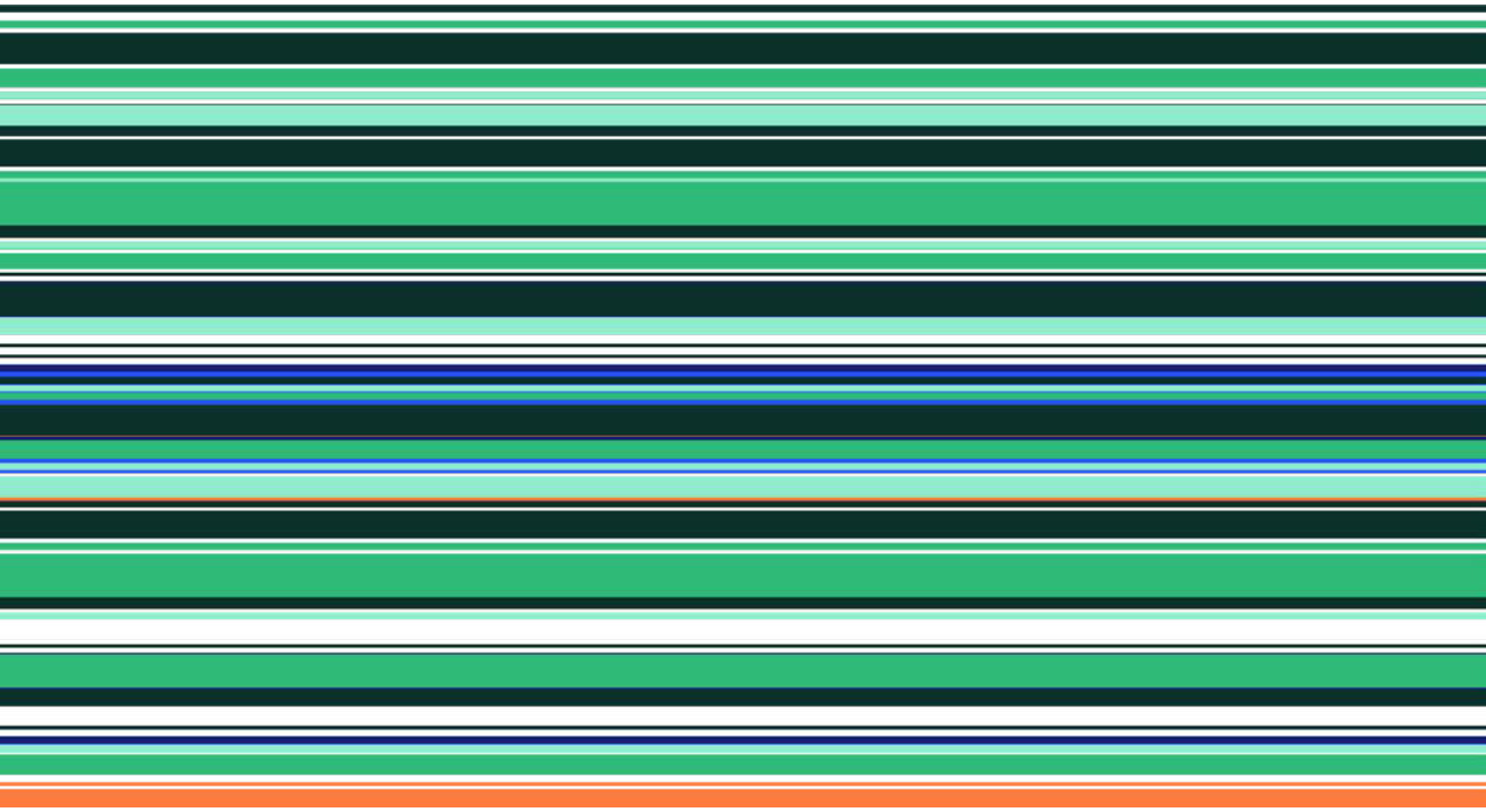
SUSE



White Paper

Diving Deep into Kubernetes Networking

2022

SUSE

Contents

Introduction

Kubernetes has evolved into a strategic platform for deploying and scaling applications in data centers and the cloud. It provides built-in abstractions for efficiently deploying, scaling, and managing applications. Kubernetes also addresses concerns such as storage, networking, load balancing, and multi-cloud deployments.

Networking is a critical component for the success of a Kubernetes implementation. Network components in a Kubernetes cluster control interaction at multiple layers, from communication between containers running on different hosts to exposing services to clients outside of a cluster. The requirements within each environment are different, so before we choose which solution is the most appropriate, we have to understand how networking works within Kubernetes and what benefits each solution provides.

Goals for this book

This book introduces various networking concepts related to Kubernetes that an operator, developer, or decision maker might find useful. Networking is a complex topic and even more so when it comes to a distributed system like Kubernetes. It is essential to understand the technology, the tooling, and the available choices. These choices affect an organization's ability to scale the infrastructure and the applications running on top of it.

The reader is expected to have a basic understanding of containers, Kubernetes, and operating system fundamentals.

How this book is organized

In this book, we cover Kubernetes networking from the basics to the advanced topics. We start by explaining Docker container networking, as Docker is a fundamental component of Kubernetes. We then introduce Kubernetes networking, its unique model and how it seamlessly scales. In doing so, we explain the abstractions that enable Kubernetes to communicate effectively between applications. We touch upon the Container Network Interface (CNI) specification and how it relates to Kubernetes, and finally, we do a deep dive into some of the more popular CNI plugins for Kubernetes including Flannel, Calico, Multus CNI, Cilium, Kube-Vip, and MetalLB. We discuss load balancing, DNS and how to expose applications to the outside world.

This eBook covers Kubernetes networking concepts, but we do not intend for it to be a detailed explanation of Kubernetes in its entirety. For more information on Kubernetes, we recommend reading the eBook, Kubernetes Management for Dummies as well as the Kubernetes documentation.

SUSE

An Introduction to Networking with Docker

Docker follows a unique approach to networking that is very different from the Kubernetes approach. Understanding how Docker works help later in understanding the Kubernetes model, since Docker containers are the fundamental unit of deployment in Kubernetes.

Docker Networking Types

When a Docker container launches, the Docker engine assigns it a network interface with an IP address, a default gateway, and other components, such as a routing table and DNS services. By default, all addresses come from the same pool, and all containers on the same host can communicate with one another. We can change this by defining the network to which the container should connect, either by creating a custom user-defined network or by using a network provider plugin.

The network providers are pluggable using drivers. We connect a Docker container to a particular network by using the `--net` switch when launching it.

The following command launches a container from the *busybox* image and joins it to the host network. This container prints its IP address and then exits.

```
docker run --rm --net=host busybox ip addr
```

Docker offers five network types, each with a different capacity for communication with other network entities.

a) **Host Networking:** The container shares the same IP address and network namespace as that of the host. Services running inside of this container have the same network capabilities as services running directly on the host.

b) **Bridge Networking:** The container runs in a private network internal to the host. Communication is open to other containers in the same network. Communication with services outside of the host goes through network address translation (NAT) before exiting the host. (*This is the default mode of networking when the --net option isn't specified*)

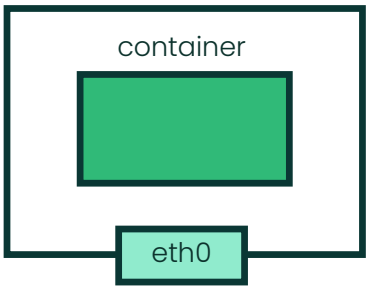
c) **Custom bridge network:** This is the same as Bridge Networking but uses a bridge explicitly created for this (and other) containers. An example of how to use this would be a container that runs on an exclusive "database" bridge network. Another container can have an interface on the default bridge and the database bridge, enabling it to communicate with both networks.

d) **Container-defined Networking:** A container can share the address and network configuration of another container. This type enables process isolation between containers, where each container runs one service but where services can still communicate with one another on the `localhost` address.

e) **No networking:** This option disables all networking for the container.

Host Networking

The host mode of networking allows the Docker container to share the same IP address as that of the host and disables the network isolation otherwise provided by network namespaces. The container's network stack is mapped directly to the host's network stack. All interfaces and addresses on the host are visible within the container, and all communication possible to or from the host is possible to or from the container.



If you run the command `ip addr` on a host (or `ifconfig -a` if your host doesn't have the `ip` command available), you will see information about the network interfaces.

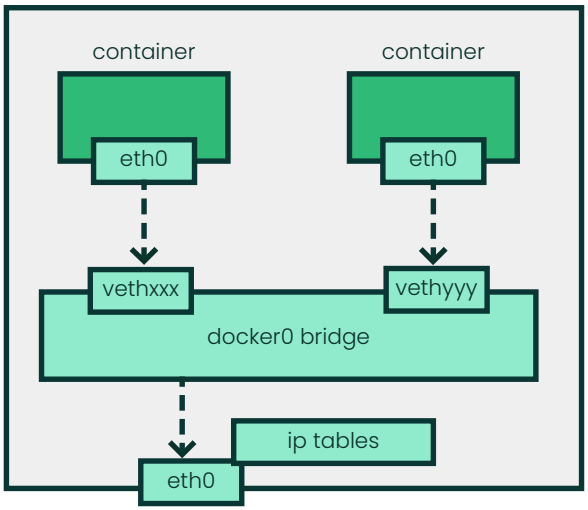
```
 $ ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 52:54:00:6b:21:9e brd ff:ff:ff:ff:ff:ff
    inet 192.168.121.5/24 brd 192.168.121.255 scope global dynamic eth0
       valid_lft 2517sec preferred_lft 2517sec
    inet6 fe80::5054:ff:fe6b:219e/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 52:54:00:04:94:6c brd ff:ff:ff:ff:ff:ff
    inet 70.0.78.56/16 brd 70.0.255.255 scope global dynamic eth1
       valid_lft 150262sec preferred_lft 150262sec
    inet6 fe80::5054:ff:fe04:946c/64 scope link
       valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    link/ether 02:42:d0:3e:db:dd brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:d0ff:fe3e:dbdd/64 scope link
       valid_lft forever preferred_lft forever
```

If you run the same command from a container using host networking, you will see *the same information*.

```
 $ docker run -it --net=host busybox ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 52:54:00:6b:21:9e brd ff:ff:ff:ff:ff:ff
    inet 192.168.121.5/24 brd 192.168.121.255 scope global dynamic eth0
       valid_lft 2388sec preferred_lft 2388sec
    inet6 fe80::5054:ff:fe6b:219e/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 52:54:00:04:94:6c brd ff:ff:ff:ff:ff:ff
    inet 70.0.78.56/16 brd 70.0.255.255 scope global dynamic eth1
       valid_lft 150133sec preferred_lft 150133sec
    inet6 fe80::5054:ff:fe04:946c/64 scope link
       valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    link/ether 02:42:d0:3e:db:dd brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:d0ff:fe3e:dbdd/64 scope link
       valid_lft forever preferred_lft forever
```

Bridge Networking

In a standard Docker installation, the Docker daemon creates a bridge on the host with the name of `docker0`. When a container launches, Docker then creates a virtual ethernet device for it. This device appears within the container as `eth0` and on the host with a name like `vethxxx` where `xxx` is a unique identifier for the interface. The `vethxxx` interface is added to the `docker0` bridge, and this enables communication with other containers on the same host that also use the default bridge.



To demonstrate using the default bridge, run the following command on a host with Docker installed. Since we are not specifying the network - the container will connect to the default bridge when it launches.

Run the `ip addr` and `ip route` commands inside of the container. You will see the IP address of the container with the `eth0` interface:

```
 $ docker run -it --rm busybox /bin/sh
/ # ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
10: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
/ # ip route show
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link  src 172.17.0.2
/ #
```

In another terminal connected to the host, run the `ip addr` command. You will see the corresponding interface created for the container. In the image below it is named `veth5dd2b68@if9`. Yours will be different.

```
 $ ip addr | grep -A 50 veth
10: veth5dd2b68@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP
    link/ether 7e:5d:7a:5d:df:0c brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::7c5d:7aff:fe5d:df0c/64 scope link
       valid_lft forever preferred_lft forever
 $
```

Although Docker mapped the container IPs on the bridge, network services running inside of the container are not visible outside of the host. To make them visible, the Docker Engine must be told when launching a container to map ports from that container to ports on the host. This process is called publishing. For example, if you want to map port 80 of a container to port 8080 on the host, then you would have to publish the port as shown in the following command:

```
docker run --name nginx -p 8080:80 nginx
```

By default, the Docker container can send traffic to any destination. The Docker daemon creates a rule within Netfilter that modifies outbound packets and changes the source address to be the address of the host itself. The Netfilter configuration allows inbound traffic via the rules that Docker creates when initially publishing the container's ports.

The output included below shows the Netfilter rules created by Docker when it publishes a container's ports.

```
 $ docker run -p 8080:80 --name web -d nginx
bc2176e860bb744cb384f67ed52370424b1248c1df0f334bb9ca075231b2e743

 $
 $ docker inspect web --format='{{json .NetworkSettings.IPAddress}}'
"172.17.0.2"
 $
```

```
 $ sudo iptables -L -t nat -v
Chain PREROUTING (policy ACCEPT 80 packets, 20563 bytes)
 pkts bytes target     prot opt in     out     source               destination
  129 67899 DOCKER     all  --  any    any     anywhere             anywhere             ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT 36 packets, 13251 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 50 packets, 3988 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DOCKER     all  --  any    any     anywhere            !127.0.0.0/8          ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 50 packets, 3988 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MASQUERADE  all  --  any    !docker0  172.17.0.0/16        anywhere
   11  1112 RETURN     all  --  any    any     192.168.122.0/24     base-address.mcast.net/24
    0     0 RETURN     all  --  any    any     192.168.122.0/24     255.255.255.255
    0     0 MASQUERADE  tcp  --  any    any     192.168.122.0/24    !192.168.122.0/24     masq ports: 1024-65535
    0     0 MASQUERADE  udp  --  any    any     192.168.122.0/24    !192.168.122.0/24     masq ports: 1024-65535
    0     0 MASQUERADE  all  --  any    any     192.168.122.0/24    !192.168.122.0/24
    0     0 MASQUERADE  tcp  --  any    any     172.17.0.2           172.17.0.2           tcp dpt:http

Chain DOCKER (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  docker0 any     anywhere             anywhere
    0     0 DNAT       tcp  --  !docker0 any    anywhere             anywhere             tcp dpt:http-alt to:172.17.0.2:80
 $
```

The next image shows the NAT table within Netfilter:

```
 $ sudo iptables -L -t filter -v
Chain INPUT (policy ACCEPT 769 packets, 124K bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 ACCEPT     udp  --  virbr0 any     anywhere             anywhere             udp dpt:domain
    0     0 ACCEPT     tcp  --  virbr0 any     anywhere             anywhere             tcp dpt:domain
    0     0 ACCEPT     udp  --  virbr0 any     anywhere             anywhere             udp dpt:bootps
    0     0 ACCEPT     tcp  --  virbr0 any     anywhere             anywhere             tcp dpt:bootps

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DOCKER-ISOLATION  all  --  any    any     anywhere             anywhere
    0     0 ACCEPT     all  --  any    docker0  anywhere             anywhere             ctstate RELATED,ESTABLISHED
    0     0 DOCKER     all  --  any    docker0  anywhere             anywhere
    0     0 ACCEPT     all  --  docker0 !docker0  anywhere             anywhere
    0     0 ACCEPT     all  --  docker0 docker0  anywhere             anywhere
    0     0 ACCEPT     all  --  any    virbr0  anywhere             192.168.122.0/24     ctstate RELATED,ESTABLISHED
    0     0 ACCEPT     all  --  virbr0 any     192.168.122.0/24     anywhere
    0     0 ACCEPT     all  --  virbr0 virbr0  anywhere             anywhere
    0     0 REJECT     all  --  any    virbr0  anywhere             anywhere             reject-with icmp-port-unreachable
    0     0 REJECT     all  --  virbr0 any     anywhere             anywhere             reject-with icmp-port-unreachable

Chain OUTPUT (policy ACCEPT 767 packets, 84530 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 ACCEPT     udp  --  any    virbr0  anywhere             anywhere             udp dpt:bootpc

Chain DOCKER (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 ACCEPT     tcp  --  !docker0 docker0  anywhere             172.17.0.2           tcp dpt:http

Chain DOCKER-ISOLATION (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  any    any     anywhere             anywhere
 $
```

Custom Bridge Network

There is no requirement to use the default bridge on the host; it's easy to create a new bridge network and attach containers to it. This provides better isolation and interoperability between containers, and custom bridge networks have better security and features than the default bridge.

- All containers in a custom bridge can communicate with the ports of other containers on that bridge. This means that you do not need to publish the ports explicitly. It also ensures that the communication between them is secure. Imagine an application in which a backend container and a database container need to communicate and where we also want to make sure that no external entity can talk to the database. We do this with a custom bridge network in which only the database container and the backend containers reside. You can explicitly expose the backend API to the rest of the world using port publishing.
- The same is true with environment variables - environment variables in a bridge network are shared by all containers on that bridge.
- Network configuration options such as MTU can differ between applications. By creating a bridge, you can configure the network to best suit the applications connected to it.

To create a custom bridge network and two containers that use it, run the following commands:

```
$docker network create mynetwork
$docker run -it --rm --name=container-a
--network=mynetwork busybox /bin/sh
$docker run -it --rm --name=container-b
--network=mynetwork busybox /bin/sh
```

Container-defined Network

A specialized case of custom networking is when a container joins the network of another container. This is similar to how a Pod works in Kubernetes.

