ISOVALENT
now part of cisco

# Migrating from MetalLB to Cilium

Nico Vibert | Published: Feb 12, 2024 | Updated: Nov 07, 2024 | Cilium

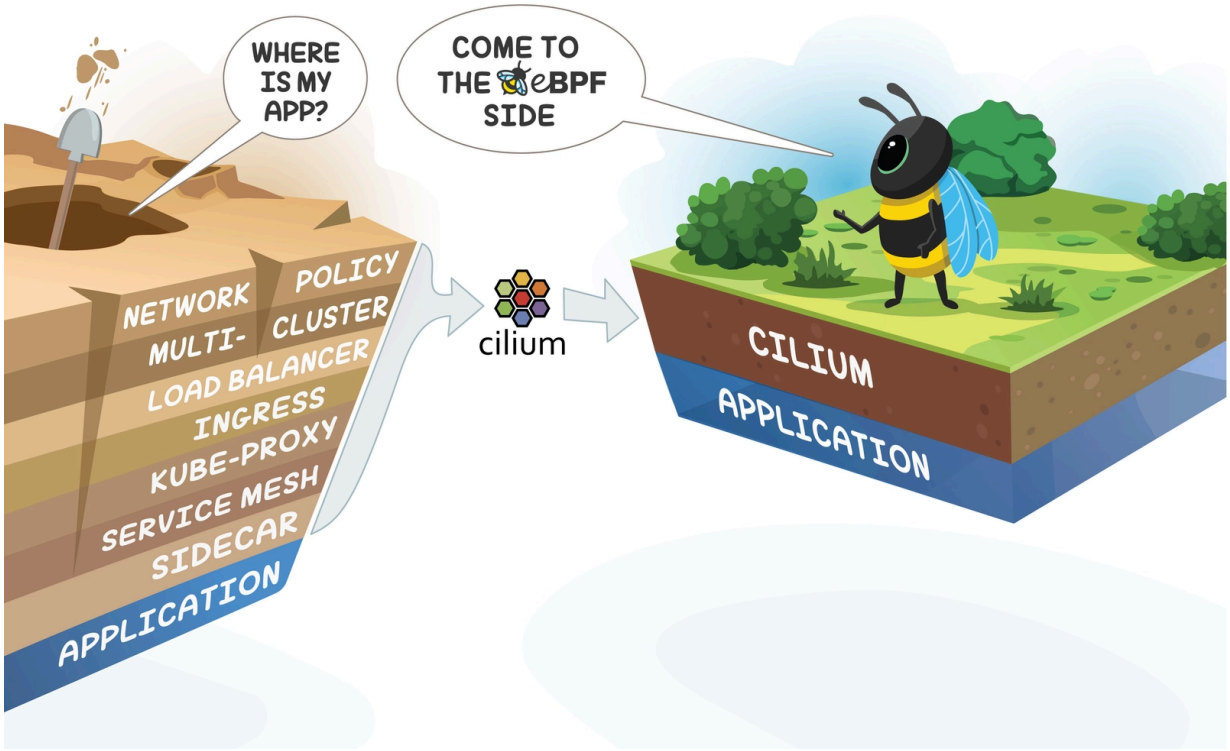


**Table of contents**



In this tutorial, we are going to explore how to migrate from MetalLB to Cilium.

Before I go into the technical details, let me explain the reasoning behind the cover above. It was actually the illustration of the booth we had at KubeCon US in 2023.

It was originally inspired by an illustration that my colleague Bill posted on Twitter.

Let me describe it – not only for accessibility purposes but also in the likelihood Twitter/X will be gone by the time you read this post. The original tweet by @theburningmonk shows two people on a deserted surface, with one asking the other: "I'm pretty sure the application is somewhere around here". Unbeknownst to them, the application is buried under them, under several layers of cloud native tools – Load-balancer, Ingress, Kube-proxy, Service Mesh and Side-car.

In response, Bill replied: "Fixed it, just needed one Cilium" and pasted a logo of Cilium on top of all the cloud native tools – as if to say, Cilium can do it all.

I've talked several times before about how Cilium can help folks deal with the overwhelming volume of cloud native networking tools (for example, with Cilium's Gateway API support) and even made one of my predictions that users will look at simplifying their stack of tools.

As highlighted in the Cilium 1.14 release blog post, one tool that Cilium users might not need any longer is MetalLB.

# Use cases for MetalLB

Let me start by saying – MetalLB is a very popular and reliable bare metal load-balancer tool.

MetalLB is indeed used to connect a cluster to the rest of the network – with BGP or, most commonly in home labs, with Layer 2 announcements (I will explain what this refers to later). It also provides IP Address Management (IPAM) capabilities for Kubernetes Services of the type LoadBalancer and generally acts as a Load Balancer for self-managed clusters (the cloud managed Kubernetes offerings tend to automatically do IPAM and LoadBalancing for you).

And MetalLB was used by Cilium in its early support of BGP, it was used in Cilium's own CI/CD pipelines and was even used in our popular Isovalent Cilium labs.

