OneCloud
Consulting

Kubernetes Networking
90 Minutes
will start at :05

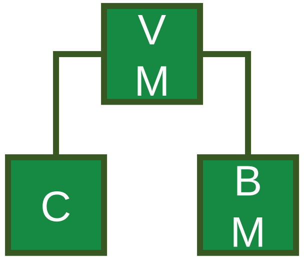
John Swartz, CCIE#4426
John.Swartz@OneCloudInc.com
Class Files: https://1cld.us/IE
Class Eval: https://1cld.us/HS
Linkedin.com/in/swartz

Challenges with Container Networking

Security

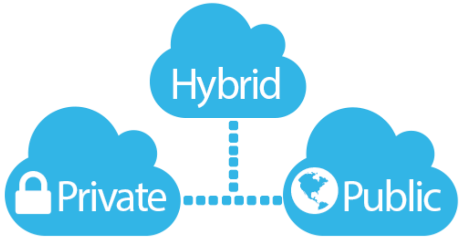
- App Isolation
- Micro-segmentation
- Monitoring & Visibility

Complex Deployments



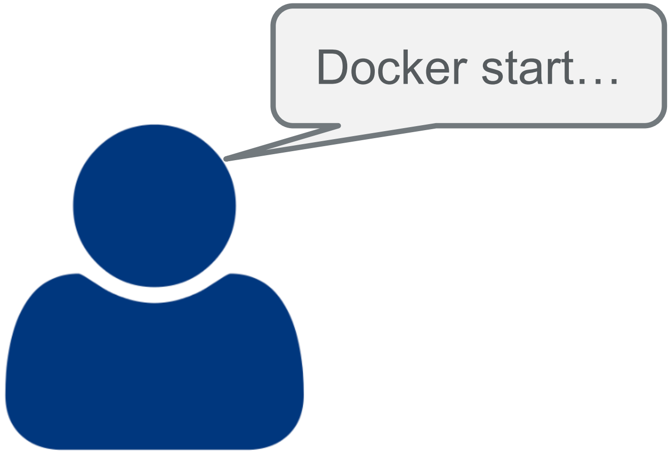
- Connect containers to VMs and bare metal servers
- DC GW Integration

Cloud



- Private Cloud
- Public Cloud
- Hybrid Cloud

Docker Networking – Bridge Mode

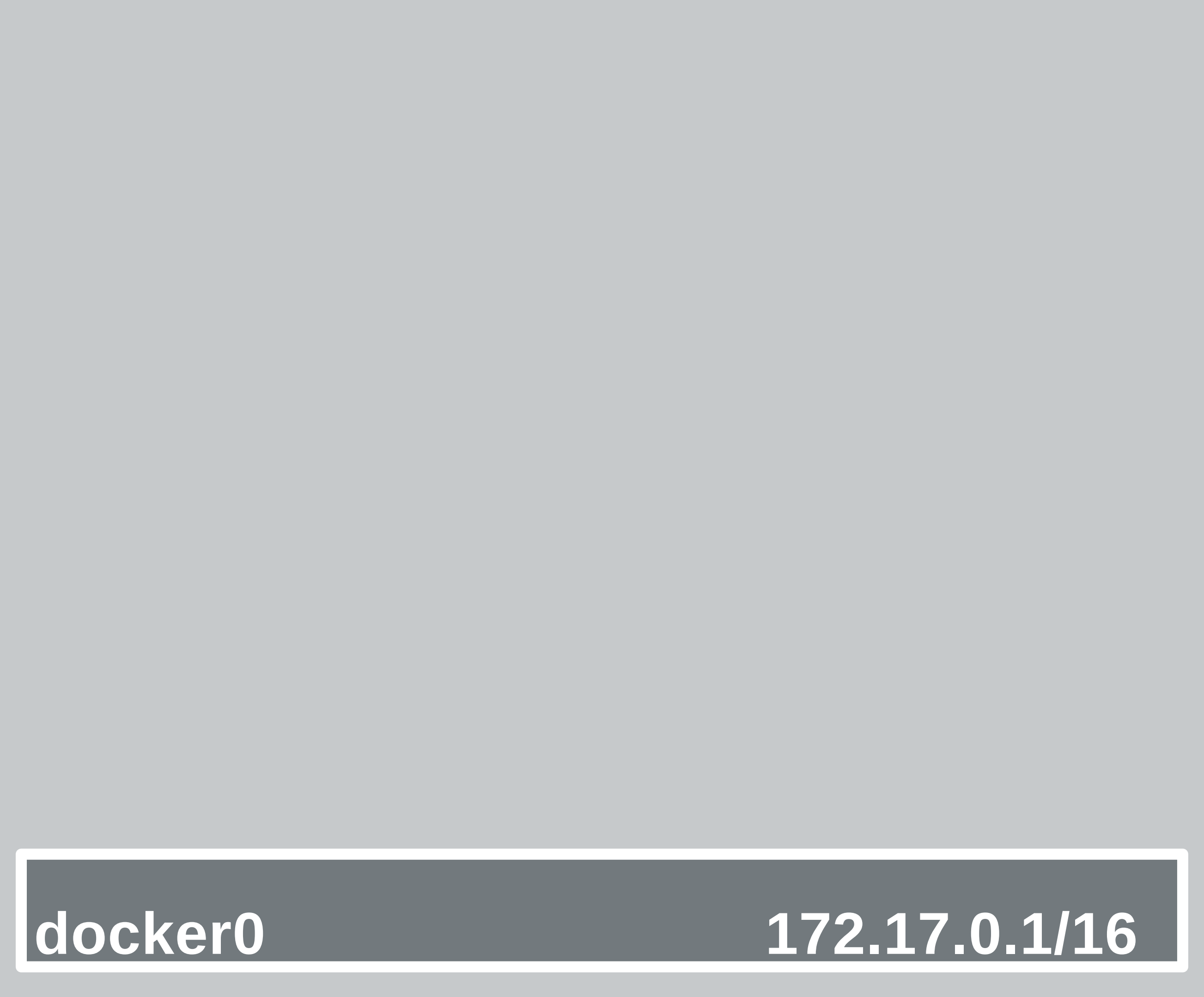


Docker start…

Docker Networking (Cont.)

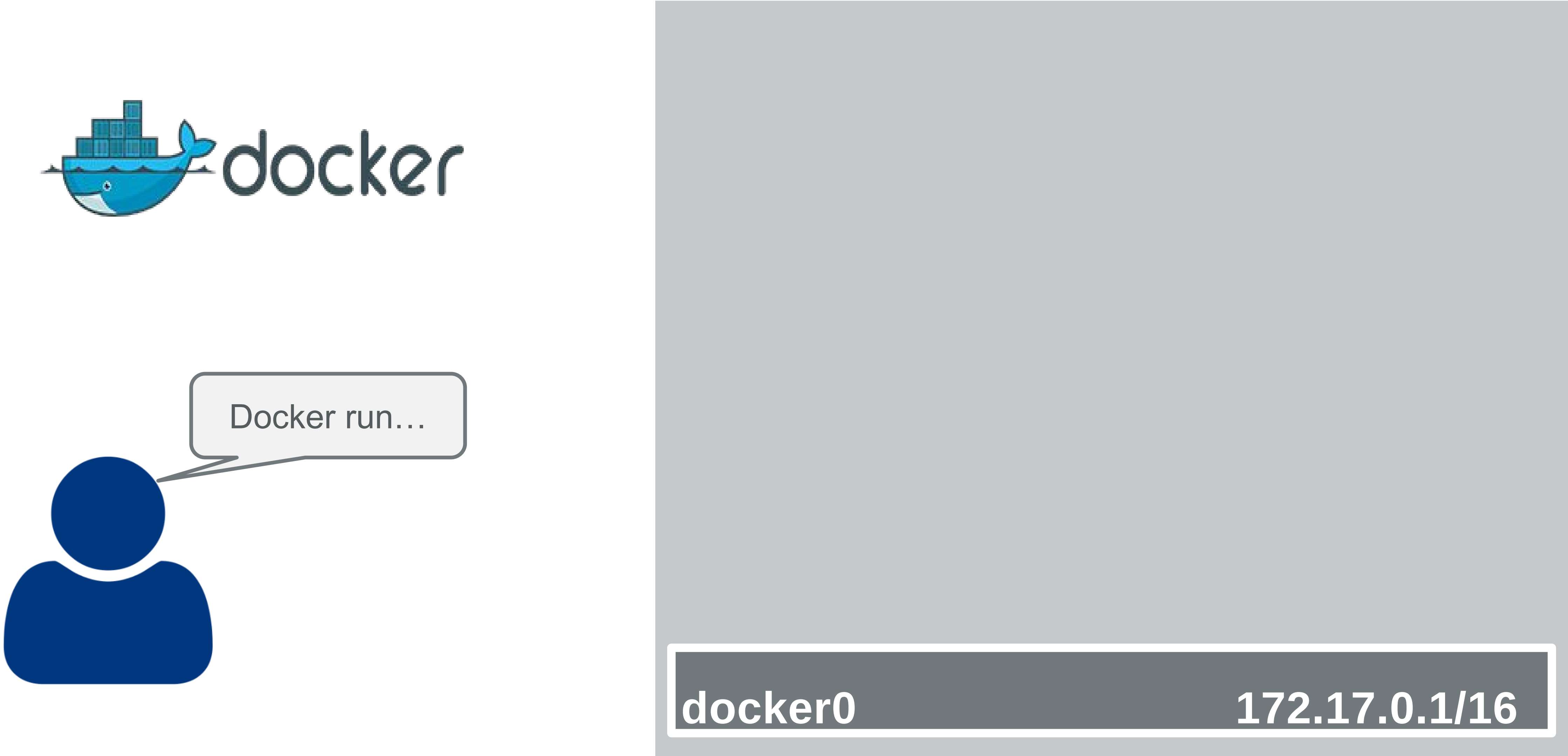
Docker start…

Docker Networking (Cont.)



docker0 172.17.0.1/16

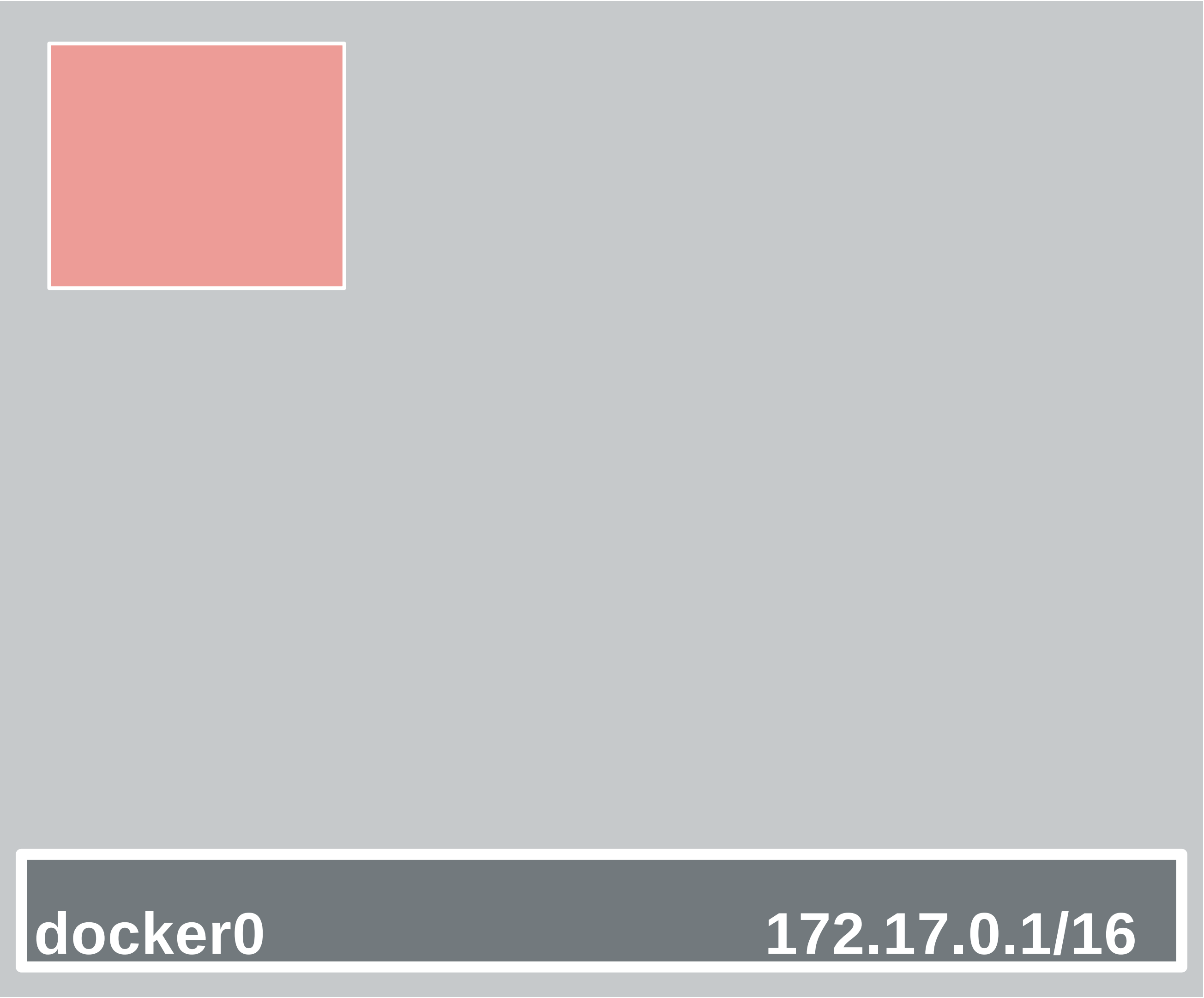
Docker Networking (Cont.)



Docker run…

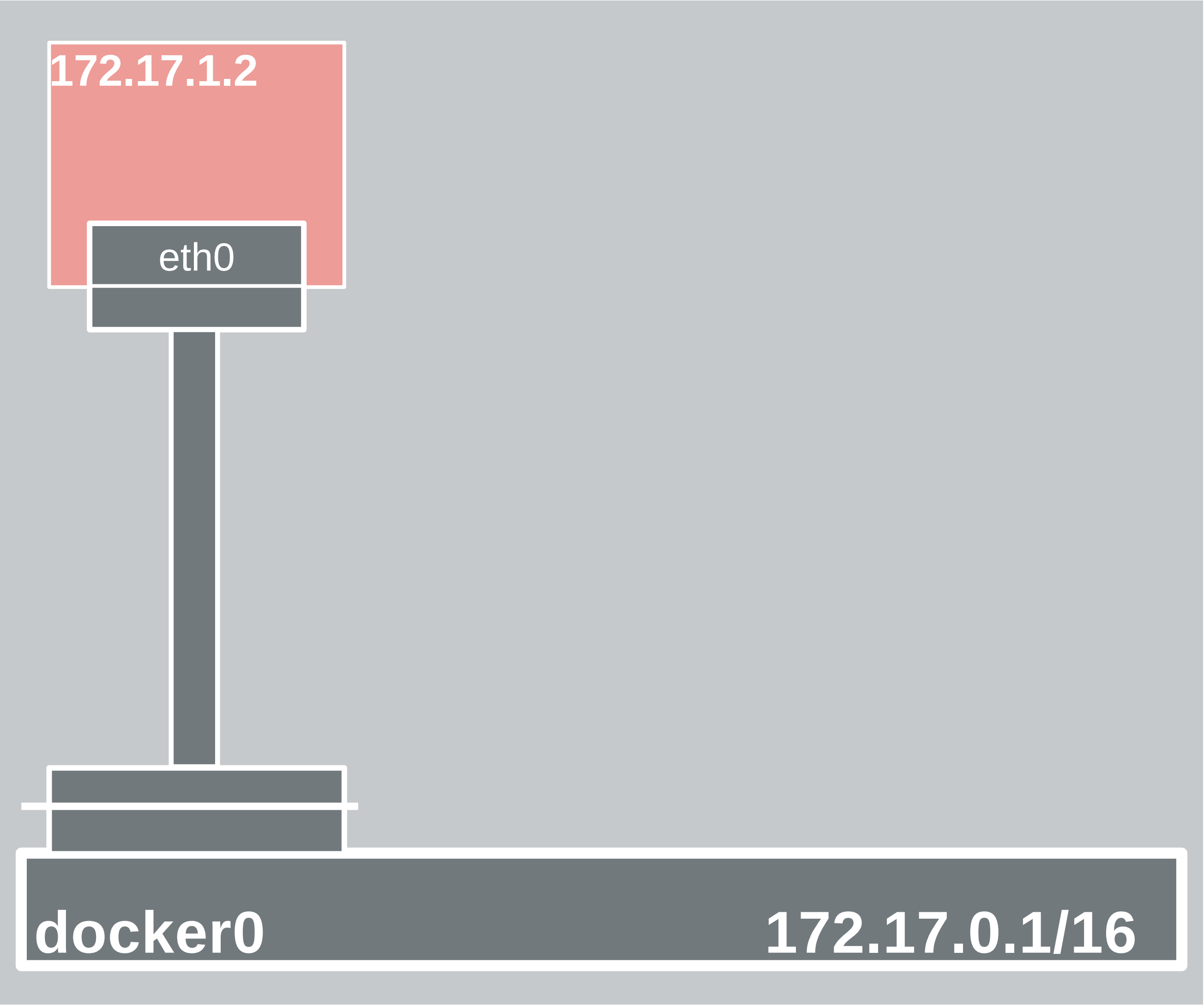
docker0 172.17.0.1/16

Docker Networking (Cont.)

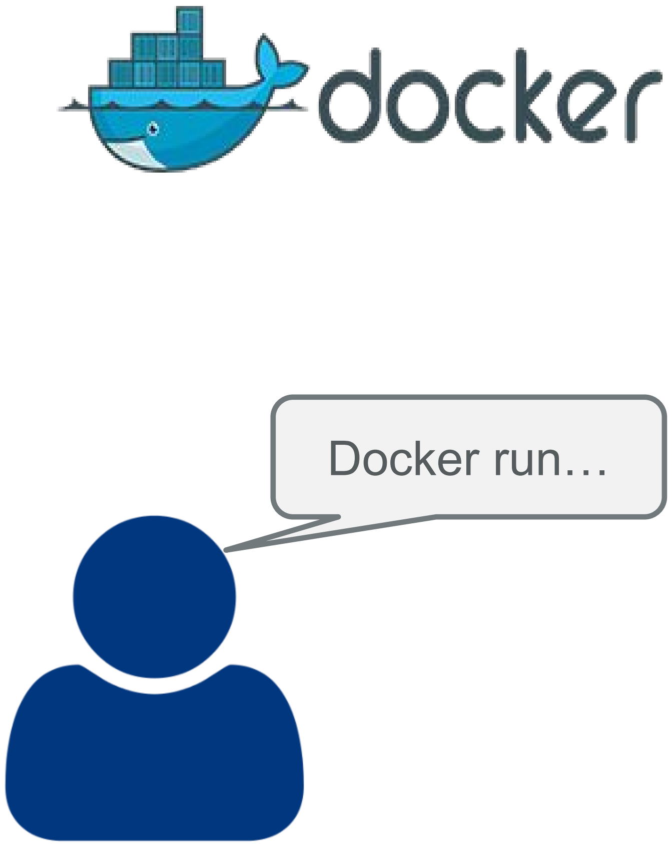


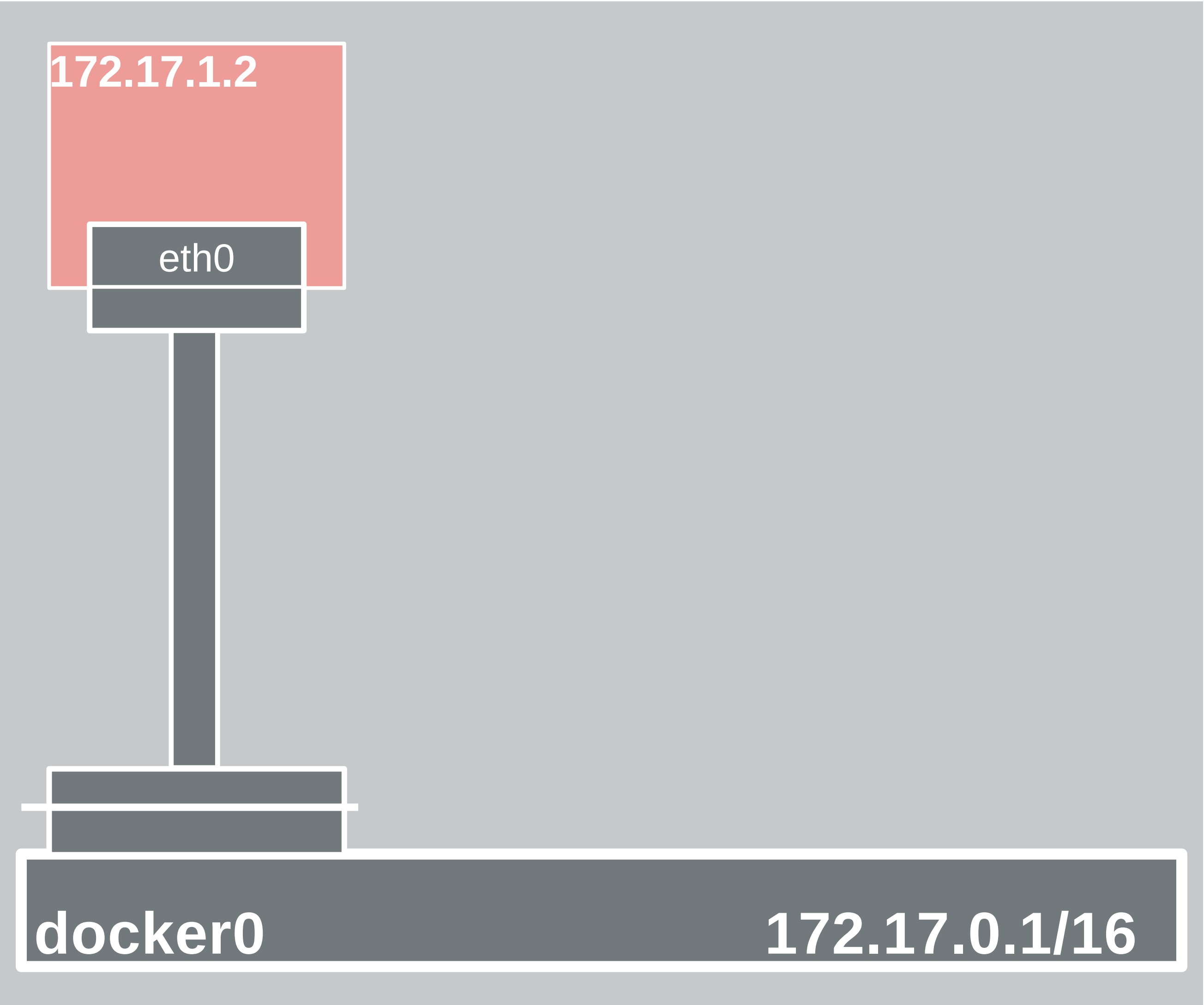
docker0 172.17.0.1/16

Docker Networking (Cont.)

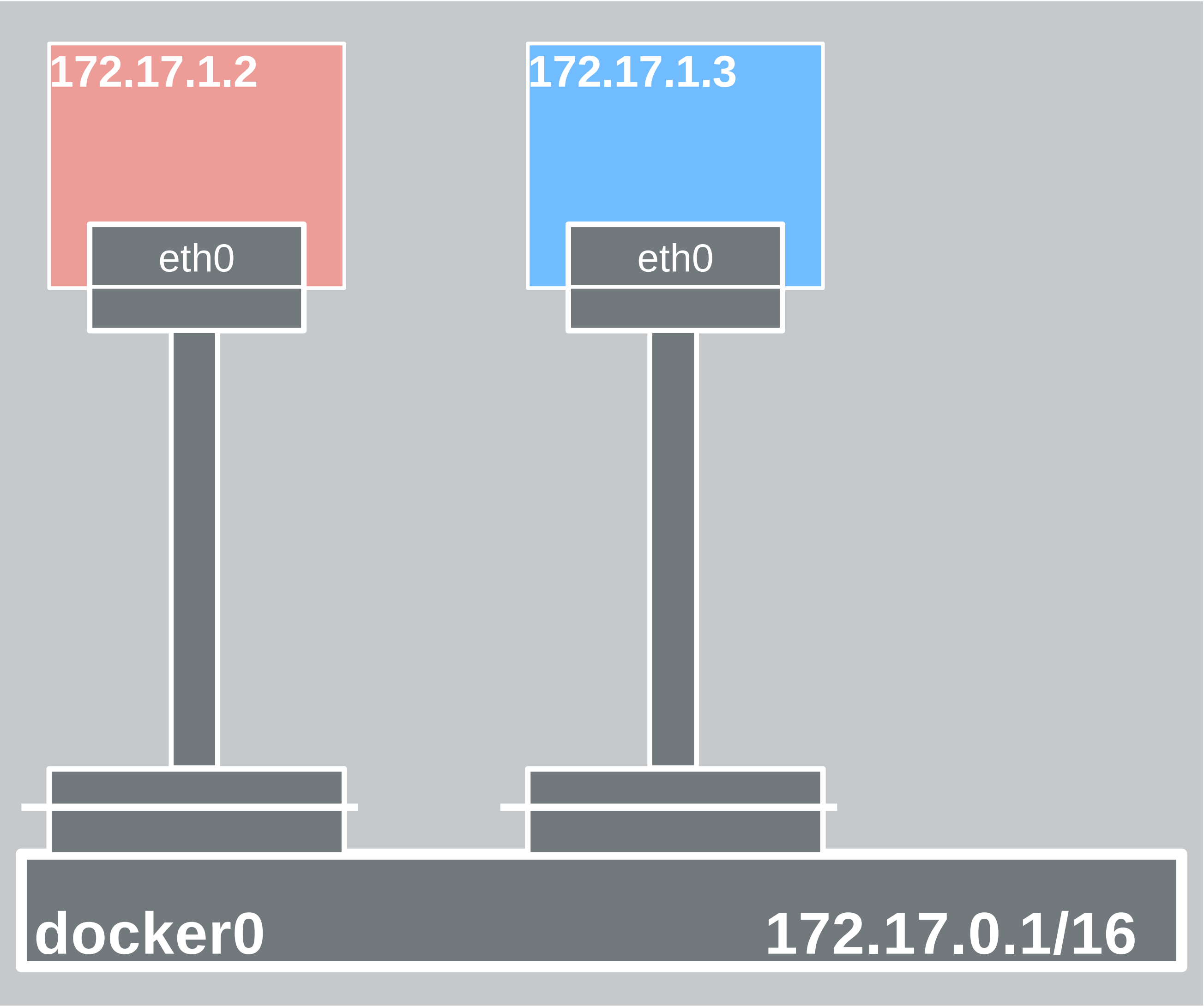


Docker Networking (Cont.)