The following commands launch two containers that share the same network namespace and thus share the same IP address. Services running on one container can talk to services running on the other via the `localhost` address.

```
$docker run -it --rm --name=container-a busybox /bin/sh
$docker run -it --rm --name=container-b --network=container:container-a busybox /bin/sh
```

No Networking

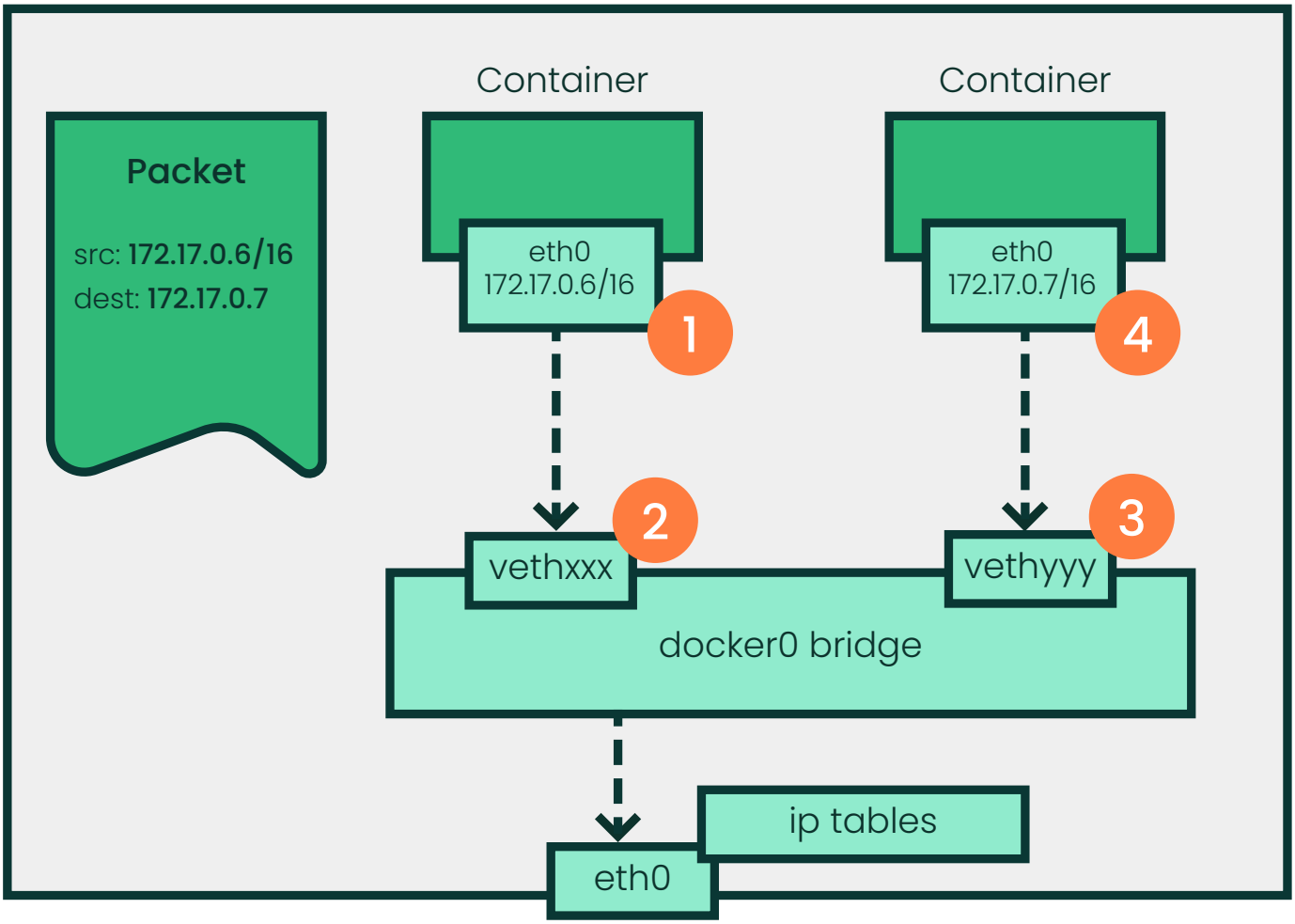
This mode is useful when the container does not need to communicate with other containers or with the outside world. It is not assigned an IP address, and it cannot publish any ports.

```
docker run --net=none --name busybox busybox ip a
```

SUSE

Container-to-Container Communication

How do two containers on the same bridge network talk to one another?



In the above diagram, two containers running on the same host connect via the `docker0` bridge. If `172.17.0.6` (on the left-hand side) wants to send a request to `172.17.0.7` (the one on the right-hand side), the packets move as follows:

1. A packet leaves the container via `eth0` and lands on the corresponding `vethxxx` interface.
2. The `vethxxx` interface connects to the `vethyyy` interface via the `docker0` bridge.
3. The `docker0` bridge forwards the packet to the `vethyyy` interface.
4. The packet moves to the `eth0` interface within the destination container.

We can see this in action by using `ping` and `tcpdump`. Create two containers and inspect their network configuration with `ip addr` and `ip route`. The default route for each container is via the `eth0` interface.

```
 $ docker run -it --rm --name=bb1 busybox /bin/sh
/ #
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
19: eth0@if20: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
/ #
/ # ip route
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link  src 172.17.0.2
/ #
```

```
 $ docker run -it --rm --name=bb2 busybox /bin/sh
/ #
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
21: eth0@if22: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
/ #
/ # ip route
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link  src 172.17.0.3
/ #
```

Ping one container from the other, and let the command run so that we can inspect the traffic. Run `tcpdump` on the `docker0` bridge on the host machine. You will see in the output that the traffic moves between the two containers via the `docker0` bridge.

```
 $ sudo tcpdump -i docker0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on docker0, link-type EN10MB (Ethernet), capture size 262144 bytes
13:50:51.934917 ARP, Request who-has 172.17.0.3 tell 172.17.0.2, length 28
13:50:51.934987 ARP, Reply 172.17.0.3 is-at 02:42:ac:11:00:03 (oui Unknown), length 28
13:50:51.935080 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 0, length 64
13:50:51.935149 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 0, length 64
13:50:52.935850 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 1, length 64
13:50:52.935913 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 1, length 64
13:50:53.936596 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 2, length 64
13:50:53.936655 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 2, length 64
13:50:54.936959 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 3, length 64
13:50:54.937027 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 3, length 64
13:50:55.937725 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 4, length 64
13:50:55.937803 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 4, length 64
13:50:56.938254 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 5, length 64
13:50:56.938323 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 5, length 64
13:50:57.129070 ARP, Request who-has 172.17.0.2 tell 172.17.0.3, length 28
13:50:57.129111 ARP, Reply 172.17.0.2 is-at 02:42:ac:11:00:02 (oui Unknown), length 28
13:50:57.939018 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 6, length 64
13:50:57.939093 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 6, length 64
13:50:58.939400 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 7, length 64
13:50:58.939498 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 7, length 64
13:50:59.939878 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 8, length 64
13:50:59.939955 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 8, length 64
```

Container Communication Between Hosts

So far we have discussed scenarios in which containers communicate within a single host. While interesting, real-world applications require communication between containers running on different hosts.

Cross-host networking usually uses an *overlay network*, which builds a mesh between hosts and employs a large block of IP addresses within that mesh. The network driver tracks which addresses are on which host and shuttles packets between the hosts as necessary for inter-container communication.

Overlay networks can be encrypted or unencrypted. Unencrypted networks are acceptable for environments in which all of the hosts are within the same LAN, but because overlay networks enable communication between hosts across the Internet, consider the security requirements when choosing a network driver. If the packets traverse a network that you don't control, encryption is a better choice.

The overlay network functionality built into Docker is called Swarm. When you connect a host to a swarm, the Docker engine on each host handles communication and routing between the hosts.

Other overlay networks exist, such as IPVLAN, VxLAN, and MACVLAN. More solutions are available for Kubernetes.

For more information on pure-Docker networking implementations for cross-host networking (including Swarm mode and libnetwork), please refer to the documentation available at the Docker website, https://docs.docker.com/.

Interlude: Netfilter and iptables rules

In the earlier section on Docker networking, we looked at how Docker handles communication between containers. On a Linux host, the component which handles this is called Netfilter, or more commonly by the command used to configure it: `iptables`.

Netfilter manages the rules that define network communication for the Linux kernel. These rules permit, deny, route, modify, and forward packets. It organizes these rules into tables according to their purpose.

The Filter Table

Rules in the Filter table control if a packet is allowed or denied. Packets which are allowed are forwarded whereas packets which are denied are either rejected or silently dropped.

The NAT Table

These rules control network address translation. They modify the source or destination address for the packet, changing how the kernel routes the packet.

The Mangle Table

The headers of packets which go through this table are altered, changing the way the packet behaves. Netfilter might shorten the TTL, redirect it to a different address, or change the number of network hops.

Raw Table

This table marks packets to bypass the iptables stateful connection tracking.

Security Table

This table sets the SELinux security context marks on packets. Setting the marks affects how SELinux (or systems that can interpret SELinux security contexts) handle the packets. The rules in this table set marks on a per-packet or per-connection basis.

Netfilter organizes the rules in a table into chains. Chains are the means by which Netfilter hooks in the kernel intercept packets as they move through processing. Packets flow through one or more chains and exit when they match a rule.

A rule defines a set of conditions, and if the packet matches those conditions, an action is taken. The universe of actions is diverse, but examples include:

- Block all connections originating from a specific IP address.
- Block connections to a network interface.
- Allow all HTTP/HTTPS connections.
- Block connections to specific ports.

The action that a rule takes is called a *target*, and represents the decision to *accept*, *drop*, or *forward* the packet.

The system comes with five default chains that match different phases of a packet's journey through processing: PREROUTING, INPUT, FORWARD, OUTPUT, and POSTROUTING. Users and programs may create additional chains and inject rules into the system chains to forward packets to a custom chain for continued processing. This architecture allows the Netfilter configuration to follow a logical structure, with chains representing groups of related rules.

Docker creates several chains, and it is the actions of these chains that handle communication between containers, the host, and the outside world.

SUSE

An Introduction to Kubernetes Networking

Kubernetes networking builds on top of the Docker and Netfilter constructs to tie multiple components together into applications. Kubernetes resources have specific names and capabilities, and we want to understand those before exploring their inner workings.

Pods

The smallest unit of deployment in a Kubernetes cluster is the Pod, and all of the constructs related to scheduling and orchestration assist in the deployment and management of Pods.

In the simplest definition, a Pod encapsulates one or more containers. Containers in the same Pod always run on the same host. They share resources such as the network namespace and storage.

Each Pod has a routable IP address assigned to it, not to the containers running within it. Having a shared network space for all containers means that the containers inside can communicate with one another over the `localhost` address, a feature not present in traditional Docker networking.

The most common use of a Pod is to run a single container. Situations where different processes work on the same shared resource, such as content in a storage volume, benefit from having multiple containers in a single Pod. Some projects inject containers into running Pods to deliver a service. An example of this is the Istio service mesh, which uses this injected container as a proxy for all communication.

Because a Pod is the basic unit of deployment, we can map it to a single instance of an application. For example, a three-tier application that runs a user interface (UI), a backend, and a database would model the deployment of the application on Kubernetes with three Pods. If one tier of the application needed to scale, the number of Pods in that tier could scale accordingly.

Workloads

Production applications with users run more than one instance of the application. This enables fault tolerance, where if one instance goes down, another handles the traffic so that users don't experience a disruption to the service. In a traditional model that doesn't use Kubernetes, these types of deployments require that an external person or software monitors the application and acts accordingly.

Kubernetes recognizes that an application might have unique requirements. Does it need to run on every host? Does it need to handle state to avoid data corruption? Can all of its pieces run anywhere, or do they need special scheduling consideration? To accommodate those situations where a default structure won't give the best results, Kubernetes provides abstractions for different workload types.

ReplicaSet

The ReplicaSet maintains the desired number of copies of a Pod running within the cluster. If a Pod or the host on which it's running fails, Kubernetes launches a replacement. In all cases, Kubernetes works to maintain the desired state of the ReplicaSet.

Deployment

A Deployment manages a ReplicaSet. Although it's possible to launch a ReplicaSet directly or to use a ReplicationController, the use of a Deployment gives more control over the rollout strategies of the Pods that the ReplicaSet controller manages. By defining the desired states of Pods through a Deployment, users can perform updates to the image running within the containers and maintain the ability to perform rollbacks.

DaemonSet

A DaemonSet runs one copy of the Pod on each node in the Kubernetes cluster. This workload model provides the flexibility to run daemon processes such as log management, monitoring, storage providers, or network providers that handle Pod networking for the cluster.

StatefulSet

A StatefulSet controller ensures that the Pods it manages have durable storage and persistent identity. StatefulSets are appropriate for situations where Pods have a similar definition but need a unique identity, ordered deployment and scaling, and storage that persists across Pocd rescheduling.

Pod Networking

The Pod is the smallest unit in Kubernetes, so it is essential to first understand Kubernetes networking in the context of communication between Pods. Because a Pod can hold more than one container, we can start with a look at how communication happens between containers in a Pod. Although Kubernetes can use Docker for the underlying container runtime, its approach to networking differs slightly and imposes some basic principles:

- Any Pod can communicate with any other Pod without the use of network address translation (NAT). To facilitate this, Kubernetes assigns each Pod an IP address that is routable within the cluster.
- A node can communicate with a Pod without the use of NAT.
- A Pod's awareness of its address is the same as how other resources see the address. The host's address doesn't mask it.

These principles give a unique and first-class identity to every Pod in the cluster. Because of this, the networking model is more straightforward and does not need to include port mapping for the running container workloads. By keeping the model simple, migrations into a Kubernetes cluster require fewer changes to the container and how it communicates.

The Pause Container

A piece of infrastructure that enables many networking features in Kubernetes is known as the pause container. This container runs alongside the containers defined in a Pod and is responsible for providing the network namespace that the other containers share. It is analogous to joining the network of another container that we described in the User Defined Network section above.

The pause container was initially designed to act as the `init` process within a PID namespace shared by all containers in the Pod. It performed the function of reaping zombie processes when a container died. PID namespace sharing is now disabled by default, so unless it has been explicitly enabled in the kubelet, all containers run their process as PID 1.

If we launch a Pod running Nginx, we can inspect the Docker container running within the Pod.

```
 $ kubectl run nginx --image=nginx
deployment.apps "nginx" created
 $ kubectl get pods -o wide | grep nginx
nginx-64f497f8fd-2c4mh   1/1       Running   0          58s       192.168.2.245   k8s-n-4
 $
```

When we do so, we see that the container does not have the network settings provided to it. The pause container which runs as part of the Pod is the one which gives the networking constructs to the Pod.

Note: Run the commands below on the host where the `nginx` Pod is scheduled.

```
 $ docker ps | grep nginx
f2464c7efc15        nginx                  "nginx -g 'daemon of…"   10 minutes ago
 Up 10 minutes                          k8s_nginx_nginx-64f497f8fd-2c4mh_default_d8ba3
700-c554-11e8-a1a4-5254006b219e_0
b2c327fefdf3        k8s.gcr.io/pause:3.1   "/pause"                 11 minutes ago
 Up 11 minutes                          k8s_POD_nginx-64f497f8fd-2c4mh_default_d8ba370
0-c554-11e8-a1a4-5254006b219e_0
 $
 $ docker inspect f2464c7efc15 --format='{{json .NetworkSettings}}'
{"Bridge":"","SandboxID":"","HairpinMode":false,"LinkLocalIPv6Address":"","LinkLocalIPv
6PrefixLen":0,"Ports":{},"SandboxKey":"","SecondaryIPAddresses":null,"SecondaryIPv6Addr
esses":null,"EndpointID":"","Gateway":"","GlobalIPv6Address":"","GlobalIPv6PrefixLen":0
,"IPAddress":"","IPPrefixLen":0,"IPv6Gateway":"","MacAddress":"","Networks":{}}
 $

 $ docker inspect b2c327fefdf3 --format='{{json .NetworkSettings}}'
{"Bridge":"","SandboxID":"cd7d6f2e0b8f397dd05750372878dfc5604da8481ffe54d5e8b661463d13e
d93","HairpinMode":false,"LinkLocalIPv6Address":"","LinkLocalIPv6PrefixLen":0,"Ports":{
},"SandboxKey":"/var/run/docker/netns/cd7d6f2e0b8f","SecondaryIPAddresses":null,"Second
aryIPv6Addresses":null,"EndpointID":"","Gateway":"","GlobalIPv6Address":"","GlobalIPv6P
refixLen":0,"IPAddress":"","IPPrefixLen":0,"IPv6Gateway":"","MacAddress":"","Networks":
{"none":{"IPAMConfig":null,"Links":null,"Aliases":null,"NetworkID":"70d4a5bdb3244d50e75
9853147a5bb1dbcfc9676aa6d53a7891b701e297d78c8","EndpointID":"e421ac6a3008c1013b70efaeb2
31c759eaa44f2d56ca16664663a9612188e9b5","Gateway":"","IPAddress":"","IPPrefixLen":0,"IP
v6Gateway":"","GlobalIPv6Address":"","GlobalIPv6PrefixLen":0,"MacAddress":"","DriverOpt
s":null}}}
 $
```

Intra-Pod Communication

Kubernetes follows the IP-per-Pod model where it assigns a routable IP address to the Pod. The containers within the Pod share the same network space and communicate with one another over `localhost`. Like processes running on a host, two containers cannot each use the same network port, but we can work around this by changing the manifest.

Inter-Pod Communication

Because it assigns routable IP addresses to each Pod, and because it requires that all resources see the address of a Pod the same way, Kubernetes assumes that all Pods communicate with one another via their assigned addresses. Doing so removes the need for an external service discovery mechanism.

Kubernetes Service

Pods are ephemeral. The services that they provide may be critical, but because Kubernetes can terminate Pods at any time, they are unreliable endpoints for direct communication. For example, the number of Pods in a ReplicaSet might change as the Deployment scales it up or down to accommodate changes in load on the application, and it is unrealistic to expect every client to track these changes while communicating with the Pods. Instead, Kubernetes offers the Service resource, which provides a stable IP address and balances traffic across all of the Pods behind it. This abstraction brings stability and a reliable mechanism for communication between microservices.

Services which sit in front of Pods use a *selector* and *labels* to find the Pods they manage. All Pods with a label that matches the selector receive traffic through the Service. Like a traditional load balancer, the service can expose the Pod functionality at any port, irrespective of the port in use by the Pods themselves.

Kube-proxy

The kube-proxy daemon that runs on all nodes of the cluster allows the Service to map traffic from one port to another.

This component configures the Netfilter rules on all of the nodes according to the Service's definition in the API server. From Kubernetes 1.9 onward it uses the netlink interface to create IPVS rules. These rules direct traffic to the appropriate Pod.

Kubernetes Service Types

A service definition specifies the type of Service to deploy, with each type of Service having a different set of capabilities.

ClusterIP

This type of Service is the default and exists on an IP that is only visible within the cluster. It enables cluster resources to reach one another via a known address while maintaining the security boundaries of the cluster itself. For example, a database used by a backend application does not need to be visible outside of the cluster, so using a service of type ClusterIP is appropriate. The backend application would expose an API for interacting with records in the database, and a frontend application or remote clients would consume that API.

NodePort

A Service of type NodePort exposes the same port on every node of the cluster. The range of available ports is a cluster-level configuration item, and the Service can either choose one of the ports at random or have one designated in its configuration. This type of Service automatically creates a ClusterIP Service as its target, and the ClusterIP Service routes traffic to the Pods.

External load balancers frequently use NodePort services. They receive traffic for a specific site or address and forward it to the cluster on that specific port.

LoadBalancer

When working with a cloud provider for whom support exists within Kubernetes, a Service of type LoadBalancer creates a load balancer in that provider's infrastructure. The exact details of how this happens differ between providers, but all create the load balancer asynchronously and configure it to proxy the request to the corresponding Pods via NodePort and ClusterIP Services that it also creates.

In a later section, we explore Ingress Controllers and how to use them to deliver a load balancing solution for a cluster.

DNS

As we stated above, Pods are ephemeral, and because of this, their IP addresses are not reliable endpoints for communication. Although Services solve this by providing a stable address in front of a group of Pods, consumers of the Service still want to avoid using an IP address. Kubernetes solves this by using DNS for service discovery.

The default internal domain name for a cluster is `cluster.local`. When you create a Service, it assembles a subdomain of `namespace.svc.cluster.local` (where *namespace* is the namespace in which the service is running) and sets its name as the hostname. For example, if the service was named *nginx* and ran in the *default* namespace, consumers of the service would be able to reach it as `nginx.default.svc.cluster.local`. If the service's IP changes, the hostname remains the same. There is no interruption of service.

The default DNS provider for Kubernetes is KubeDNS, but it's a pluggable component. Beginning with Kubernetes 1.11 CoreDNS is available as an alternative. In addition to providing the same basic DNS functionality within the cluster, CoreDNS supports a wide range of plugins to activate additional functionality.

Service Mesh

Modern applications are typically composed of distributed collections of microservices, each of which performs a discrete business function. As a network of microservices changes and grows, the interactions between them can become increasingly difficult to manage and understand. In such situations, it is useful to have a service mesh as a dedicated infrastructure layer to control service-to-service communication over a network.

A service mesh controls the delivery of service requests in an application so that separate parts of an application can communicate with each other. Service meshes can make service-to-service communication fast, reliable and secure.

