Isle of Cats: Inside Calico Networking for Kubernetes

Kubernetes Atlanta Meetup
28 August 2019

TIGERA

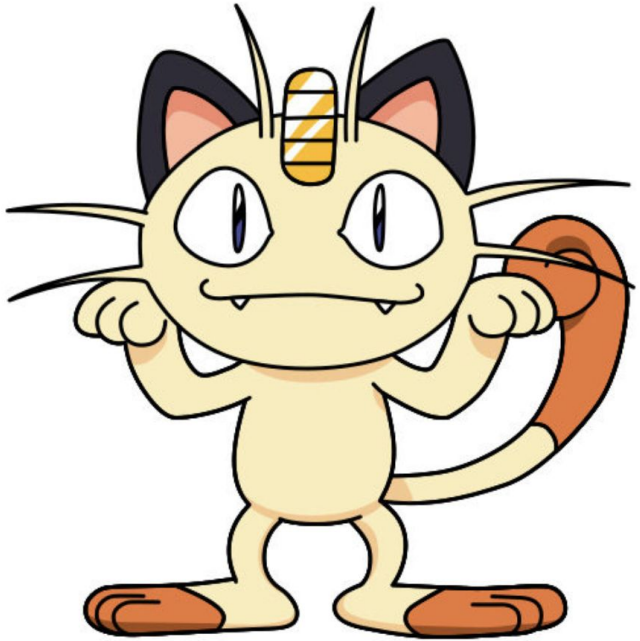
Drew Oetzel - Senior Technical Solutions Engineer

- Working with enterprise software since the late 90s
- 7 years at Splunk, honing his security skills
- 2.5 years at Mesosphere, then Heptio mastering the art of distributed systems, containers, and all that goes along with them
- Outside of tech ask him about history, gardening, or what he’s doing to try to curb his Reddit addiction!

TIGERA

WHO IS TIGERA?

A super secret Pokemon?



No, but we do have a conference room named after Meowth!

Who Is Tigera?

Tigera Secure

- Zero Trust Network Security
- Visibility, Traceability and Remediation for Dynamic Applications
- Continuous Compliance & Enterprise Controls
- Security that Spans Multi-Cloud and Legacy Environments

Tigera Calico

- Open-source
- Scalable, distributed control plane
- Policy-driven network security
- No overlay required
- Widely deployed, and proven at scale

Bloomberg GitHub salesforce NTT Communications IBM CommonwealthBank orange virtustream EMC² YAHOO! box BBVA

SQUARESPACE Giant Swarm ebay idealo USA TODAY YAHOO! JAPAN at&t CenturyLink Ford JPMorganChase

TIGERA

Tigera Strategic Partnerships

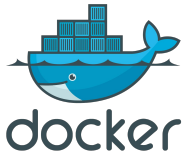
Integrated with ACS Engine, Collaborating on Azure Kubernetes Service (AKS) and Windows

Default network policy for Azure Container Service for Kubernetes (EKS) & Heptio/AWS Quick Start

Partnered to support network policy in Google Container Engine (GKE)

Partner to implement connectivity and security for IBM Kubernetes Service, IBM Cloud Private

OpenShift primed partner. Certified integration with OpenShift Container Platform



Default networking and policy for Docker EE Kubernetes

TIGERA

TIGERA: Leader in Network Security for Kubernetes

PROJECT CALICO

Inventor and Maintainer of Project Calico, 100K known clusters

Tigera Secure

Extend firewalls to Kubernetes, continuous compliance, and more!

Deployed to Complex, Multi-Cloud Environments

box workday. UBISOFT Bloomberg Morgan Stanley AT&T Ford VISA

ATLASSIAN JPMorganChase DISCOVER

WHAT IS CALICO?

“Networking” for containers.

Open source project with worldwide contributors backed by the commercial enterprise Tigera, Inc.

Most used CNI plugin for Kubernetes worldwide.

www.projectcalico.org

OK, BUT WHAT CAN CALICO DO FOR ME?

Efficiently hand out IPs to your pods. (IPAM)

- **Can use different subnets per namespace or rack / VPC**
- **Can assign static IP addresses to pods**

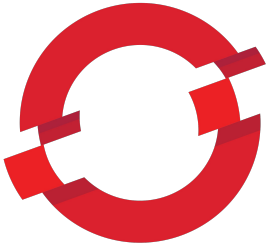
Secure your inner K8s networking with policies.

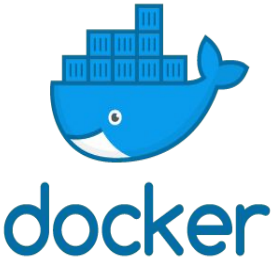
- **Secure your East/West traffic**
- **Prevent malicious behavior from spreading**
- **Block insider threat / insider error**

Mix and match components - can run Calico for just network policies with other CNIs

Where to run the Calico CNI?

Upstream K8s - bare metal or cloud

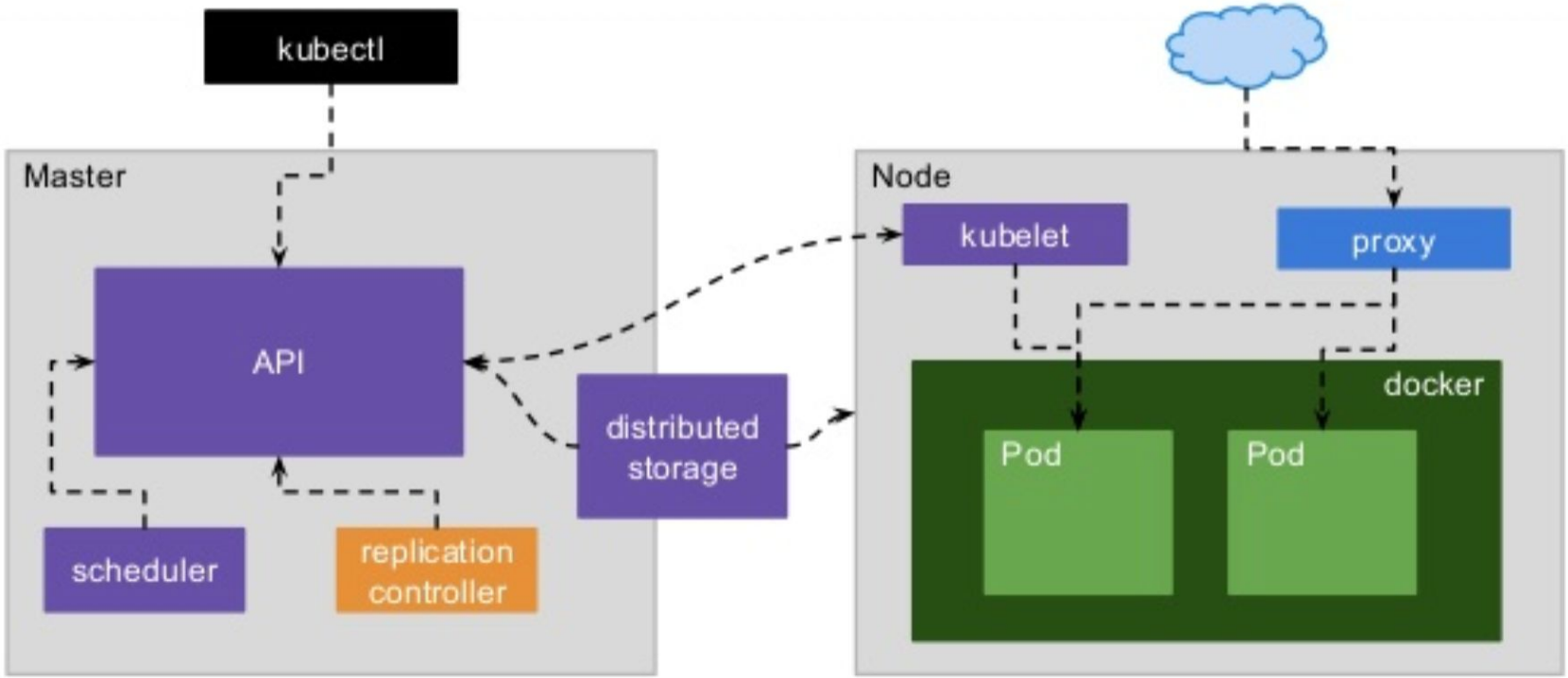




The artist formerly known as Mesosphere

RUNNING CALICO with KUBERNETES

Kubernetes architecture



INSTALLING CALICO on KUBERNETES

- **Calico is usually the first thing you install right after you setup Kubernetes - in fact the cluster won't be "ready" until you do.**
- **Calico is can be configured to communicate directly with etcd or it can communicate with the Kubernetes API Datastore**
- **Be sure you select the correct installation yaml from the web page depending on which option you choose**

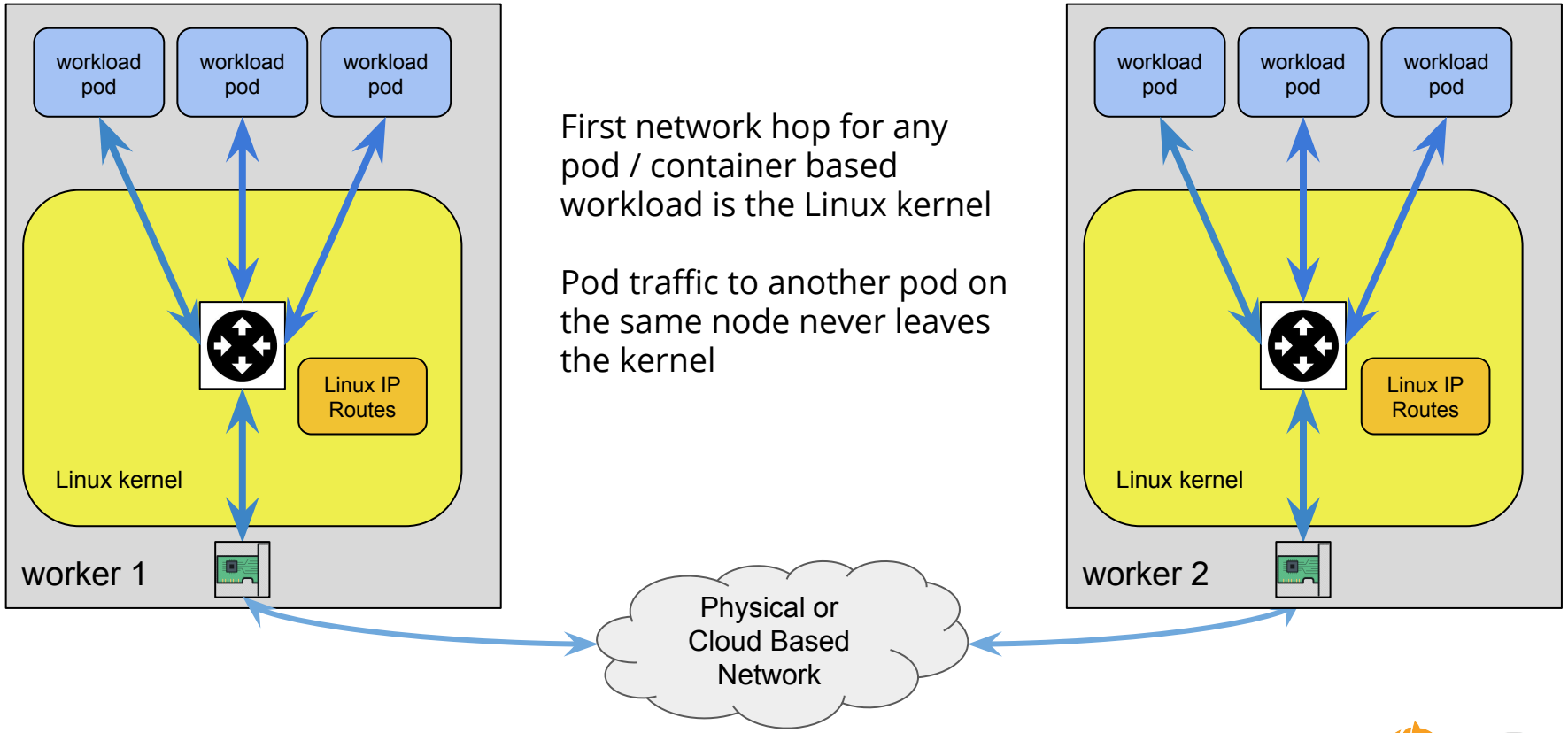
API Datastore:

- https://docs.projectcalico.org/v3.7/getting-started/kubernetes/installation/calico#installing-with-the-kubernetes-api-datastore50-nodes-or-less

Etcd Directly:

- https://docs.projectcalico.org/v3.7/getting-started/kubernetes/installation/calico#installing-with-the-etcd-datastore

