latency compared to kube-proxy.



Above and beyond

- **Video: Everything you need to know about Kubernetes Services networking**
- **Blog: Introducing the Calico eBPF dataplane**
- **Blog: Hands on with Calico eBPF native service handling**

Kubernetes Ingress

In this chapter you will learn:

- What is Kubernetes Ingress?
- Why use ingress?
- What are the differences between different ingress implementations?
- How does ingress and network policy interact?
- How does ingress and services fit together under the covers?

What is Kubernetes Ingress?

Kubernetes Ingress builds on top of Kubernetes **Services** to provide load balancing at the application layer, mapping HTTP and HTTPS requests with particular domains or URLs to Kubernetes services. Ingress can also be used to terminate SSL / TLS before load balancing to the service.

The details of how Ingress is implemented depend on which **Ingress Controller** you are using. The Ingress Controller is responsible for monitoring Kubernetes **Ingress** resources and provisioning / configuring one or more ingress load balancers to implement the desired load balancing behavior.

Unlike Kubernetes services, which are handled at the network layer (L3-4), ingress load balancers operate at the application layer (L5-7). Incoming connections are terminated at the load balancer so it can inspect the individual HTTP / HTTPS requests. The requests are then forwarded via separate connections from the load balancer to the chosen service backing pods. As a result, network policy applied to the backing pods can restrict access to only allow connections from the load balancer, but cannot restrict access to specific original clients.

Why use Kubernetes Ingress?

Given that Kubernetes **Services** already provide a mechanism for load balancing access to services from outside of the cluster, why might you want to use Kubernetes Ingress?

The mainline use case is if you have multiple HTTP / HTTPS services that you want to expose through a single external IP address, perhaps with each service having a different URL path, or perhaps as multiple different domains. This is lot simpler from a client configuration point of view than exposing each service outside of the cluster using Kubernetes Services, which would give each service a separate external IP address.

If on the other hand, your application architecture is fronted by a single “front end” microservice then Kubernetes Services likely already meet your needs. In this case you might prefer to not add Ingress to the picture, both from a simplicity point of view, and potentially also so you can more easily restrict access to specific clients using network policy. In effect, your “front end” microservice already plays the role of Kubernetes Ingress, in a way that is not that dissimilar to **in-cluster ingress** solutions discussed below.

Types of Ingress solutions

Broadly speaking there are two types of ingress solutions:

- In-cluster ingress - where ingress load balancing is performed by pods within the cluster itself.
- External ingress - where ingress load balancing is implemented outside of the cluster by appliances or cloud provider capabilities.

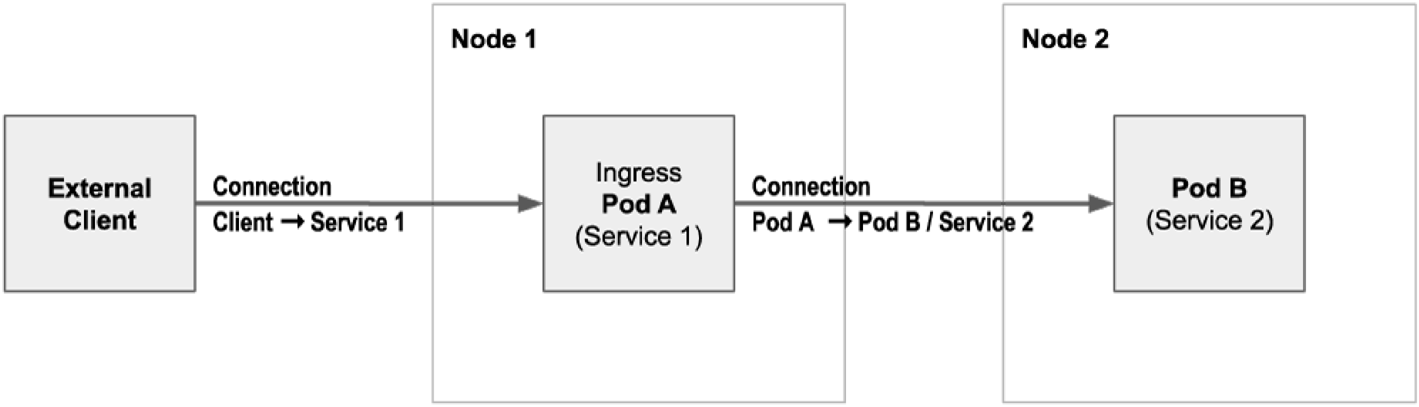
In-cluster ingress solutions

In-cluster ingress solutions use software load balancers running in pods within the cluster itself. There are many different ingress controllers to consider that follow this pattern, including for example the NGINX ingress controller.

The advantages of this approach are that you can:

- horizontally scale your ingress solution up to the limits of Kubernetes
- choose the ingress controller that best suits your specific needs, for example, with particular load balancing algorithms, or security options.

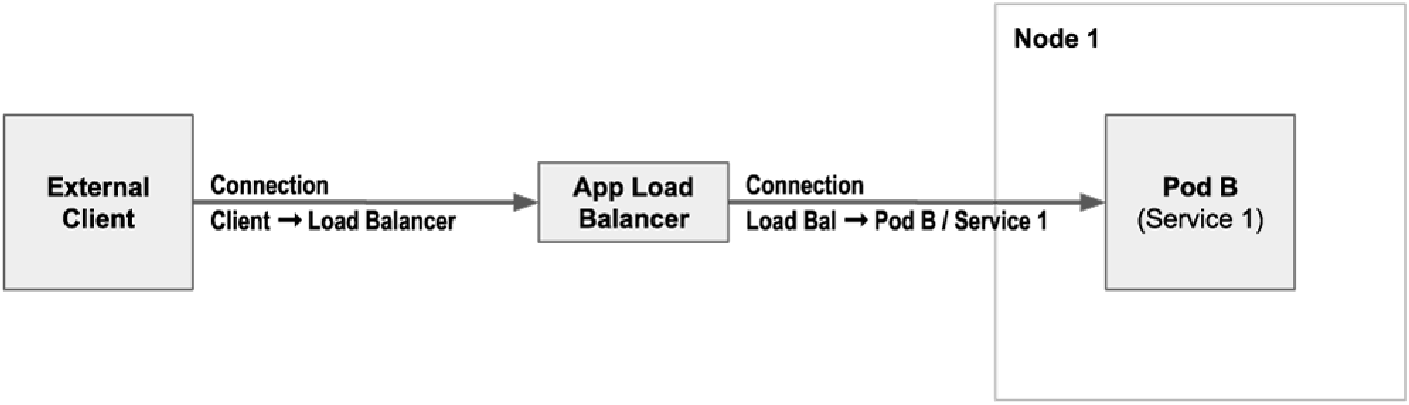
To get your ingress traffic to the in-cluster ingress pods, the ingress pods are normally exposed externally as a Kubernetes service, so you can use any of the standard ways of accessing the service from outside of the cluster. A common approach is use an external network load balancer or service IP advertisement, with `externalTrafficPolicy:local` . This minimizes the number of network hops, and retains the client source IP address, which allows network policy to be used to restrict access to the ingress pods to particular clients if desired.



External ingress solutions

External ingress solutions use application load balancers outside of the cluster. The exact details and features depend on which ingress controller you are using, but most cloud providers include an ingress controller that automates the provisioning and management of the cloud provider's application load balancers to provide ingress.

The advantages of this type of ingress solution is that your cloud provider handles the operational complexity of the ingress for you. The downsides are a potentially more limited set of features compared to the rich range of in-cluster ingress solutions, and the maximum number of services exposed by ingress being constrained by cloud provider specific limits.



Note that most application load balancers support a basic mode of operation of forwarding traffic to the chosen service backing pods via the **node port** of the corresponding service.

In addition to this basic approach of load balancing to service node ports, some cloud providers support a second mode of application layer load balancing, which load balances directly to the pods backing each service, without going via node-ports or other kube-proxy service handling. This has the advantage of eliminating the potential second network hop associated with node ports load balancing to a pod on a different node. The potential disadvantage is that if you are operating at very high scales, for example with hundreds of pods backing a service, you may exceed the application layer load balancers maximum limit of IPs it can load balance to in this mode. In this case switching to an in-cluster ingress solution is likely the better fit for you.

