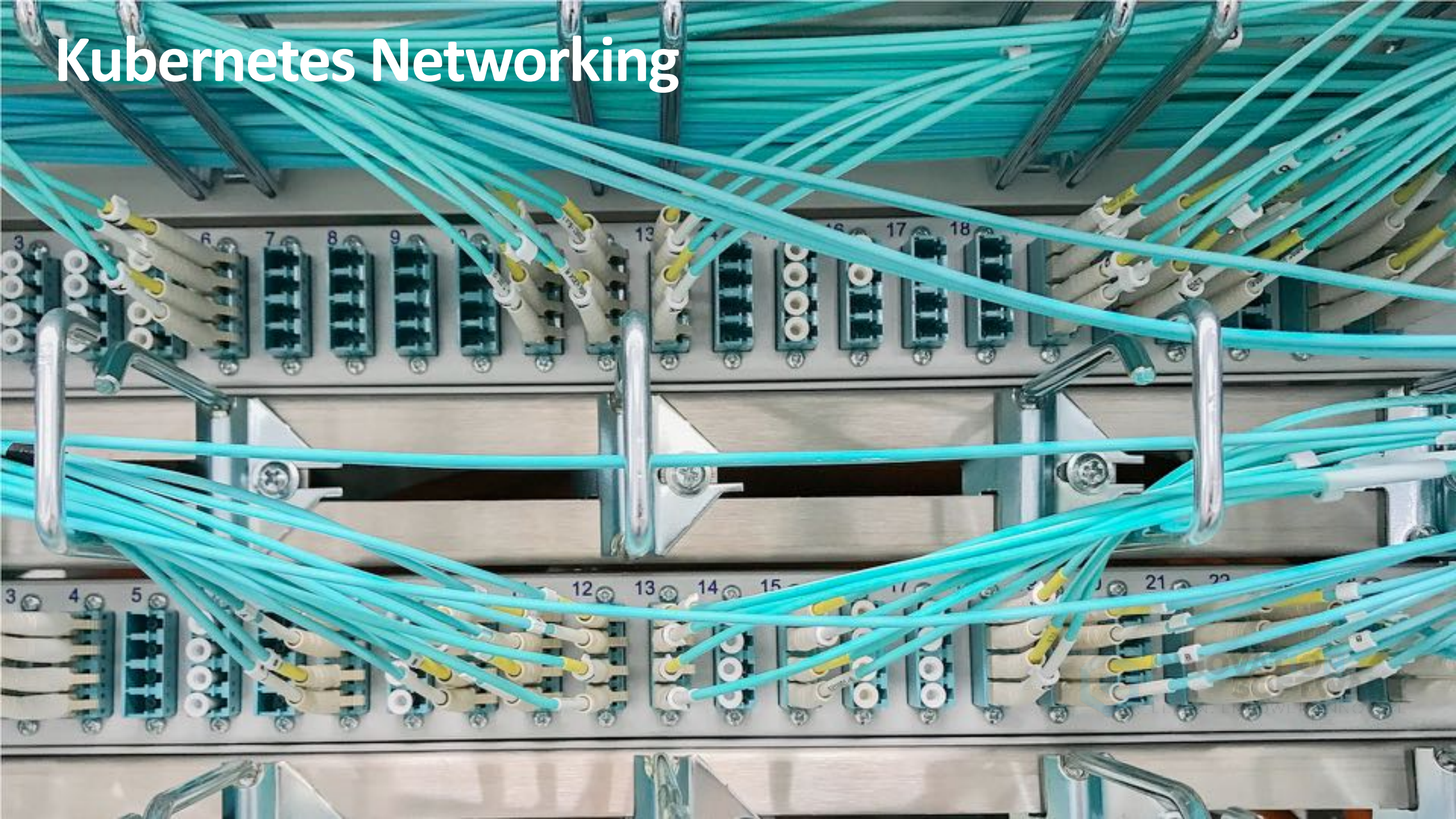

kubernetes

Kubernetes Networking



Networking Problems and Solutions

1. Highly-coupled container-to-container communications => linux
2. Pod-to-Pod communications => various
3. Pod-to-Service communications => service
4. External-to-Service communications => service or ingress

Kubernetes Networking Model Requirements

- 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

Container to Container Communication

- Containers live inside of pods
- Group of one or more containers that are always co-located and co-scheduled, and run in a shared context
- Containers within a POD share an IP address and port space, and can find each other via localhost (net namespace).
- Communicate with each other using standard inter-process communications (ipc namespace).

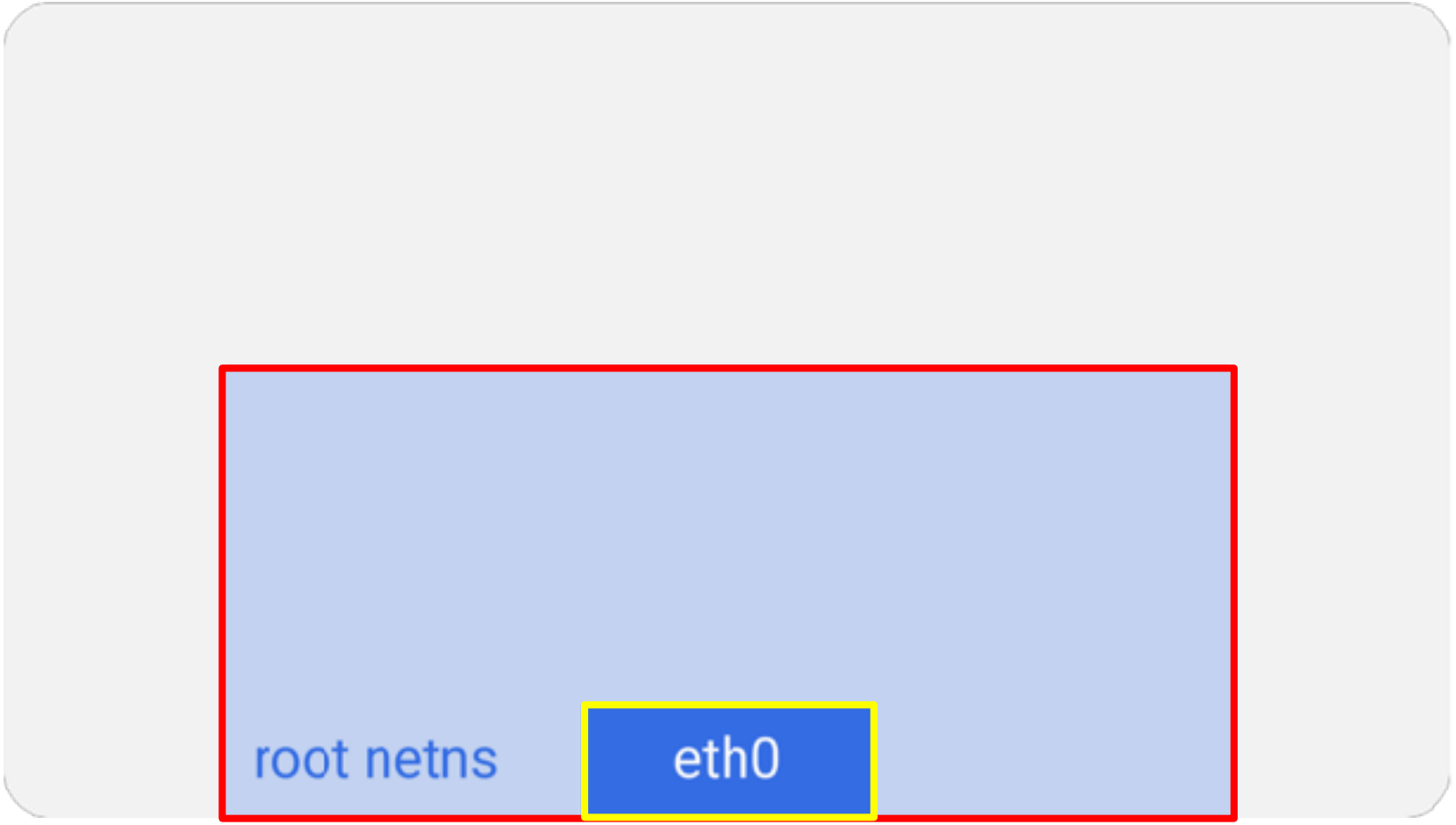
Pod to Pod Requirements

- Pods can communicate with all other pods without NAT
- Regardless of which host they land on
- Every pod gets its own IP address
- No need to explicitly create links between pods

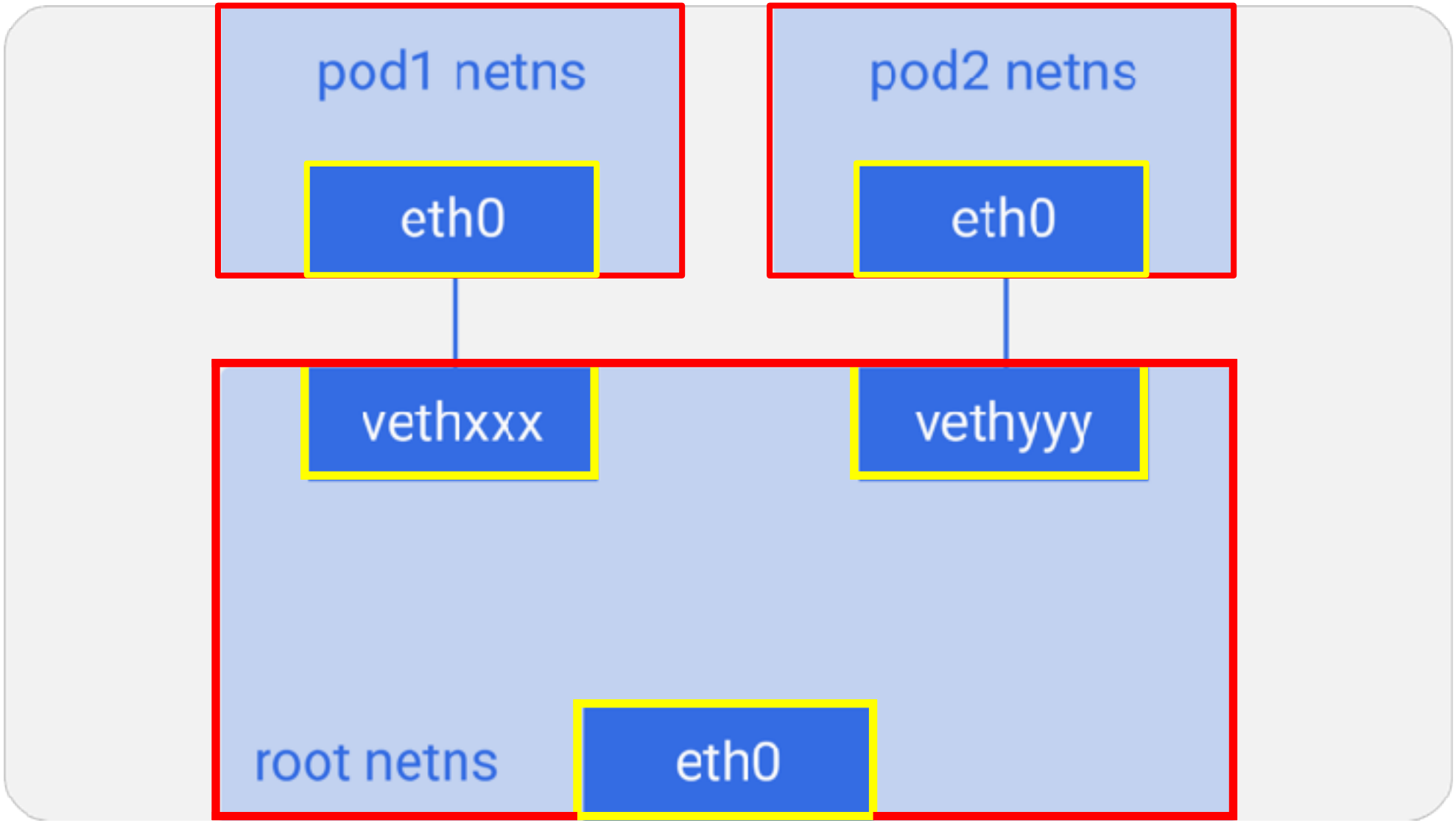
Pod Networking Approaches

- **Using an overlay network**
  - An overlay network obscures the underlying network architecture from the pod network through traffic encapsulation (for example vxlan).
  - Encapsulation reduces performance, though exactly how much depends on your solution.
- **Without an overlay network**
  - Configure the underlying network fabric (switches, routers, etc.) to be aware of pod IP addresses.
  - This does not require the encapsulation provided by an overlay, and so can achieve better performance.

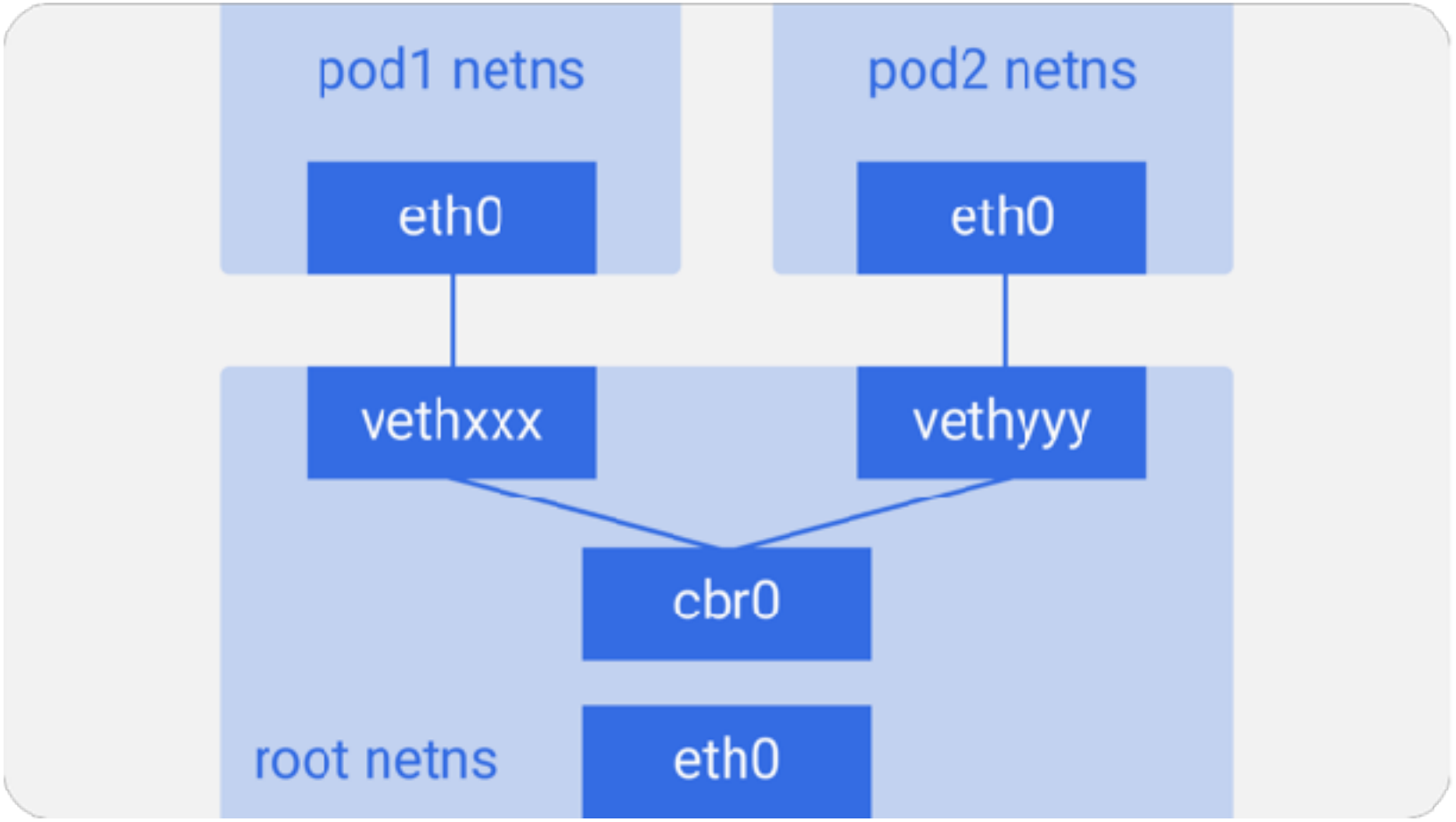
Node Namespace



Node Namespaces

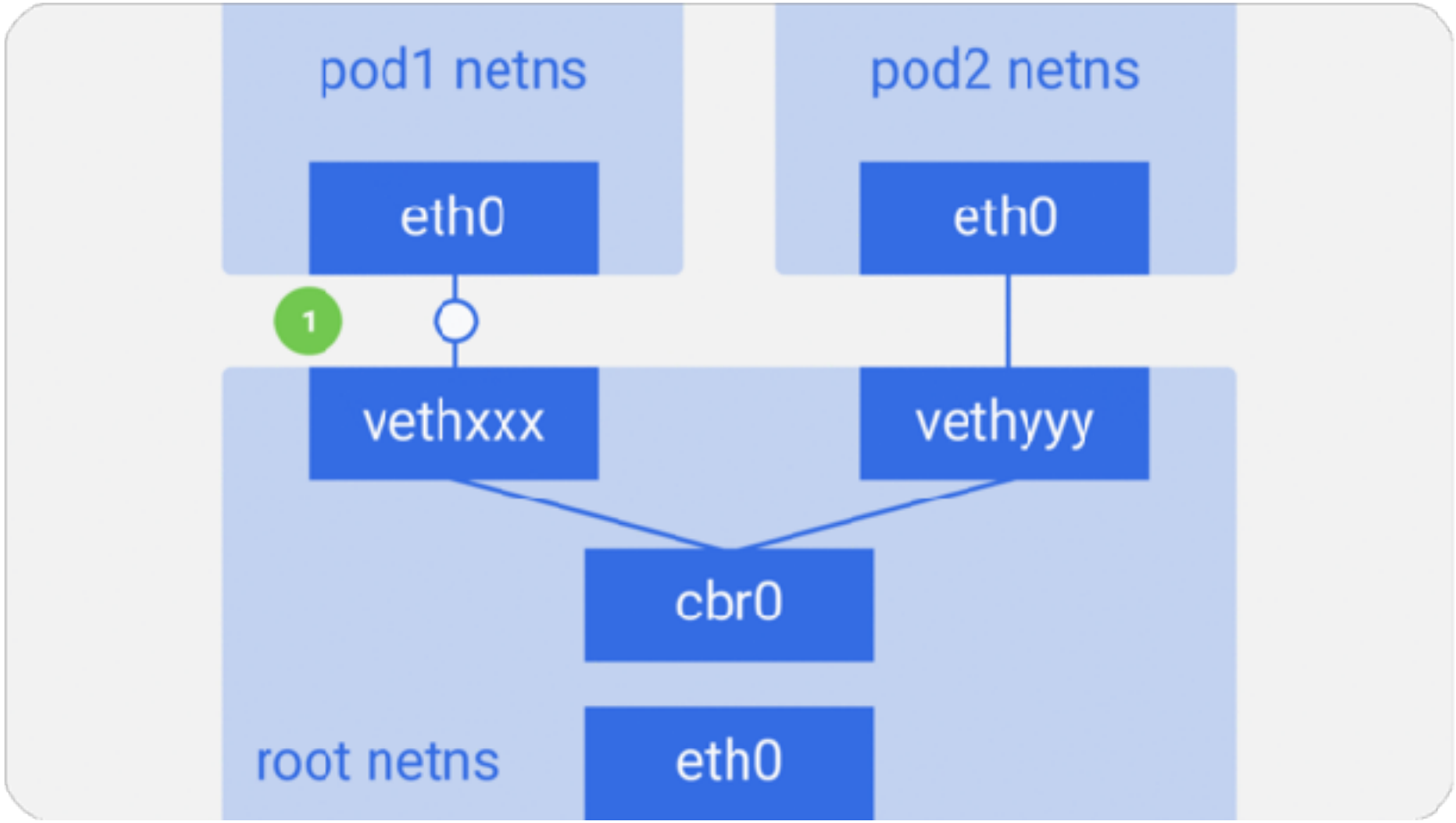


Pod to Host


pod1 netns
pod2 netns
eth0
eth0
vethxxx
vethyyy
cbr0
root netns
eth0

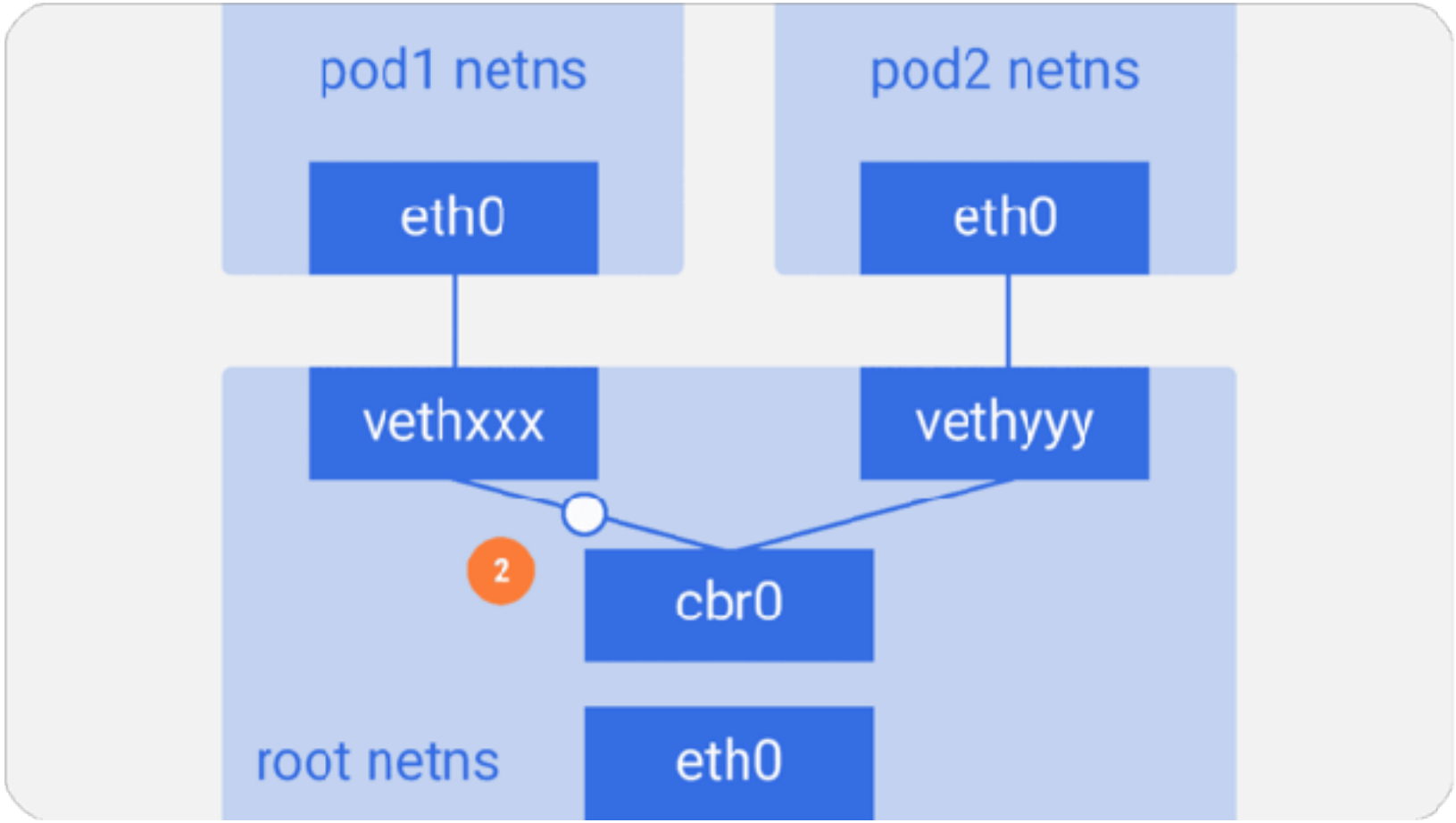
Single Host POD to POD Network

1. Packet leaves pod1's netns at eth0 and enters the root netns at vethxxx.



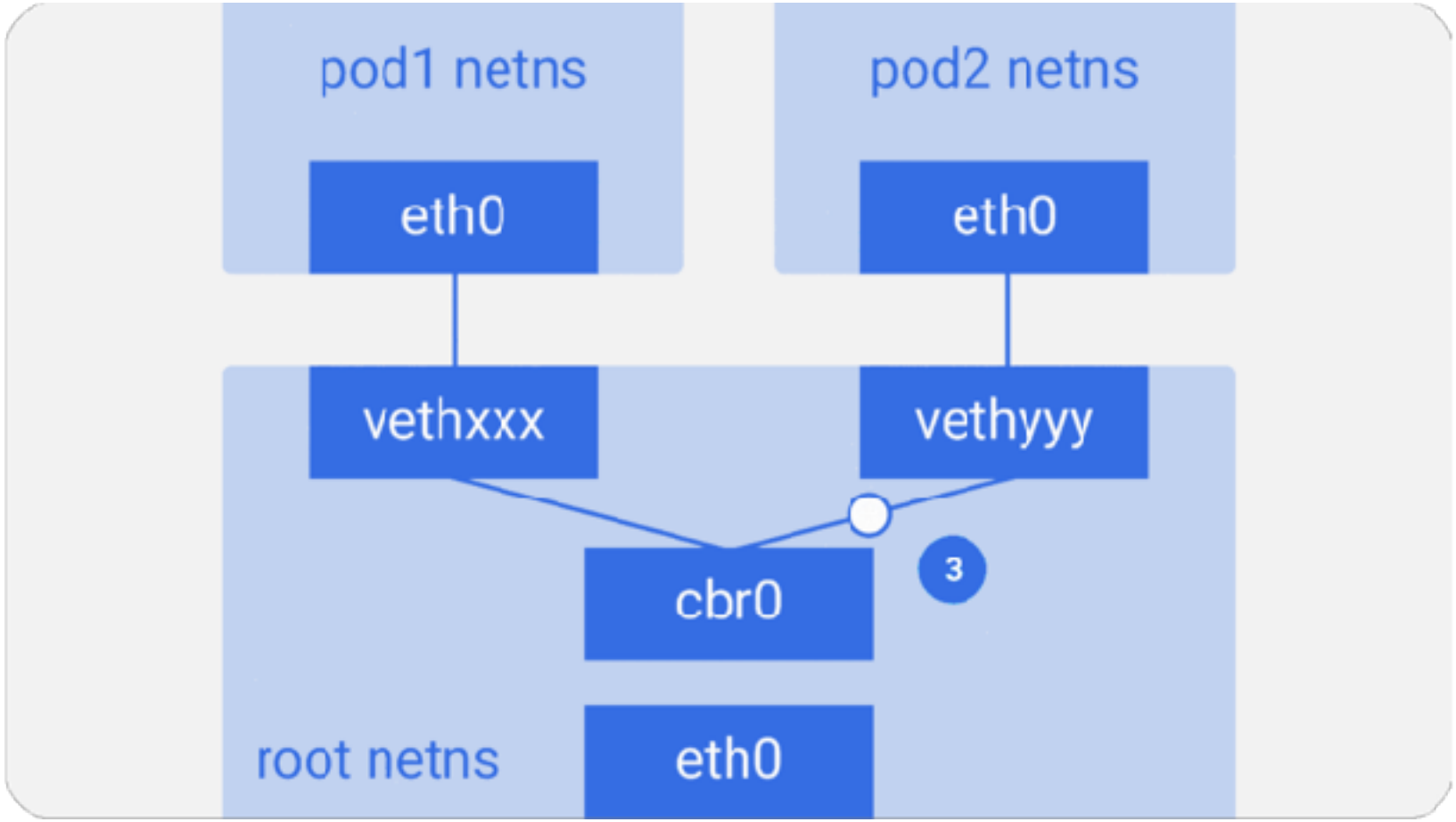
Single Host POD to POD Networking

2. It's passed on to cbr0, which discovers the destination using an ARP request, saying "who has this IP?"



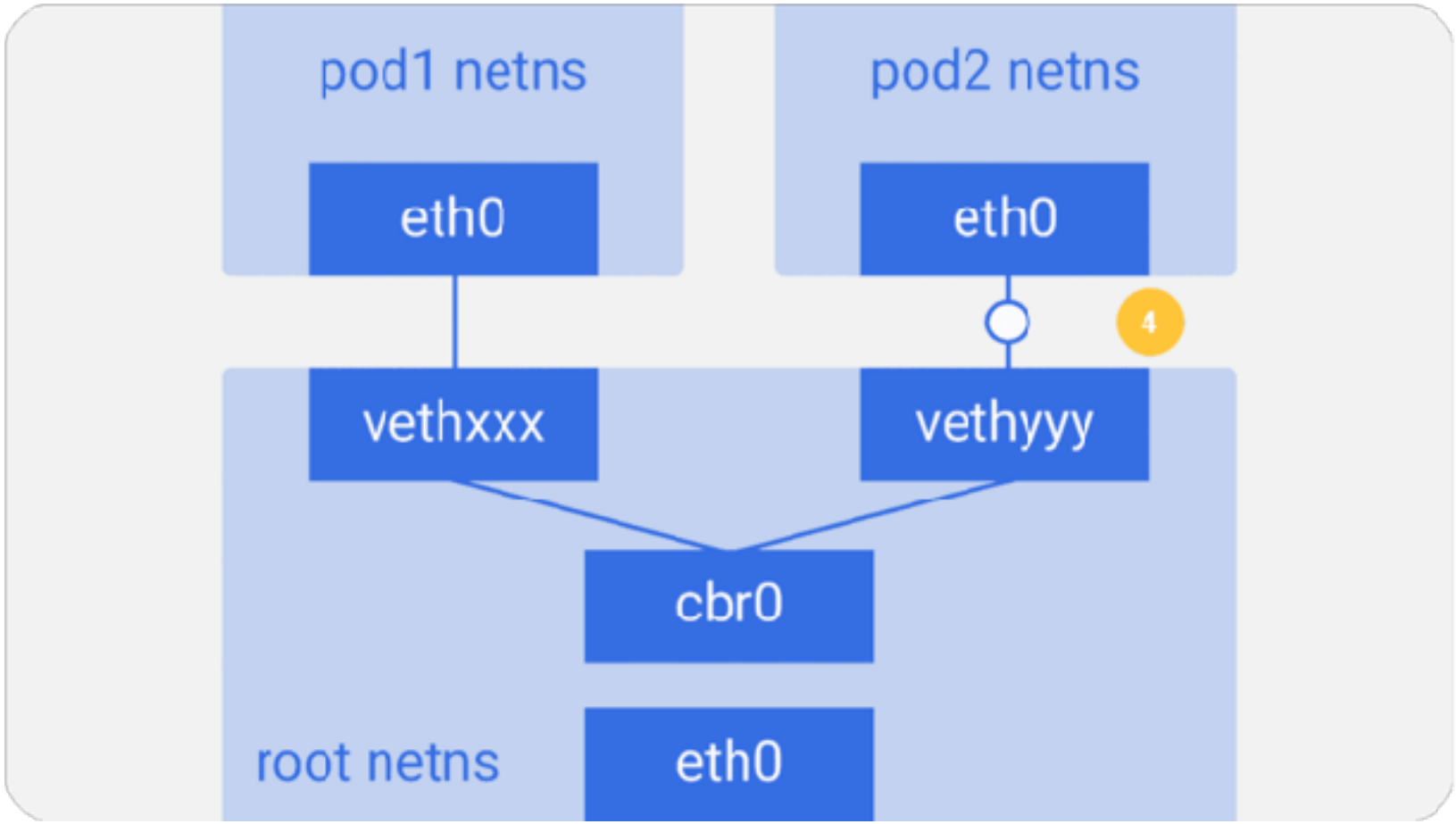
Single Host POD to POD Networking

3. vethyyy says it has that IP, so the bridge knows where to forward the packet.