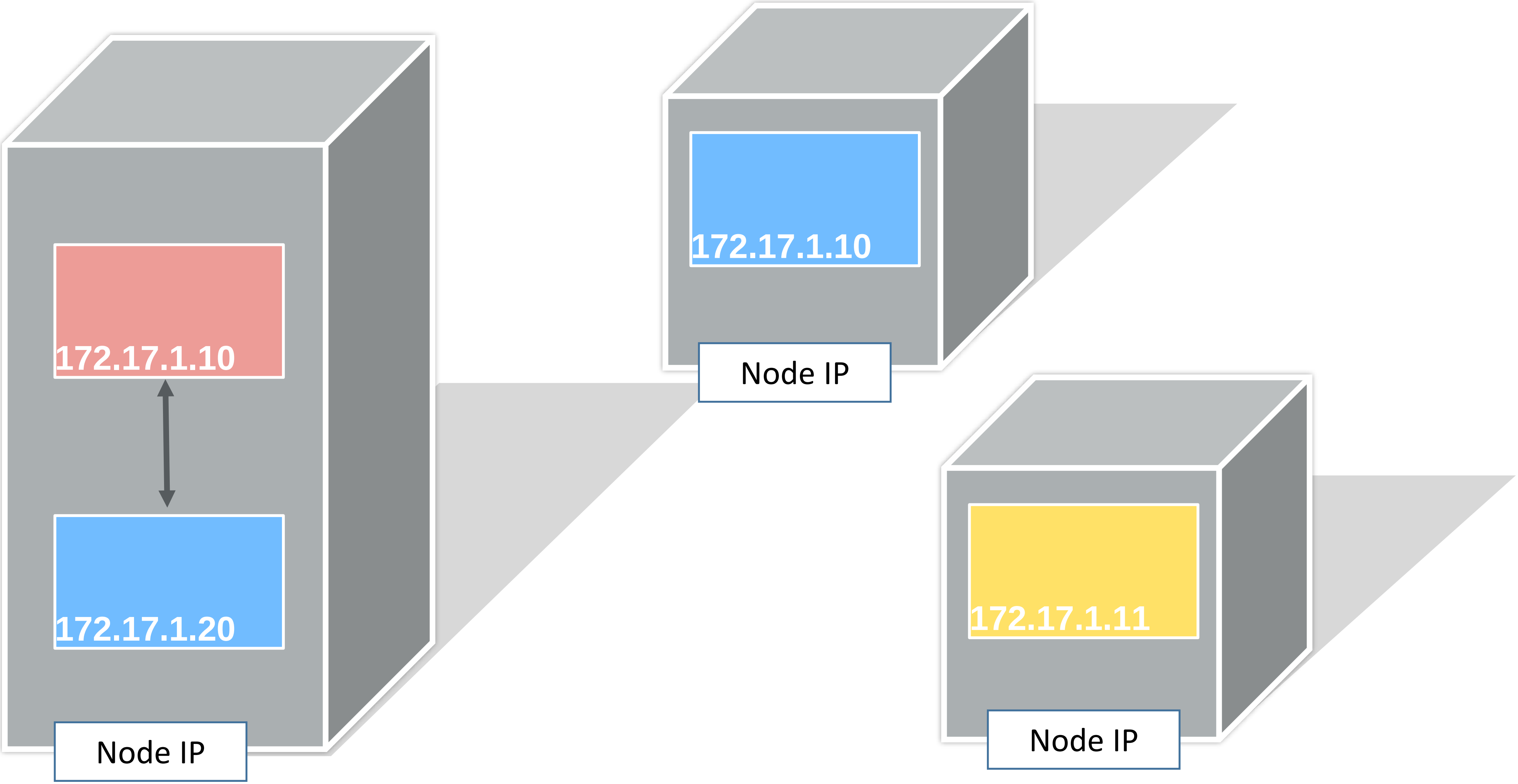




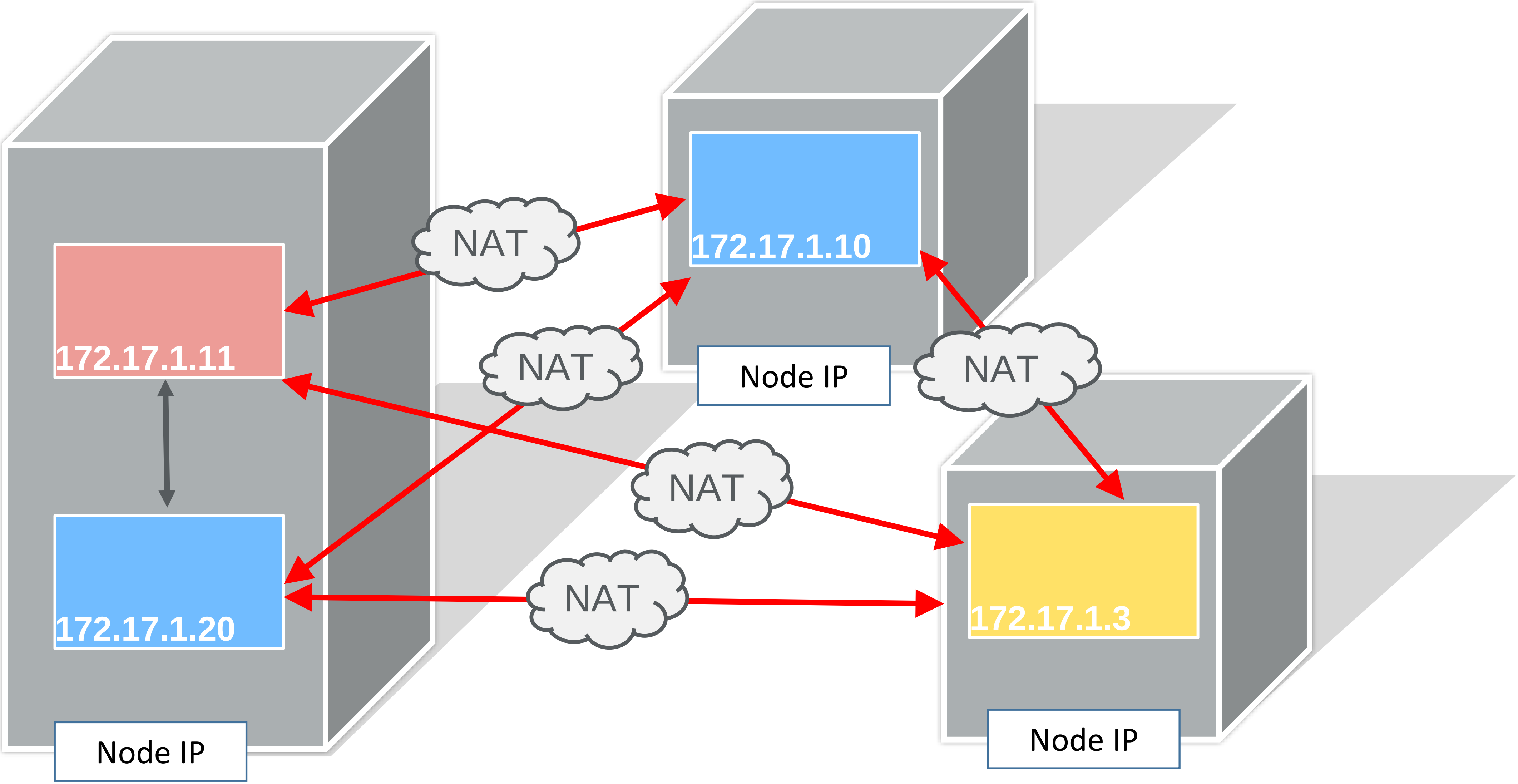
Docker Networking (Cont.)



Docker Networking (Cont.)



Docker Networking (Cont.)

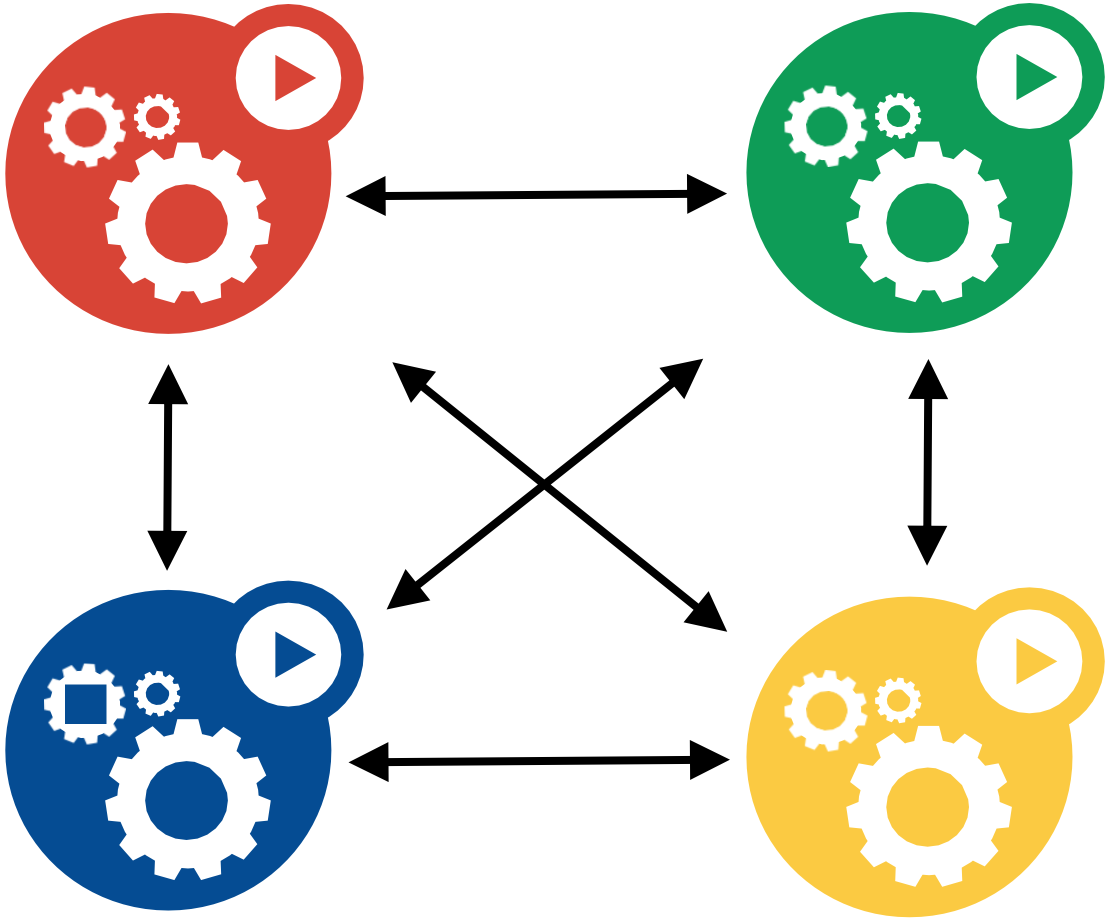


Kubernetes Networking – CNI Model

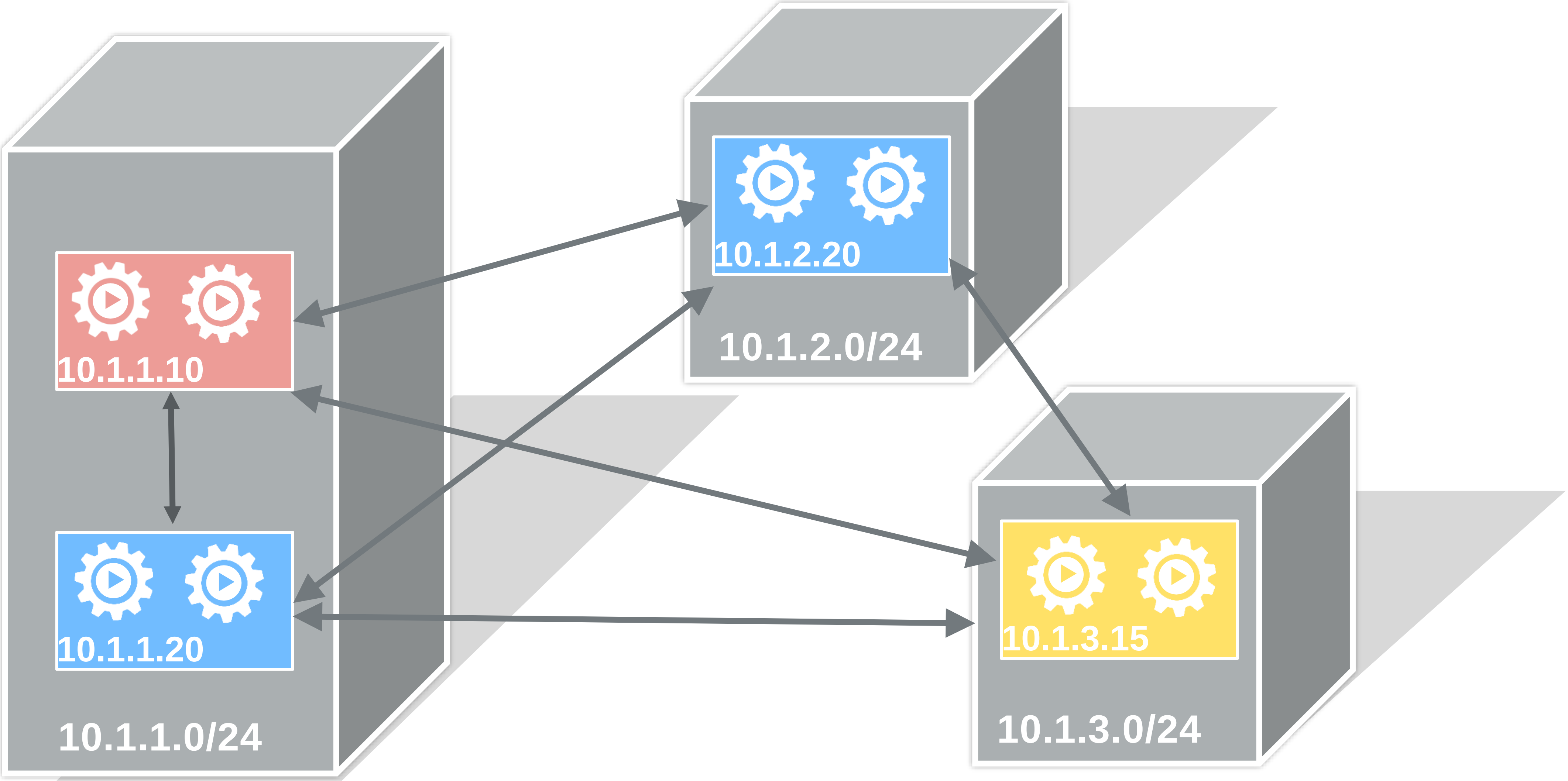
- IPs are **routable**
  - vs docker default private IP

- Pods can reach each other without NAT
  - All containers can communicate with all other containers without NAT
  - All nodes can communicate with all containers (and vice-versa) without NAT
  - The IP that a container sees itself as is the same IP that others see it as

- **This is a fundamental requirement**
  - can be L3 routed
  - can be underlayed
  - can be overlayed (SDN)



Kubernetes Networking

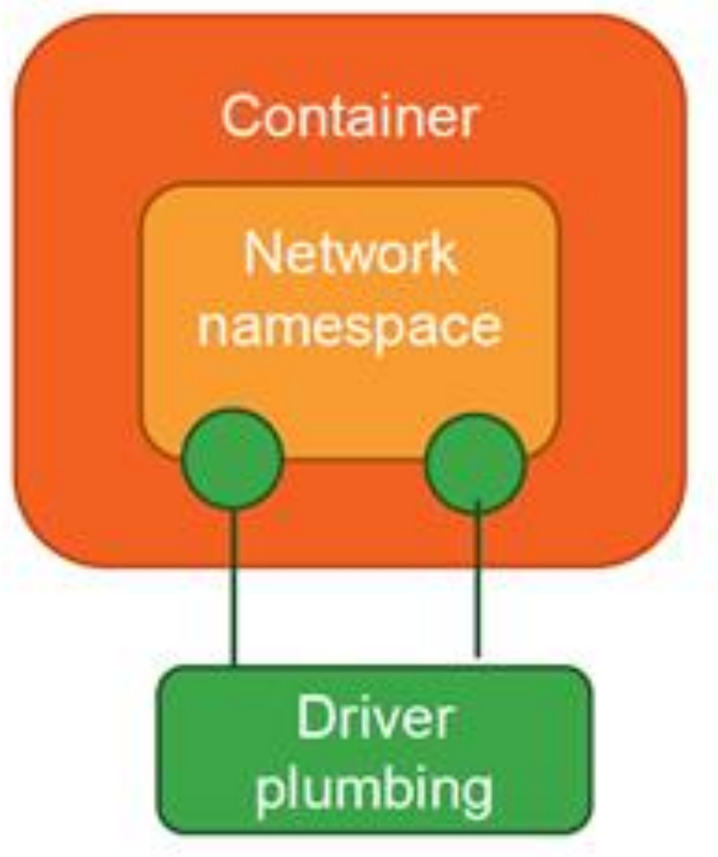


Container Network Interface - CNI

- The Container Network Interface (CNI) is a container networking specification proposed by CoreOS and adopted by projects such as Apache Mesos, Cloud Foundry, Kubernetes.

- CNI provides an interface between the network namespace and network plugin
  - Plugin is a executable that does the network plumbing
  - Allocate an IP address (by calling an IPAM plugin)

- Driver has freedom to manipulate container network namespace.



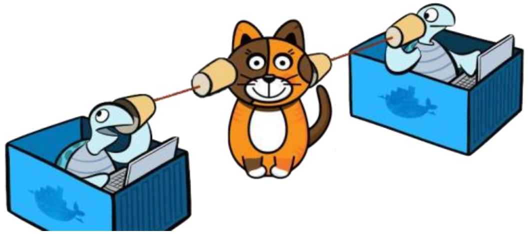
Kubernetes CNI Plugins

On GCE/GKE

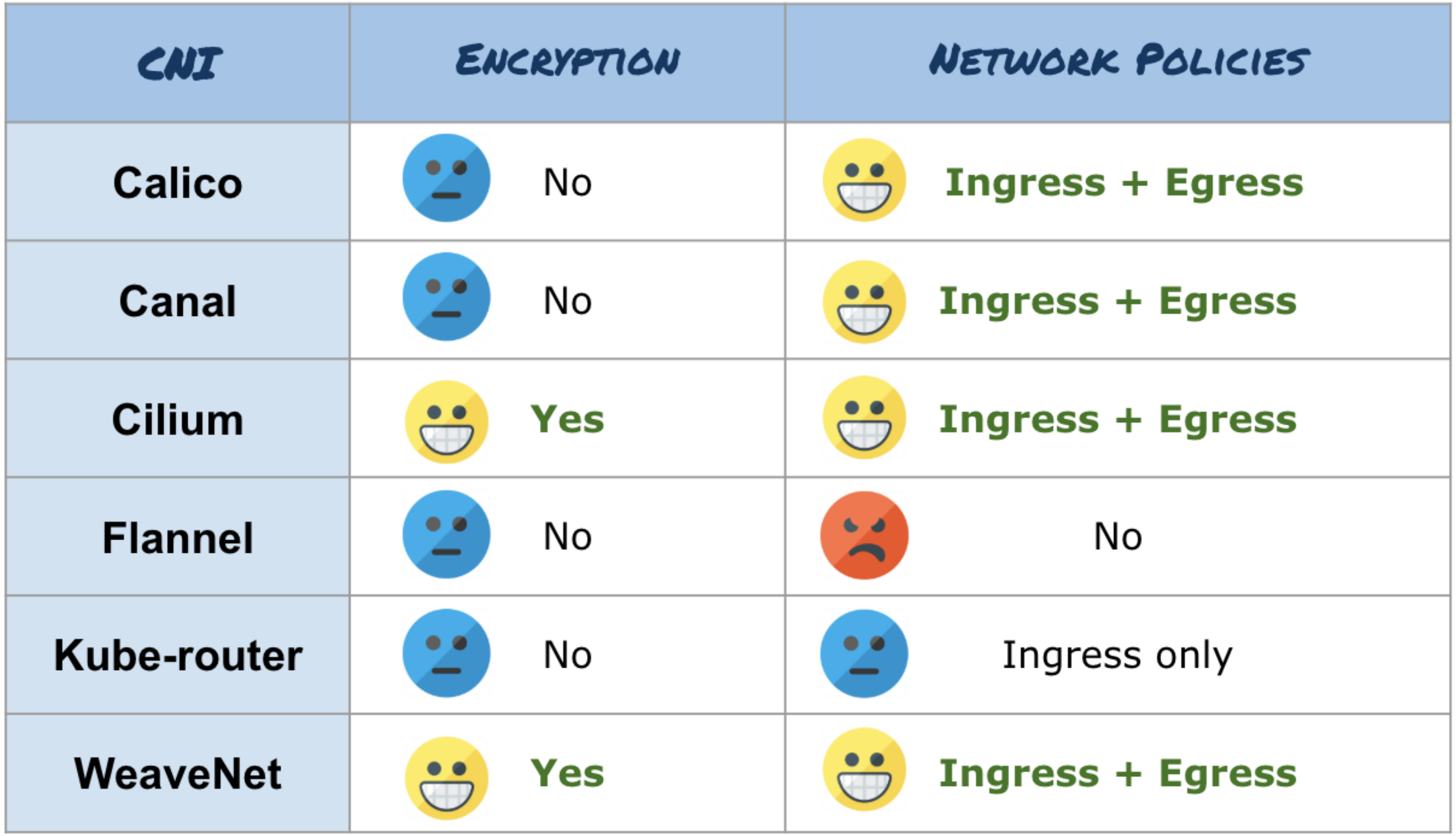
- GCE Advanced Routes (program the fabric)
- “Everything to 10.1.1.0/24, send to this host”

Plenty of other ways

- AWS: Route Tables
- Weave
- Calico
- Flannel
- OVS
- OpenContrail
- Cisco Contiv
- Others...



Kubernetes CNI Plugins Comparison



CNI	Encryption	Network Policies
Calico	😐 No	😁 Ingress + Egress
Canal	😐 No	😁 Ingress + Egress
Cilium	😁 Yes	😁 Ingress + Egress
Flannel	😐 No	😠 No
Kube-router	😐 No	😐 Ingress only
WeaveNet	😁 Yes	😁 Ingress + Egress

https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-april-2019-4a9886efe9c4

Container Networking

- Intra POD Communication
  - Inside a Pod, containers share the Network Namespaces, so that they can reach to each other via localhost.

- POD to POD communication – Within Cluster (East-West)
  - On same host
  - On different host
  - POD to a Service

- Communication Between the External World and Pods
  - Communication initiated from a POD to external world (South-North)
  - Communication initiated from external world to the Pods (North-South)

POD IP

```
## Every POD gets an IP
$ kubectl get pods -o wide
NAME                          READY   STATUS     RESTARTS   AGE         IP                   NODE
web-59765d756f-7b7tm            1/1     Running          0       14s    192.168.146.102   pod46-master.onecloud.com
```

```
## Network Namespace owned by the pause container
$ docker ps -a
8833a9b75b93       nginx                                              "nginx -g 'daemon of…"   2 minutes ago       Up 2 minutes
k8s_web_web-59765d756f-7b7tm_default_b205909c-01ae-4a49-a9b1-d8efb749035d_0
bbec0889d038       k8s.gcr.io/pause:3.1                                       "/pause"             2 minutes ago       Up 2 minutes
k8s_POD_web-59765d756f-7b7tm_default_b205909c-01ae-4a49-a9b1-d8efb749035d_0
```

Intra POD Communication

```
apiVersion: v1
kind: Pod
metadata:
  name: twocontainers
spec:
  containers:
  - name: nginx
    image: nginx
  - name: shell
    image: centos
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
```

```
$ kubectl get pods -o wide
NAME            READY   STATUS    RESTARTS   AGE     IP             NODE
twocontainers   2/2     Running      0       117s    10.233.74.22   pod48-node

$ kubectl exec twocontainers -c shell -i -t – bash

$ ip a | grep inet
inet 10.233.74.22/32

$  curl localhost
<h1>Welcome to nginx!</h1>
```

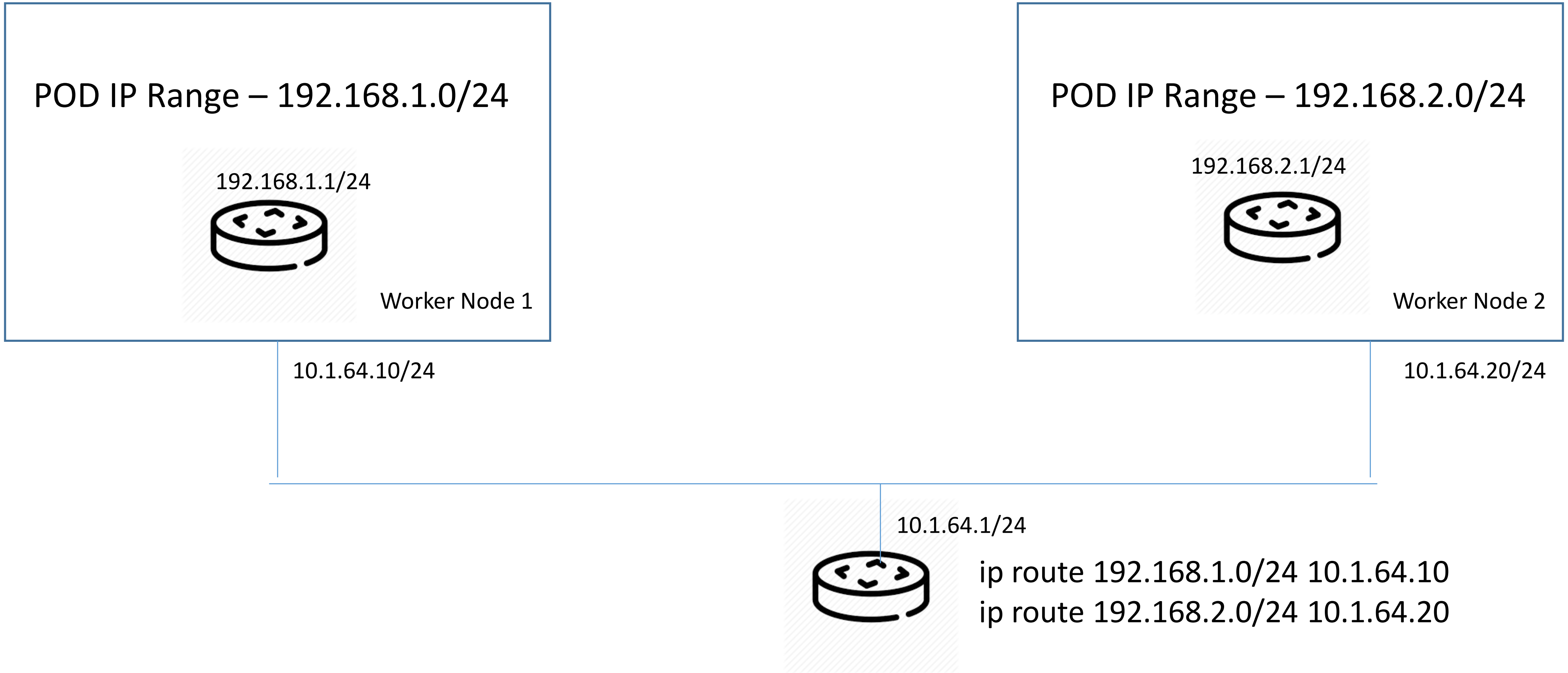
Pod To Pod Communication – Across Nodes

- Across Nodes, the CIDR allocated per Node needs reachability

- Kubernetes does not care how that is achieved
  - Routed Network
  - Bridged Network
  - Overlay Network