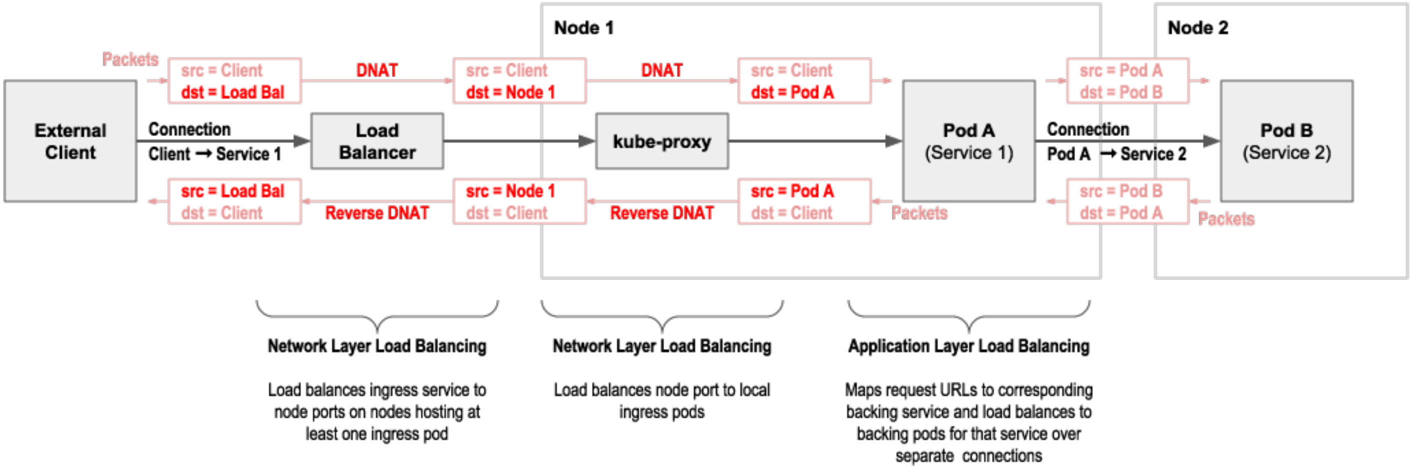
Show me everything!

All the above diagrams focus on connection level (L5-7) representation of ingress and services. You can learn more about the network level (L3-4) interactions involved in handling the connections, including which scenarios client source IP addresses are maintained, in the **About Kubernetes Services** guide.

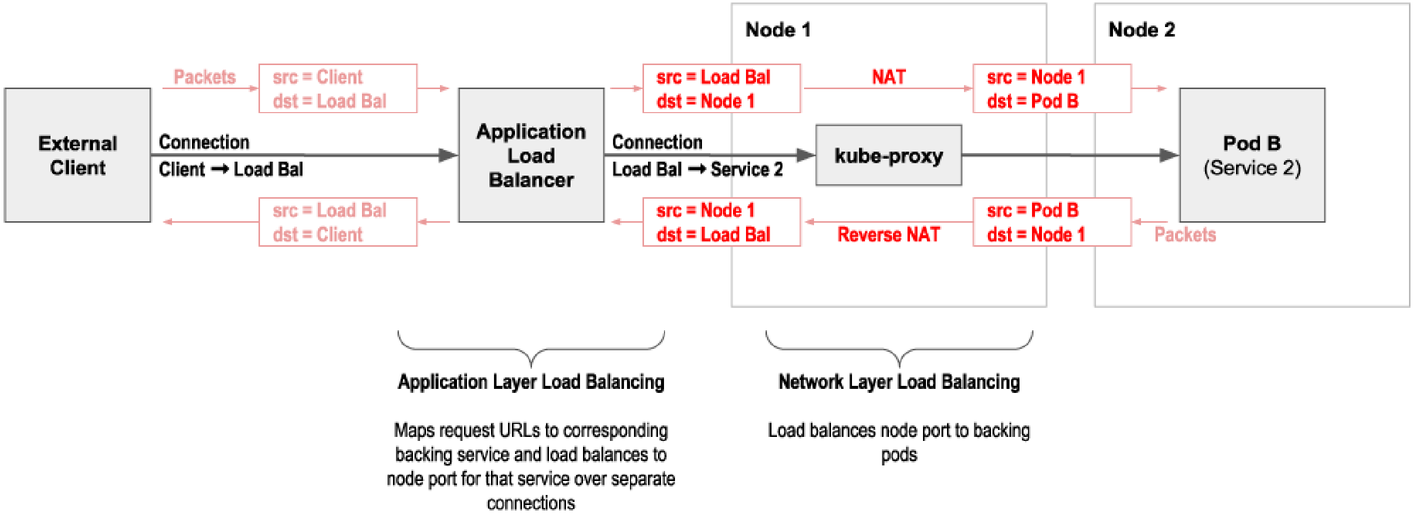
If you are already up to speed on how services work under the covers, here are some more complete diagrams that show details of how services are load balanced at the network layer (L3-4).

Note:
You can successfully use ingress without needing to understand this next level of detail! So feel free to skip over these diagrams if you don't want to dig deeper into how services and ingress interact under the covers.

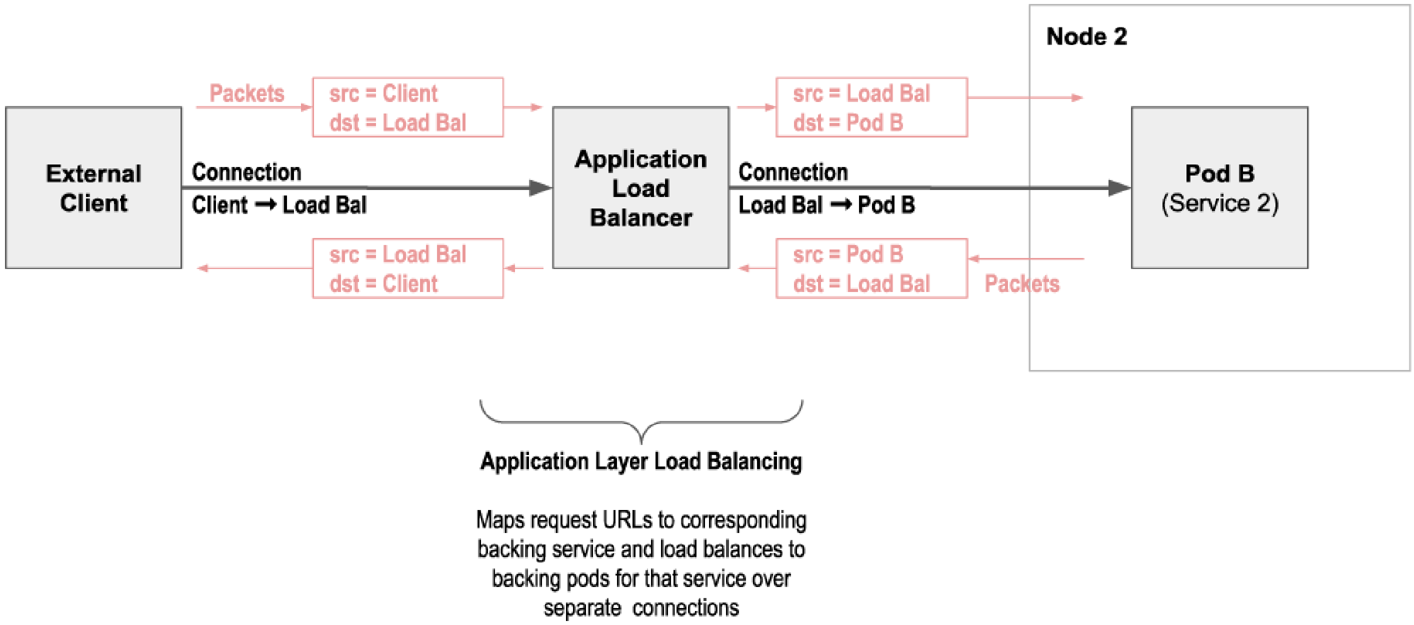
In-cluster ingress solution exposed as service type `LoadBalancer` with `externalTrafficPolicy:local`



External ingress solution via node ports



External ingress solution direct to pods



Above and beyond

- Video: Everything you need to know about Kubernetes Ingress networking
- Video: Everything you need to know about Kubernetes Services networking

Kubernetes Egress

In this chapter you will learn:

- What is Kubernetes Egress?
- Why should you restrict egress traffic and how can you do it?
- What is “NAT outgoing” and when is it used?
- What is an egress gateway, and why might you want to use one?

What is Kubernetes Egress?

In this guide we are using the term Kubernetes egress to describe connections being made from pods to anything outside of the cluster.

In contrast to ingress traffic, where Kubernetes has the **Ingress** resource type to help manage the traffic, there is no Kubernetes Egress resource. Instead, how the egress traffic is handled at a networking level is determined by the Kubernetes network implementation / CNI plugin being used by the cluster. In addition, if a service mesh is being used, this can add egress behaviors on top of those the network implementation provides.

There are three areas of behavior worth understanding for egress traffic, so you can choose a networking and/or service mesh setup that best suits your needs:

- Restricting egress traffic
- Outgoing NAT behavior
- Egress gateways

Restricting egress traffic

It’s a common security requirement and best practice to restrict outgoing connections from the cluster. This is normally achieved using **Network Policy** to define egress rules for each microservice, often in conjunction with a **default deny** policy that ensures outgoing connections are denied by default, until a policy is defined to explicitly allow specific traffic.

One limitation when using Kubernetes Network Policy to restrict access to specific external resources, is that the external resources need to be specified as IP addresses (or IP address ranges) within the policy rules. If the IP addresses associated with an external resource change, then every policy that referenced those IP addresses needs to be updated with the new IP addresses. This limitation can be circumvented using Calico **Network Sets**, or Calico Enterprise’s support for **domain names** in policy rules.