Intra Pod and Intra Node Networking

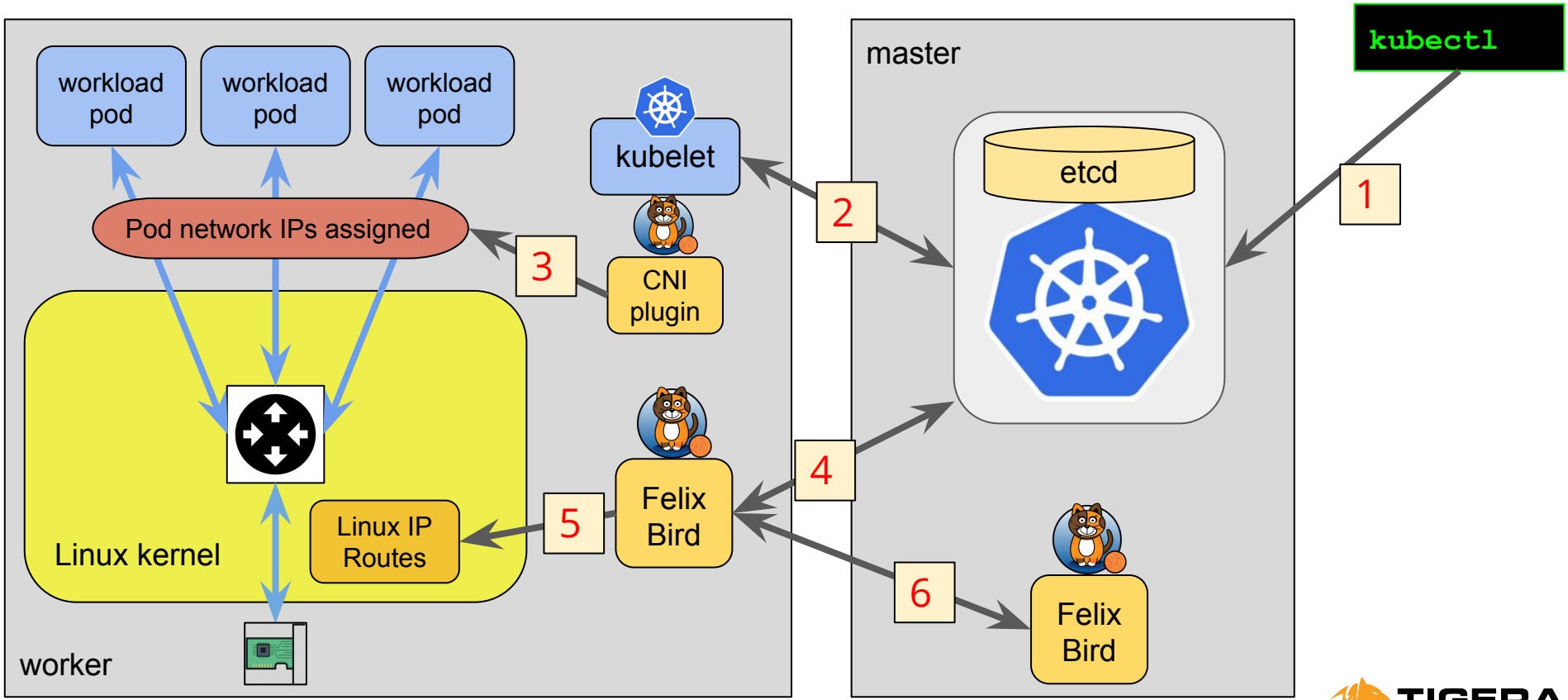


First network hop for any pod / container based workload is the Linux kernel

Pod traffic to another pod on the same node never leaves the kernel

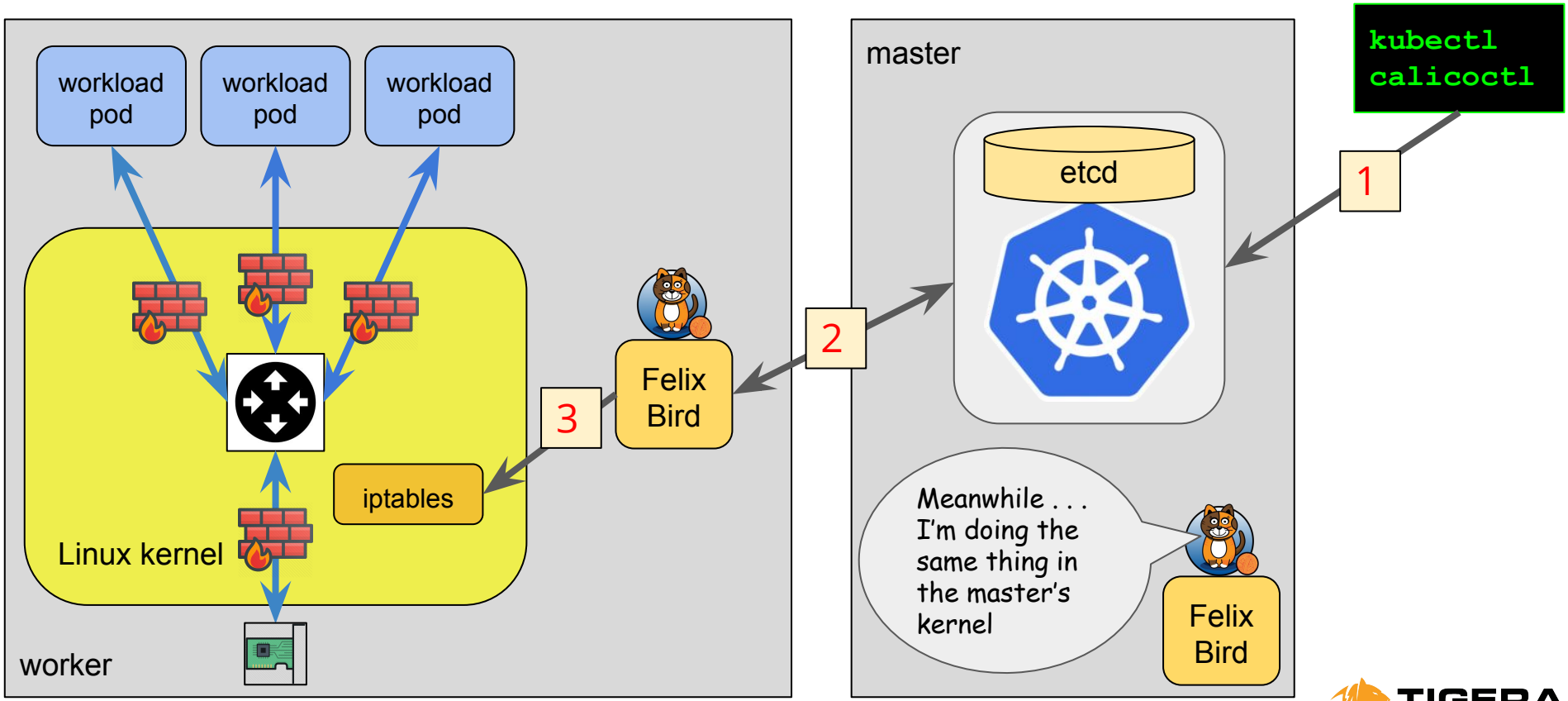
TIGERA

High Level Calico CNI Architecture



TIGERA

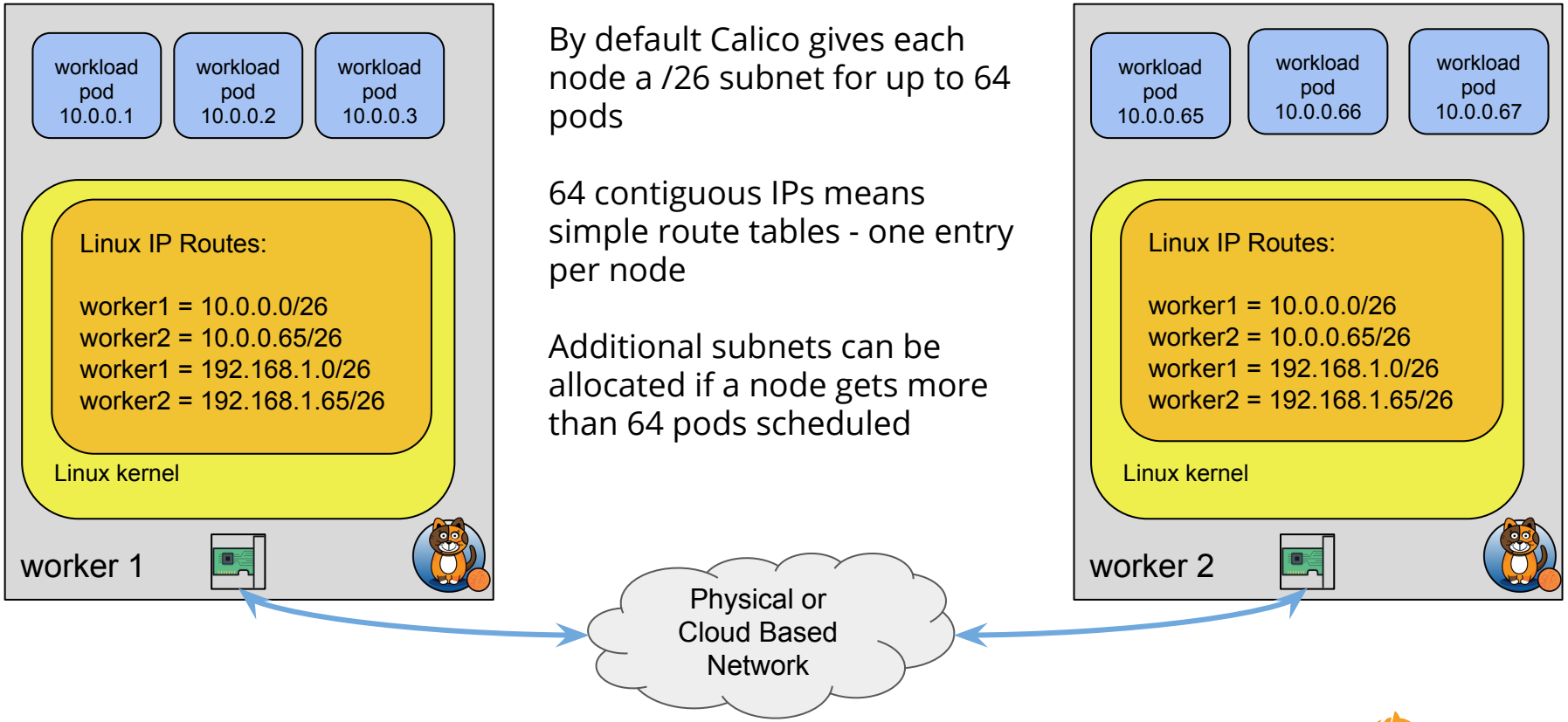
Calico High Level Policy Architecture



TIGERA

Calico CNI and IPAM in Action

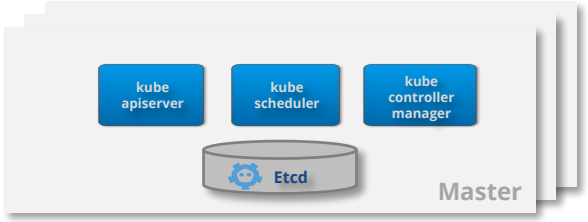
Calico Gives Each Node a Subnet