Single Host POD to POD Networking

4. The packet reaches vethyyy, crosses the pipe-pair and reaches pod2's netns.

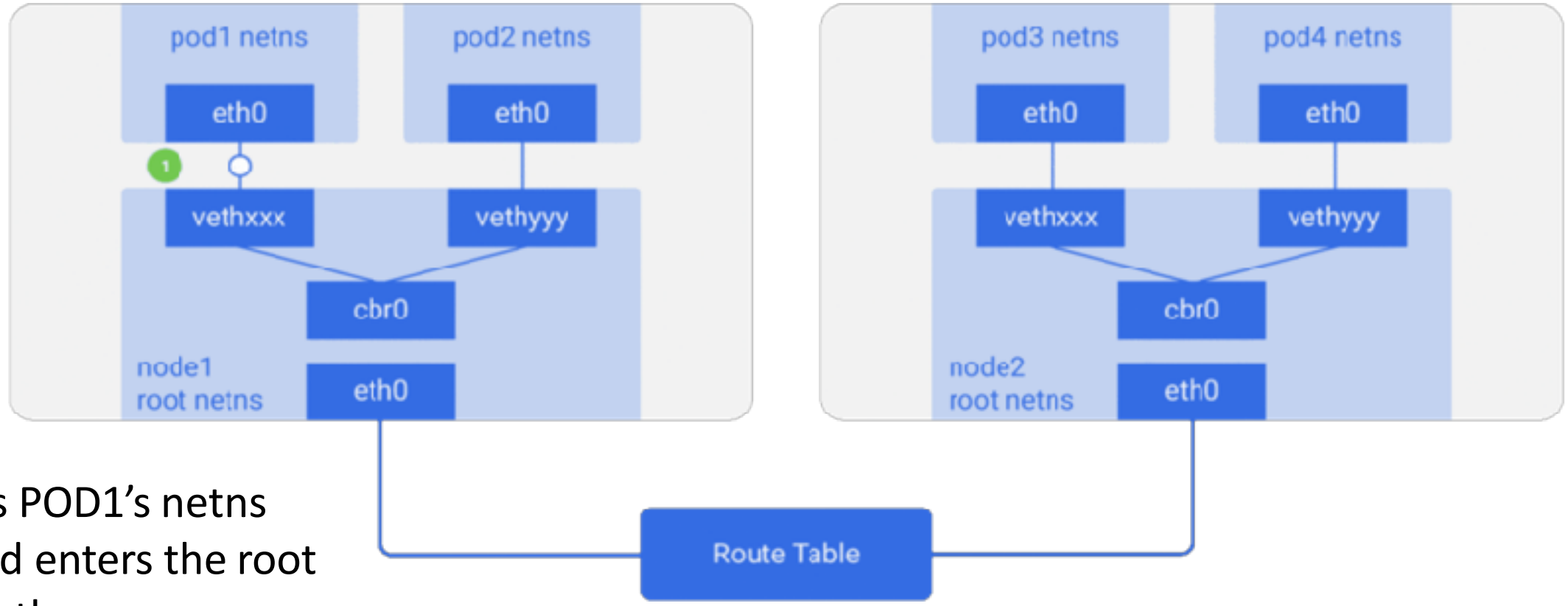


Kubernetes Inter-node Communication

Inter-node Communication

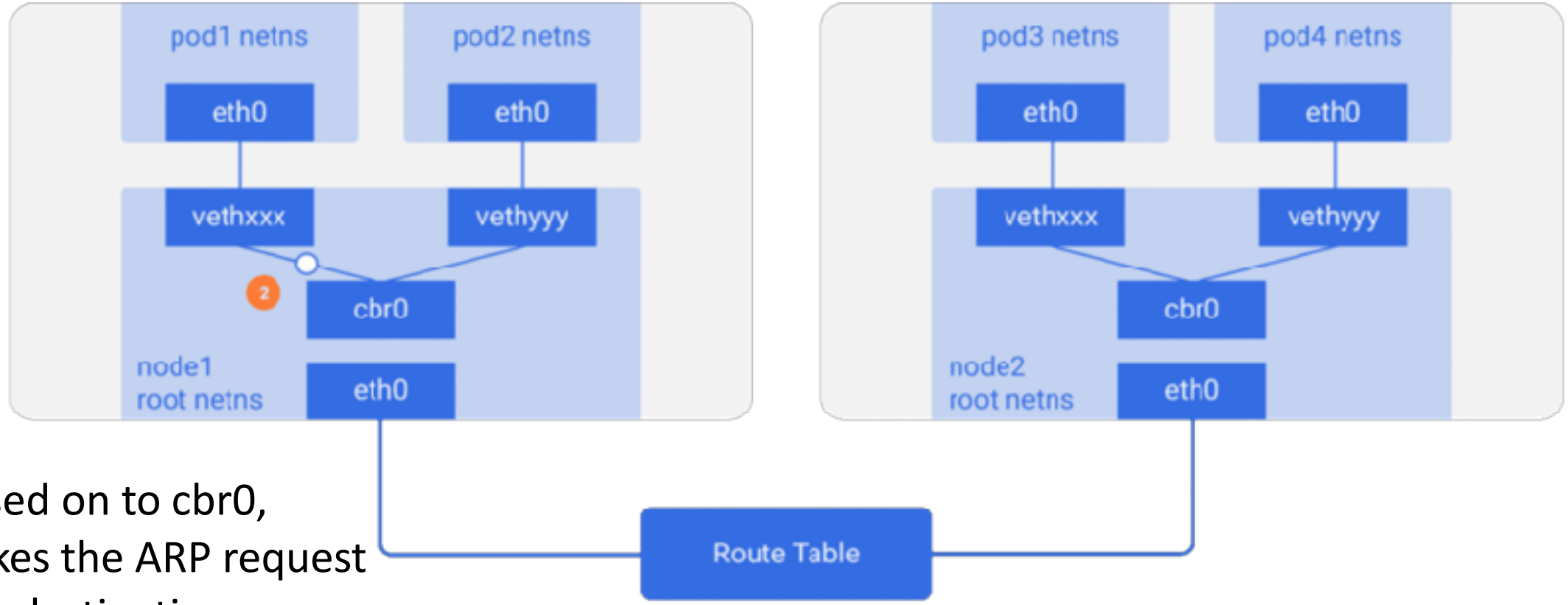
- PODs must be reachable across nodes
- Kubernetes doesn’t care how:
  - L2 (ARP across nodes)
  - L3 (IP routing across nodes)
  - Overlay networks

Inter-node Communication



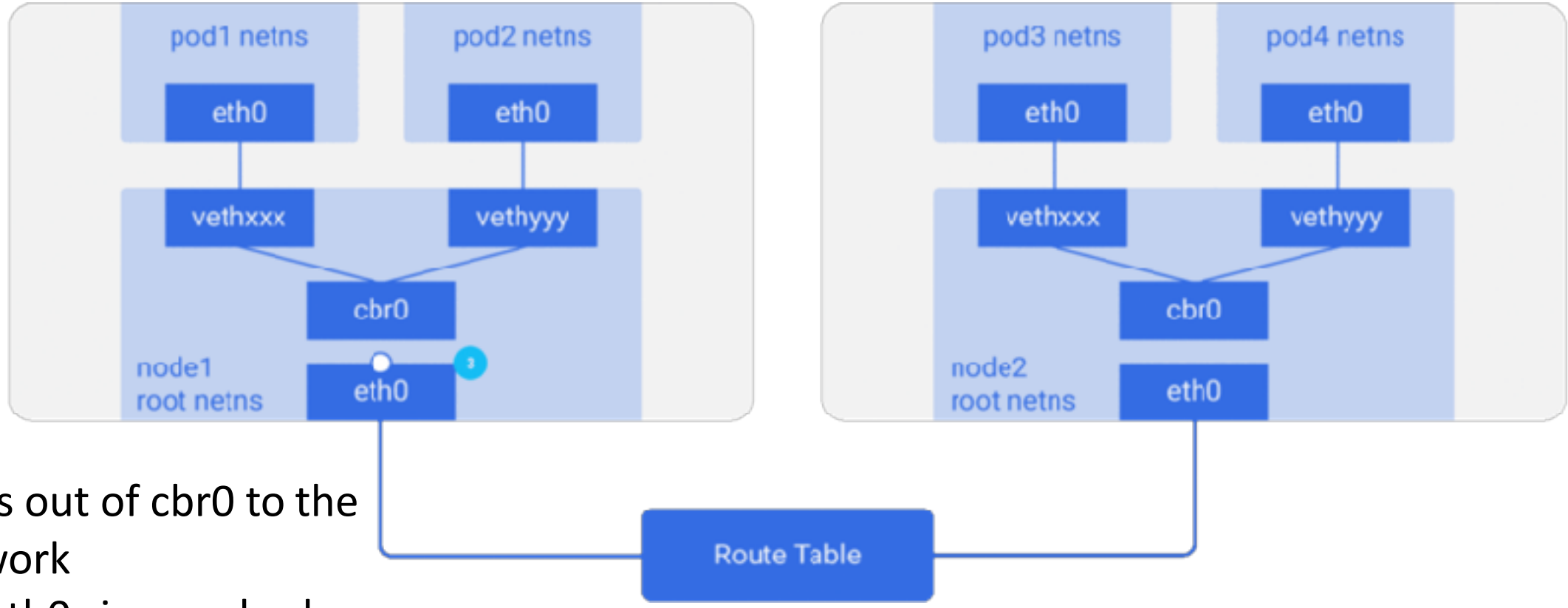
1. It leaves POD1's netns at eth0 and enters the root netns at vethxxx.

Inter-node Communication



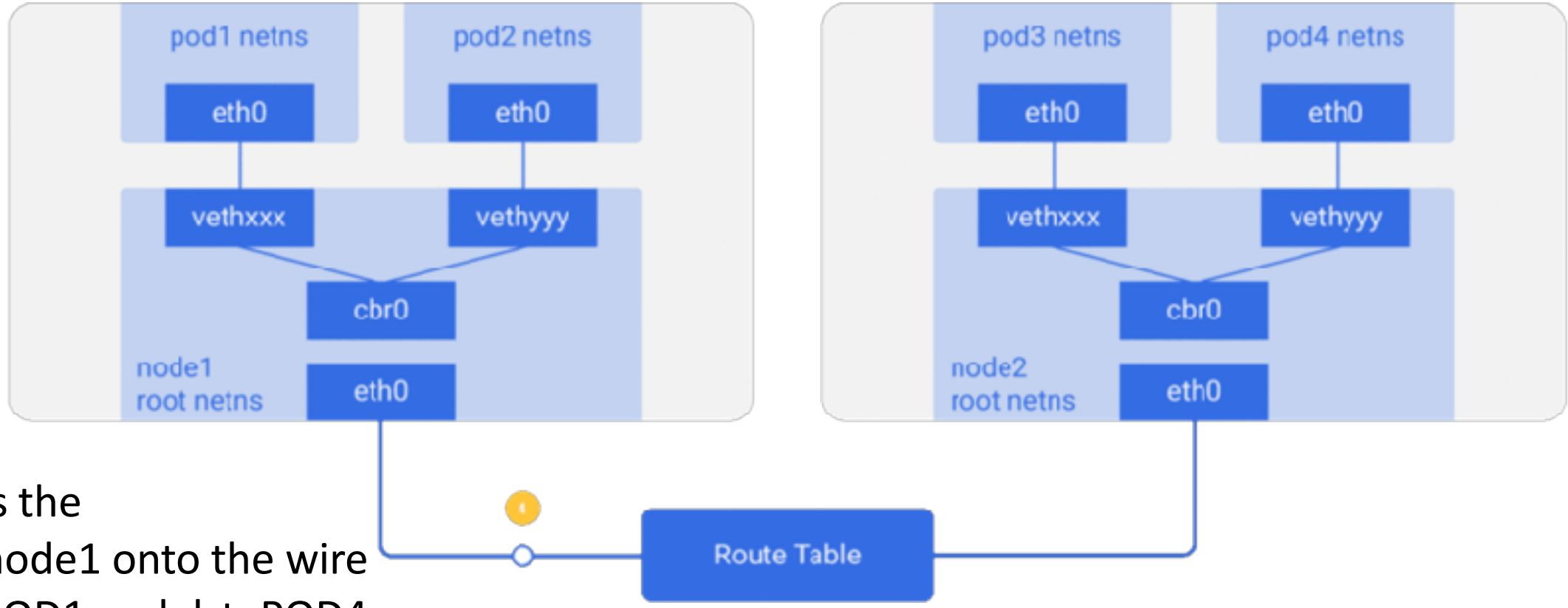
2. It's passed on to cbr0, which makes the ARP request to find the destination.

Inter-node Communication



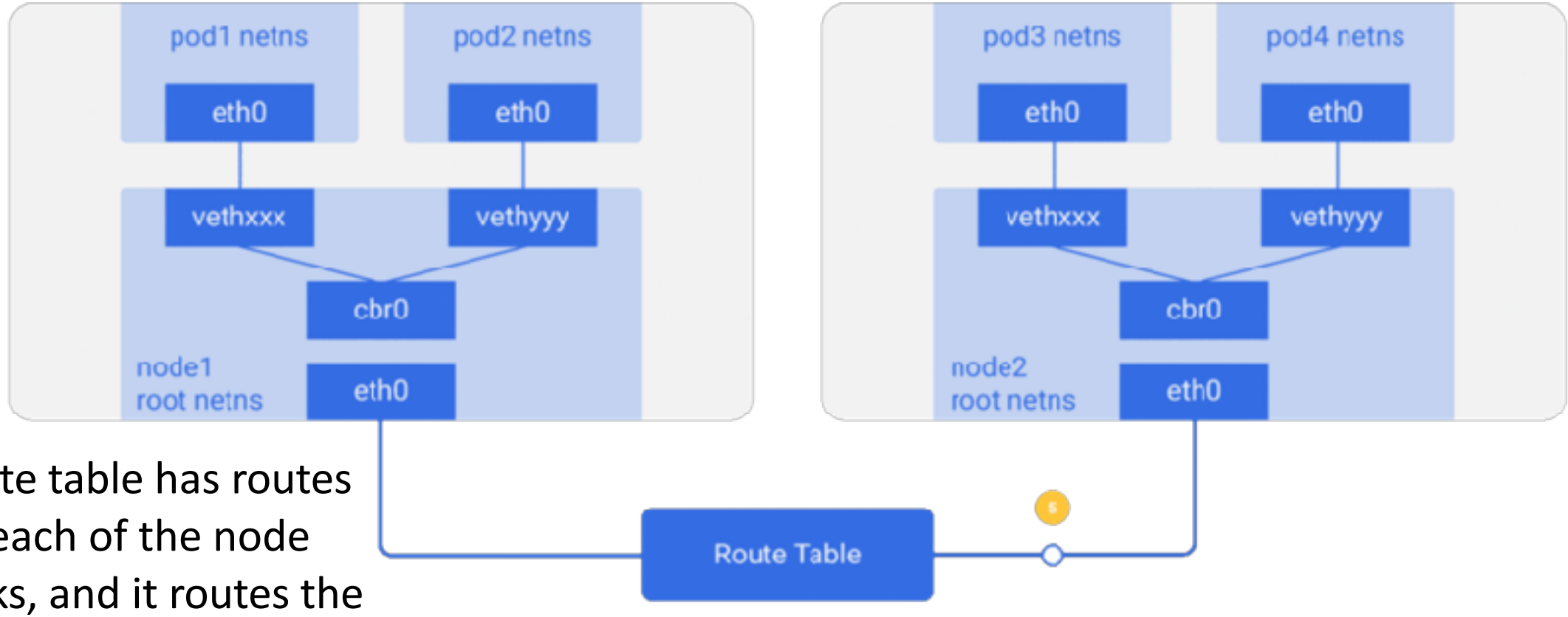
3. It comes out of cbr0 to the main network interface eth0 since nobody on this node has the IP address for POD4.

Inter-node Communication



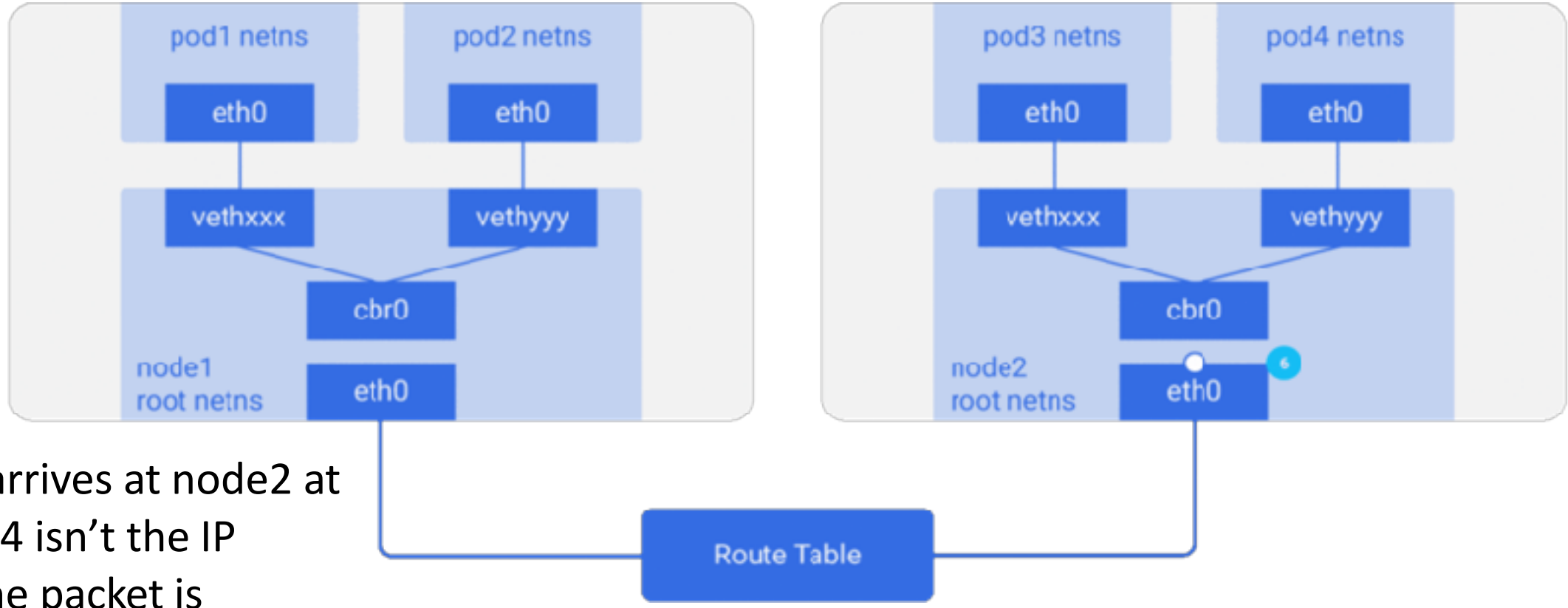
4. It leaves the machine node1 onto the wire with src=POD1 and dst=POD4.

Inter-node Communication



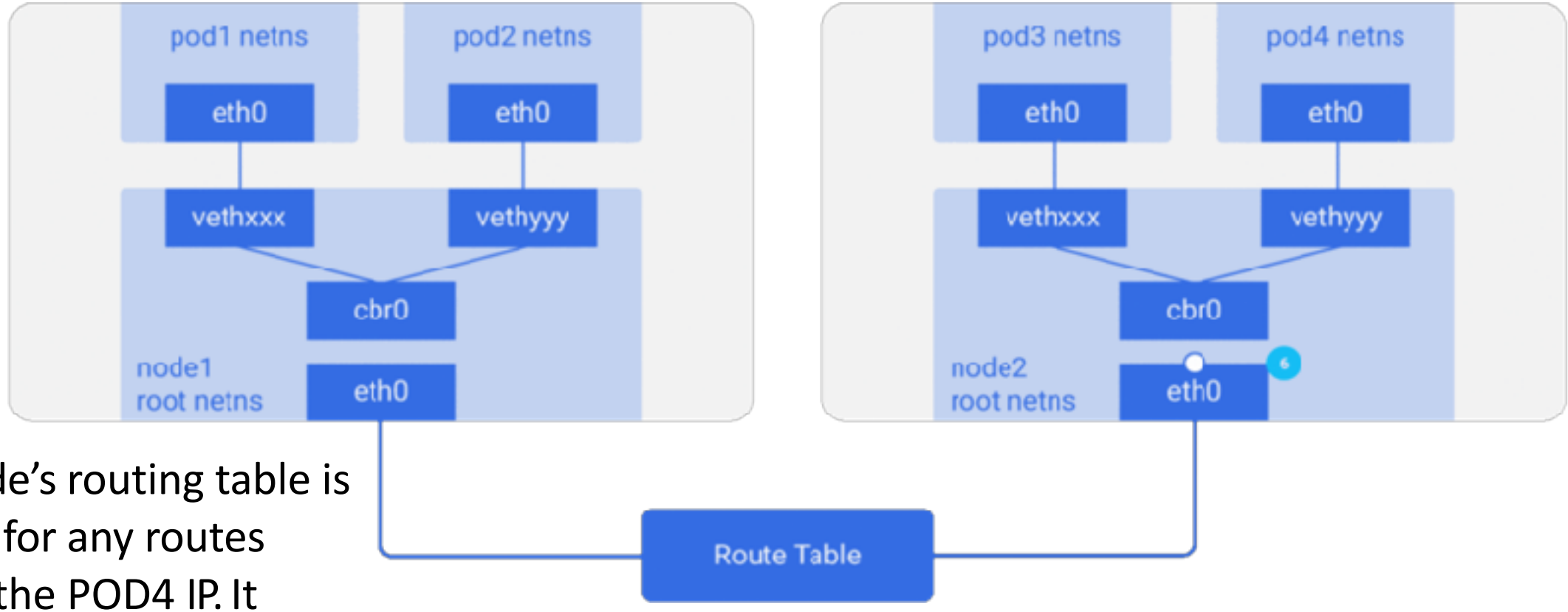
5. The route table has routes setup for each of the node CIDR blocks, and it routes the packet to the node whose CIDR block contains the POD4 IP.

Inter-node Communication



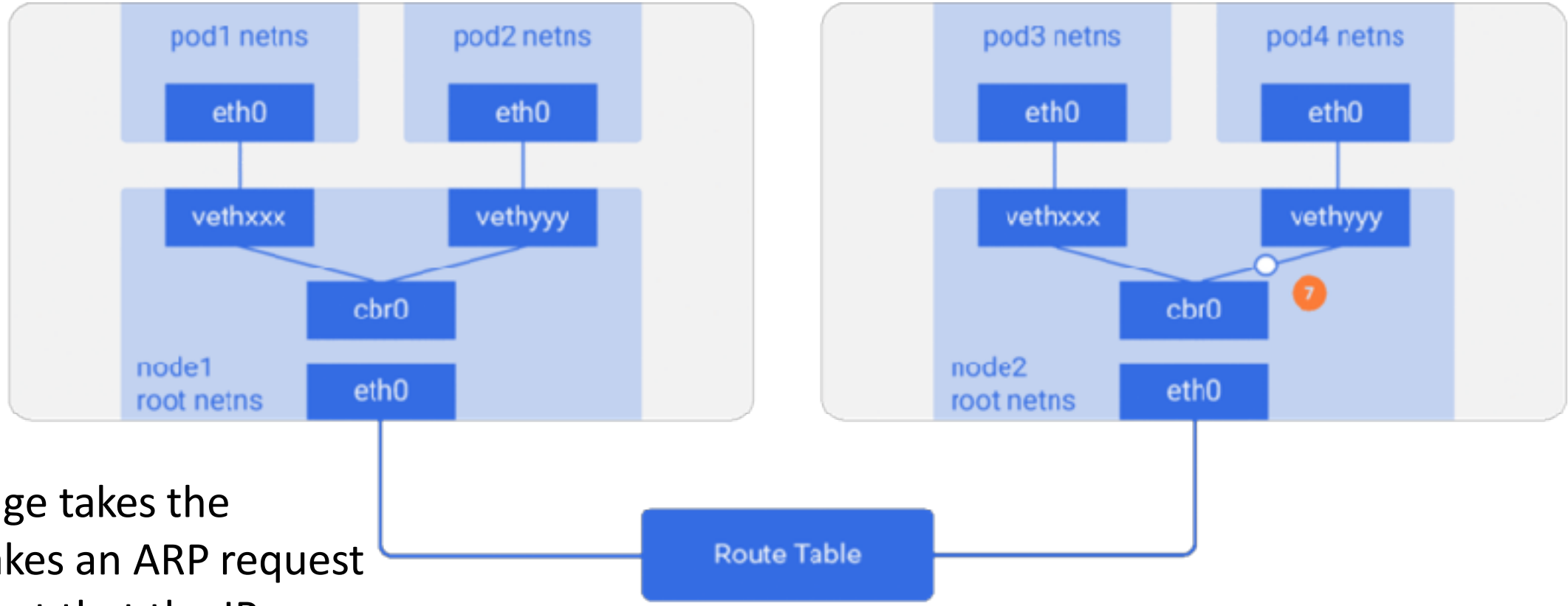
6. Packet arrives at node2 at eth0. POD4 isn't the IP of eth0, the packet is forwarded to cbr0, nodes are configured with IP forwarding enabled.