In addition to using network policy, service meshes typically allow you to configure which external services each pod can access. In the case of Istio, Calico can be integrated to enforce network policy at the service mesh layer, including **L5-7 rules**, as another alternative to using IP addresses in rules. To learn more about the benefits of this kind of approach, read our **Adopt a zero trust network model for security** guide.

Note in addition to everything mentioned so far, perimeter firewalls can also be used to restrict outgoing connections, for example to allow connections only to particular external IP address ranges, or external services. However, since perimeter firewalls typically cannot distinguish individual pods, the rules apply equally to all pods in the cluster. This provides some defense in depth, but cannot replace the requirement for network policy.

NAT outgoing

Network Address Translation (**NAT**) is the process of mapping an IP address in a packet to a different IP address as the packet passes through the device performing the NAT. Depending on the use case, NAT can apply to the source or destination IP address, or to both addresses.

In the context of Kubernetes egress, NAT is used to allow pods to connect to services outside of the cluster if the pods have IP addresses that are not routable outside of the cluster (for example, if the pod network is an overlay).

For example, if a pod in an overlay network attempts to connect to an IP address outside of the cluster, then the node hosting the pod uses SNAT (Source Network Address Translation) to map the non-routable source IP address of the packet to the node's IP address before forwarding on the packet. The node then maps response packets coming in the opposite direction back to the original pod IP address, so packets flow end-to-end in both directions, with neither pod or external service being aware the mapping is happening.

In most clusters this NAT behavior is configured statically across the whole of the cluster. When using Calico, the NAT behavior can be configured at a more granular level for particular address ranges using **IP pools**. This effectively allows the scope of "non-routable" to be more tightly defined than just "inside the cluster vs outside the cluster", which can be useful in some enterprise deployment scenarios.

Egress gateways

Another approach to Kubernetes egress is to route all outbound connections via one or more egress gateways. The gateways SNAT (Source Network Address Translation) the connections so the external service being connected to sees the connection as coming from the egress gateway. The main use case is to improve security, either with the egress gateway performing a direct security role in terms of what connections it allows, or in conjunction with perimeter firewalls (or other external entities). For example, so that perimeter firewalls see the connections coming from well known IP addresses (the egress gateways) rather than from dynamic pod IP addresses they don't understand.

Egress gateways are not a native concept in Kubernetes itself, but are implemented by some Kubernetes network implementations and some service meshes. For example, Calico Enterprise provides egress gateway functionality, plus the ability to map namespaces (or even individual pods) to specific egress gateways. Perimeter firewalls (or other external security entities) can then effectively provide per namespace security controls, even though they do not have visibility to dynamic pod IP addresses.

As an alternative approach to egress gateways, Calico allows you to control pod IP address ranges based on namespace, or node, or even at the individual pod level. Assuming no outgoing NAT is required, this provides a very simple way for perimeter firewalls (or other external security entities) to integrate with Kubernetes for both ingress and egress traffic. (Note that this approach relies on having enough address space available to sensibly assign IP address ranges, for example to each namespace, so it can lead to IP address range exhaustion challenges for large scale deployments. In these scenarios, using egress gateways is likely to be a better option.)

Above and beyond

- Adopt a zero trust network model for security
- Use external IPs or networks rules in policy
- Enforce network policy using Istio
- Use HTTP methods and paths in policy rules
- Restrict a pod to use an IP address in a specific range
- Assign IP addresses based on topology
- ✓ Advanced egress access controls

eBPF

eBPF is a Linux kernel feature that allows fast yet safe mini-programs to be loaded into the kernel in order to customise its operation.

In this chapter you will learn:

- General background on eBPF.
- Various uses of eBPF.
- How Calico uses eBPF in the eBPF dataplane.

What is eBPF?

eBPF is a virtual machine embedded within the Linux kernel. It allows small programs to be loaded into the kernel, and attached to hooks, which are triggered when some event occurs. This allows the behaviour of the kernel to be (sometimes heavily) customised. While the eBPF virtual machine is the same for each type of hook, the capabilities of the hooks vary considerably. Since loading programs into the kernel could be dangerous; the kernel runs all programs through a very strict static verifier; the verifier sandboxes the program, ensuring it can only access allowed parts of memory and ensuring that it must terminate quickly.

Why is it called eBPF?

eBPF stands for "extended Berkeley Packet Filter". The Berkeley Packet Filter was an earlier, more specialised virtual machine that was tailored for filtering packets. Tools such as `tcpdump` use this "classic" BPF VM to select packets that should be sent to userspace for analysis. eBPF is a considerably extended version of BPF that is suitable for general purpose use inside the kernel. While the name has stuck, eBPF can be used for a lot more than just packet filtering.

What can eBPF do?

Types of eBPF program

There are several classes of hooks to which eBPF programs can be attached within the kernel. The capabilities of an eBPF program depend hugely on the hook to which it is attached:

- **Tracing** programs can be attached to a significant proportion of the functions in the kernel. Tracing programs are useful for collecting statistics and deep-dive debugging of the kernel. Most tracing hooks only allow read-only access to the data that the function is processing but there are some that allow data to be modified. The Calico team use tracing programs to help debug Calico during development; for example, to figure out why the kernel unexpectedly dropped a packet.

- **Traffic Control** (`tc`) programs can be attached at ingress and egress to a given network device. The kernel executes the programs once for each packet. Since the hooks are for packet processing, the kernel allows the programs to modify or extend the packet, drop the packet, mark it for queueing, or redirect the packet to another interface. Calico's eBPF dataplane is based on this type of hook; we use tc programs to load balance Kubernetes services, to implement network policy, and, to create a fast-path for traffic of established connections.

- **XDP**, or "eXpress Data Path", is actually the name of an eBPF hook. Each network device has an XDP ingress hook that is triggered once for each incoming packet before the kernel allocates a socket buffer for the packet. XDP can give outstanding performance for use cases such as DoS protection (as supported in Calico's standard Linux dataplane) and ingress load balancing (as used in facebook's Katran). The downside of XDP is that it requires network device driver support to get good performance, and it doesn't inter-work with pod networking very well.

- Several types of **socket** programs hook into various operations on sockets, allowing the eBPF program to, for example, change the destination IP of a newly-created socket, or force a socket to bind to the "correct" source IP address. Calico uses such programs to do connect-time load balancing of Kubernetes Services; this reduces overhead because there is no DNAT on the packet processing path.

- There are various security-related hooks that allow for program behaviour to be policed in various ways. For example, the seccomp hooks allow for syscalls to be policed in fine-grained ways.

- And… probably a few more hooks by the time you ready this; eBPF is under heavy development in the kernel.

The kernel exposes the capabilities of each hook via "helper functions". For example, the `tc` hook has a helper function to resize the packet, but that helper would not be available in a tracing hook. One of the challenges of working with eBPF is that different kernel versions support different helpers and lack of a helper can make it impossible to implement a particular feature.

BPF maps

Programs attached to eBPF hooks are able to access BPF "maps". BPF maps have two main uses:

- They allow BPF programs to store and retrieve long-lived data.
- They allow communication between BPF programs and user-space programs. BPF programs can read data that was written by userspace and vice versa.

There are many types of BPF maps, including some special types that allow jumping between programs, and, some that act as queues and stacks rather than strictly as key/value maps. Calico uses maps to keep track of active connections, and, to configure the BPF programs with policy and service NAT information. Since map accesses can be relatively expensive, Calico aims to do a single map lookup only for each packet on an established flow.

The contents of bpf maps can be inspected using the command-line tool, `bpftool` , which is provided with the kernel.

Calico's eBPF dataplane

Calico's eBPF dataplane is an alternative to our standard Linux dataplane (which is iptables based). While the standard dataplane focuses on compatibility by inter-working with kube-proxy, and your own iptables rules, the eBPF dataplane focuses on performance, latency and improving user experience with features that aren't possible in the standard dataplane. As part of that, the eBPF dataplane replaces kube-proxy with an eBPF implementation. The main "user experience" feature is to preserve the source IP of traffic from outside the cluster when traffic hits a NodePort; this makes your server-side logs and network policy much more useful on that path.