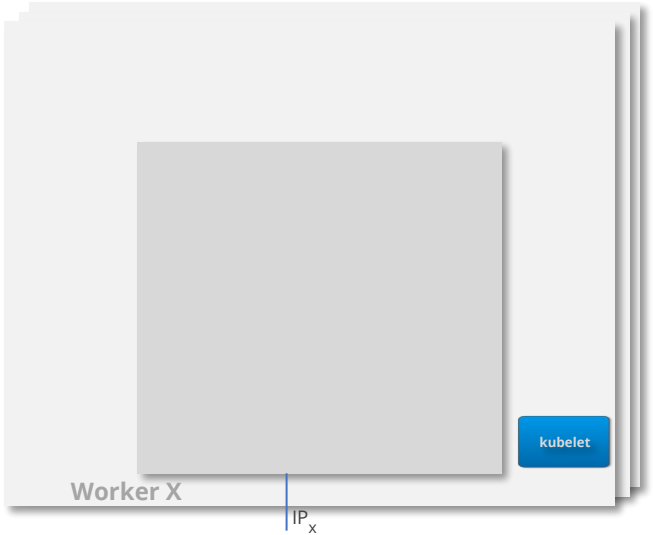


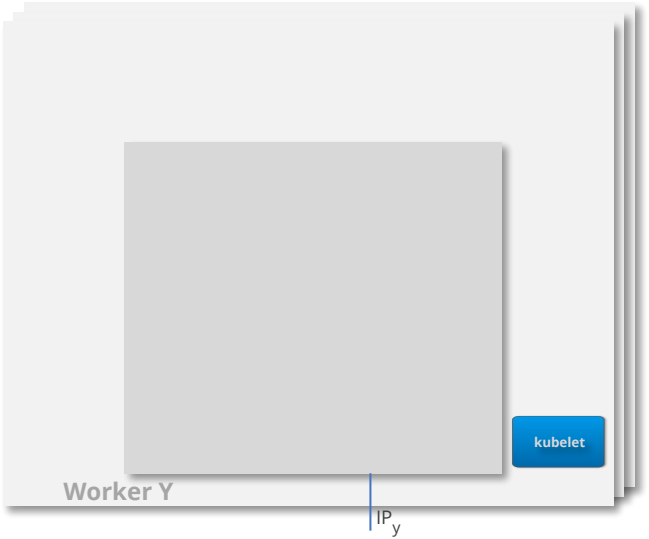
TIGERA

Pod Lifecycle

Lets start with baseline K8s nodes







Pod Lifecycle

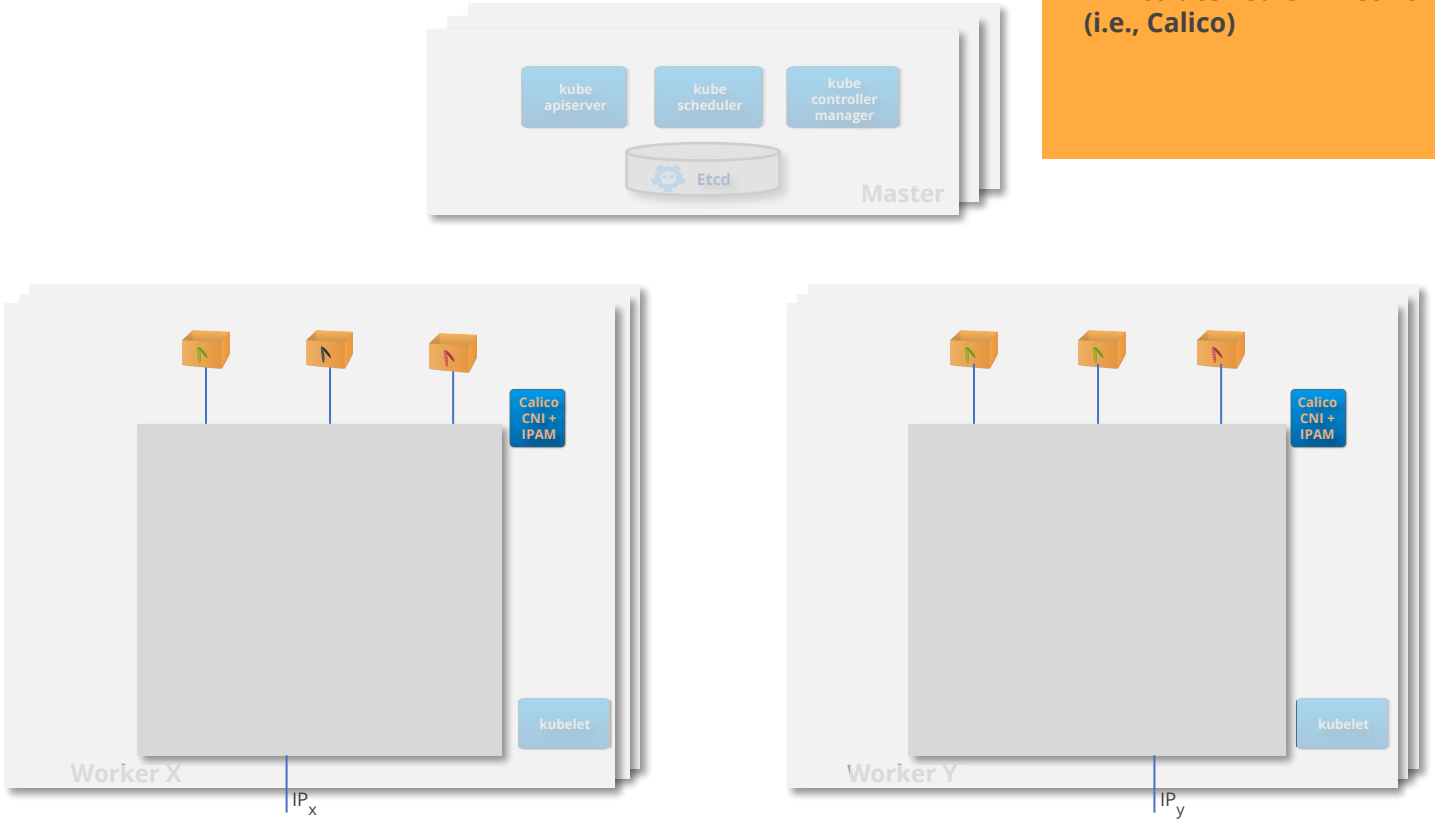
Pods get scheduled on worker nodes

Pod Lifecycle: CNI

Kubelet on worker node calls CNI

Pod Lifecycle: CNI

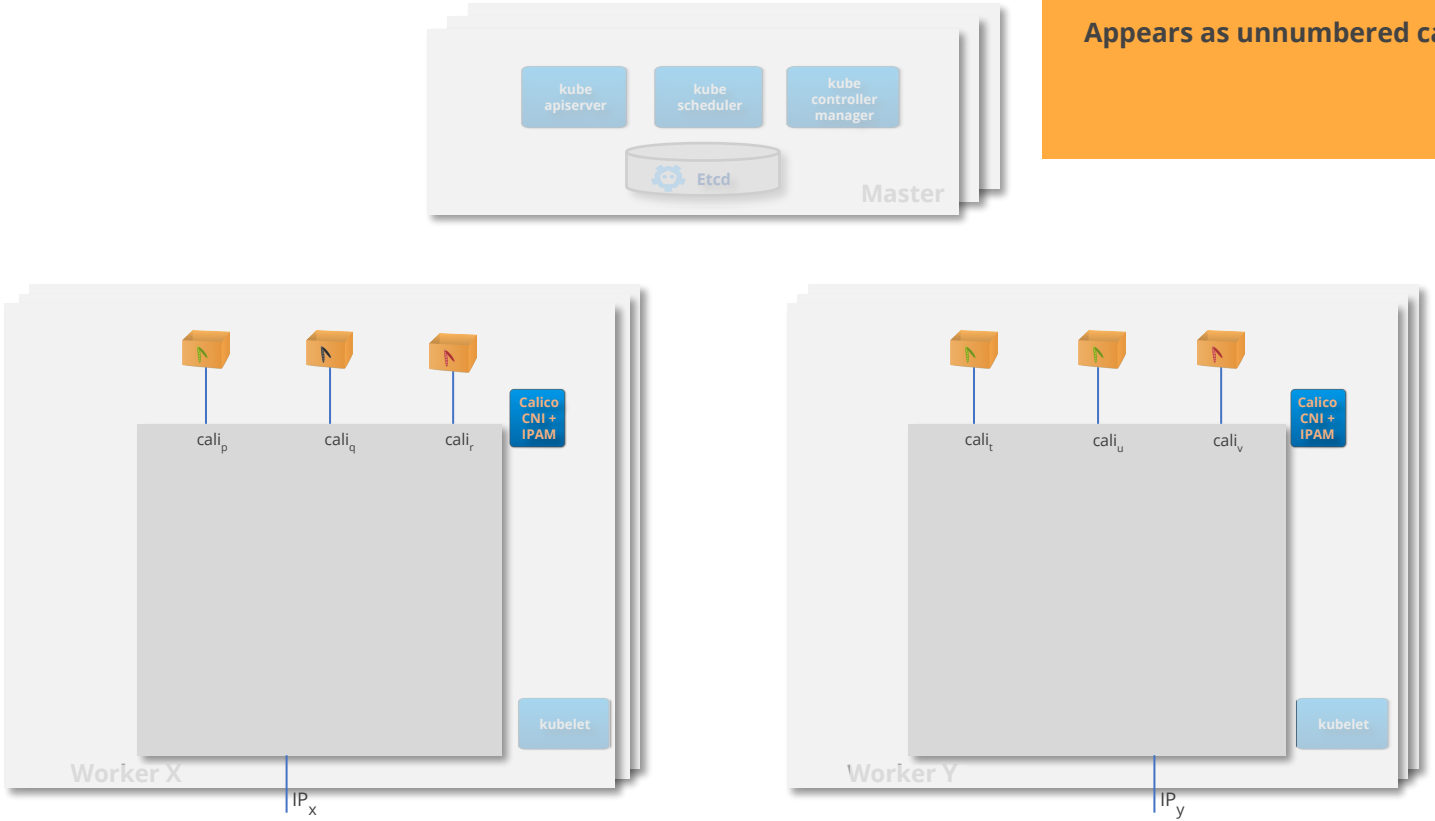
Configuration in /etc/cni/net.d provides link to desired CNI network plugin (i.e., Calico)



Pod Lifecycle: CNI

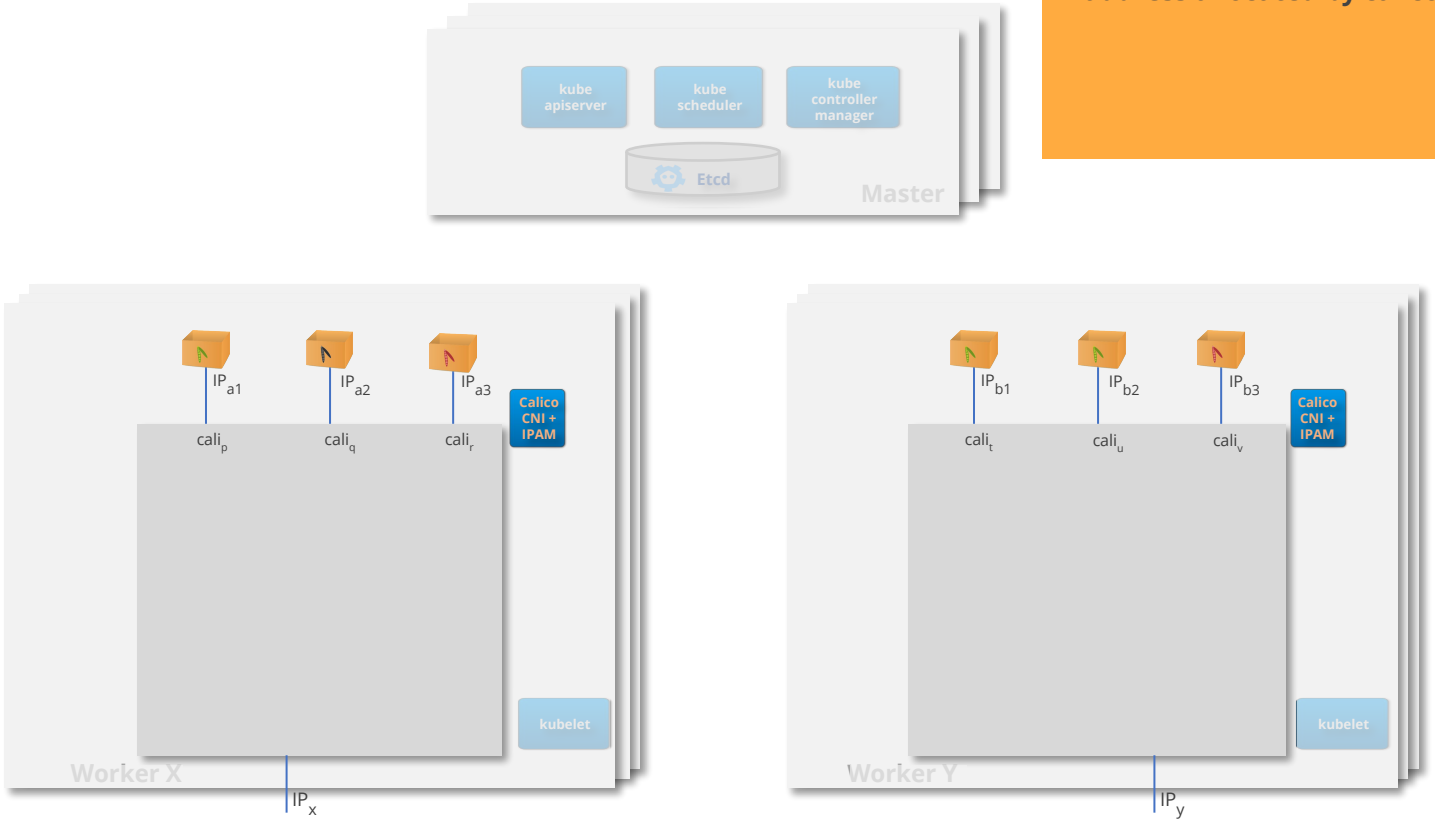
Calico-cni creates virtual ethernet (veth)