Inter-node Communication



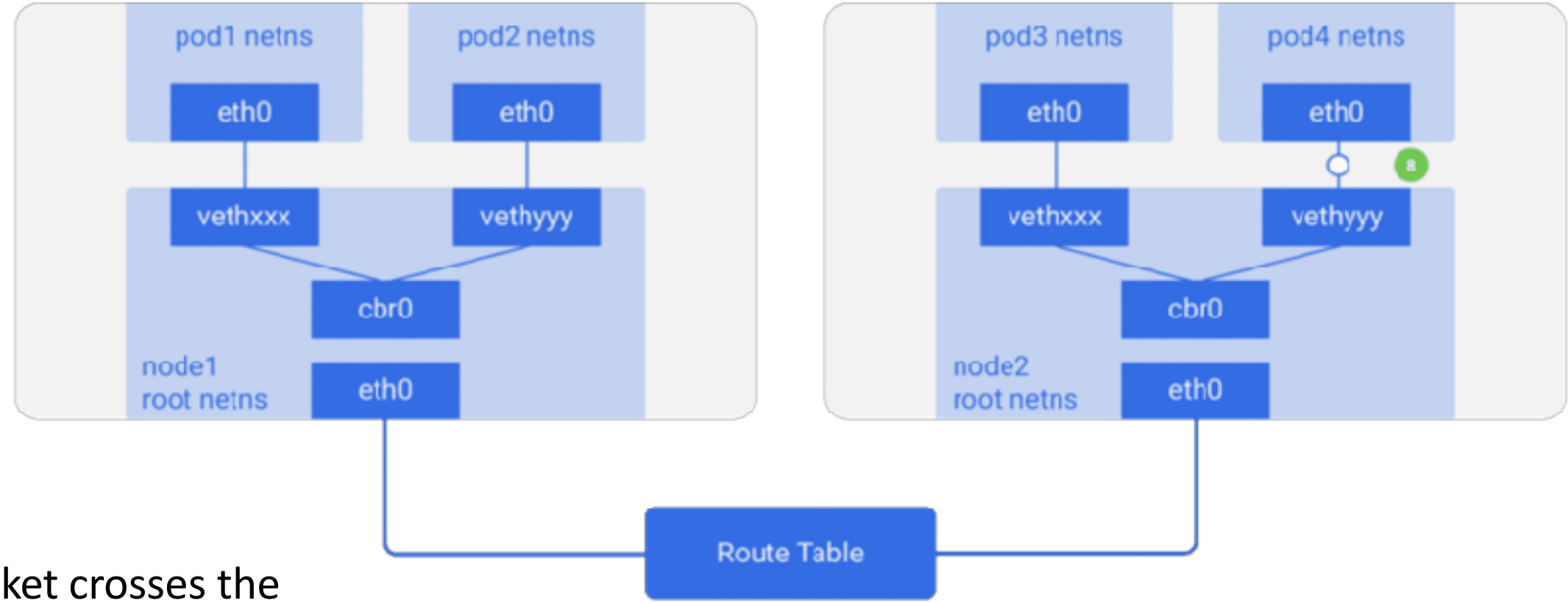
6. The node's routing table is looked up for any routes matching the POD4 IP. It finds cbr0 as the destination for this node's CIDR block.

Inter-node Communication



7. The bridge takes the packet, makes an ARP request and finds out that the IP belongs to vethyyy.

Inter-node Communication



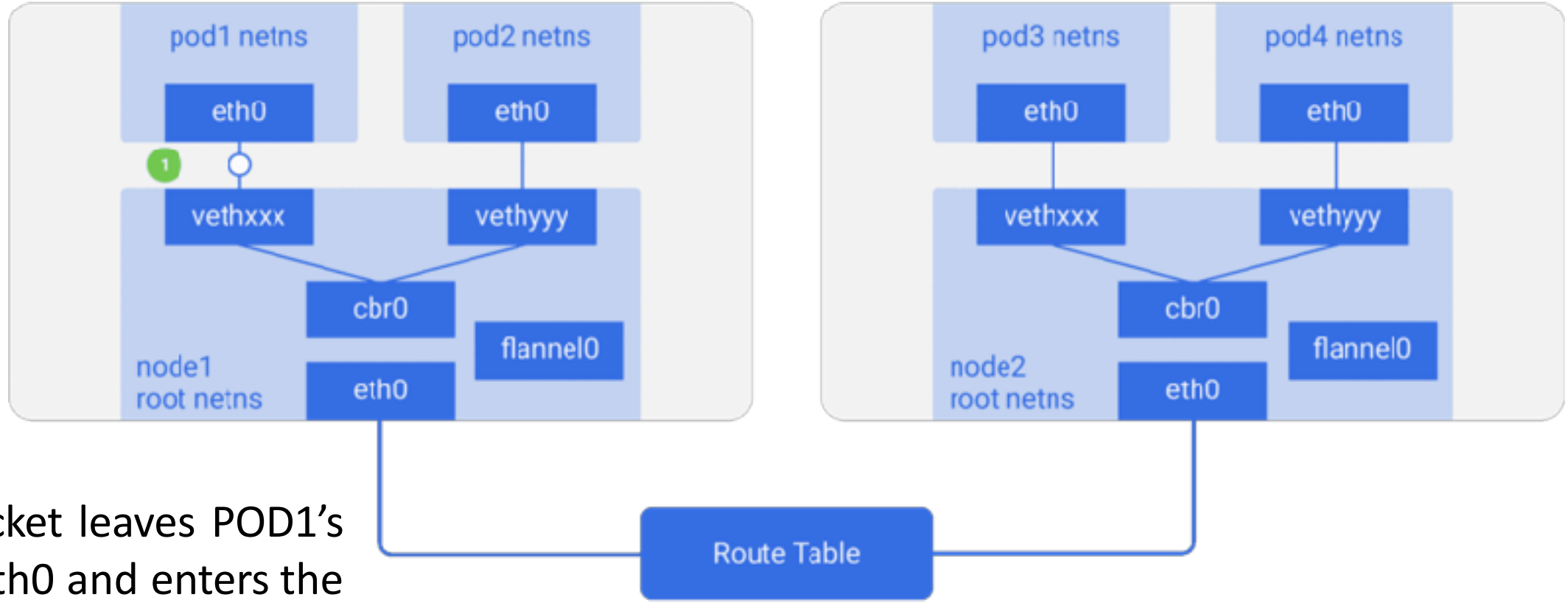
8. The packet crosses the pipe-pair and reaches POD4

Kubernetes Overlay Networking

Overlay Networking

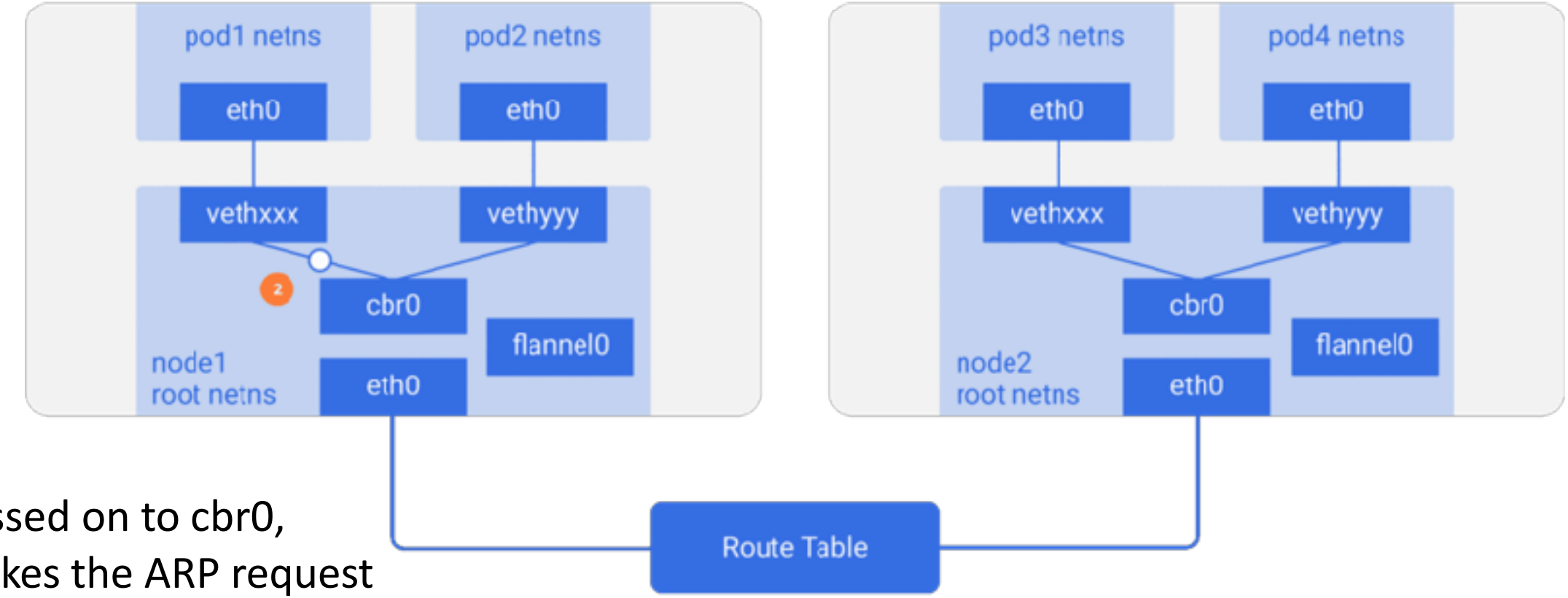
- Not required but very common
- Provide virtual IP space
- Avoid route limits (Public Cloud)
- Encapsulating a packet-in-packet which traverses the native network across nodes.
- Can reduce throughput

Flannel Overlay Networking



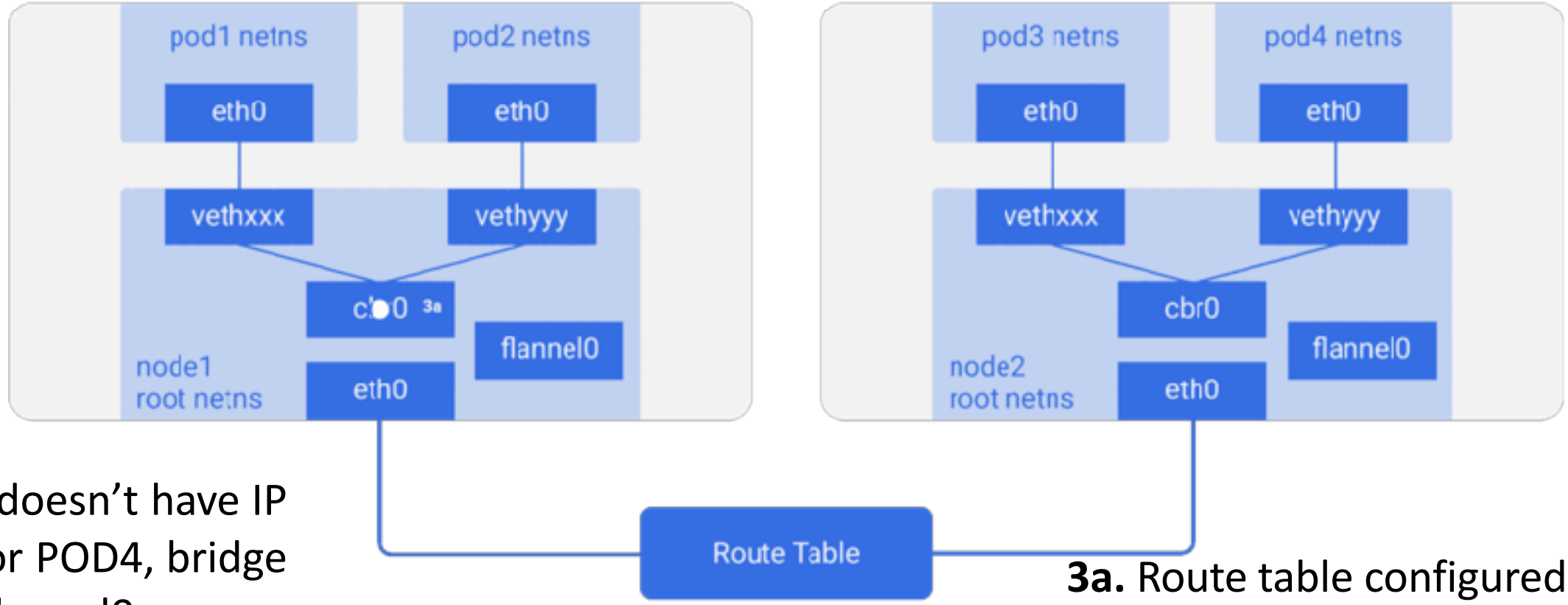
1. The packet leaves POD1's netns at eth0 and enters the root netns at vethxxx.

Flannel Overlay Networking



2. It's passed on to cbr0, which makes the ARP request to find the destination.

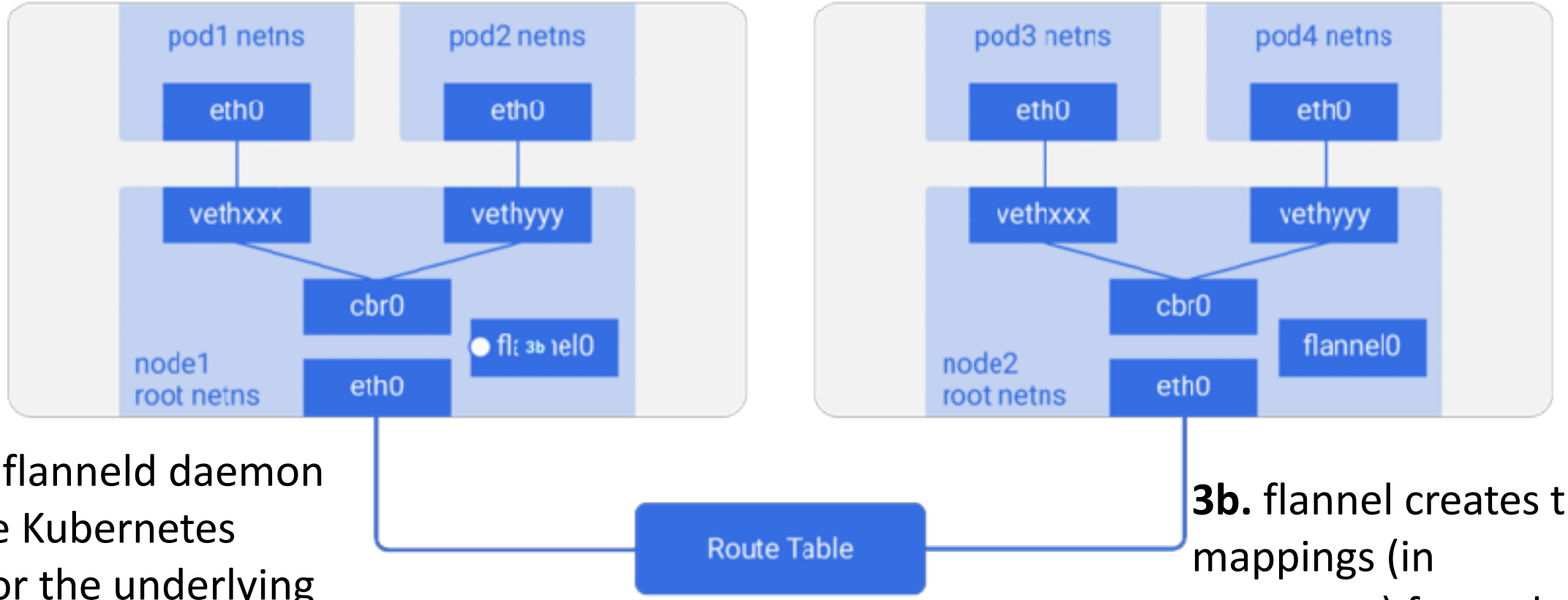
Flannel Overlay Networking



3a. Node doesn't have IP address for POD4, bridge sends to flannel0.

3a. Route table configured with flannel0 as the target for the pod network range .

Flannel Overlay Networking



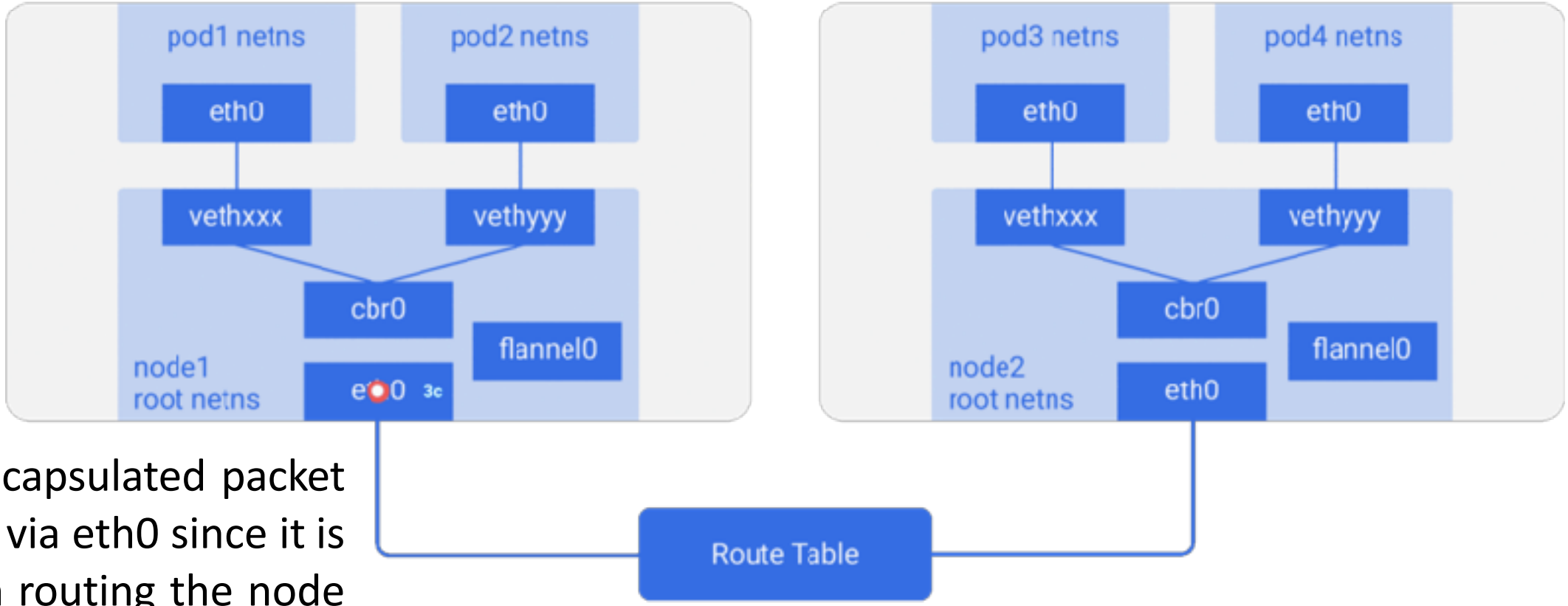
3b. As the flanneld daemon talks to the Kubernetes apiserver or the underlying etcd, it knows about all the pod IPs, what nodes they're on.

3b. flannel creates the mappings (in userspace) for pods IPs to node IPs.

Flannel Overlay Networking

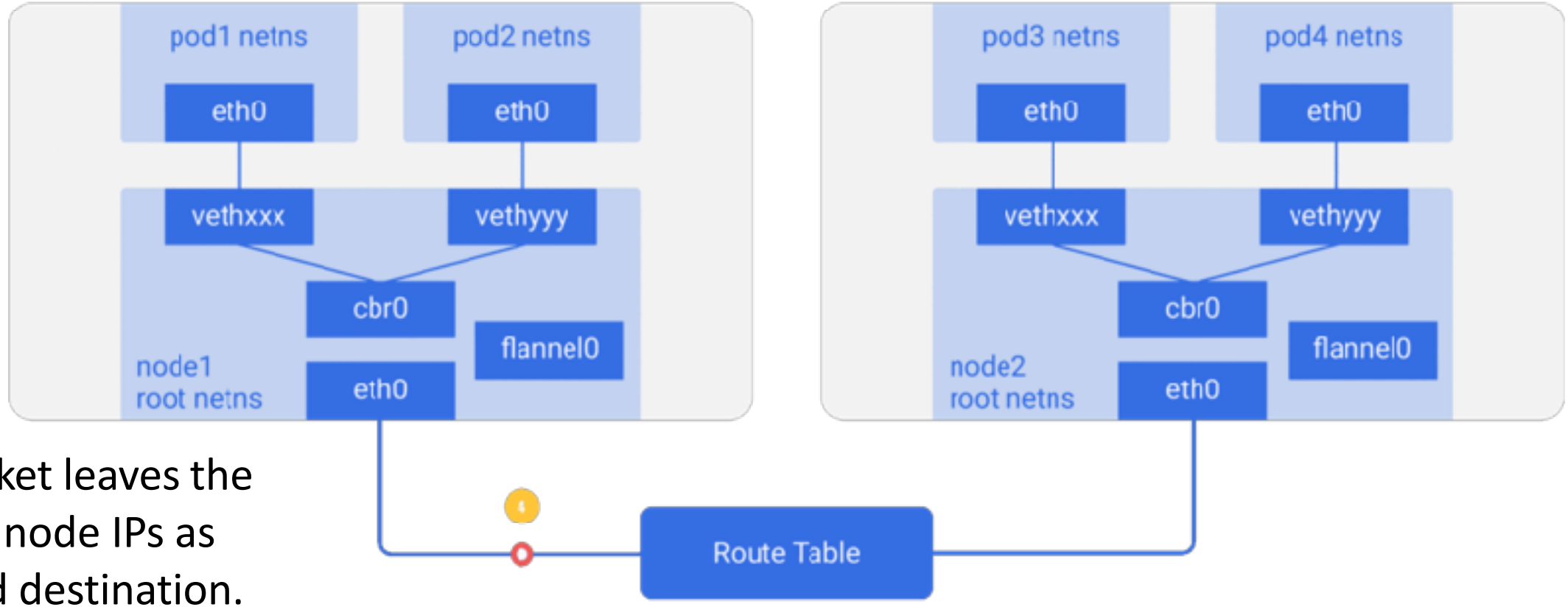
flannel0 takes this packet and wraps it in a UDP packet with extra headers changing the source and destinations IPs to the respective nodes, and sends it to a special vxlan port (generally 8472).

Flannel Overlay Networking



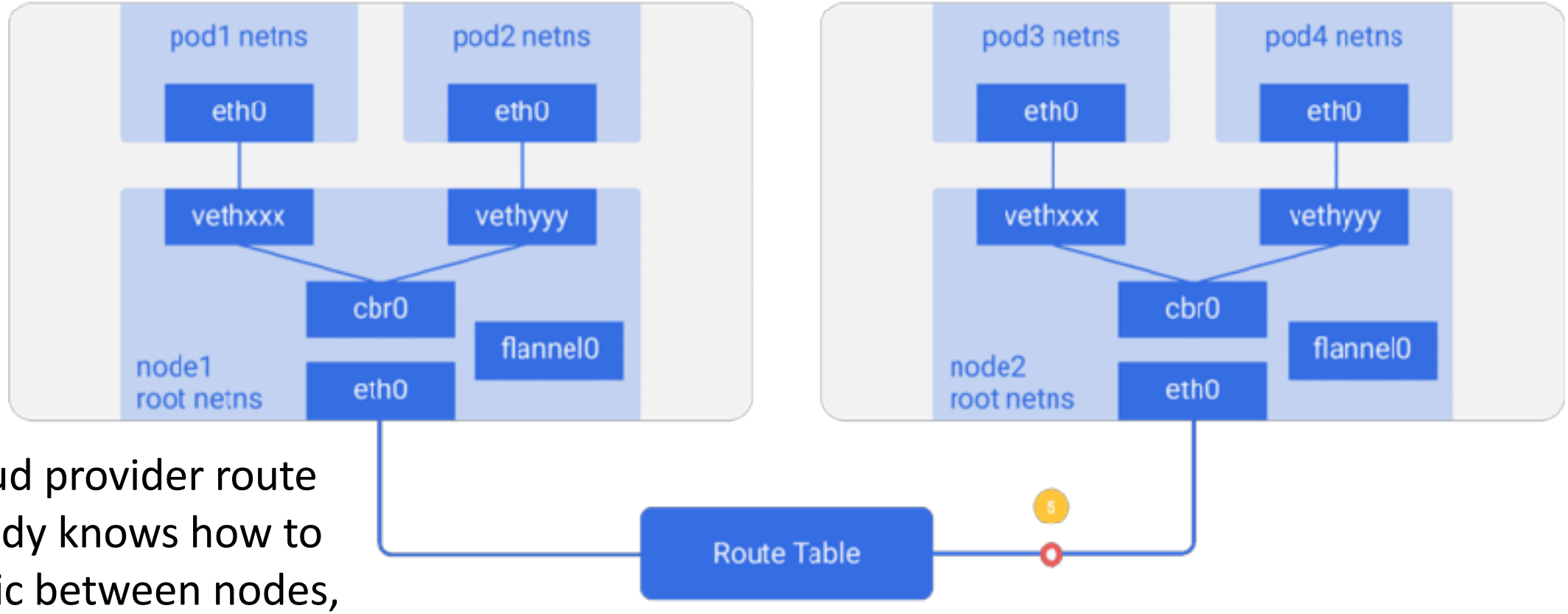
3c. The encapsulated packet is sent out via eth0 since it is involved in routing the node traffic.

Flannel Overlay Networking



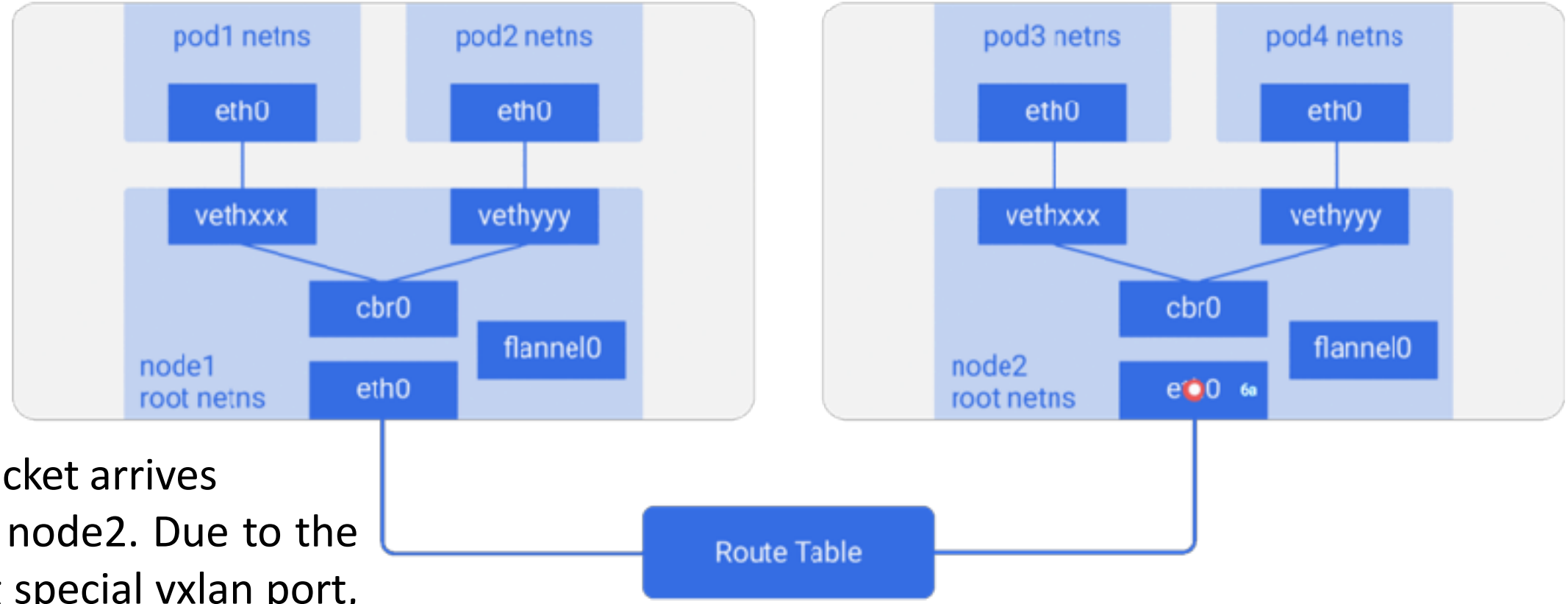
4. The packet leaves the node with node IPs as source and destination.