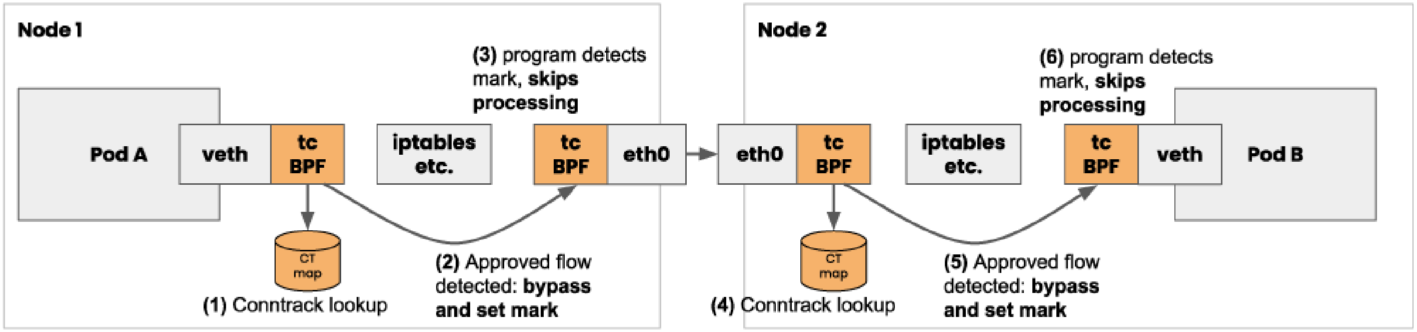
Feature comparison

While the eBPF dataplane has some features that the standard Linux dataplane lacks, the reverse is also true:

Factor	Standard Linux Dataplane	eBPF dataplane
Throughput	Designed for 10GBit+	Designed for 40GBit+
First packet latency	Low (kube-proxy service latency is bigger factor)	Lower
Subsequent packet latency	Low	Lower
Preserves source IP within cluster	Yes	Yes
Preserves external source IP	Only with `externalTrafficPolicy: Local`	Yes
Direct Server Return	Not supported	Supported (requires compatible underlying network)
Connection tracking	Linux kernel's conntrack table (size can be adjusted)	BPF map (fixed size)
Policy rules	Mapped to iptables rules	Mapped to BPF instructions
Policy selectors	Mapped to IP sets	Mapped to BPF maps
Kubernetes services	kube-proxy iptables or IPVS mode	BPF program and maps
IPIP	Supported	Supported (no performance advantage due to kernel limitations)
VXLAN	Supported	Supported
Wireguard	Supported	Supported
Other routing	Supported	Supported
Supports third party CNI plugins	Yes (compatible plugins only)	Yes (compatible plugins only)
Compatible with other iptables rules	Yes (can write rules above or below other rules)	Partial; iptables bypassed for workload traffic
XDP DoS Protection	Supported	Not supported (yet)
IPv6	Supported	Not supported (yet)
Host endpoint policy	Supported	Not supported (yet)
Enterprise version	Available	Not supported (yet)

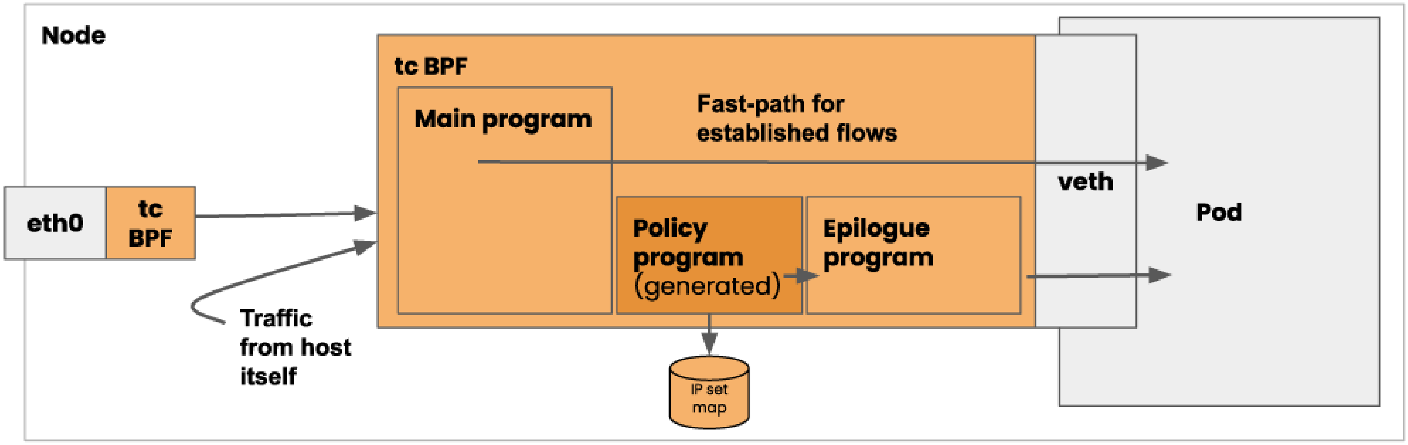
Architecture overview

Calico's eBPF dataplane attaches eBPF programs to the `tc` hooks on each Calico interface as well as your data and tunnel interfaces. This allows Calico to spot workload packets early and handle them through a fast-path that bypasses iptables and other packet processing that the kernel would normally do.

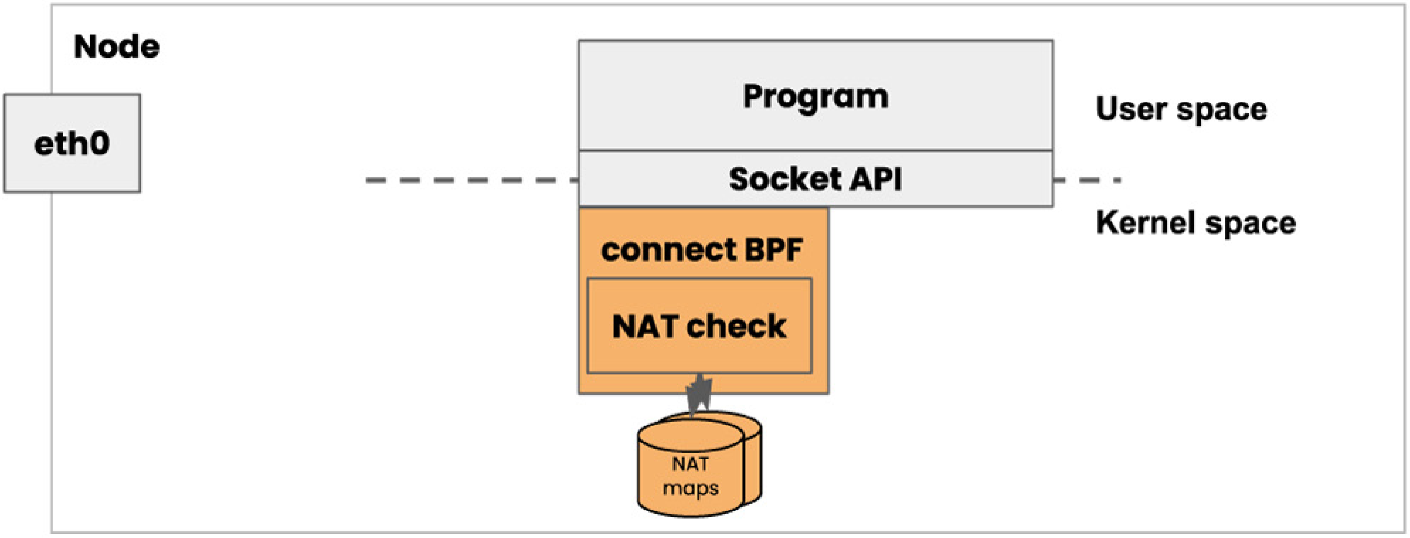

Node 1
Pod A
veth
tc BPF
iptables etc.
tc BPF
eth0
CT map
(1) Conntrack lookup
(2) Approved flow detected: bypass and set mark
(3) program detects mark, skips processing
Node 2
eth0
tc BPF
iptables etc.
tc BPF
veth
Pod B
CT map
(4) Conntrack lookup
(5) Approved flow detected: bypass and set mark
(6) program detects mark, skips processing

The logic to implement load balancing and packet parsing is pre-compiled ahead of time and relies on a set of BPF maps to store the NAT frontend and backend information. One map stores the metadata of the service, allowing for `externalTrafficPolicy` and "sticky" services to be honoured. A second map stores the IPs of the backing pods.

In eBPF mode, Calico converts your policy into optimised eBPF bytecode, using BPF maps to store the IP sets matched by policy selectors.



To improve performance for services, Calico also does connect-time load balancing by hooking into the socket BPF hooks. When a program tries to connect to a Kubernetes service, Calico intercepts the connection attempt and configures the socket to connect directly to the backend pod's IP instead. This removes all NAT overhead from service connections.



Above and beyond

- For more information and performance metrics for the eBPF dataplane, see the **announcement blog post**.
- If you'd like to try eBPF mode in your Kubernetes cluster, follow the **Enable the eBPF dataplane** guide.

Calico

What is Calico?

Calico is an open source networking and network security solution for containers, virtual machines, and native host-based workloads. Calico supports a broad range of platforms including Kubernetes, OpenShift, Docker EE, OpenStack, and bare metal services.

