Zone Based Security with Calico

Kubernetes in a Zone Based Architecture World

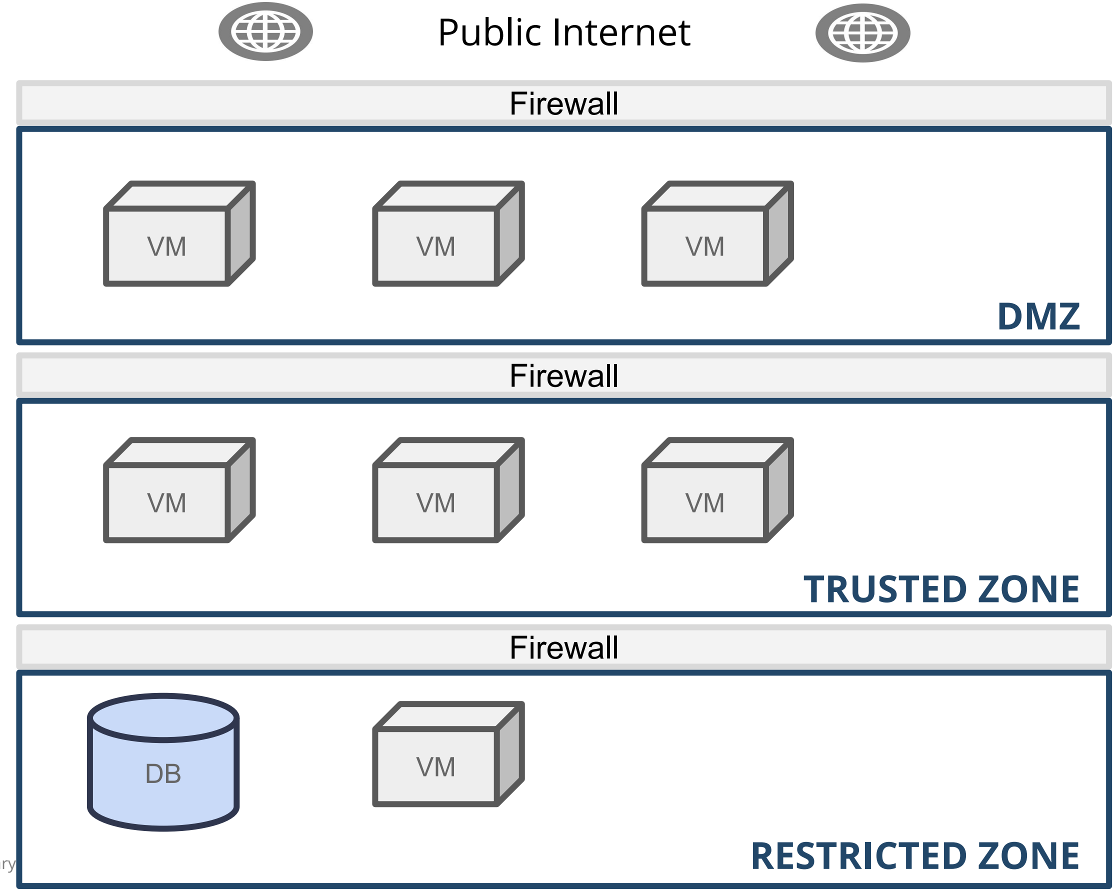
TIGERA

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!