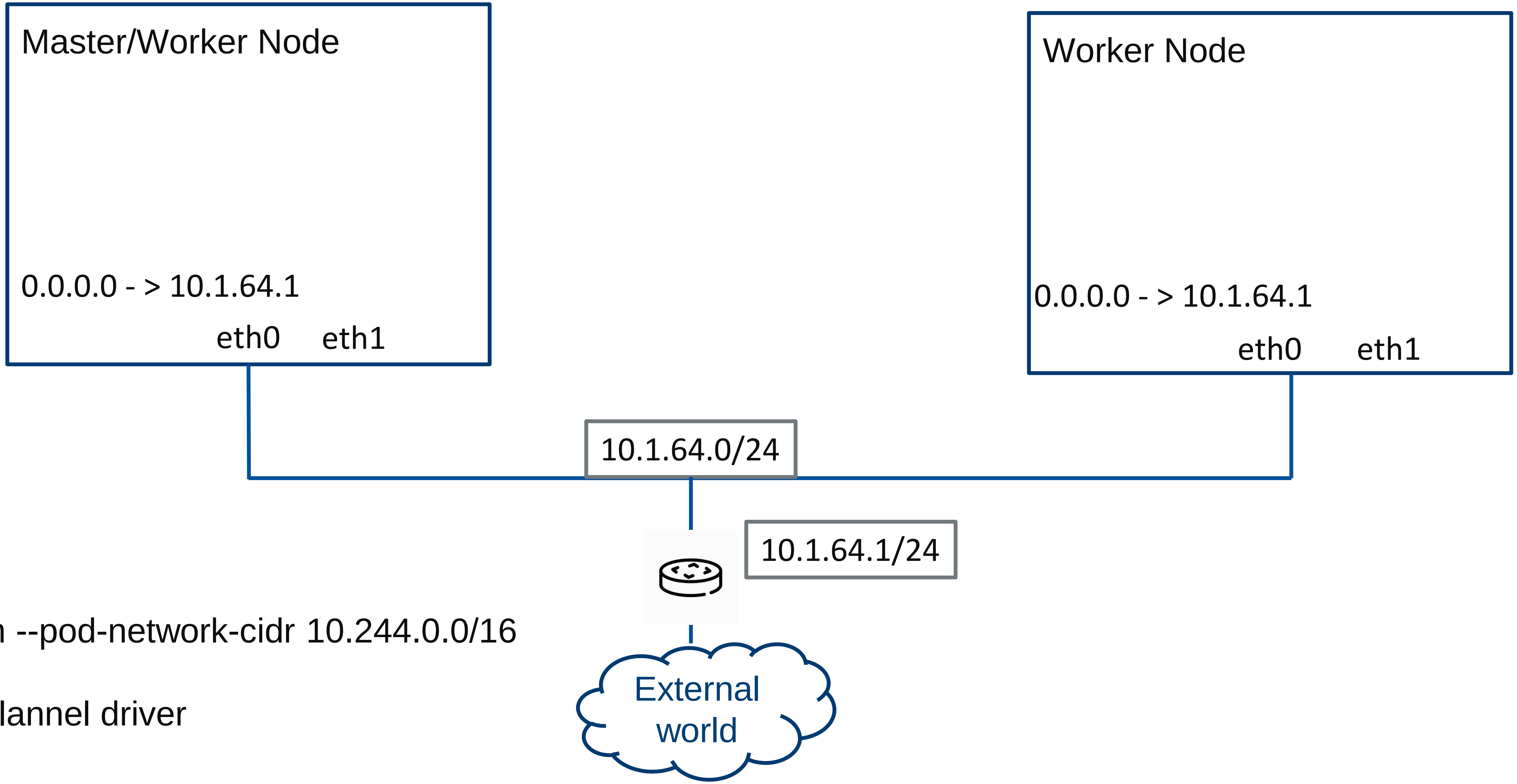
- Overlay Network Encapsulation
  - IPnIP
  - VXLAN

L3 Communication – Across Nodes



```
ip route 192.168.1.0/24 10.1.64.10
ip route 192.168.2.0/24 10.1.64.20
```

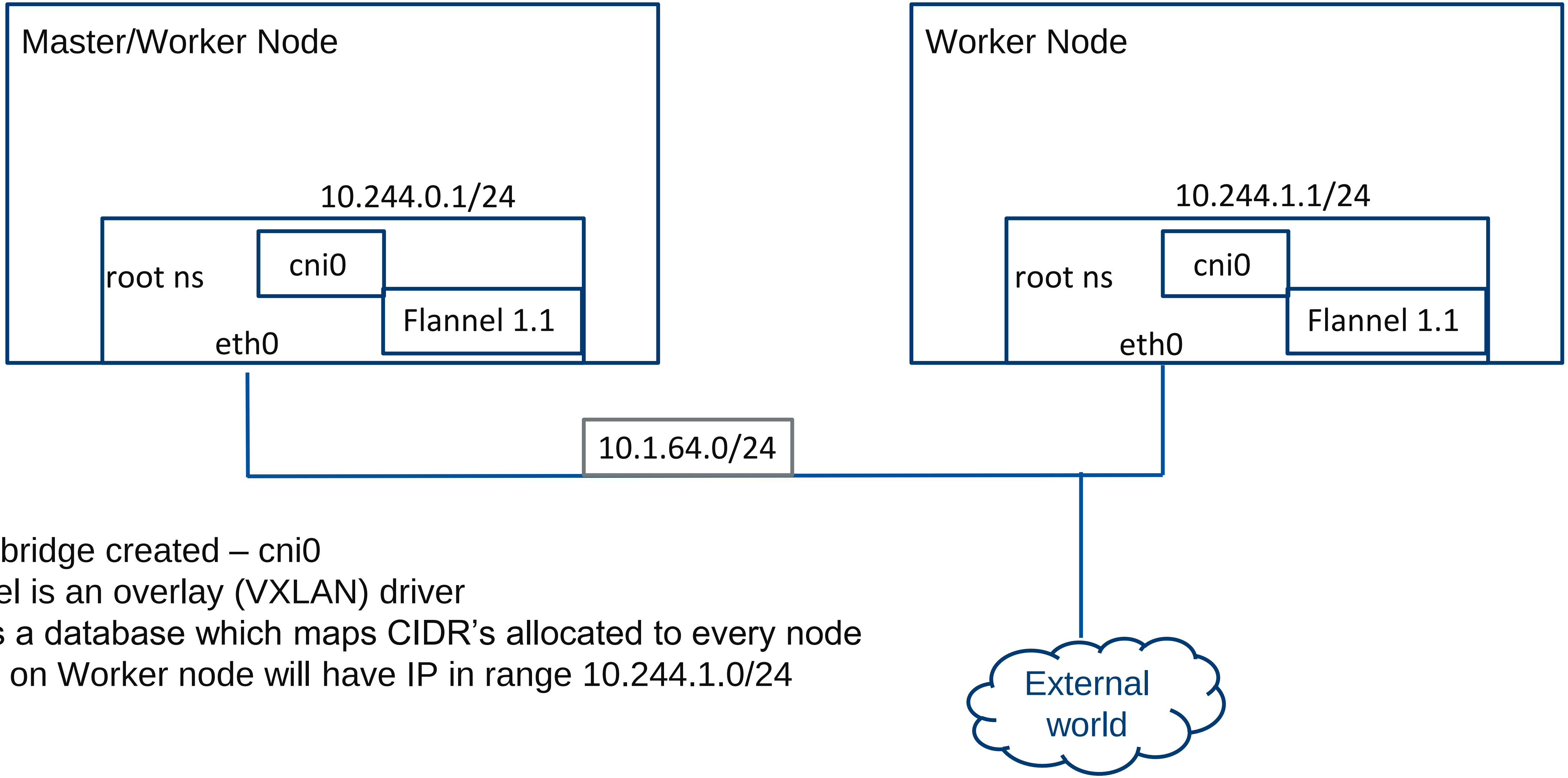
Lab Network



kubeadm --pod-network-cidr 10.244.0.0/16

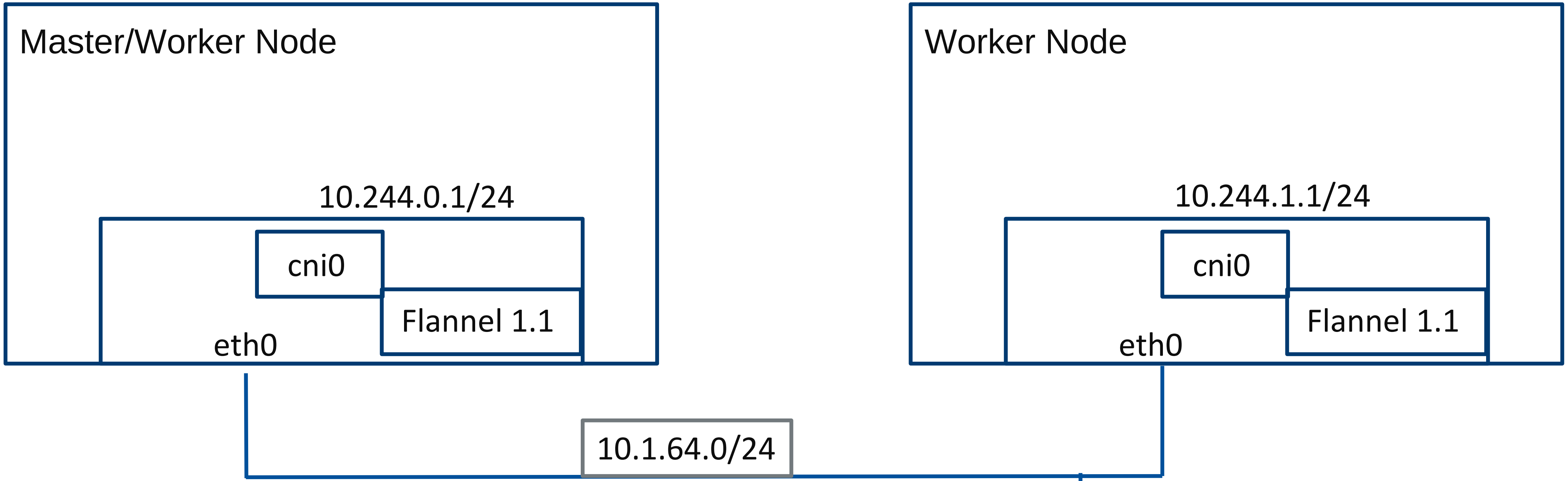
Loaded flannel driver

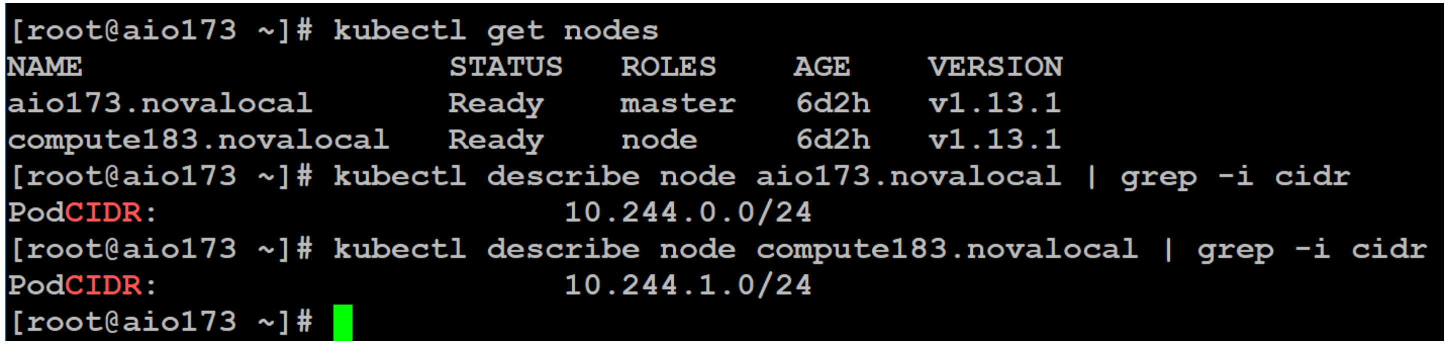
Flannel Driver Loaded



Linux bridge created – cni0
Flannel is an overlay (VXLAN) driver
Keeps a database which maps CIDR's allocated to every node
PODs on Worker node will have IP in range 10.244.1.0/24

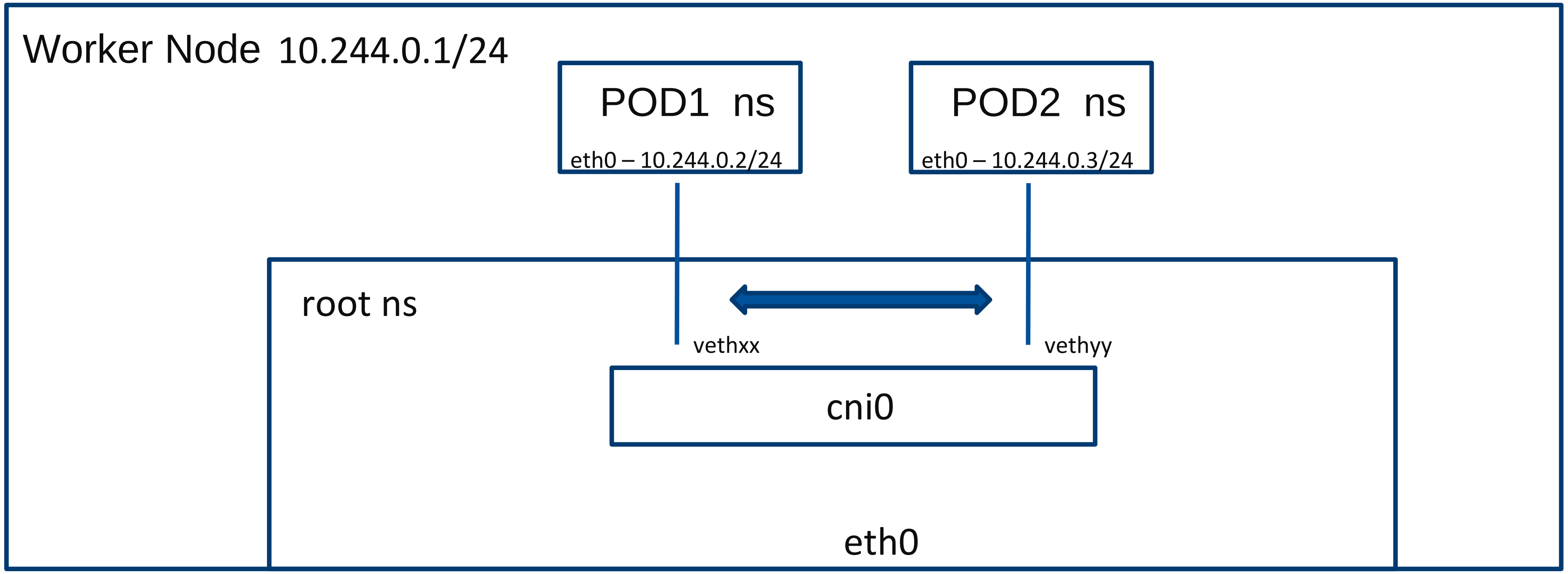
Querying Allocated CIDR per Node




```
[root@aio173 ~]# kubectl get nodes
NAME                   STATUS    ROLES     AGE     VERSION
aio173.novalocal       Ready     master    6d2h    v1.13.1
compute183.novalocal   Ready     node      6d2h    v1.13.1
[root@aio173 ~]# kubectl describe node aio173.novalocal | grep -i cidr
PodCIDR:                      10.244.0.0/24
[root@aio173 ~]# kubectl describe node compute183.novalocal | grep -i cidr
PodCIDR:                      10.244.1.0/24
[root@aio173 ~]# █
```

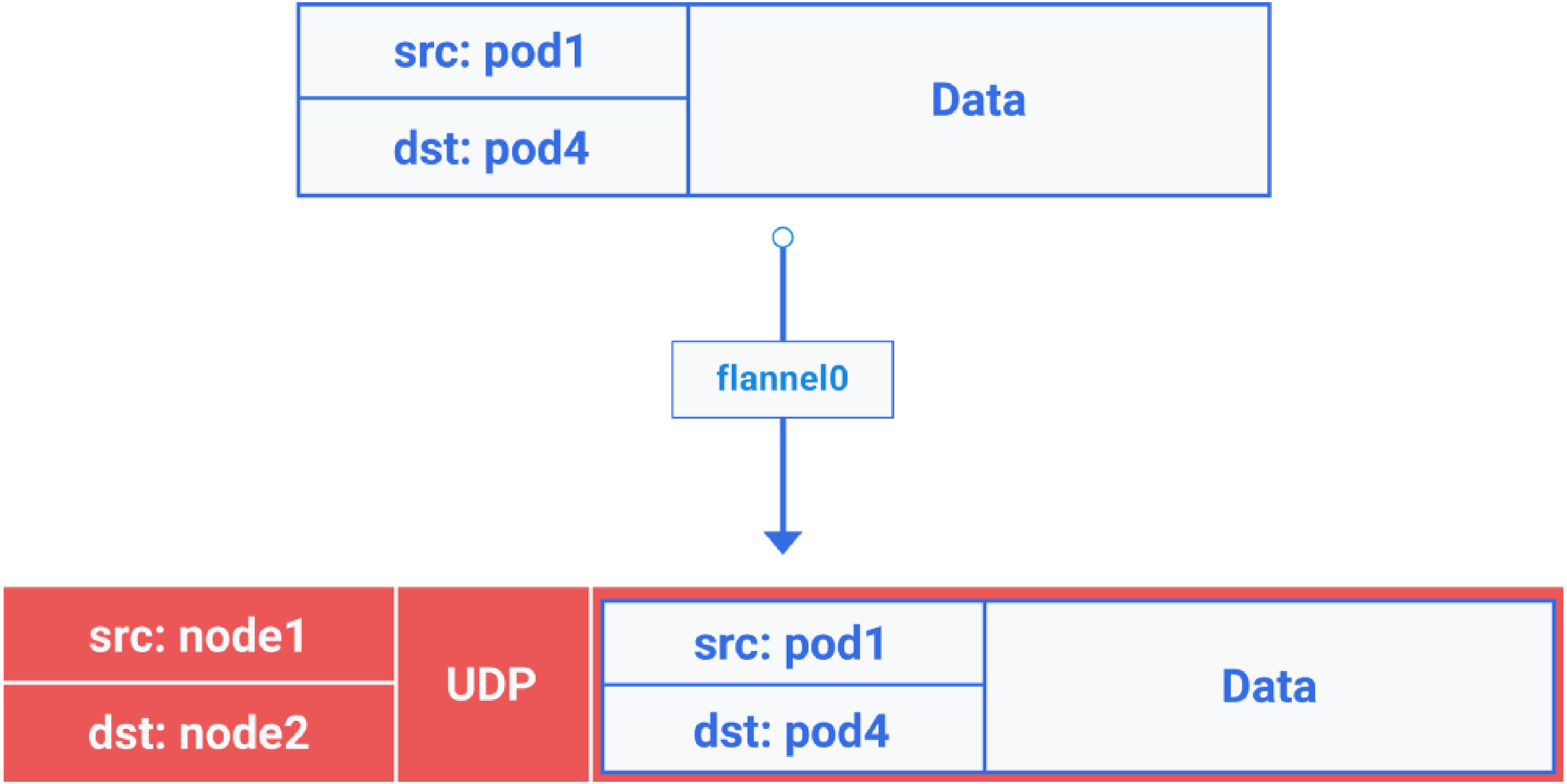
External world

Pod To Pod Communication – Same Node

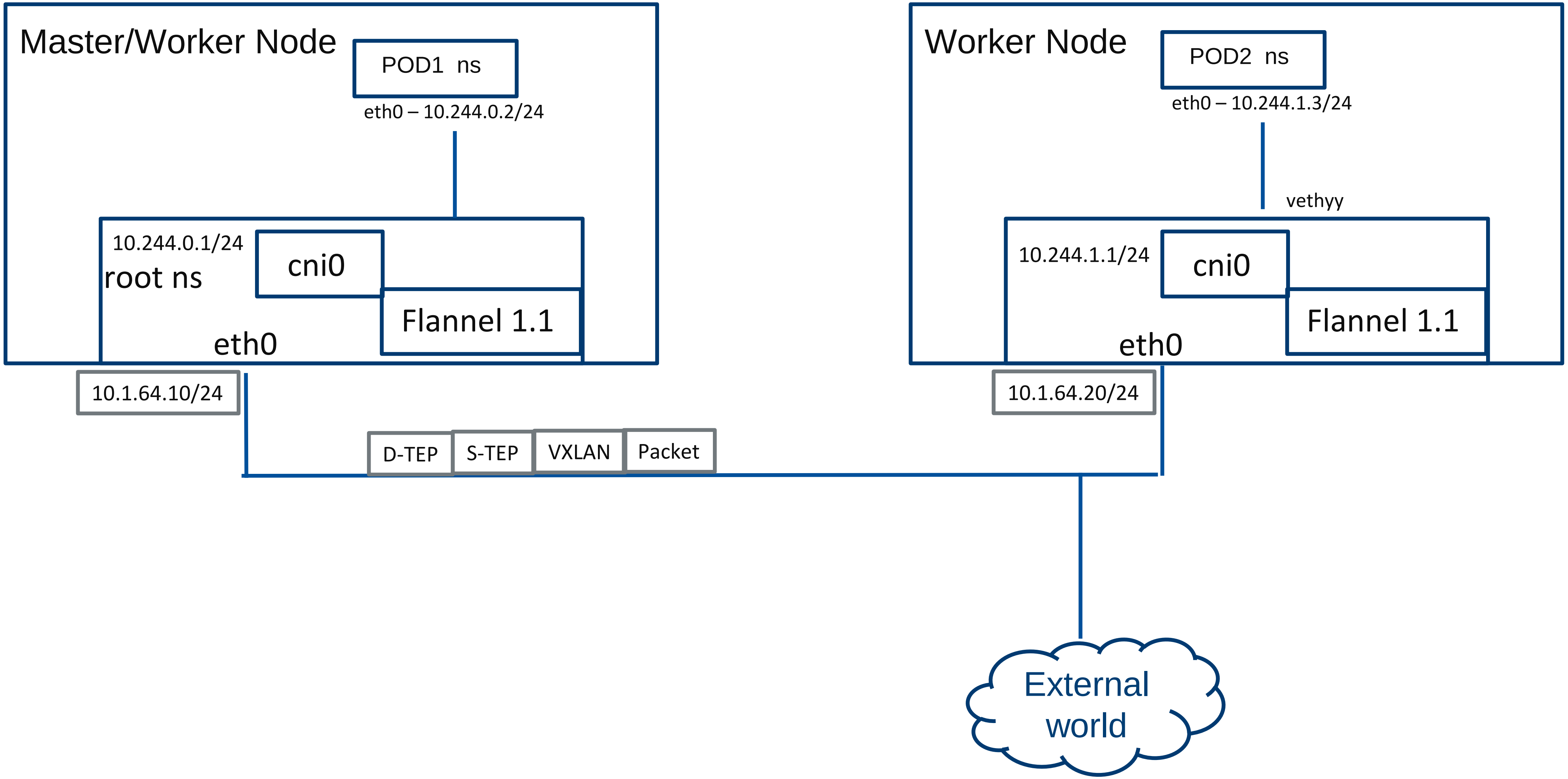


- Each POD has its own namespace
- IP allocation as per CIDR block per node
- eth0 per container piped to a veth pair
- POD1 ARPs for POD2 IP – Traffic bridges as all L2 traffic