Flannel Overlay Networking



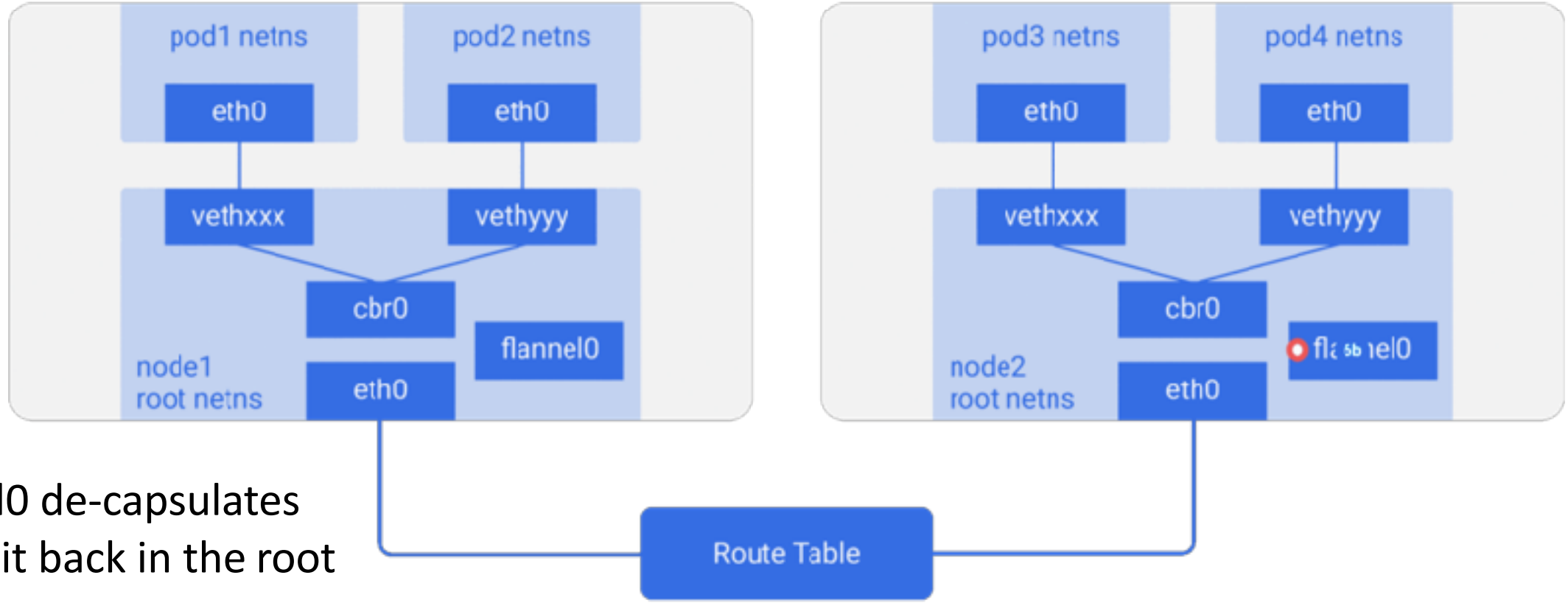
5. The cloud provider route table already knows how to route traffic between nodes, so it sends the packet to destination node2.

Flannel Overlay Networking



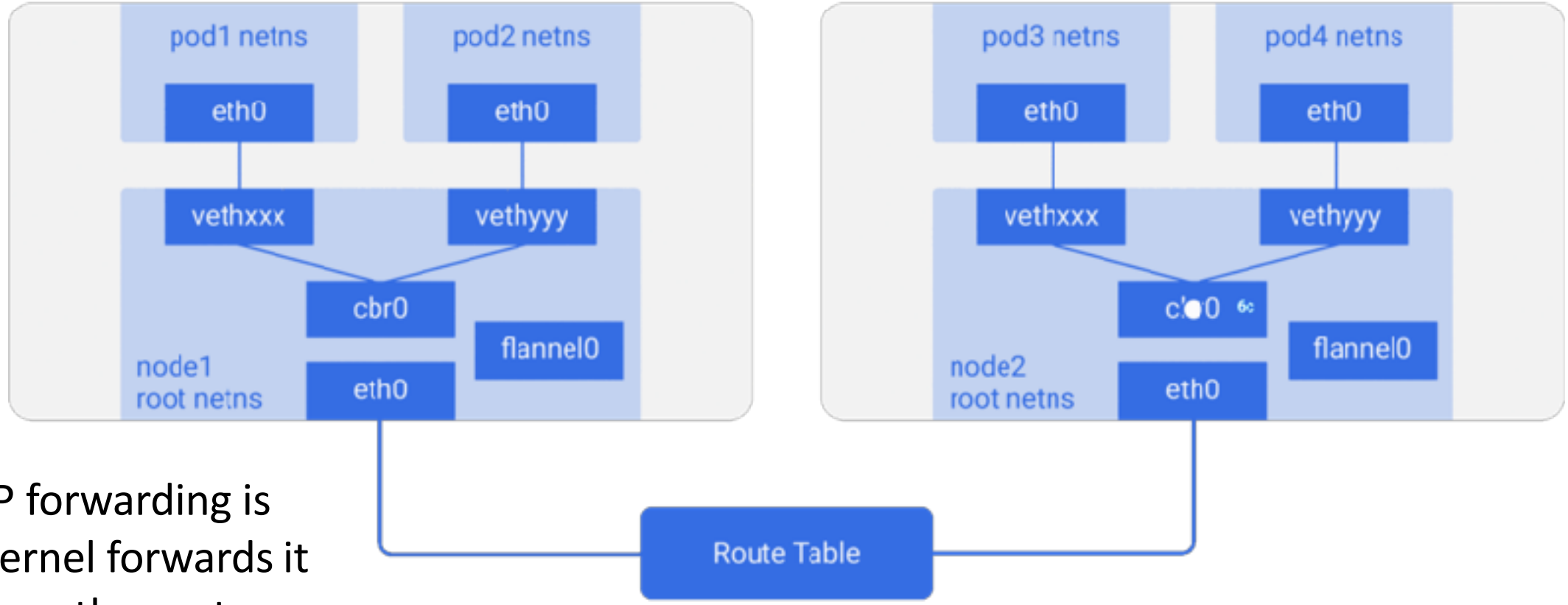
6a. The packet arrives at eth0 of node2. Due to the port being special vxlan port, kernel sends the packet to flannel0.

Flannel Overlay Networking



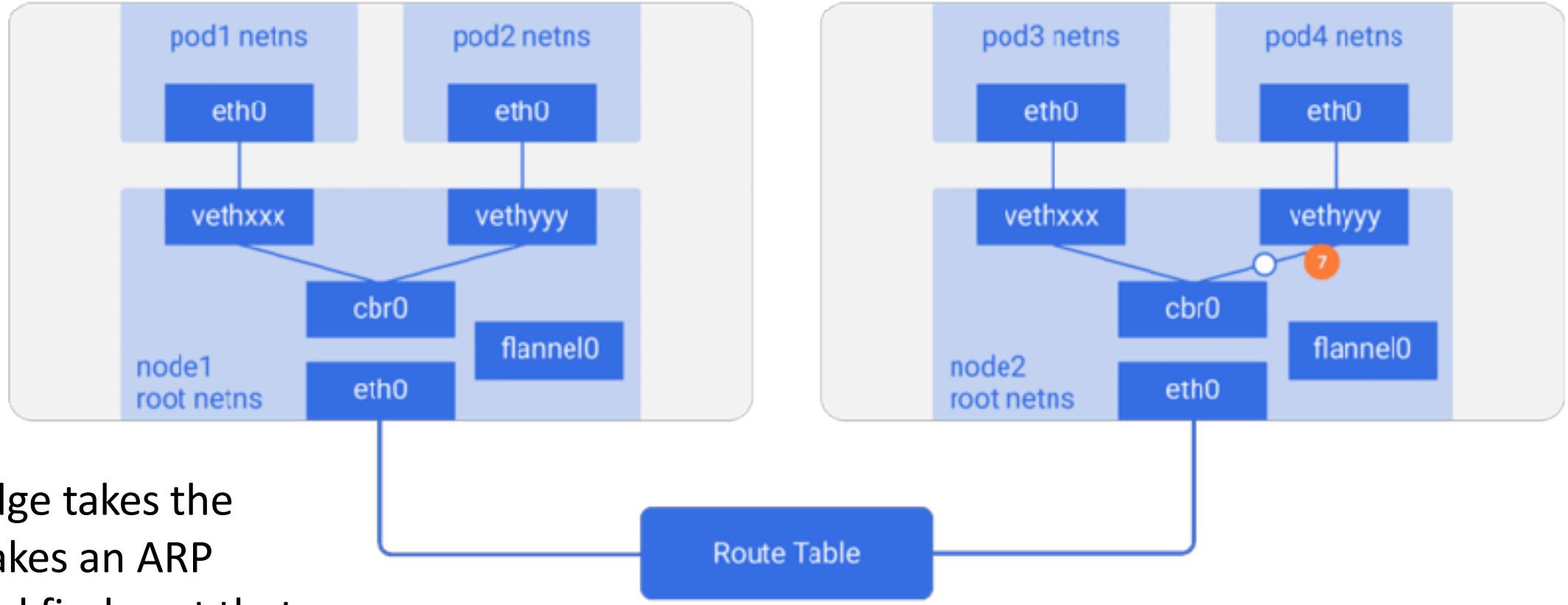
6b. flannel0 de-capsulates and emits it back in the root network namespace.

Flannel Overlay Networking



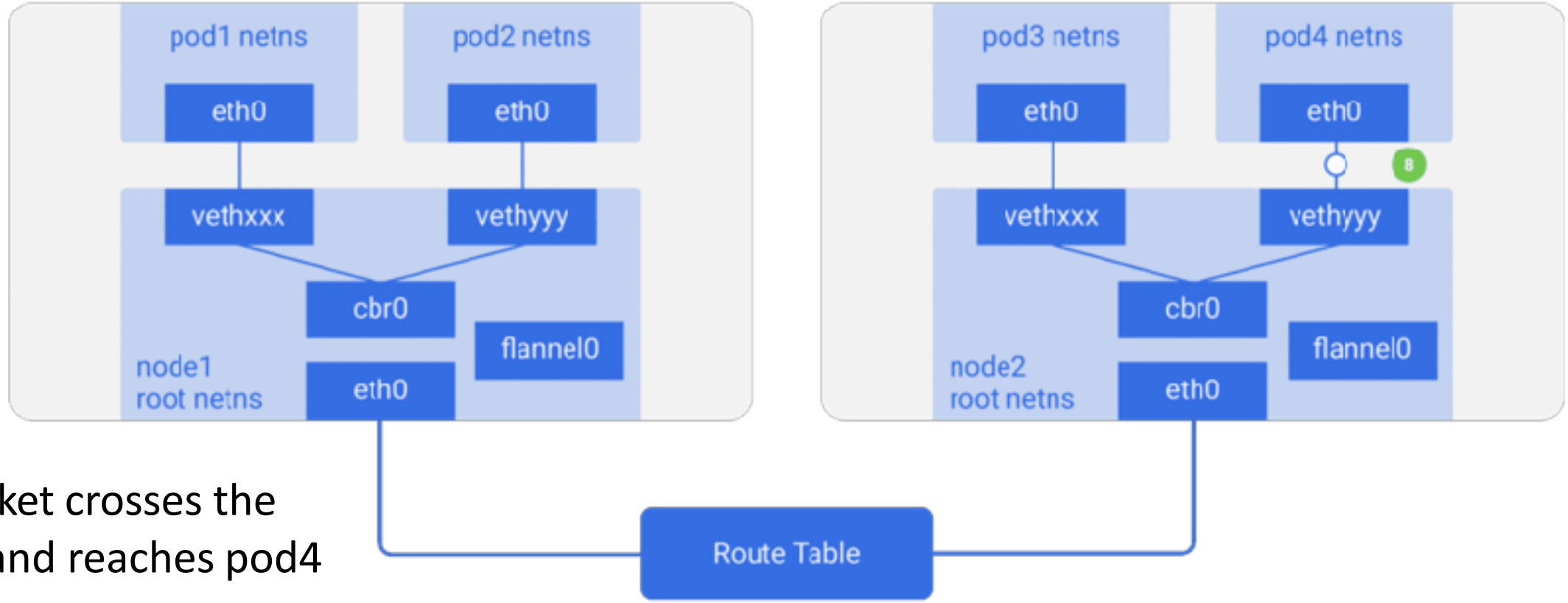
6c. Since IP forwarding is enabled, kernel forwards it to cbr0 as per the route tables.

Flannel Overlay Networking



7. The bridge takes the packet, makes an ARP request and finds out that the IP belongs to vethyyy.

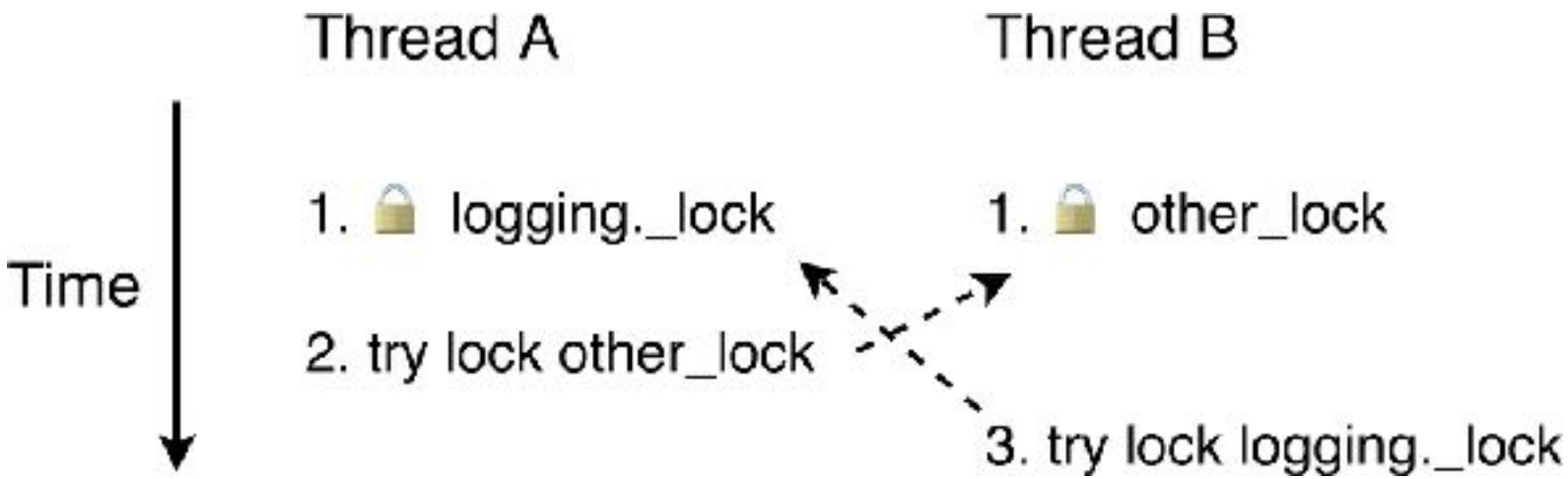
Flannel Overlay Networking



8. The packet crosses the pipe-pair and reaches pod4

Liveness & Readiness Probes

Health Checks: deadlock



Health Checks

- Simple process health checks are enabled by default.
- Kubelet checks to see if container is responding and if not recreates it.

Go deadlock example code

```
lockOne := sync.Mutex{}
lockTwo := sync.Mutex{}
go func() {
    lockOne.Lock();
    lockTwo.Lock();
    ...
}()
lockTwo.Lock();
lockOne.Lock();
```

Health Checks

Kubernetes provides 3 types of application health checks

- HTTP: Kubelet calls a web hook and looks for return status between 200 and 399 for success.
- Container Exec: Kubelet will execute a command inside your container, exit status=0 is a success
- TCP Socket: Kubelet will attempt to open a socket to your container. If it establishes a connection, the container is considered healthy

In all cases, if the Kubelet discovers a failure the container is restarted.

Health Checks

Kubernetes refers to it’s application health checks as:

- Liveness Probe: Check to confirm application is running as expected
- Readiness Probe: Check to confirm application has completed startup configuration

Health Checks

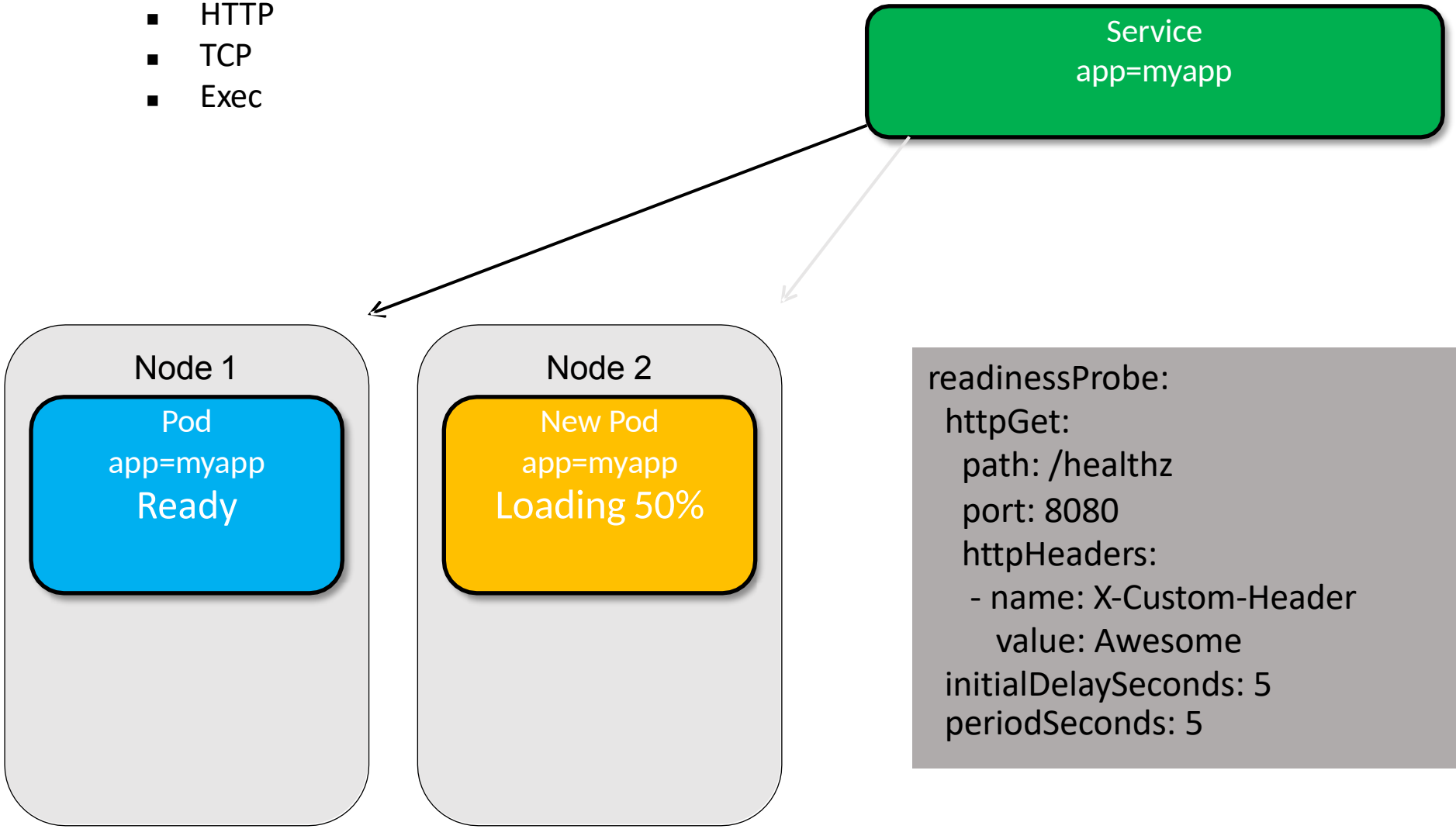
- Execute command to check if application is responding
- Set delay in seconds
  - How long before performing first check
- Set period in seconds.
  - How often it should run

livenessProbe example

```
livenessProbe:
  exec:
  command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

Health Checks: livenessProbe

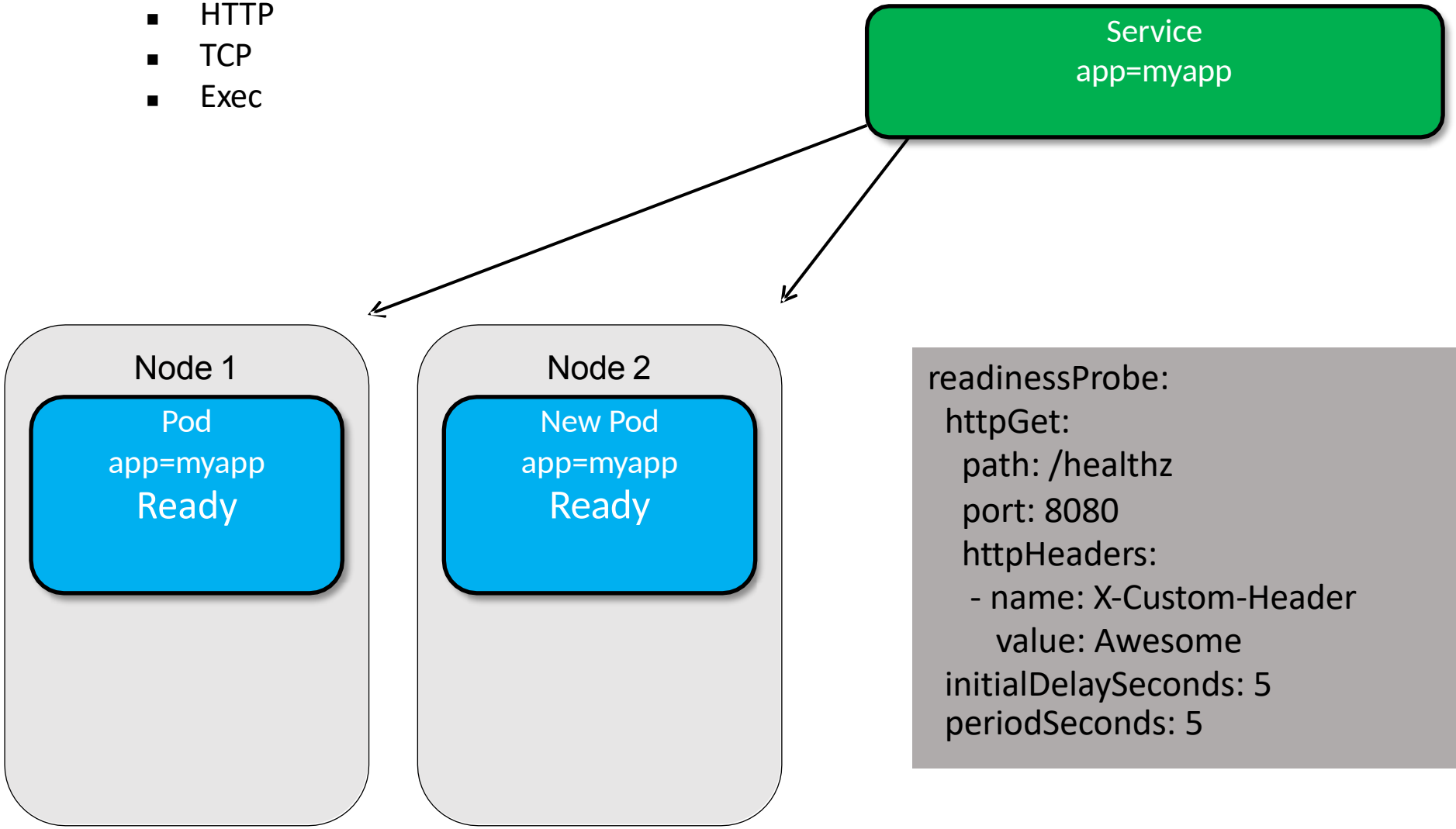
- Do not send traffic to Pod until passes.
  - HTTP
  - TCP
  - Exec




```
readinessProbe:
  httpGet:
    path: /healthz
    port: 8080
    httpHeaders:
     - name: X-Custom-Header
       value: Awesome
  initialDelaySeconds: 5
  periodSeconds: 5
```

Health Checks: livenessProbe

- Do not send traffic to Pod until passes.
  - HTTP
  - TCP
  - Exec




```
readinessProbe:
  httpGet:
    path: /healthz
    port: 8080
    httpHeaders:
     - name: X-Custom-Header
       value: Awesome
  initialDelaySeconds: 5
  periodSeconds: 5
```

Health Checks

- Same as livenessProbe but traffic won’t be sent to it if probe fails.

readinessProbe example

```
readinessProbe:
  httpGet:
    path: /healthz
    port: 8080
    httpHeaders:
     - name: X-Custom-Header
       value: Awesome
  initialDelaySeconds: 5
  periodSeconds: 5
```

Health Check Lab

- Create POD with liveness check
- Verify check is working as expected
- Create POD with HTTP liveness check
- Confirm HTTP liveness probes are working correctly
- Create deployment with liveness and readiness probe
- Confirm checks are working

Role Based Access Controls

RBAC

- Considered stable in Kubernetes 1.8 and later
- To enable start apiserver with
  - --authorization-mode=RBAC
  - This allows for more fine-grained control of access permissions

RBAC Roles

- Role used to grant access to resources in a single namespace

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
    resources: ["pods"]
    verbs: ["get", "watch", "list"]
```

RBAC ClusterRole

- ClusterRole used to grant access to resources that are cluster specific, not namespace specific