Appears as unnumbered cali*** on host



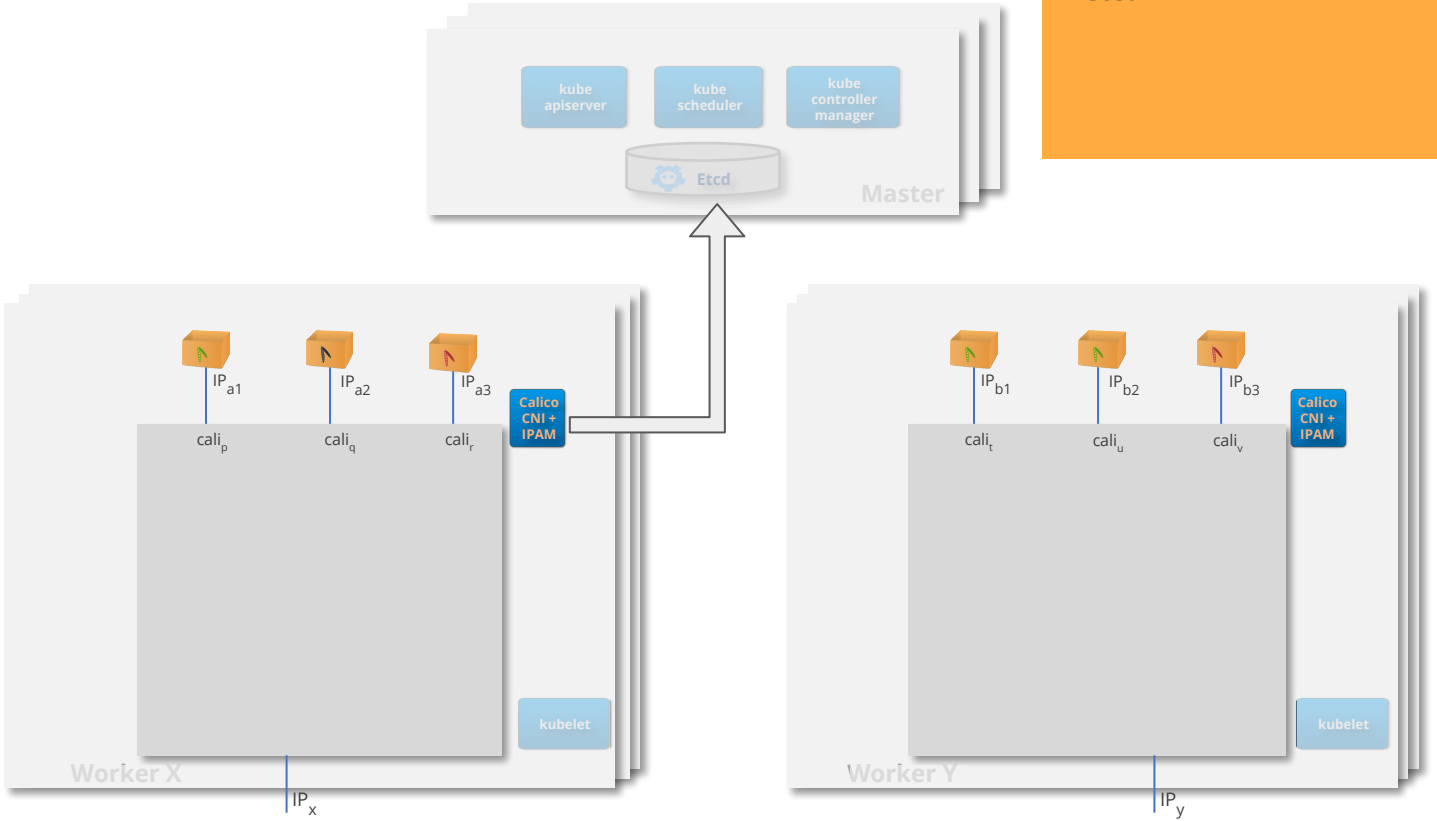
Pod Lifecycle: Address Block Assignment

And appears as eth0 inside pod, with IP address allocated by calico-ipam

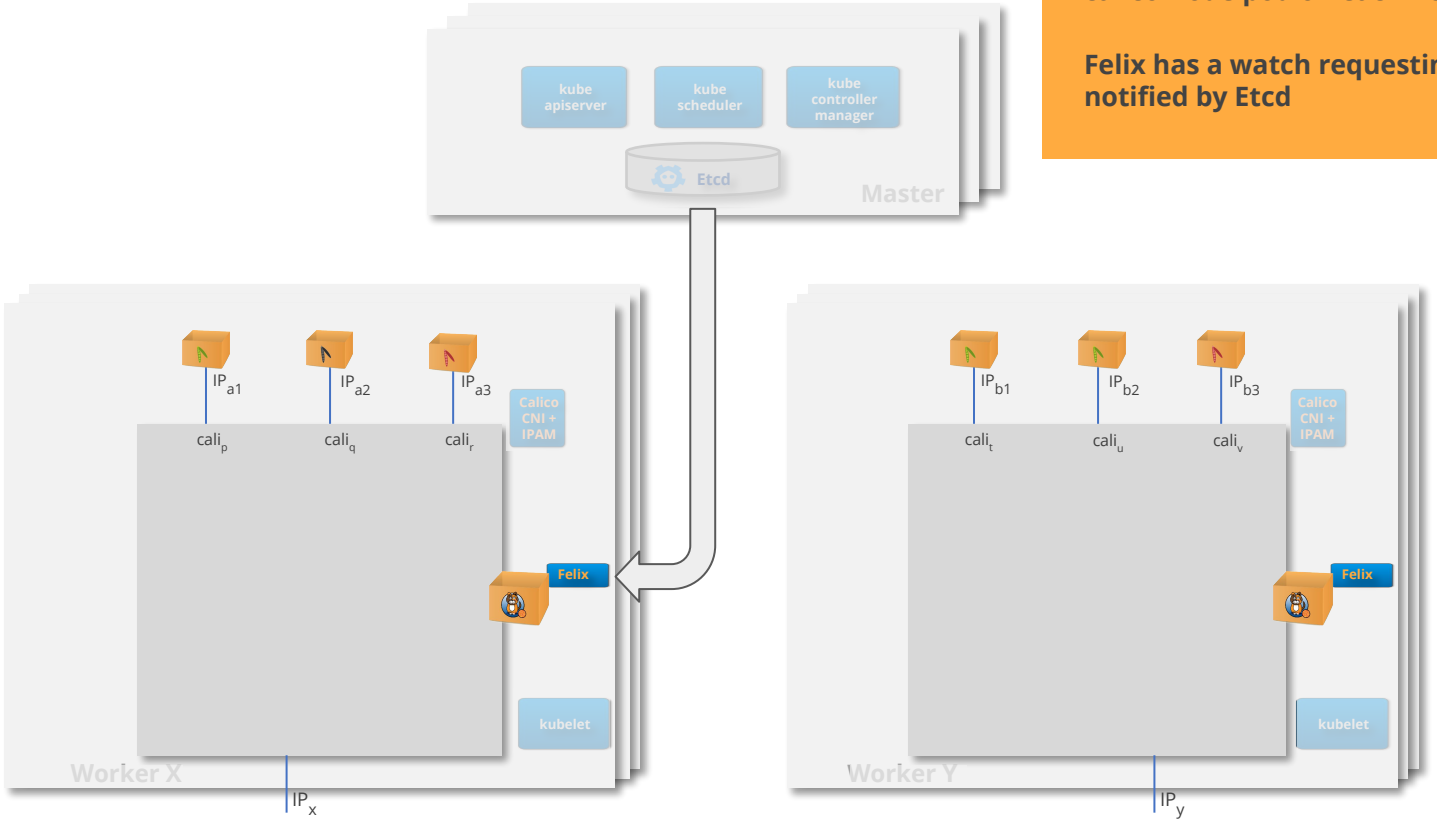


Pod Lifecycle

Workload (pod) information written to etcd



Pod Lifecycle

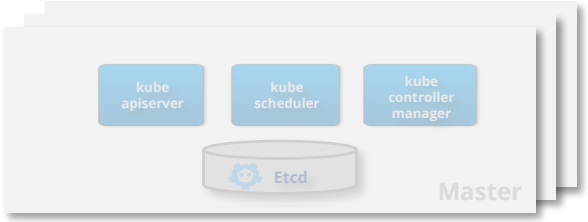


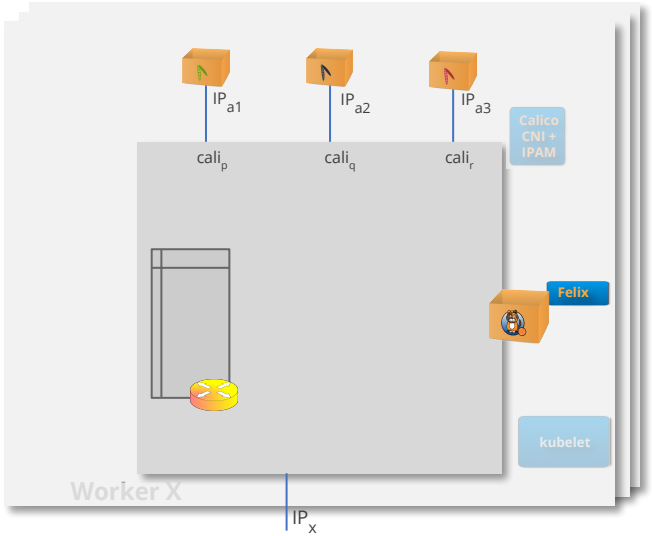
Felix daemon process runs inside the calico-node pod on each host

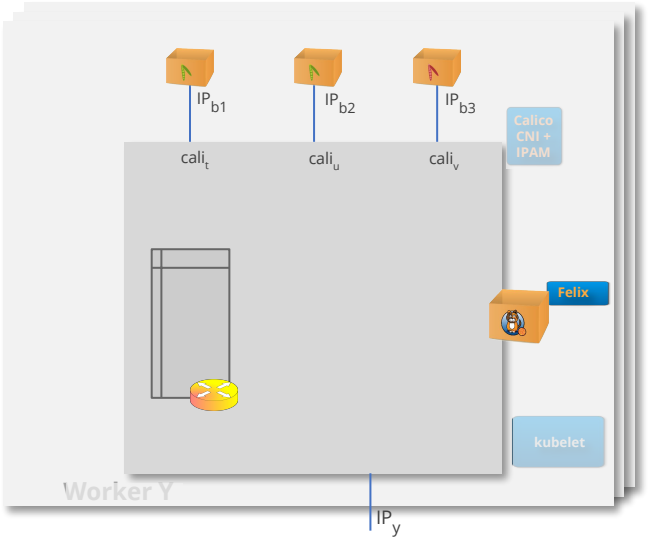
Felix has a watch requesting to be notified by Etcd