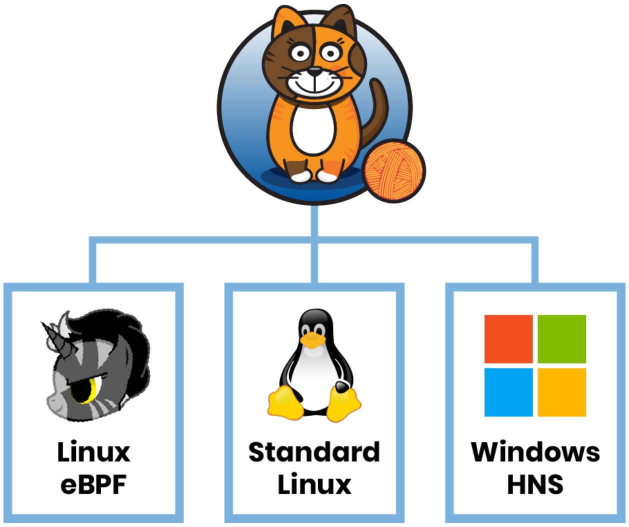
Whether you opt to use Calico's eBPF data plane or Linux's standard networking pipeline, Calico delivers blazing fast performance with true cloud-native scalability. Calico provides developers and cluster operators with a consistent experience and set of capabilities whether running in public cloud or on-prem, on a single node, or across a multi-thousand node cluster.

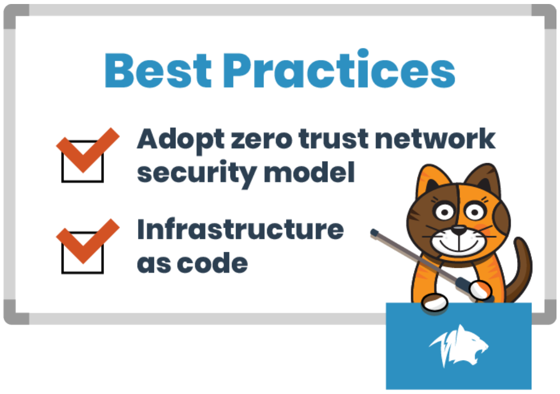
Why use Calico?

Choice of dataplanes

Calico gives you a choice of dataplanes, including a pure Linux eBPF dataplane, a standard Linux networking dataplane, and a Windows HNS dataplane. Whether you prefer cutting edge features of eBPF, or the familiarity of the standard primitives that existing system administrators already know, Calico has you covered.

Whichever choice is right for you, you'll get the same, easy to use, base networking, network policy and IP address management capabilities, that have made Calico the most trusted networking and network policy solution for mission-critical cloud-native applications.





Best practices for network security

Calico's rich network policy model makes it easy to lock down communication so the only traffic that flows is the traffic you want to flow. Plus with built in support for Wireguard encryption, securing your pod-to-pod traffic across the network has never been easier.

Calico's policy engine can enforce the same policy model at the host networking layer and (if using Istio & Envoy) at the service mesh layer, protecting your infrastructure from compromised workloads and protecting your workloads from compromised infrastructure.

Performance

Depending on your preference, Calico uses either Linux eBPF or the Linux kernel's highly optimized standard networking pipeline to deliver high performance networking. Calico's networking options are flexible enough to run without using overlays in most environments, avoiding the overheads of packet encap/decap. Calico's control plane and policy engine has been fine tuned over many years of production use to minimize overall CPU usage and occupancy.

Scalability

Calico's core design principles leverage best practice cloud-native design patterns combined with proven standards based network protocols trusted worldwide by the largest internet carriers. The result is a solution with exceptional scalability that has been running at scale in production for years. Calico's development test cycle includes regularly testing multi-thousand node clusters. Whether you are running a 10 node cluster, 100 node cluster, or more, you reap the benefits of the improved performance and scalability characteristics demanded by the largest Kubernetes clusters.

Interoperability

Calico enables Kubernetes workloads and non-Kubernetes or legacy workloads to communicate seamlessly and securely. Kubernetes pods are first class citizens on your network and able to communicate with any other workload on your network. In addition Calico can seamlessly extend to secure your existing host based workloads (whether in public cloud or on-prem on VMs or bare metal servers) alongside Kubernetes. All workloads are subject to the same network policy model so the only traffic that is allowed to flow is the traffic you expect to flow.

SQUARESPACE
virtustream.
Giant Swarm
GitHub
ocado
USA TODAY
apigee
orange
box
YAHOO! JAPAN

Real world production hardened

Calico is trusted and running in production at large enterprises including SaaS providers, financial services companies, and manufacturers. The largest public cloud providers have selected Calico to provide network security for their hosted Kubernetes services (Amazon EKS, Azure AKS, Google GKE, and IBM IKS) running across tens of thousands of clusters.

Full Kubernetes network policy support

Calico's network policy engine formed the original reference implementation of Kubernetes network policy during the development of the API. Calico is distinguished in that it implements the full set of features defined by the API giving users all the capabilities and flexibility envisaged when the API was defined. And for users that require even more power, Calico supports an extended set of network policy capabilities that work seamlessly alongside the Kubernetes API giving users even more flexibility in how they define their network policies.

Contributor community

The Calico open source project is what it is today thanks to 200+ contributors across a broad range of companies. In addition Calico is backed by Tigera, founded by the original Calico engineering team, and committed to maintaining Calico as the leading standard for Kubernetes network security.

Calico Enterprise compatible

Calico Enterprise builds on top of open source Calico to provide additional higher-level features and capabilities:

- Hierarchical network policy
- Egress access controls (DNS policies, egress gateways)
- Network visualization and troubleshooting
- Network policy recommendations
- Network policy preview and staging
- Compliance controls and reporting
- Intrusion detection (suspicious activity, anomaly detection)
- Multi-cluster management with multi-cloud federation

CALICO ENTERPRISE

Learn More

Calico Enterprise

In this chapter you will learn:

- What is Calico Enterprise?
- What are the features that help you at these important stages?
    - Pre-production
    - Production, initial application rollout
    - Production, mass migration

What is Calico Enterprise?

Calico Enterprise is the commercial product and extension of **Calico open source**. It provides the same secure application connectivity across multi-cloud and legacy environments as Calico, but adds enterprise control and compliance capabilities for mission-critical deployments.

In each stage on your path to production with Kubernetes, you'll likely encounter roadblocks that you've already solved with traditional firewalls. The challenge is getting the same level of security and visibility in the highly-dynamic environment of Kubernetes.

In the following sections, we'll look at common obstacles in each stage, and the Calico Enterprise features that will keep you moving forward.

Pilot, pre-production

In this stage, your Kubernetes journey will likely feel fragile, and on the verge of derailment. Each team knows what they want, but it is hard to remap tools and processes to Kubernetes workflows. If any of the following issues sound familiar, know that they are common in the pre-production stage. They reflect what customers are telling us at Tigers about the disruptive nature of Kubernetes.

- "Each new service requires a firewall rule change. The change control process is derailing my project."
- "I need multi-tenancy in my clusters for tenant isolation requirements. Kubernetes adds an additional layer of complexity that breaks traditional methods of tenant isolation."
- "Our applications that interact with the public internet require a firewall, and the security team needs to manage those rules and have visibility into their enforcement."
- "Applications have different security and compliance requirements. When managed centrally, our projects are severely delayed. We need a way to decentralize security policy creation and enforcement to speed things up."

- "Our legacy security policy management for cloud-based environments does not work very well and is very slow. I need a modern approach to enforcing security and extending their Kubernetes security policies to host and manage cloud-based environments."

The following are the **top three issues** that block many Kubernetes initiatives in the pre-production phase. These are all solvable using Calico Enterprise.

Egress access controls

Establishing service-to-service connectivity within your cluster is easy. But how do you enable some of your workloads to securely connect to services like Amazon RDS, ElasticCache, etc. that are external to the cluster? And what about APIs, and endpoints outside the cluster?

When traffic from specific applications leaves the cluster to access an external destination, it can be useful to control the source IP of that traffic. For example, you may want an additional firewall around the cluster to police external accesses from the cluster, and grant access only from authorized workloads within the cluster.

Calico Enterprise provides two solutions:

DNS policies for per-node firewalls
With Calico Enterprise DNS policy, you can enforce your egress security policies within your cluster by applying fine-grained access controls between a workload and the external services. DNS policy supports using fully-qualified domain names and DNS endpoints in your policy rules. Unless otherwise allowed by the policy, no other pods can communicate with the DNS endpoint(s).

Egress gateway for in-depth defense
If you need to enforce security policies at an external control point (e.g. a firewall), you can implement a Calico Enterprise egress gateway. You configure the external firewall to allow outbound connections only from particular source IPs, and configure the intended cluster workloads so their outbound traffic has one of the source IPs.

You can also use egress gateways to direct all outbound traffic from a particular application to leave the cluster through a specific node or nodes. In this case, you schedule the gateways to the desired nodes, and configure the application pods/namespaces to use those gateways.