VXLAN Data Header

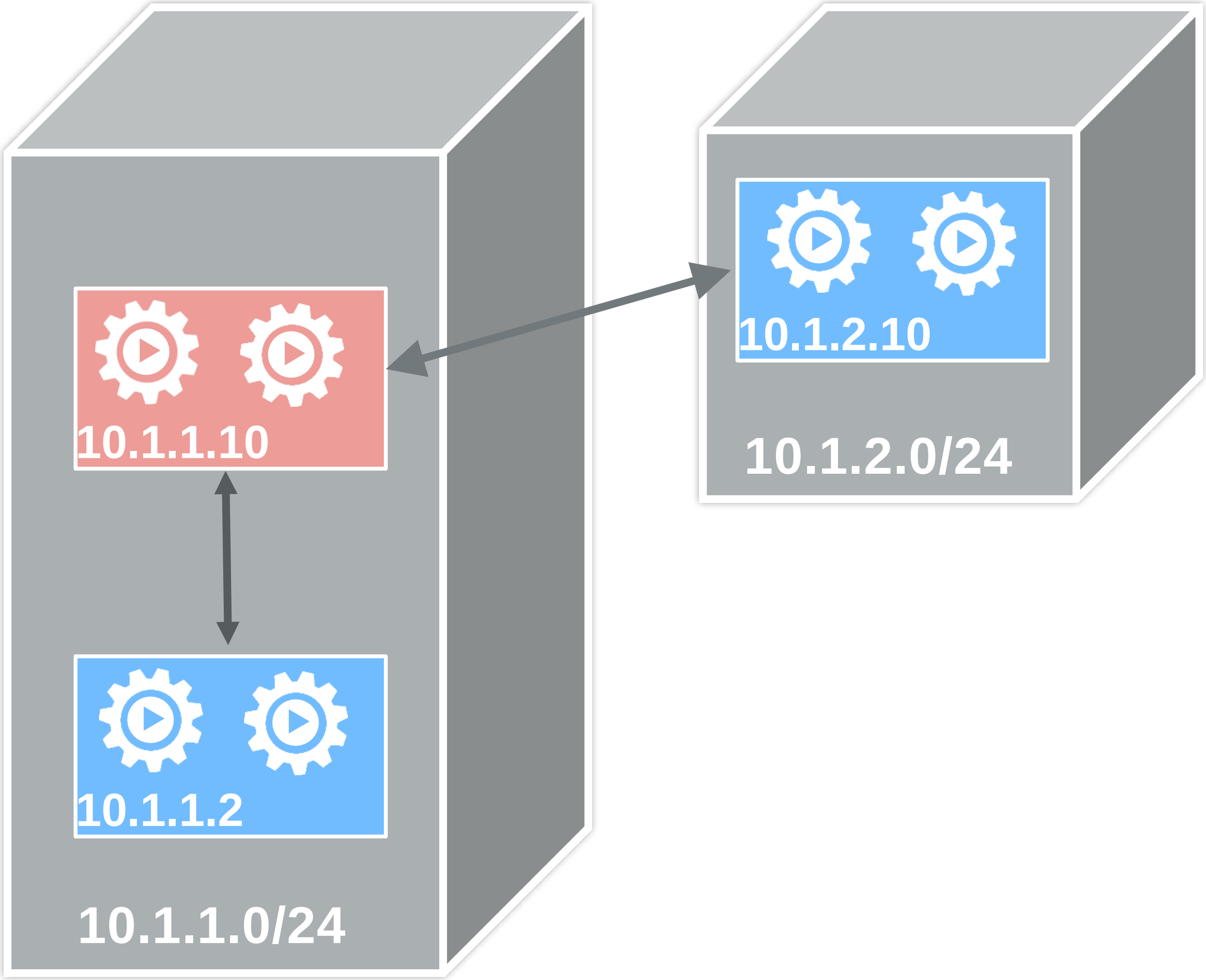


Pod To Pod Communication – Across Nodes



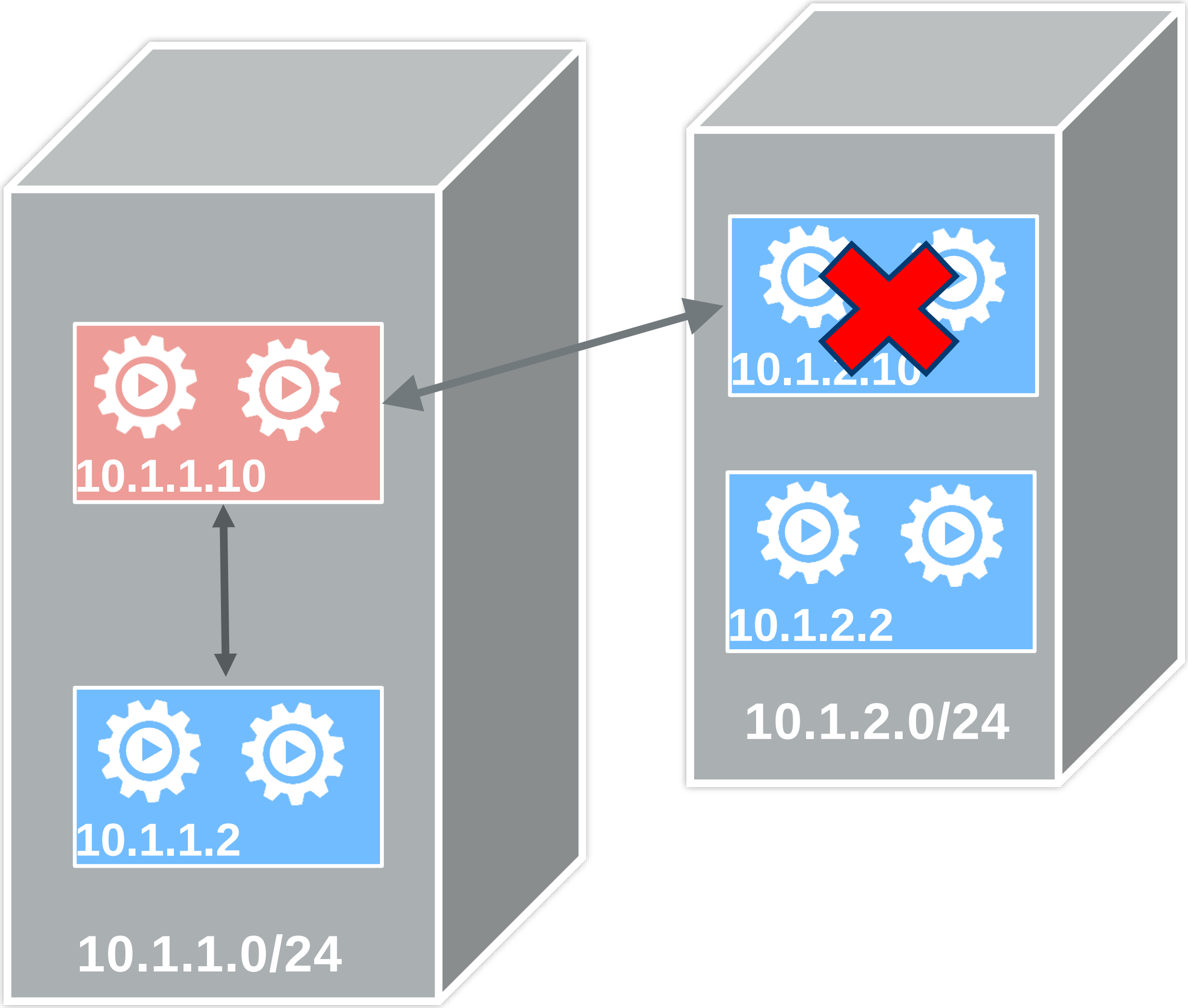
Accessing a POD

- **PODs** are ephemeral units
  - Scaled up down
  - Resources like IP addresses allocated to it cannot be static.

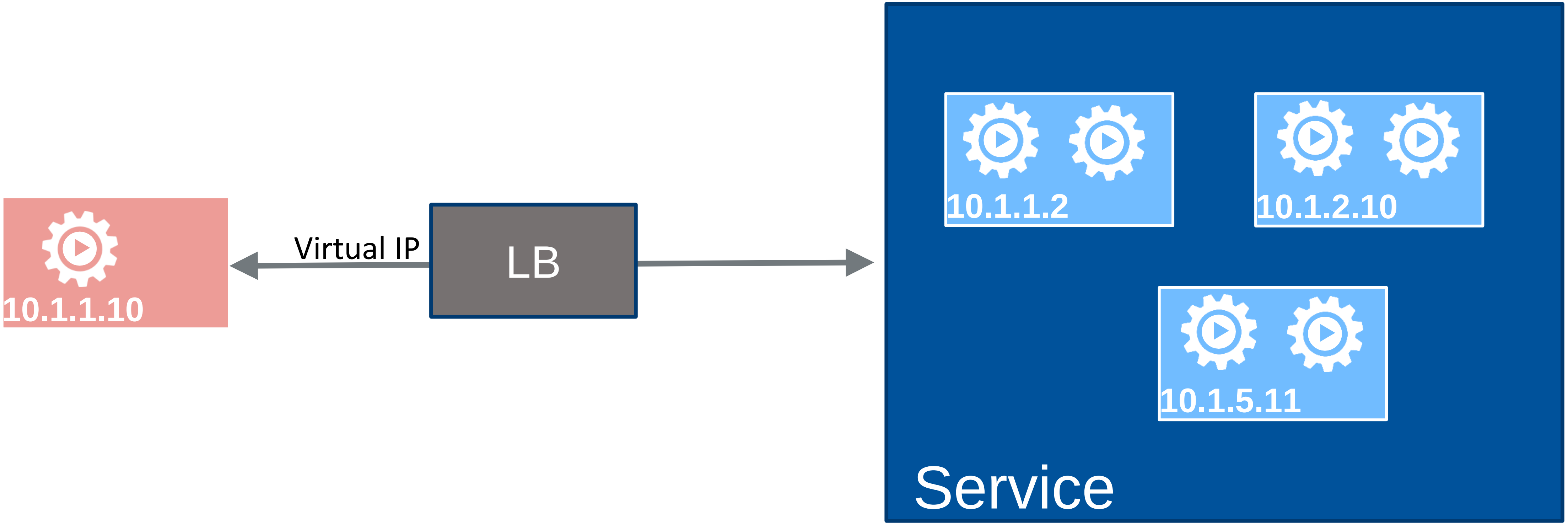


Need for a VIP

- If new PODs come up, how is it accessed?
- How does a POD speak to a group of PODs consistently?
- How are a group of Pods accessed externally?



Need for a VIP (Cont.)



- VIP provides load balancing, horizontal scaling, assistance in performing rolling updates.
- As Blue PODs change, the way the Pink POD accesses services from them does not as the VIP does not.

DNS Based Service Discovery

- The kube-dns service listens for service and endpoint events from the Kubernetes API and updates its DNS records as needed. These events are triggered when you create, update or delete Kubernetes services and their associated pods.
    - Core-DNS is default installed with kubeadm (version v1.11 and later)

- kubelet sets each new pod's /etc/resolv.conf nameserver option to the cluster IP of the kube-dns service

- Applications running in containers can then resolve hostnames such as example-service.namespace to the correct cluster IP addresses.

```
/etc/resolv.conf

nameserver 10.32.0.10
search namespace.svc.cluster.local svc.cluster.local cluster.local
```

DNS Based Service Discovery

```
# On regular PODs resolv.conf reflects the DNS entry
nameserver 10.96.0.10
search namespace.svs.cluster.local svc.cluster.local cluster.local



# DNS Pods running in kube-system namespace
$ kubectl get pods -n kube-system | grep coredns
coredns-5c98db65d4-dvq2j                    1/1     Running   1        9d
coredns-5c98db65d4-q44ns                    1/1     Running   1        9d

# Cluster IP Allocated
$  kubectl get svc -n kube-system
NAME                                        TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)                  AGE
kube-dns                                    ClusterIP   10.96.0.10      <none>        53/UDP,53/TCP,9153/TCP   9d
```

Linux iptables

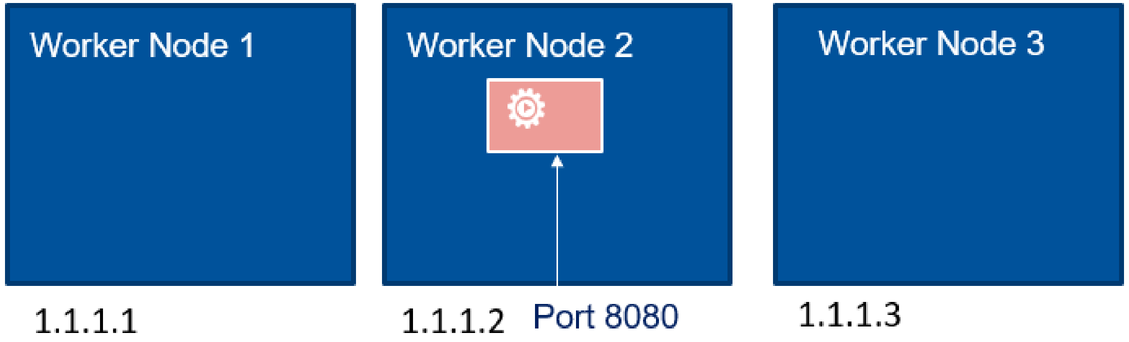
- iptables is a user-space utility program that allows a system administrator to configure the tables provided by the Linux kernel firewall (netfilter)

- Provides firewalling capability and also features like NAT and Load Balancing

- To view all iptables rules on a node, the iptables-save command can be used

Accessing a Pod from Outside Cluster HostNetwork

- Use the host networking for the POD to be reachable via the host IP.
- POD can directly see the network interfaces of the host
- POD is reachable fron external world on HostIP:Port



```
apiVersion: v1
kind: Pod
metadata:
  name: nodejs
  namespace:
  labels:
spec:
  containers:
  - name: nodejs
    image: nodejs
    ports:
    - containerPort: 8080
  hostNetwork: true

$ kubectl get pods -o wide
NAME      IP
nodejs    10.1.64.226
```

Services

- A **Kubernetes Service** is an abstraction which defines a logical set of Pods and a policy by which to access them- referred to as a microservice.

- The set of Pods targeted by a Service is determined by Label Selectors.

- Services are the primary mode of communication in Kubernetes

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

Port mapping possible

Service Types

- Services are the primary mode of communication in Kubernetes.

- ServiceType:
  - Is only accessible within the cluster (East-West)
  - Is accessible from within the cluster and the external world
  - Maps to an external entity which resides outside the cluster

ClusterIP Service Type

- Primarily for East-West load balancing
- Exposes the service on a cluster-internal IP
  - 10.96.0.0/12 is the default range
  - 1 IP per Service (ClusterIP)
- Could be backed by 1 or more PODs
- This is the **default** ServiceType

```
kubectl expose deploy/nginx --port=80 \
--target-port=80 --name= my-internal-service
```

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

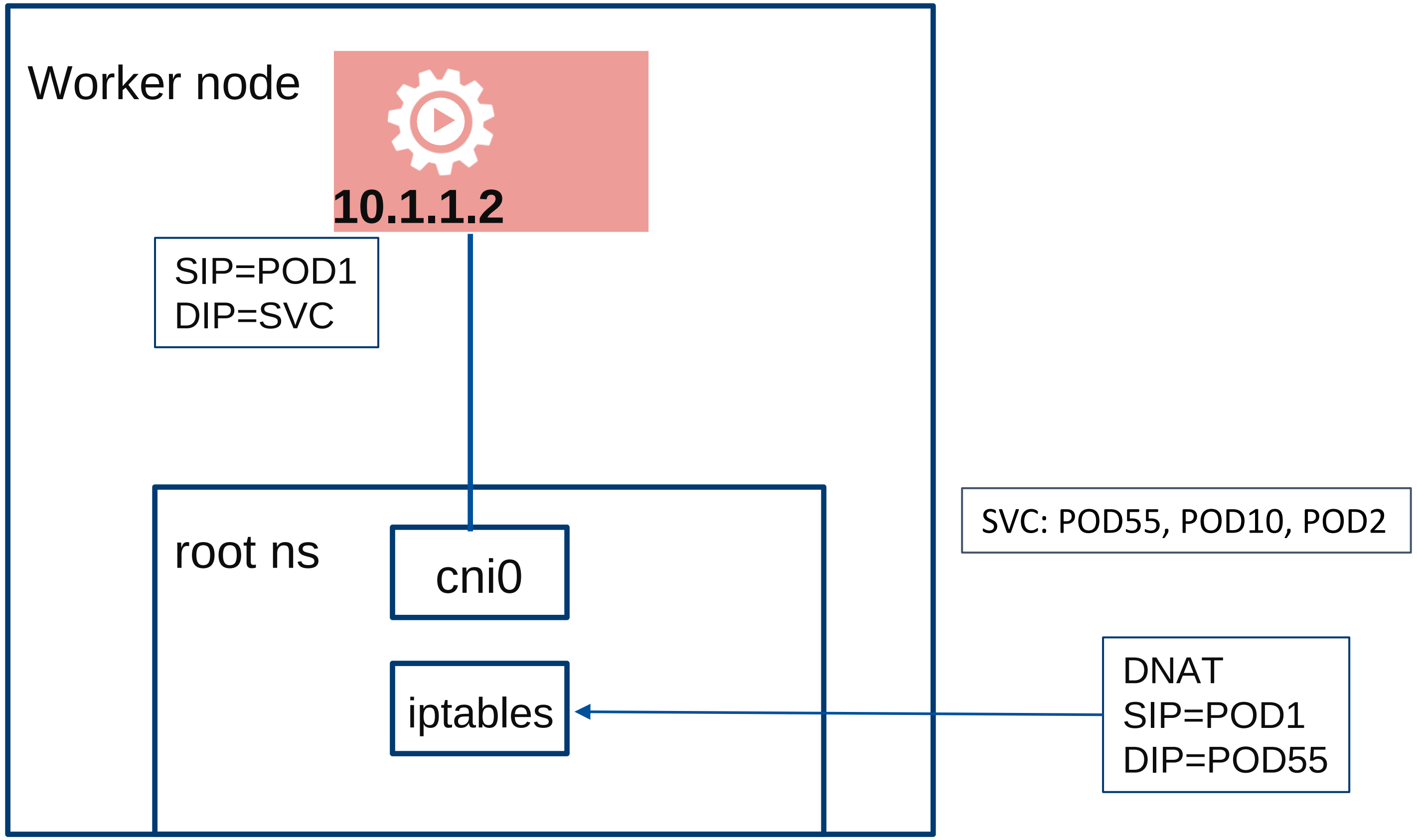
```
$ kubectl get endpoints my-internal-service
NAME          ENDPOINTS                                      AGE
my-internal-service
10.244.0.36:80,10.244.1.71:80,10.244.1.72:80   19s
```

Cluster IP

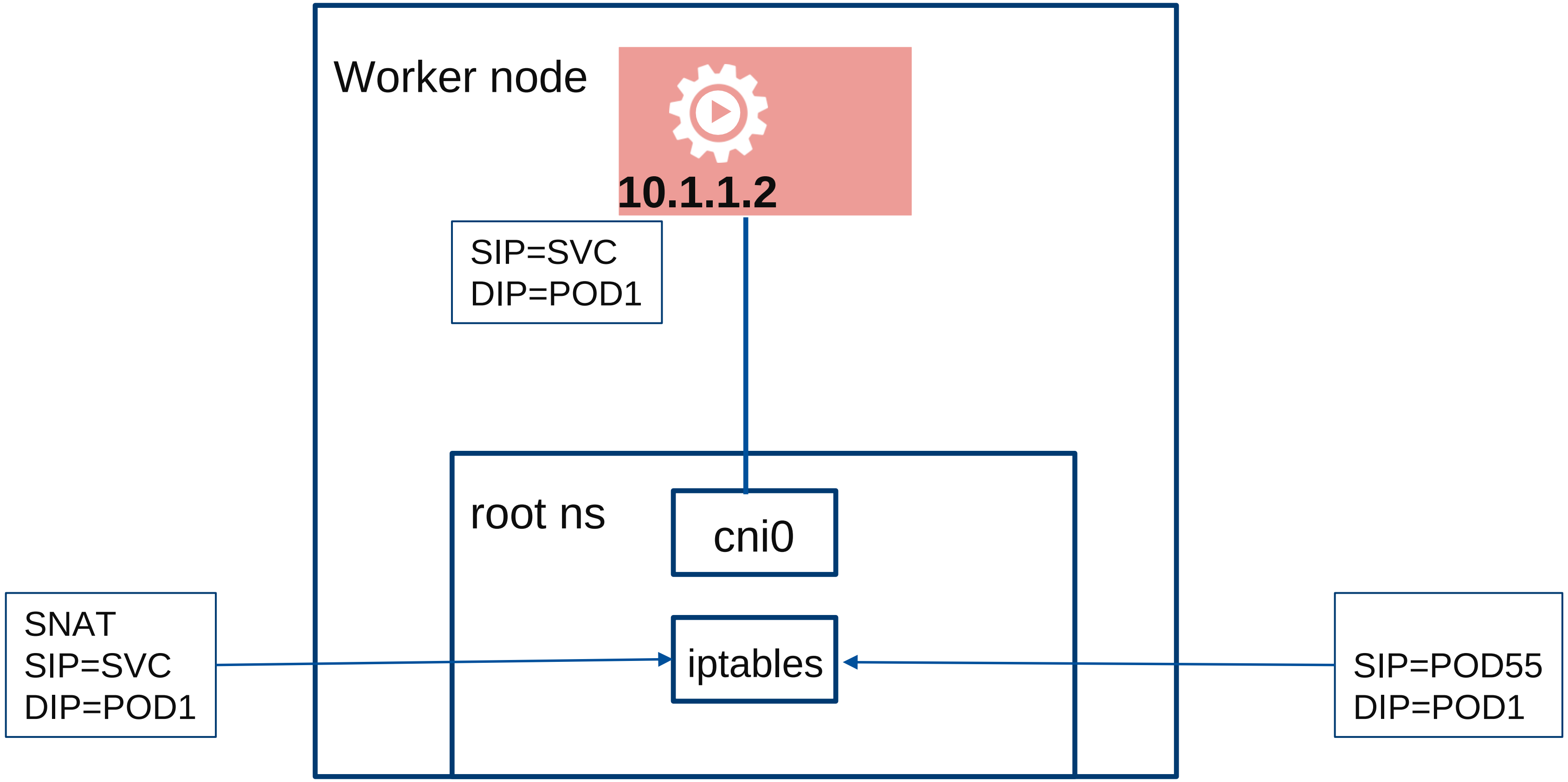
```
$ kubectl get svc
NAME          TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
nginx-service   ClusterIP   10.109.161.179   <none>       80/TCP    3m1s

$ kubectl describe svc nginx-service
Name:             nginx-service
Namespace:        default
Labels:           run=nginx
Annotations:      <none>
Selector:         run=nginx
Type:             ClusterIP
IP:               10.109.161.179
Port:             <unset>  80/TCP
TargetPort:       80/TCP
Endpoints:        10.244.0.36:80,10.244.1.71:80,10.244.1.72:80
Session Affinity:  None
Events:           <none>
```

POD to Cluster IP

- Communication between api-server and kube-proxy on every node.
- kube-proxy running on every node changes iptables (NAT) rules
- Round-robin balancing to individual PODs



Cluster IP to POD



Worker node

10.1.1.2

SIP=SVC
DIP=POD1

root ns

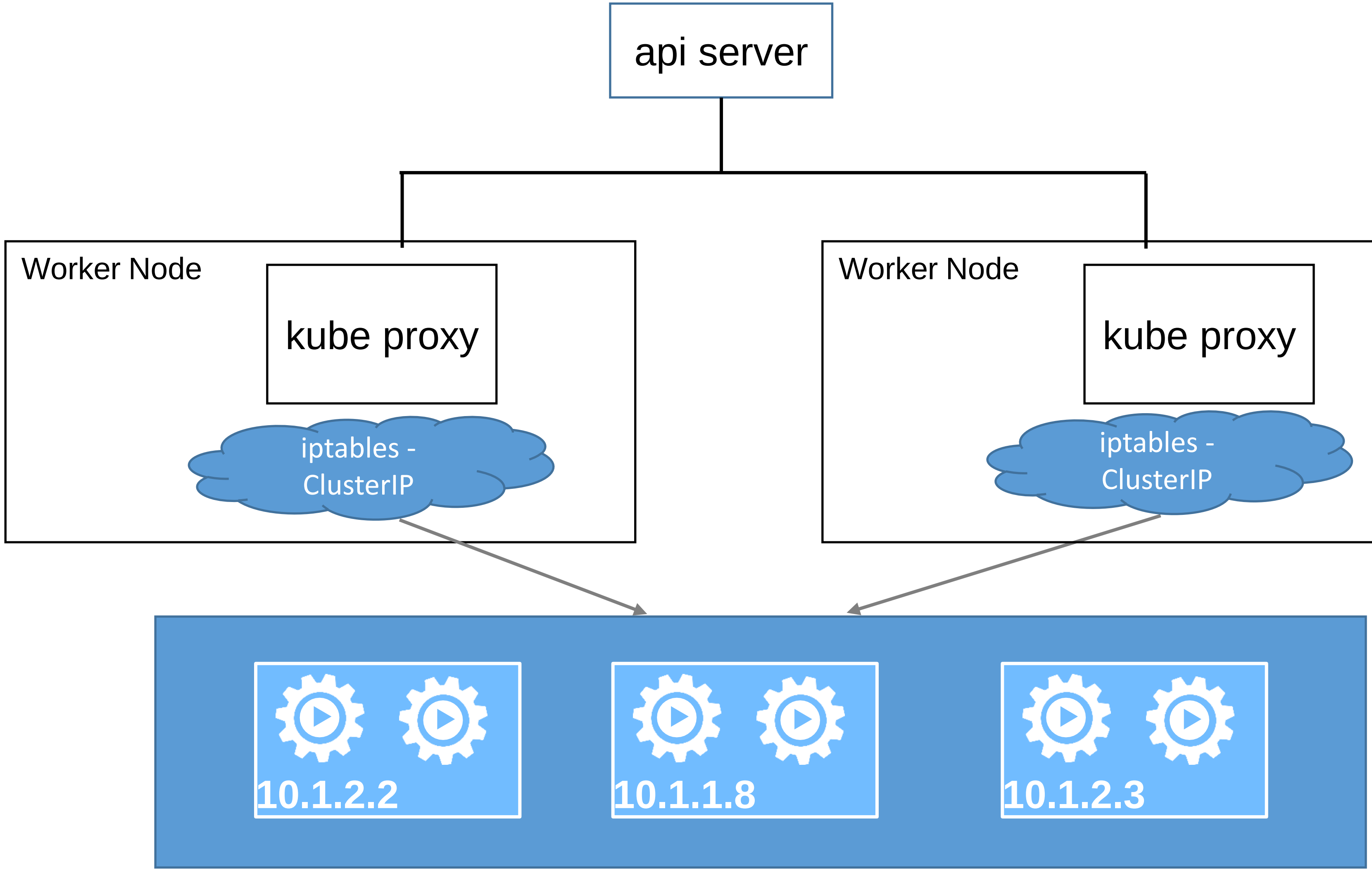
cni0

iptables

SNAT
SIP=SVC
DIP=POD1

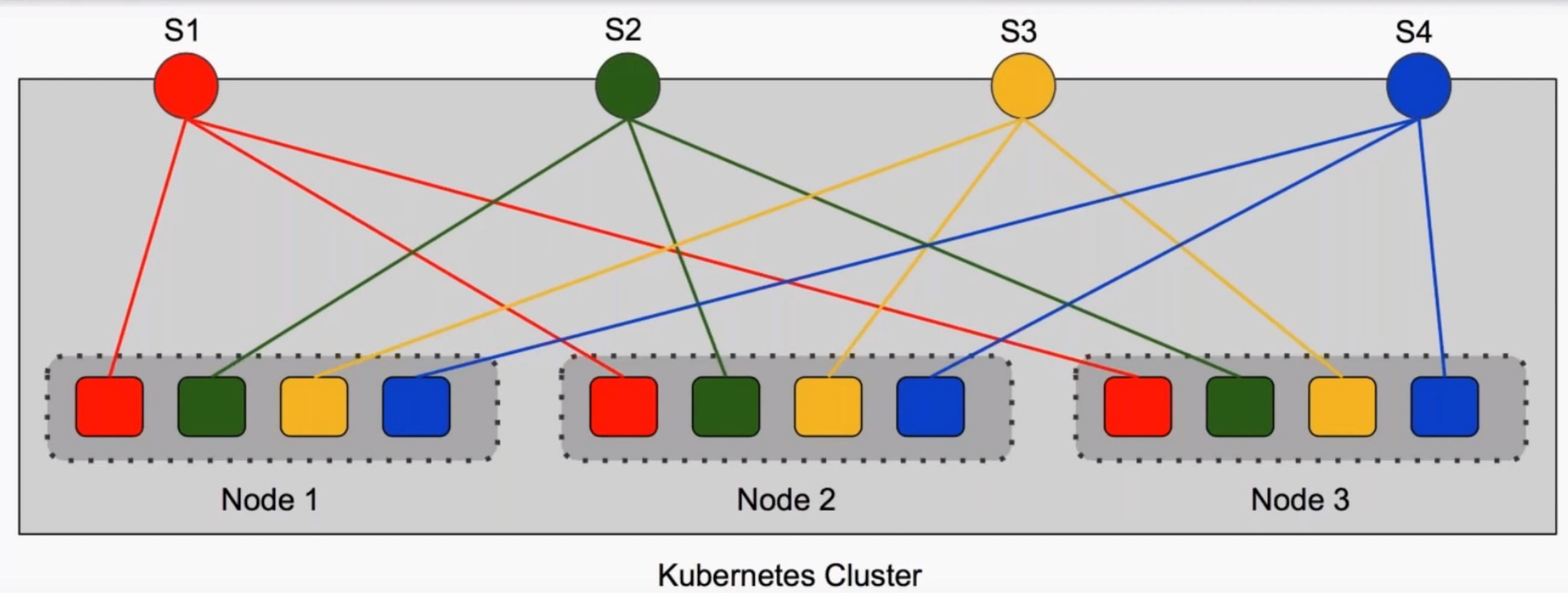
SIP=POD55
DIP=POD1

Cluster IP

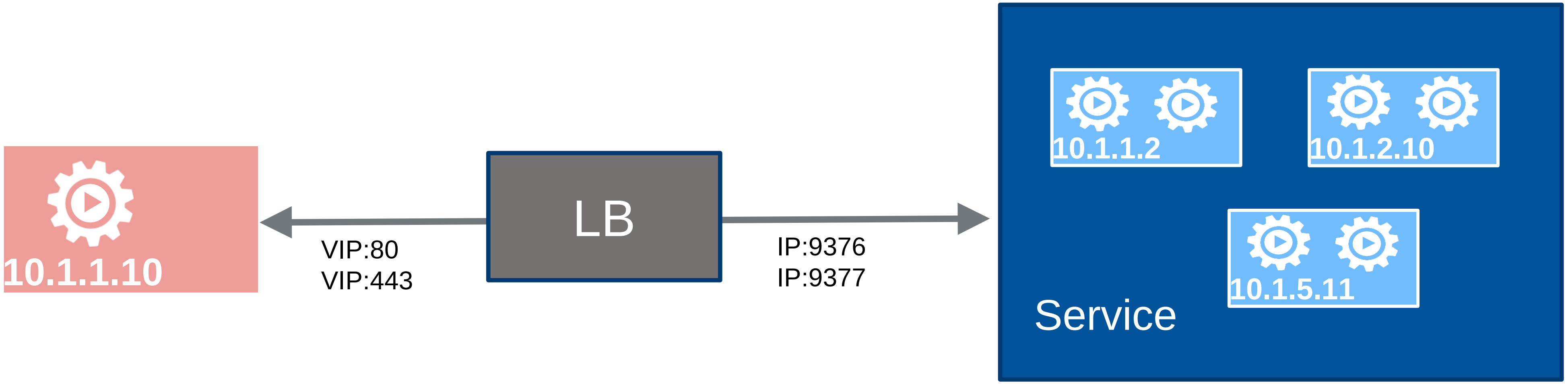

api server
Worker Node
kube proxy
iptables - ClusterIP
Worker Node
kube proxy
iptables - ClusterIP
10.1.2.2
10.1.1.8
10.1.2.3