Pod Lifecycle

Felix adds local interface routes into Linux routing table

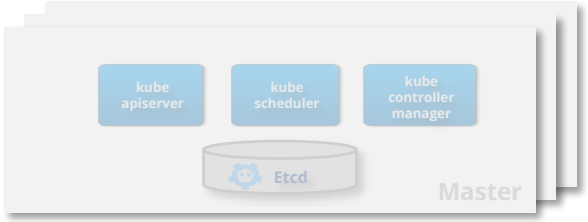


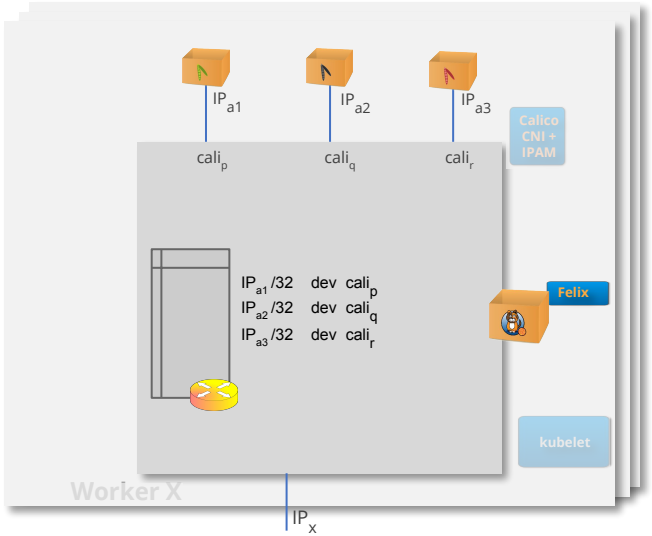


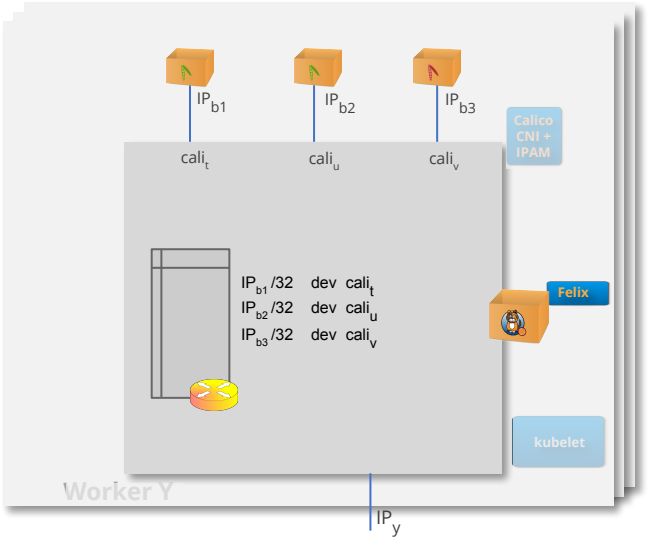


Pod Lifecycle

Route added for each pod's IP pointing to corresponding cali* virtual ethernet**



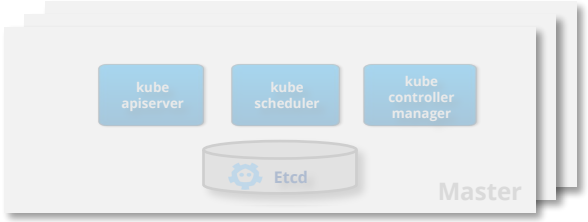


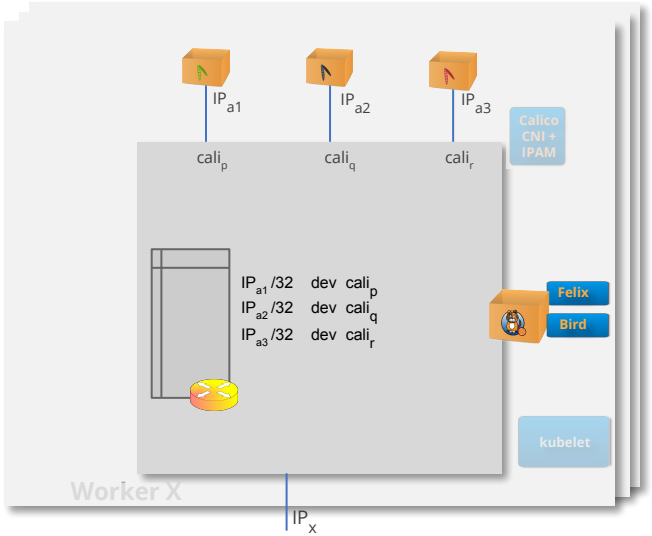


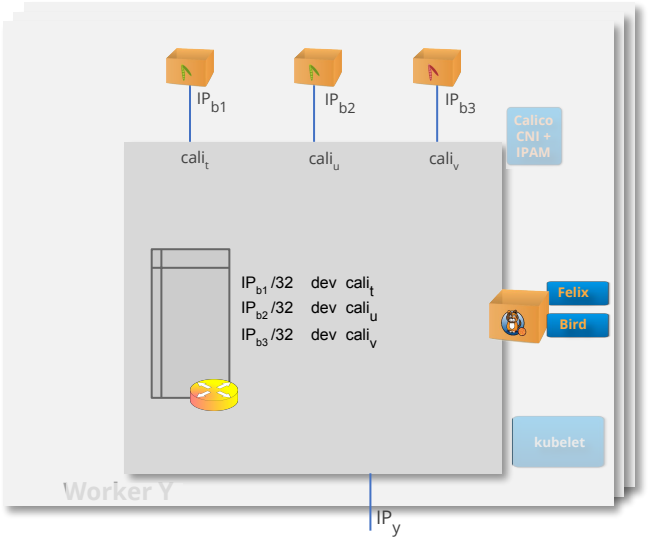
Pod Lifecycle

BGP daemon (BIRD) process runs inside the calico-node pod on each host

By default, each BGP daemon is configured to peer with all other hosts



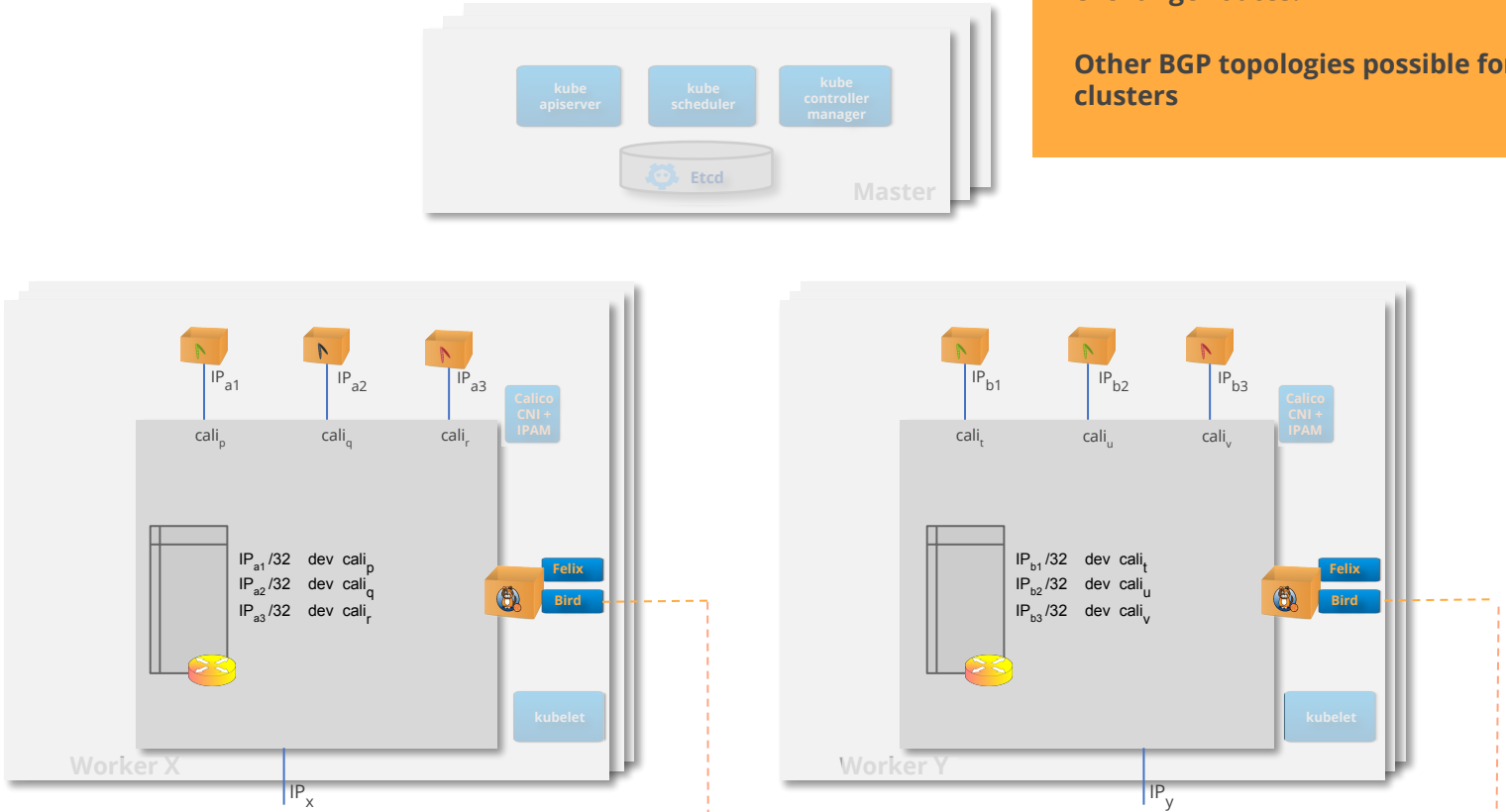




Pod Lifecycle

Nodes create a BGP full mesh to exchange routes.

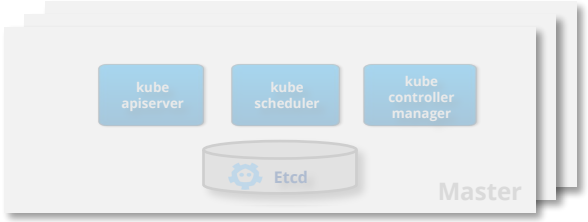
Other BGP topologies possible for large clusters

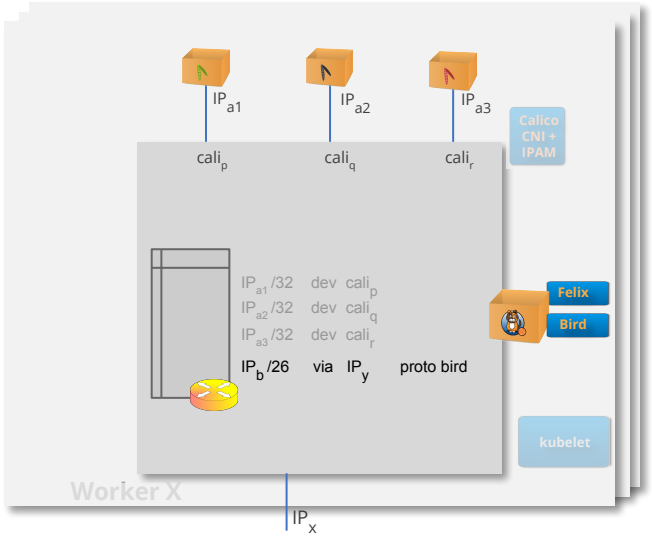


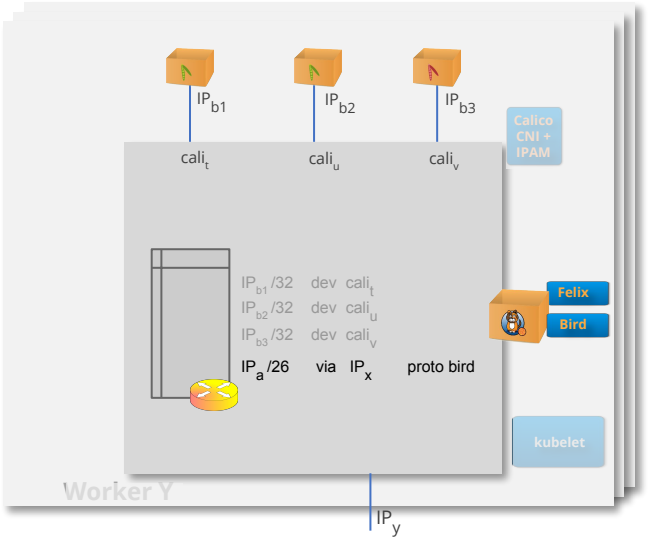
Pod Lifecycle

Routes for Pod IP addresses exchanged among nodes via BGP.

Each Linux node simply routes packets using the default routing table.







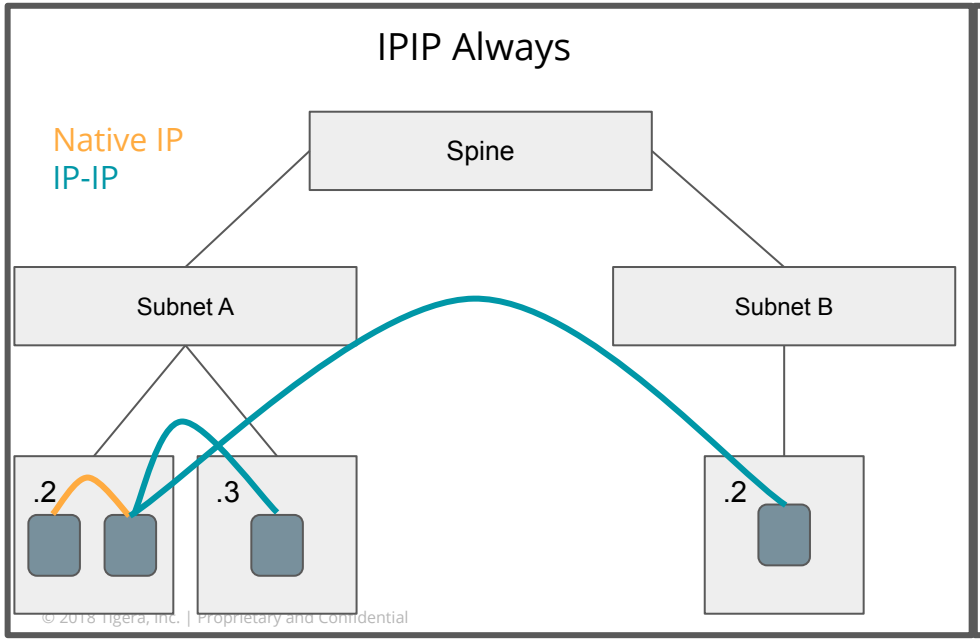
Overlay vs. Flat Network

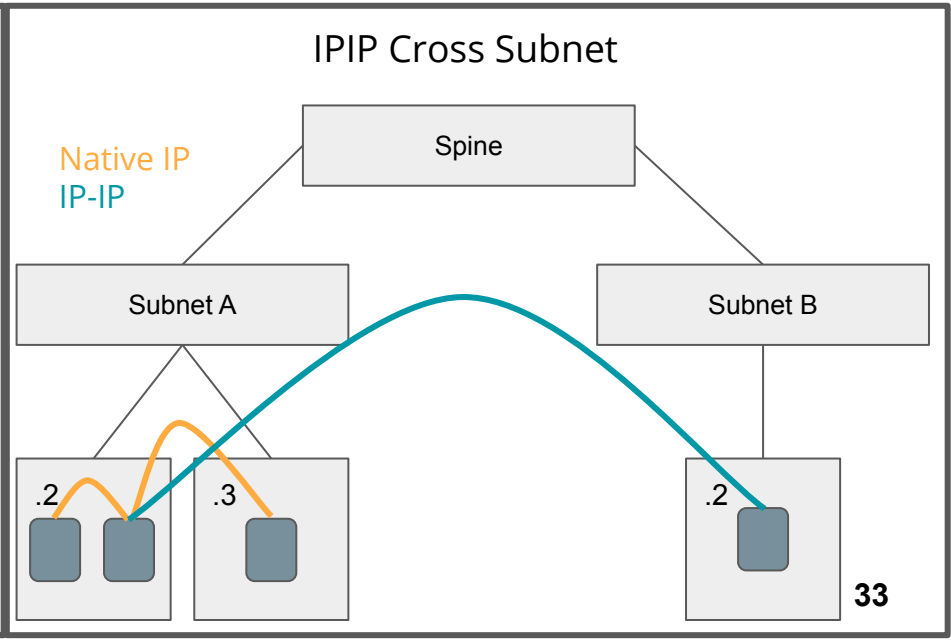
Calico Networking Options: To Overlay or Not To Overlay

- How much control you have over your networking infrastructure outside of Kubernetes will determine which of the various overlay options you should chose
  - If you have full control - e.g. private data center your pods can choose to not be overlayed - BGP enabled routers will get the packet to the right node
  - If you don't have full control - e.g. any of the public clouds, you can opt for IP in IP networking or a traditional VXLAN
    - IPIP is preferred since it's slightly more efficient with smaller headers than VXLAN
    - VXLAN is required in Azure cloud environments