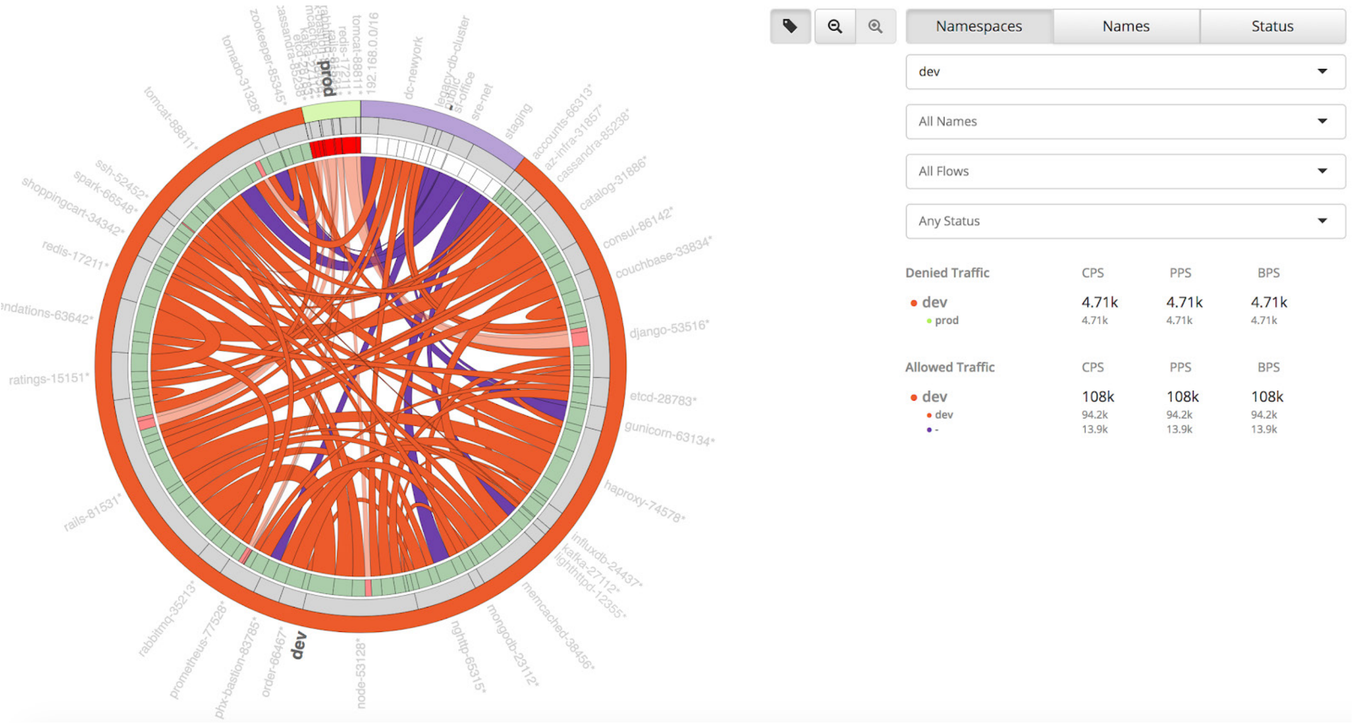
Visibility, traceability, troubleshooting

Visibility and troubleshooting in Kubernetes is often the biggest challenge to moving forward at this stage. Applications running on Kubernetes platforms are constantly changing IP addresses and locations, so traditional log flows are not helpful when debugging issues. Further, Kubernetes network connectivity does not natively log traffic or provide visibility into namespaces, pods, labels, policy traffic, and accepted/denied connections. So when connectivity breaks, it's hard to identify which policy denied the traffic. And manually traversing syslogs on individual nodes doesn't scale.

There isn't a security team on the planet that will allow an initiative to move forward without:

- **Detailed visualization of security policies and traffic flows** to quickly discover, debug, and resolve Kubernetes connectivity issues
- **Visibility into Kubernetes network flow logs** including Namespace, Label, and network policy metadata
- **Comprehensive network visibility** and evidence of reduced time to determine root cause

Calico Enterprise bundles Elasticsearch and Kibana with bi-directional flow logs for all pods and host connections. Using tools like FlowViz, you can quickly drill down and pinpoint which policies are allowing and denying traffic between their services.



dev

All Names

All Flows

Any Status

Denied Traffic	CPS	PPS	BPS
● dev	4.71k	4.71k	4.71k
● prod	4.71k	4.71k	4.71k

Allowed Traffic	CPS	PPS	BPS
● dev	108k	108k	108k
● dev	94.2k	94.2k	94.2k
● -	13.9k	13.9k	13.9k

- All network flows include source and destination namespaces, pods, labels, and the policies that evaluate each flow.
- All connection attempts inside and outside the cluster are automatically generated so you can quickly identify the source of connectivity issues.

Calico Enterprise also continually monitors traffic for anomalies, and automatically generates alerts. If a compromise is found, containers are automatically quarantined to prohibit any lateral movement.

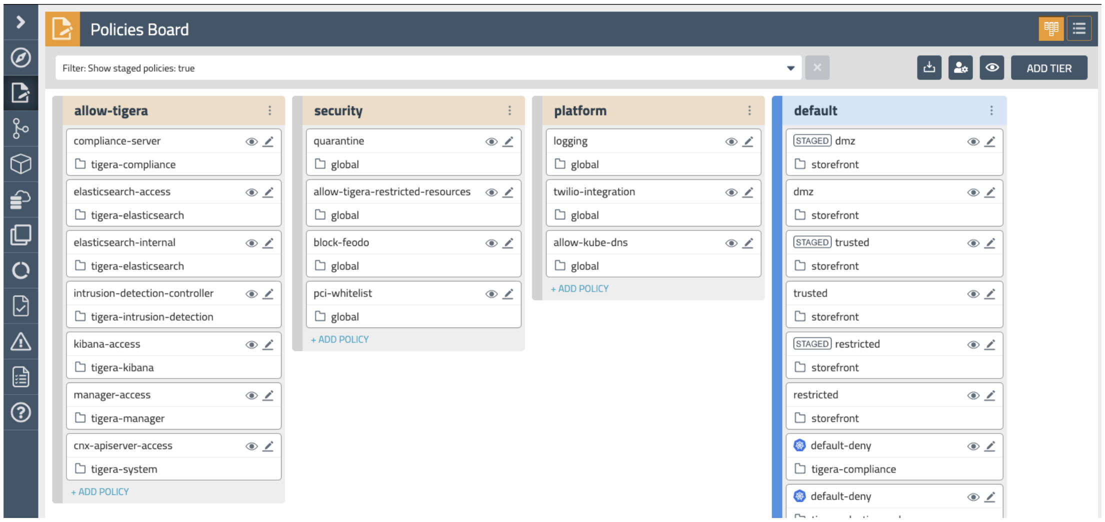
Packet capture is a valuable debugging tool used in daily operations and incident response for microservices and applications. But manually setting up packet capturing can be tedious. Calico Enterprise provides an easy way to capture packets using the widely-known “pcap” format, and export them to visualization tools like WireShark.

Security controls for segmentation

Just like traditional firewalls, applications may need workload isolation and segmentation in Kubernetes. Typical examples are “dev should not talk to prod”, and “microservices in the DMZ can communicate with the public internet, but not directly with backend databases”.

Calico Enterprise provides these isolation features:

- **Hierarchical policy using tiers**
  A tier organizes related network policies. You can then use standard RBAC to restrict access and control who can interact with a tier. To ensure policy changes cannot be made or overwritten by others, you use a tier visualization tool to drag and move tiers in a “higher-precedence” tier ordering workflow.



Policies Board

Filter: Show staged policies: true

ADD TIER

allow-tigera
- compliance-server — tigera-compliance
- elasticsearch-access — tigera-elasticsearch
- elasticsearch-internal — tigera-elasticsearch
- intrusion-detection-controller — tigera-intrusion-detection
- kibana-access — tigera-kibana
- manager-access — tigera-manager
- cnx-apiserver-access — tigera-system
- + ADD POLICY

security
- quarantine — global
- allow-tigera-restricted-resources — global
- block-feodo — global
- pci-whitelist — global
- + ADD POLICY

platform
- logging — global
- twilio-integration — global
- allow-kube-dns — global
- + ADD POLICY

default
- STAGED dmz — storefront
- dmz — storefront
- STAGED trusted — storefront
- trusted — storefront
- STAGED restricted — storefront
- restricted — storefront
- default-deny — tigera-compliance
- default-deny

- **Alerts** on changes to your security controls

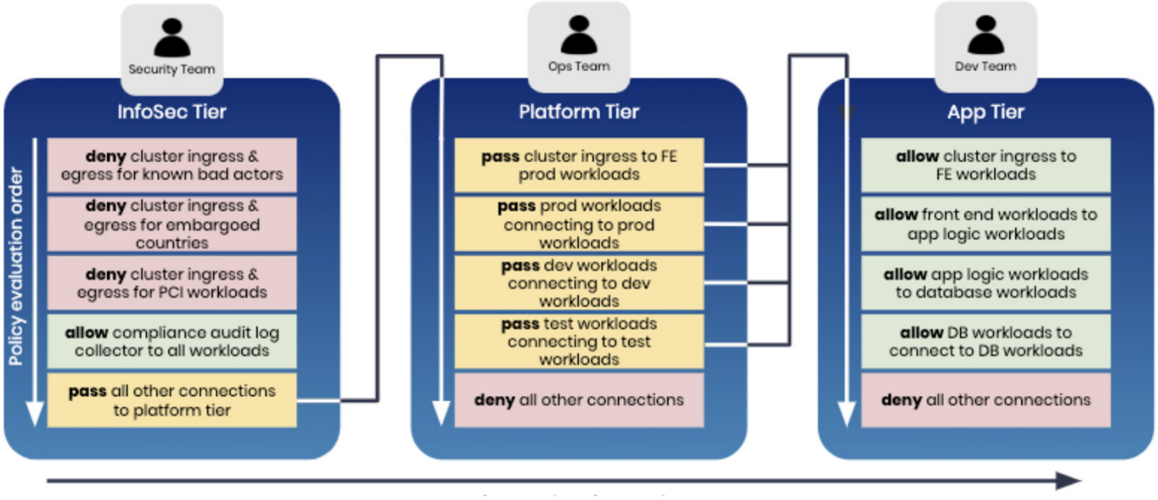
Production, initial apps

As you start deploying apps in Kubernetes, the focus often shifts to network policy lifecycle, automation, and self-service. But many enterprises have existing security tools and processes in place. Leveraging existing firewalls (like AWS and Fortigate) can bridge the gap and minimize disruption to existing workflows and investments.