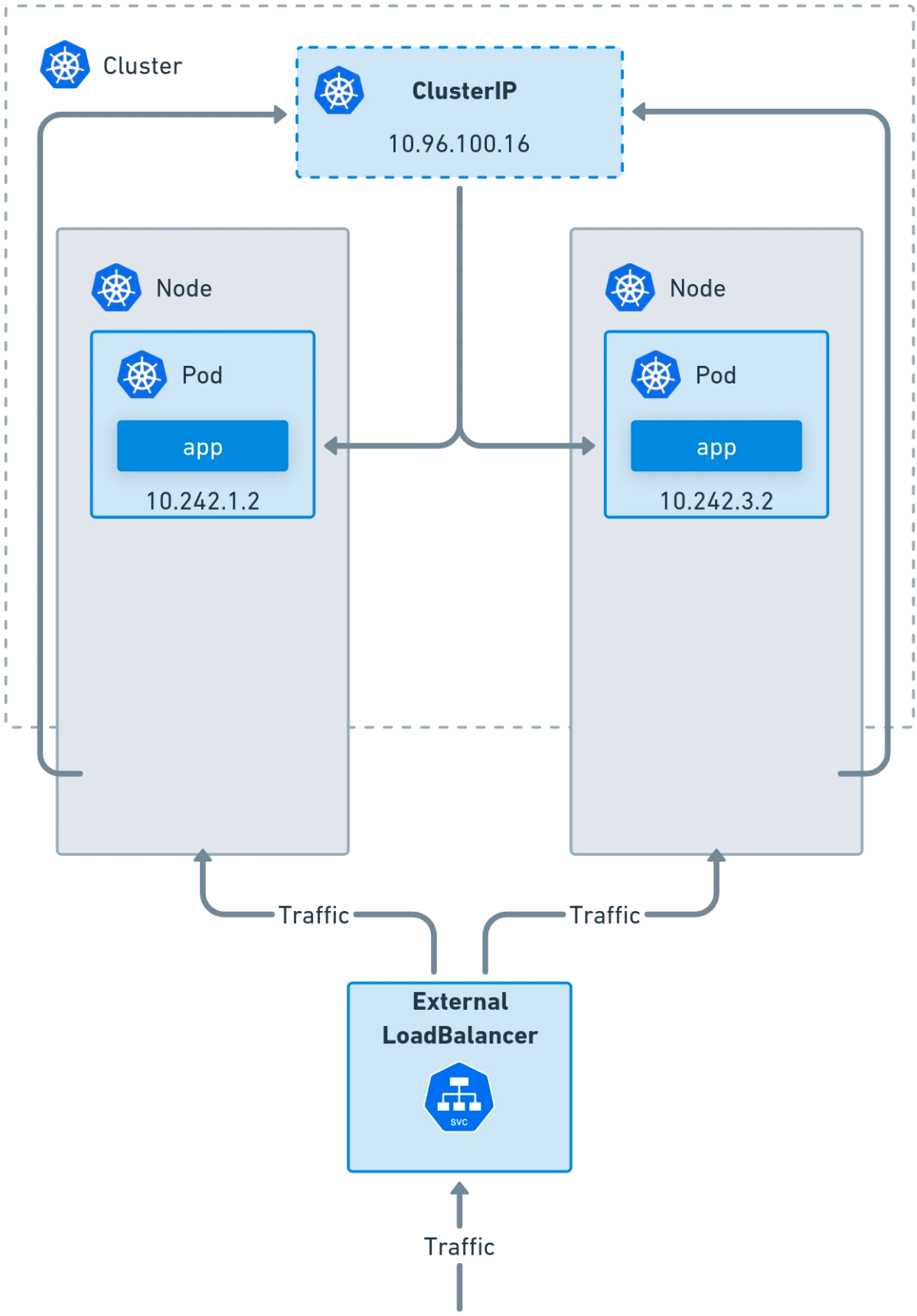
I used the past tense purposely in the previous sentence: Cilium no longer leverages MetalLB for BGP (we use GoBGP instead), it has been removed from the GitHub Actions CI/CD testing and is no longer used in our labs. As one of the lab maintainers, I had no issue with MetalLB – I simply wanted to reduce the number of tools and dependencies.

Let's walk through how Cilium supports these features and how we migrated from MetalLB in our labs.

# Load-Balancer IPAM

First, we are going to look at one of the key use cases supported by MetalLB – IP address management – and we will walk through how Cilium addresses it.

As described in the Kubernetes documentation, Kubernetes Services of the type LoadBalancer "expose the Service externally using an external load balancer. Kubernetes does not directly offer a load balancing component; you must provide one, or you can integrate your Kubernetes cluster with a cloud provider."



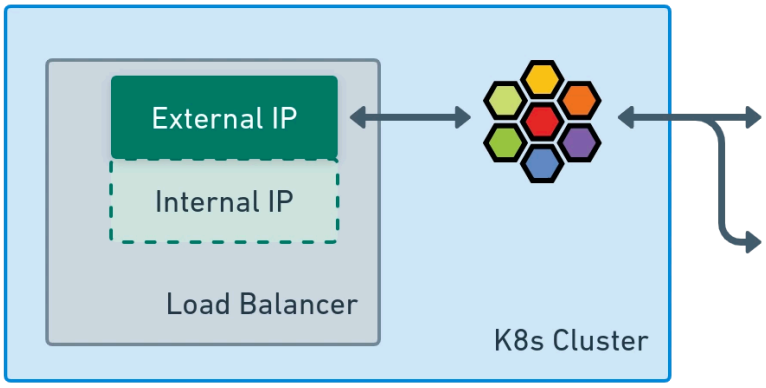

One of the tasks operated by such external load balancer is to **assign an external IP address to the Kubernetes Services**. As mentioned, the tool of choice for this had often been MetalLB. This is particularly relevant in

self-managed Kubernetes clusters as, when using cloud managed Kubernetes clusters, an IP and DNS entry will automatically be assigned to Kubernetes Services of the type LoadBalancer.