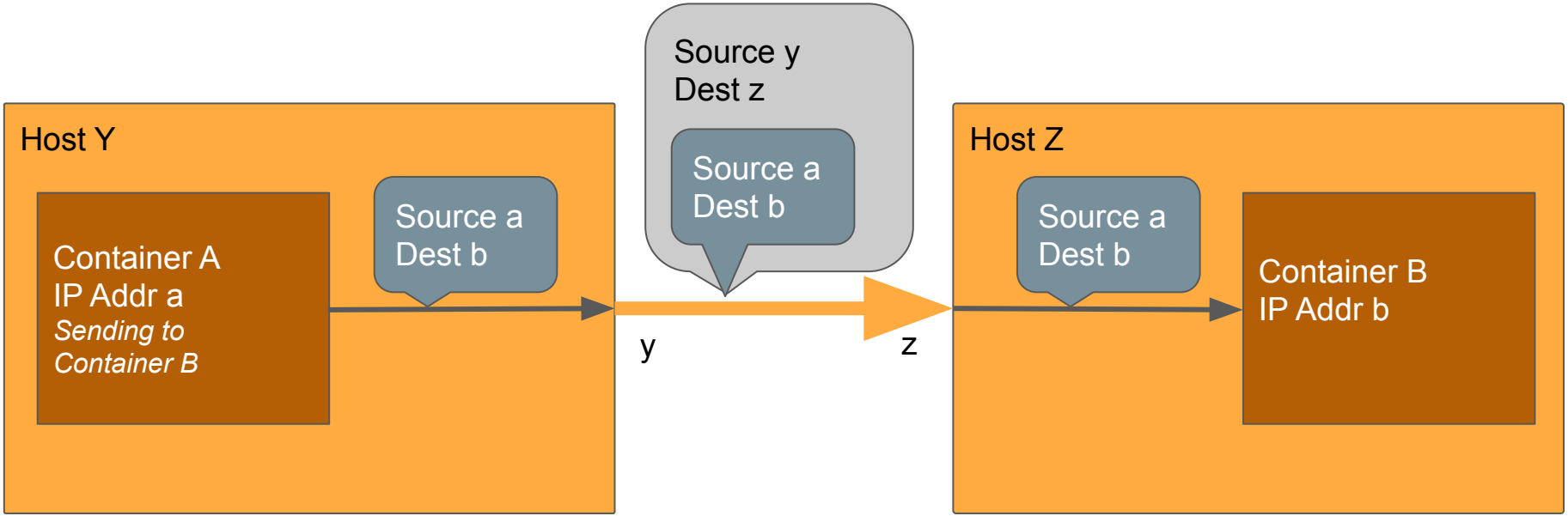
Calico Networking Options: IP in IP Options

- Calico can operate in two modes with IP in IP
  - IP in IP always encapsulates all IP traffic except traffic that never leaves the node
  - IP in IP cross subnet encapsulates traffic that crosses a subnet boundary, but not traffic for within the subnet





IP-IN-IP VISUALIZED



IP Pools and the Calico CNI

TIGERA

IP Pools

- **IP Pools** can be associated with namespaces, workloads, or topology
- **Topology**: designate certain nodes with a node label in Kubernetes, then reference that label in the pool definition
- **Namespace**: add a Calico annotation to the namespace definition
- **Workload**: same as namespace – though any annotation set at the namespace level will take priority over a workspace annotation
  - Example annotation added to namespace or workload yaml:
    ```
    annotations: "cni.projectcalico.org/ipv4pools": "[\"dmz-pool\"]"
    ```
  - Note all the "\" escapes are necessary

IP Pool Definition - Topology

dmz_pool.yaml

```
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
   name: dmz-pool
spec:
   cidr: 10.0.1.0/24
   ipipMode: Always
    natOutgoing: true
    nodeSelector: type: "dmz-node"
```

This will define an IP Pool named dmz-pool.

When running multiple subnets for pods it's usually a good idea to set natOutgoing to true and IPIP mode to always.

nodeSelector is one way to specify IP pool use per topology

IP Pools – Namespaces and workloads

dmz_ns.yaml

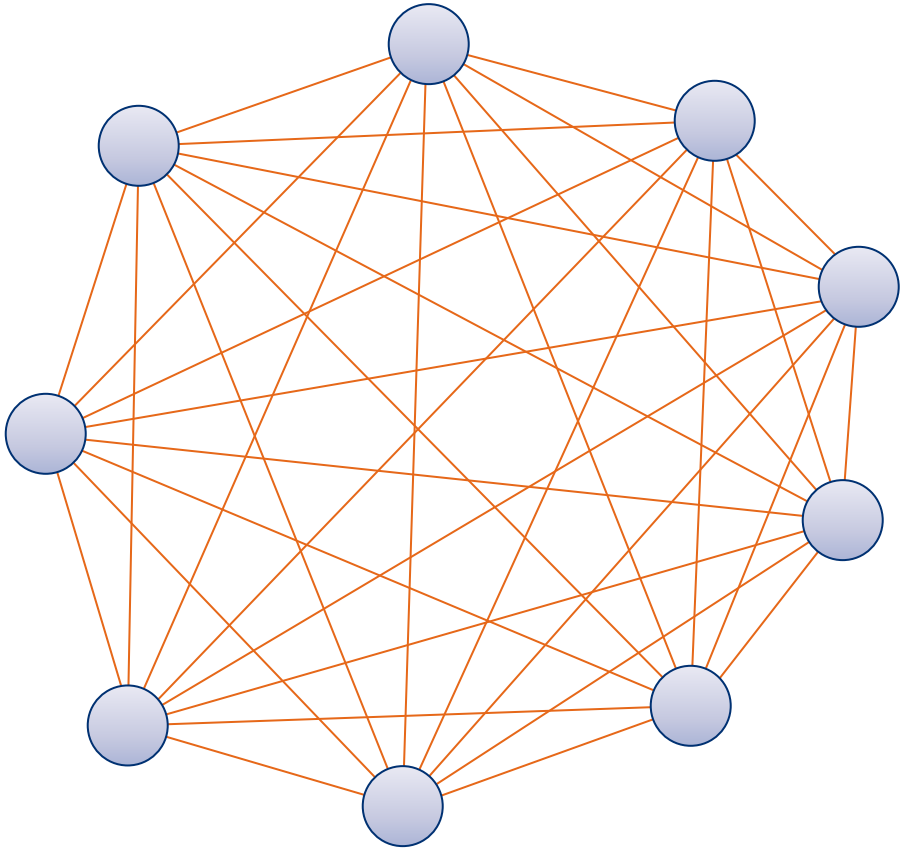
```
apiVersion: v1
kind: Namespace
metadata:
  name: dmz
  labels:
    location: dmz
  annotations:
 "cni.projectcalico.org/ipv4pools": "[\"dmz-pool\"]"
```

To assign an IP Pool to a namespace add and annotation to the namespace definition referencing the IP Pools name.

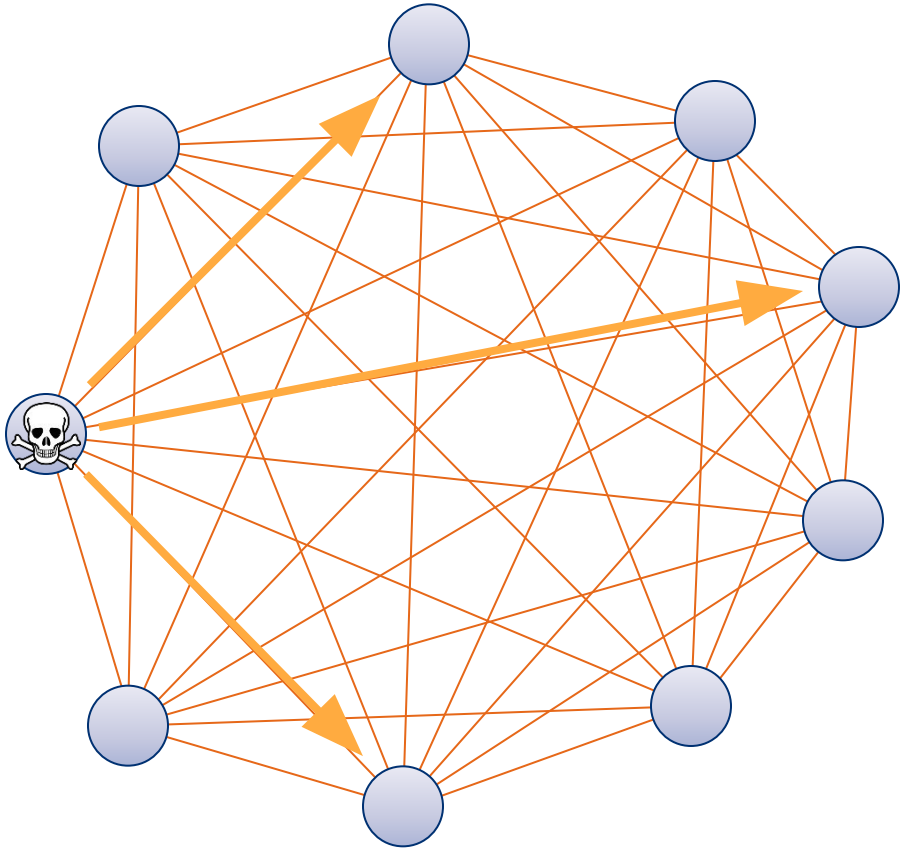
Declarative Network Policy

MOTIVATION FOR NETWORK POLICY

Consider a Kubernetes cluster of n services. Of the n^2 possible connections between all your services, only a tiny, tiny fraction of them are actually useful for your application.

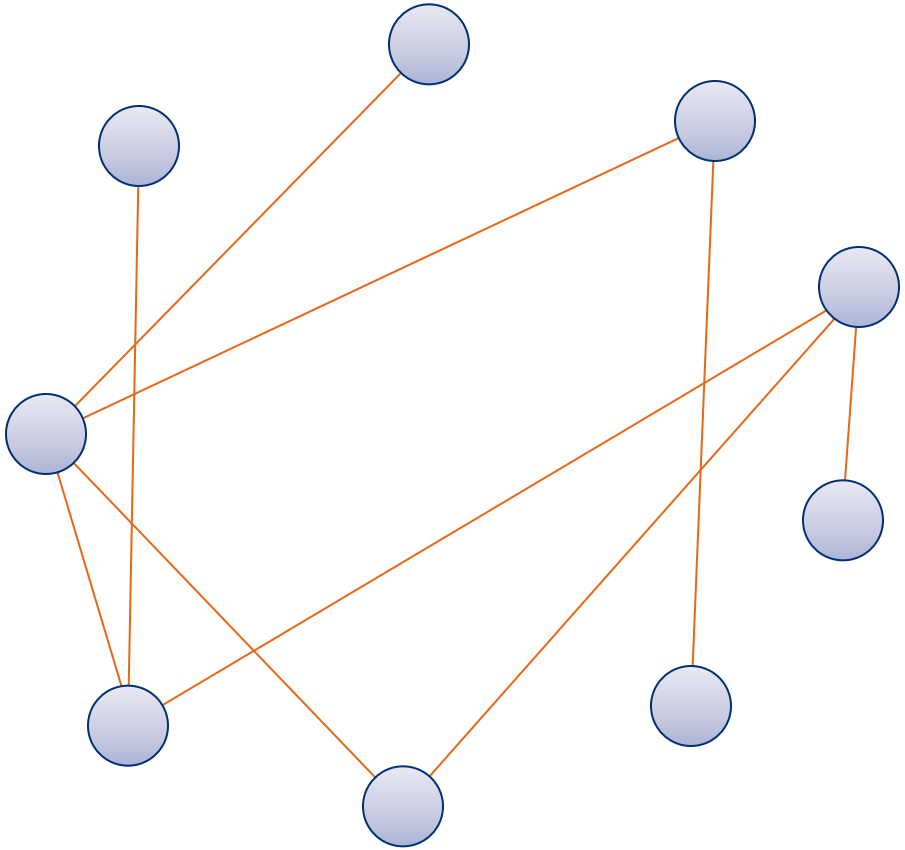


The rest are only useful for an attacker.