Services

An IP per ClusterIP – 10.96.0.0/12 range



MultiPort Services



```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
    selector:
      color: blue
    ports:
      - name: http
        protocol: TCP
        port: 80
        targetPort: 9376
      - name: https
        protocol: TCP
        port: 443
        targetPort: 9377
```

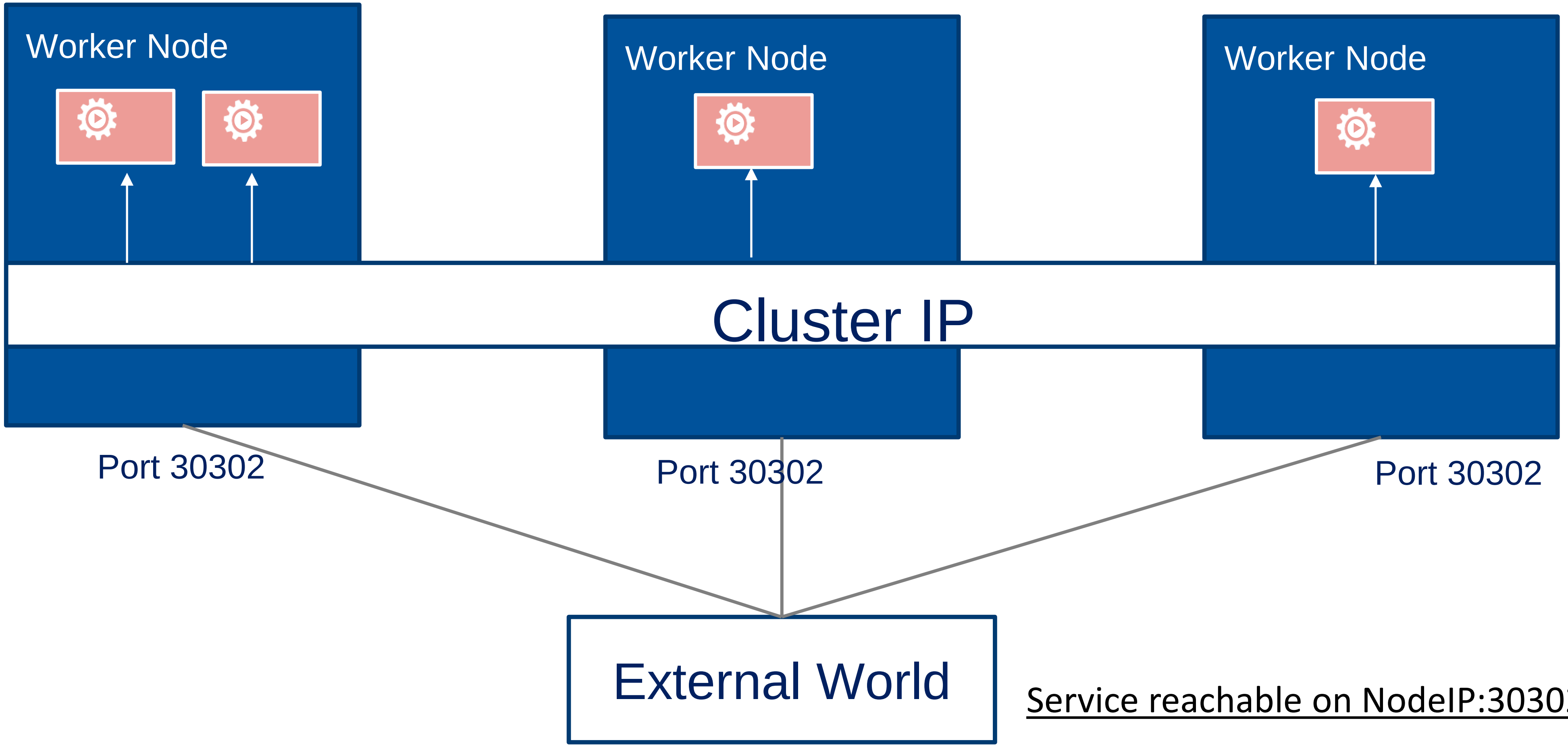
Multiple Port on the VIP can be exposed for different backend ports

NodePort ServiceType

- A NodePort service is one of the ways to get external traffic directly to your service.
- NodePort opens a specific port on all the Nodes, and any traffic that is sent to this port is forwarded to the service.
- You’ll be able to contact the NodePort service, from outside the cluster, by requesting <NodeIP>:<NodePort>
- NodePort default range: 30000 – 32767
- Once routed to a Node, it is sent to ClusterIP

```
apiVersion: v1
kind: Service
metadata:
  name: my-nodeport-service
spec:
  selector:
    app: my-app
  type: NodePort
  ports:
  - name: http
    port: 80
    targetPort: 80
    nodePort: 30036
    protocol: TCP
```

NodePort ServiceType (Cont.)



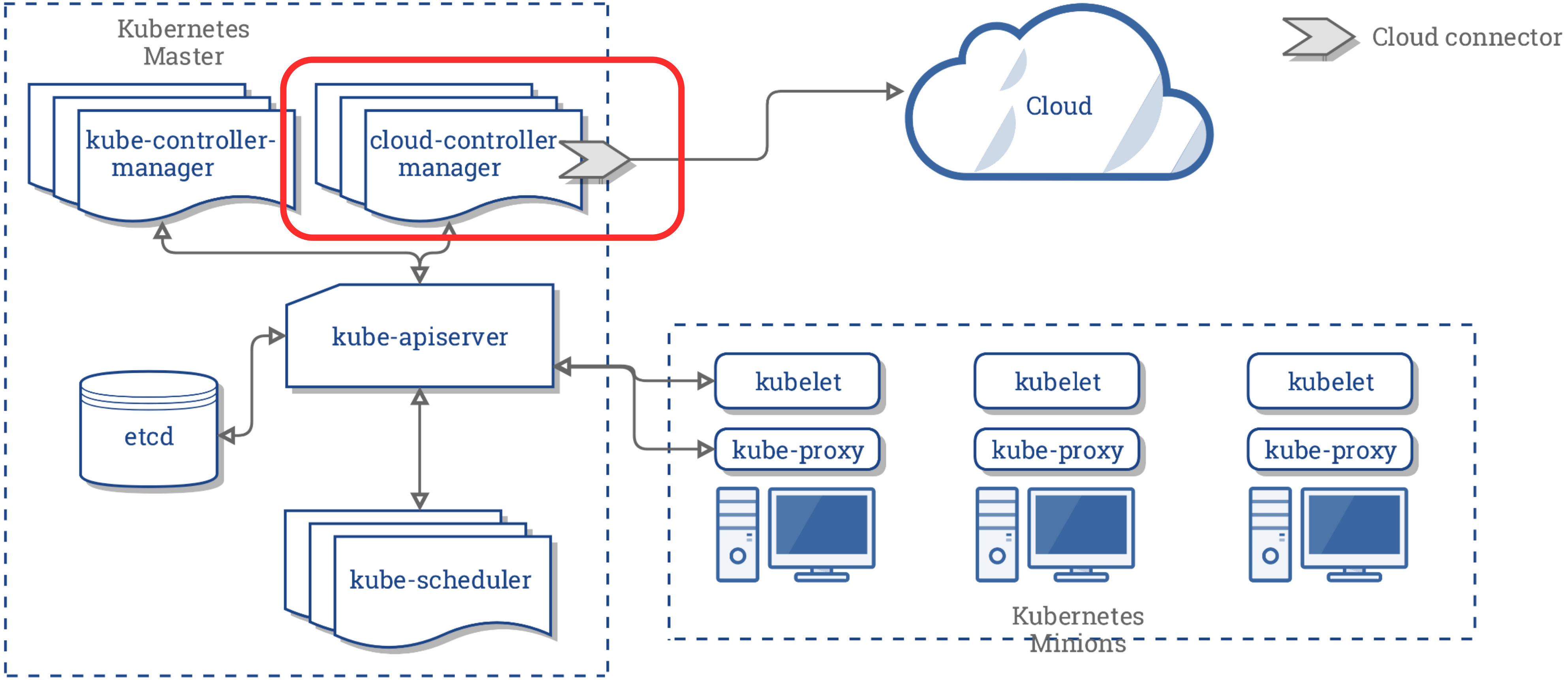
Service reachable on NodeIP:30302

Load Balancer ServiceType

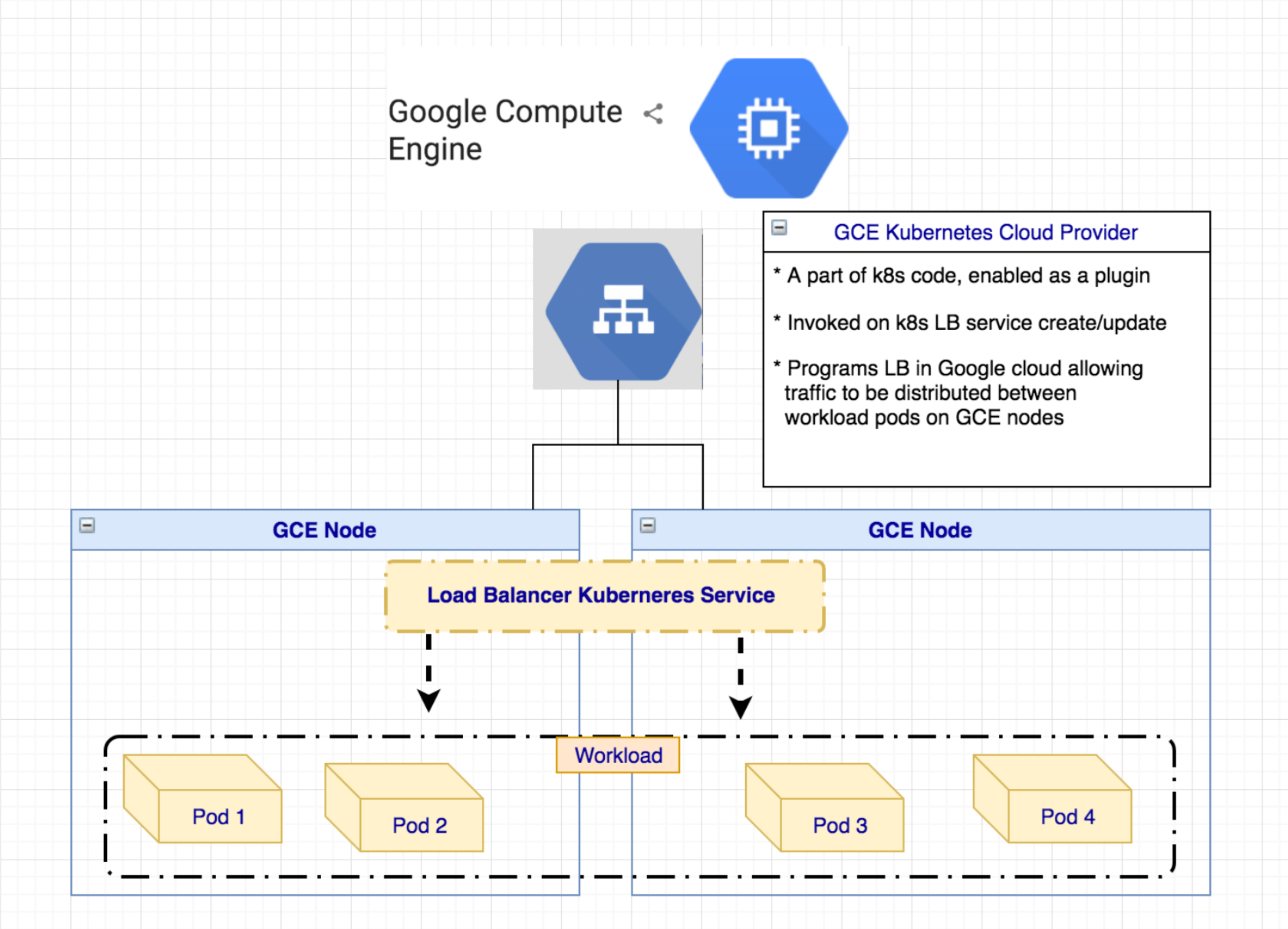
- Exposes the service externally using a cloud provider’s load balancer.
- NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.
- Exposes the service externally using a cloud provider’s load balancer.
  - On AWS for example it spins up an ELB.

```
apiVersion: v1
kind: Service
metadata:
  name: my-lb-service
spec:
  selector:
    app: my-app
  type: LoadBalancer ←
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
```

Kubernetes Architecture



Load Balancer - GCP



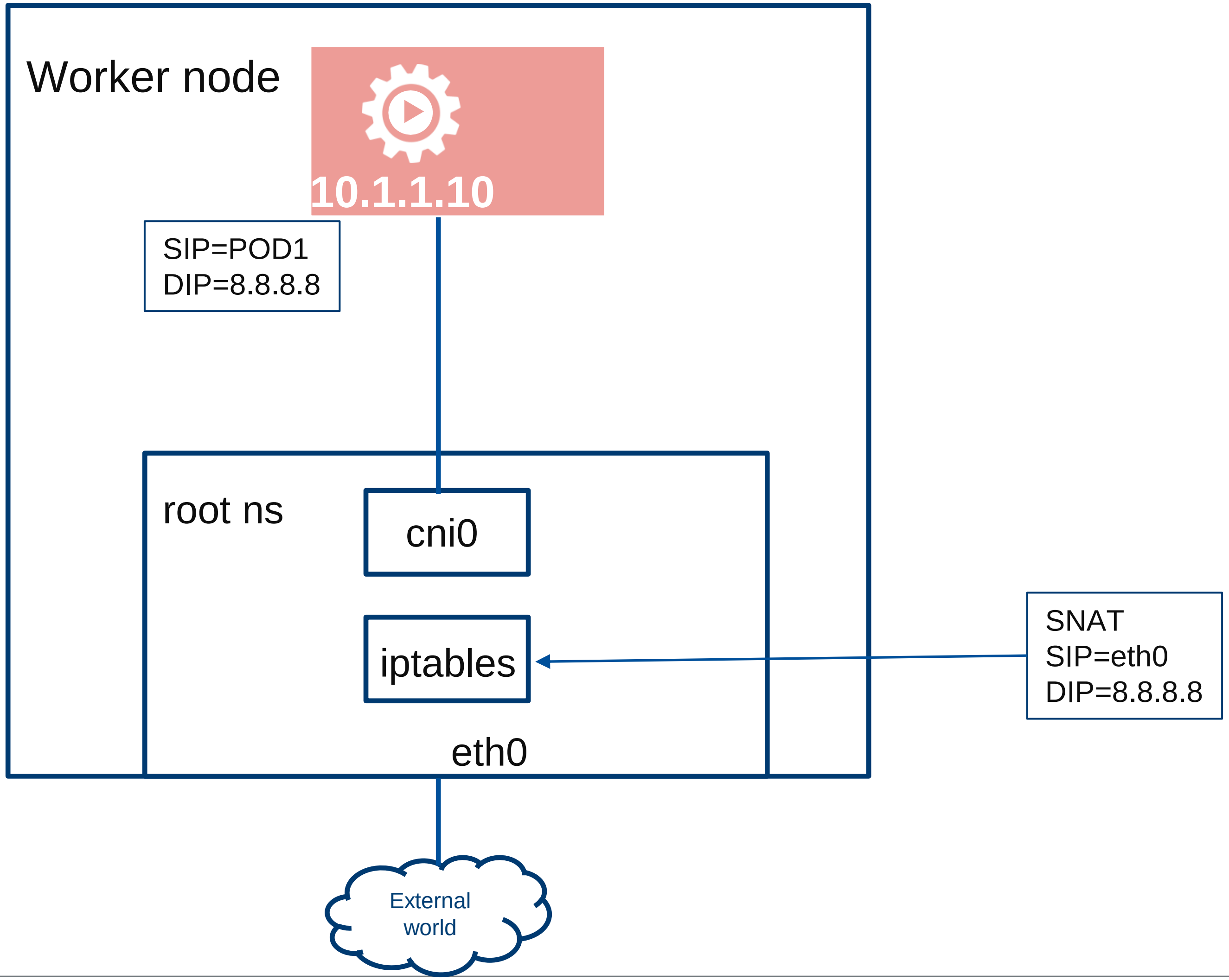
GCE Kubernetes Cloud Provider

* A part of k8s code, enabled as a plugin

* Invoked on k8s LB service create/update

* Programs LB in Google cloud allowing traffic to be distributed between workload pods on GCE nodes

POD to External World (South – North)



Services Summary

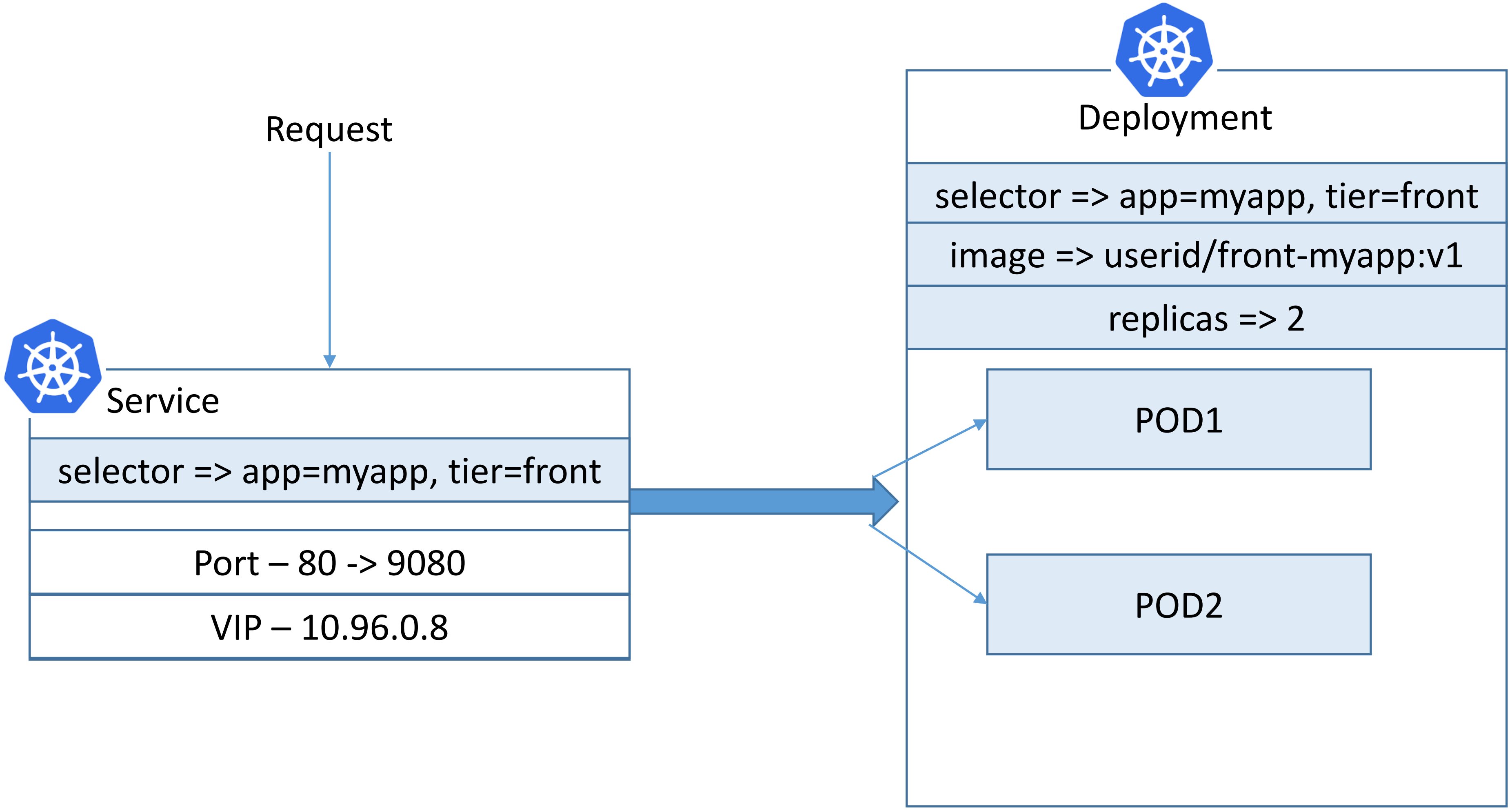
- Cluster IP
  - East-West Traffic exposed as a VIP per service

- Node Port
  - North-South Traffic reachable as NodeIP:NodePort

- Load Balancer
  - North-South Traffic (automated) exposed via a LB to NodeIP:NodePort

- Ingress
  - Allows simple host or URL based HTTP routing.
  - An ingress controller is responsible for reading the Ingress Resource information and programming data forwarding rules.

Microservice



Request

Service

selector => app=myapp, tier=front

Port – 80 -> 9080

VIP – 10.96.0.8

Deployment

selector => app=myapp, tier=front

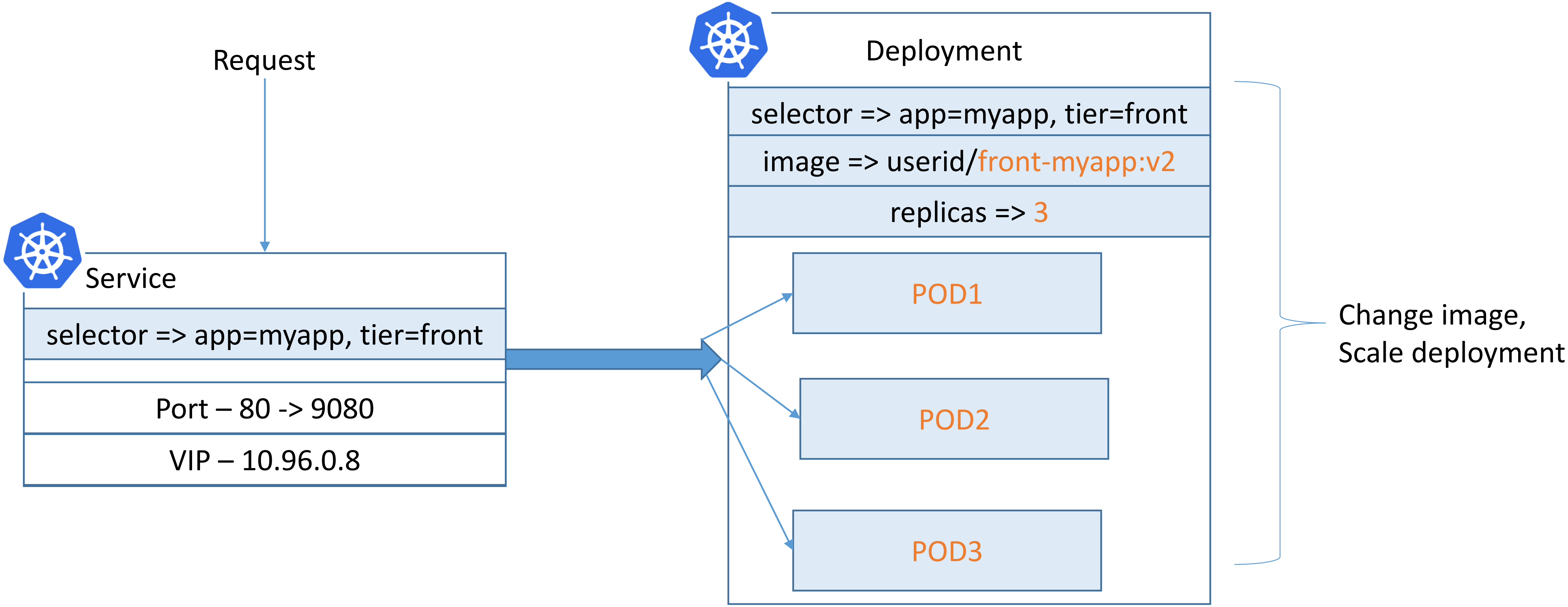
image => userid/front-myapp:v1

replicas => 2

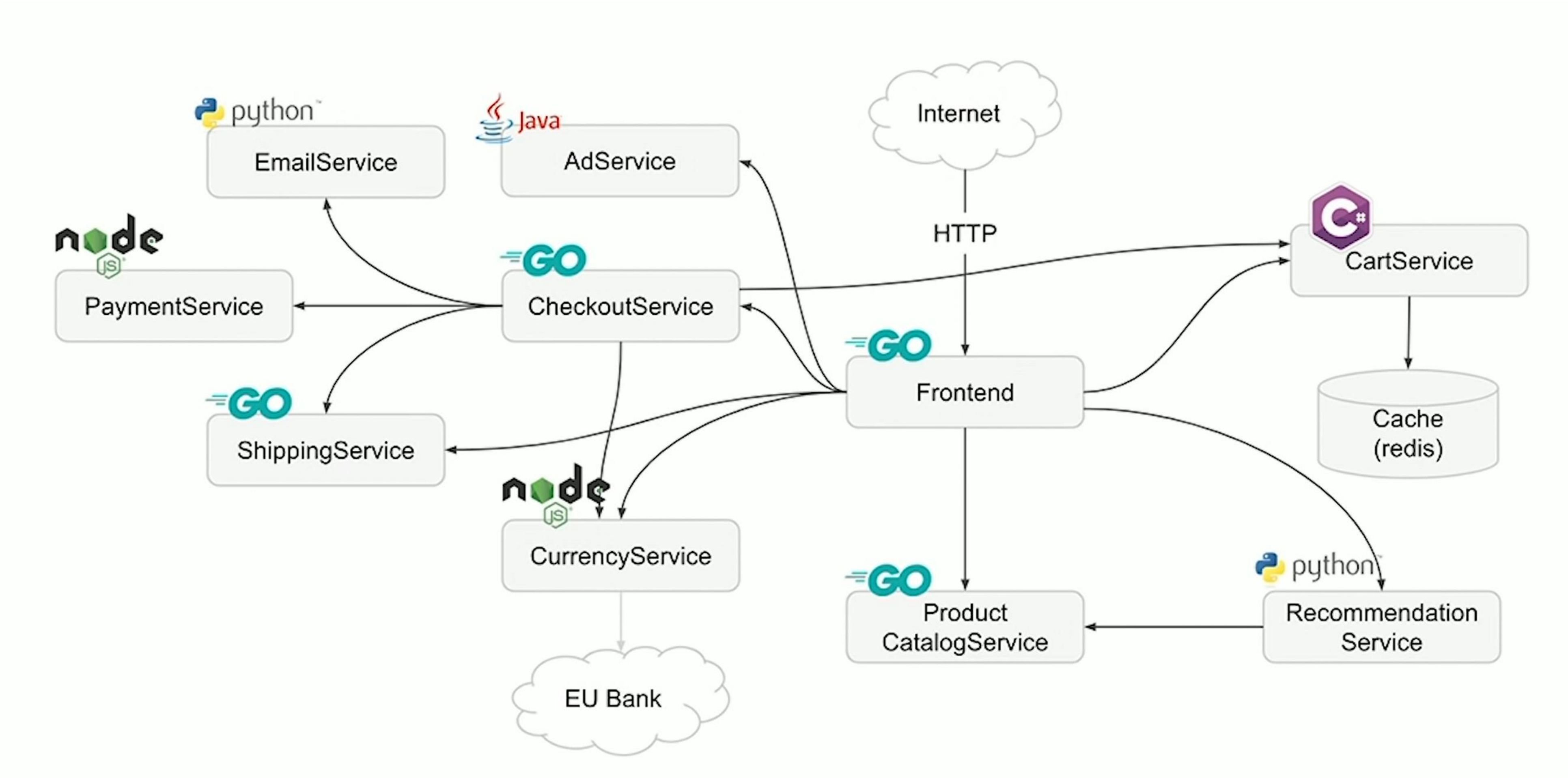
POD1

POD2

Microservice



Demo App



Hipster Shop Demo App - https://github.com/GoogleCloudPlatform/microservices-demo