Core features provided by a service mesh typically include:

- **Traffic Management** such as ingress and egress routing, circuit breaking, and mirroring.
- **Security** with resources to authenticate and authorize traffic and users, including mTLS.
- **Observability** of logs, metrics, and distributed traffic flows.

In addition, service meshes may offer service discovery, load balancing, metrics, and failure recovery, and even more complex operational requirements such as A/B testing, canary deployments, rate limiting, encryption, and end-to-end authentication.

Istio Service Mesh

Istio is an open-source tool that makes it easier for DevOps teams to observe, secure, control, and troubleshoot the traffic within a complex network of microservices. Its features offer a uniform and efficient way to secure, connect, and monitor services. Users can gain load balancing, service-to-service authentication, and monitoring, generally with few or no service code changes. Its control plane brings features that include:

- Secure service-to-service communication in a cluster with TLS encryption, identity-based authentication and authorization
- Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic
- Granular control of traffic behavior with routing rules, retries, failovers, and fault injection
- A pluggable policy layer and configuration API supporting access controls, rate limits and quotas
- Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress

Functionally, Istio has two components: the data plane and the control plane. The data plane provides communication between services. Istio uses a proxy to intercept all your network traffic, allowing a broad set of application-aware features based on configuration you set. An Envoy proxy is deployed along with each service that you start in your cluster, or runs alongside services running on VMs. This enables Istio to understand the traffic being sent and make decisions based on what type of traffic it is, and which services are communicating. The Istio control plane takes your desired configuration, and its view of the services, and dynamically programs the proxy servers, updating them as the rules or the environment changes.

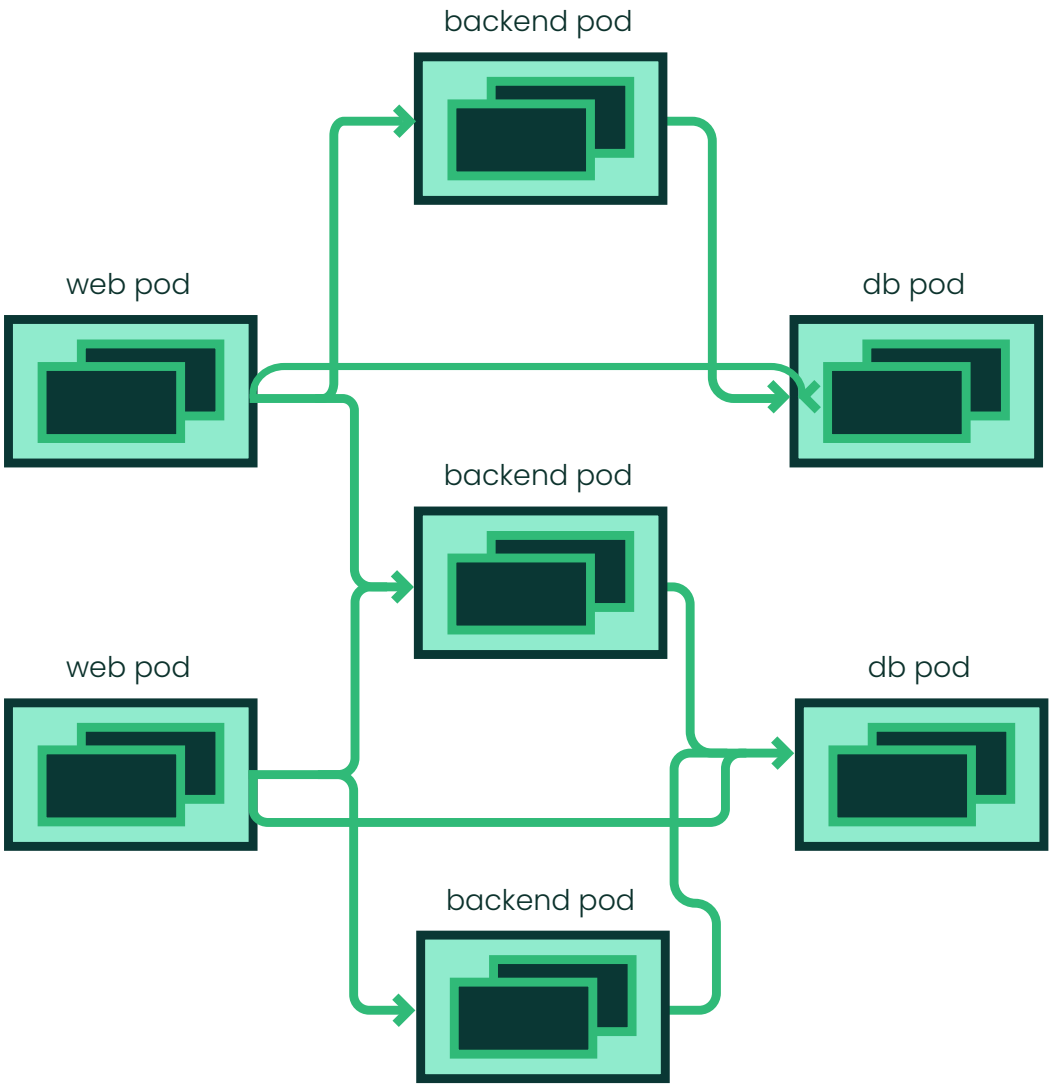
Network Policy

In an enterprise deployment of Kubernetes the cluster often supports multiple projects with different goals. Each of these projects has different workloads, and each of these might require a different security policy.

Pods, by default, do not filter incoming traffic. There are no firewall rules for inter-Pod communication. Instead, this responsibility falls to the NetworkPolicy resource, which uses a specification to define the network rules applied to a set of Pods.

> The network policies are defined in Kubernetes, but the CNI plugins that support network policy implementation do the actual configuration and processing. In a later section, we look at CNI plugins and how they work.

The image below shows a standard three-tier application with a UI, a backend service, and a database, all deployed within a Kubernetes cluster.



Requests to the application arrive at the web Pods, which then initiate a request to the backend Pods for data. The backend Pods process the request and perform CRUD operations against the database Pods.

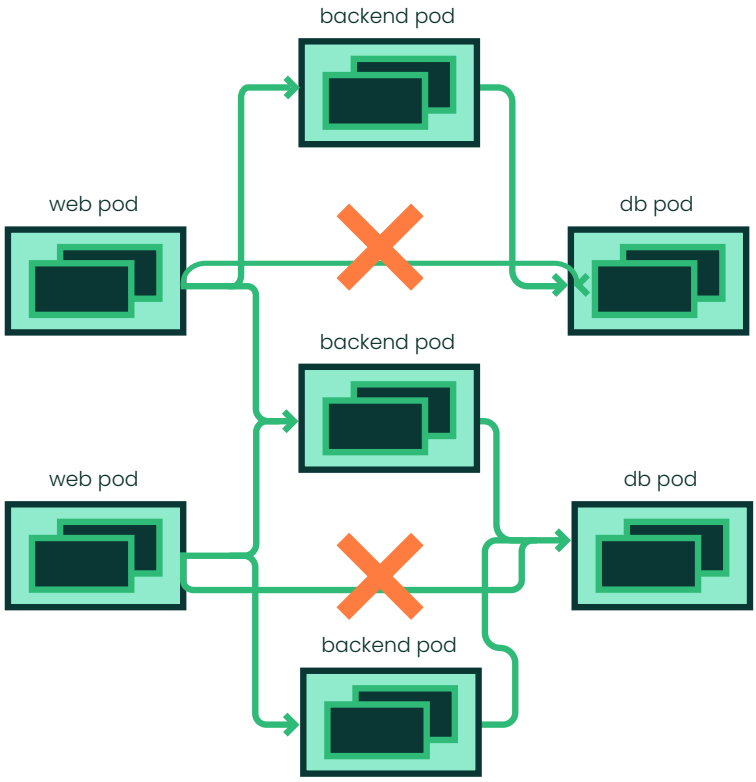
If the cluster is not using a network policy, any Pod can talk to any other Pod. Nothing prevents the web Pods from communicating directly with the database Pods. If the security requirements of the cluster dictate a need for clear separation between tiers, a network policy enforces it.

The policy defined below states that the database Pods can only receive traffic from the Pods with the labels `app=myapp` and `role=backend`. It also defines that the backend Pods can only receive traffic from Pods with the labels `app=myapp` and `role=web`.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: backend-access-ingress
spec:
 podSelector:
   matchLabels:
     app: myapp
     role: backend
 ingress:
 - from:
   - podSelector:
       matchLabels:
         app: myapp
         role: web

kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: db-access-ingress
spec:
 podSelector:
   matchLabels:
     app: myapp
     role: db
 ingress:
 - from:
   - podSelector:
       matchLabels:
         app: myapp
         role: backend
```

With this network policy in place, Kubernetes blocks communication between the web and database tiers.



How a Network Policy Works

In addition to the fields used by all Kubernetes manifests, the specification of the NetworkPolicy resource requires some extra fields.

podSelector

This field tells Kubernetes how to find the Pods to which this policy applies. Multiple network policies can select the same set of Pods, and the ingress rules are applied sequentially. The field is not optional, but if the manifest defines a key with no value, it applies to all Pods in the namespace.

policyTypes

This field defines the direction of network traffic to which the rules apply. If missing, Kubernetes interprets the rules and only applies them to ingress traffic unless egress rules also appear in the rules list. This default interpretation simplifies the manifest's definition by having it adapt to the rules defined later.

Because Kubernetes always defines an ingress policy if this field is unset, a network policy for egress-only rules must explicitly define the `policyType` of `Egress`.

egress

Rules defined under this field apply to egress traffic from the selected Pods to destinations defined in the rule. Destinations can be an IP block (`ipBlock`), one or more Pods (`podSelector`), one or more namespaces (`namespaceSelector`), or a combination of both `podSelector` and `nameSpaceSelector`.

```
egress:
- to:
  - ipBlock:
      cidr: 10.0.0.0/24
  ports:
  - protocol: TCP
    port: 5978
```

The following rule permits traffic from the Pods to any address in `10.0.0.0/24` and only on TCP port 5978:

The next rule permits outbound traffic for Pods with the labels `app=myapp` and `role=backend` to any host on TCP or UDP port 53:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: db-egress-denyall
spec:
 podSelector:
   matchLabels:
     app: myapp
     role: backend
 policyTypes:
 - Egress
 egress:
 - ports:
   - port: 53
     protocol: UDP
   - port: 53
     protocol: TCP
```

Egress rules work best to limit a resource's communication to the other resources on which it relies. If those resources are in a specific block of IP addresses, use the `ipBlock` selector to target them, specifying the appropriate ports:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: db-egress-denyall
spec:
 podSelector:
   matchLabels:
     app: myapp
     role: backend
 policyTypes:
 - Egress
 egress:
 - ports:
   - port: 53
     protocol: UDP
   - port: 53
     protocol: TCP
 - to:
   - ipBlock:
       cidr: 10.0.0.0/24
   ports:
   - protocol: TCP
     port: 3306
```

Ingress

Rules listed in this field apply to traffic that is inbound to the selected Pods. If the field is empty, all inbound traffic will be blocked. The example below permits inbound access from any address in `172.17.0.0/16` unless it's within `172.17.1.0/24`. It also permits traffic from any Pod in the namespace `myproject`.

(Note the subtle distinction in how the rules are listed. Because `namespaceSelector` is a separate item in the list, it matches with an or value. Had `namespaceSelector` been listed as an additional key in the first list item, it would permit traffic that came from the specified `ipBlock` and was also from the namespace `myproject`.)

```
ingress:
- from:
  - ipBlock:
      cidr: 172.17.0.0/16
      except:
      - 172.17.1.0/24
  - namespaceSelector:
      matchLabels:
        project: myproject
  - podSelector:
      matchLabels:
        role: frontend
ports:
  - protocol: TCP
    port: 6379
```

The next policy permits access to the Pods labeled `app=myapp` and `role=web` from all sources, external or internal.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: web-allow-all-access
spec:
 podSelector:
   matchLabels:
     app: myapp
     role: web
 ingress:
 - from: []
```

Consider, however, that this allows traffic to any port on those Pods. Even if no other ports are listening, the principle of least privilege states that we only want to expose what we need to expose for the services to work. The following modifications to the NetworkPolicy take this rule into account by only allowing inbound traffic to the ports where our Service is running.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: web-allow-all-access-specific-port
spec:
 podSelector:
   matchLabels:
     app: myapp
     role: web
 ingress:
 - ports:
   - port: 8080
   from: []
```

Apart from opening incoming traffic on certain ports, you can also enable all traffic from a set of Pods inside the cluster. This enables a few trusted applications to reach out from the application on all ports and is especially useful when workloads in a cluster communicate with each other over many random ports. The opening of traffic from certain Pods is achieved by using labels as described in the policy below.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: web-allow-internal-port80
spec:
 podSelector:
   matchLabels:
     app: “myapp”
     role: “web”
 ingress:
 - ports:
   - port: 8080
   from:
   - podSelector:
       matchLabels:
         app: “mytestapp”
         role: “web-test-client”
```

Even if a Service listens on a different port than where the Pod’s containers listen, use the container ports in the network policy. Ingress rules affect inter-Pod communication, and the policy does not know about the abstraction of the service.

Container Networking Interface

The Container Networking Interface (CNI) project is also under the governance of the CNCF. It provides a specification and a series of libraries for writing plugins to configure network interfaces in Linux containers.

The specification requires that providers implement their plugin as a binary executable that the container engine invokes. Kubernetes does this via the Kubelet process running on each node of the cluster.

The CNI specification expects the container runtime to create a new network namespace before invoking the CNI plugin. The plugin is then responsible for connecting the container's network with that of the host. It does this by creating the virtual Ethernet devices that we discussed earlier.

Kubernetes and CNI

Kubernetes natively supports the CNI model. It gives its users the freedom to choose the network provider or product best suited for their needs.

To use the CNI plugin, pass `--network-plugin=cni` to the Kubelet when launching it. If your environment is not using the default configuration directory (`/etc/cni/net.d`), pass the correct configuration directory as a value to `--cni-conf-dir`. The Kubelet looks for the CNI plugin binary at `/opt/cni/bin`, but you can specify an alternative location with `--cni-bin-dir`.

The CNI plugin provides IP address management for the Pods and builds routes for the virtual interfaces. To do this, the plugin interfaces with an IPAM plugin that is also part of the CNI specification. The IPAM plugin must also be a single executable that the CNI plugin consumes. The role of the IPAM plugin is to provide to the CNI plugin the gateway, IP subnet, and routes for the Pod.

Networking with Flannel

Flannel is one of the most straightforward network providers for Kubernetes. It operates at Layer 3 and offloads the actual packet forwarding to a backend such as VxLAN or IPSec. It assigns a large network to all hosts in the cluster and then assigns a portion of that network to each host. Routing between containers on a host happens via the usual channels, and Flannel handles routing between hosts using one of its available options.

Flannel uses etcd to store the map of what network is assigned to which host. The target can be an external deployment of etcd or the one that Kubernetes itself uses.

Flannel does not provide an implementation of the NetworkPolicy resource.

Running Flannel with Kubernetes

Flannel Pods roll out as a DaemonSet, with one Pod assigned to each host. To deploy it within Kubernetes, use the `kube-flannel.yaml` manifest from the Flannel repository on Github.

Once Flannel is running, it is not possible to change the network address space or the backend communication format without cluster downtime.

Network Type	Backend	Key features
Overlay	VxLAN	— Fast, but with no interhost encryption — Suitable for private/secure networks
Overlay	IPSec	— Encrypts traffic between hosts — Suitable when traffic traverses the Internet
Non Overlay	Host-gw	— Good performance — Cloud agnostic
Non Overlay	AWS VPC	— Good performance — Limited to Amazon's cloud

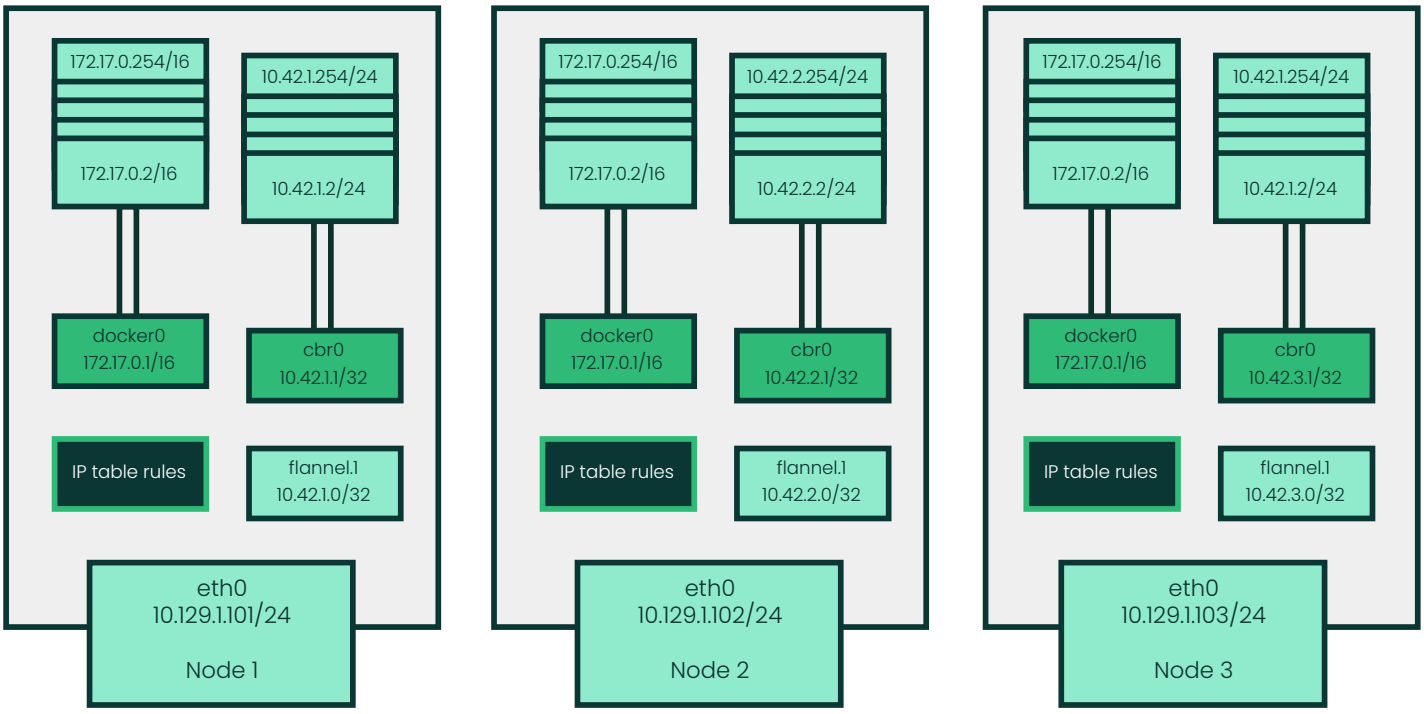
Flannel Backends

VxLAN

VxLAN is the simplest of the officially supported backends for Flannel. Encapsulation happens within the kernel, so there is no additional overhead caused by moving data between the kernel and user space.