Self-service with control and compliance

To deploy a microservice to a secure cluster, you need a network policy so the service can communicate with other services and APIs. Many enterprise processes require centralized "policy reviews" to ensure a deployment doesn't advertently override an important security policy for sensitive workloads. If you have daily policy reviews for 100-1000 microservices, deployments will be significantly delayed.

Calico Enterprise provides tiered network policy with standard RBAC to secure clusters so important policies cannot be overridden or violated. The concept of a tier has been around for a long time, and is critical in Kubernetes to allow more autonomy for developers to write their own policies. Tiers provide the guardrails required by security teams, which leads to more delegation of trust across multiple teams.



Security Team

InfoSec Tier

- **deny** cluster ingress & egress for known bad actors
- **deny** cluster ingress & egress for embargoed countries
- **deny** cluster ingress & egress for PCI workloads
- **allow** compliance audit log collector to all workloads
- **pass** all other connections to platform tier

Ops Team

Platform Tier

- **pass** cluster ingress to FE prod workloads
- **pass** prod workloads connecting to prod workloads
- **pass** dev workloads connecting to dev workloads
- **pass** test workloads connecting to test workloads
- **deny** all other connections

Dev Team

App Tier

- **allow** cluster ingress to FE workloads
- **allow** front end workloads to app logic workloads
- **allow** app logic workloads to database workloads
- **allow** DB workloads to connect to DB workloads
- **deny** all other connections

- **Workload two-factor authentication**
  You can authenticate and authorize workloads based attributes including network and cryptographic identity (equivalent to two-factor authentication for workloads). All network flows are logged with the necessary workload identity and metadata information — so you can demonstrate compliance with security policies.

- **Audit logging**
  All changes to security policies are logged. Combined with flow logging, you can quickly and easily demonstrate what policies are in place, and the history of enforcement.

Policy lifecycle and automation

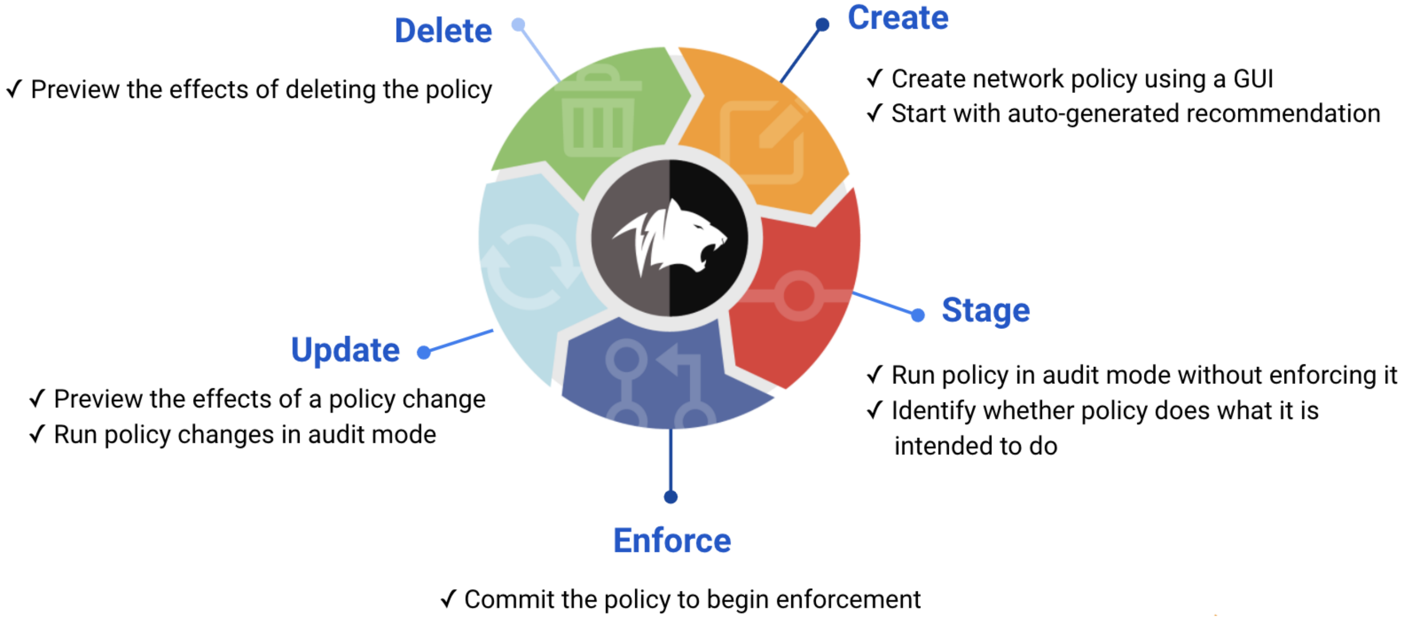
If network policies are misconfigured, they can cause connectivity issues and outages. In traditional hierarchical environments, platform and security teams preview changes to network security policies before they go live. But this doesn't scale. Modern CI/CD processes need teams to move away from hierarchy, and more towards autonomy.

Calico Enterprise provides full network policy lifecycle controls with RBAC including:

- **Policy recommendation**
  New users can generate a policy recommendation — to see valid policies and learn how to write efficient network policies.

- **Preview policy**
  Preview the impacts of policy changes before you apply them.

- **Stage policy and update policy**
  Stage network policies to observe possible traffic implications before enforcing the policy.



Delete
✓ Preview the effects of deleting the policy

Create
✓ Create network policy using a GUI
✓ Start with auto-generated recommendation

Stage
✓ Run policy in audit mode without enforcing it
✓ Identify whether policy does what it is intended to do

Enforce
✓ Commit the policy to begin enforcement

Update
✓ Preview the effects of a policy change
✓ Run policy changes in audit mode

Extend firewalls to Kubernetes

Traditional firewalls are still very much alive in IT security architecture, deploying microservices for applications that create/accept internet or database connections. Defining a firewall rule for ingress or egress access controls 1) does not work in this architecture, and 2) if implemented incorrectly, can block deployments, causing service disruptions. Also, when you define a zone-based security architecture in a cluster using a firewall, you must route all service-to-service traffic through the firewall, which adds latency to applications.

Calico Enterprise provides integrations with popular network security tools to reduce disruption to your existing security workflows.

- **AWS security groups integration**
  Combine AWS security groups with Calico Enterprise network policy to enforce granular access control between Kubernetes pods and AWS VPC resources.

- **Fortigate firewall integration**
  Use Calico Enterprise network policy to control traffic from Kubernetes clusters in FortiGate firewalls. This allows security teams to retain firewall responsibility, freeing up more time for ITOps.