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
    resources: ["secrets"]
    verbs: ["get", "watch", "list"]
```

RBAC RoleBinding

- Grant permissions defined in a role to user or set of users

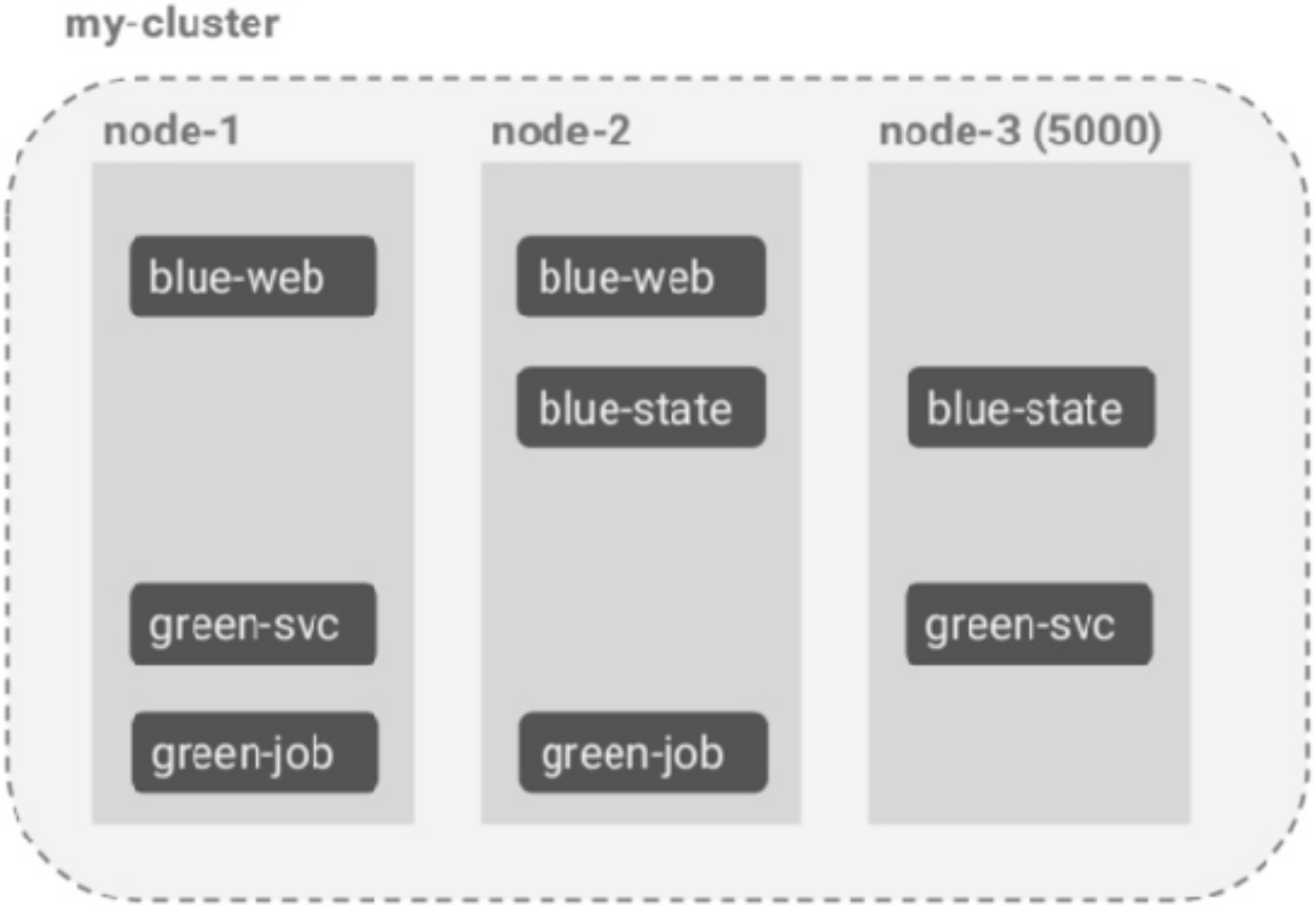
```
# This role binding allows "jane" to read pods in the "default" namespace.
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
   kind: Role
   name: pod-reader
   apiGroup: rbac.authorization.k8s.io
```

Without fine-grained Access:

- Authorization at cluster level
- All pods have same authorization

Without controlled scheduling:

- Lack flexibility for multi-workload

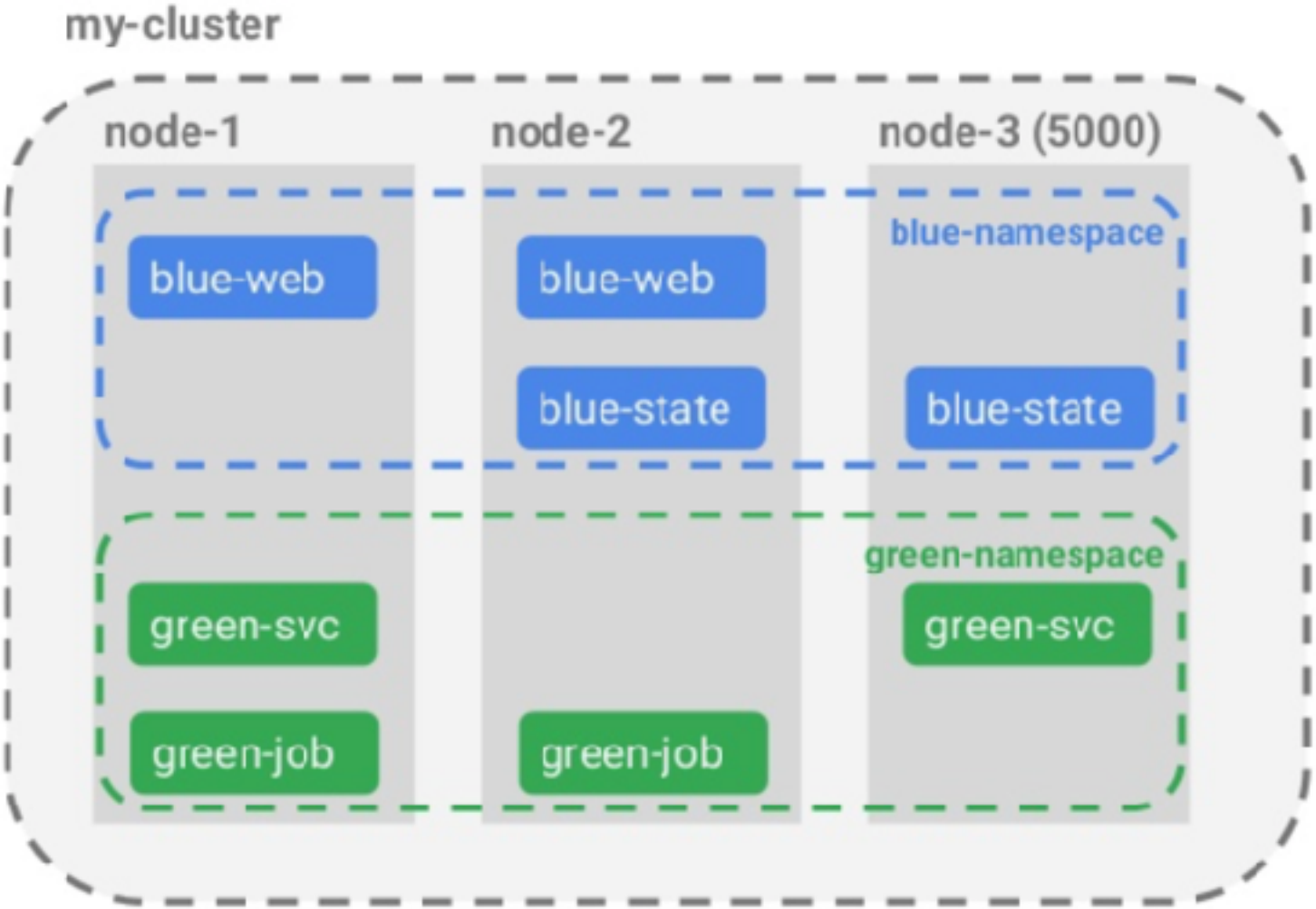


Introducing RBAC:

- Per-namespace/ resource, role, action

Examples:

- **Alice** can list **Eng** services, but not **HR**
- **Bob** can create Pods in **Test** namespace, but not in **Prod**
- **Scheduler** can read **Pods** but not **Secrets**



RBAC lab

- Create user
- Generate SSL credentials for new user
- Create RBAC Role and RoleBinding
- Assign correct permissions to user
- Confirm RBAC is working as expected

Kubernetes Services

Services

- Persistent way to access logical set of PODs
- Set of PODs (usually) determined by a Label Selector
- Kubernetes-native apps: Kubernetes offers a simple Endpoints API that is updated whenever the set of Pods in a Service changes.
- Non-native apps: Kubernetes offers a virtual-IP-based bridge to Services which redirects to the backend Pods.

Service Definition

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

Types of Services

- ClusterIP
- NodePort
- LoadBalancer

- Ingress is not a service

ClusterIP

- Virtual IP that every node can route to

NodePort

- Same port open on every node (default range: 30000-32767)

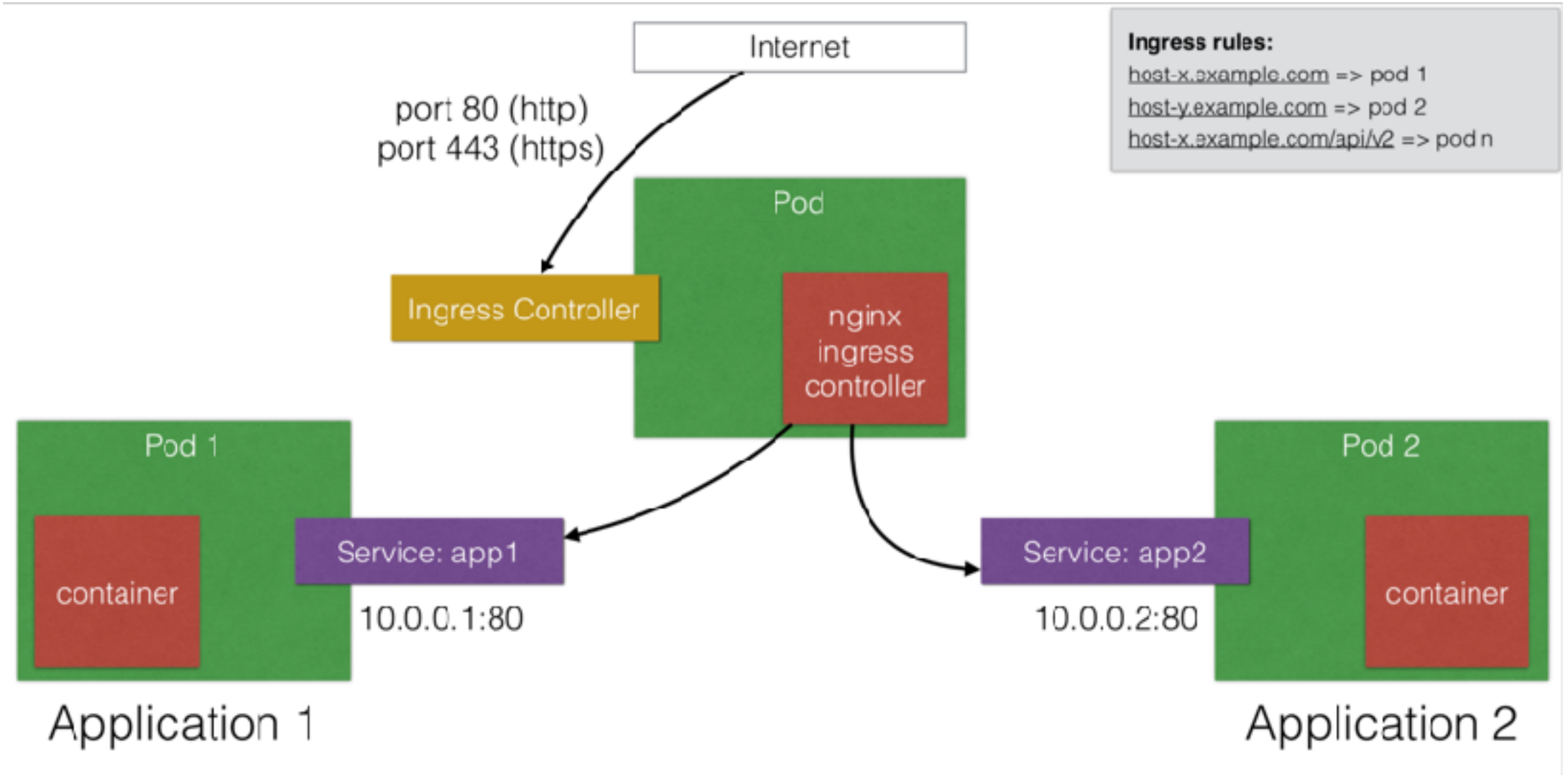
LoadBalancer

- External L4 network load balancer
- Requires cloud / IaaS coordination

Ingress

- L7 (instead of L4)
- HTTP (instead of tcp/udp)
- Ingress can route paths differently:
  - /api/app1 -> app1 pods
  - /api/app2 -> app2 pods
- Requires ingress provider (might be cloud, might be in-cluster)

Ingress

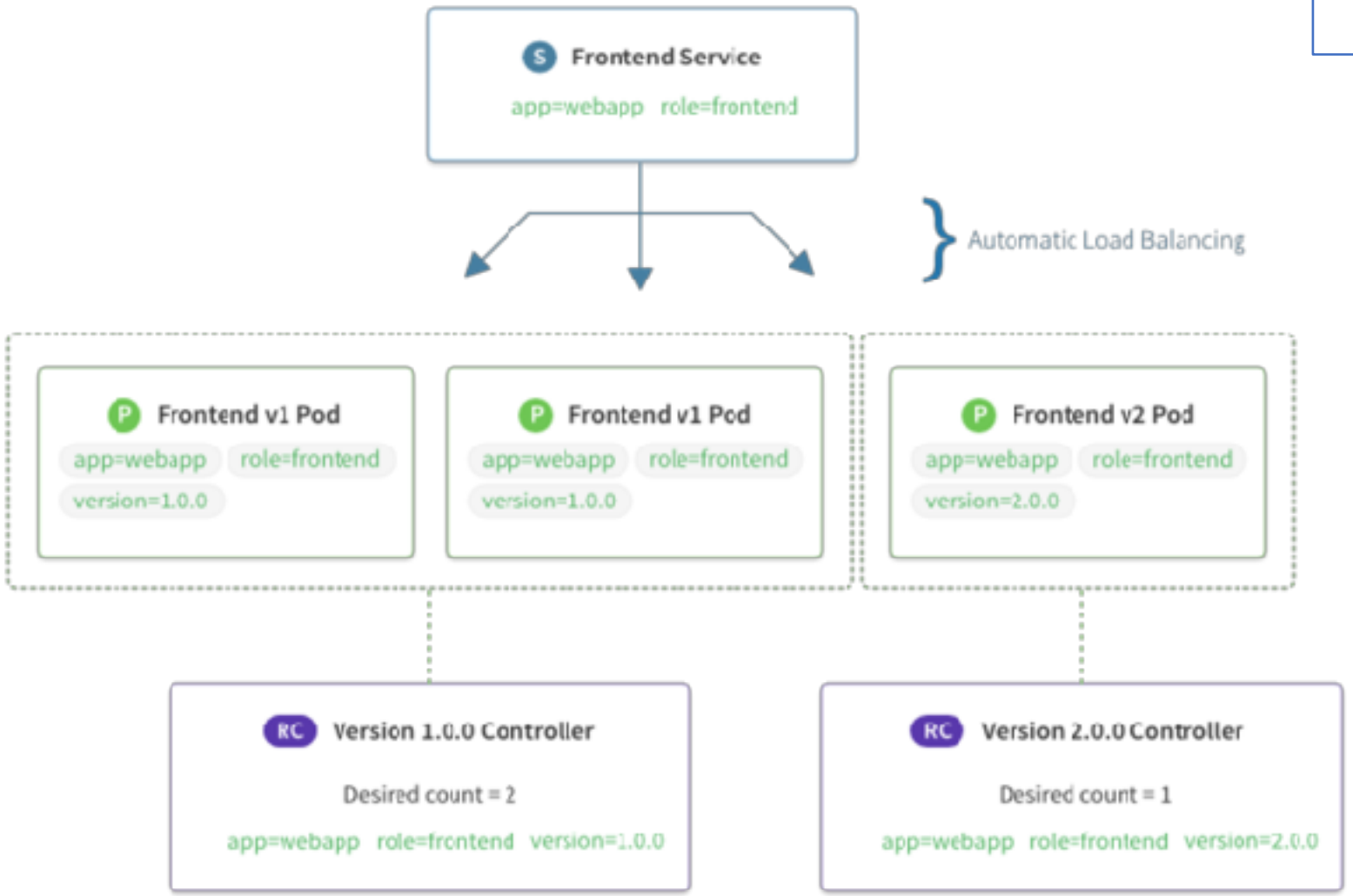

Internet
port 80 (http)
port 443 (https)
Ingress rules:
host-x.example.com => pod 1
host-y.example.com => pod 2
host-x.example.com/api/v2 => pod n
Pod
Ingress Controller
nginx ingress controller
Pod 1
container
Service: app1
10.0.0.1:80
Service: app2
10.0.0.2:80
Pod 2
container
Application 1
Application 2

Ingress Definition

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: helloworld-rules
spec:
  rules:
  - host: helloworld-v1.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: helloworld-v1
          servicePort: 80
  - host: helloworld-v2.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: helloworld-v2
          servicePort: 80
```

Services

my-service
app=MyApp
Port=80
targetPort=9276



Services

- Services can map incoming port to any targetPort
- By default the targetPort will be set to the same value as the port field.
- NOTE: targetPort can be a string, referring to the name of a port in the backend Pods.
- Services support TCP and UDP for protocols. The default is TCP.

Services mapping string

Deployment Specification

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 8080
          name: http-web
          protocol: TCP
```

Services targetPort mapped to string

Service specification

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

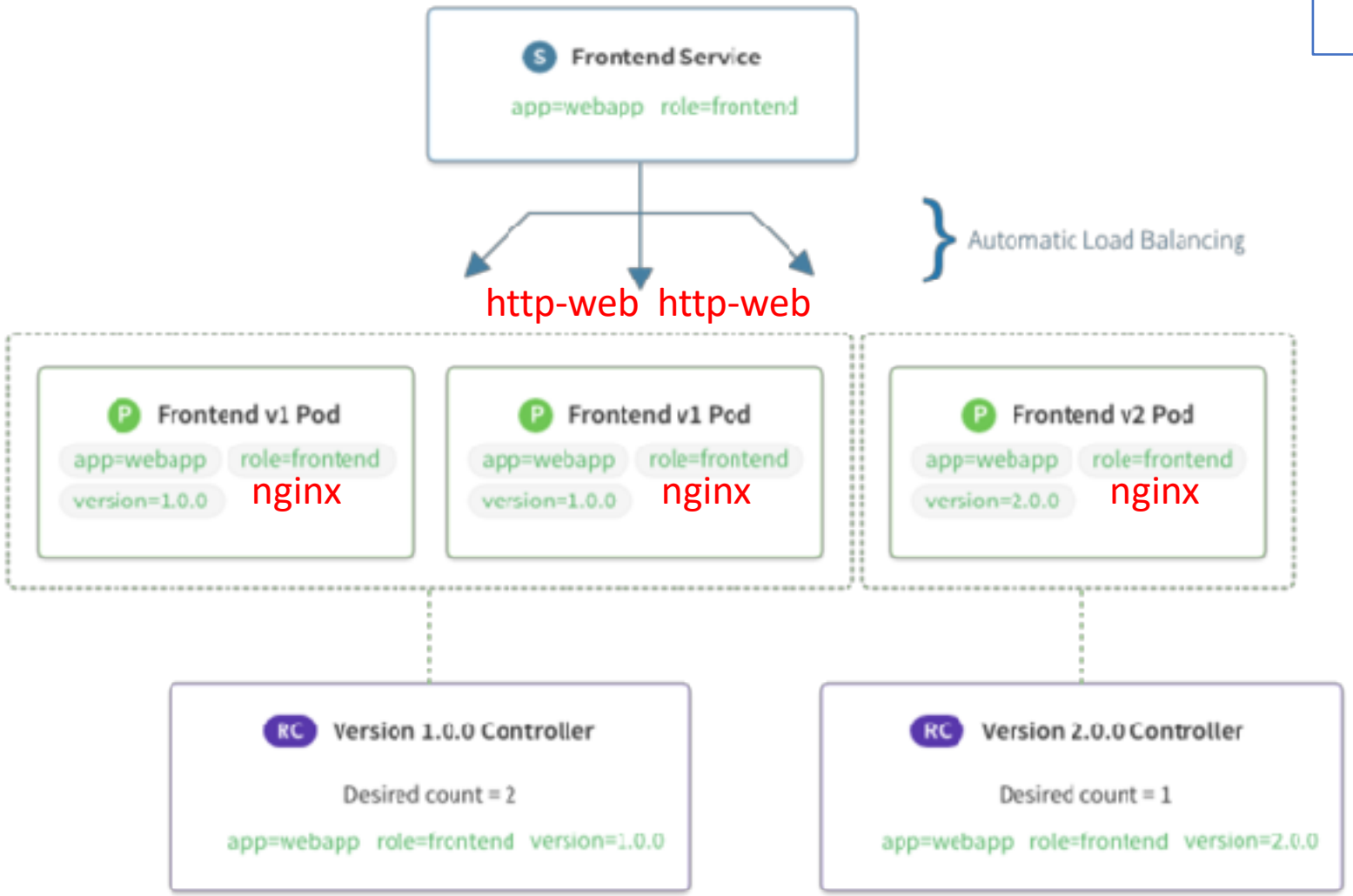
Services diagram mapping string

my-service

app=nginx

Port=80

targetPort=http-web



Services without selectors

- Why would you want a Service that’s not mapped to a selector?
- Use an external database cluster in production, but in test you use your own databases.
- Point your service to a service in another namespace or on another cluster.
- Migrating your workload to Kubernetes and some of your backends run outside of Kubernetes
- More flexibility

Kubernetes DNS

Kubernetes DNS

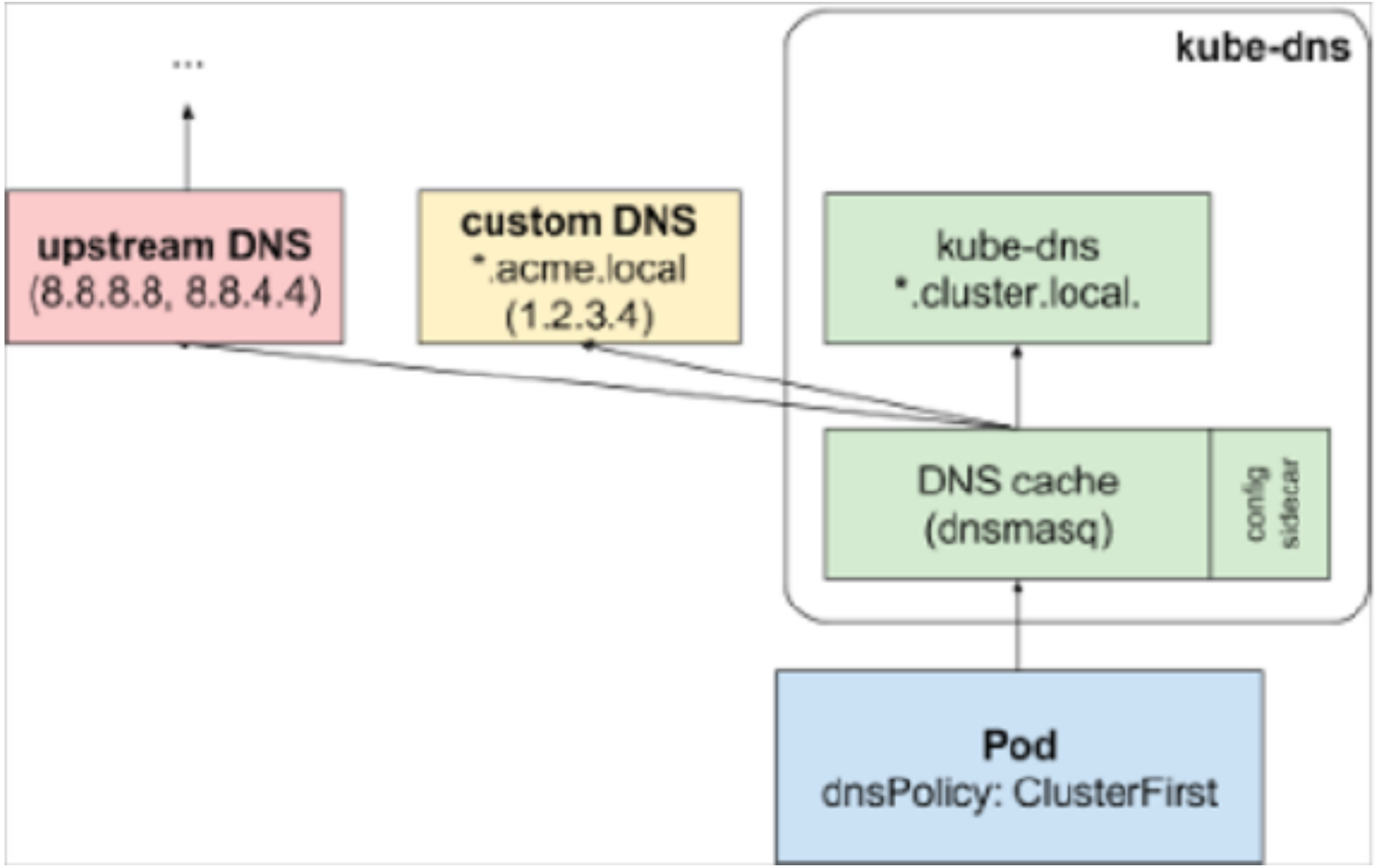
- Kubernetes Schedules a DNS POD, which sits in “Pending” until a network is created.
- Objects that have DNS names:
  - Everything
- DNS enables PODs in different namespaces to communicate easily

Kubernetes DNS

```
apiVersion: v1
kind: Service
metadata:
    name: default-subdomain
spec:
   selector:
      app: MyApp
ports:
   - name: http
     protocol: TCP
     port: 80
     targetPort: 9376
   - name: https
     protocol: TCP
     port: 443
     targetPort: 9377
```

```
apiVersion: v1
kind: Pod
metadata:
     name: busybox1
     labels:
        name: busybox
spec:
   hostname: busybox-1  subdomain:
   default-subdomain  containers:
   - image: busybox
     command:
       - sleep
       - “3600”
     name: busybox
```

Kubernetes DNS



Kubernetes DNS

- Use a ConfigMap to specify:
  - custom stub domains
  - upstream nameservers

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: kube-dns
    namespace: kube-system
data:
  stubDomains: |
    {"acme.local": ["1.2.3.4"]}
  upstreamNameservers: |
    ["8.8.8.8", "8.8.4.4"]
```

Debugging DNS

- Simple POD to use for testing
  - YAML file to deploy busybox

```
apiVersion: v1
kind: Pod
metadata:
    name: busybox
    namespace: default
spec:
  containers:
  - name: busybox
    image: busybox
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

Debugging DNS

- Simple POD to use for testing
  - YAML file to deploy busybox

```
kubectl exec -ti busybox – nslookup kubernetes.default

Server:    10.96.0.10

Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local
Name:      kubernetes.default
Address 1: 10.96.0.1 kubernetes.default.svc.cluster.local
```

Debugging DNS

- If nslookup fails check local DNS config first!
- Check the resolv.conf

```
kubectl exec busybox cat /etc/resolv.conf

nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local ec2.internaloptions
ndots:5
```

Debugging DNS

- Problems with kube-dns add-on or associated Services:

```
kubectl exec –ti busybox – nslookup kubernetes.default

Server:         10.0.0.10
Address 1:    10.0.0.10
nslookup: can’t resolve ‘kubernetes.default’
```

Debugging DNS

- Check to make sure DNS pod is running
- Check for errors in kube-dns PODs logs

```
kubectl get pods -n kube-system –l k8s-app=kube-dns

NAME                          READY     STATUS              RESTARTS          AGE
kube-dns-6f4fd4bdf-pr659      3/3       Running             0                 6d
```

Service Discovery: Env Var

Service Discovery also possible via environment variable:

```
> kubectl run -it --restart=Never --rm busybox --image=busybox env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=busybox
TERM=xterm
SIMPLE_API_PORT_80_TCP_PROTO=tcp
NGINX_DEPLOYMENT_SERVICE_HOST=10.55.246.243
NGINX_DEPLOYMENT_PORT_80_TCP_PORT=80
NGINX_SERVICE_PORT=80
SIMPLE_API_PORT=tcp://10.55.241.109:80
KUBERNETES_PORT_443_TCP=tcp://10.55.240.1:443
NGINX_SERVICE_HOST=10.55.255.228
KUBERNETES_SERVICE_HOST=10.55.240.1
SIMPLE_API_PORT_80_TCP_PORT=80
NGINX_PORT=tcp://10.55.255.228:80
SIMPLE_API_SERVICE_PORT=80
NGINX_DEPLOYMENT_SERVICE_PORT=80
NGINX_DEPLOYMENT_PORT_80_TCP_PROTO=tcp
KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
NGINX_PORT_80_TCP=tcp://10.55.255.228:80
SIMPLE_API_PORT_80_TCP=tcp://10.55.241.109:80
KUBERNETES_PORT_443_TCP_PORT=443
SIMPLE_API_SERVICE_HOST=10.55.241.109
```

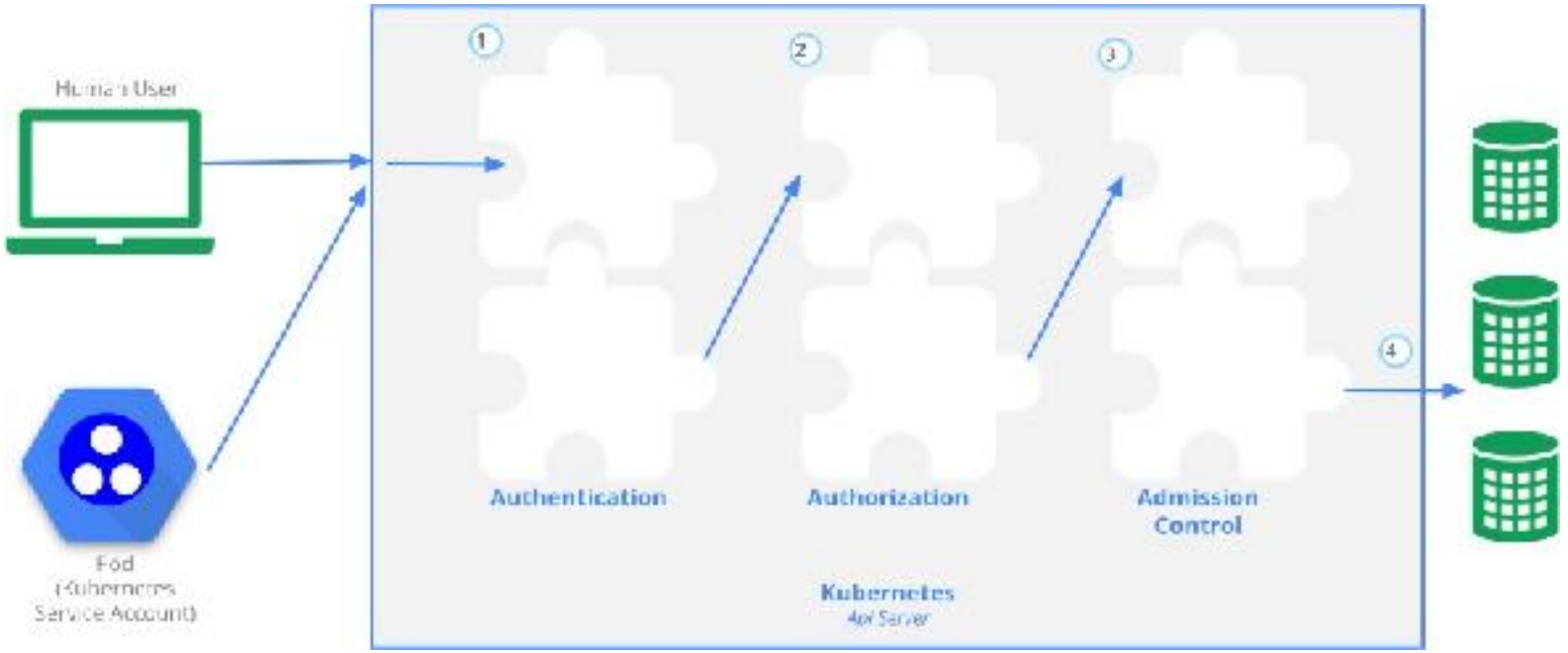
DNS and Services Lab

- Create database deployment
- Expose database using a Service
- Create Wordpress deployment
- Expose Wordpress using a Service

NOTE: Investigate YAML files to see how DNS is being used between services.

Kubernetes API & Security

API Authorization Flow



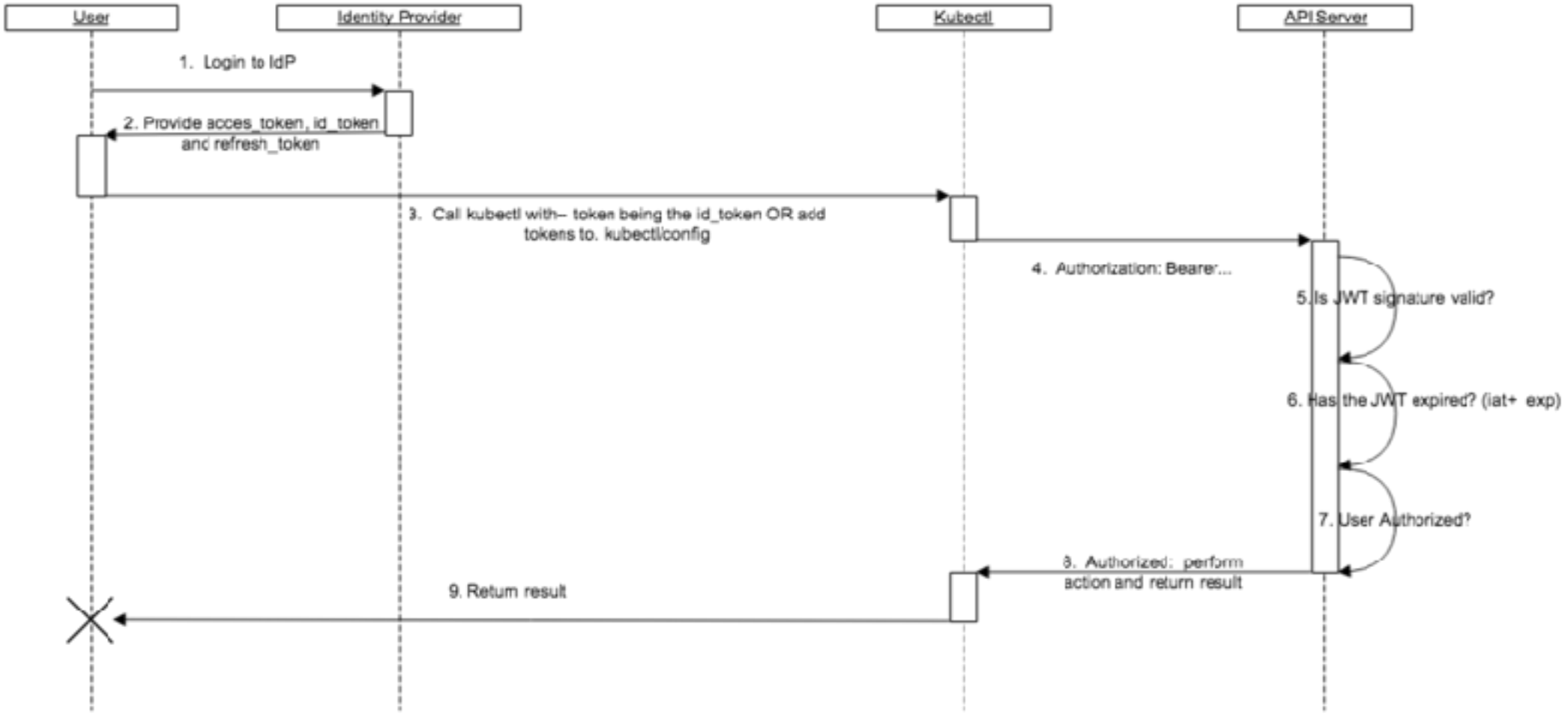
API Authentication

- In a typical Kubernetes cluster, the API serves on port 443
- Typically **$USER/.kube/config** contains the root certificate
- If request cannot be authenticated it's rejected with HTTP status code 401
- Kubernetes can use "usernames" for access control decisions and in request logging, it does not have a user object nor does it store usernames or other information about users in its object store.

API Authentication

- types: user, service account
- users are managed by independent service, not kubernetes cluster
- service accounts are managed by kubernetes api
- Authentication modules include Client Certificates, Password, and Plain Tokens, Bootstrap Tokens, and JWT Tokens (used for service accounts).

API Auth with Identity Provider



API Authorization

Request must include:

- username
- action
- object affected by action

API Authorization

- Example of policy for Bob

```
{
    "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
    "kind": "Policy",
    "spec": {
            "user": "bob",
            "namespace": "projectCaribou",
            "resource": "pods",
           "readonly": true
    }
}
```

API

Authorization

This request will be authorized for Bob

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "projectCaribou",
      "verb": "get",
      "group": "unicorn.example.org",
      "resource": "pods"
    }
  }
}
```

API Access (kubectl proxy)

- kubectl proxy creates a tunnel from local machine to API endpoint

Admission Controllers

- Additional custom security modules which may reject the request
- Must be enabled on the api server when kube-apiserver is started
- Examples:
  - AlwaysPullImages
    - Security enhancement (checks image access every time new Pod created)
  - NameSpaceAutoProvision
  - Priority
    - Checks priorityClassName field, fails if not listed
  - many more!

Additional info: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

Monitoring

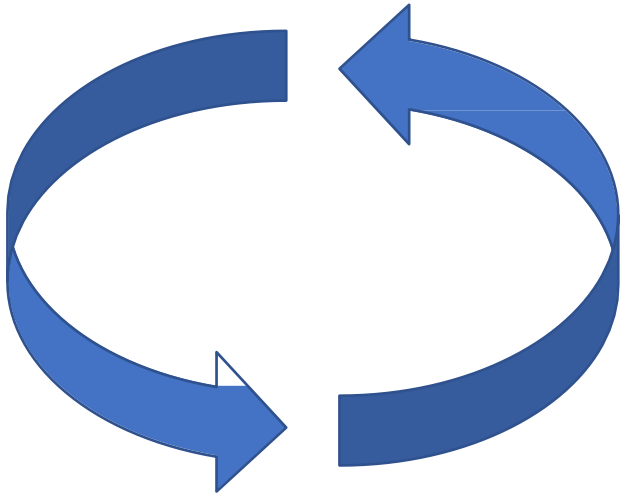
Monitoring

Why monitor?


HYUNDAI
PANAMA

Monitoring

Quality Assurance



Continuous Improvement

Monitoring

Not about collecting data

- Why vs How

Monitoring

Why monitor?

- Know when things go wrong
- Debug and learn from issues
- Historical view/Trends
- Drive technical/business decisions
- Feed into other systems/processes (QA, Security, automation)

Monitoring

Challenges

- Monitoring tools are limited
  - Technically
  - Conceptually
- Tools don’t scale well and are difficult to manage
- Operational practices don’t align with business needs

Monitoring

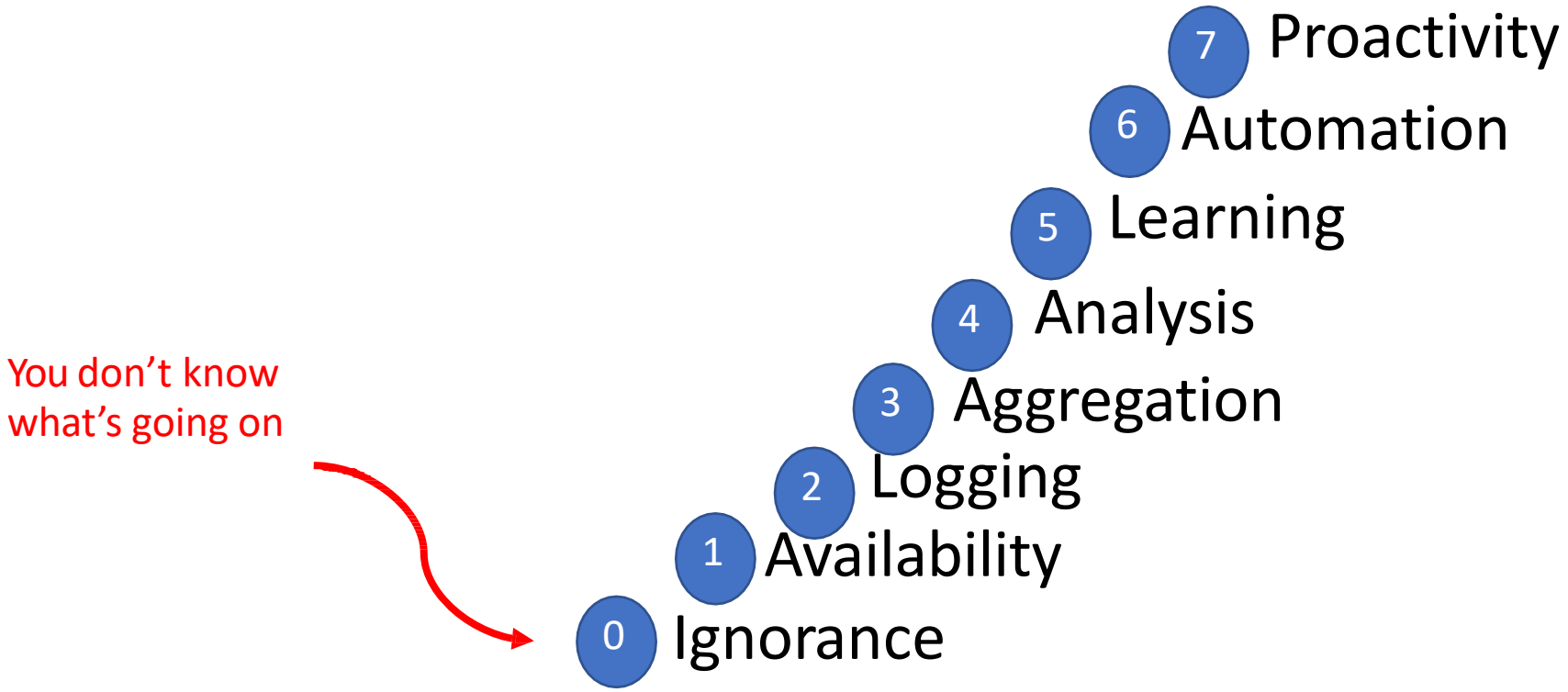
Challenges

Example:

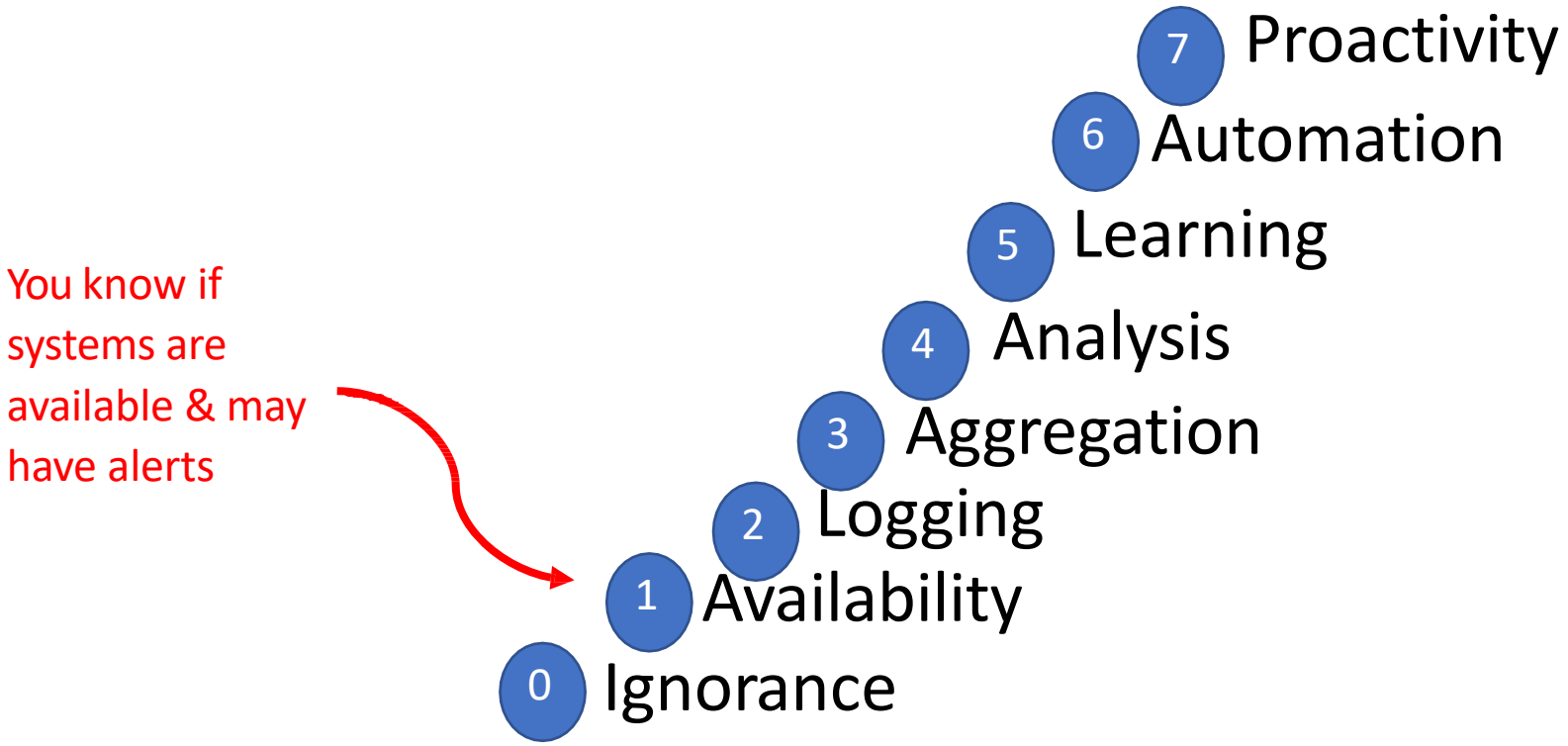
Customers care about increased latency and it’s in your SLAs. Monitoring only supports alerting on individual machines performance and not the entire clusters.

Result: Engineers get a bunch of non critical/false-positive alerts and are woken up for non-issues. This leads to fatigue and ignoring alerts.

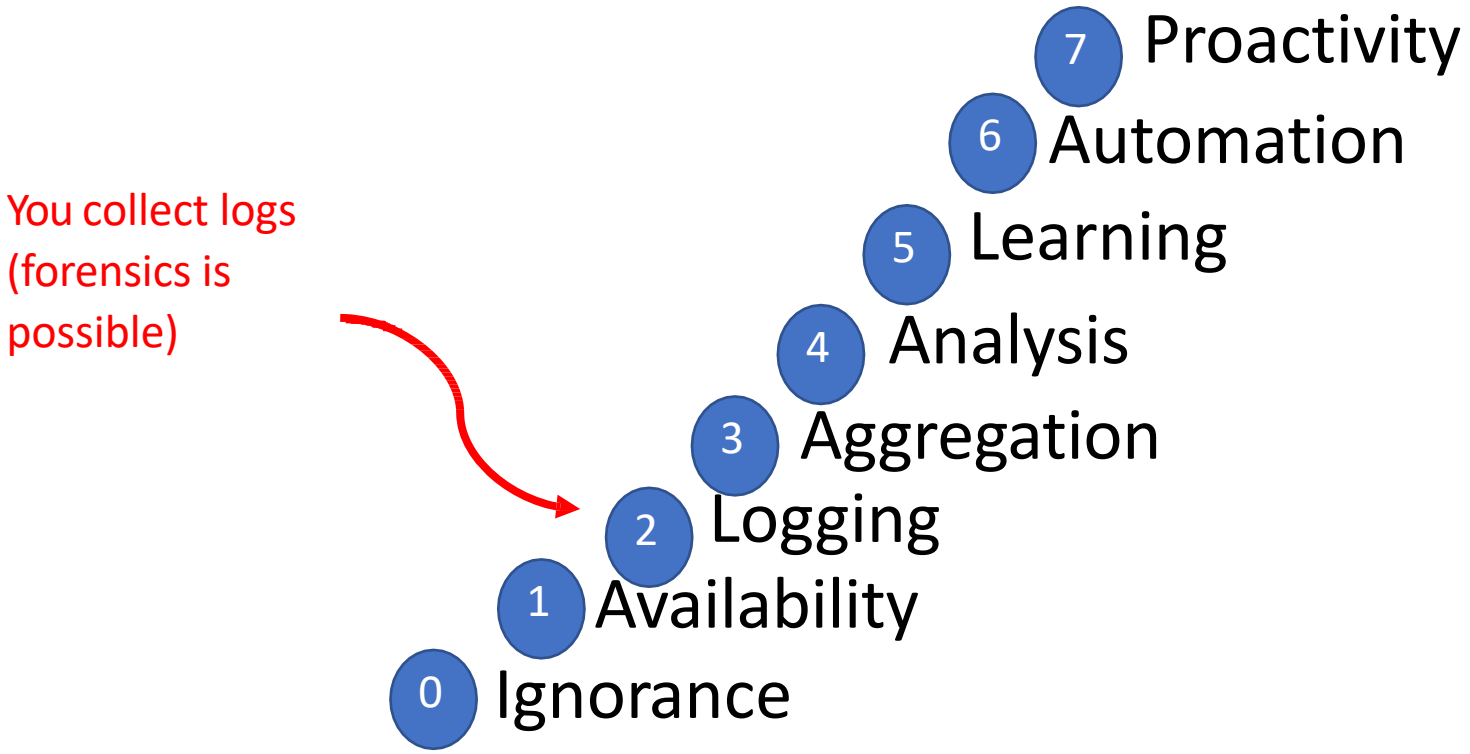
Monitoring


You don’t know what’s going on
0 Ignorance
1 Availability
2 Logging
3 Aggregation
4 Analysis
5 Learning
6 Automation
7 Proactivity

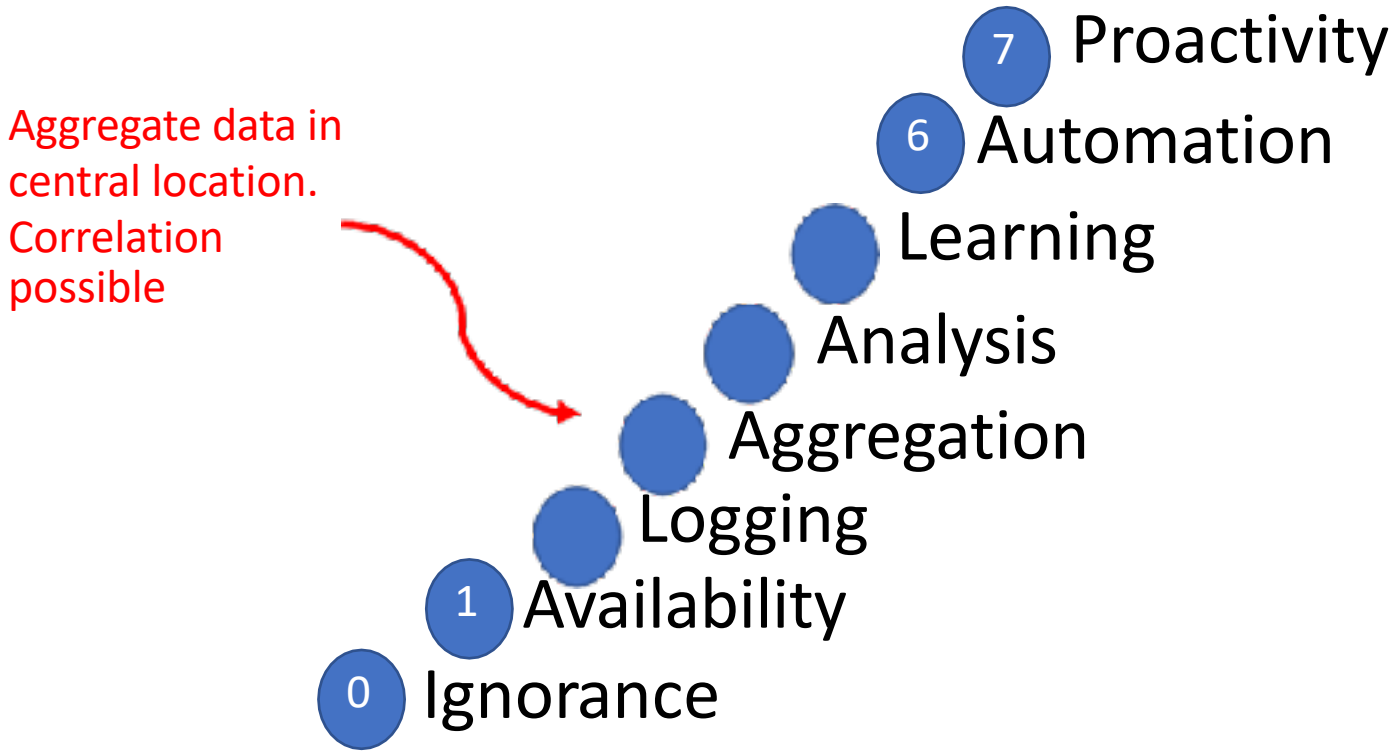
Monitoring


You know if systems are available & may have alerts
0 Ignorance
1 Availability
2 Logging
3 Aggregation
4 Analysis
5 Learning
6 Automation
7 Proactivity

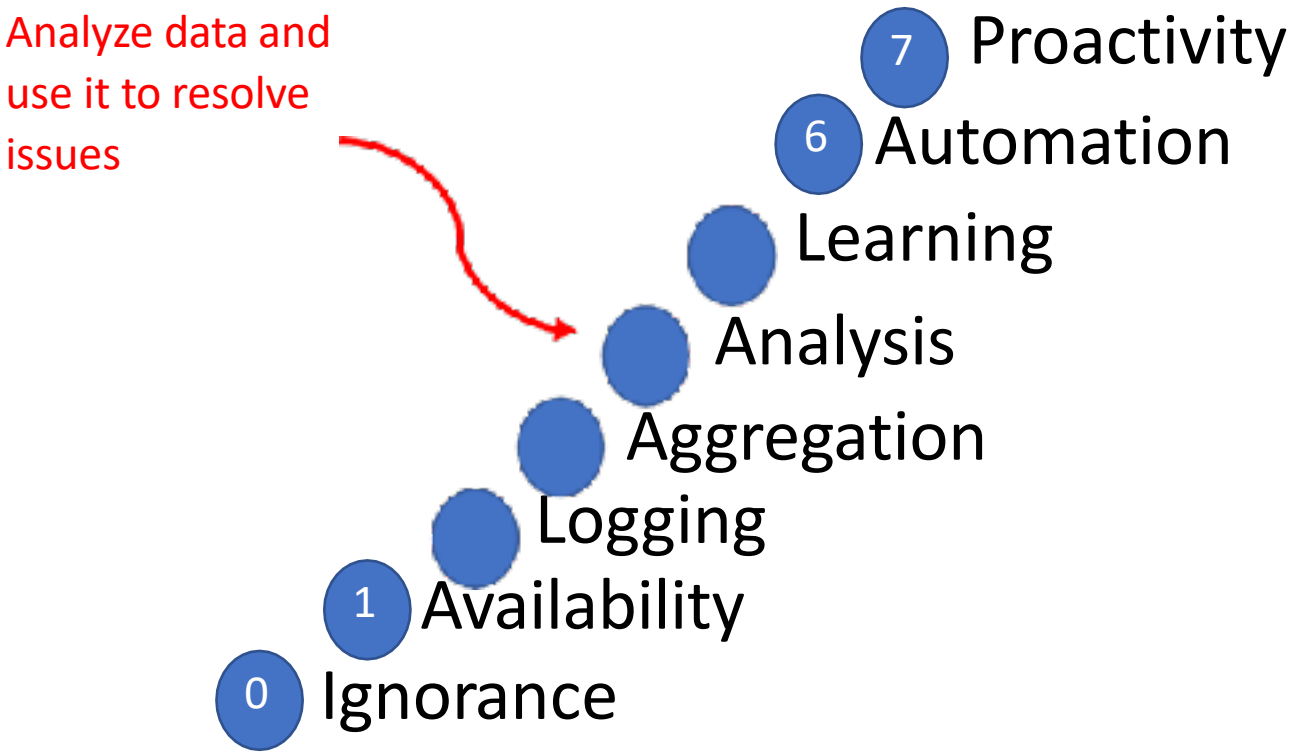
Monitoring


You collect logs (forensics is possible)
7 Proactivity
6 Automation
5 Learning
4 Analysis
3 Aggregation
2 Logging
1 Availability
0 Ignorance