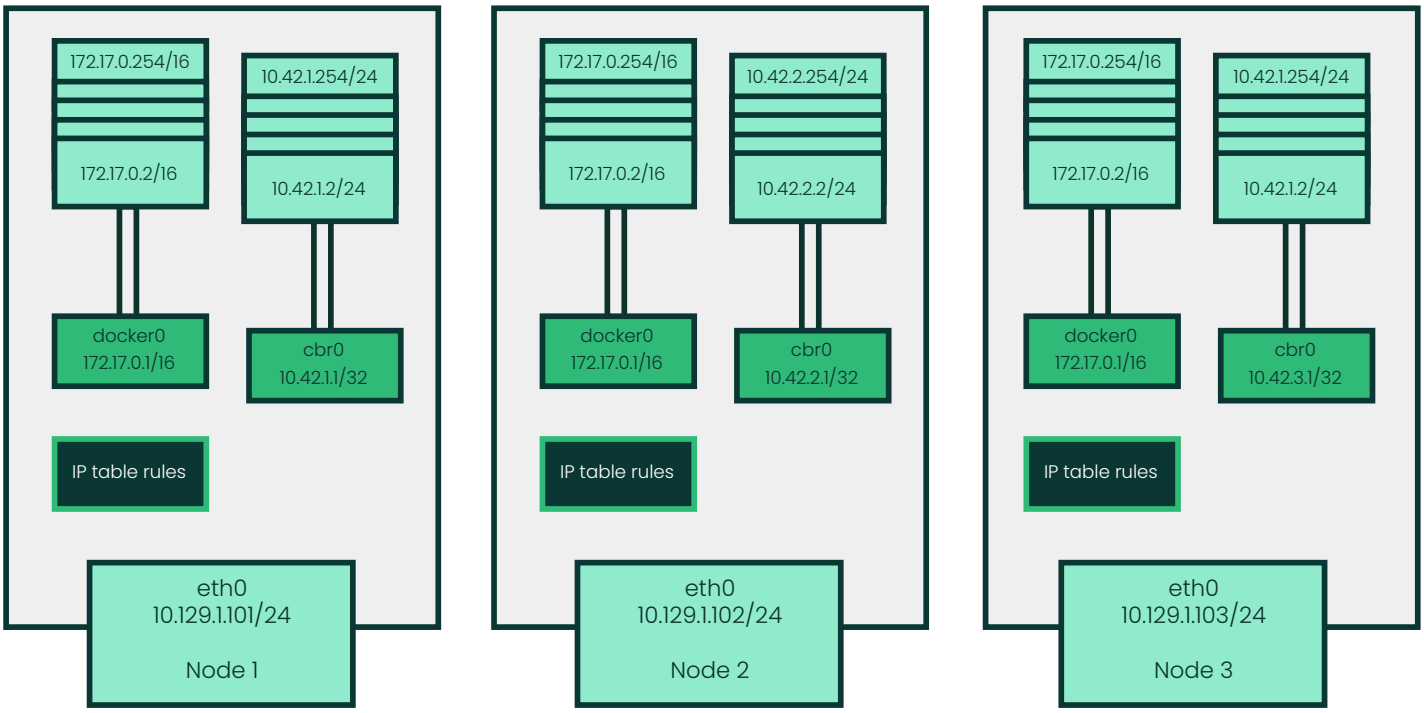
The VxLAN backend creates a Flannel interface on every host. When a container on one node wishes to send traffic to a different node, the packet goes from the container to the bridge interface in the host's network namespace. From there the bridge forwards it to the Flannel interface because the kernel route table designates that this interface is the target for the non-local portion of the overlay network. The Flannel network driver wraps the packet in a UDP packet and sends it to the target host.

Once it arrives at its destination, the process flows in reverse, with the Flannel driver on the destination host unwrapping the packet, sending it to the bridge interface, and from there the packet finds its way into the overlay network and to the destination Pod.



Host-gw

The Host-gw backend provides better performance than VxLAN but requires Layer 2 connectivity between hosts. It operates by creating IP routes to subnets via remote machine addresses.



Unlike VxLAN, no Flannel interface is created when using this backend. Instead, each node sends traffic directly to the destination node where the remote network is located.

This backend may require additional network configuration if used in a cloud provider where inter-host communication uses virtual switches.

UDP

The UDP backend is insecure and should only be used for debugging or if the kernel does not support VxLAN.

Networking with Calico

Architecture

Calico operates at Layer 3 and assigns every workload a routable IP address. It prefers to operate by using BGP without an overlay network for the highest speed and efficiency, but in scenarios where hosts cannot directly communicate with one another, it can utilize an overlay solution such as VxLAN or IP-in-IP.

Calico supports network policies for protecting workloads and nodes from malicious activity or aberrant applications.

The Calico networking Pod contains a CNI container, a container that runs an agent that tracks Pod deployments and registers addresses and routes, and a daemon that announces the IP and route information to the network via the Border Gateway Protocol (BGP). The BGP daemons build a map of the network that enables cross-host communication.

Calico requires a distributed and fault-tolerant key/value datastore, and deployments often choose etcd to deliver this component. Calico uses it to store metadata about routes, virtual interfaces, and network policy objects. The Felix agent in the `calico-node` Pod communicates with etcd to publish this information. Calico can use a dedicated HA deployment of etcd, or it can use the Kubernetes etcd datastore via the Kubernetes API. Please see the Calico deployment documentation to understand the functional restrictions that are present when using the Kubernetes API for storing Calico data.

The final piece of a Calico deployment is the controller. Although presented as a single object, it is a set of controllers that run as a control loop within Kubernetes to manage policy, workload endpoints, and node changes.

- *The Policy Controller* watches for changes in the defined network policies and translates them into Calico network policies.
- *The Namespace Controller* watches namespaces and programs Calico profiles.
- *The Serviceaccount Controller* watches service accounts and programs Calico profiles.
- Calico stores Pod information as workload endpoints. The *Workload Endpoint Controller* watches for updates to labels on the Pod and updates the workload *endpoints.*
- *The Node Controller* loop watches for the addition or removal of Kubernetes nodes and updates the kvdb with the corresponding data.

Users can manage Calico objects within the Kubernetes cluster via the command-line tool `calicoctl`. The tool's only requirement is that it can reach the Calico datastore.

Install Calico with Kubernetes

The latest instructions for installing Calico are present on the Calico Project website at https://docs.projectcalico.org. For this section, you need a Kubernetes cluster running the Calico network backend.

When the cluster is ready, deploy a Pod running Nginx:

```
 $ kubectl run nginx --image=nginx
deployment.apps "nginx" created
 $ kubectl get pods -o wide | grep nginx
nginx-64f497f8fd-2c4mh   1/1       Running   0          58s       192.168.2.245   k8s-n-4
 $
```

Note the IP address and the `eth0` interface within the Pod:

```
 $ kubectl exec -it nginx-64f497f8fd-2c4mh -- /bin/bash
root@nginx-64f497f8fd-2c4mh:/#
root@nginx-64f497f8fd-2c4mh:/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default qlen 1000
    link/ipip 0.0.0.0 brd 0.0.0.0
4: eth0@if118: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 0a:30:14:fa:20:60 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.2.245/32 scope global eth0
       valid_lft forever preferred_lft forever
root@nginx-64f497f8fd-2c4mh:/#
```

In the output below, note that the routing table indicates that a local interface (`cali106d129118f`) handles traffic for the IP address of the Pod. The `calico-node` Pod creates this interface and propagates the routes to other nodes in the cluster.

```
 $ ip route get 192.168.2.245
192.168.2.245 dev cali106d129118f src 192.168.121.196
    cache
 $
 $ ip a | grep -A 15 cali106d129118f
118: cali106d129118f@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::ecee:eeff:feee:eeee/64 scope link
       valid_lft forever preferred_lft forever
 $
```

Kubernetes scheduled our Pod to run on `k8s-n-1`. If we look at the route table on the other two nodes, we see that each directs 192.168.2.0/24 to 70.0.80.117, which is the address of `k8s-n-1`.

```
 $ hostname
k8s-n-3
 $
 $ ip route show
default via 70.0.0.1 dev eth1 proto static metric 100
70.0.0.0/16 dev eth1 proto kernel scope link src 70.0.78.228
70.0.0.0/16 dev eth1 proto kernel scope link src 70.0.78.228 metric 100
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1
192.168.0.0/24 via 70.0.78.56 dev tunl0 proto bird onlink
blackhole 192.168.1.0/24 proto bird
192.168.1.11 dev cali536e2b52742 scope link
192.168.1.18 dev cali33e889f96ba scope link
```

```
192.168.2.0/24 via 70.0.80.117 dev tunl0 proto bird onlink
192.168.3.0/24 via 70.0.78.174 dev tunl0 proto bird onlink
192.168.4.0/24 via 70.0.78.20 dev tunl0 proto bird onlink
192.168.5.0/24 via 70.0.78.110 dev tunl0 proto bird onlink
192.168.6.0/24 via 70.0.82.237 dev tunl0 proto bird onlink
192.168.7.0/24 via 70.0.78.66 dev tunl0 proto bird onlink
192.168.8.0/24 via 70.0.77.252 dev tunl0 proto bird onlink
192.168.9.0/24 via 70.0.78.171 dev tunl0 proto bird onlink
192.168.121.0/24 dev eth0 proto kernel scope link src 192.168.121.198 metric 100
 $
```

```
 $ hostname
k8s-n-2
 $
 $ ip route show
default via 70.0.0.1 dev eth1 proto static metric 100
70.0.0.0/16 dev eth1 proto kernel scope link src 70.0.78.174 metric 100
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1
192.168.0.0/24 via 70.0.78.56 dev tunl0 proto bird onlink
192.168.1.0/24 via 70.0.78.228 dev tunl0 proto bird onlink
192.168.2.0/24 via 70.0.80.117 dev tunl0 proto bird onlink
blackhole 192.168.3.0/24 proto bird
192.168.3.53 dev calicb95a28e612 scope link
192.168.4.0/24 via 70.0.78.20 dev tunl0 proto bird onlink
192.168.5.0/24 via 70.0.78.110 dev tunl0 proto bird onlink
192.168.6.0/24 via 70.0.82.237 dev tunl0 proto bird onlink
192.168.7.0/24 via 70.0.78.66 dev tunl0 proto bird onlink
192.168.8.0/24 via 70.0.77.252 dev tunl0 proto bird onlink
192.168.9.0/24 via 70.0.78.171 dev tunl0 proto bird onlink
192.168.121.0/24 dev eth0 proto kernel scope link src 192.168.121.78 metric 100
 $
```

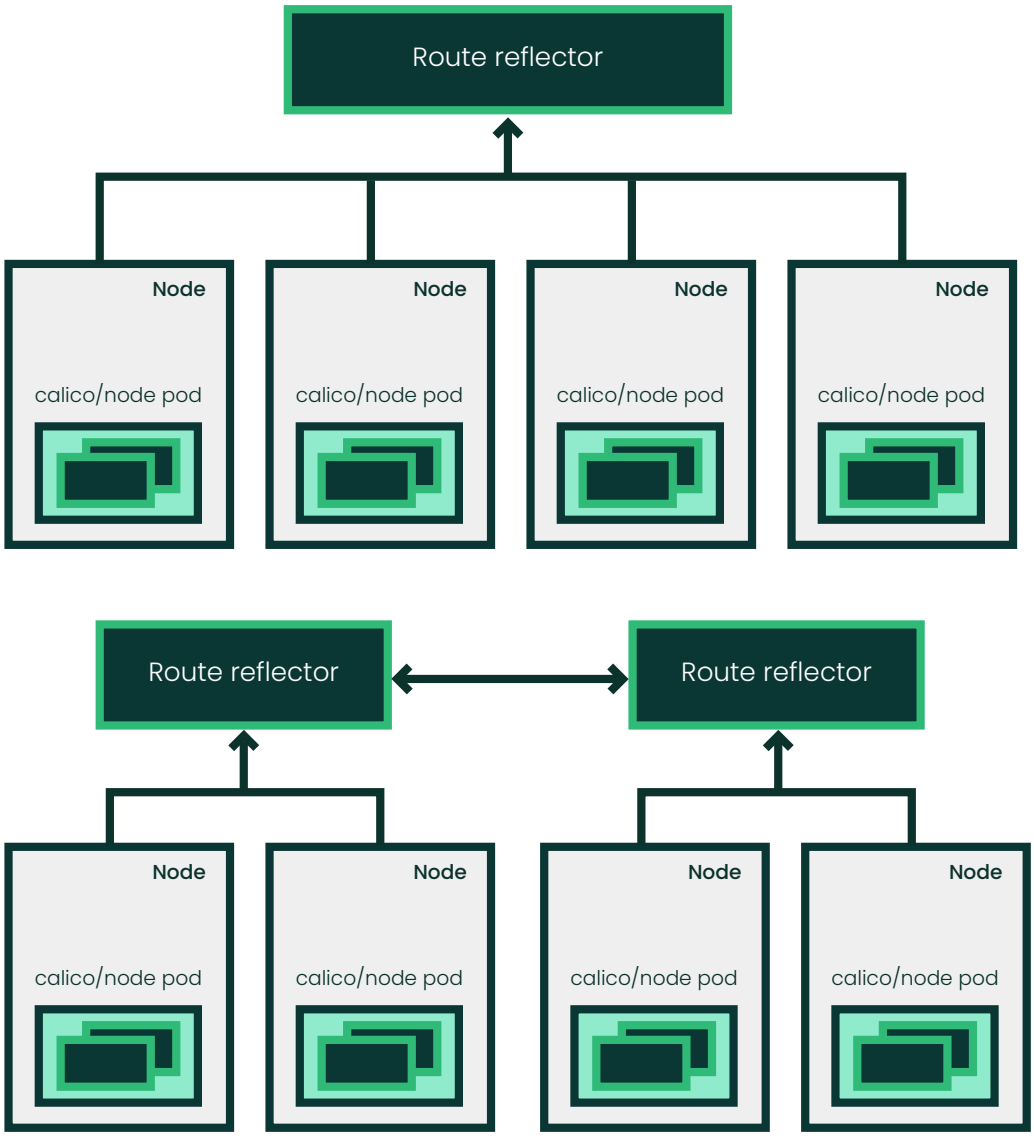
Using BGP for Route Announcements

Full Mesh Topology

Each node where Calico runs behaves as a virtual router. The `calico-node` Pod runs the Felix agent and the BIRD BGP daemon. BIRD is responsible for announcing the routes served by the host where it runs. Calico defaults to creating a full node-to-node mesh topology where each node builds a peering session with every other node in the cluster. At a small scale this works well, but as the cluster grows, we need to deploy a more efficient method for route propagation.

Using a BGP Route Reflector

We can achieve considerable improvements by utilizing a route reflector in our topology. This peer acts as a hub, and all other nodes build peering relationships with it. When a node announces a route to the reflector, it propagates this route to all other nodes with which it peers. It is not unusual to have two or more reflectors for fault tolerance or scale. Nodes connect to one or more of them to distribute the load of maintaining and announcing routes evenly across the cluster.



Two configurations of route reflectors: a single route reflector (top) and multiple route reflectors configured within a Kubernetes cluster (bottom).

Before we can use a route reflector, we first have to disable the default node-to-node BGP peering in the Calico configuration. We do this by setting `nodeToNodeMeshEnabled` to `false` in the `BGPConfiguration` resource, as demonstrated below:

```
apiVersion: projectcalico.org/v3
 kind: BGPConfiguration
 metadata:
   name: default
 spec:
   logSeverityScreen: Info
   nodeToNodeMeshEnabled: false
   asNumber: 63400
```

Next, use `calicoctl` to show the autonomous system number (ASN) for each node in the Kubernetes cluster.

```
calicoctl get nodes --output=wide
```

The `calico-node` Pods use one of two methods to build the peering relationship with external peers: global peering or per-node peering.

Global BGP Peering

If the network has a device that we want to have all of the nodes peer with, we can create a global `BGPPeer` resource within the cluster. Doing it this way assures that we only have to create the configuration once for it to be applied correctly everywhere.

```
$ calicoctl create -f - << EOF
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: bgppeer-global
  peerIP: <IP>
  scope: global
spec:
  asNumber: <ASN>
EOF
```

Use the ASN retrieved above and the IP of the external peer.

To remove a global BGP peer, use the `calicoctl` command:

```
$ calicoctl delete bgpPeer <IP> --scope=global
```

You can view the current list of BGP Peers with the following:

```
$ calicoctl get bgpPeer --scope=global
```

Per Node BGP Peering

To create a network topology where only a subset of nodes peers with certain external devices, we create a per-node `BGPPeer` resource within the cluster.

```
$ cat << EOF | calicoctl create -f -
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: bgppeer-2
  peerIP: <IP>
  Node: <NODENAME>
spec:
  asNumber: <ASN>
EOF
```

As before, use the ASN for the Calico network and the IP of the BGP peer. Specify the node to which this configuration applies.

You can remove a per-node BGP peer or view the current per-node configuration with `calicoctl:`

```
$ calicoctl delete bgpPeer <IP> --scope=node --node=<NODENAME>
$ calicoctl get bgpPeer --node=<NODENAME>
```

Using IP-in-IP

If we're unable to use BGP, perhaps because we're using a cloud provider or another environment where we have limited control over the network or no permission to peer with other routers, Calico's IP-in-IP mode encapsulates packets before sending them to other nodes.

To enable this mode, define the `ipipMode` field on the `IPPool` resource:

```
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: project1IPPool
spec:
  cidr: 10.11.12.0/16
  ipipMode: CrossSubnet
  natOutgoing: true
```

After activating IP-in-IP, Calico wraps inter-Pod packets in a new packet with headers that indicate the source of the packet is the host with the originating Pod, and the target of the packet is the host with the destination Pod. The Linux kernel performs this encapsulation and then forwards the packet to the destination host where it is unwrapped and delivered to the destination Pod.

IP-in-IP has two modes of operation:

1. `Always:` This is the default mode if an `IPPool` resource is defined.
2. `CrossSubnet:` This only performs IP encapsulation for traffic which crosses subnet boundaries. Doing this provides a performance benefit on networks where cluster members within separate Layer 2 boundaries have routers between them because it performs encapsulation intelligently, only using it for the cross-subnet traffic.

For the `CrossSubnet` mode to work, each Calico node must use the IP address and subnet mask for the host. For more information on this, see the Calico documentation for IP-in-IP.