Cilium introduced support for Load-Balancer IP Address Management in Cilium 1.13 and we've seen a quick adoption of this feature. Let's review with an example.

## Load Balancer IP Address Management with Cilium

First, note this feature is enabled by default but dormant until the first IP Pool is added to the cluster. From this pool of IP addresses (defined in a `CiliumLoadBalancerIPPool`) will be allocated IPs to Kubernetes Services of the type `LoadBalancer`.




```yaml
# IP pool, label selector
apiVersion: "cilium.io/v2alpha1"
kind: CiliumLoadBalancerIPPool
metadata:
  name: "pool-green"
spec:
  cidrs:
  - cidr: "40.0.10.0/24"
  serviceSelector:
    matchLabels:
      color: green
```

```yaml
# Service, label selector
apiVersion: v1
kind: Service
metadata:
  name: service-green
  namespace: tenant-b
  labels:
    color: green
spec:
  type: LoadBalancer
  ports:
  - port: 1234
```

LoadBalancer Services can be automatically created when exposing Services externally via the Ingress or Gateway API Resources but for this example, we'll create a Service of the type LoadBalancer manually.

Let's review a simple IP Pool:

```
$ cat pool.yaml
# No selector, match all
apiVersion: "cilium.io/v2alpha1"
kind: CiliumLoadBalancerIPPool
metadata:
name: "pool"
spec:
cidrs:
- cidr: "20.0.10.0/24"
```

IP addresses from the `20.0.10.0/24` range will be assigned to a LoadBalancer Service.

Before we deploy the pool, check what happens when we deploy a Service of the type `LoadBalancer`. The `service-blue` Service is a simple Service that is labelled `color:blue` and is in the `tenant-a` namespace.

```
$ cat service-blue.yaml
apiVersion: v1
kind: Service
metadata:
name: service-blue
namespace: tenant-a
labels:
color: blue
spec:
type: LoadBalancer
ports:
- port: 1234
```

Let's deploy it. At first, no IP address has been allocated yet (the `External-IP` is still `<pending>`).

```
$ kubectl apply -f service-blue.yaml
service/service-blue created
$ kubectl get svc/service-blue -n tenant-a
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service-blue LoadBalancer 10.96.242.86 <pending> 1234:32543/TCP
23s
```

Next, we deploy the Cilium IP Pool:

```
$ kubectl apply -f pool.yaml
ciliumloadbalancerippool.cilium.io/pool created
```

Let us check again the Service – an IP address from the `20.0.10.0/24` pool has been assigned in the field `EXTERNAL-IP`.

```
$ kubectl get svc/service-blue -n tenant-a
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service-blue LoadBalancer 10.96.242.86 20.0.10.123
1234:32543/TCP 4m11s
```

## Service Selectors

Operators may want to have applications and services assigned IP addresses from specific ranges. This might be useful to then enforce network security rules on an traditional border firewall.

For example, you may want test or production services to get IP addresses from different ranges and apply different rules on the firewall as

permissions to the test network might be looser than to the production one.

In our example, we will keep using colors. Services (in the `tenant-b` namespace) tagged with `blue`, `yellow` or `red` will be allowed to pick up IP addresses from a `primary` colors IP pool.

This would be done by using a Service Selector, with a regular expression such as the one defined in `pool-primary.yaml`:

```
YAML:
$ cat pool-primary.yaml
apiVersion: "cilium.io/v2alpha1"
kind: CiliumLoadBalancerIPPool
metadata:
name: "pool-primary"
spec:
cidrs:
- cidr: "60.0.10.0/24"
serviceSelector:
matchExpressions:
- {key: color, operator: In, values: [yellow, red, blue]}
```

The expression `- {key: color, operator: In, values: [yellow, red, blue]}` checks whether the Service requesting the IP has a label with the key `color` with a value of either `yellow`, `red` or `blue`.

Let's deploy it:

```
$ kubectl apply -f pool-primary.yaml
ciliumloadbalancerippool.cilium.io/pool-primary created
```

This time, we'll try this with a Service that hasn't got the right label and one with the right label.

```
$ kubectl apply -f service-red.yaml
service/service-red created
$ kubectl apply -f service-green.yaml
service/service-green created
$ kubectl get svc -n tenant-b --show-labels
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE LABELS
service-green LoadBalancer 10.96.138.190 <pending>
1234:32359/TCP 9s color=green
service-red LoadBalancer 10.96.254.23 60.0.10.145
1234:31758/TCP 9s color=red
```

Review their assigned IPs and labels in the output above. Unlike `service-red`, `service-green` does not get an IP address. It's simply because it doesn't match the regular expression defined in the previously defined pool (green is not one of the primary colors).