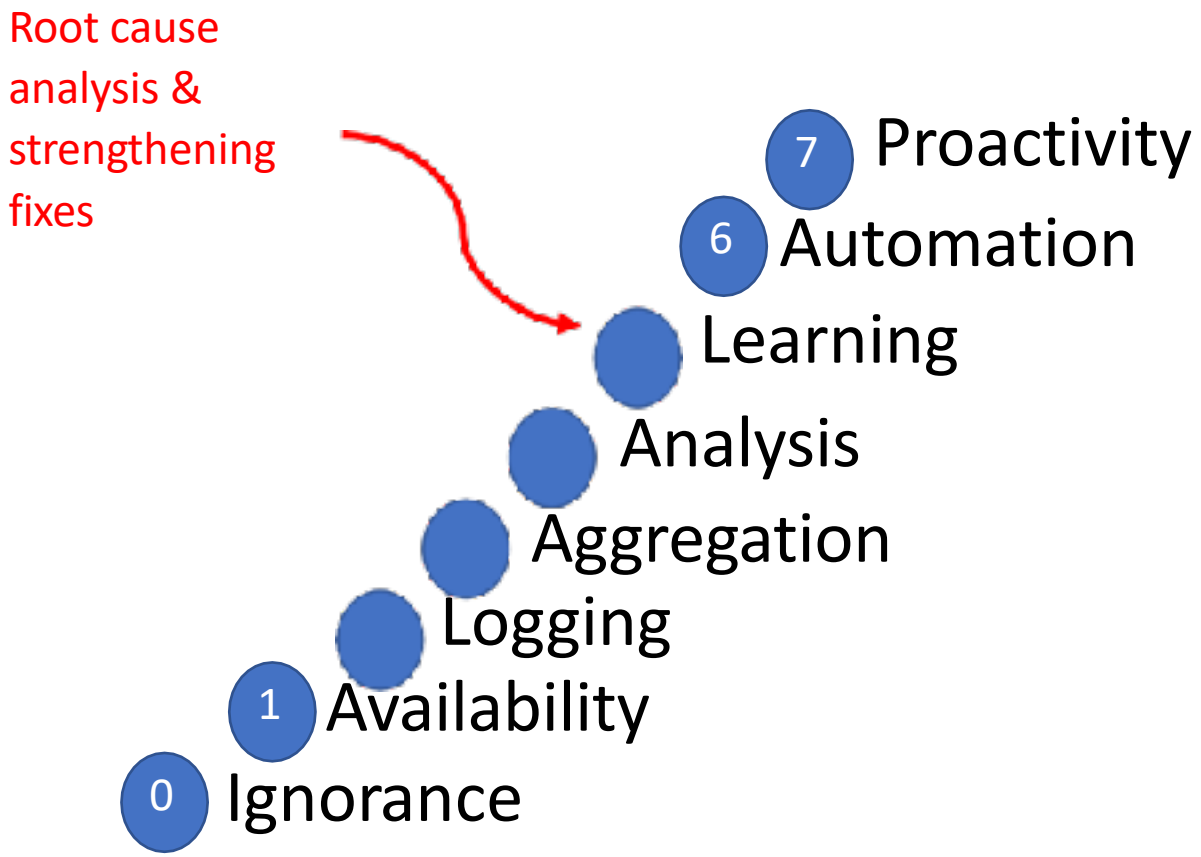
Monitoring


Aggregate data in central location. Correlation possible
7
Proactivity
6
Automation
Learning
Analysis
Aggregation
Logging
1
Availability
0
Ignorance

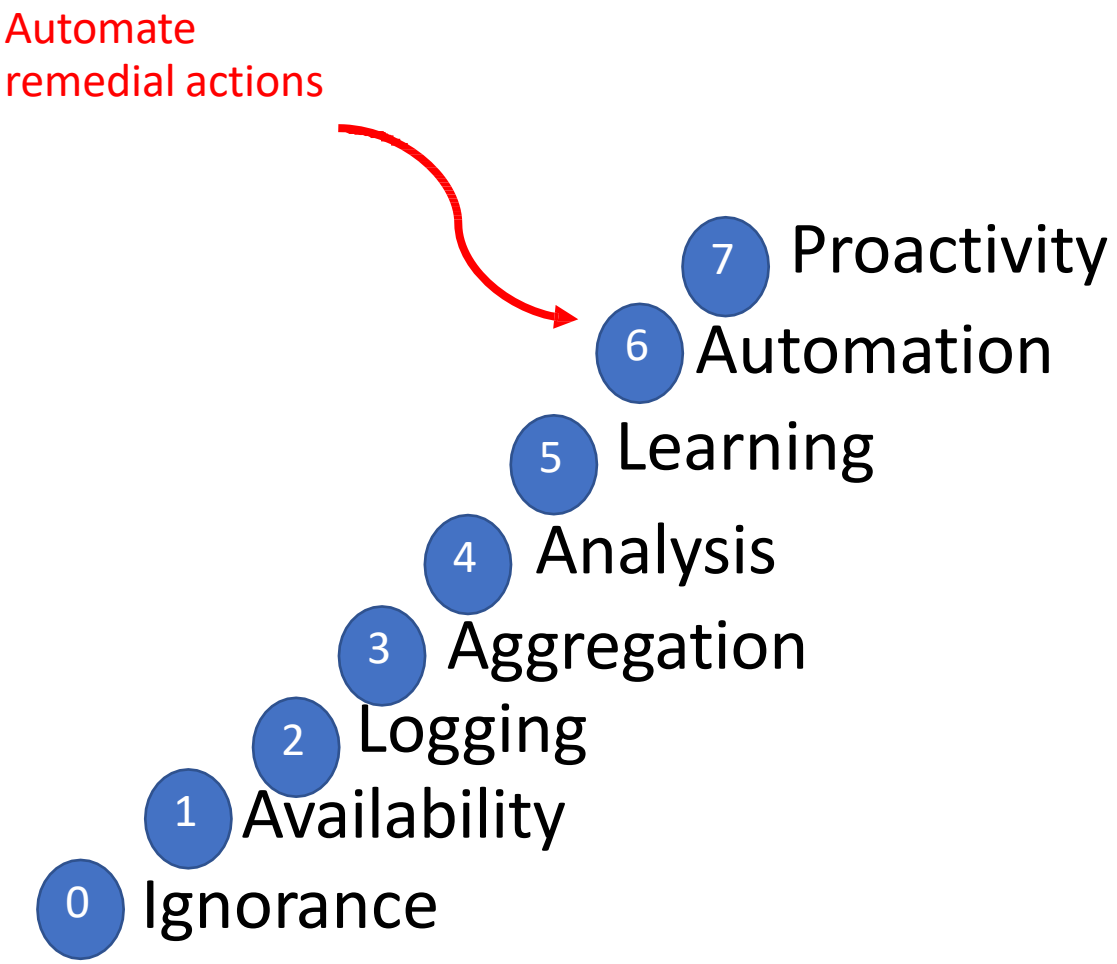
Monitoring


Analyze data and use it to resolve issues
7
Proactivity
6
Automation
Learning
Analysis
Aggregation
Logging
1
Availability
0
Ignorance

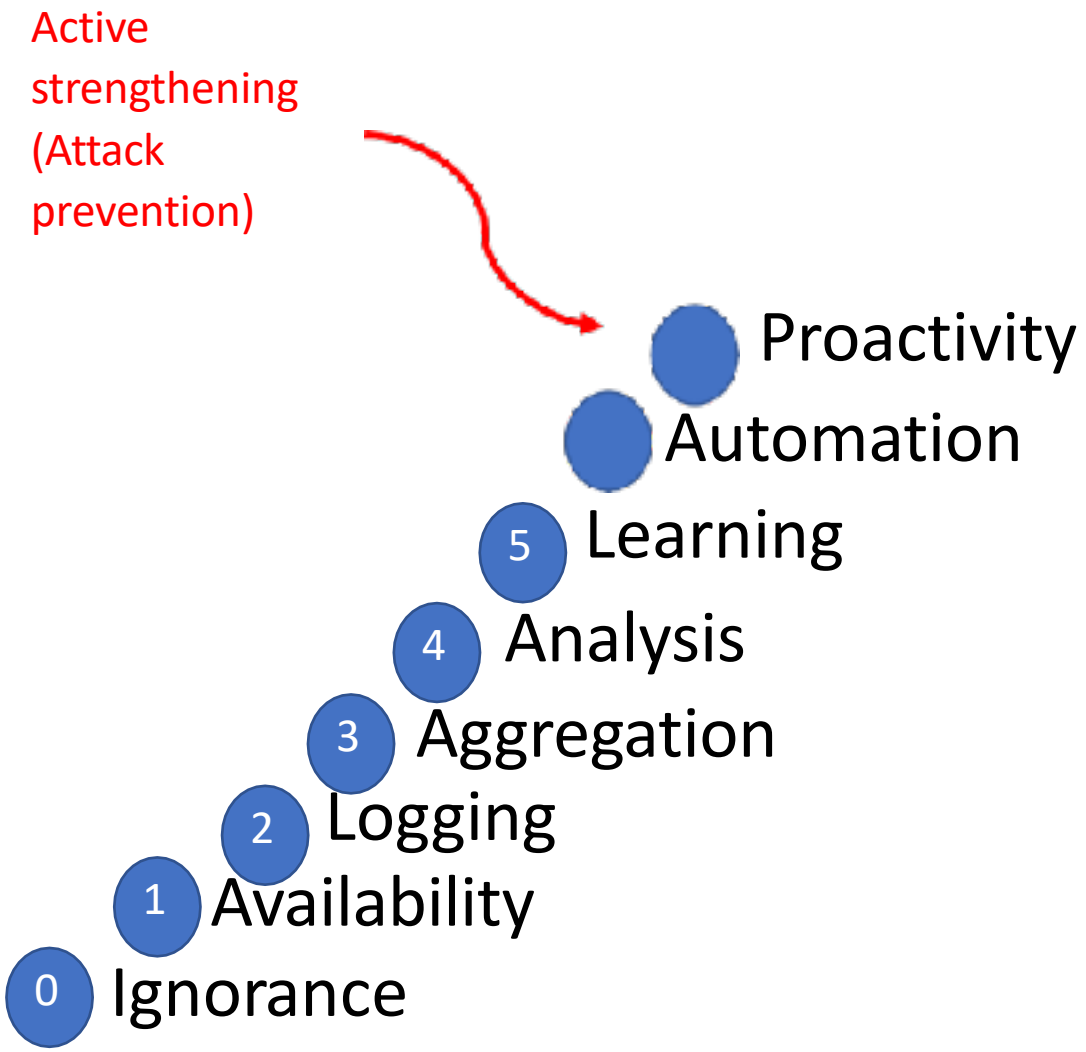
Monitoring


Root cause analysis & strengthening fixes
7
Proactivity
6
Automation
Learning
Analysis
Aggregation
Logging
1
Availability
0
Ignorance

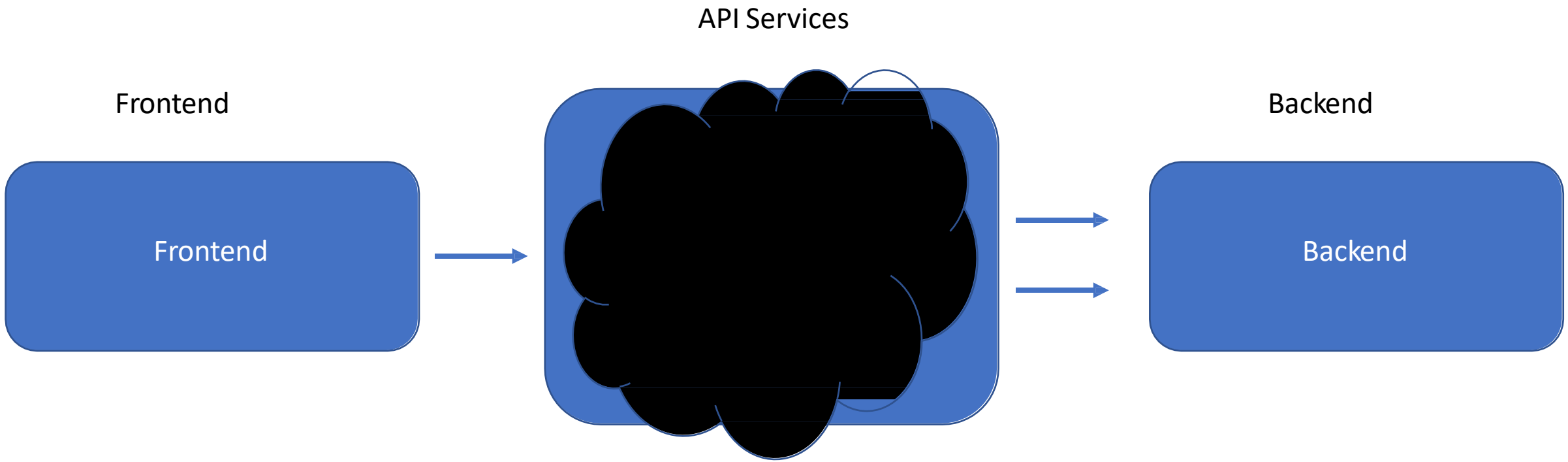
Monitoring


Automate
remedial actions
7
Proactivity
6
Automation
5
Learning
4
Analysis
3
Aggregation
2
Logging
1
Availability
0
Ignorance

Monitoring

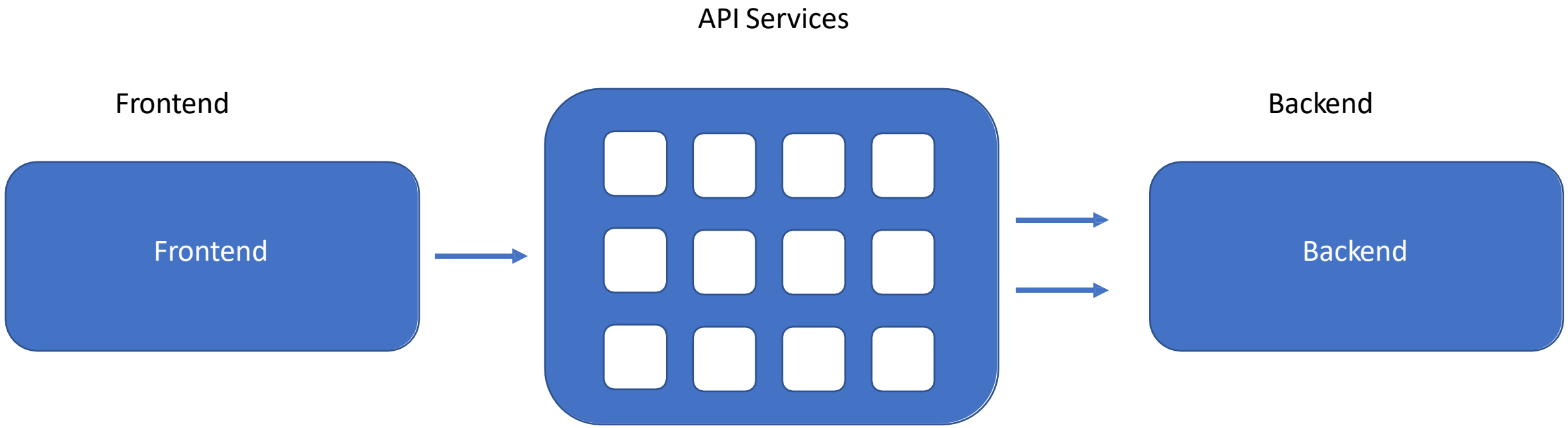

Active strengthening (Attack prevention)
Proactivity
Automation
5
Learning
4
Analysis
3
Aggregation
2
Logging
1
Availability
0
Ignorance