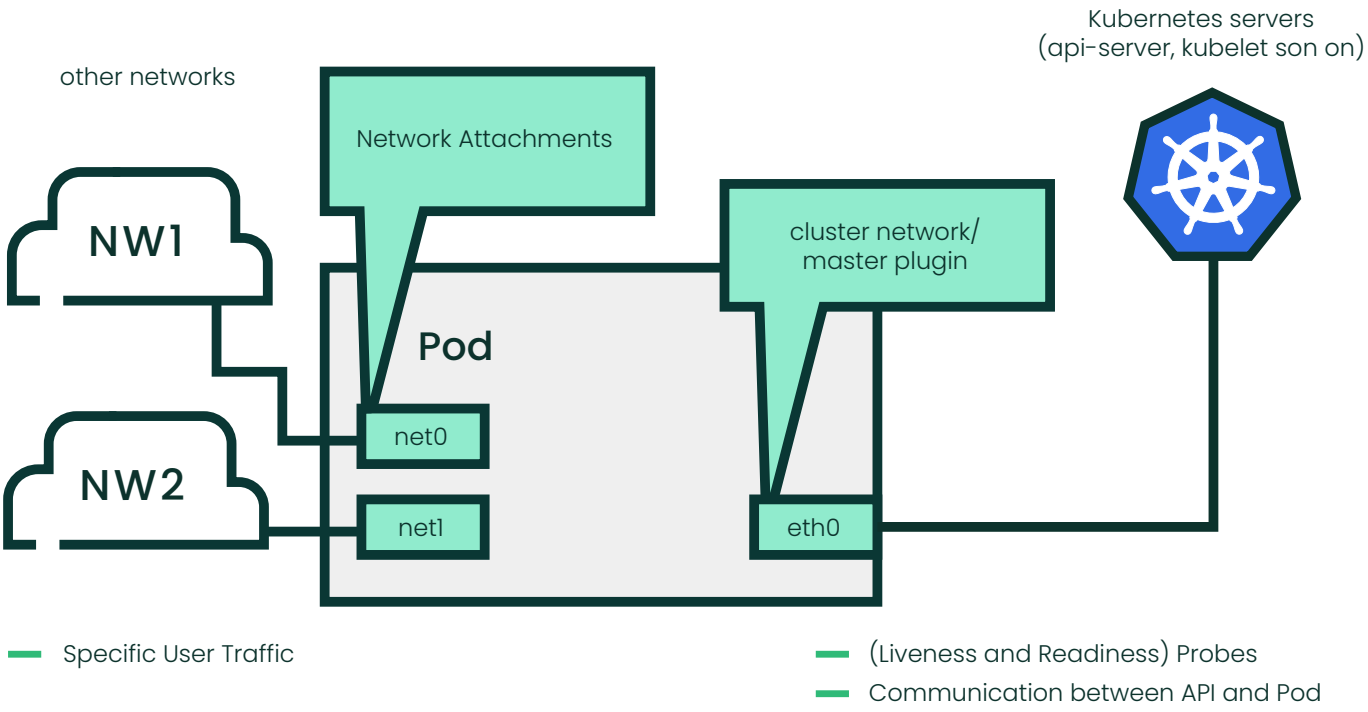
Networking with Multus CNI

Architecture

Multus CNI is a container network interface (CNI) plugin for Kubernetes that enables users to attach multiple network interfaces to pods. A typical Kubernetes pod has only one network interface, apart from a loopback. Multus can create a multi-homed pod with multiple interfaces. To accomplish this, Multus acts as a "meta-plugin", a CNI plugin that can call multiple other CNI plugins.

Multus supports the multi-networking feature in Kubernetes using Custom Resources Definition (CRD)-based network objects to extend the Kubernetes application programming interface (API). This function is important because multiple interfaces are employed by network functions to separate control, management and data/user network planes. Interfaces are also used to support different protocols, software stacks, tuning, and configuration requirements. Multus enables pods not only have multiple network interface connections, but also use advanced networking functions—including port mirroring and bandwidth capping—attached to those interfaces.

The illustration below shows network interfaces attached to a pod with three interfaces, as provisioned by Multus CNI: eth0, net0 and net1. eth0 connects Kubernetes cluster network to connect with Kubernetes server/services (e.g. Kubernetes api-server, kubelet and so on). net0 and net1 are additional network attachments and connect to other networks by using other CNI plugins (e.g. vlan/vxlan/ptp).



A pod with three interfaces, as provisioned by Multus CNI.

SUSE

Install Multus with Kubernetes

The latest instructions for installing Multus CNI are present on the project website at https://github.com/k8snetworkplumbingwg/multus-cni. Existing users of Multus who need more detail can refer to the comprehensive usage guide.

For a quickstart with Multus, you need to have configured a default network—that is, a CNI plugin that's used for your pod-to-pod connectivity—and a Kubernetes CNI plugin to serve as your pod-to-pod network. The recommended method is to deploy Multus using a Daemonset that spins up pods which install a Multus binary and configure Multus for usage.

First, clone the GitHub repository:

```
git clone https://github.com/k8snetworkplumbingwg/multus-cni.git && cd multus-cni
```

Then, apply a YAML file with `kubectl` from this repo:

```
$ cat ./deployments/multus-daemonset-thick-plugin.yml | kubectl apply -f -
```

The Multus daemonset:

- Starts a Multus daemonset that places a Multus binary on each node in `/opt/cni/bin`
- Reads the first alphabetical configuration file in `/etc/cni/net.d`, and auto-generates a new configuration file for Multus as `/etc/cni/net.d/00-multus.conf`
- Creates a `/etc/cni/net.d/multus.d` directory on each node with authentication information for Multus to access the Kubernetes API.

Validate your installation

Ensure that the Multus pods ran without error. You can gain an overview by looking at:

```
$ kubectl get pods --all-namespaces | grep -i multus
```

It's possible to further validate by looking at the /`etc/cni/net.d/` directory and ensuring that the auto-generated /`etc/cni/net.d/00-multus.conf` corresponds to the first configuration file.

Create additional interfaces

You can create configurations for each of the additional interfaces that attach to pods by creating Custom Resources. Part of the quickstart installation creates a custom resource definition (CRD) to store configurations for each interface.

CNI Configurations

CNI configurations are JSON, with a structure that has several key features:

1. `cniVersion:` Defines the version used for each CNI plugin.
2. `type:` Commands CNI which binary to call on disk. Typically, these binaries are stored in `/opt/cni/bin` on each node, and CNI executes this binary. This example specifies the `loopback` binary (which create a loopback-type network interface). If this is your first time installing Multus, verify that the plugins that are in the "type" field are actually on disk in the `/opt/cni/bin` directory.
3. `additional:` This field is an example. Each CNI plugin can specify a JSON configuration parameter, specific to the binary being called in the `type` field.

Consider the example CNI configuration:

```
{
  "cniVersion": "0.3.0",
  "type": "loopback",
  "additional": "information"
}
```

It is not necessary to reload or refresh the Kubelets when CNI configurations change because they are read on each creation / deletion of pods. When a configuration changes, it will apply the next time a pod is created. It may be necessary to restart existing pods that need the new configuration.

Store a Configuration as a Custom Resource

To create an additional interface, consider the creation of a macvlan interface for pods to use. Start by creating a custom resource that defines the CNI configuration for interfaces. Note in the following command that there's a kind: `NetworkAttachmentDefinition`. This is the name of the configuration—a custom extension of Kubernetes that defines how to attach networks to pods. The `config` field is a CNI configuration as explained earlier.

Give the configuration a name using the `name` field under `metadata` —this is also how to tell pods to use this configuration. The name in this example is `macvlan-conf`—because the demonstration creates a configuration for macvlan.

Use the following command to create this example configuration:

```
cat <<EOF | kubectl create -f -
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: macvlan-conf
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "macvlan",
      "master": "eth0",
      "mode": "bridge",
      "ipam": {
        "type": "host-local",
        "subnet": "192.168.1.0/24",
        "rangeStart": "192.168.1.200",
        "rangeEnd": "192.168.1.216",
        "routes": [
          { "dst": "0.0.0.0/0" }
        ],
        "gateway": "192.168.1.1"
      }
    }'
EOF
```

This example uses `eth0` as the master parameter. The master parameter should match the interface name on the host's cluster. Use `kubectl` to see the new configurations via:

```
kubectl get network-attachment-definitions
```

Or, get more detail by describing them:

```
kubectl describe network-attachment-definitions macvlan-conf
```

Create a Pod that Attaches an Additional Interface

Creating a pod will look familiar to any user who has created a pod previously but it will have a special `annotations` field. In this instance, you can add an annotation called `k8s.v1.cni.cncf.io/networks`. This field takes a comma delimited list of the names of your `NetworkAttachmentDefinitions` created above. The command below has the annotation of `k8s.v1.cni.cncf.io/networks: macvlan-conf` where `macvlan-conf` is the name used previously to create the configuration. As an example of a pod that sleeps for a long time, enter the command:

```
cat <<EOF | kubectl create -f -
apiVersion: v1
kind: Pod
metadata:
  name: samplepod
  annotations:
    k8s.v1.cni.cncf.io/networks: macvlan-conf
spec:
  containers:
  - name: samplepod
    command: ["/bin/ash", "-c", "trap : TERM INT; sleep infinity & wait"]
    image: alpine
EOF
```

The following command reveals the interfaces are attached to the pod:

```
$ kubectl exec -it samplepod -- ip a
```

Note the 3 interfaces:

- `lo` a loopback interface
- `eth0` our default network
- `net1` the new interface we created with the macvlan configuration.

Network Status Annotations

For additional confirmation, use `kubectl describe pod samplepod` to review the annotations section, which should display information such as:

```
Annotations:    k8s.v1.cni.cncf.io/networks: macvlan-conf
                k8s.v1.cni.cncf.io/network-status:

                [{
                     "name": "cbr0",
                     "ips": [
                     "10.244.1.73"
                ],
                     "default": true,
                     "dns": {}
                },{
                     "name": "macvlan-conf",
                     "interface": "net1",
                     "ips": [
                     "192.168.1.205"
                ],
                     "mac": "86:1d:96:ff:55:0d",
                     "dns": {}
}]
```

This metadata indicates that there are two CNI plugins running successfully.

It is possible to add more interfaces to a pod by creating more custom resources, then referring to them in pod's annotation. It is also feasible to reuse configurations. To attach two macvlan interfaces to a pod, create a pod like so:

```
cat <<EOF | kubectl create -f -apiVersion: v1kind: Podmetadata:  name: samplepod
annotations:    k8s.v1.cni.cncf.io/networks: macvlan-conf,macvlan-confspec:  containers:
- name: samplepod    command: ["/bin/ash", "-c", "trap : TERM INT; sleep infinity & wait"]
image: alpineEOF
```

The annotation now reads `k8s.v1.cni.cncf.io/networks: macvlan-conf,macvlan-conf`. The same configuration is used twice, separated by a comma. In the event another custom resource is created with the name `foo`, the annotation would read `k8s.v1.cni.cncf.io/networks: foo,macvlan-conf`, expandable to any number of attachments.

Additional Installation Options

As an alternative to installing via daemonset using the quick-start guide, users can opt to:

- Download binaries from release page
- Install by Docker image from Docker Hub
- Roll-your-own and build from source (see Development)

SUSE

Networking with Cilium

Architecture

Cilium is designed to secure network connectivity transparently between application services deployed using Linux container management platforms such as Docker and Kubernetes. At the foundation of Cilium is the Linux kernel technology eBPF.

eBPF can run sandboxed programs in an operating system kernel, enabling users to safely and efficiently extend the capabilities of the kernel without changing kernel source code, loading kernel modules, or changing container configuration. This enables dynamic insertion of powerful security visibility and control logic within Linux.

By leveraging Linux eBPF, Cilium can insert security visibility and enforcement based on service / pod / container identity, in contrast to IP address identification in traditional systems. It can also filter on application-layer (e.g. HTTP). As a result, Cilium decouples security from addressing, and provides stronger security isolation by operating at the HTTP-layer in addition to providing traditional Layer 3 and Layer 4 segmentation.

Cilium offers multiple functions for security and networking:

1. **Protecting and securing APIs transparently**

Cilium offers the ability to secure modern application protocols such as REST/HTTP, gRPC and Kafka. Traditional firewalls operate at Layer 3 and 4—a protocol running on a particular port is either completely trusted or blocked entirely. Cilium provides the ability to filter on individual application protocol requests such as:

- Allow all HTTP requests with method `GET` and `path /public/.*`. Deny all other requests.
- Allow `service1` to produce on Kafka topic topic1 and `service2` to consume on `topic1`. Reject all other Kafka messages.
- Require the HTTP header X-`Token: [0-9]+` to be present in all REST calls.

2. **Securing service-to-service communication based on identities**

Cilium assigns a security identity to groups of application containers that share security policies. The identity is then associated with all network packets emitted by the application containers, allowing identity validation at the receiving node. Security identity management is performed using a key-value store.

3. **Securing access to and from external services**

Label-based security is the tool of choice for cluster internal access control. To secure access to and from external services, Cilium supports traditional CIDR-based security policies for both ingress and egress.

4. **Simple Networking**

Cilium offers a simple, flat Layer 3 network with the ability to span multiple clusters connecting all application containers. IP allocation uses host scope allocators, so that each host can allocate IPs without any coordination between hosts. Cilium supports the following multi node networking models:

- **Overlay:** Cilium offers encapsulation-based virtual network spanning all hosts. VXLAN and Geneve are baked in but users can enable all encapsulation formats supported by Linux. This mode has minimal infrastructure

and integration requirements: it works on almost any network infrastructure because the only requirement is IP connectivity between hosts.

- **Native Routing:** Cilium enables use of the regular routing table of the Linux host. The network must be capable to route the IP addresses of the application containers. This mode is for advanced users and requires awareness of the underlying networking infrastructure. It works especially well with:
  - Native IPv6 networks
  - In conjunction with cloud network routers
  - If you are already running routing daemons

5. **Load Balancing**

Cilium implements distributed load balancing for traffic between application containers and to external services. It can fully replace components such as kube-proxy. The load balancing is implemented in eBPF using efficient hashtables, which enables almost unlimited scale.

For north-south type load balancing, Cilium's eBPF implementation is optimized for maximum performance. It can be attached to XDP (eXpress Data Path) and it supports direct server return (DSR), as well as Maglev consistent hashing. For east-west type load balancing, Cilium performs efficient service-to-backend translation in the Linux kernel's socket layer (e.g. at TCP connect time) to avoid per-packet NAT operations overhead in lower layers.

6. **Bandwidth Management**

Cilium implements bandwidth management through efficient EDT-based (Earliest Departure Time) rate-limiting with eBPF for container traffic as it leaves a node. Compared to traditional approaches such as HTB (Hierarchy Token Bucket) or TBF (Token Bucket Filter) as used in the bandwidth CNI plugin, Cilium can reduce transmission tail latencies for applications and helps avoid locking under multi-queue NICs.

7. **Monitoring and Troubleshooting**

Operating any distributed system requires the ability to gain visibility and to troubleshoot issues. Cilium aims to outperform tools such as `tcpdump` and `ping` with tooling to provide:

- Event monitoring with metadata: When a packet is dropped, the tool reports the source and destination IP of the packet, as well as the full label information of both the sender and receiver.
- Policy decision tracing, to help understand why a packet is being dropped or a request rejected.
- Metrics export via Prometheus for integration with dashboards.
- Hubble, an observability platform written for Cilium, offers service dependency maps, operational monitoring and alerting, and flow log-based application and security visibility.

Requirements

Most modern Linux distributions meet the minimum requirements for Cilium:

1. Running Cilium using the container image `cilium/cilium` requires the host system to meet these requirements:
   - Linux kernel >= 4.9.17
2. Running Cilium as a native process on your host (i.e. **not** running the `cilium/cilium` container image) entails these additional requirements:
   - clang+LLVM >= 10.0
   - iproute2 with eBPF templating patches

3. Running Cilium without Kubernetes entails these additional requirements:
   - Key-Value store etcd >= 3.1.0 or consul >= 0.6.4

For more information about system requirements, visit: https://docs.cilium.io/en/v1.9/operations/system_requirements/.

Install Cilium with Kubernetes

Cilium offers a wide range of installation options. In this exercise we will walk through installation of Cilium on Rancher Kubernetes Engine (RKE). RKE is a CNCF-certified Kubernetes distribution that runs entirely within Docker containers. It removes most host dependencies and presents a stable path for deployment, upgrades, and rollbacks, to help reduce common frustrations of installation complexity.

Other options for installation include:

- Creating a Sandbox environment
- Self-Managed Kubernetes
- Managed Kubernetes
- Installer Integrations

Installation using Rancher Kubernetes Engine

The latest instructions for installing Cilium on RKU are present on the project website at https://docs.cilium.io/en/v1.9/gettingstarted/k8s-install-rke/.

As a first step, install a cluster based on the RKE Installation Guide. When creating the cluster, make sure to change the default network plugin in the config.yaml file by changing this:

```
Network:
  options:
    flannel_backend_type:“vxlan”
  plugin: “canal”
```

To this:

```
network:
  plugin: none
```

Then, install Cilium via the provided `quick-install.yaml`. (Note that `quick-install.yaml` is a pre-rendered Cilium chart template. The template is generated using helm template command with default configuration parameters without any customization.)

```
kubectl apply -f
https://raw.githubusercontent.com/cilium/cilium/v1.9/install/kubernetes/quick-install.yaml
```

Restart Unmanaged Pods

To ensure that all pods which have been running before Cilium was deployed have network connectivity provided by Cilium and NetworkPolicy applies to them, restart all pods that are not running in host-networking mode:

```
kubectl get pods --all-namespaces -o custom-columns=NAMESPACE:.metadata.namespace,NAME:.
metadata.name,HOSTNETWORK:.spec.hostNetwork --no-headers=true | grep '<none>' | awk
'{print "-n "$1" "$2}' | xargs -L 1 -r kubectl delete pod
pod "event-exporter-v0.2.3-f9c896d75-cbvcz" deleted
pod "fluentd-gcp-scaler-69d79984cb-nfwwk" deleted
pod "heapster-v1.6.0-beta.1-56d5d5d87f-qw8pv" deleted
pod "kube-dns-5f8689dbc9-2nzft" deleted
pod "kube-dns-5f8689dbc9-j7x5f" deleted
pod "kube-dns-autoscaler-76fcd5f658-22r72" deleted
pod "kube-state-metrics-7d9774bbd5-n6m5k" deleted
pod "l7-default-backend-6f8697844f-d2rq2" deleted
pod "metrics-server-v0.3.1-54699c9cc8-7l5w2" deleted
```

Validate the Installation

You can monitor as Cilium and all required components are being installed:

```
kubectl -n kube-system get pods --watch
NAME                               READY   STATUS              RESTARTS   AGE
cilium-operator-cb4578bc5-q52qk    0/1     Pending             0          8s
cilium-s8w5m                       0/1     PodInitializing     0          7s
coredns-86c58d9df4-4g7dd           0/1     ContainerCreating   0          8m57s
coredns-86c58d9df4-4l6b2           0/1     ContainerCreating   0          8m57s
```

It may take a couple of minutes for all components to come up:

```
cilium-operator-cb4578bc5-q52qk         1/1     Running   0          4m13s
cilium-s8w5m                            1/1     Running   0          4m12s
coredns-86c58d9df4-4g7dd                1/1     Running   0          13m
coredns-86c58d9df4-4l6b2                1/1     Running   0          13m
```

Deploy the Connectivity Test

To check connectivity between pods, you can deploy a connectivity check. Create a separate namespace for this, such as `kubectl create ns cilium-test`.