Let's now use an IP Pool that matches the `color:green` label:

```
YAML:
$ cat pool-green.yaml
apiVersion: "cilium.io/v2alpha1"
kind: CiliumLoadBalancerIPPool
metadata:
name: "pool-green"
spec:
cidrs:
- cidr: "40.0.10.0/24"
serviceSelector:
matchLabels:
color: green

$ kubectl apply -f pool-green.yaml
ciliumloadbalancerippool.cilium.io/pool-green created

$ kubectl get svc/service-green -n tenant-b
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service-green LoadBalancer 10.96.138.190 40.0.10.96
1234:32359/TCP 3m47s
```

This time, an IP address from the `40.0.10.0/24` was assigned to `service-green`.

## Requesting a specific IP

Users might want to request a specific set of IPs from a particular range. Cilium supports two methods to achieve this: the `.spec.loadBalancerIP` method (legacy, deprecated in K8S 1.24) and an annotation-based method.

The former method has now been deprecated because it did not support dual-stack services (only a single IP could be requested). With the latter

annotation method, a list of requested (v4 or v6) IPs will be specified instead. We will look at an Dual Stack example later on.

Review this `yellow` service. Note the annotation used to requested specific IPs. In this scenario, we are requesting 4 IP addresses:

```
YAML:
$ cat service-yellow.yaml
apiVersion: v1
kind: Service
metadata:
name: service-yellow
namespace: tenant-c
labels:
color: yellow
annotations:
"io.cilium/lb-ipam-ips":
"30.0.10.100,40.0.10.100,50.0.10.100,60.0.10.100"
spec:
type: LoadBalancer
ports:
- port: 1234
```

We're also going to use another pool, this time, matching on the namespace ( `tenant-c` in our example).

```
YAML:
$ cat pool-yellow.yaml
# Namespace selector
apiVersion: "cilium.io/v2alpha1"
kind: CiliumLoadBalancerIPPool
metadata:
name: "pool-yellow"
spec:
cidrs:
- cidr: "50.0.10.0/24"
serviceSelector:
matchLabels:
"io.kubernetes.service.namespace": "tenant-c"
```

Let's deploy this service and pool and observe which IP addresses have been allocated:

```
$ kubectl apply -f service-yellow.yaml
service/service-yellow created
$ kubectl apply -f pool-yellow.yaml
ciliumloadbalancerippool.cilium.io/pool-yellow created
$ kubectl get svc/service-yellow -n tenant-c
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service-yellow LoadBalancer 10.96.56.113
50.0.10.100,60.0.10.100 1234:30026/TCP 3s
```

Note that the Service was allocated two of the requested IP addresses:

- `50.0.10.100` because it matches the namespace ( `tenant-c` ).
- `60.0.10.100` because it matches the primary colors labels.

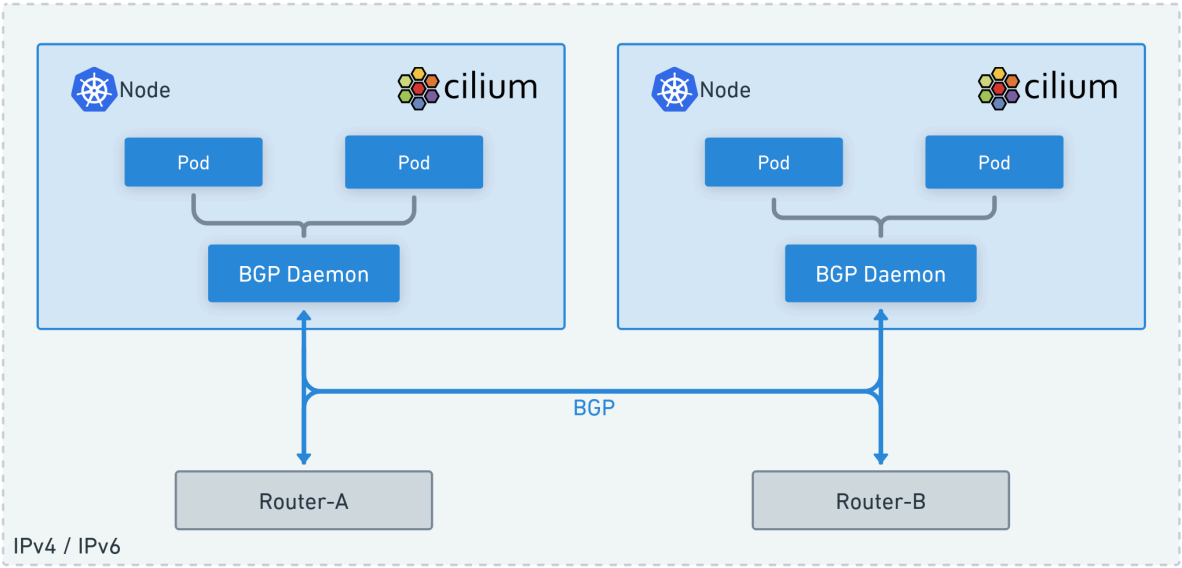
Two other IP addresses were not assigned:

- `30.0.10.100` is not part of any defined pools.
- `40.0.10.100` is part of an existing pool but its `serviceSelector` doesn't match with the Service (remember that this is part of the `pool-green` pool which matched on a green label).