Kubernetes Services Limitations

- For North-South traffic, Type LoadBalancer
  - 1 LB per resource – External Public IP
  - DNS Configuration (CNAMEs)
  - Certificate Management
  - Log / Device Management

Ingress Controllers

Using only Services to handle HTTP Traffic

- Node Port can be used to expose a service outside of the Kubernetes cluster. This is only a single service that gets exposed at a time

- Load balancing on multiple node ports can be difficult

- No intelligence to route traffic to certain services based on a host or path

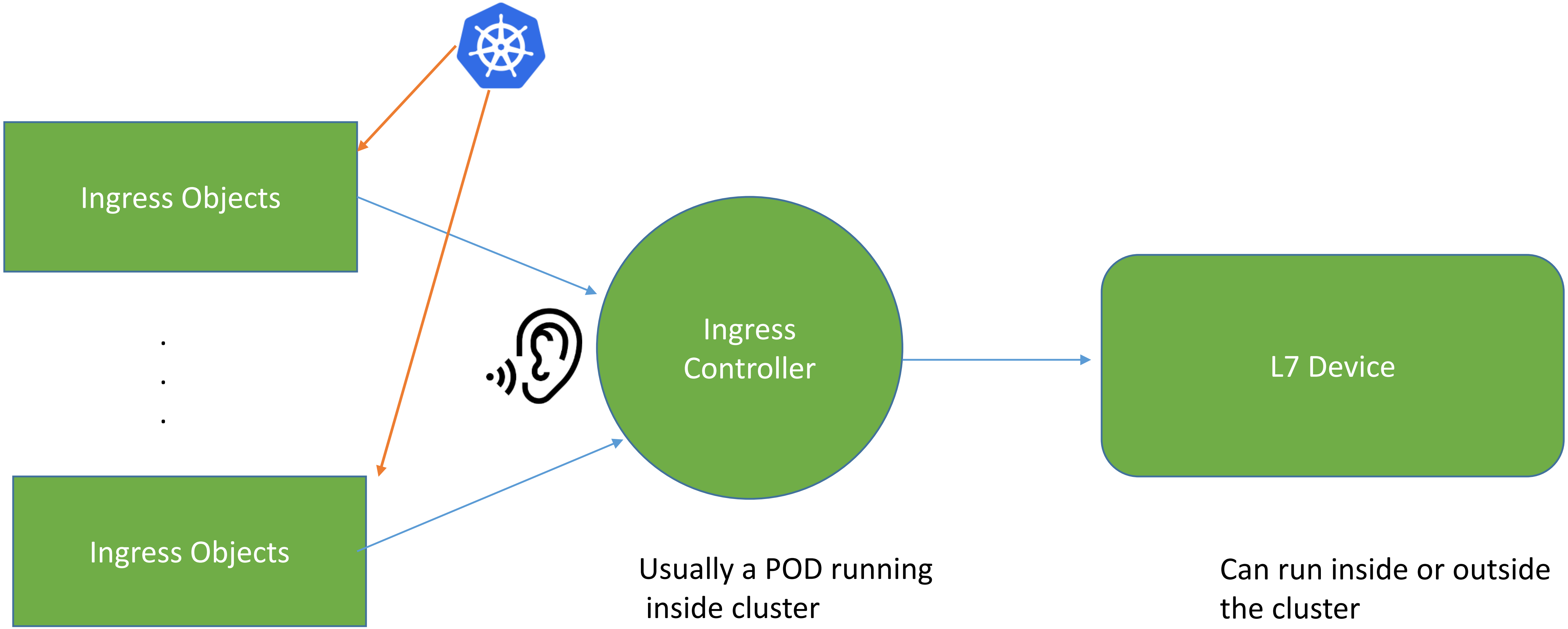
- Can be difficult to scale and manage as your application grows

Ingress – Layer 7 LB

- API resource – kind: Ingress
- Ingress is NOT a type of service
  - Built on top of Services
- Requires
  - Ingress Resource
    - Specifies rules
  - Ingress Controller (Third Party Proxy)
    - Application which based on api-server changes for ingress, updates forwarding rules.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-ingress
rules:
  - host: foo.mydomain.com
    http:
      paths:
      - backend:
          serviceName: foo
- host: mydomain.com
    http:
      paths:
      - path: /bar/*
        backend:
          serviceName: bar
```

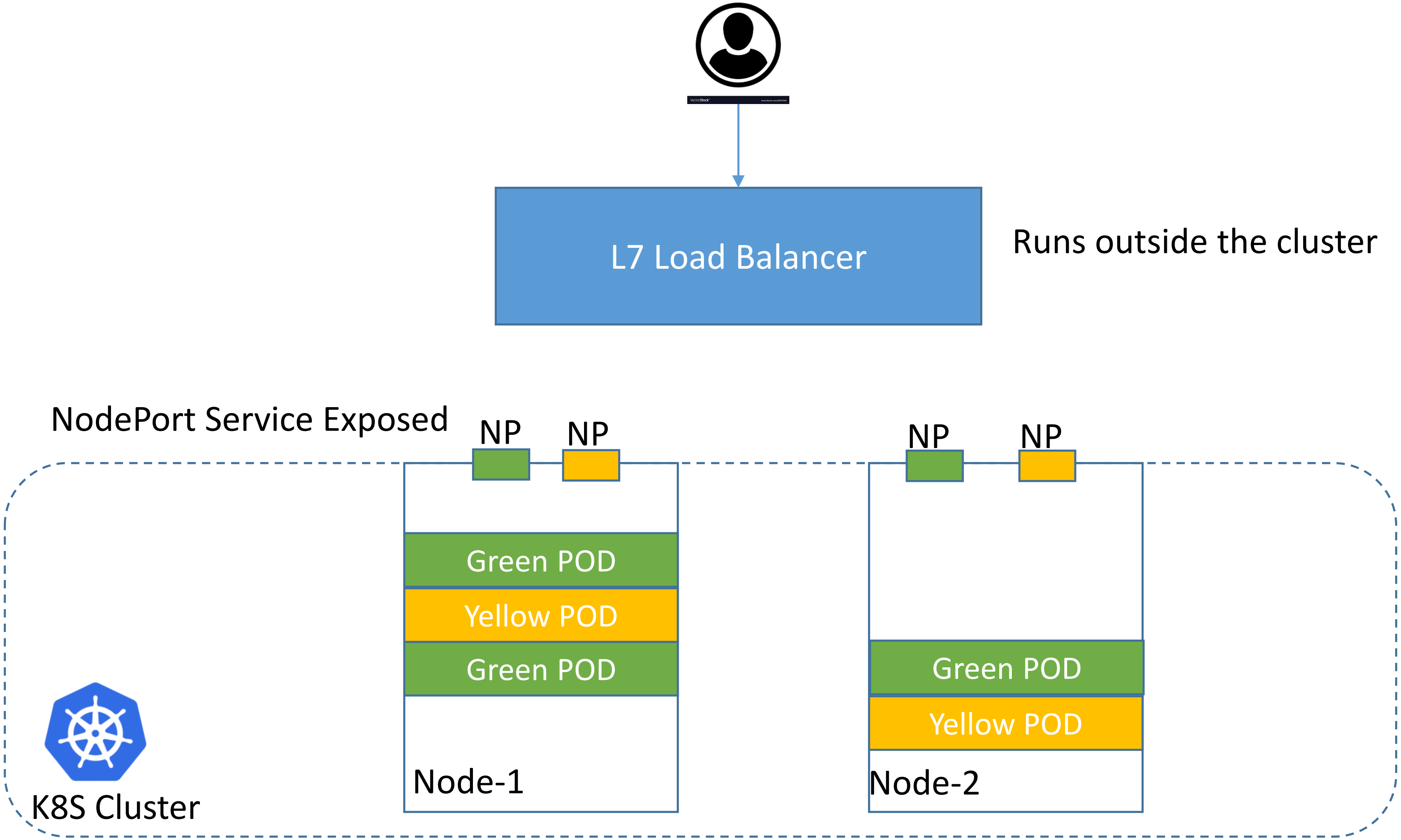
Components



Usually a POD running inside cluster

Can run inside or outside the cluster

Flow – External L4-7 Device

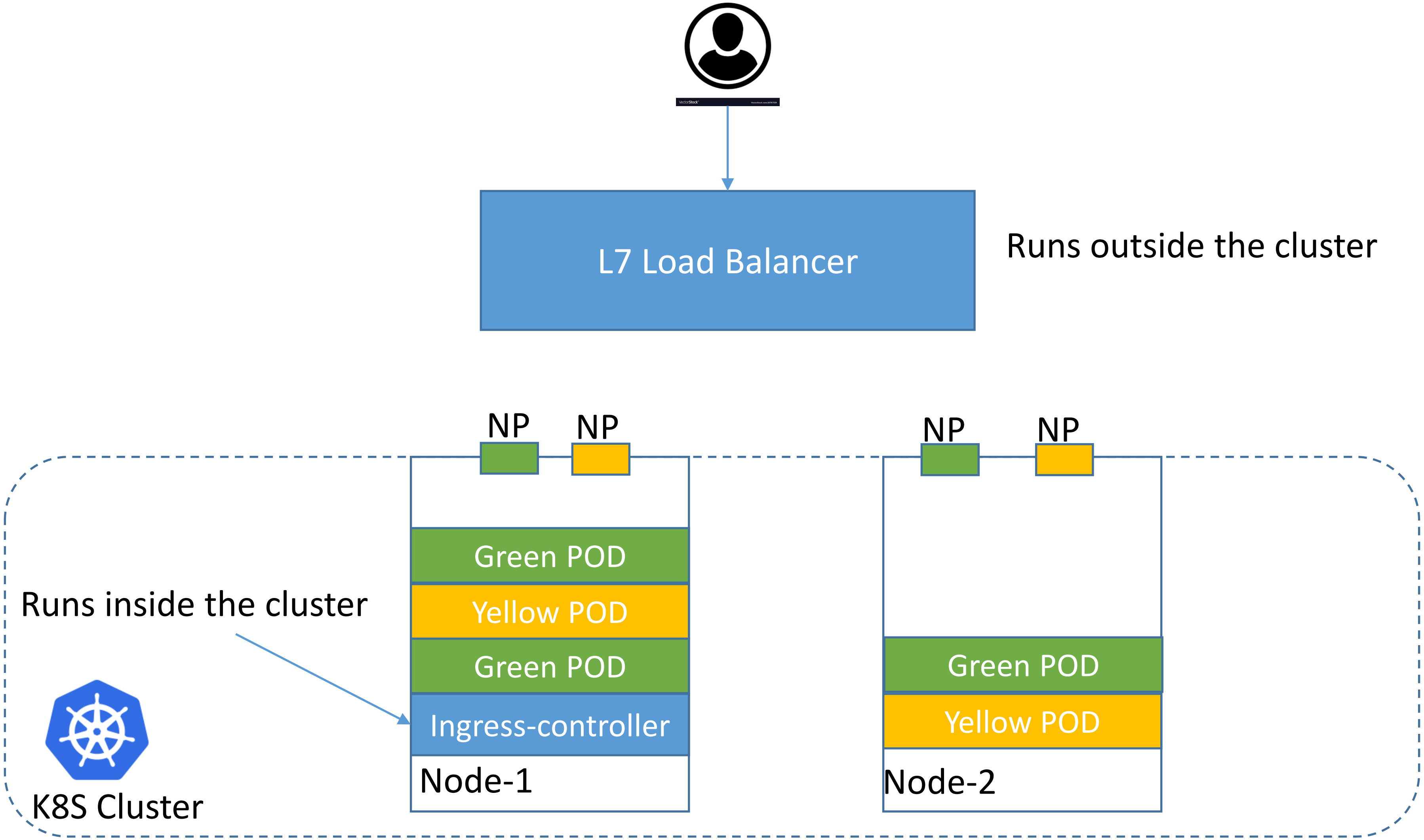


L7 Load Balancer

Runs outside the cluster

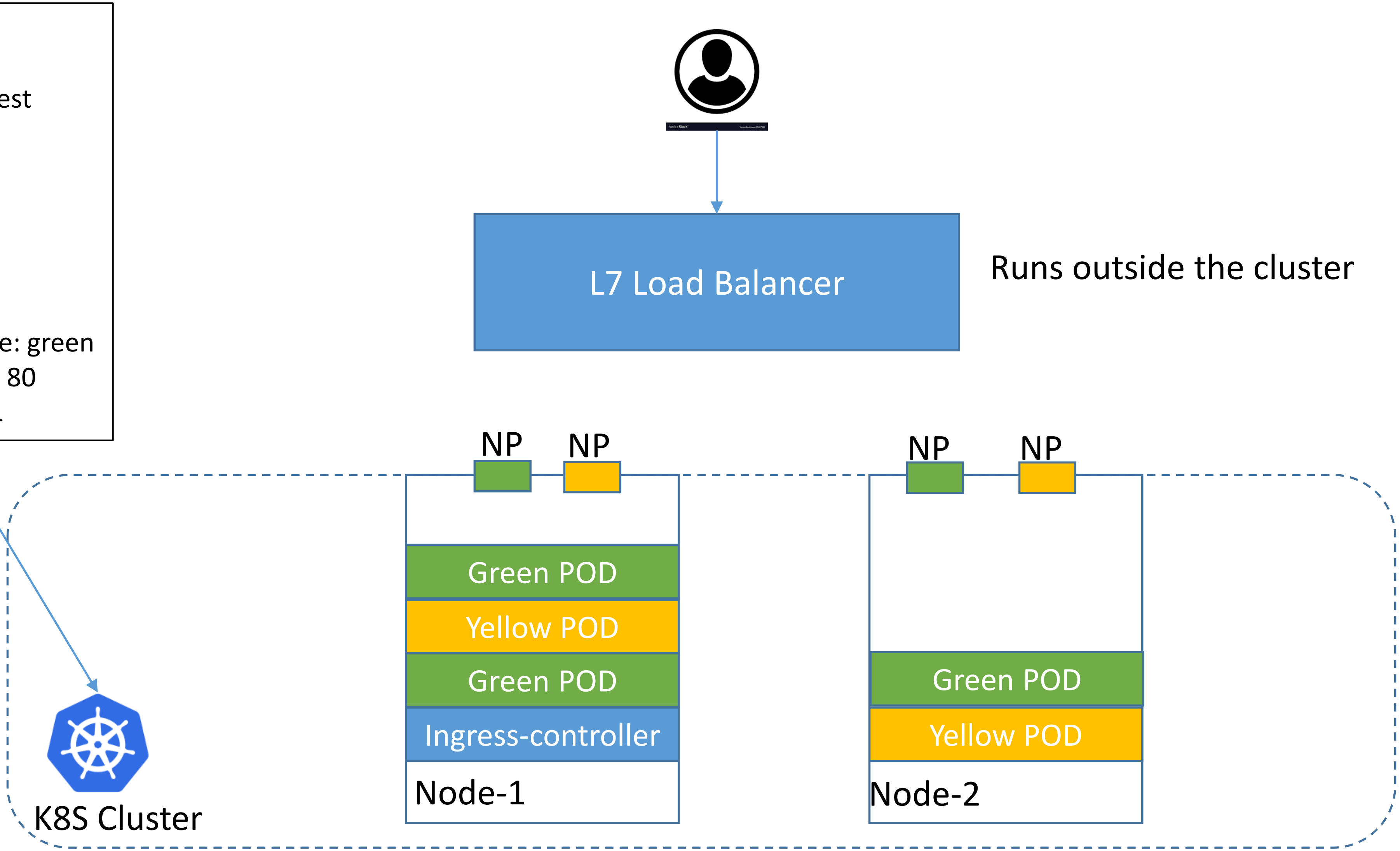
NodePort Service Exposed

Flow – External L4-7 Device



Flow – External L4-7 Device

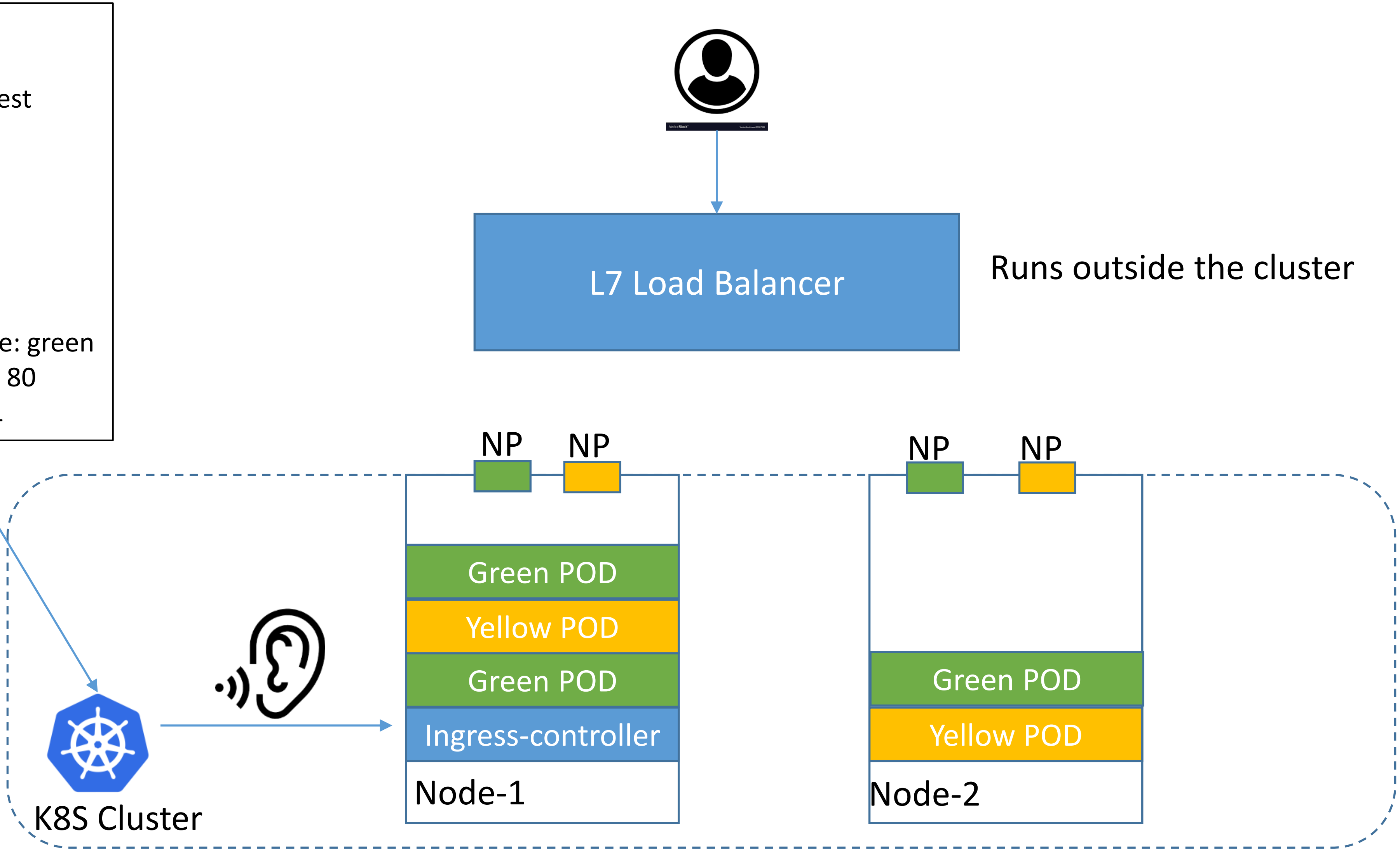
```
kind: Ingress
metadata:
  name: ingress-test
spec:
 rules:
 - host: foo.com
   http:
     paths:
     - backend:
         serviceName: green
         servicePort: 80
       path: /nginx1
```



Runs outside the cluster

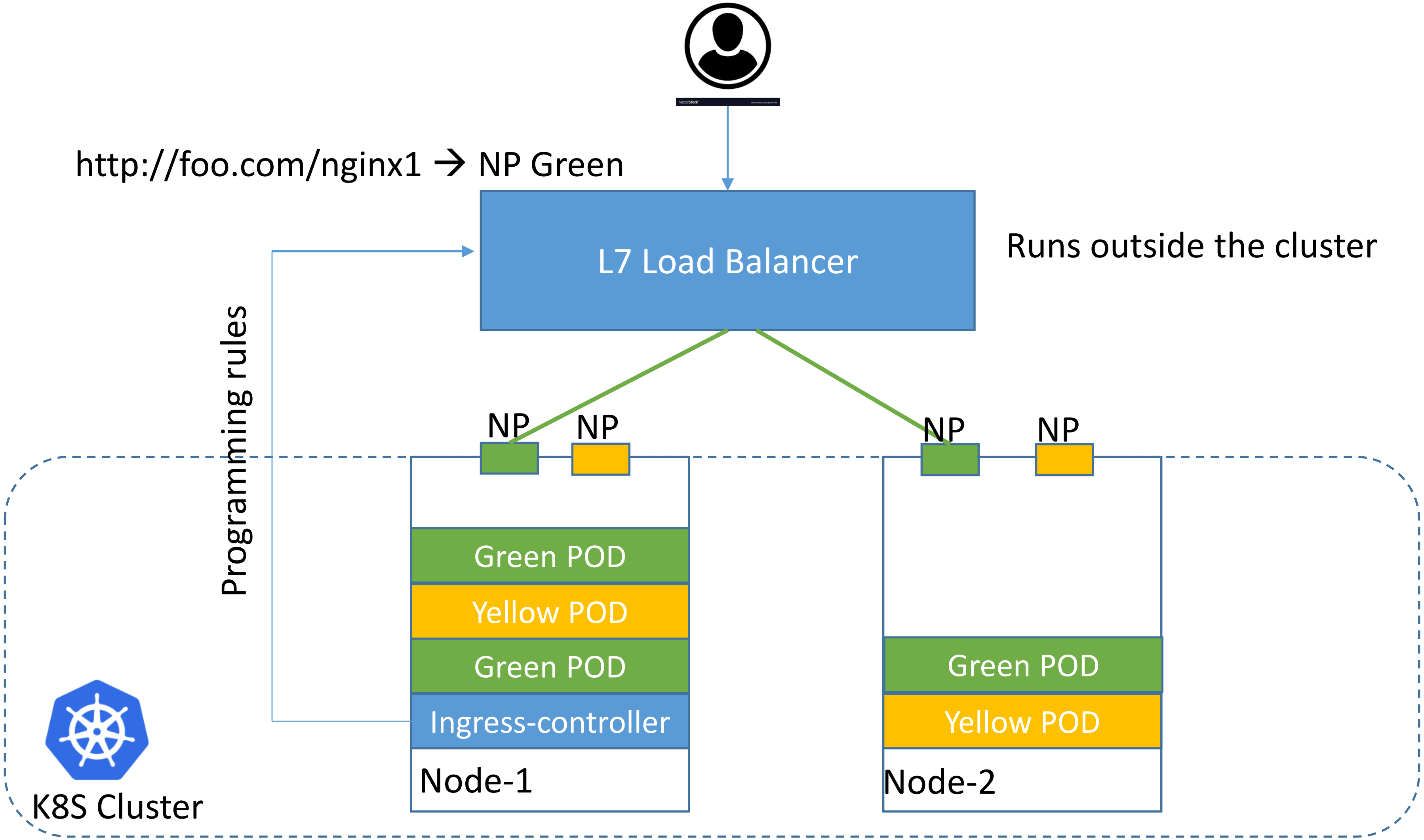
Flow – External L4-7 Device