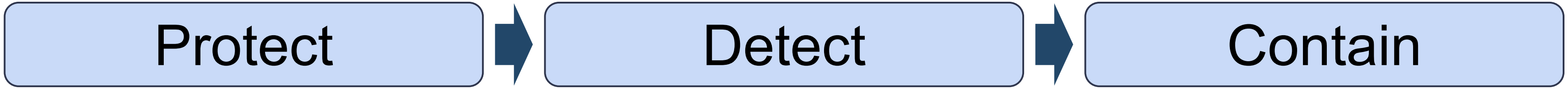
TIGERA

Zone based architecture



Zone Based Architecture Overview

Security Zones are put in place to



Protect

Establish boundaries and manage access to applications & data

Manage inter-zone communication

Enforce policies and regulations requiring network isolation

Detect

Monitor inter-zone communication

Visibility of traffic, users, and assets

Logging and event correlation

Prevent inter-zone data leakage

Contain

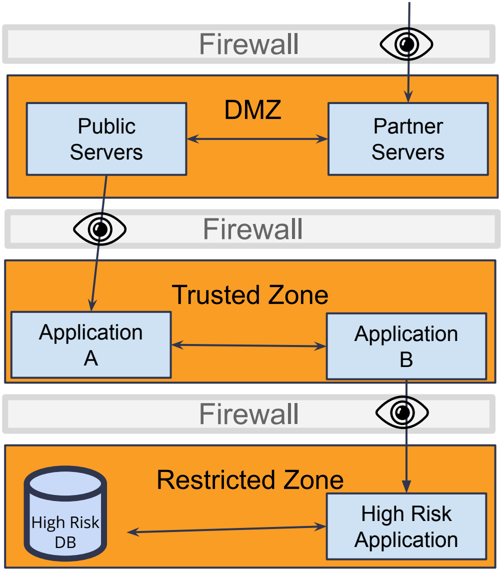
Control communications inbound/outbound

Only allow legitimate inter-zone connections

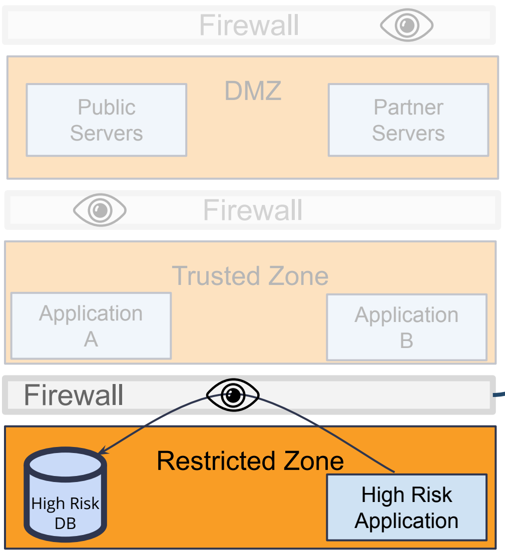
Zone Visibility

Visibility is typically between zones via Firewall.

Within zone is (mostly) trusted



When greater visibility is required, intra-zone traffic is “Hair Pinned” through firewall