# L3 Announcement over BGP

Now that we have one (or multiple) IP address(es) assigned to our Services, we need to advertise them to the rest of the network so that external clients can reach them (and the Service it's fronting).

We covered BGP in more details in a recent blog post ("Connecting your Kubernetes island to your network with Cilium BGP") but to keep it short, know that BGP on Cilium enables you simply to set up BGP peering sessions between Cilium-managed nodes and Top-of-rack (ToR) devices and to tell the rest of the network about the networks and IP addresses used by your pods and your services.




Setting up BGP can be done by using the following resources:

- `CiliumBGPClusterConfig`: Defines BGP instances and peer configurations that are applied to multiple nodes.

- `CiliumBGPPeerConfig`: A common set of BGP peering setting. It can be used across multiple peers.
- `CiliumBGPAdvertisements`: Defines prefixes that are injected into the BGP routing table.

Here is an example:

```
---
apiVersion: "cilium.io/v2alpha1"
kind: CiliumBGPClusterConfig
metadata:
name: tor
spec:
nodeSelector:
matchLabels:
kubernetes.io/hostname: clab-bgp-cplane-devel-control-plane
bgpInstances:
- name: "instance-65001"
localASN: 65001
peers:
- name: "peer-65000-tor"
peerASN: 65000
peerAddress: "172.0.0.1"
peerConfigRef:
name: "peer-config-generic"
---
apiVersion: "cilium.io/v2alpha1"
kind: CiliumBGPPeerConfig
metadata:
name: peer-config-generic
spec:
families:
- afi: ipv4
safi: unicast
advertisements:
matchLabels:
advertise: "generic"
---
apiVersion: "cilium.io/v2alpha1"
kind: CiliumBGPAdvertisement
metadata:
name: services
```

```
  labels:
  advertise: generic
  spec:
  advertisements:
  - advertisementType: "PodCIDR"
  - advertisementType: "Service"
  service:
  addresses:
  - LoadBalancerIP
  selector:
  matchLabels:
  color: yellow
  matchExpressions:
  - {key: io.kubernetes.service.namespace, operator: In, values:
  ["tenant-c"]}
```

As you can see in the YAML above, we can specify which services are advertised, using a service selector. In our example, we only want to advertise to our peers Services with the `color:yellow` label and located in the `tenant-c` namespace.

Once this peering policy is applied (and assuming that the peer has been correctly configured), the peer will see routes such as:

```
$ show bgp ipv4
BGP table version is 3, local router ID is 172.0.0.1, vrf id 0
Default local pref 100, local AS 65000
Status codes: s suppressed, d damped, h history, * valid, >
best, = multipath,
i internal, r RIB-failure, S Stale, R Removed
Nexthop codes:<a class=
```

The peer can see the Pod CIDR `10.244.0.0/24` advertised by Cilium but also the LoadBalancer Service IPs that match the filters defined in the Service Selector.

## IPv6 Support

The examples so far focused on IPv4 but both LB-IPAM and BGP on Cilium support IPv6.

Here is an example of an LB-IPAM pool that would assign both an IPv4 and IPv6 address:

```
$ yq lb-pool.yaml
---
apiVersion: "cilium.io/v2alpha1"
kind: CiliumLoadBalancerIPPool
metadata:
name: "empire-ip-pool"
labels:
org: empire
spec:
cidrs:
- cidr: "172.18.255.200/29"
- cidr: "2001:db8:dead:beef::0/64"
```

When deploying a DualStack Service with the right label, the Service receives both an IPv4 and IPv6 address:

```
$ kubectl -n batuu get svc deathstar
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
deathstar LoadBalancer 10.2.135.143
172.18.255.204,2001:db8:dead:beef::f1e 80:32488/TCP 18m
```

We can peer over IPv6 with our BGP neighbor and advertise our IPv6 prefixes accordingly:

```
$ kubectl get ciliumbgpclusterconfig control-plane -o yaml | yq
'.spec'
bgpInstances:
- localASN: 65001
name: instance-65001
peers:
- name: peer-65000
peerASN: 65000
peerAddress: fd00:10:0:1::1
peerConfigRef:
group: cilium.io
kind: CiliumBGPPeerConfig
name: generic
nodeSelector:
matchLabels:
kubernetes.io/hostname: kind-control-plane

$ kubectl get ciliumbgppeerconfig generic -o yaml | yq '.spec'
ebgpMultihop: 1
families:
- advertisements:
matchLabels:
advertise: generic
afi: ipv4
safi: unicast
- advertisements:
matchLabels:
advertise: generic
afi: ipv6
safi: unicast

$ kubectl get ciliumbgpadvertisements generic -o yaml | yq
'.spec'
advertisements:
- advertisementType: PodCIDR
```

```
  - advertisementType: Service
  selector:
  matchLabels:
  announced: bgp
  service:
  addresses:
  - LoadBalancerIP
```