Then, deploy the check with:

```
kubectl apply -n cilium-test -f https://raw.githubusercontent.com/cilium/cilium/v1.9/
examples/kubernetes/connectivity-check/connectivity-check.yaml
```

This test implements a series of deployments using various connectivity paths to connect. Connectivity paths include with / without service load-balancing, as well as and various network policy combinations. The pod name indicates the connectivity variant and the readiness; the liveness gate indicates success or failure of the test:

```
kubectl get pods -n cilium-test

NAME                                                 READY   STATUS    RESTARTS   AGE
echo-a-76c5d9bd76-q8d99                              1/1     Running   0          66s
echo-b-795c4b4f76-9wrrx                              1/1     Running   0          66s
echo-b-host-6b7fc94b7c-xtsff                         1/1     Running   0          66s
host-to-b-multi-node-clusterip-85476cd779-bpg4b      1/1     Running   0          66s
host-to-b-multi-node-headless-dc6c44cb5-8jdz8        1/1     Running   0          65s
pod-to-a-79546bc469-rl2qq                            1/1     Running   0          66s
pod-to-a-allowed-cnp-58b7f7fb8f-lkq7p                1/1     Running   0          66s
pod-to-a-denied-cnp-6967cb6f7f-7h9fn                 1/1     Running   0          66s
pod-to-b-intra-node-nodeport-9b487cf89-6ptrt         1/1     Running   0          65s
pod-to-b-multi-node-clusterip-7db5dfdcf7-jkjpw       1/1     Running   0          66s
pod-to-b-multi-node-headless-7d44b85d69-mtscc        1/1     Running   0          66s
pod-to-b-multi-node-nodeport-7ffc76db7c-rrw82        1/1     Running   0          65s
pod-to-external-1111-d56f47579-d79dz                 1/1     Running   0          66s
pod-to-external-fqdn-allow-google-cnp-78986f4bcf-btjn7 1/1     Running   0          66s
```

Enable Hubble for Cluster-Wide Visibility

Use Hubble, the component for observability in Cilium, to gain cluster-wide visibility into your network traffic. The example below shows the process for `quick-hubble-install.yaml`. Installation via Helm is also possible. If you installed Cilium 1.9.2 or newer via the provided `quick-install.yaml`, you can deploy Hubble Relay and UI on top of your existing installation with the following command:

```
kubectl apply -f
https://raw.githubusercontent.com/cilium/cilium/v1.9/install/kubernetes/quick-hubble-
install.yaml
```

Installation `via quick-hubble-install.yaml` works if the installed Cilium version is 1.9.2 or newer. Any users of Cilium 1.9.0 or 1.9.1 should upgrade to a newer version by applying the most recent Cilium `quick-install.yaml` first. As an alternate method, it is possible to manually generate a YAML manifest for the Cilium DaemonSet and Hubble Relay/UI. The generated YAML can be applied on top of an existing installation:

```
# Set this to your installed Cilium version
export CILIUM_VERSION=1.9.1
# Please set any custom Helm values you may need for Cilium,
# such as for example `--set operator.replicas=1` on single-cluster nodes.
helm template cilium cilium/cilium --version $CILIUM_VERSION \\
   --namespace $CILIUM_NAMESPACE \\
   --set hubble.tls.auto.method="cronJob" \\
   --set hubble.listenAddress=":4244" \\
   --set hubble.relay.enabled=true \\
   --set hubble.ui.enabled=true > cilium-with-hubble.yaml
# This will modify your existing Cilium DaemonSet and ConfigMap
kubectl apply -f cilium-with-hubble.yaml
```

The Cilium agent pods will be restarted in the process.

Once the Hubble UI pod is started, use port forwarding for the `hubble-ui` service. This allows opening the UI locally on a browser:

```
kubectl port-forward -n $CILIUM_NAMESPACE svc/hubble-ui --address 0.0.0.0 --address ::
12000:80
```

Then, open http://localhost:12000/ to access the UI.

Hubble UI is not the only way to get access to Hubble data. A command line tool, the Hubble CLI, is also available for installation for Linux, MacOS, and Windows users. Additional methods to implement Hubble are available in the installation documentation for RKE.

Networking with Kube-Vip

Architecture

Kube-Vip offers Kubernetes load balancing with a lightweight and multi-architecture, providing both high availability (HA) networking endpoints and additional functionality for underlying network services. It's suitable for those seeking a decoupled centralized `type: LoadBalancer` solution with a focus on high availability for a Kubernetes cluster. With a multi-architecture design, all of the components are built for Linux but are also built for both `x86` and `armv7, armhvf, ppc64le`. This means that `kube-vip` will run well in bare-metal, virtual and edge (raspberry pi or small arm SoC devices).

Kube-Vip offers two main technologies to provide high-availability and networking functions as part of a VIP/Load-balancing solution.

1. Cluster

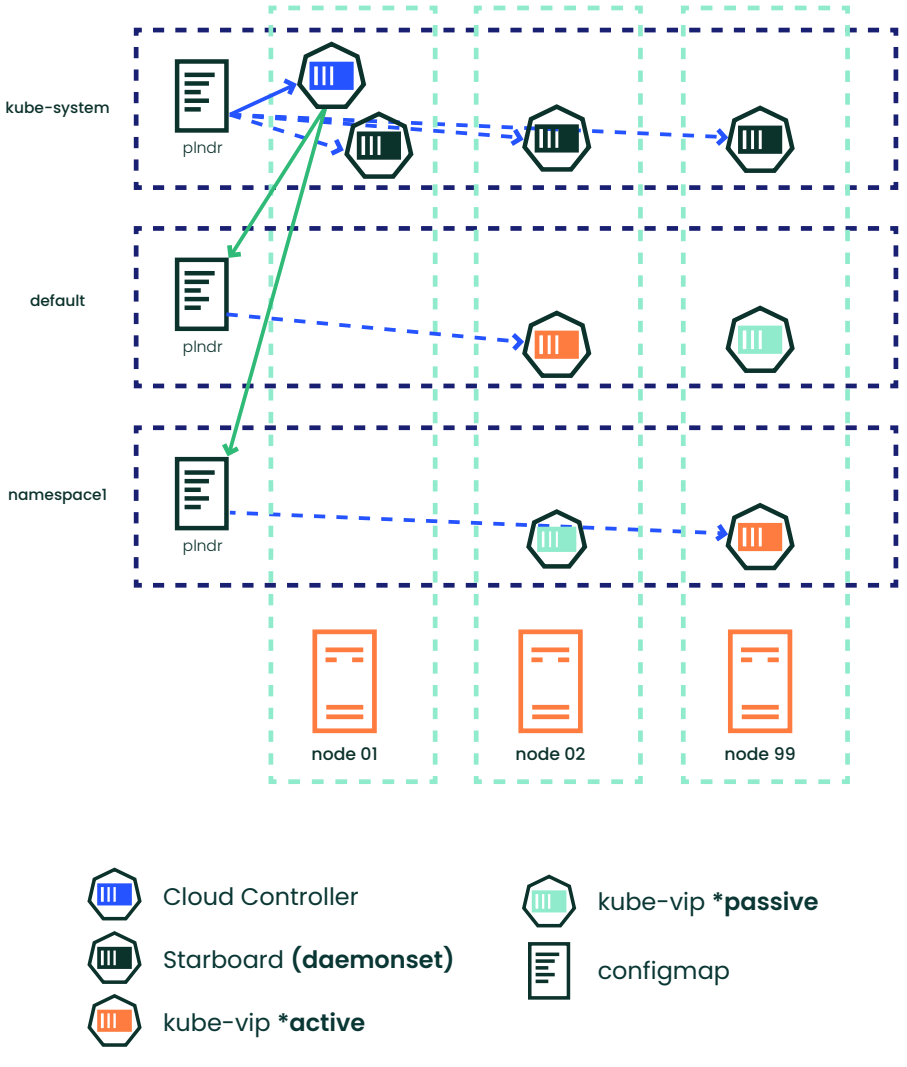
The `kube-vip` service builds a multi-node or multi-pod cluster to provide HA. In ARP mode, a leader is elected, which inherits the Virtual IP and becomes the leader of the load-balancing within the cluster. With BGP, all nodes will advertise the VIP address. When using ARP or layer2, Kube-Vip uses leader election.

2. Virtual IP

The leader in the cluster assumes the vip and has it bound to the selected interface declared in the configuration. The vip is evacuated first when the leader changes; in failure scenarios the vip is directly assumed by the next elected leader.

When the vip moves from one host to another, any host that has been using the vip will retain the previous `vip <-> MAC address` mapping until the ARP expires the old entry and retrieves a new `vip <-> MAC` mapping.

As shown in the diagram below, `kube-vip` requires Kube-Vip Deployment as well as the Plunder Cloud Provider to function. Plunder Cloud Provider uses the Kubernetes cloud-provider SDK to provide the same cloud-like services one would expect from leading cloud platforms. When a user requests functionality, the cloud provider communicates to the underlying vendor and provisions the required service. `Plunder cloud Provider` is currently designed to intercept the creation of LoadBalancers and translate that into a `kube-vip` load balancer.



Architecture of the `kube-vip` kubernetes load-balancer, including its Plunder Cloud Provider and Kube-Vip Deployment components.

Users can expect easy manifest deployment, support for management via BGP or ARP (Address resolution protocol) functionality, with support from core Equinix Metal integration (such as CCM, Packet API).

While Kube-Vip was originally created to provide a HA solution for the Kubernetes control plane, it has evolved to incorporate that same functionality into Kubernetes service type load-balancers. VIP addresses can be both IPv4 or IPv6. The Control Plane features ARP (Layer 2) or BGP (Layer 3), using either leader election or raft, with HA facilitated by kubeadm (static Pods) or K3s/and others (daemonsets).

The Service LoadBalancer uses leader election for ARP (Layer 2), and multiple nodes with BGP. Users can address pools per namespace or global, address via an existing network DHCP, or exposure to gateway via UPNP.

Install Kube-Vip with Kubernetes

The latest instructios for installing Kube-Vip are on the project website, https://kube-vip.io/. In Hybrid mode, `kube-vip` manages a virtual IP address that is passed through its configuration for a HA Kubernetes cluster. It also monitors services of `type:LoadBalancer;` once their `spec.LoadBalancerIP` is updated, most typically by a cloud controller, it will advertise this address using BGP/ARP.

Note that the "hybrid" mode is now the default mode in `kube-vip` from `0.2.3` onwards. It allows both modes to be enabled at the same time.

Create the RBAC Settings

The daemonSet runs within the Kubernetes cluster, and it needs the correct access to watch Kubernetes services and other objects. To facilitate this, start by creating a User, Role, and a binding. Apply this with the command:

```
kubectl apply -f https://kube-vip.io/manifests/rbac.yaml
```

Generate a Manifest

Next, generate a simple BGP configuration by setting the configuration details as follows:

```
export VIP=192.168.0.40
export INTERFACE=<interface>
```

Configure to Use a Container Runtime

Using the container itself is the easiest method to generate a manifest. You can create an alias for different container runtimes as follows:

containerd

```
alias kube-vip="ctr run --rm --net-host ghcr.io/kube-vip/kube-vip:0.3.7 vip"
```

Docker

```
alias kube-vip="docker run --network host --rm ghcr.io/kube-vip/kube-vip:0.3.7"
```

BGP Example

This configuration creates a manifest that will start kube-vip providing controlplane and services management. Unlike ARP, all nodes in the BGP configuration advertise virtual IP addresses. It's useful to bind the address to lo to avoid interacting with multiple devices with the same address on public interfaces. Peers can be specified in a comma separated list in the format of `address:AS:password:multihop`.

```
export INTERFACE=lo
```

```
kube-vip manifest daemonset \
    --interface $INTERFACE \
    --vip $VIP \
    --controlplane \
    --services \
    --inCluster \
    --taint \
    --bgp \
    --bgppeers 192.168.0.10:65000::false,192.168.0.11:65000::false
```

Generated Manifest

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  creationTimestamp: null
  name: kube-vip-ds
  namespace: kube-system
spec:
  selector:
    matchLabels:
      name: kube-vip-ds
  template:
    metadata:
      creationTimestamp: null
      labels:
        name: kube-vip-ds
    spec:
      containers:
      - args:
        - manager
        env:
        - name: vip_arp
          value: “false”
        - name: vip_interface
          value: lo
        - name: port
          value: “6443”
```

```
        - name: vip_cidr
          value: “32”
        - name: cp_enable
          value: “true”
        - name: cp_namespace
          value: kube-system
        - name: svc_enable
          value: “true”
        - name: bgp_enable
          value: “true”
        - name: bgp_peers
          value: “192.168.0.10:65000::false,192.168.0.11:65000::false”
        - name: vip_address
192.168.0.10:65000::false,192.168.0.11:65000::false
         value: 192.168.0.40
        image: ghcr.io/kube-vip/kube-vip:0.3.7
        imagePullPolicy: Always
        name: kube-vip
        resources: {}
        securityContext:
          capabilities:
            add:
            - NET_ADMIN
            - SYS_TIME
      hostNetwork: true
      serviceAccountName: kube-vip
      nodeSelector:
        node-role.kubernetes.io/master: “true”
      tolerations:
      - effect: NoSchedule
        key: node-role.kubernetes.io/master
  updateStrategy: {}
```

Manifest Overview

- `nodeSelector` – This feature ensures that the particular daemonset runs only on control plane nodes
- `serviceAccountName: kube-vip` – This feature specifies the user in the `rbac` to provide permissions to receive/update services
- `hostNetwork:` true – This feature entails a pod that modifies interfaces (for VIPs)
- `env {...}` – This configuration is passed into the kube-vip pod through environment variables

Equinix Metal Overview (using the Equinix Metal CCM)

For users interested in running `type:LoadBalancer` services on worker nodes only, the following example creates a daemonset that will run kube-vip. This process requires installation of the Equinix Metal CCM and configuration of the cluster/kubelet to use an external cloud provider. Equinix Metal CCM applies the BGP configuration to the node annotations, making it easier for `kube-vip` to expose load balancer addresses. The `--annotations metal.equinix.com` causes kube-vip to monitor the annotations of the worker node it is running on. After the configuration has been applied by the CCM, the `kube-vip` pod is ready to advertise BGP addresses for the service.

```
kube-vip manifest daemonset \
  --interface $INTERFACE \
  --services \
  --bgp \
  --annotations metal.equinix.com \
  --inCluster | k apply -f -
```

If `kube-vip` has been waiting for a long time, confirm that the annotations have been applied correctly by running the `describe` on the node as follows:

```
kubectl describe node k8s.bgp02
...
Annotations:        kubeadm.alpha.kubernetes.io/cri-socket: /var/run/
dockershim.sock
                    node.alpha.kubernetes.io/ttl: 0
                    metal.equinix.com/node-asn: 65000
                    metal.equinix.com/peer-asn: 65530
                    metal.equinix.com/peer-ip: x.x.x.x
                    metal.equinix.com/src-ip: x.x.x.x
```

If you find errors regarding `169.254.255.1` or `169.254.255.2` in the `kube-vip` logs, it is possible that the nodes are missing the routes to the ToR switches providing BGP peering. Nodes can be replaced with the below command:

```
GATEWAY_IP=$(curl https://metadata.platformequinix.com/metadata | jq -r
“.network.addresses[] | select(.public == false) | .gateway”)
ip route add 169.254.255.1 via $GATEWAY_IP
ip route add 169.254.255.2 via $GATEWAY_IP
```

You can also examine the logs of the Packet CCM to reveal why the node is not yet ready.

K3s Overview on Equinix Metal

Step 1: Tidy Up

Run the following:

```
rm -rf /var/lib/rancher /etc/rancher ~/.kube/*; ip addr flush dev lo;
ip addr add 127.0.0.1/8 dev lo; mkdir -p
/var/lib/rancher/k3s/server/manifests/
```

Step 2: Get rbac

Run:

```
curl https://kube-vip.io/manifests/rbac.yaml >
/var/lib/rancher/k3s/server/manifests/rbac.yaml
```

Step 3: Generate kube-vip (get EIP from CLI or UI)

Run:

```
export EIP=x.x.x.x
export INTERFACE=lo
```

```
kube-vip manifest daemonset \
  --interface $INTERFACE \
  --vip $EIP \
  --controlplane \
  --services \
  --inCluster \
  --taint \
  --bgp \
  --metal \
  --provider-config /etc/cloud-sa/cloud-sa.json | tee
/var/lib/rancher/k3s/server/manifests/vip.yaml
```

Step 4: Up Cluster

Run:

```
K3S_TOKEN=SECRET k3s server --cluster-init --tls-san $EIP --no-deploy servicelb --disable-
cloud-controller
```

Step 5: Add CCM

Run:

```
alias k="k3s kubectl" k apply -f ./secret.yaml
```

```
k apply -f https://gist.githubusercontent.com/
thebsdbox/c86dd970549638105af8d96439175a59/
raw/4abf90fb7929ded3f7a201818efbb6164b7081f0/ccm.yaml
```

Step 6: Ready for Demo

Run:

```
k apply -f https://k8s.io/examples/application/deployment.yaml k expose deployment nginx-deployment --port=80 --type=LoadBalancer --name=nginx
```

Step 7: Watch and Test

Run:

```
k get svc --watch
```

Networking with MetalLB

Architecture

MetalLB offers a load-balancer implementation for bare metal Kubernetes clusters, using standard routing protocols, that integrates with standard network equipment. Users can bring a first-class balancer for bare-metal clusters, rather than relying on “NodePort” and “externalIPs” services.

To run MetalLB, users need:

- A Kubernetes cluster running Kubernetes 1.13.0 or later, without existing network load-balancing functionality.
- A cluster network configuration that compatible with MetalLB. These include Calico, Canal, Cilium, Flannel, Kube-ovn, Kube-router, and Weave Net.
- IPv4 addresses for MetalLB to allocate.
- One or more routers capable of speaking BGP, if using the BGP operating mode.
- Traffic on port 7946 (TCP & UDP) allowed between nodes.

Note that MetalLB is designed for bare-metal clusters. Generally, even cloud providers that offer “dedicated servers” will not support the network protocols that MetalLB requires.

Install MetalLB with Kubernetes

MetalLB offers three supported methods of installation: using plain Kubernetes manifests, using Kustomize, or using Helm. Start by assessing whether you are using kube-proxy in IPVS mode; Kubernetes v1.14.2 and later requires you to enable strict ARP mode. This is not necessarily if you are using kube-router as service-proxy because it enables strict ARP by default.

Edit kube-proxy config in current cluster:

```
kubectl edit configmap -n kube-system kube-proxy
```

and set:

```
apiVersion: kubeproxy.config.k8s.io/v1alpha1
kind: KubeProxyConfiguration
mode: “ipvs”
ipvs:
  strictARP: true
```

You may also add this configuration snippet to kubeadm-config, if it is appended with `---` after the main configuration.

The following shell snippets can help automate this change:

```
# see what changes would be made, returns nonzero returncode if different
kubectl get configmap kube-proxy -n kube-system -o yaml | \
sed -e "s/strictARP: false/strictARP: true/" | \
kubectl diff -f - -n kube-system

# actually apply the changes, returns nonzero returncode on errors only
kubectl get configmap kube-proxy -n kube-system -o yaml | \
sed -e "s/strictARP: false/strictARP: true/" | \
kubectl apply -f - -n kube-system
```

Installation by Manifest

To install MetalLB using manifest, apply:

```
kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.11.0/manifests/
namespace.yaml
kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.11.0/manifests/
metallb.yaml
```

This deploys MetalLB to the cluster under the `metallb-system` namespace. Manifest components include:

- The `metallb-system/controller` deployment, the cluster-wide controller handling IP address assignments
- The `metallb-system/speaker` daemonset, the component that speaks your choice of protocol(s) to make the services reachable
- Service accounts for the controller and speaker, plus RBAC permissions required by the components to function.

The installation manifest does not include a configuration file. MetalLB's components will still start, but will remain idle until you define and deploy a configmap.

Installation with Kustomize

You can install MetalLB with Kustomize by entering the following command to point at the remote kustomization file:

```
# kustomization.yml
namespace: metallb-system

resources:
  - github.com/metallb/metallb//manifests?ref=v0.11.0
  - configmap.yml
```

If using a configMapGenerator for config file, tell Kustomize not to append a hash to the config map because MetalLB is waiting for a config map named `config`.

```
# kustomization.yml
namespace: metallb-system

resources:
  - github.com/metallb/metallb//manifests?ref=v0.11.0

configMapGenerator:
- name: config
  files:
    - configs/config

generatorOptions:
 disableNameSuffixHash: true
```

Installation with Helm

To install MetalLB with Helm, use the Helm chart repository at https://metallb.github.io/metallb.