Production, mass migration

As you move into production, the challenges will be familiar friends: threat defense, compliance, efficiency, and scalability. The goal is in sight, but you want repeatable evidence that:

- Security is buttoned up
- Processes are rock solid
- You can meet compliance requirements and SLAs
- Self-service is within arms reach (or at least moving in the right direction)

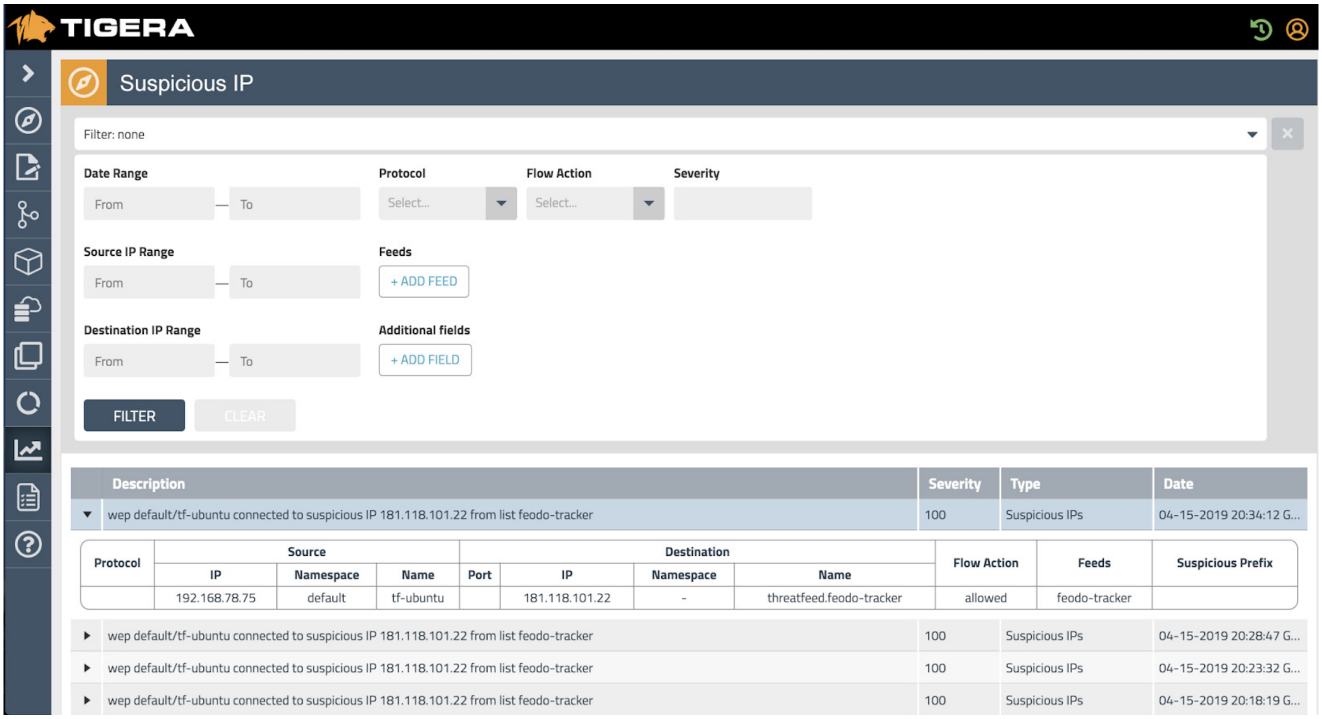
This is the stage where Kubernetes is no longer a mystery; where teams have their feet on solid ground again.

Threat defense and anomaly detection

Cybersecurity remains a top priority for any IT project. Although threat defense and machine learning are not new, managing threats in Kubernetes is different. How do you know if a workload is infected? Are there other ways to find issues other than drilling down into logs and Kibana? Traditional tools are still used, but new approaches are required for Kubernetes clusters.

Calico Enterprise Intrusion Detection System (IDS), identifies advanced persistent threats through behavior-based detection using machine learning, and a rule-based engine for active monitoring.

Threat intelligence feeds
You can detect when Kubernetes clusters communicate with suspicious IPs or domains and get full context for remediation, including which pod(s). You can also use a threat intelligence feed to power a dynamic denylist, either to or from a specific group of sensitive pods, or your entire cluster.



Filter: none

	Description	Severity	Type	Date
▼	wep default/tf-ubuntu connected to suspicious IP 181.118.101.22 from list feodo-tracker	100	Suspicious IPs	04-15-2019 20:34:12 G...

Protocol	Source			Destination				Flow Action	Feeds	Suspicious Prefix
	IP	Namespace	Name	Port	IP	Namespace	Name			
	192.168.78.75	default	tf-ubuntu		181.118.101.22	-	threatfeed.feodo-tracker	allowed	feodo-tracker	

	Description	Severity	Type	Date
▶	wep default/tf-ubuntu connected to suspicious IP 181.118.101.22 from list feodo-tracker	100	Suspicious IPs	04-15-2019 20:28:47 G...
▶	wep default/tf-ubuntu connected to suspicious IP 181.118.101.22 from list feodo-tracker	100	Suspicious IPs	04-15-2019 20:23:32 G...
▶	wep default/tf-ubuntu connected to suspicious IP 181.118.101.22 from list feodo-tracker	100	Suspicious IPs	04-15-2019 20:18:19 G...

Honeypods

Based on the well-known cybersecurity method, “honeypots,” Calico Enterprise honeypods are used to detect and counter cyber attacks. You place decoys disguised as a sensitive asset (called canary pods) at different locations in your Kubernetes cluster. You configure all valid resources to not make connections to the honeypods. Then, if any resources do reach the honeypods, you can assume the connection is suspicious, and that a resource may be compromised.

Calico Enterprise honeypods can be used to detect attacks such as:

- Data exfiltration
- Resources enumeration
- Privilege escalation
- Denial of service
- Vulnerability exploitation attempts

Anomaly detection

Detect and alert on unexpected network behavior. Alert on exploits like Shopify, Tesla, Atlassian, DOS attempts, attempted connections to botnets and command and control servers, and abnormal flow volumes/patterns.

Single management plane and federation

As processes harden and you start to scale, federation is another feature you’ll want to consider. Managing standalone clusters and multiple instances of Elasticsearch in the pre-production stage is not onerous. But going to production with 300+ clusters is not scalable; you need centralized cluster management and log storage.

Calico Enterprise provides these federation features:

Multi-cluster management allows you to securely connect multiple clusters from different cloud providers in a single management plane. Using a single UI, you can see all clusters and switch between them. This architecture is the foundation of a "single pane of glass."

Federated endpoint identity and services
In this stage, you may have multiple teams like development, QA, and others who need to work in parallel across multiple clusters — without negative impacts on each other. Federated endpoint identity and services allow you to share cross-cluster workload and host endpoints, and enable cross-cluster service discovery for a local cluster.

Protect non-cluster hosts with policy

Not all hosts run virtualized workloads (containers managed by Kubernetes or OpenShift). Some physical machines and legacy applications may not be able to move into an orchestrated cluster — but still need to securely communicate with workloads in a cluster.

Calico Enterprise lets you enforce security on hosts using the same robust Calico Enterprise network policy used for workloads.

Compliance

Existing compliance tools that rely on periodic snapshots do not provide accurate assessments of Kubernetes workloads against your compliance standards. Before moving to production, Kubernetes workloads must meet existing organizational/regulatory security and compliance requirements.

Examples of a minimum set of security controls for deployment are:

- Segment dev/stage/prod
- Implement zones
- Support internal/external compliance requirements: PCI, SOX, GDPR, HIPAA, etc.

Calico Enterprise compliance features include:

- A complete inventory of regulated workloads with evidence of enforcement of network controls
- CIS benchmark reports to assess compliance for all assets in a Kubernetes cluster
- Audit reports to see changes to network security controls
- A compliance dashboard in Calico Enterprise Manager to view and export reports from Elasticsearch
- Standard Kubernetes RBAC to grant permissions to view/manage reports

Zero trust network security model

Kubernetes is particularly vulnerable to malware because of the open nature of cluster networking. Without implementing a strong security framework like Zero Trust, you'll find it difficult to detect malware and its spread within a Kubernetes cluster.

The Zero trust network security model *assumes that something in your application or infrastructure is compromised, and is currently hosting some form of malware.* When you implement Zero Trust, networks are resilient — even when attackers breach applications or infrastructure. You can more easily spot reconnaissance activities, and make it difficult for attackers to move laterally.

Calico Enterprise operates under the assumption that your services, network, users – and anything else related to your environment – are potentially compromised. It provides a layered defense model on top of your existing network to lock down your Kubernetes workloads, without requiring any code or configuration changes.

- **Multiple sources of identity**
  Authenticates the identity of every request based on multiple sources, including the L3 network identity and x509 certificate-based cryptographic identity.

- **Multiple enforcements points**
  Calico Enterprise declarative, intent-based policies are enforced at multiple points – including the host, container, and edge layers of the application.

- **Multiple layers of encryption**
  Encryption can be enabled for all traffic within and across all environments, leveraging mutual Transport Layer Security (mTLS) for application and edge layers and IPsec for traffic between hosts.

Roadmap to production with Kubernetes

A modern CI/CD process with secure Kubernetes workloads and hosts, along with greater self-service, is totally achievable with Calico Enterprise.

Tigera offers Kubernetes networking, security, and observability for hybrid and multi-cloud. Tigera can help you define a roadmap with Calico Enterprise features for each stage in your Kubernetes journey.

- To start a roadmap to production, contact **Tigera Support**.
- Need a reference customer who is currently in production with Calico Enterprise? Contact **Tigera Support**.
- To learn more about the features described in this ebook, see **Calico Enterprise technical documentation**.

Resources and Getting Started

Do you have a question about network security for Kubernetes?
Are you interested in setting up a demo, or free trial? We'd love to hear from you!

Contact Us

Getting Started

Calico is an open source networking and network security solution for containers, virtual machines, and native host-based workloads.

Get Started

Try Calico Enterprise Now

Calico Enterprise solves several common roadblocks to enterprise adoption of Kubernetes.

Try Now

Free Training

Free Online Training on Calico. Live sessions.

Sign Up Today

NEW Free Calico Certification

Become a Kubernetes networking and network security expert.

Learn More