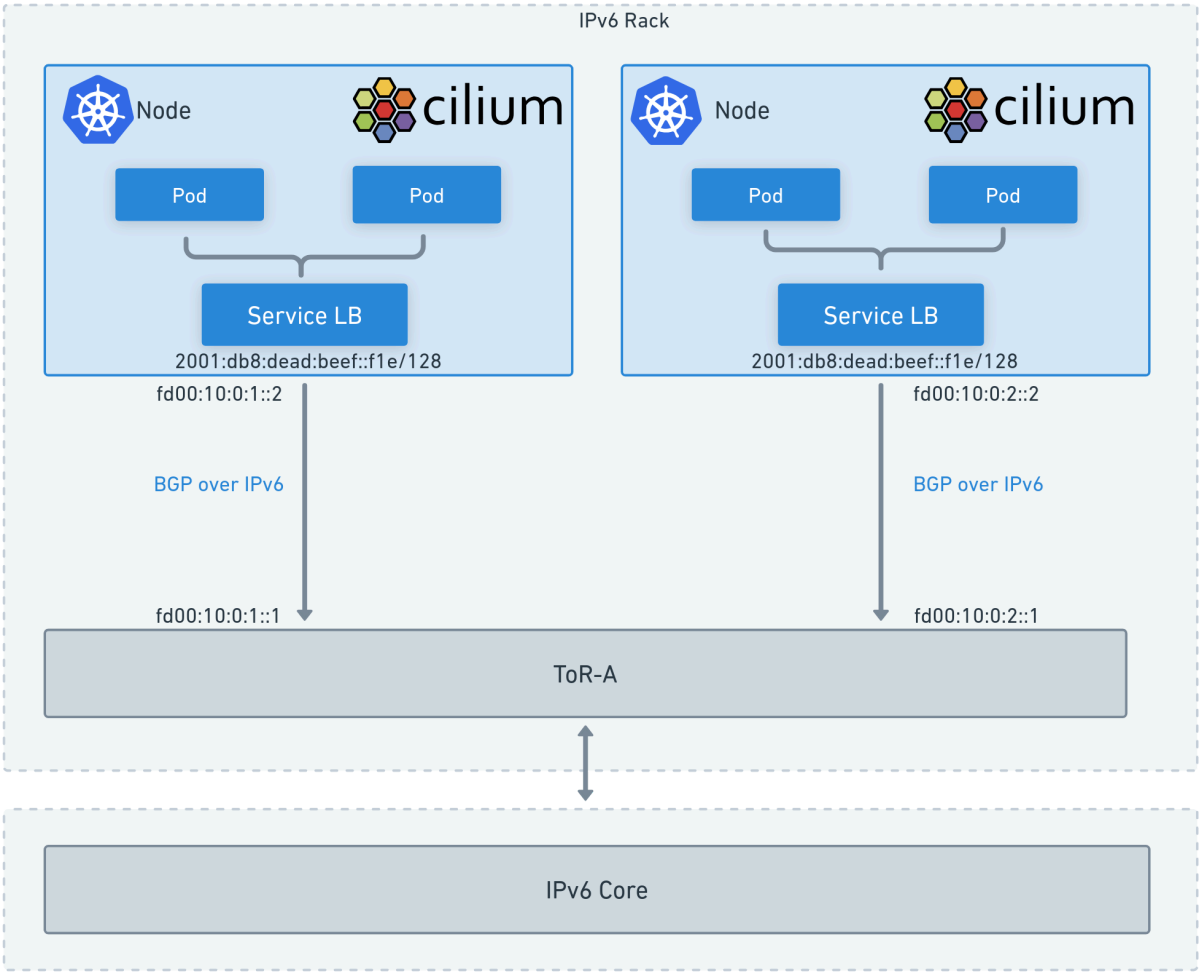
When we log onto our neighbour, we can see that the Service IPv6 address (alongside the IPv6 PodCIDR) has been received:

```
router0# show ip bgp ipv6 neighbors fd00:10:0:1::2 routes
BGP table version is 4, local router ID is 10.0.0.1, vrf id 0
Default local pref 100, local AS 65000
Status codes: s suppressed, d damped, h history, * valid, >
best, = multipath,
i internal, r RIB-failure, S Stale, R Removed
Origin codes: i - IGP, e - EGP, ? - incomplete
RPKI validation codes: V valid, I invalid, N Not found

Network Next Hop Metric LocPrf Weight Path
*> 2001:db8:dead:beef::f1e/128
fd00:10:0:1::2 0 65001 i
*> fd00:10:1::/64 fd00:10:0:1::2 0 65001 i

Displayed 2 routes and 6 total paths
```

Note how our Top of Rack device is actually peering with multiple Cilium nodes.

```
router0# show ip bgp ipv6 summary

IPv6 Unicast Summary (VRF default):
BGP router identifier 10.0.0.1, local AS number 65000 vrf-id 0
BGP table version 4
RIB entries 7, using 1344 bytes of memory
Peers 3, using 2151 KiB of memory
Peer groups 1, using 64 bytes of memory

Neighbor V AS MsgRcvd MsgSent TblVer InQ OutQ Up/Down
State/PfxRcd PfxSnt Desc
fd00:10:0:1::2 4 65001 5580 5583 0 0 0 1d22h27m 2 4 N/A
fd00:10:0:2::2 4 65002 5580 5584 0 0 0 1d22h27m 2 4 N/A
fd00:10:0:3::2 4 65003 5580 5584 0 0 0 1d22h27m 2 4 N/A

Total number of neighbors 3
```

Here is an illustration of the network topology highlighted in this paragraph:



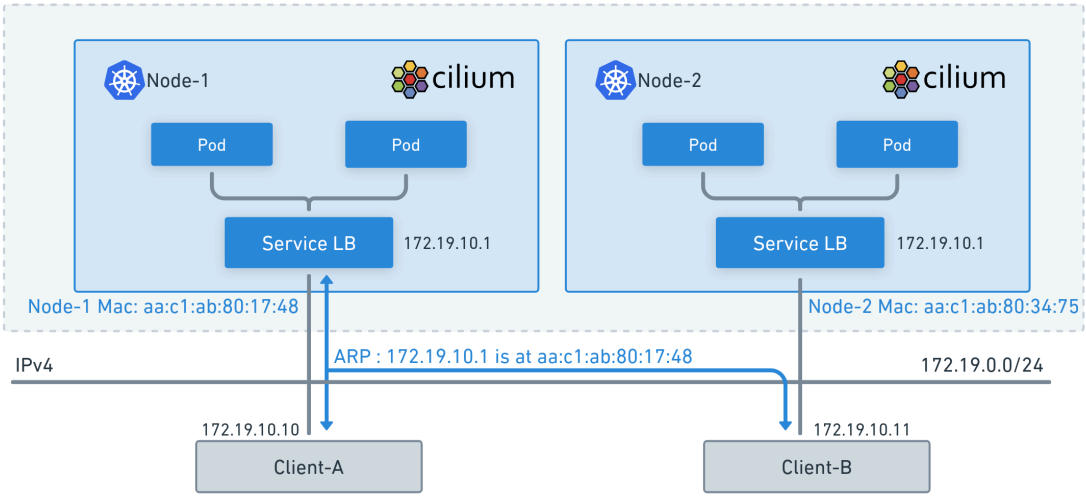

All this is nice if you love routing and networking but, for those of you with a difficult relationship with BGP, we have another option for you.

# L2 Announcement with ARP

While using BGP to advertise our Service IPs works great, not every device in the network will be BGP-capable and BGP might be overkill for some use cases, for example, in home labs.

One popular MetalLB use case that is now supported by Cilium is **Layer 2 announcement, using ARP**.

If you have local clients on the network that need access to the Service, they need to know which MAC address to send their traffic to. This is the role of ARP – the Address Resolution Protocol is used to discover the MAC address associated with a particular IP.




In the image above, our clients Client-A and Client-B are located on the same network as the Kubernetes LoadBalancer Service. In this case, BGP is superfluous; the clients simply need to understand which MAC address is associated with the IP address of our service so that they know to which host they need to send the traffic to.

What Cilium can do is reply to ARP requests from clients for LoadBalancer IPs or External IPs and enable these clients to access their services.

The feature can be tested in our Cilium LB-IPAM and L2 Announcements lab:

To enable this feature, use the following flags:

```
kubeProxyReplacement: strict
l2announcements:
enabled: true
devices: {eth0, net0}
externalIPs:
enabled: true
```

Let's verify it's enabled with the following command:

```
$ cilium config view | grep enable-l2
enable-l2-announcements true
enable-l2-neigh-discovery true
```

Why don't we walk through another Star Wars-inspired example? We want to advertise the following Death Star service locally. Notice how the DeathStar service has an external IP address ( `12.0.0.101` ) and the `org:empire` label.

```
# kubectl get svc deathstar --show-labels
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE LABELS
deathstar ClusterIP 10.96.95.207 12.0.0.101 80/TCP 72s
org=empire
```

Let's advertise this Service locally with a simple `CiliumL2AnnouncementPolicy` . The policy below is pretty simple to understand. It will advertise locally the IP addresses of any Services with the label `org: empire` , whether they are External IPs or LoadBalancer IPs. Only the worker nodes will be replying to ARP requests (the node selector expression excludes the control plane node).

```
# cat layer2-policy.yaml
---
apiVersion: "cilium.io/v2alpha1"
kind: CiliumL2AnnouncementPolicy
metadata:
name: l2announcement-policy
spec:
serviceSelector:
matchLabels:
org: empire
nodeSelector:
matchExpressions:
- key: node-role.kubernetes.io/control-plane
operator: DoesNotExist
interfaces:
- ^eth[0-9]+
externalIPs: true
loadBalancerIPs: true
```

Once deployed, a client from outside the cluster can access the service:

```
$ curl -s --max-time 2 http://12.0.0.101/v1/
{
"name": "Death Star",
"hostname": "deathstar-7848d6c4d5-27m7j",
"model": "DS-1 Orbital Battle Station",
"manufacturer": "Imperial Department of Military Research,
Sienar Fleet Systems",
"cost_in_credits": "1000000000000",
"length": "120000",
"crew": "342953",
"passengers": "843342",
"cargo_capacity": "1000000000000",
"hyperdrive_rating": "4.0",
"starship_class": "Deep Space Mobile Battlestation",
"api": [
"GET /v1",
"GET /v1/healthz",
"POST /v1/request-landing",
"PUT /v1/cargobay",
"GET /v1/hyper-matter-reactor/status",
"PUT /v1/exhaust-port"
]
}
```