```
helm repo add metallb https://metallb.github.io/metallb
helm install metallb metallb/metallb
```

You may specify a values file on installation. This is recommended practice to provide configs in Helm values:

```
helm install metallb metallb/metallb -f values.yaml
```

MetalLB configs are set in `values.yaml` under `configInLine:`

```
configInline:
  address-pools:
   - name: default
     protocol: layer2
     addresses:
     - 198.51.100.0/24
```

Load Balancers and Ingress Controllers

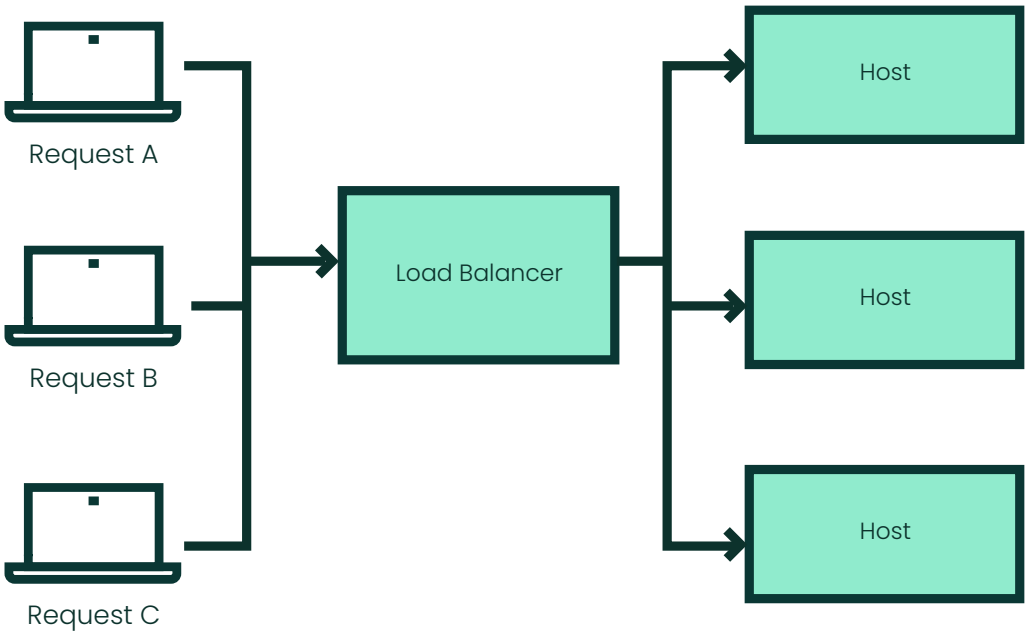
Up until now, we've focused on how to configure networking and how the various providers work in a Kubernetes cluster. While these systems define and control communication within the cluster and between its nodes, they do not, on their own, address how traffic from outside of the cluster finds its way to a destination or what part DNS plays in that process. To understand the full picture, we need to explore how Kubernetes approaches load balancing and DNS.

The Benefits of Load Balancers

A load balancer provides valuable features for any deployment, whether it's running inside or outside of the Kubernetes cluster. In addition to distributing load across multiple backends, a load balancer can also move TLS processing to a central location, route traffic based on the requester's hardware or browser, the requested site, or a path within the URL, or it can enable canary deployments and zero-downtime upgrades.

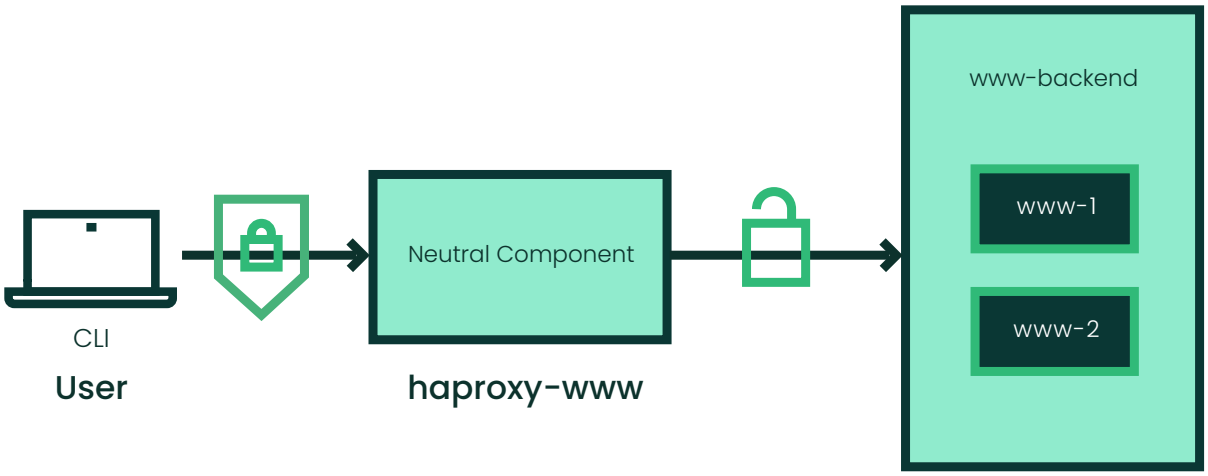
Load Distribution

When client requests arrive, the load balancer directs them across a pool of worker nodes commonly referred to as *backends*. Because the load balancer presents itself as the endpoint for the site, the clients don't know anything about these backends. The load balancer tracks the health and number of connections to each backend, and it works according to its configured policy to evenly distribute the traffic. If a backend fails or becomes overloaded, the load balancer stops sending traffic to it until it returns to a healthy state. This scenario enables *horizontal scaling*, where a site can scale capacity by adding and removing backends.



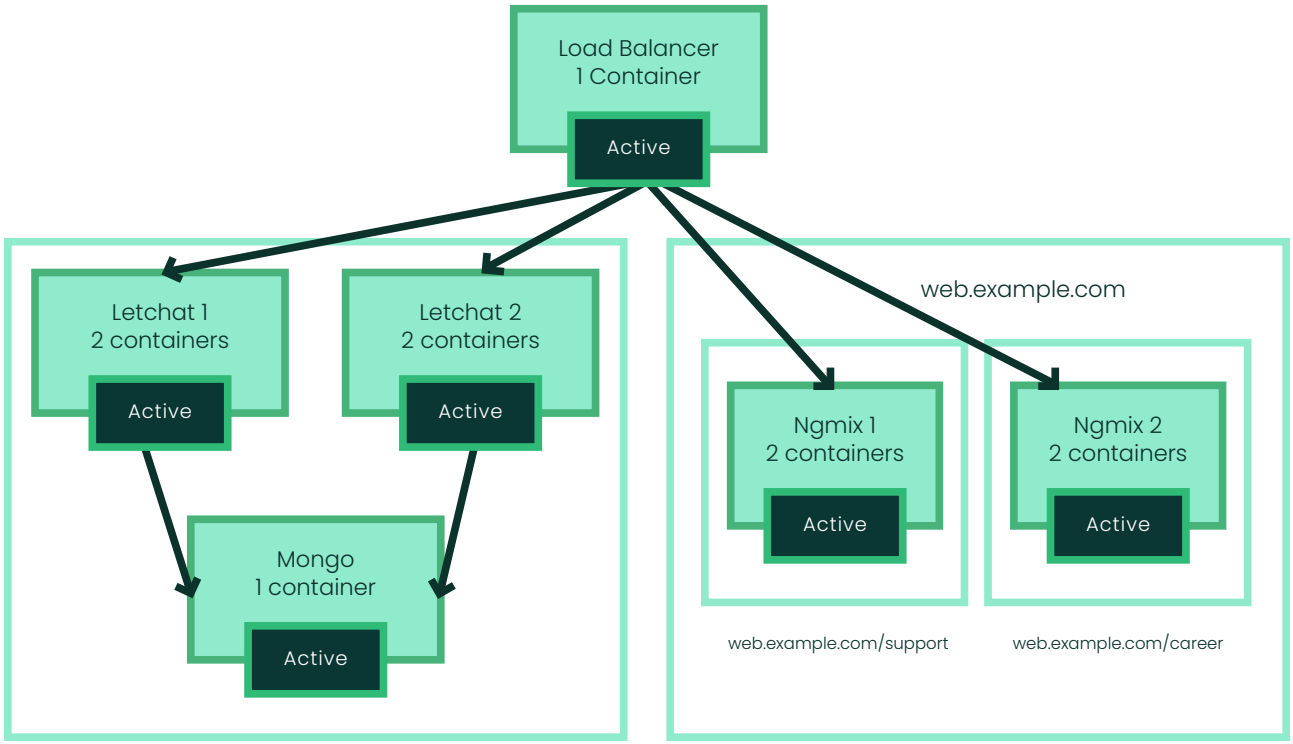
SSL/TLS Termination

The overhead of encrypting and decrypting data can impact the performance of a backend, so deployments frequently move this work to the load balancer. Encrypted traffic lands on the load balancer, which decrypts it and forwards it to a backend. By operating with a decrypted data stream, the load balancer can make informed decisions about how to route the data because it's now able to see more than the basic metadata present in the flow.



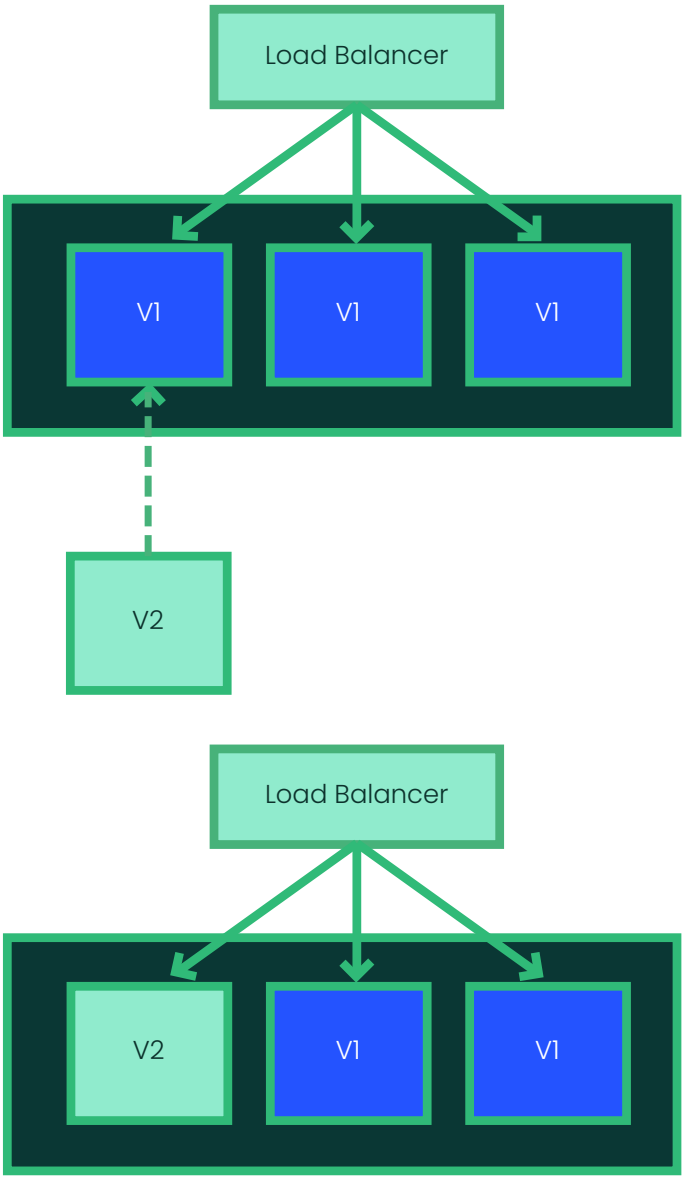
Routing By HTTP Host or Path

Organizations who run multiple applications frequently group them under the same logical namespace: their domain name. In this scenario, a load balancer routes traffic based on parameters such as the requested host or site (the Host header), or by the path requested in the URL.



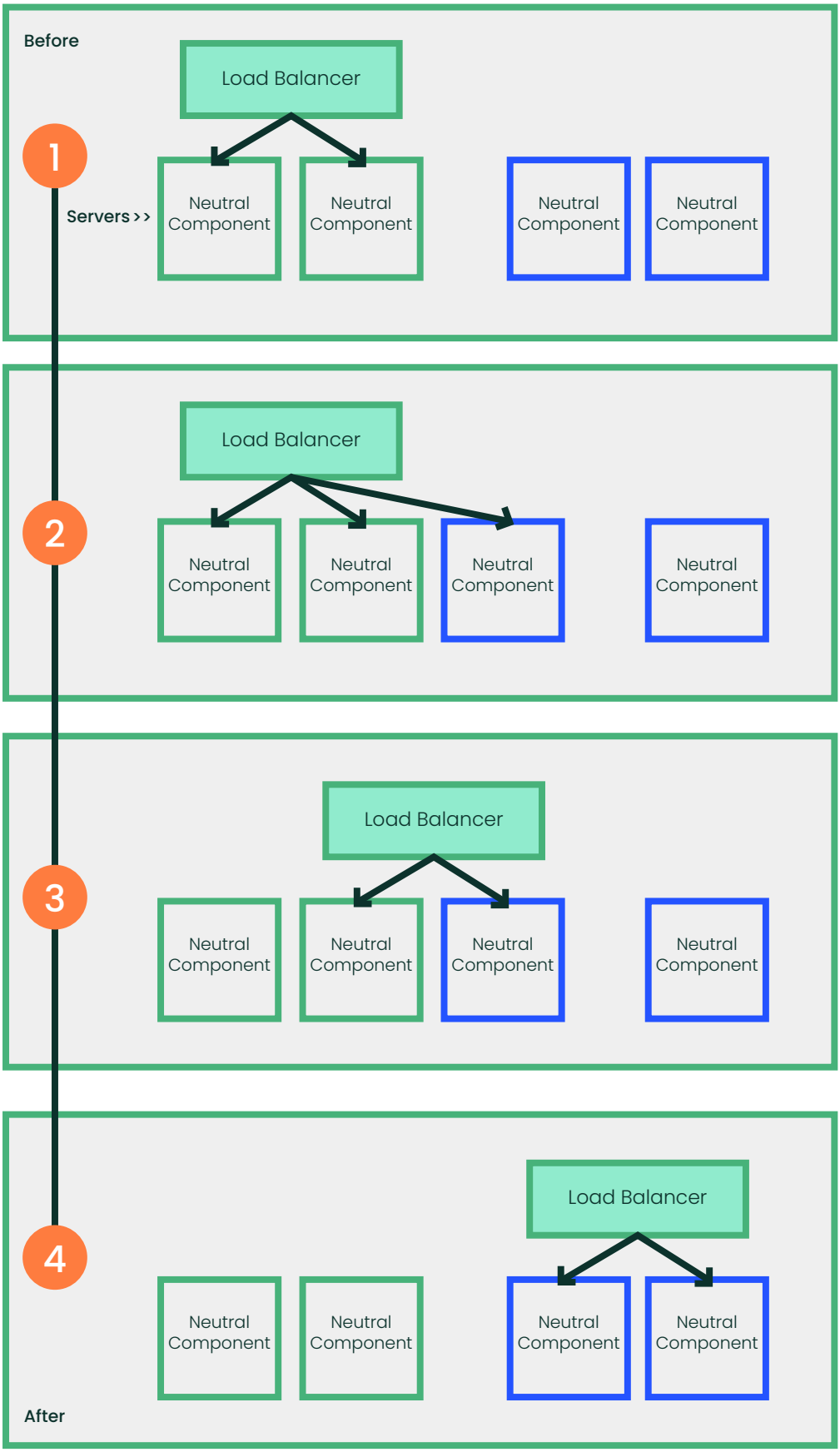
Upgrades and Feature Flags

When a load balancer receives an HTTP request, the headers contain a wealth of extra information such as the browser, the device, the operating system, and more. Site maintainers can use this information to route a subset of the traffic to a different destination, perhaps to give an optimized experience to a particular class of mobile device, to test a new feature before rolling it out everywhere, or to see the effect of different changes to the content and determine which one has the more significant impact.



Load balancers also provide a way to roll out upgrades safely. Site administrators first deploy the new version of the website or application to a new set of backends and test it outside of the standard rotation. When ready, they incrementally add the new backends to the pool and rotate the old backends out. The load balancers keep existing traffic on the old backends and direct new traffic to the new backends. Over time the sessions connected to the old backends close, and only new sessions remain. The old backends are then terminated.

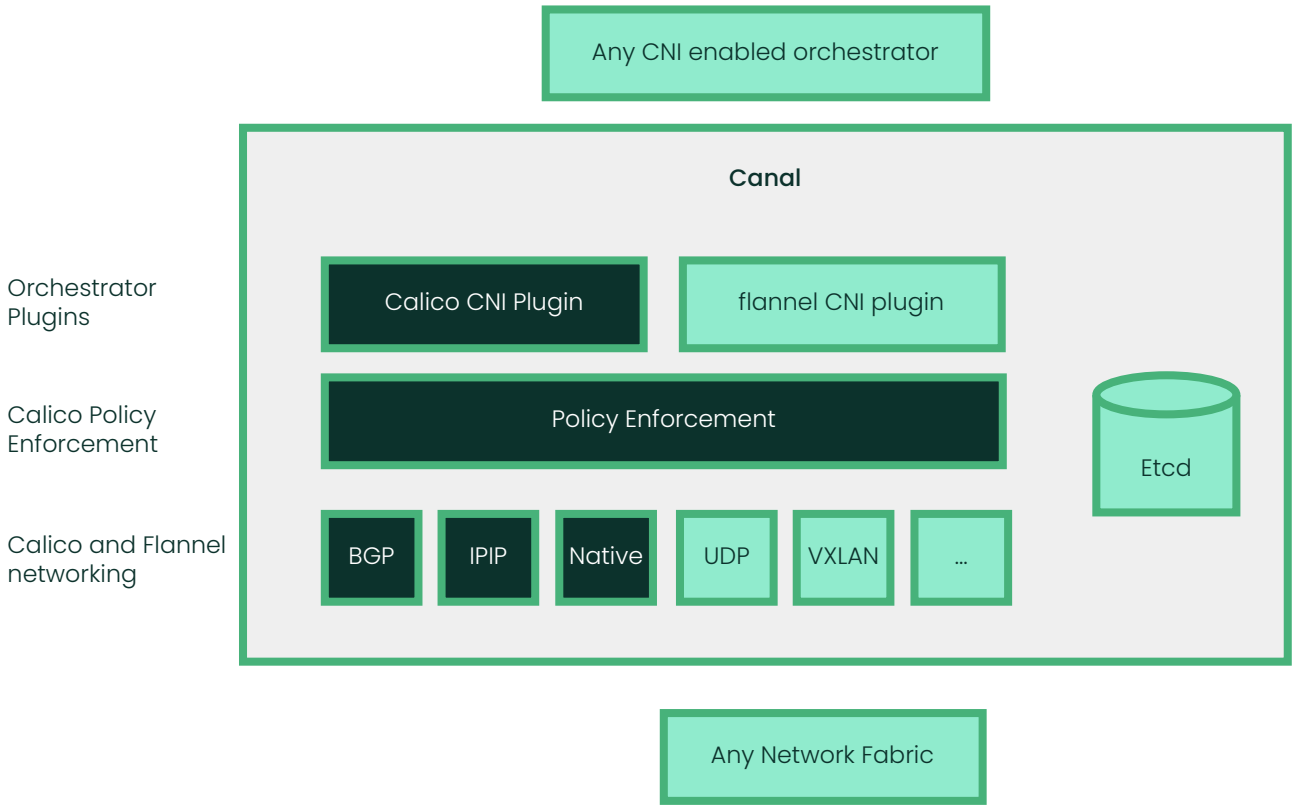
In the event of an unforeseen issue, the admins can quickly rotate the old backends into the pool and remove the new ones, returning the site to its previous, working state.


Before
Load Balancer
1
Servers >>
Neutral Component
Neutral Component
Neutral Component
Neutral Component
Load Balancer
2
Neutral Component
Neutral Component
Neutral Component
Neutral Component
Load Balancer
3
Neutral Component
Neutral Component
Neutral Component
Neutral Component
Load Balancer
4
Neutral Component
Neutral Component
Neutral Component
Neutral Component
After

Networking with Flannel and Calico (Canal)

For some time an effort to integrate Flannel's easy overlay networking engine and Calico's network policy enforcement ran under the project name Canal. The maintainers deprecated it as a separate project, and instead, the Calico documentation contains instructions on deploying Flannel and Calico together (see previous section on installing Calico on Kubernetes).

They only abandoned the name and status; the result remains the same. Flannel provides an overlay network using one of its backends, and Calico provides granular access control to the running workloads with its network policy implementation.



Load Balancing in Kubernetes

Kubernetes either can create internal load balancers using Kubernetes resources such as Services and Ingresses, or it can deploy and manage external load balancers such as those provided by AWS, GCP, F5, and others.

Internal Load Balancing

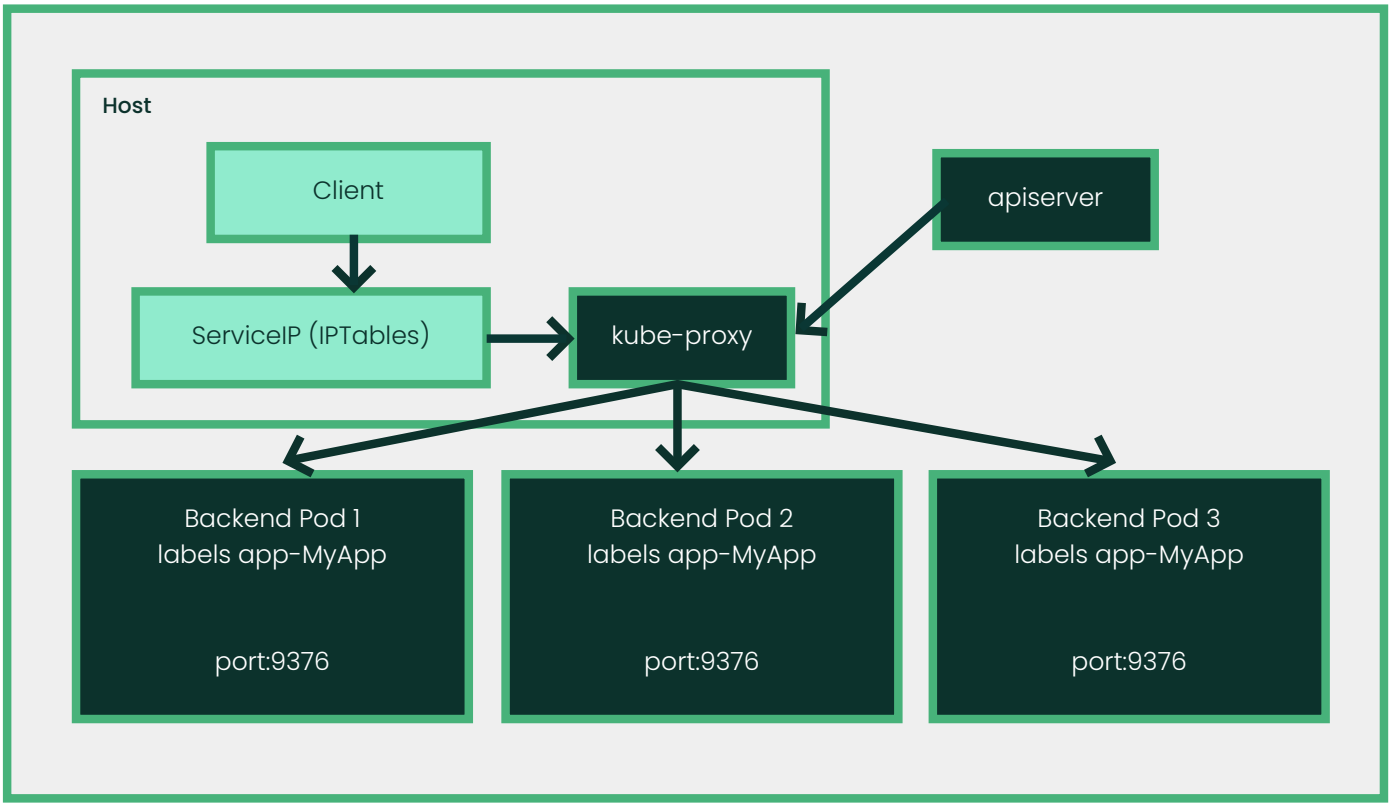
The easiest and simplest load balancer in Kubernetes is the Service. A Service routes traffic via round-robin to one or more replicas running within the cluster. The Service finds the replicas via a *selector*, which is a key/value pair that it looks for in the Pod labels. Any Pod that matches the selector is a candidate for traffic, and the Service sends each subsequent request to the next Pod in the list.

Services receive a stable IP address within the cluster, and if the cluster runs a DNS component like KubeDNS or CoreDNS, it also receives a DNS name in the format of `{name}.{namespace}.svc.cluster.local.` For example, applications within the cluster that want to communicate with a Service named *my-service* in the *default* namespace would send traffic to `my-service.default.svc.cluster.local.`

The following manifest creates a simple load balancer:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

When traffic arrives at the Service, `kube-proxy` forwards it to the appropriate backend.



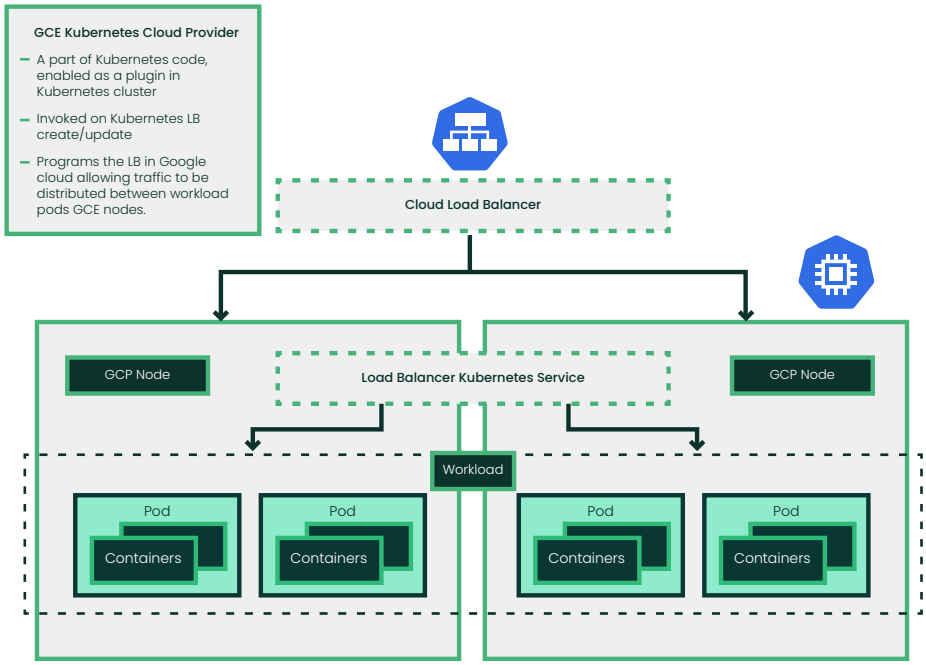
External Load Balancing

Layer 4

A load balancer that works at Layer 4 only routes traffic based on the TCP or UDP port. It does not look inside the packets or the data stream to make any decisions.

A Kubernetes Service of the type `LoadBalancer` creates a Layer 4 load balancer outside of the cluster, but it only does this if the cluster knows how. External load balancers require that the cluster use a supported cloud provider in its configuration and that the configuration for the cloud provider includes the relevant access credentials when required.

Once created, the *Status* field of the service shows the address of the external load balancer.



The following manifest creates an external Layer 4 load balancer:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
  clusterIP: 10.0.171.239
  loadBalancerIP: 78.11.24.19
  type: LoadBalancer
status:
  loadBalancer:
ingress:
    - ip: 146.148.47.155
```