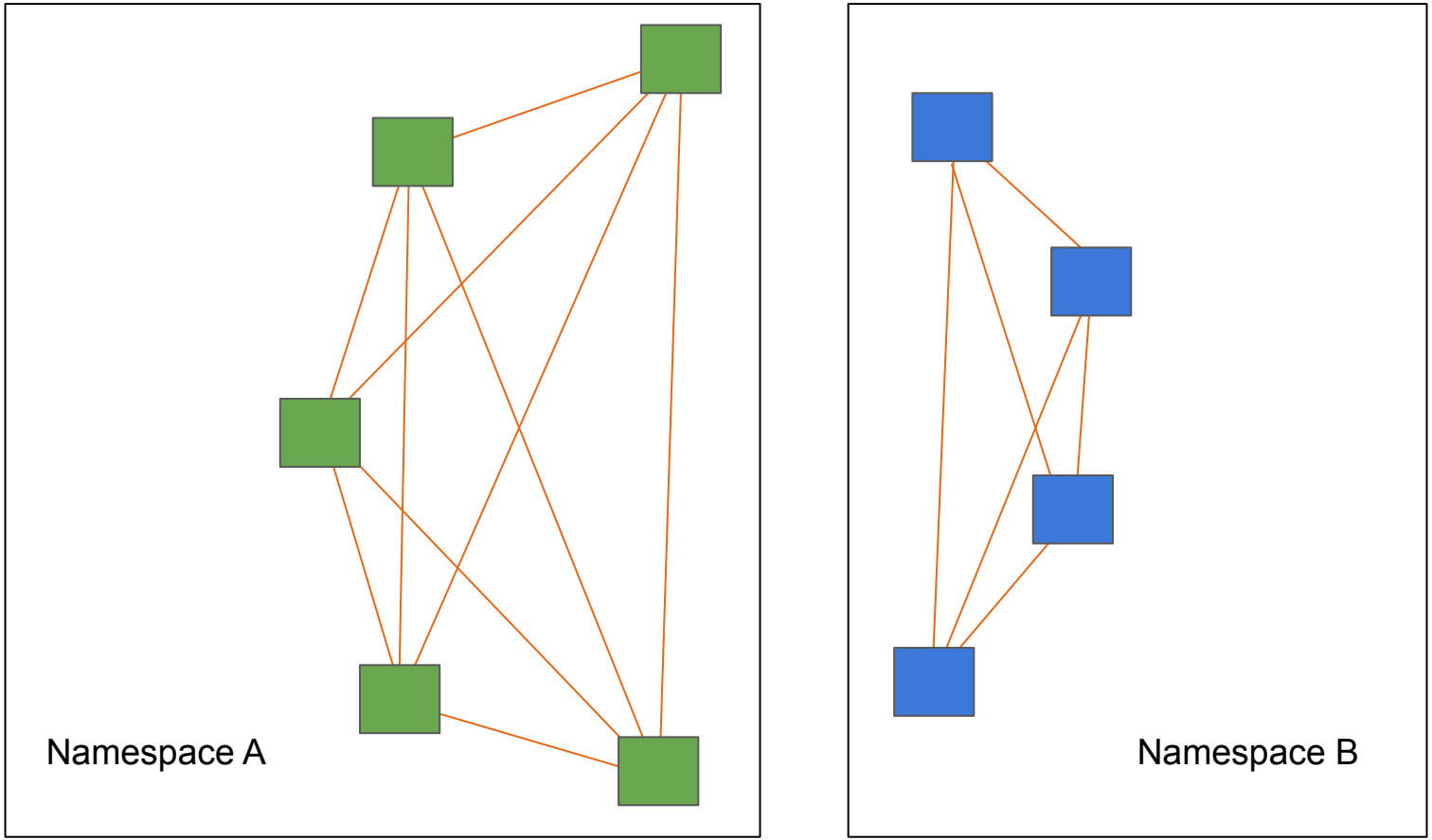


We know that our frontend load balancers don't connect directly to the backend database. dev containers do not connect to prod, and so on.

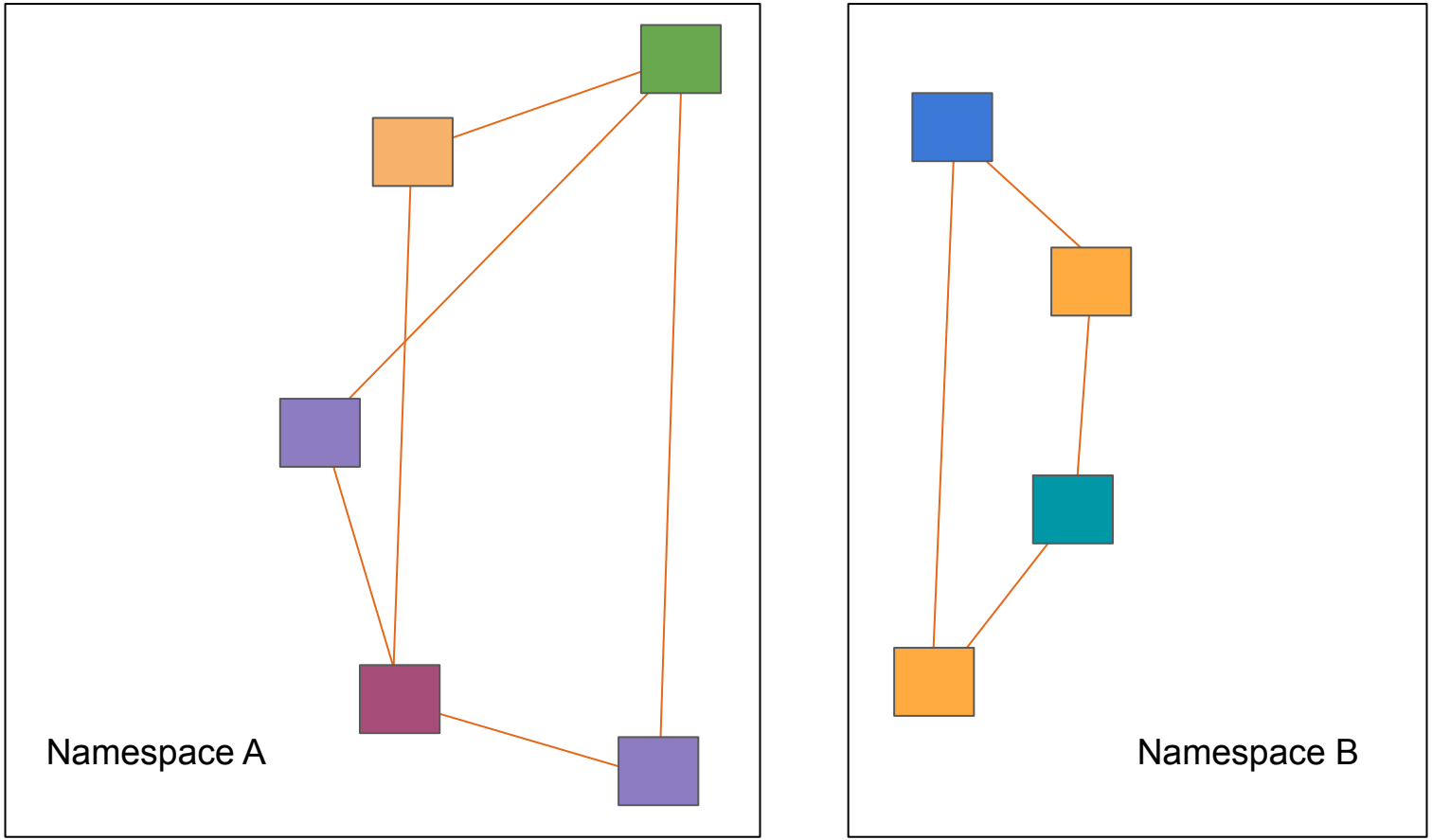
Each connection path that we can eliminate will reduce complexity and improve security.



Namespace Isolation



Custom Isolation



EXAMPLE NETWORK POLICY USE CASES

- Stage separation - isolate dev / test / prod instances
- Translation of traditional firewall rules - e.g. 3-tier
- “Tenant” separation - e.g. typically use namespaces for different teams within a company - but without network policy, they are not network isolated
- Fine-grained firewalls - reduce attack surface within microservice-based applications
- Compliance - e.g. PCI, HIPAA
- Any combination of the above

KEY KUBERNETES CONCEPT: LABELS

- Labels are **key/value pairs** that are attached to objects, such as pods.
- Labels are intended to be used to specify identifying attributes of objects that are **meaningful and relevant to users**, but do not directly imply semantics to the core system.
- Labels can be used to **organize and to select subsets of objects**.
- Labels can be attached to objects at **creation time** and subsequently added and **modified at any time**.
- Each Key must be unique for a given object.

```
"labels": {
  "key1" : "value1",
  "key2" : "value2"
}
```

Source: https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

Key Kubernetes Concept: Label Selectors

Via a *label selector*, the client/user can identify a set of objects. The label selector is the **core grouping primitive in Kubernetes.**

- Equality-based (==, !=)
- Set membership (in, notin, exists)

```
environment = production
tier != frontend
```

```
environment in (production, qa)
tier notin (frontend, backend)
partition
!partition
```


TIGERA

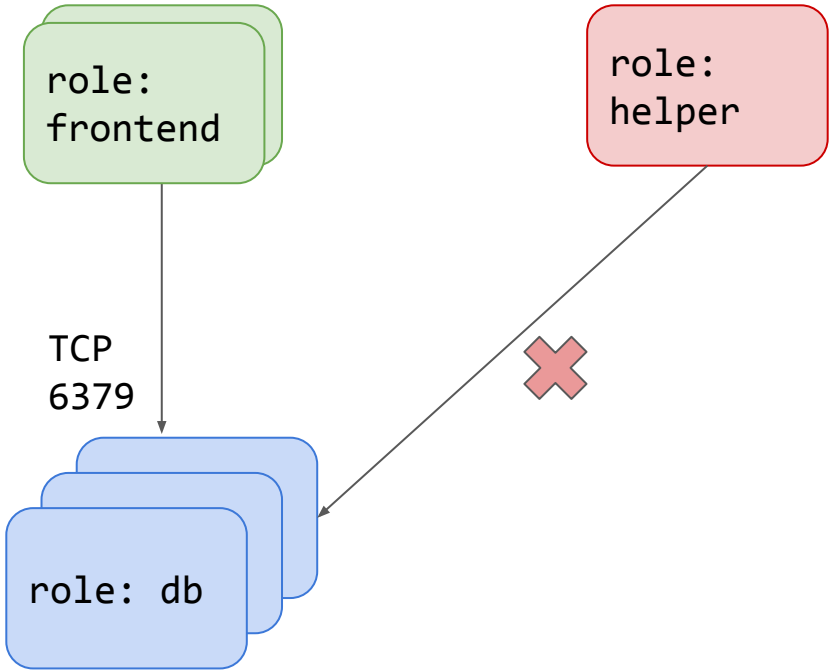
Kubernetes NetworkPolicy Resource

- Specifies how groups of pods are allowed to communicate with each other and other network endpoints using:
  - Pod label selector
  - Namespace label selector
  - Protocol + Ports
  - More to come in the future
- Pods selected by a NetworkPolicy:
  - Are isolated by default
  - Are allowed incoming traffic if that traffic matches a NetworkPolicy ingress rule.
- Requires a controller to implement the API

https://kubernetes.io/docs/concepts/services-networking/network-policies/

Kubernetes Network Policy Example

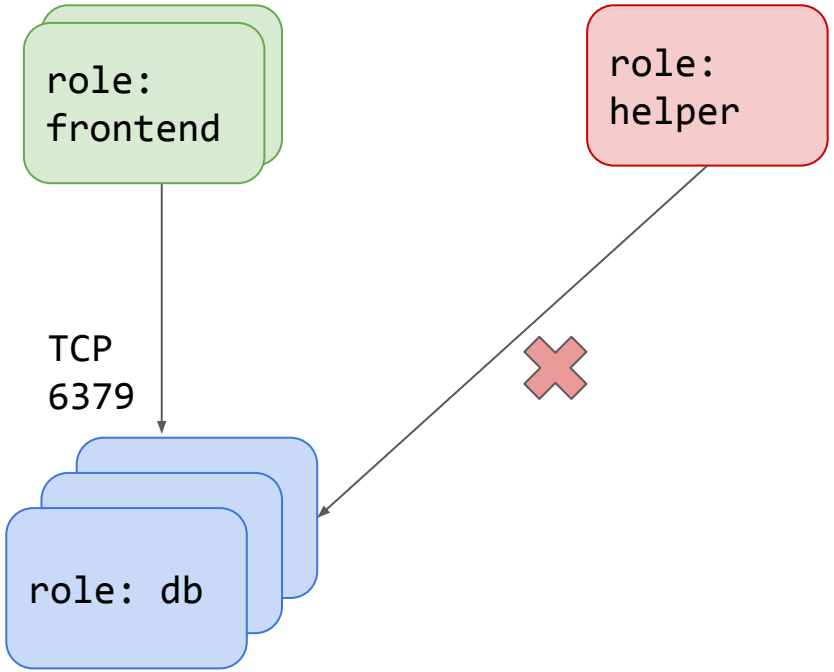
```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-network-policy
  namespace: my-namespace
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
```



TIGERA

Kubernetes Network Policy Example

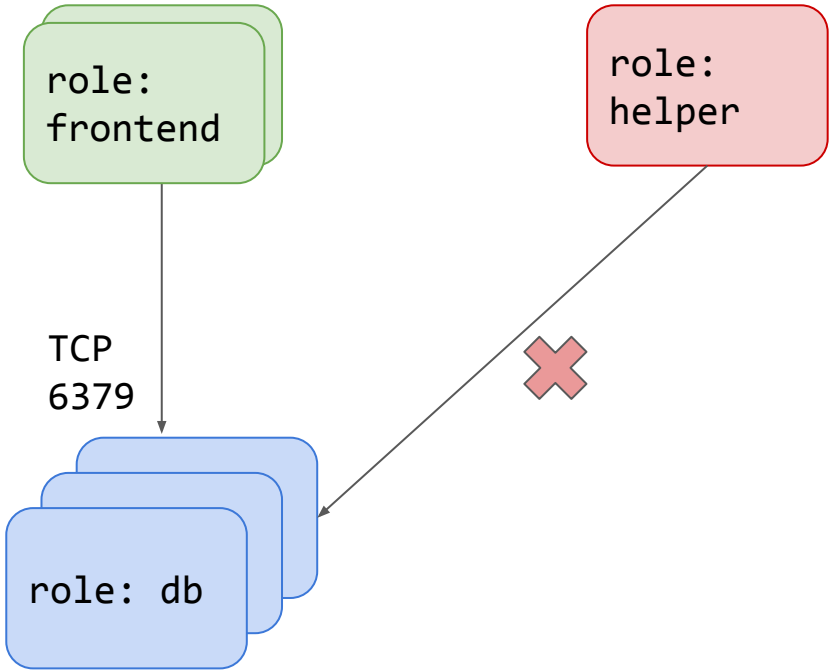
```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-network-policy
  namespace: my-namespace
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
```