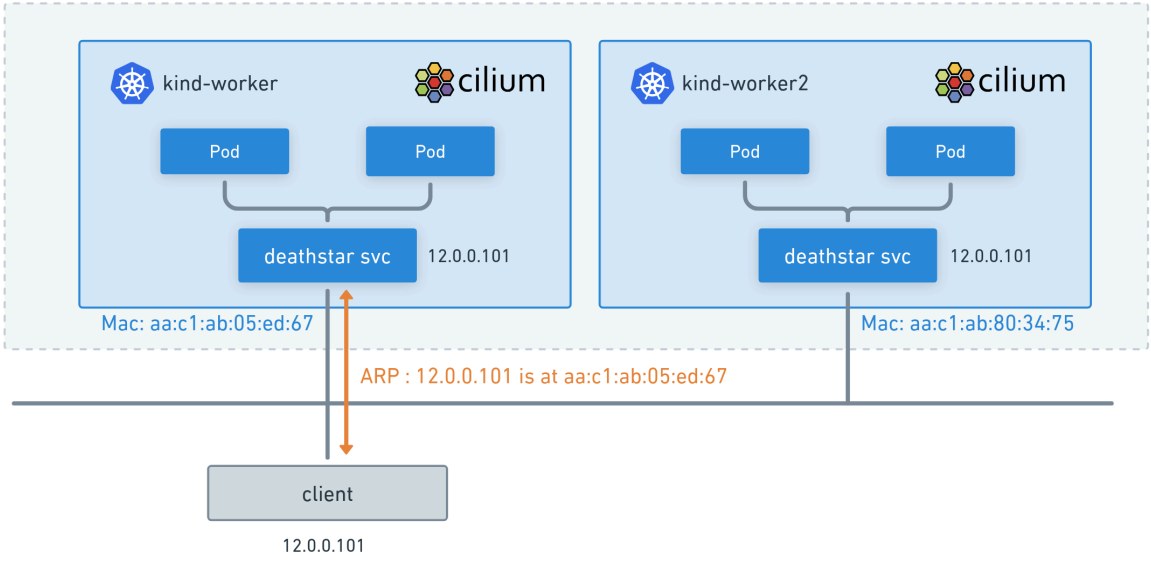
If you really want to know what happens under the hood, here is a screenshot of termshark showing the ARP request and reply. In this instance, the client (with an IP of `12.0.0.1`) wants to access a Service at `12.0.0.101` and sends an ARP request as a broadcast asking "Who has 12.0.0.101? Tell 12.0.0.1".

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

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

The Cilium node will reply back to the client with its own MAC address (“12.0.0.101 is at aa:c1:ab:05:ed:67”).

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

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

This works great and is a fantastic alternative when BGP is unwanted or unneeded.



The only drawback with Layer 2 Announcement is that it does not currently support IPv6.

## Translating MetalLB configuration to Cilium

So far, we've reviewed how many of the common MetalLB use cases can be natively addressed by Cilium. To help you migrate from MetalLB, here is a sample configuration of our own migration from MetalLB to Cilium's LB-IPAM and L2 Announcement features.

Here is the MetalLB configuration we used in our popular Service Mesh and Gateway API labs:

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
name: default
namespace: metallb-system
spec:
addresses:
- 172.18.255.193-172.18.255.206
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
name: l2advertisement
namespace: metallb-system
spec:
ipAddressPools:
- default
nodeSelectors:
- matchLabels:
kubernetes.io/hostname: NodeA
interfaces:
- eth0
```

Here is the equivalent Cilium configuration:

```
apiVersion: "cilium.io/v2alpha1"
kind: CiliumLoadBalancerIPPool
metadata:
name: "pool"
spec:
cidrs:
- cidr: "172.18.255.192/28"
---
apiVersion: "cilium.io/v2alpha1"
kind: CiliumL2AnnouncementPolicy
metadata:
name: l2policy
spec:
loadBalancerIPs: true
interfaces:
- eth0
nodeSelector:
matchLabels:
kubernetes.io/hostname: NodeA
```

Both configurations would allocate an IP address from the `172.18.255.192/28` IP range and would announce the LoadBalancer IP from the `eth0` network interfaces of the `NodeA` node.

# Conclusion

With every release, Cilium becomes so much more than just a Container Network Interface (CNI). It is now an Ingress controller, a Gateway API implementation, a VPN and, as you saw in this blog post, it can also act as a bare metal load balancer. And while many users will continue to happily use MetalLB, those of you who already use Cilium may decide that one fewer tool to manage already helps reducing the operational fatigue.

Thanks for reading.

# Learn More

- Lab: Cilium LoadBalancer IPAM and L2 Service Announcement
- Lab: Cilium LoadBalancer IPAM and BGP Service Advertisement
- Video: Layer 2 Announcement with Cilium

AUTHOR

**Nico Vibert**

Senior Staff Technical Marketing Engineer

Nico Vibert is a Senior Staff Technical Marketing Engineer at Isovalent, the company behind the open-source cloud-native solution Cilium.
Prior to joining Isovalent, Nico worked in many different roles—operations and support, design and architecture, and technical pre-sales—at companies such as HashiCorp, VMware, and Cisco.
In his current role, Nico focuses primarily on creating content to make networking a more approachable field and regularly speaks at events like KubeCon, VMworld, and Cisco Live.
Nico has held over 15 networking certifications, including the Cisco Certified Internetwork Expert CCIE (# 22990) and is now the Lead Subject Matter Expert on the Cilium Certified Associate (CCA) certification.