Because a Layer 4 load balancer does not look into the packet stream, it only has basic capabilities. If a site runs multiple applications, every one of them requires an external load balancer. Escalating costs make that scenario inefficient.

Furthermore, because the `LoadBalancer` Service type requires a supported external cloud provider, and because Kubernetes only supports a small number of providers, many sites instead choose to run a Layer 7 load balancer inside of the cluster.

Layer 7

The Kubernetes resource that handles load balancing at Layer 7 is called an Ingress, and the component that creates Ingresses is known as an Ingress Controller.

The Ingress Resource

The Ingress resource defines the rules and routing for a particular application. Any number of Ingresses can exist within a cluster, each using a combination of host, path, or other rules to send traffic to a Service and then on to the Pods.

The following manifest defines an Ingress for the site `foo.bar.com`, sending `/foo` to the `s1` Service and `/bar` to the `s2` Service:

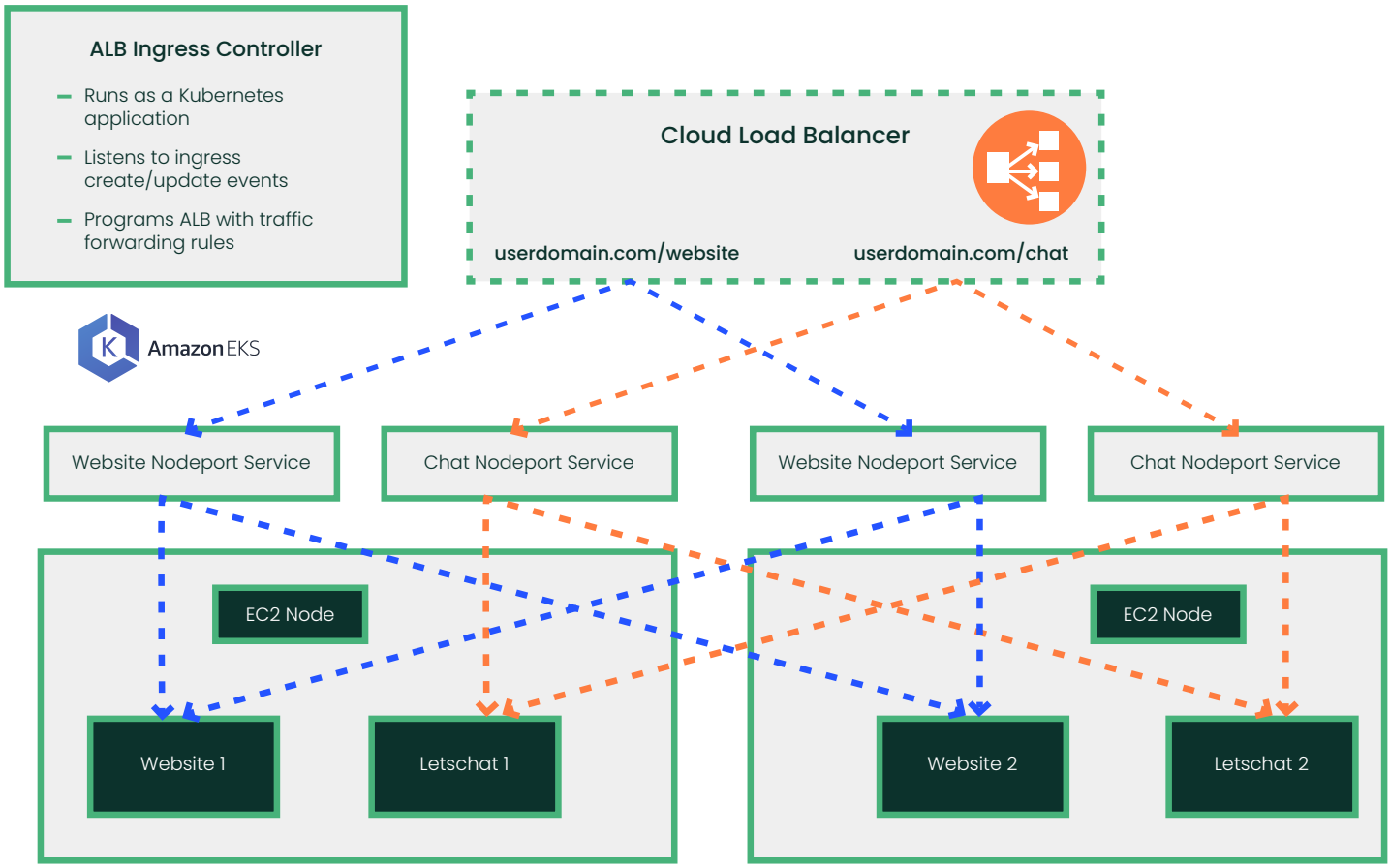
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
  annotations:
    nginx.ingress.Kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: s1
          servicePort: 80
      - path: /bar
        backend:
          serviceName: s2
          servicePort: 80
```

The Ingress Controller

An Ingress Controller listens for requests to create or modify Ingresses within the cluster and converts the rules in the manifests into configuration directives for a load balancing component. That component is either a software load balancer such as Nginx, HAProxy, or Traefik, or it's an external load balancer such as an Amazon ALB or an F5 Big/IP.

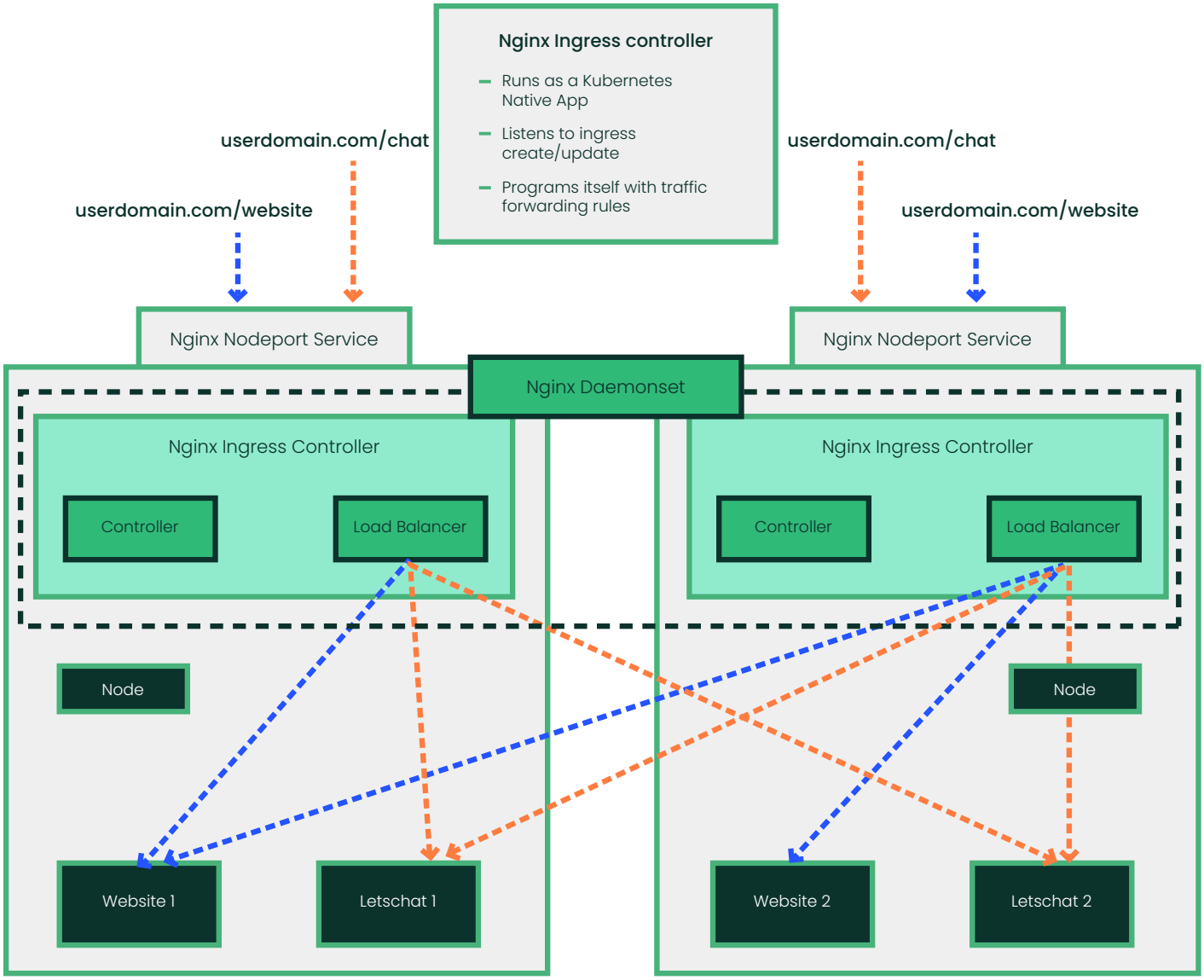
When working with an external load balancer the Ingress Controller is a lightweight component that translates the Ingress resource definitions from the cluster into API calls that configure the external piece.

The following diagram shows an Ingress Controller managing an Amazon ALB.



In the case of internal software load balancers, the Ingress Controller combines the management and load balancing components into one piece. It uses the instructions in the Ingress resource to reconfigure itself.

The following diagram shows a Nginx Ingress Controller working within a cluster.



Kubernetes uses annotations to control the behavior of the Ingress Controller. Although each controller has a list of accepted annotations, their use activates advanced features such as canary deployments, default backends, timeouts, redirects, CORS configuration, and more.

Load Balancing with Cloud Providers

Load balancers have a couple of limitations you should be aware of.

First, load balancers can only handle one IP address per service, which means if you run multiple services in your cluster, you must have a load balancer for each service. Running multiples load balancers can be expensive.

Second, if you want to use a load balancer with a Hosted Kubernetes cluster (i.e., clusters hosted in GKE, EKS, or AKS), the load balancer must be running within that cloud provider's infrastructure. In other words, cluster deployments on Amazon EKS, Google GKE, Azure AKS, and RKE on EC2 are supported by layer-4 load balancers from their respective cloud provider. Amazon EKS and Google GKE provide layer-7 load balancer support; layer 7 load balancer support on RKE on EC2 is provided by Nginx Ingress Controller, and is not supported on Azure AKS.

On cloud providers which support external load balancers, setting the `type` field to `LoadBalancer` provisions a load balancer for your Service. The actual creation of the load balancer happens asynchronously, and information about the provisioned balancer is published in the Service's `.status.loadBalancer` field

For example:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  clusterIP: 10.0.171.239
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
    - ip: 192.0.2.127
```

Traffic from the external load balancer is directed at the backend Pods. The cloud provider decides how it is load balanced.

Some cloud providers allow you to specify the `loadBalancerIP`, in which case the load-balancer is created with the user-specified `loadBalancerIP`. If the `loadBalancerIP` field is not specified, the loadBalancer is set up with an ephemeral IP address. If you specify a `loadBalancerIP` but your cloud provider does not support the feature, the `loadbalancerIP` field that you set is ignored.

Additional documentation from Kubernetes can help you properly configure your load balancer for a given cloud provider.

Conclusion

Kubernetes is powerful. It takes a simple container engine like Docker and elevates it to a level of usability appropriate for production environments. What starts as a series of Netfilter rules on a single host grows with Kubernetes to span multiple hosts, or even multiple disparate networks separated by geographical boundaries, with load balancers distributing traffic efficiently on any deployment. After reading this book, you're ready to make informed decisions about which networking approach to use, their capabilities, and how to leverage Kubernetes resources to connect the outside world to the applications running inside the cluster.

SUSE
Maxfeldstrasse 5
90409
Nuremberg
www.suse.com

For more information, contact SUSE at:
+1 800 796 3700 (U.S./Canada)
+49 (0)911-740 53-0 (Worldwide)

Thank You