Monitoring



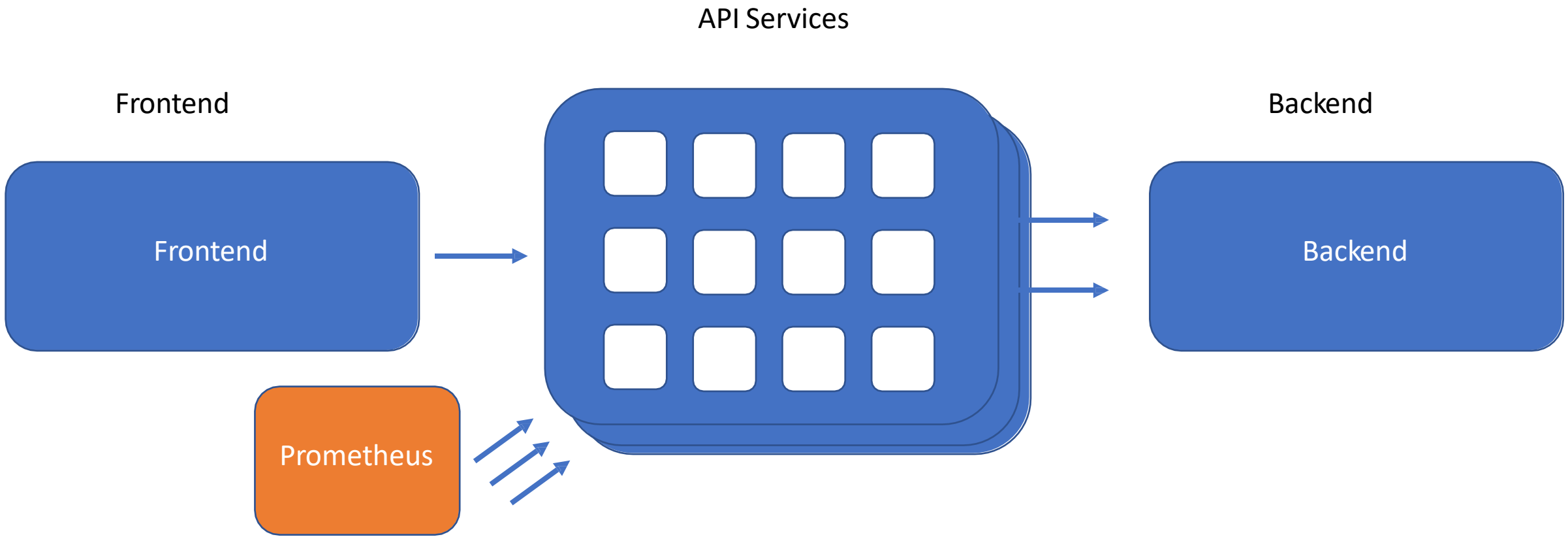
Limited visibility

Monitoring



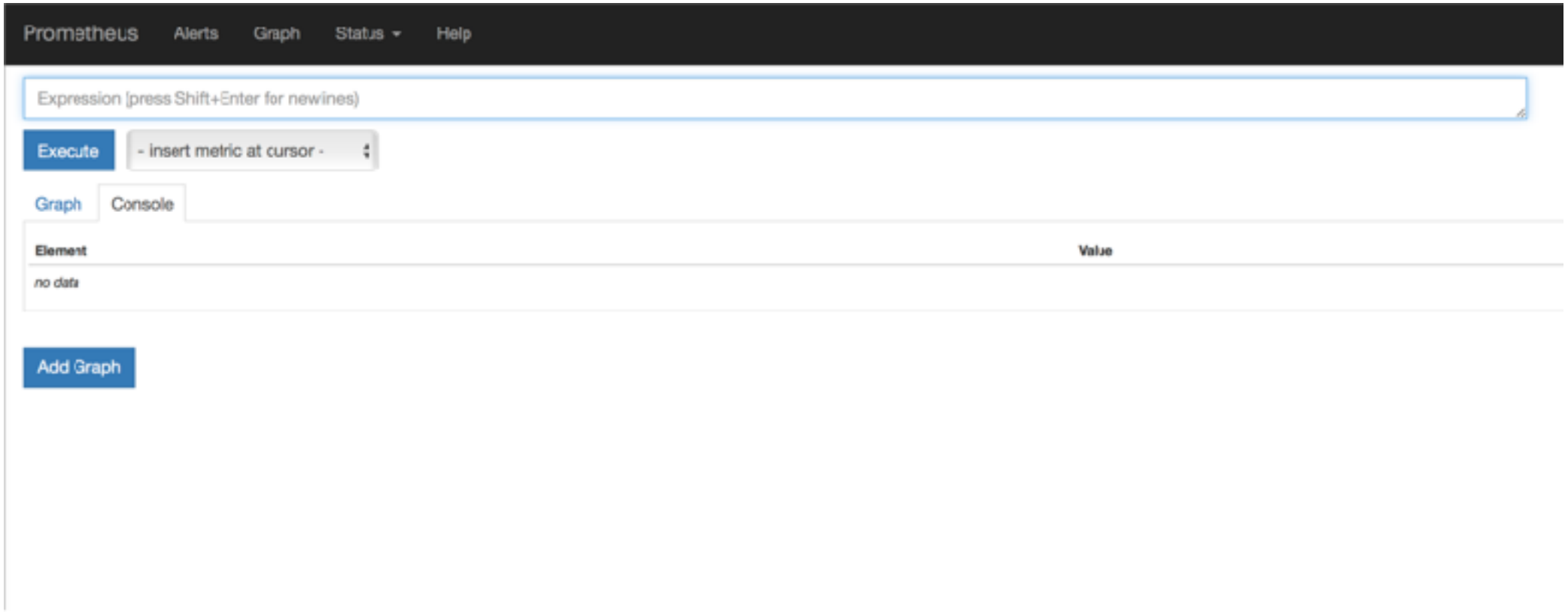
Monitor Internals

Monitoring


API Services
Frontend
Backend
Frontend
Backend
Prometheus
Monitor Services

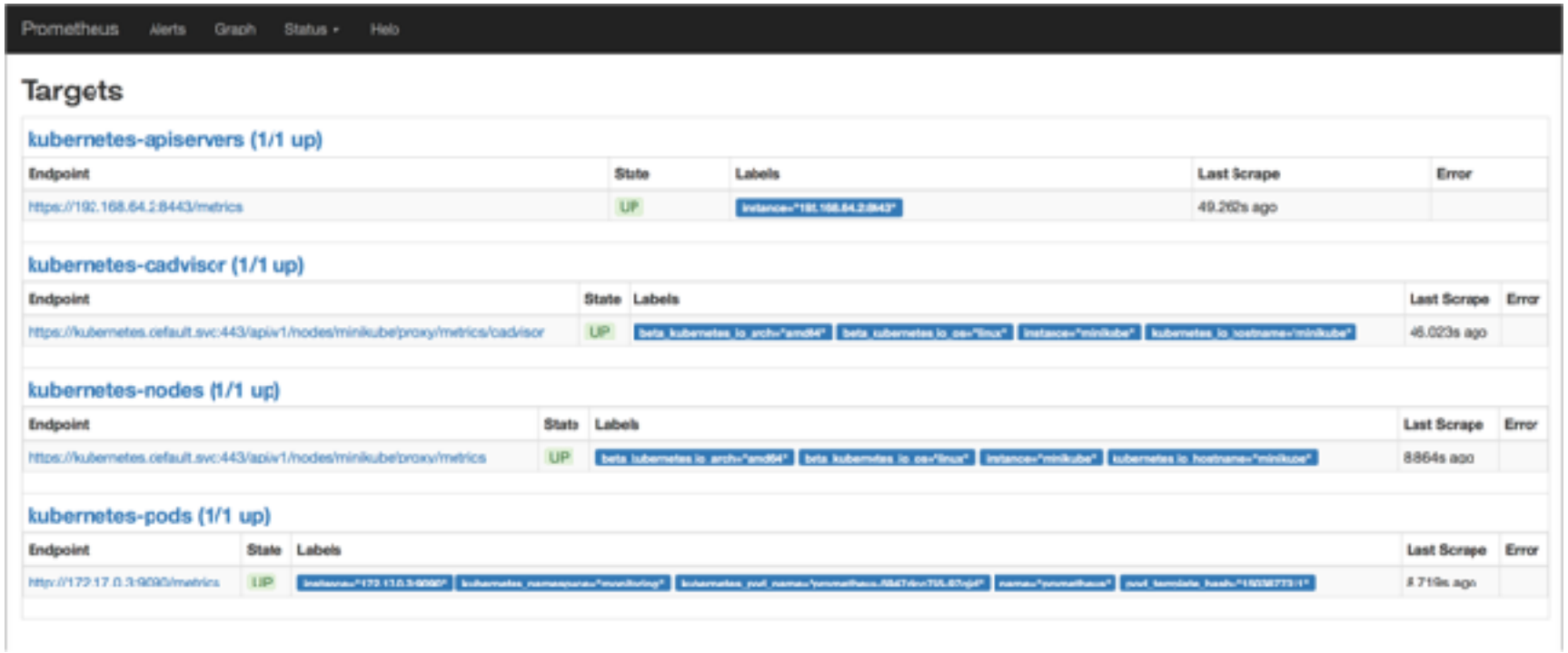
Monitoring

Prometheus



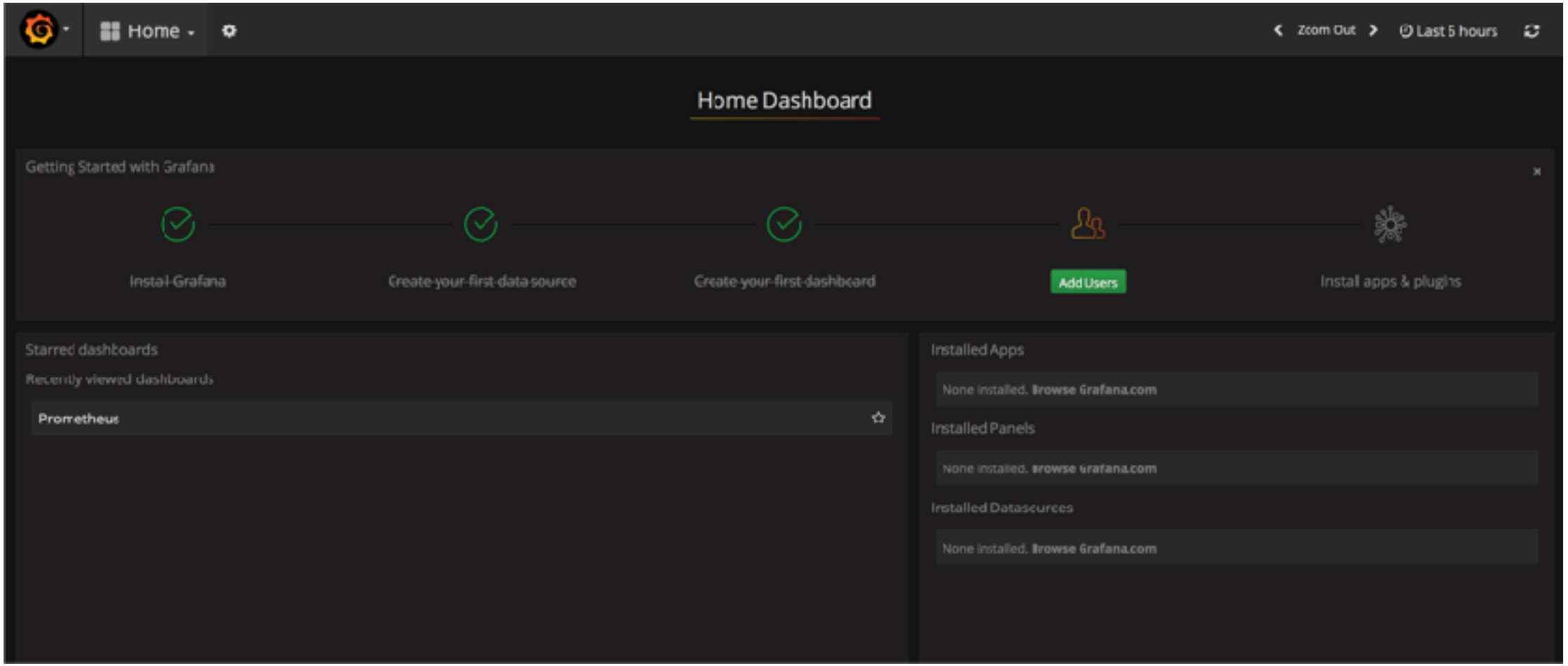
Monitoring

Prometheus



Monitoring

Grafana

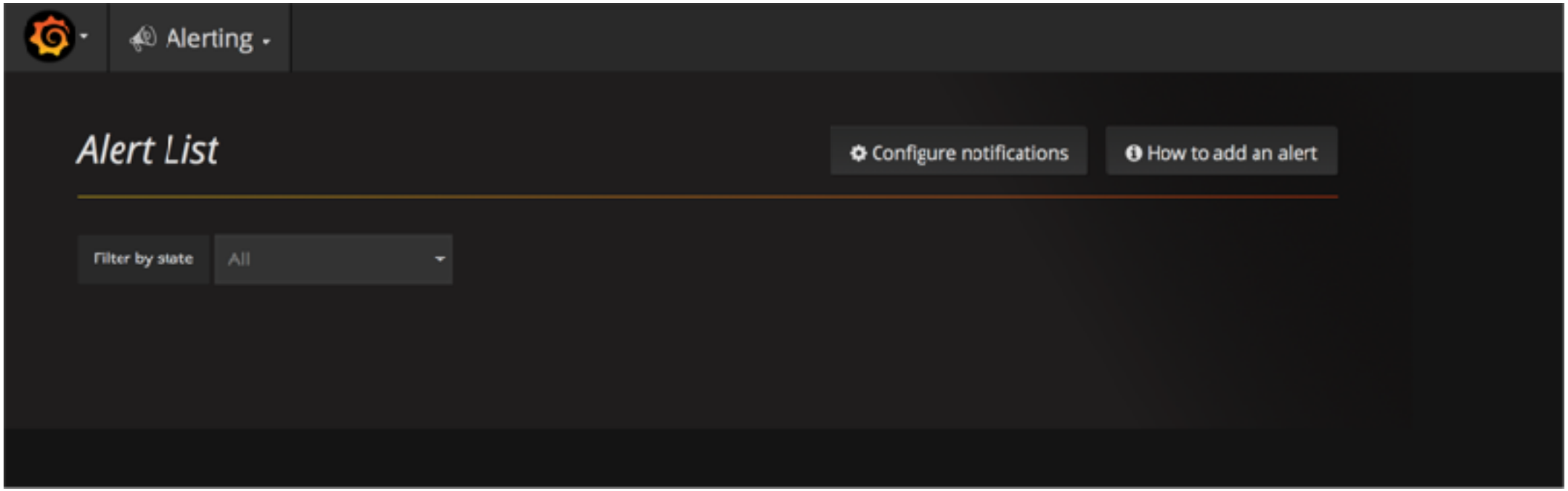


Monitoring

Grafana

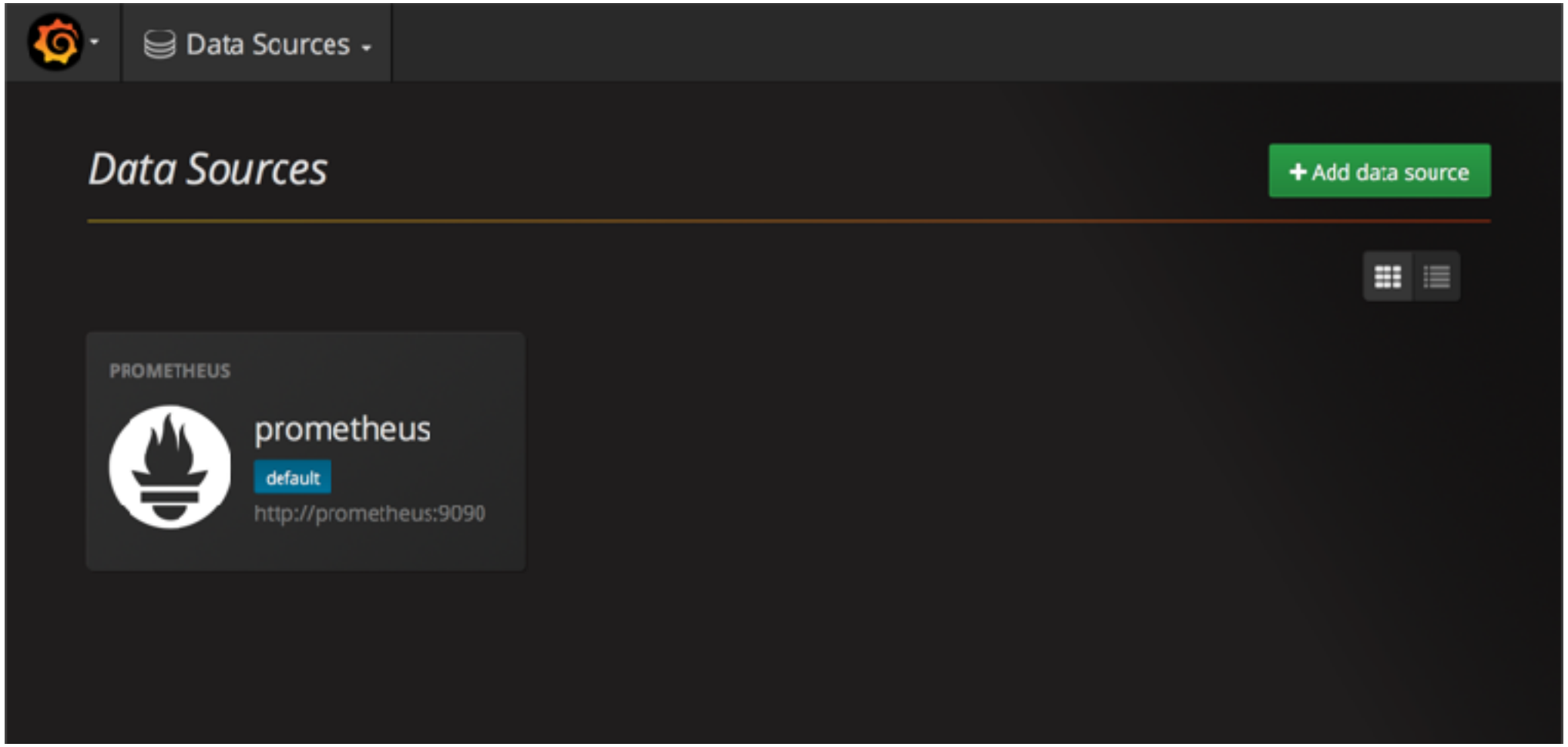
Monitoring

Grafana



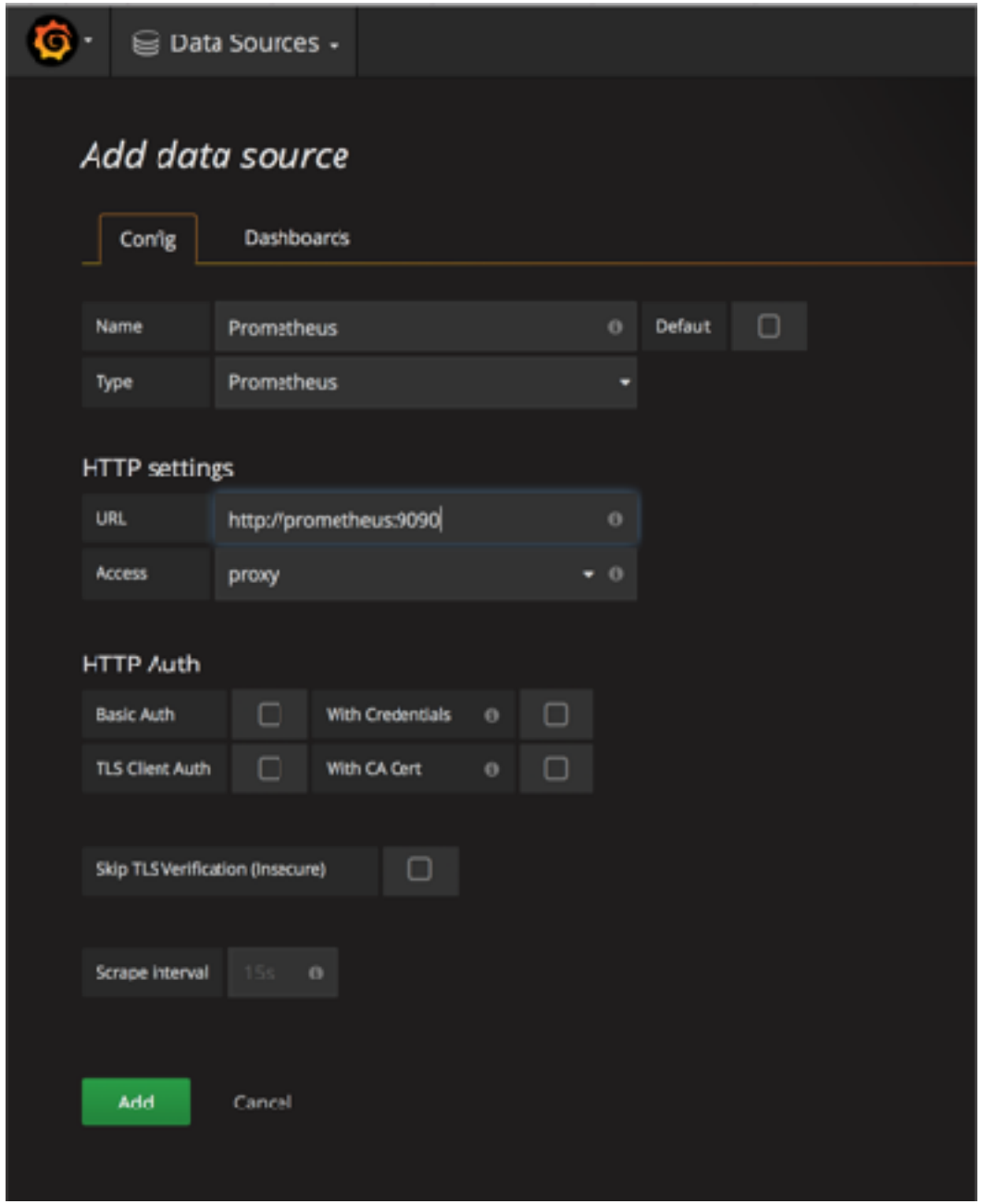
Monitoring

Grafana



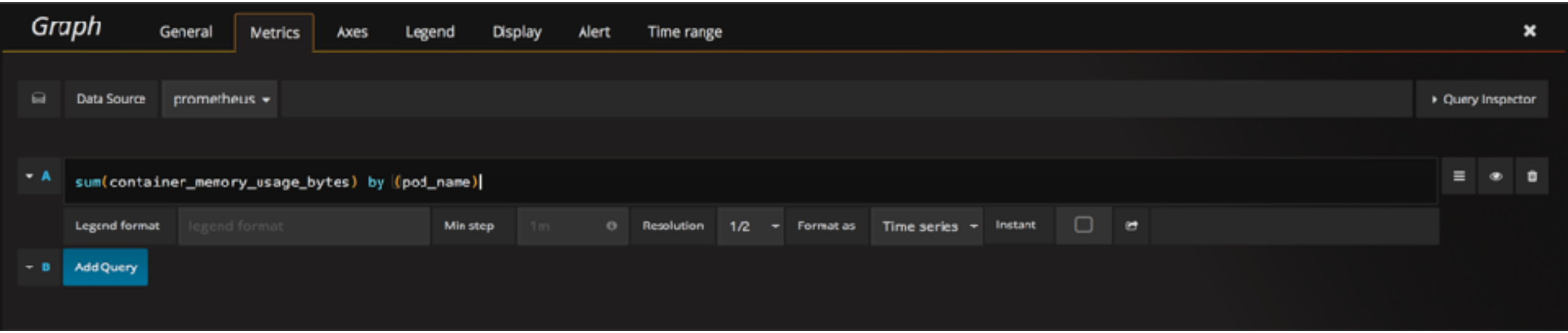
Monitoring

Grafana
&
Prometheus



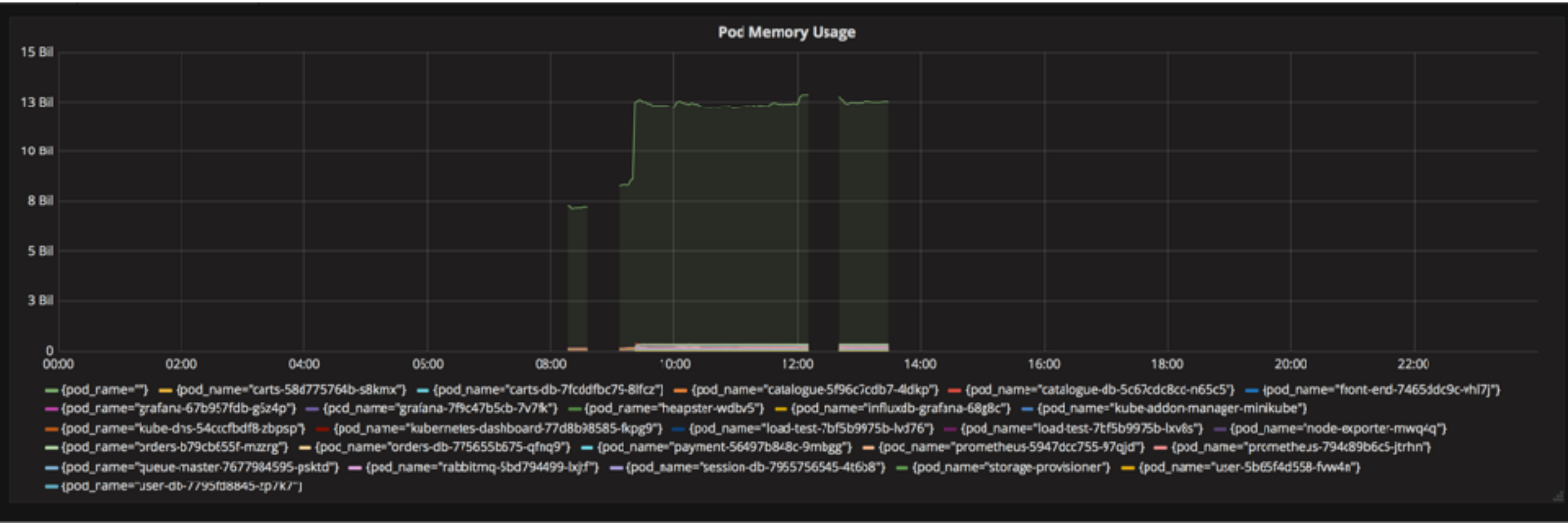
Monitoring

Add graph



Monitoring

Grafana graph



Kubernetes Secrets

Application Secrets

What secrets do applications have?

- Database credentials
- API credentials & endpoints (Twitter, Facebook etc.)
- Infrastructure API credentials (Google, Azure, AWS)
- Private keys (TLS, SSH)
- Many more!

Application Secrets

It is a bad idea to include these secrets in your code.

- Accidentally push up to GitHub with your code
- Push into your file storage and forget about
- Etc.

Application Secrets

There are bots crawling GitHub searching for secrets

Real life example:

Dev put keys out on GitHub, woke up next morning with a ton of emails and missed calls from Amazon

- 140 instances running under Dev's account.
- $2,375 worth of Bitcoin mining

Create Secret

- Designed to hold all kinds of sensitive information
- Can be used by Pods (filesystem & environment variables) and the underlying kubelet when pulling images

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
Type: Opaque
data:
   password: mmyWfoidfIuL==
   username: NyhdOKwB
```

Pod Secret

```
kind: Pod
metadata:
   name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
    - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
```

Volume Secret

```
spec:
  containers
  - name: mycontainer
    image: redis
    volumeMounts:
    - name: “secrets”
      mountPath: “/etc/my-secrets”
      readOnly: true
  volumes:
   - name: “secrets”
      secret:
        secretName: “mysecret”
```

Secrets Lab

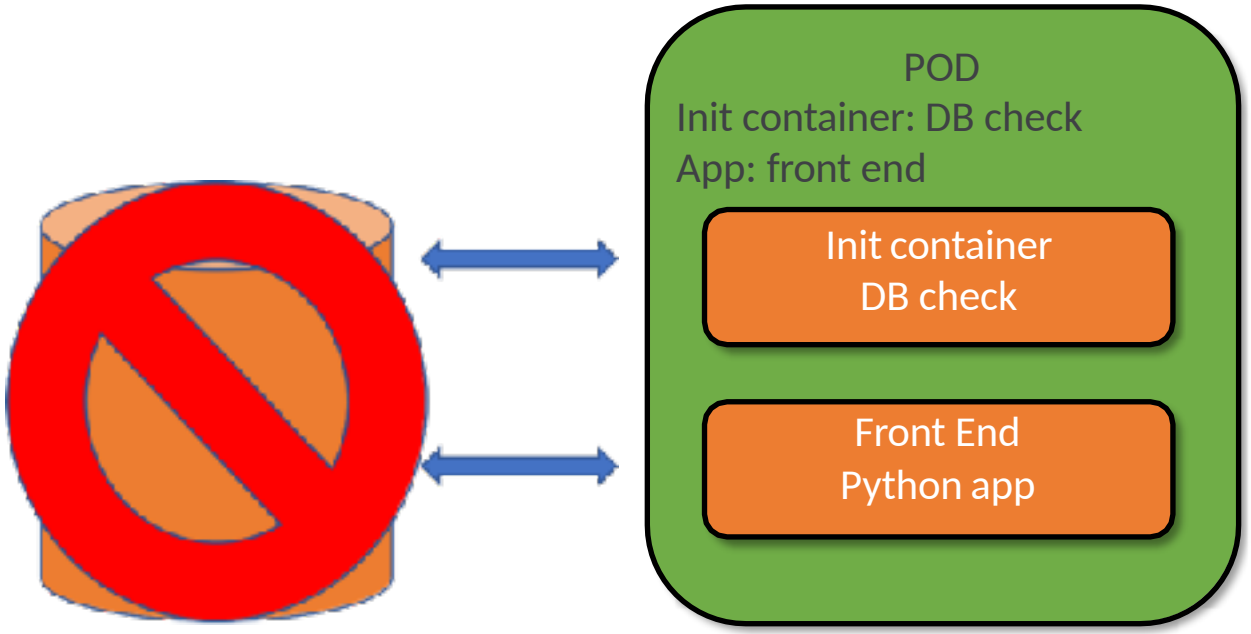
- Create secrets file
- Deploy application that uses secrets
- Decrypt secrets

Kubernetes Init Containers

Init containers

Application orchestration

- Init container runs first to completion
  - If failure, POD is restarted until successful.



Init containers

- Support same features as application containers.
  - Resource limits
  - Volumes
  - Security settings
  - Etc.
- Do not support readiness probes since they must run to completion before POD can be “ready”

Kubernetes 1.5 syntax

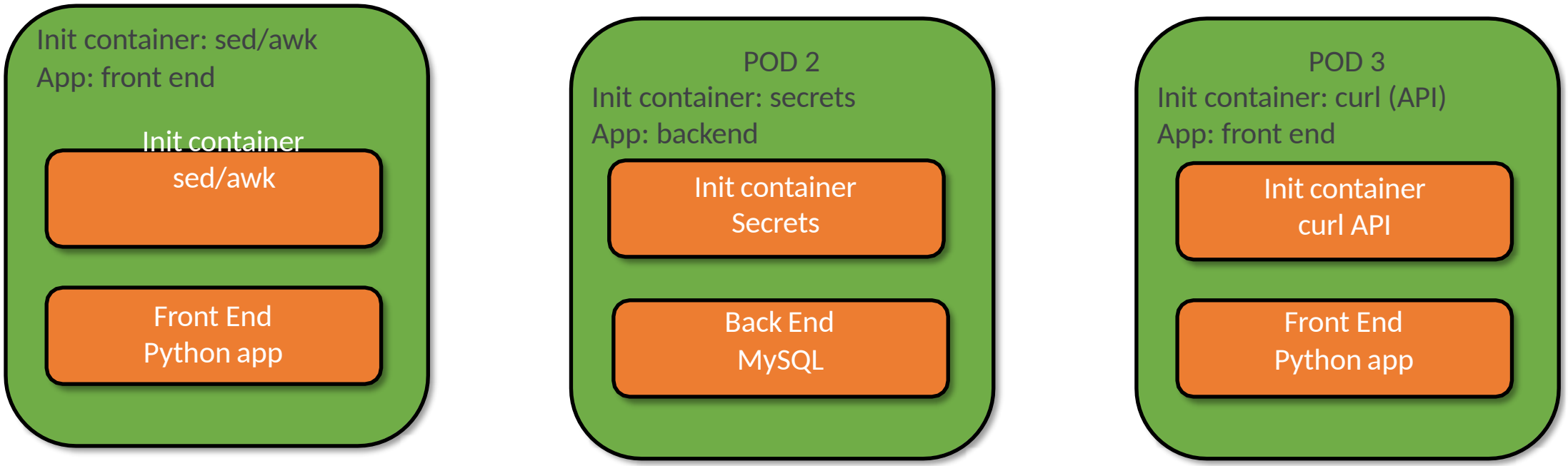
```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
  annotations:
    pod.beta.kubernetes.io/init-containers: '[
        {
            "name": "init-myservice",
            "image": "busybox",
            "command": ["sh", "-c", "until nslookup myservice; do echo waiting for myservice; sleep 2; done;"]
        },
        {
            "name": "init-mydb",
            "image": "busybox",
            "command": ["sh", "-c", "until nslookup mydb; do echo waiting for mydb; sleep 2; done;"]
        }
    ]'
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
```

Kubernetes 1.6+ syntax

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
  - name: init-mydb
    image: busybox
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

Init container use-cases

- Lots of reasons



Init containers Lab

- Deploy POD with init containers that wait for services to come online
- Investigate POD status
- Create services
- Confirm init containers executed and POD was started

- Create POD with init container that fetched webpage for nginx web server.

Local Kubernetes Cluster

Minikube: Run Kubernetes All-In-One

Run single-node Kubernetes cluster on local machine

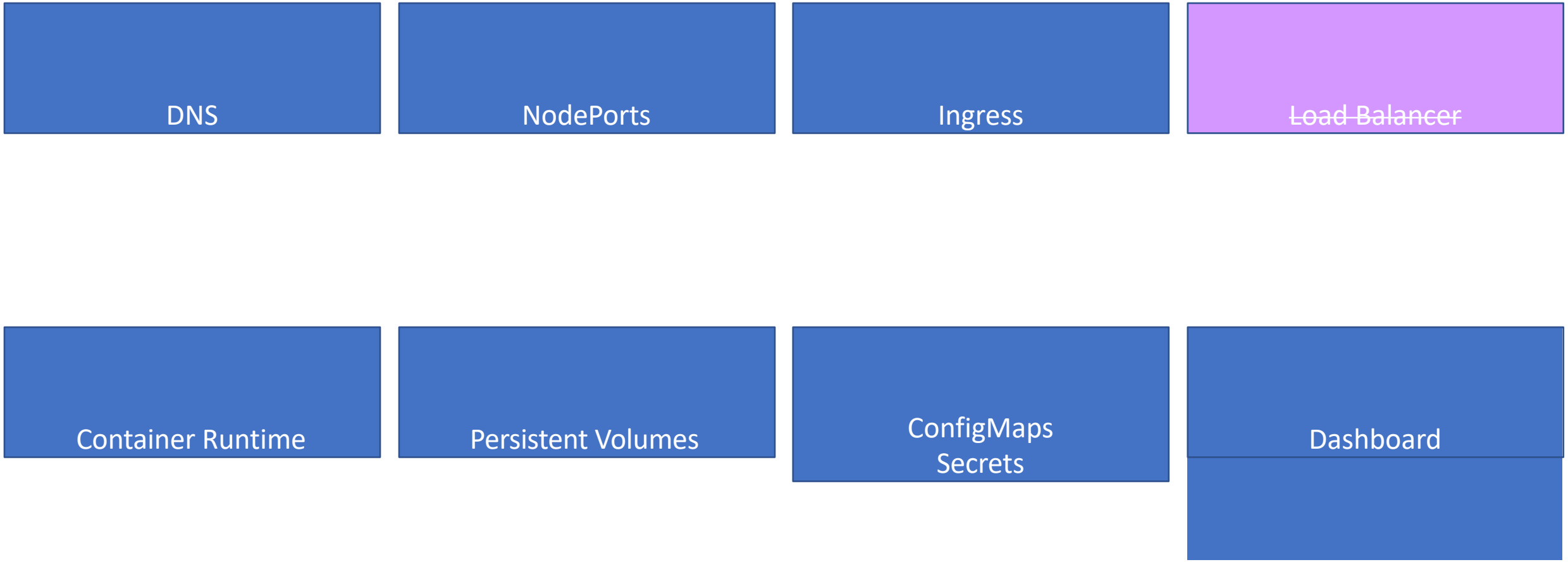
- Minikube runs on all 3 operating systems
  - Windows, Mac, and Linux

- VirtualBox
- Hyper-V

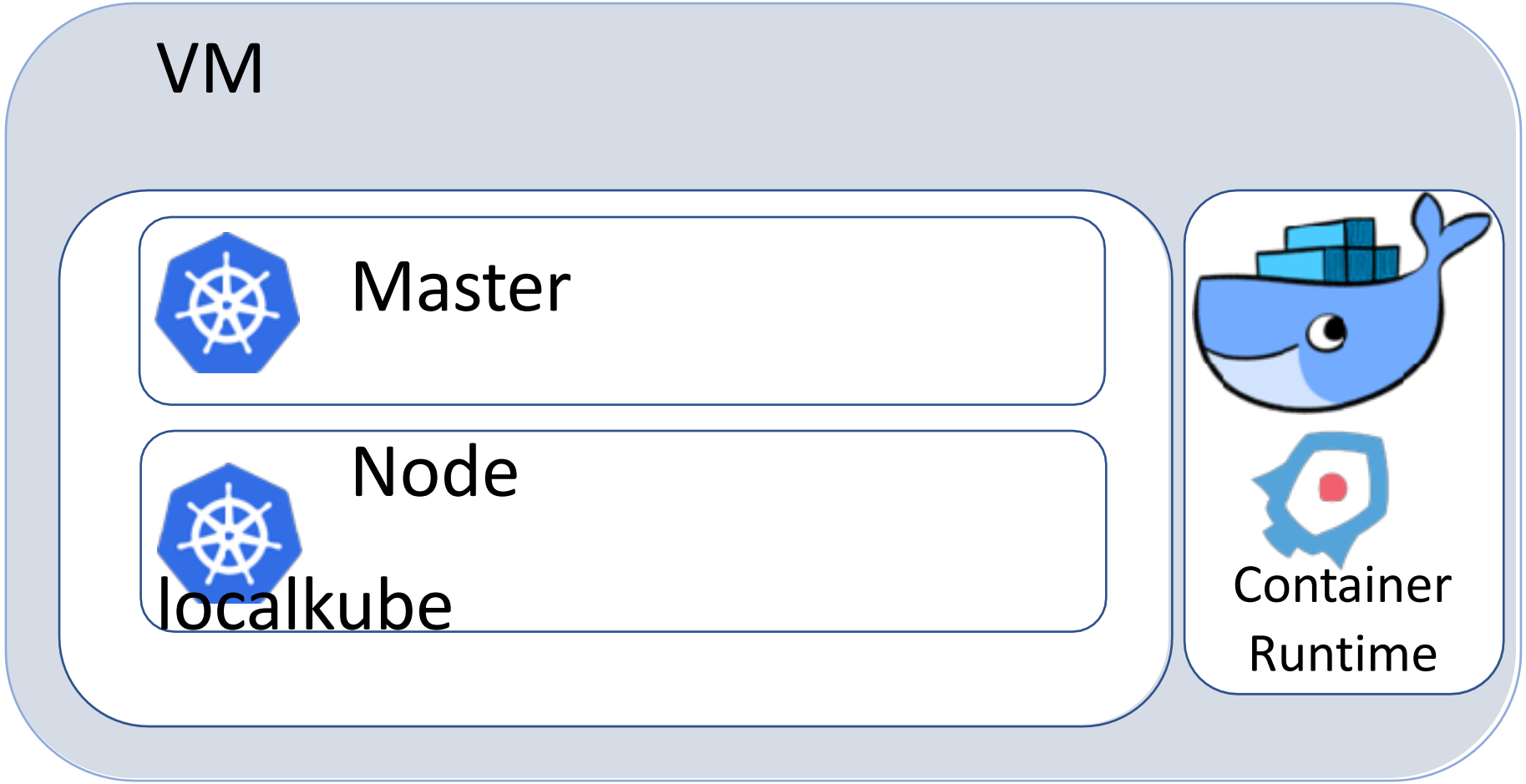
- VirtualBox
- Xhyve
- Vmware Fusion

- VirtualBox
- KVM

Minikube: Supported Features



Minikube: Architecture



Capstone Lab

- Create ConfigMap that contains value of “success”
- Create secret with a passphrase (i.e. “KubernetesRocks”
- Deploy web application:
  - Serves webpage with contents of ConfigMap
  - Secret is available at /secrets/passphrase.txt
- Create service for web application
  - Served on port 8081
  - Available externally