```
kind: Ingress
metadata:
  name: ingress-test
spec:
 rules:
 - host: foo.com
   http:
     paths:
     - backend:
         serviceName: green
         servicePort: 80
       path: /nginx1
```



Runs outside the cluster

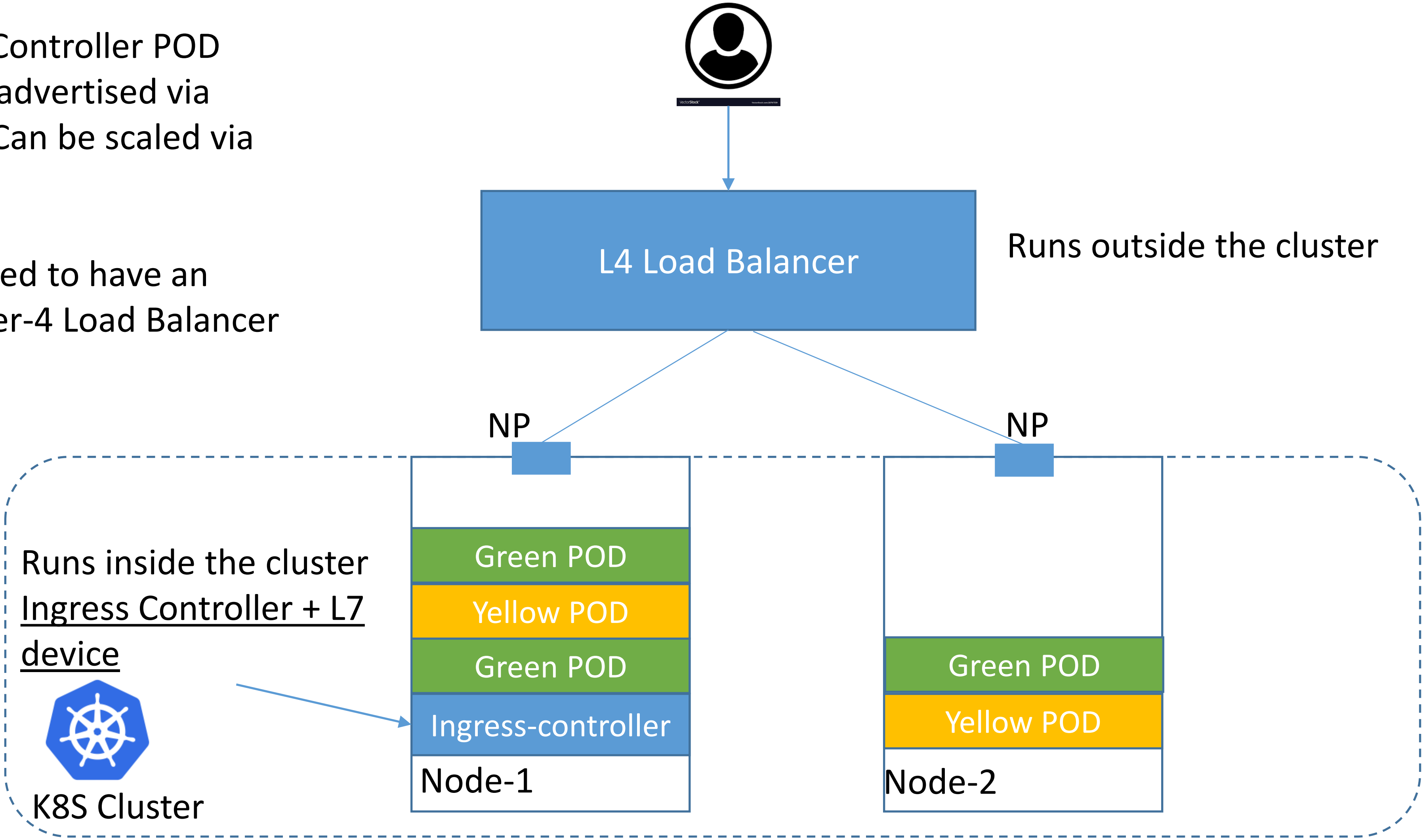
Flow – External L4-7 Device



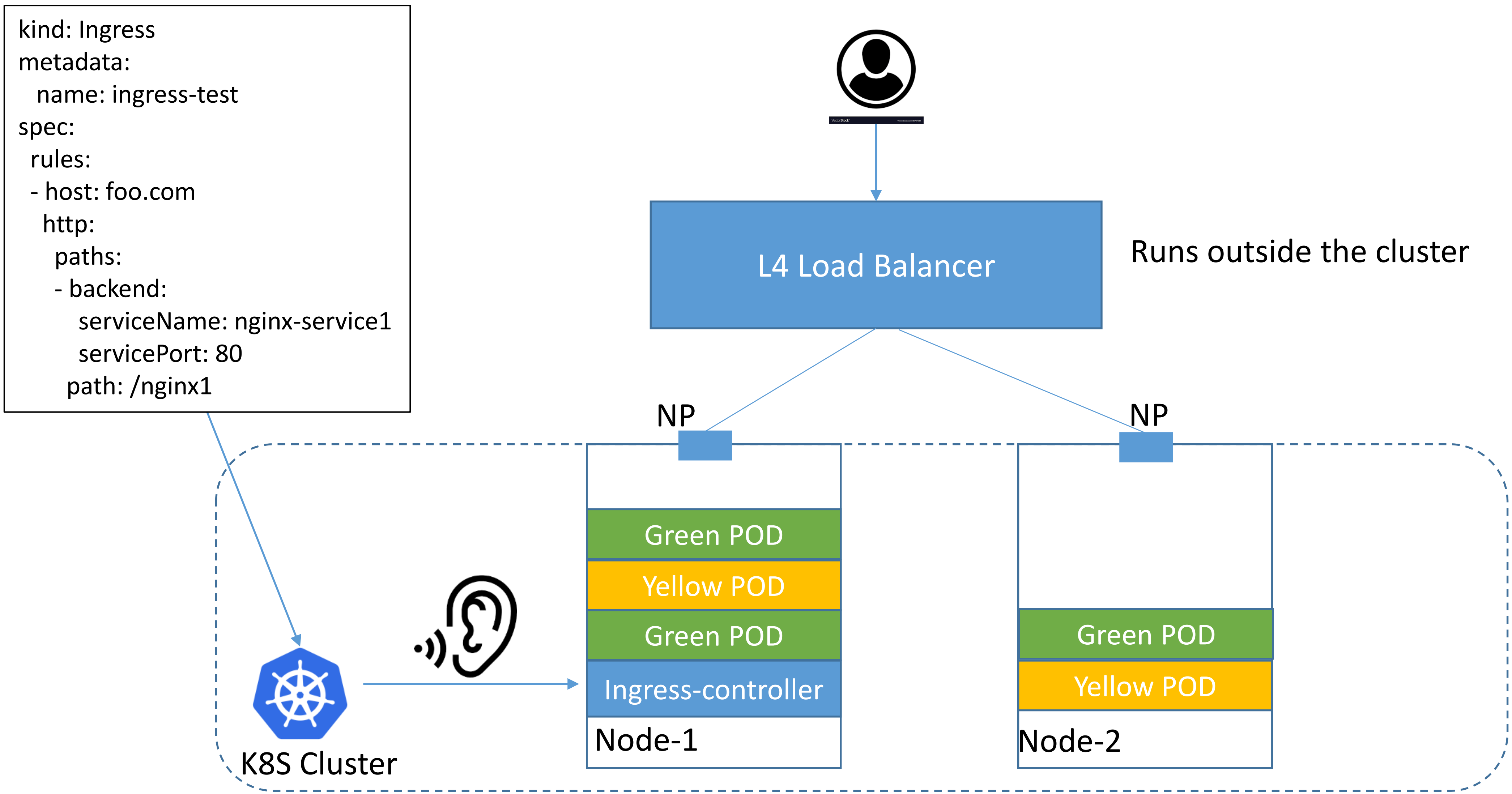
Flow – L4-7 Device Internal to the Cluster

- The Ingress Controller POD needs to be advertised via NodePort – Can be scaled via deployment

- Recommended to have an external Layer-4 Load Balancer



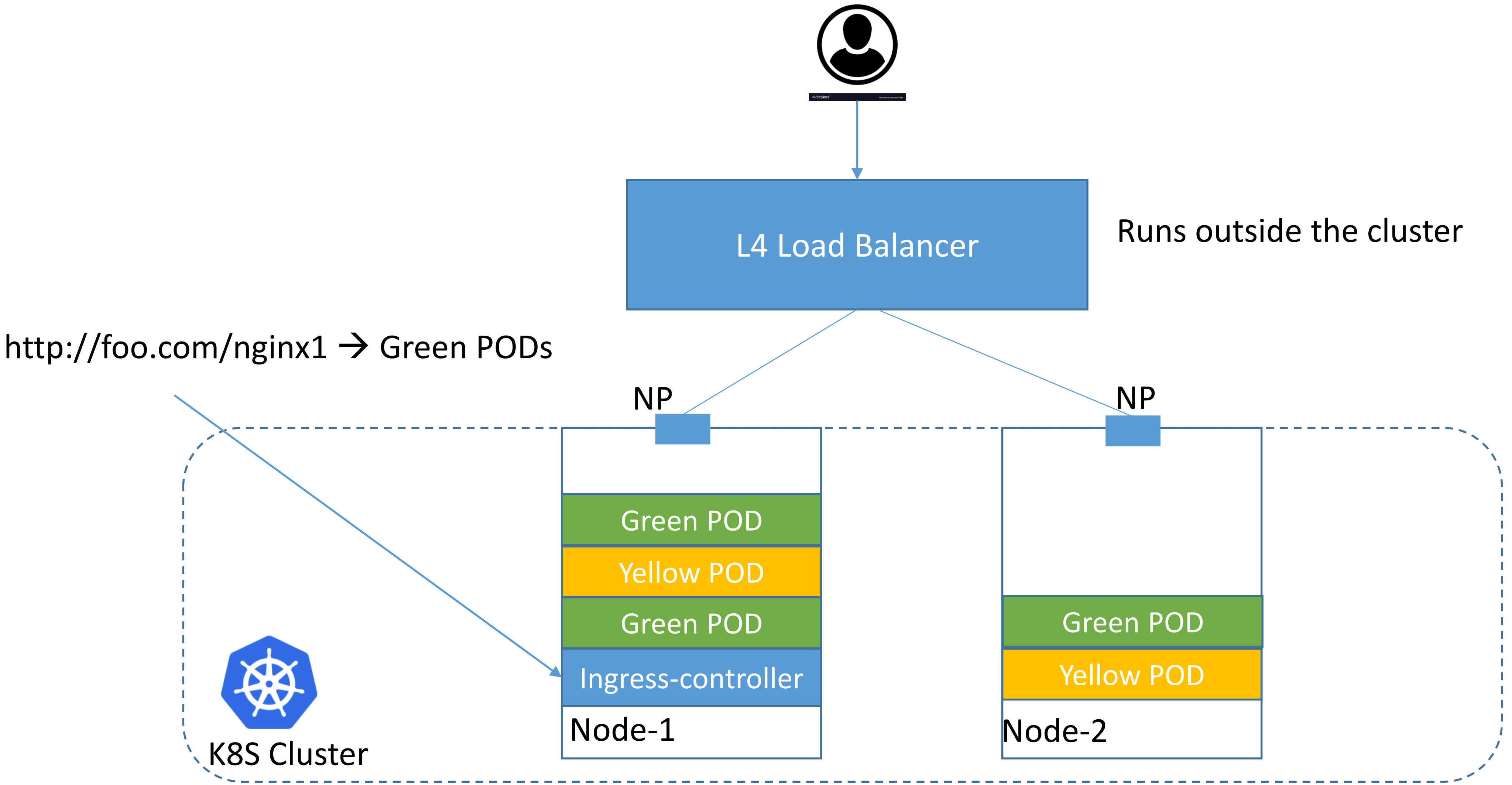
Flow – L4-7 Device Internal to the Cluster



```
kind: Ingress
metadata:
  name: ingress-test
spec:
 rules:
 - host: foo.com
   http:
     paths:
     - backend:
         serviceName: nginx-service1
         servicePort: 80
       path: /nginx1
```

Runs outside the cluster

Flow – L4-7 Device Internal to the Cluster

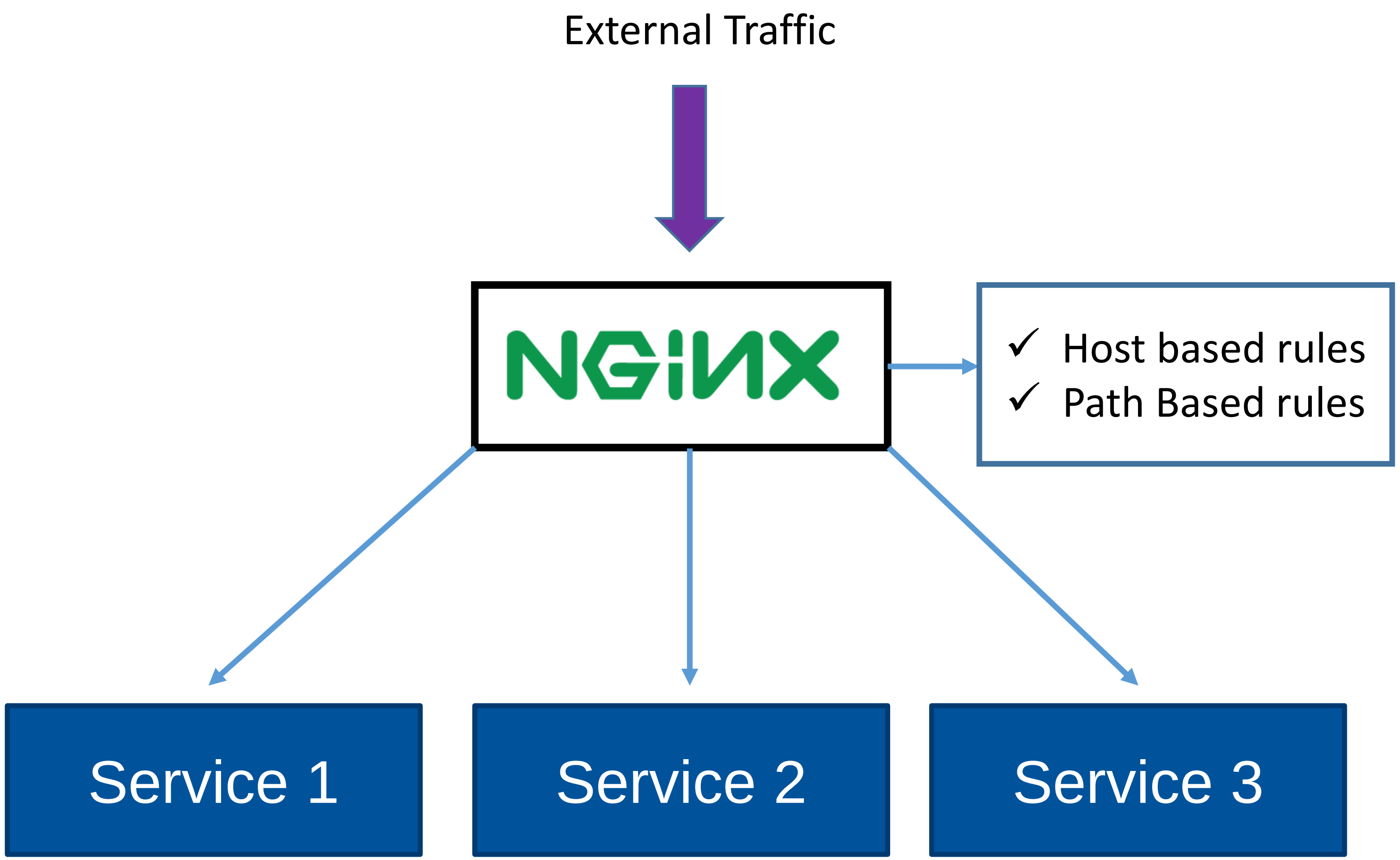


Ingress Controllers Solutions

- NGINX Ingress Controller for Kubernetes.
- Contour is an Envoy based ingress controller
- F5 BIG-IP Controller for Kubernetes.
- HAProxy based ingress Istio based ingress controller Control Ingress Traffic.
- Traefik is a fully featured ingress controller

Ingress Controllers

- An ingress controller is a 3rd party controller used for routing of HTTP / HTTPS traffic to its intended service
- Can be based on both a physical or virtual solution
- Traffic is redirected based on a rule set.
- Can also provide SSL / TLS termination
- HostIP or NodePort termination



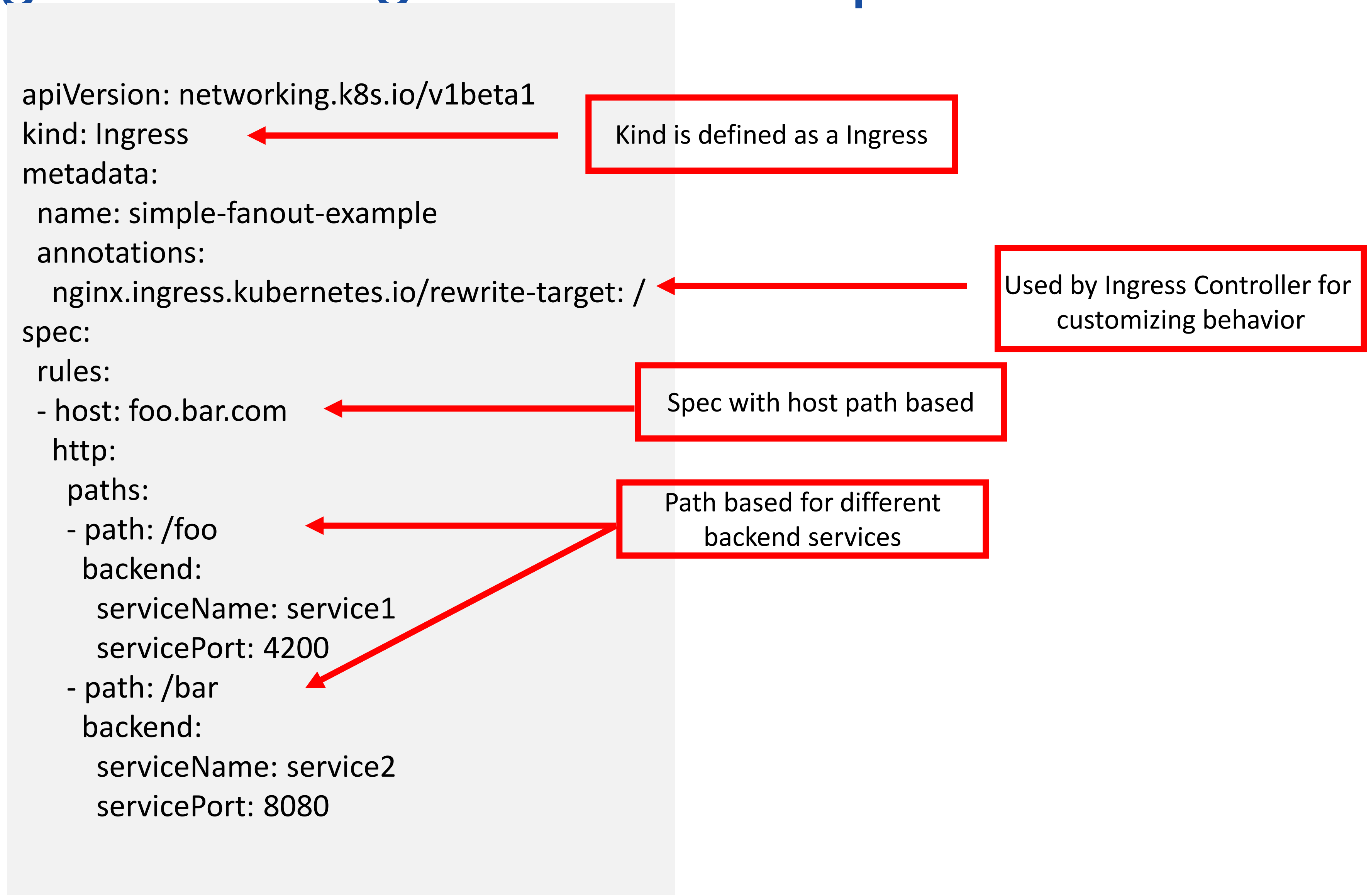
Basic Ingress Configuration

- **Host based routing** – Direct HTTP/HTTPS traffic based on a URL. This is the starting point for all ingress rulesets (example: foo.com)

- **Path based routing** – Direct traffic to a particular service based on a path. For example "/foo" will direct traffic to service one, and "/bar" will direct traffic to service two

- **Host + Path based routing** – This allows you to direct traffic using a combination of host and path routing. This allows you to map multiple services to a single site.

Ingress Configuration Example



```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: simple-fanout-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: service1
          servicePort: 4200
      - path: /bar
        backend:
          serviceName: service2
          servicePort: 8080
```

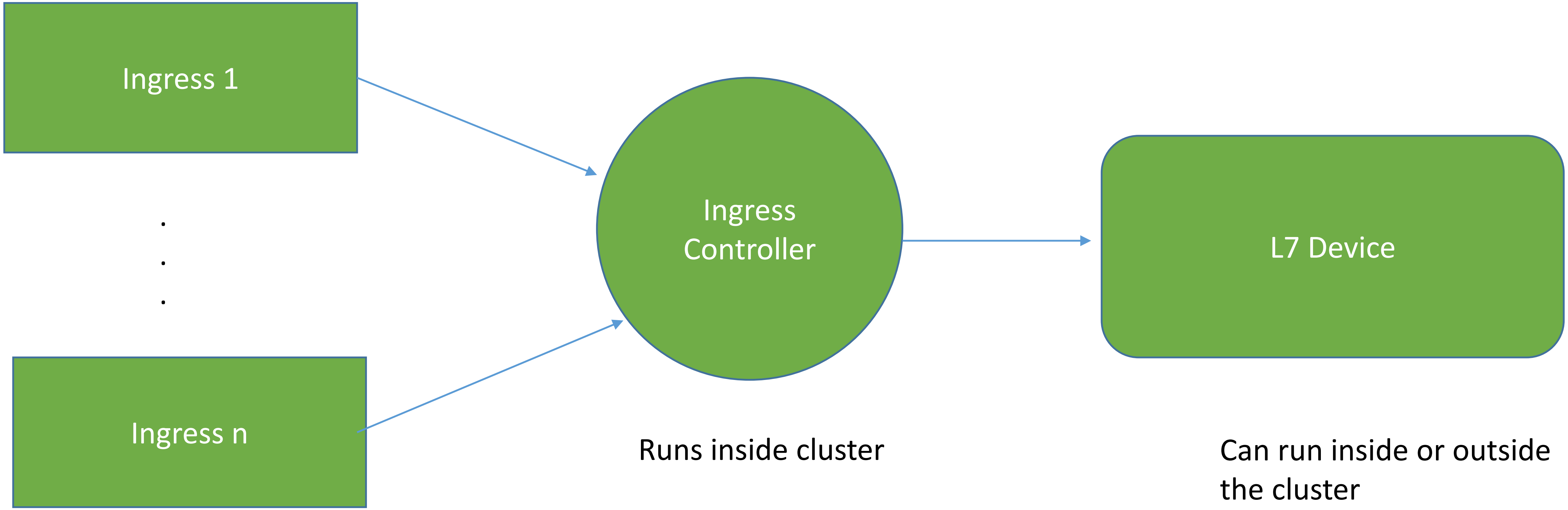
Kind is defined as a Ingress

Used by Ingress Controller for customizing behavior

Spec with host path based

Path based for different backend services

Ingress – Layer 7 LB



Ingress 1

.
.
.

Ingress n

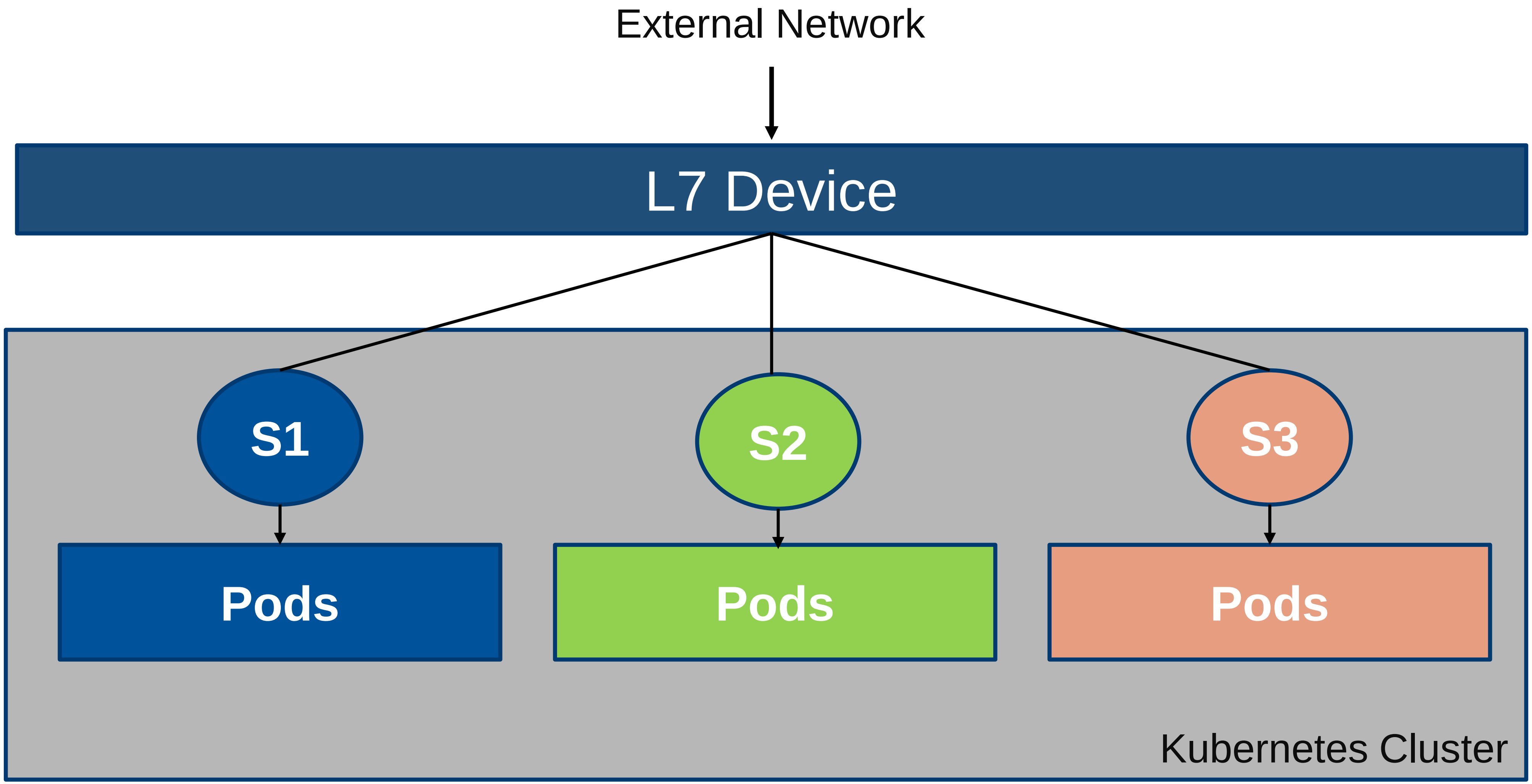
Ingress Controller

L7 Device

Runs inside cluster

Can run inside or outside the cluster

Ingress – Layer 7 LB



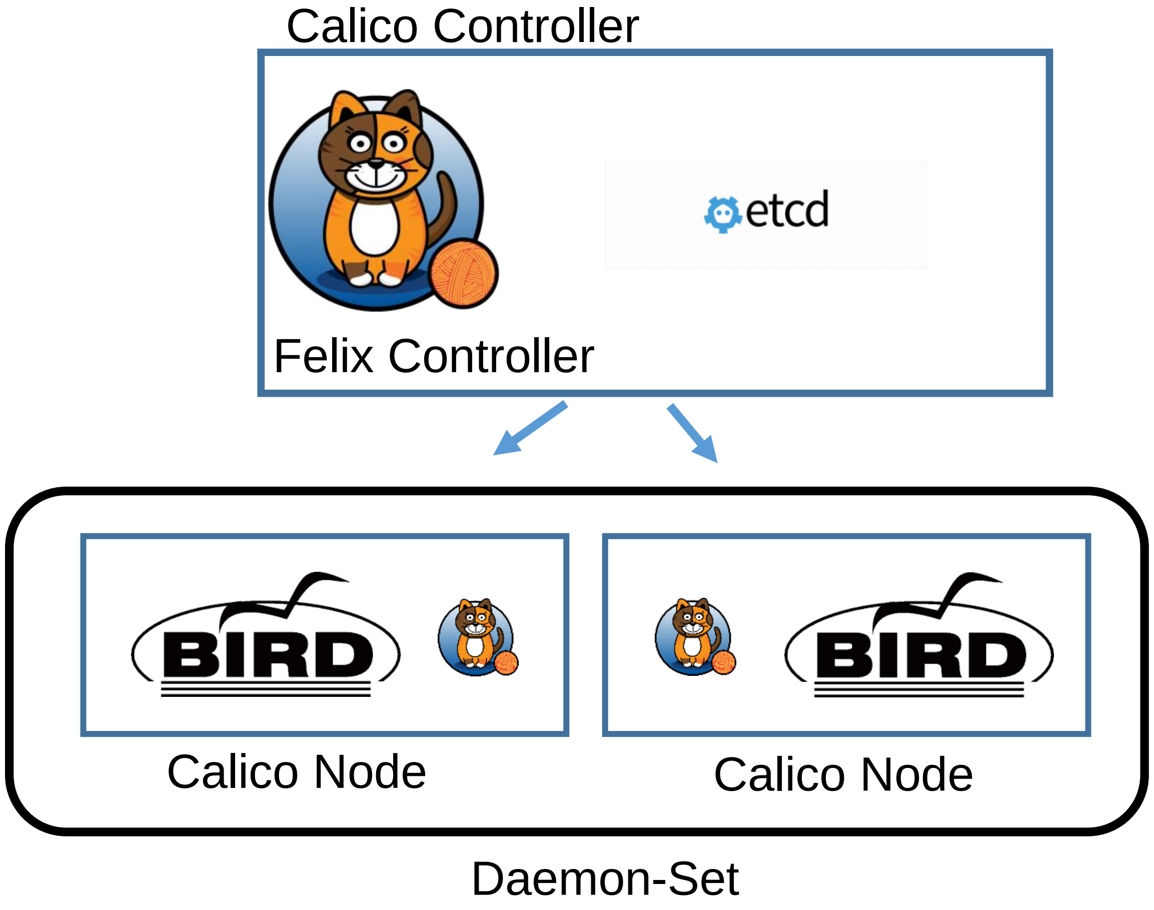
Introduction to the Calico CNI

Calico CNI

- Open source solution that provides networking for Docker EE, Openshift, OpenStack, Bare Metal and of course Kubernetes
- Supports BGP for advanced routing capabilities
- Fully Supports Kubernetes Network Policies for security
- Has its own Network policy framework for advanced options
- Has it’s own CLI utility for configuration (calicoctl)

Calico Architecture

- **calico-felix** (node)– writes to linux routing table and also iptables.
- **BIRD** (node) – A daemon that runs on the calico node that handles BGP
- **Calico Controller** – Monitors Kubernetes API and takes action on that. The calicoctl utility interacts directly with the controller.



Calicoctl utility

- Used for creating resources in Calico
- Similar to kubectl command line structure
- Options you can configure:
  - BGP Configuration
  - IP Pools (POD IPs)
  - Change encapsulation types (VXLAN or IPnIP)
  - Network Policies (discussed later)

Example:

```
$ calicoctl get ippool -o wide
NAME                   CIDR              NAT   IPIPMODE   DISABLED   SELECTOR
default-ipv4-ippool   192.168.0.0/16    true   Always      true       all()
pod-pool               10.244.0.0/16    true   Always      false      all()
```

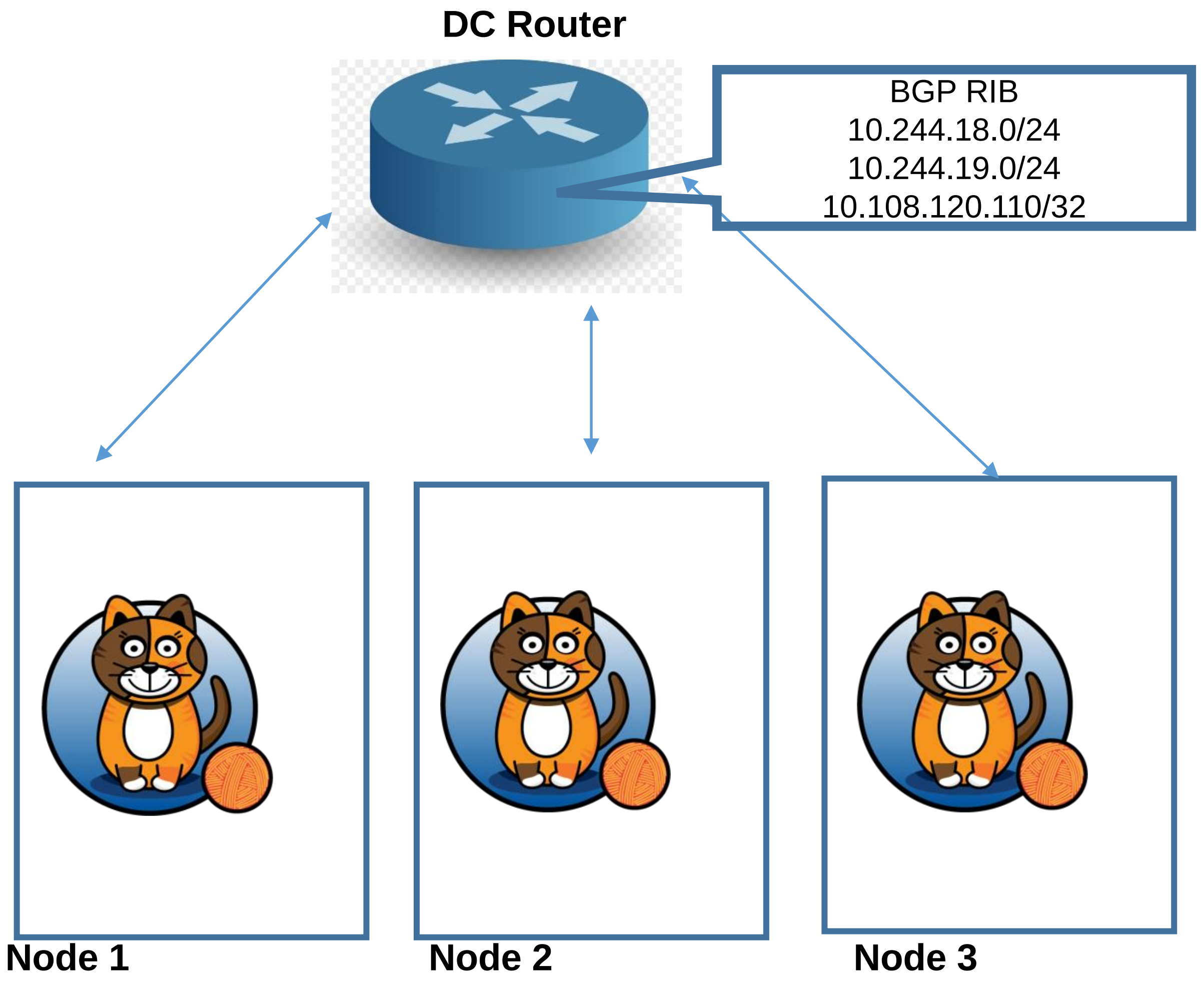
Calico IP Pools and Encapsulation

- Calico supports both VXLAN (as of 3.7) and IPinIP encapsulation depending on the environment you are operating in.
- Can be specified on a per IP Pool bases
- You can create multiple IP Pools and use label selectors for your deployments
- Calico will create a new tunnel interface (IPnIP) or VXLAN interface for each IP Pool

```
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: test-pool
spec:
  cidr: 10.245.0.0/16
  vxlanMode: Always
  natOutgoing: true
```

BGP Configuration

- BGP can be configured for route advertisement of:
  - Pod CIDR
  - Cluster IP CIDR (advertised as /32 for each)
- This allows for direct access from a customer network to the pod or cluster IP.
- Can be used to eliminate the need for north-south services node ports
- Calico supports Route Reflectors



Calico Initial BGP Configuration

- Initial configuration of BGP to set up AS number as well as if you wish to use RR or a full mesh configuration.
- This configuration is required before peering with a device

```
apiVersion: projectcalico.org/v3
 kind: BGPConfiguration
 metadata:
   name: default
 spec:
   logSeverityScreen: Info
   nodeToNodeMeshEnabled: true
   asNumber: 64548
```

```
$ calicoctl get bgpconfig -o wide
NAME      LOGSEVERITY   MESHENABLED   ASNUMBER
default   Info          true          64548
```

Kubernetes Network Security

Network Policies

- By default, all traffic (ingress/egress) to pods in Kubernetes is allowed irrespective of namespace
- Networking Policy API Object allows L3/L4 isolation (aka Network ACL’s)
  - Not a firewall
- Policy is applied to POD ports
  - Implemented in iptables
- Stateful in nature

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: egress-pod2
  namespace: default
spec:
  podSelector:
    matchLabels:
      run: pod2
  egress:
  - to:
   - podSelector: {}
   policyTypes:
   - Egress
```

Network Policies

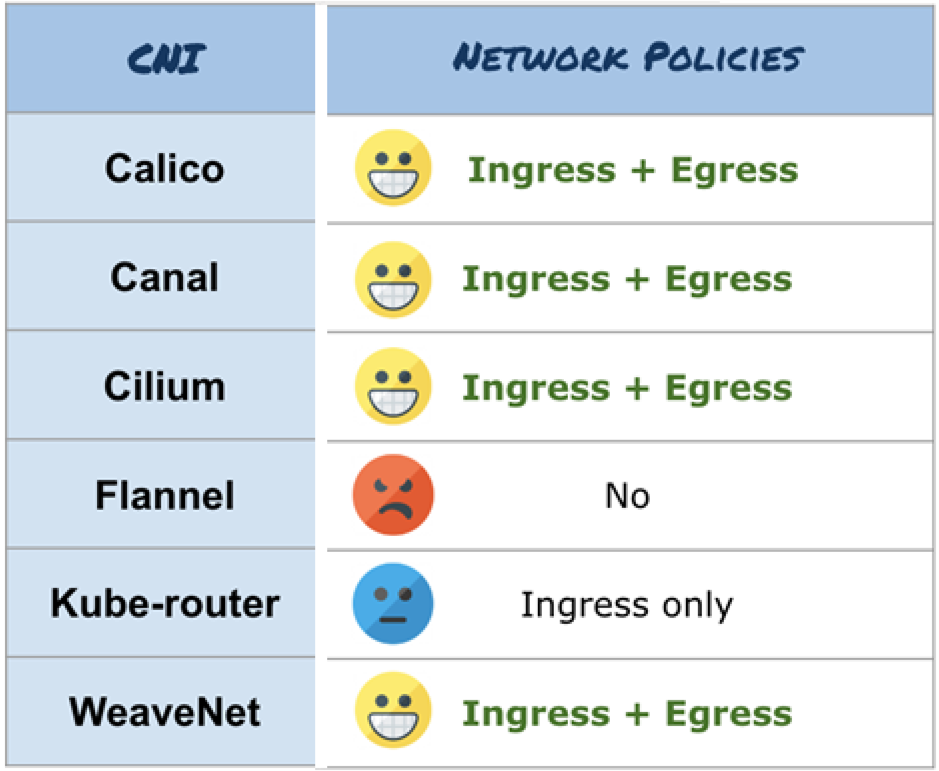
- Network Policies can be implemented on a namespace to block traffic to pods based on label selectors.

- The following rules apply:
  - Pods not defined in a network policy default to allow all
  - Pods defined in a network policy have an explicit deny unless specified
    - White-list model
  - Network policies can be used to control both ingress and egress traffic to a pod

- Network policies are ONLY supported for certain CNI plugins. Not all plugins support this feature.

Network Policies and CNI

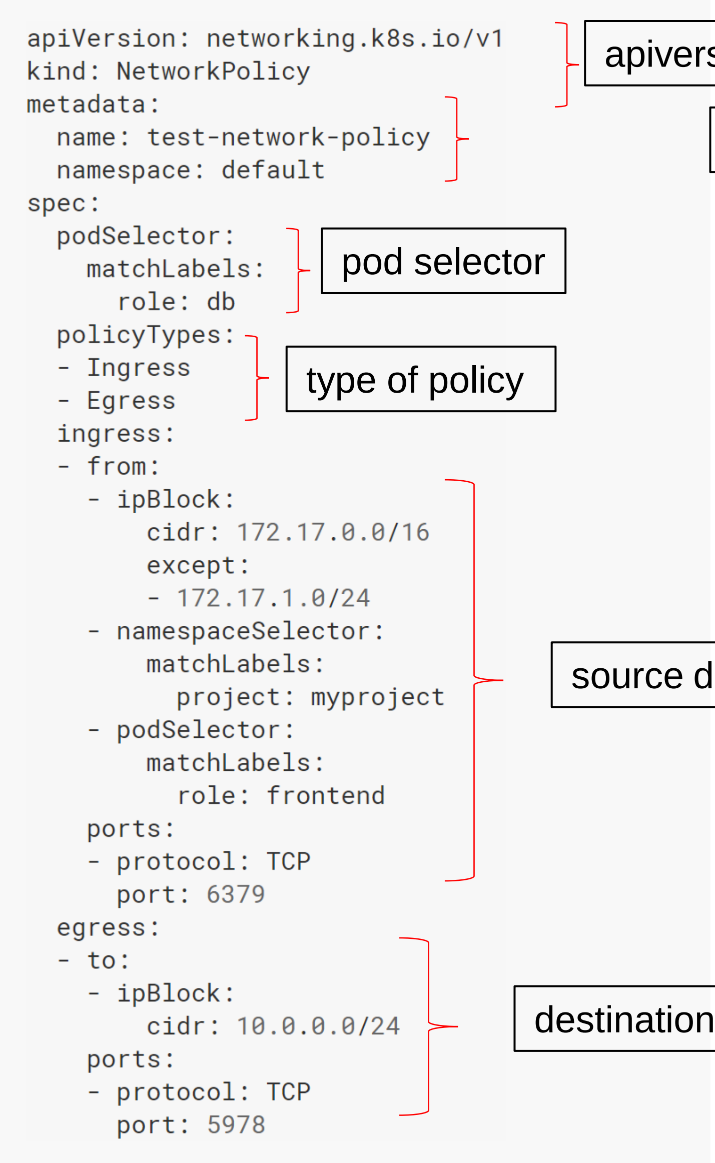
- CNI Plugin has to support Network Policy
- Canal = Flannel + Calico
- AWS CNI + Calico



CNI	Network Policies
Calico	😁 Ingress + Egress
Canal	😁 Ingress + Egress
Cilium	😁 Ingress + Egress
Flannel	😠 No
Kube-router	😐 Ingress only
WeaveNet	😁 Ingress + Egress

Network Policy Object

- Empty pod selector selects all pods in the namespace
- Whitelist ingress/egress rules
  - Ingress mandatory



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

apiversion and kind

metadata, namespaced resource

pod selector

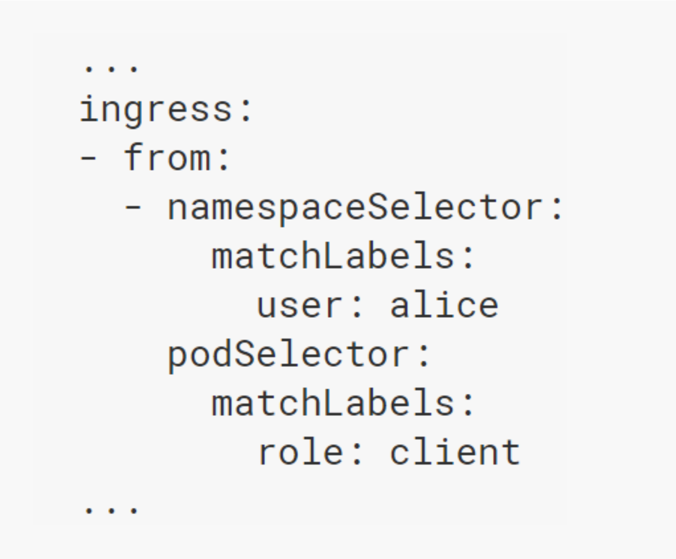
type of policy

source definition

destination definition

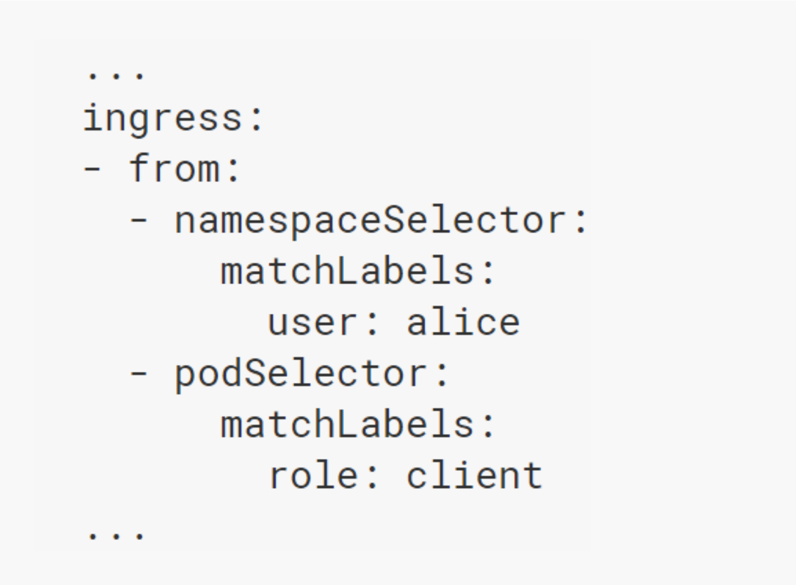
Match Criteria

- The following can be used to match traffic for a network policy:
  - **PodSelector** – Use to match particular pods based on a label
  - **NamespaceSelector** – Used to match an entire namespace
  - **IPBlock** – Used to match a CIDR block
  - **PodSelector + NamespaceSelector** – use to match namespaces and pods for a more granular policy



```
...
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        user: alice
    podSelector:
      matchLabels:
        role: client
...
```

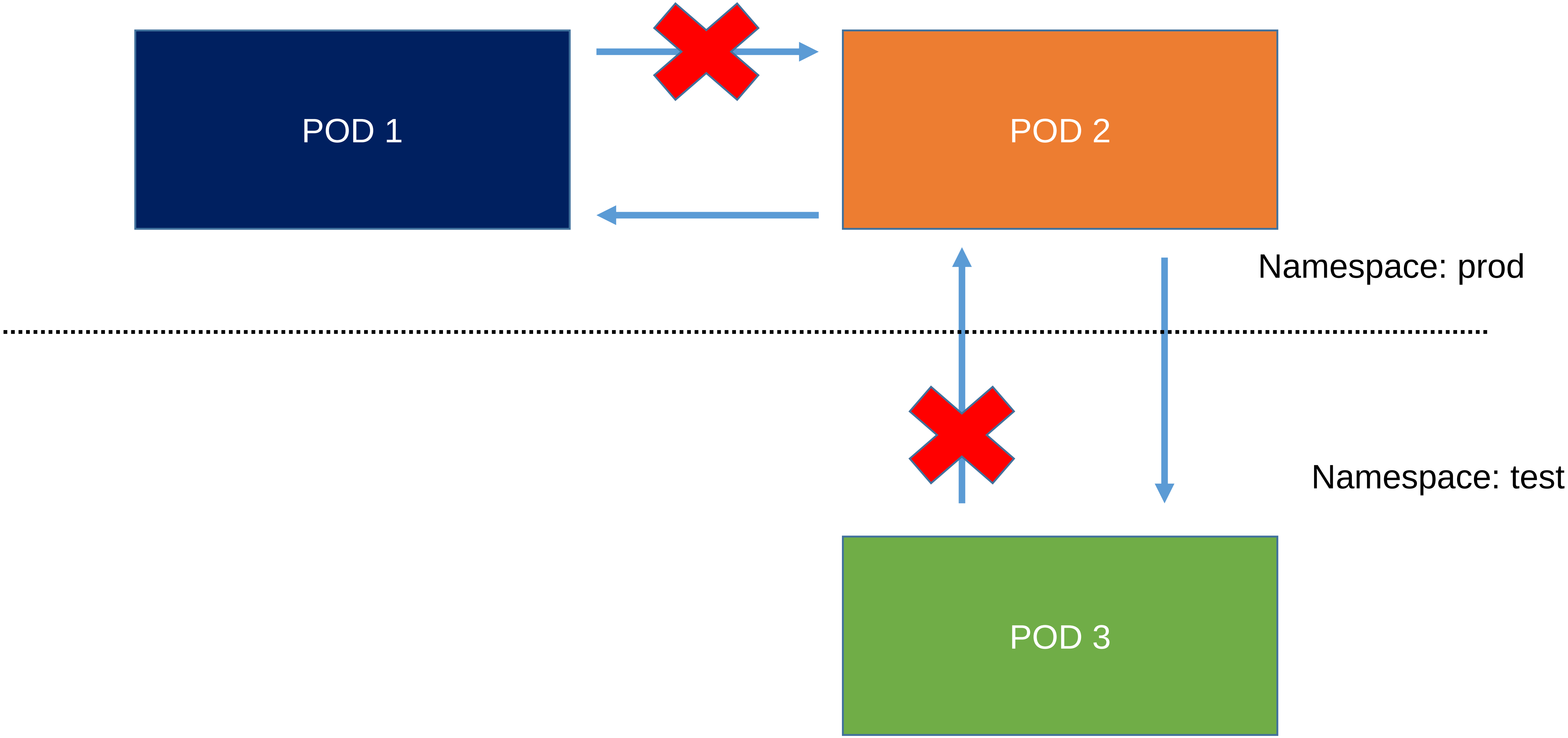
allow connections from Pods with the label role=client in namespaces with the label user=alice



```
...
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        user: alice
  - podSelector:
      matchLabels:
        role: client
...
```

allows connections from Pods in the local Namespace with the label role=client, OR
from any Pod in any namespace with the label user=alice

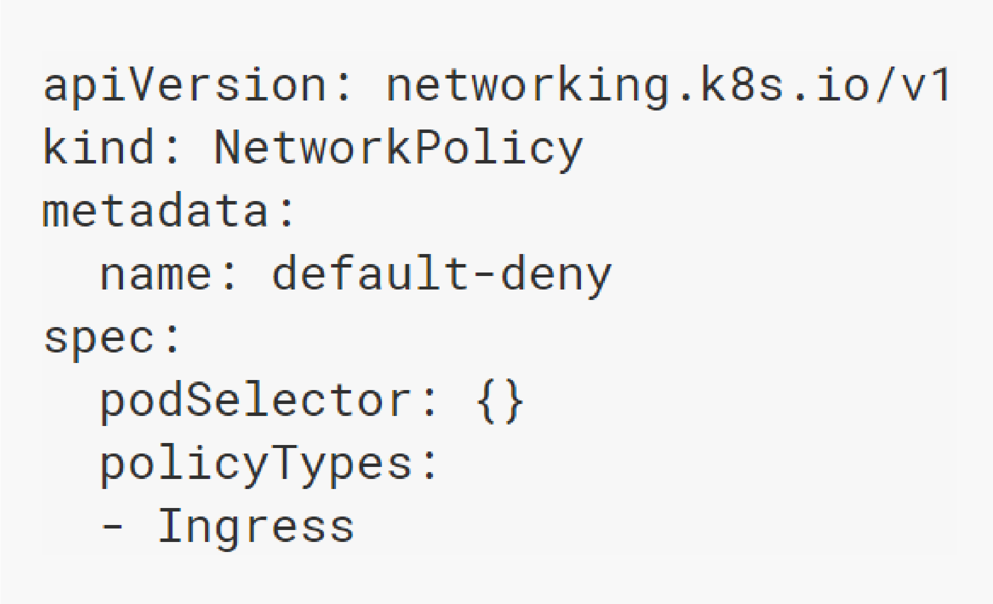
Network Policy Example



Default Policies for Denying Ingress or Egress

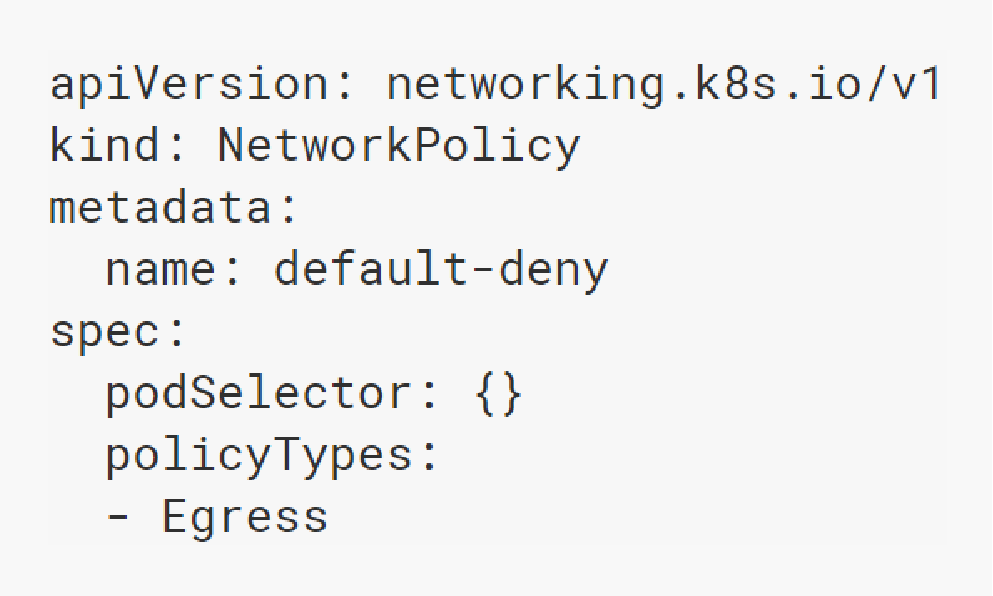
- By default, pods that are not defined in a network policy are allowed to send and receive traffic. You can change this behavior with a default policy.

Deny all Ingress



```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

Deny all Egress



```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Egress
```

Things to watch out for..

- Once a POD is matched with a policy, there is a default deny if the packet doesn’t match a specific policy.

- When enforcing egress policies, **Make sure you allow DNS**. Remember, DNS is used for not only name resolution but service discovery in Kubernetes.

Calico Network Policies

- As of Calico 3.7, Calico has it’s own network policies feature
- Supports much more granular policy definitions:
  - Policy ordering and priority
  - Actions: allow, deny, log, pass
  - Match criteria: ports, protocols and service accounts, IP CIDR, IP Versions and more
- **Optional packet handling controls**: disable connection tracking, apply before DNAT, apply to forwarded traffic and/or locally terminated traffic
- Global Network policies can be created for policies across all namespaces
- Follows the same rules as a Kubernetes service

Calico Global Network Policy Example

- Global Network policies apply to all end points in globally
- In this example, if a pod with the color of ‘blue’ tries to talk to color ‘red’, packets are dropped

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: deny-blue
spec:
  selector: color == 'red'
  ingress:
  - action: Deny
    protocol: TCP
    source:
      selector: color == 'blue'
```

OneCloud
Consulting

Thank You!