TIGERA

Kubernetes Network Policy Example

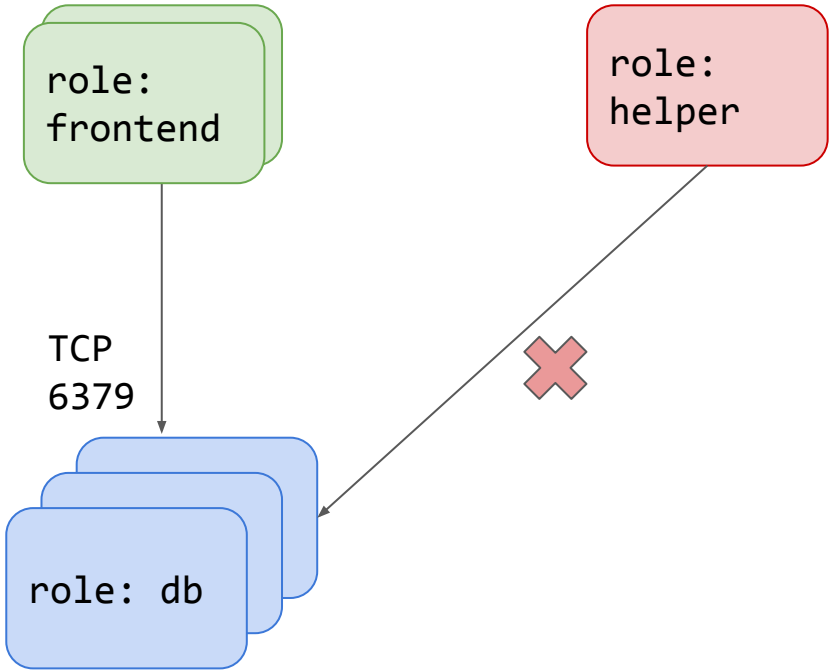
```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-network-policy
  namespace: my-namespace
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
```



TIGERA

Kubernetes Network Policy Example

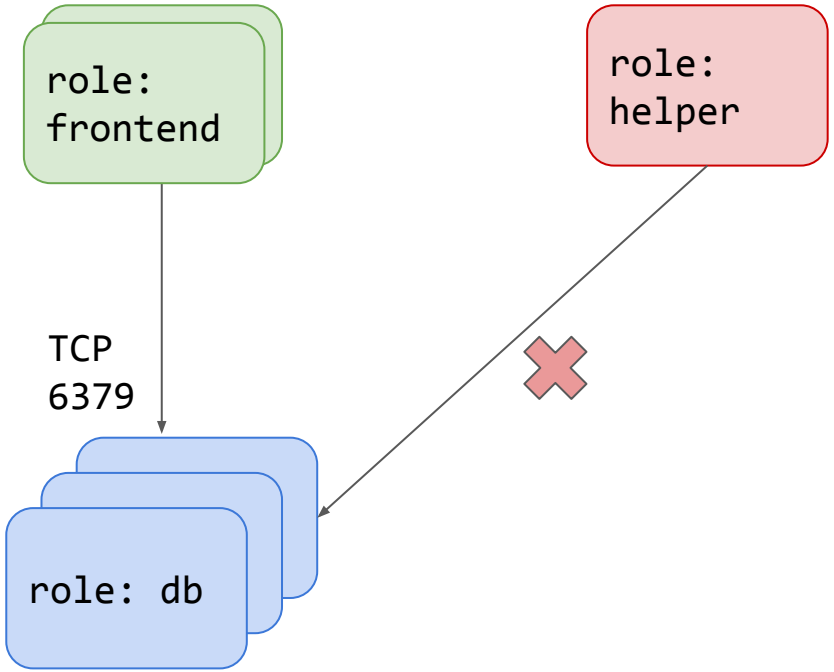
```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-network-policy
  namespace: my-namespace
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
```



TIGERA

Kubernetes Network Policy Example

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-network-policy
  namespace: my-namespace
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
```



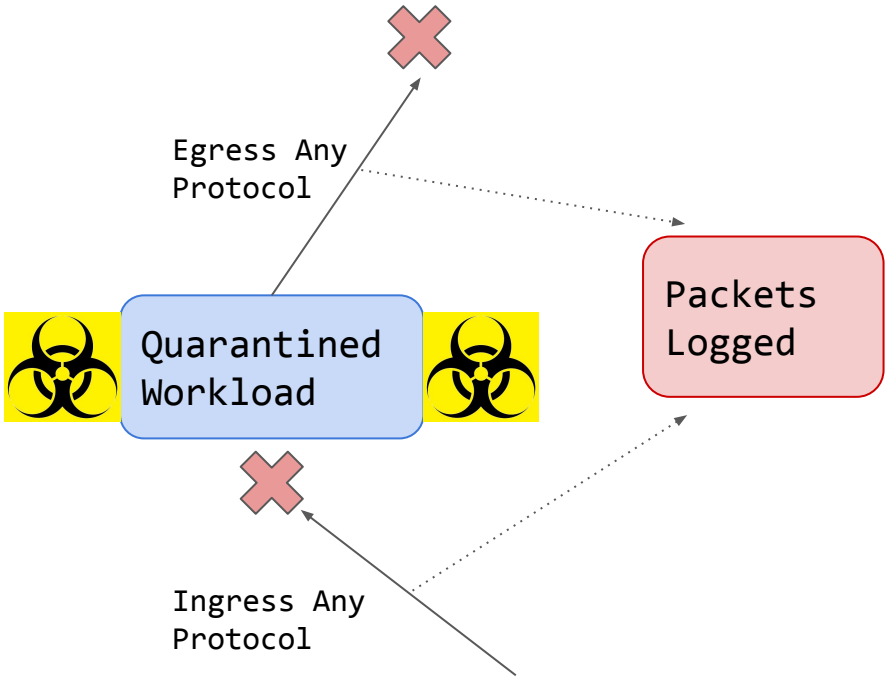
TIGERA

Advanced use cases beyond Kubernetes Network Policy

- **Global** network policies (rather than being limited to apply to pods in a single namespace)
- Ability to specify policy rules that **reference labels of pods in other namespaces** (rather than only being allowed to allow/deny all pods from another namespace)
- Ability to specify policy rules that **reference service account labels**
- **Richer label expressions** and **traffic specification** (e.g. port ranges, protocols other than tcp/udp, negative matching on protocol type, ICMP type)
- **Deny** rules
- **Policy order**/prioritization (which becomes necessary when you have mix of deny and allow rules)
- **Network sets** - ability to apply labels to a set of IP addresses, so they can be selected by label selector expressions
- Support for **non-Kubernetes nodes** (e.g. standalone hosts) within the policy domain,
- Policy options specific to **host endpoints** (apply-on-forward, pre-DNAT, doNotTrack)
- If you need these capabilities, use Calico directly instead of via Kubernetes Network Policy API
- https://docs.projectcalico.org/v3.7/security/calico-network-policy

Calico Network Policy Example

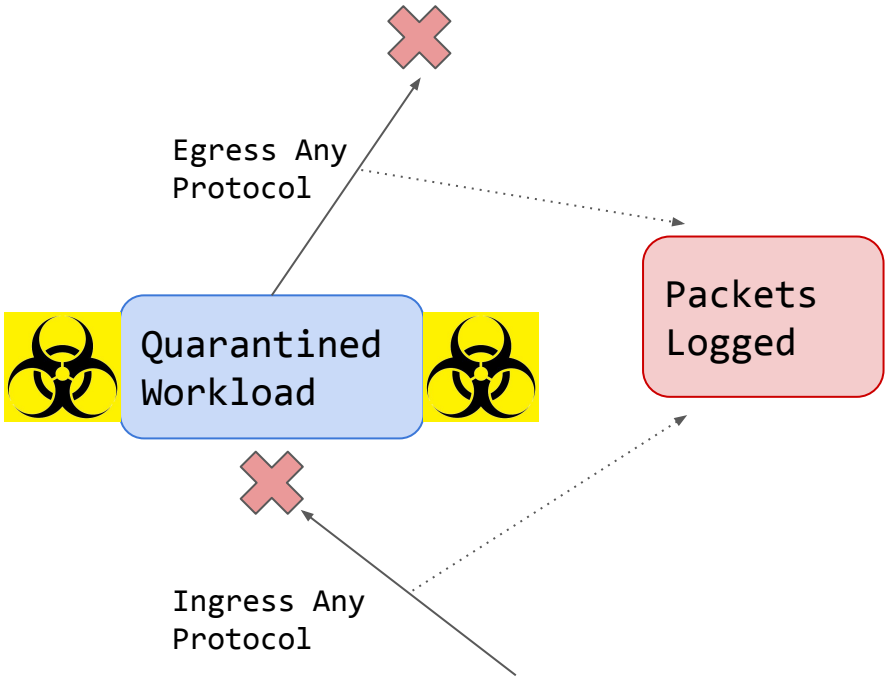
```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: security-operations.quarantine
spec:
  order: 200
  selector: quarantine == "true"
  ingress:
    - action: Log
      source: {}
      destination: {}
    - action: Deny
      source: {}
      destination: {}
  egress:
    - action: Log
      source: {}
      destination: {}
    - action: Deny
      source: {}
      destination: {}
    types:
    - Ingress
    - Egress
```



TIGERA

Calico Network Policy Example

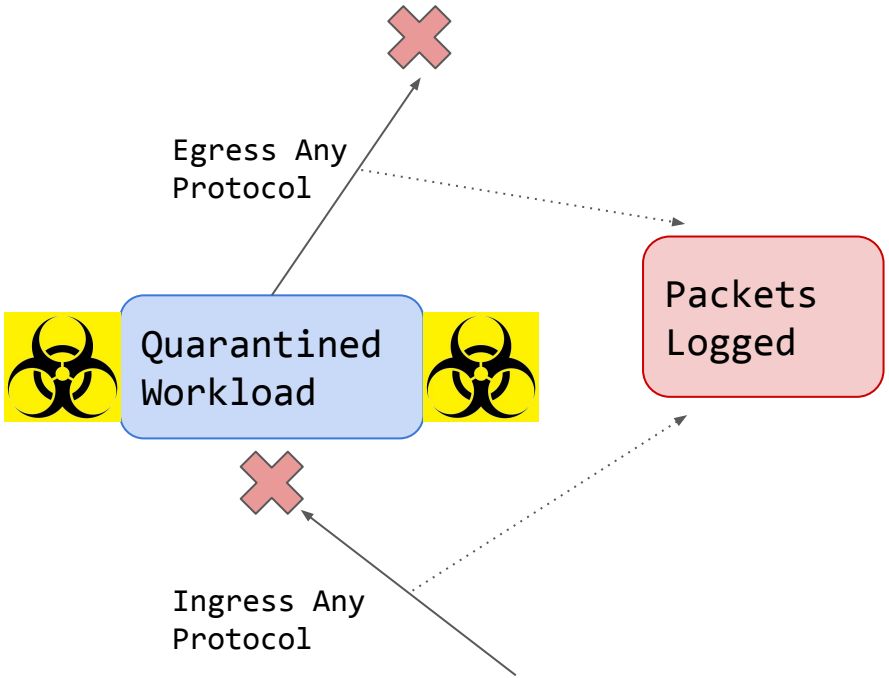
```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: security-operations.quarantine
spec:
  order: 200
  selector: quarantine == "true"
  ingress:
    - action: Log
      source: {}
      destination: {}
    - action: Deny
      source: {}
      destination: {}
  egress:
    - action: Log
      source: {}
      destination: {}
    - action: Deny
      source: {}
      destination: {}
    types:
    - Ingress
    - Egress
```



TIGERA

Calico Network Policy Example

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: security-operations.quarantine
spec:
  order: 200
  selector: quarantine == "true"
  ingress:
    - action: Log
      source: {}
      destination: {}
    - action: Deny
      source: {}
      destination: {}
  egress:
    - action: Log
      source: {}
      destination: {}
    - action: Deny
      source: {}
      destination: {}
  types:
  - Ingress
  - Egress
```



TIGERA

Calicoctl: The Calico Command Line

- Calicoctl is the command line utility for advanced Calico configuration
- Download: https://github.com/projectcalico/calicoctl/releases
- Reference: https://docs.projectcalico.org/latest/reference/calicoctl/

FELIX TIP: Calicoctl is available for Linux, MacOS and Windows. Download it for your favorite desktop operating system!

Thank you!