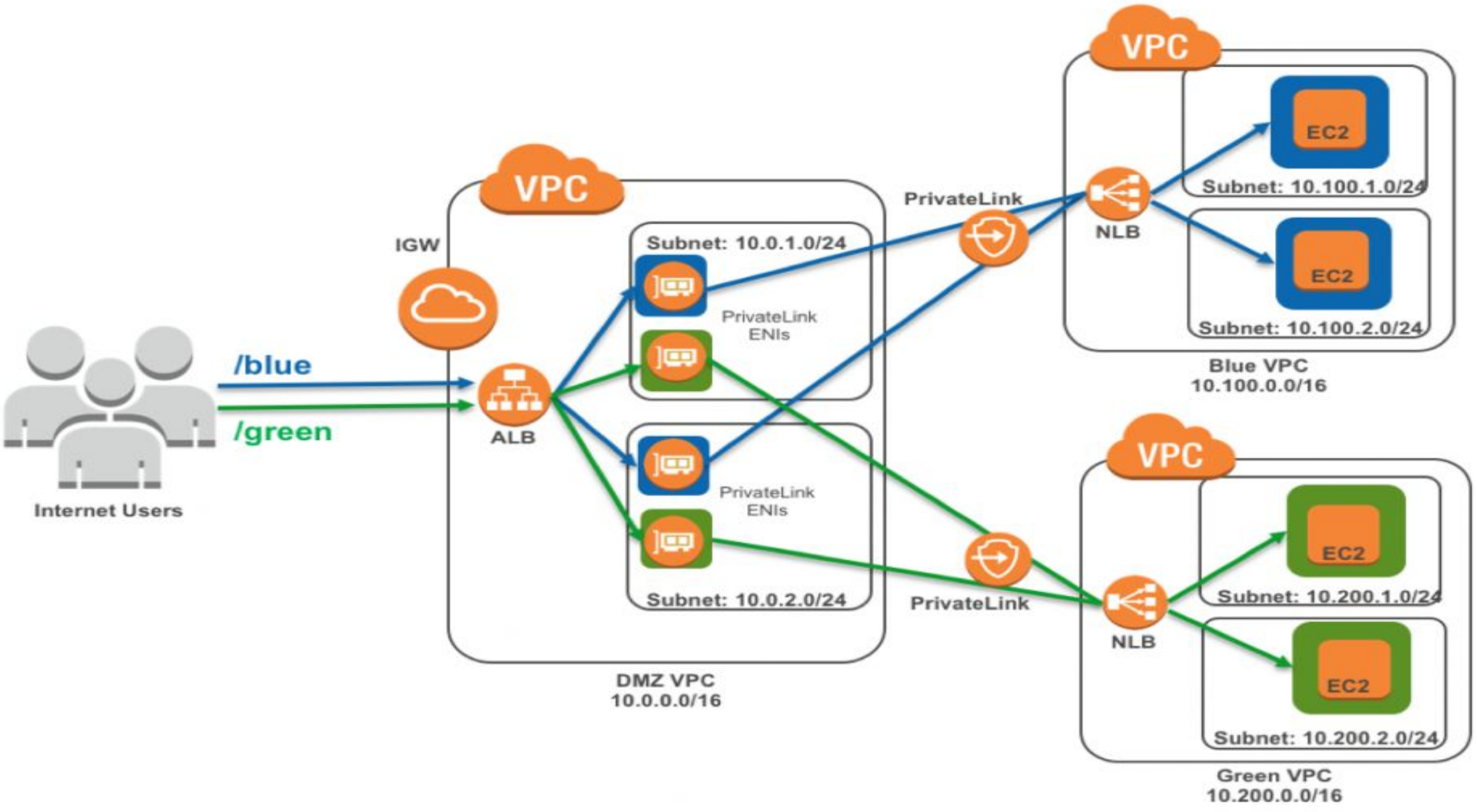
Firewalls are sized by Gbps throughput

Hair pinning becomes expensive

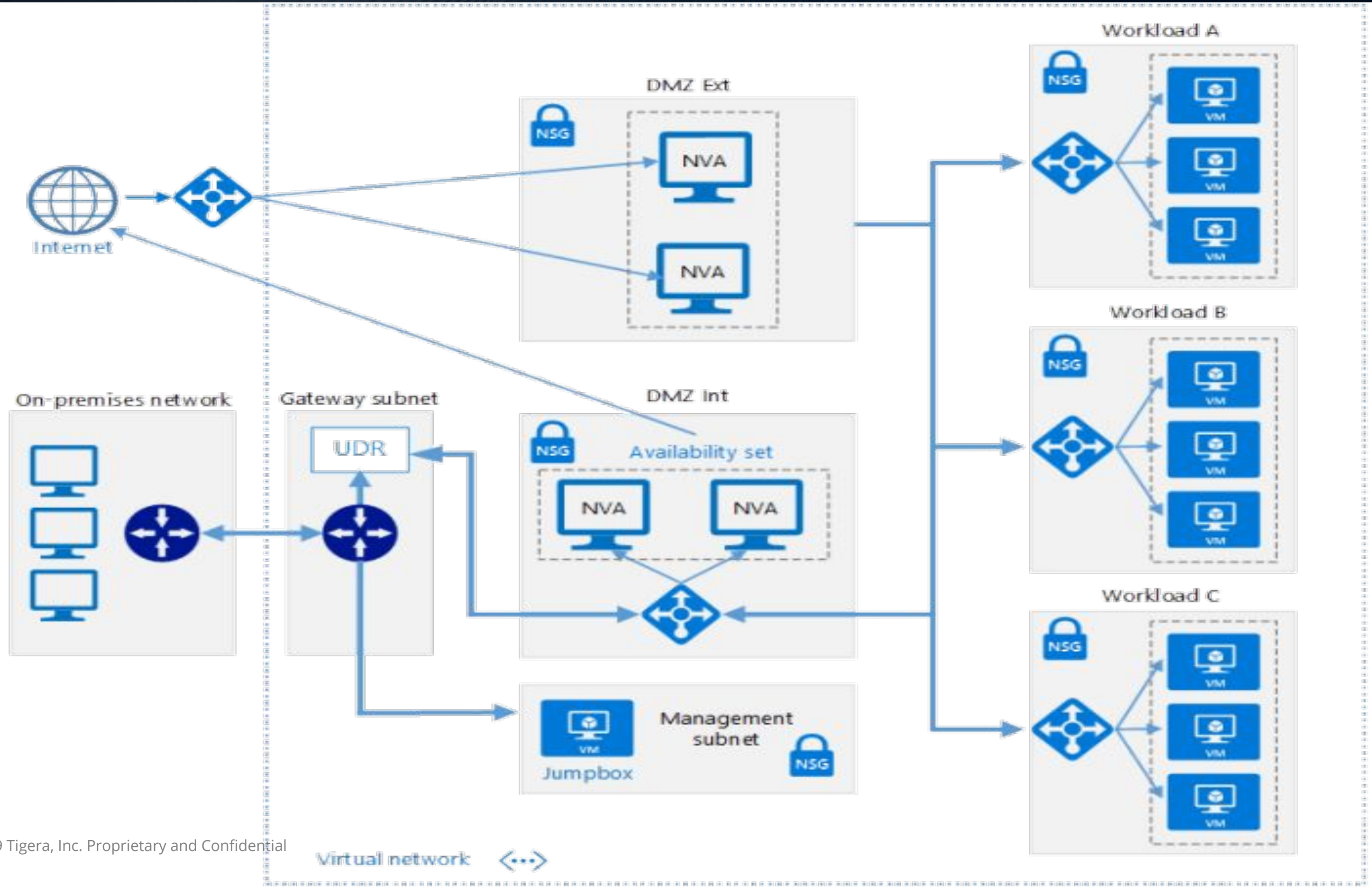
High throughput firewalls cost millions $$$

Migration to the cloud

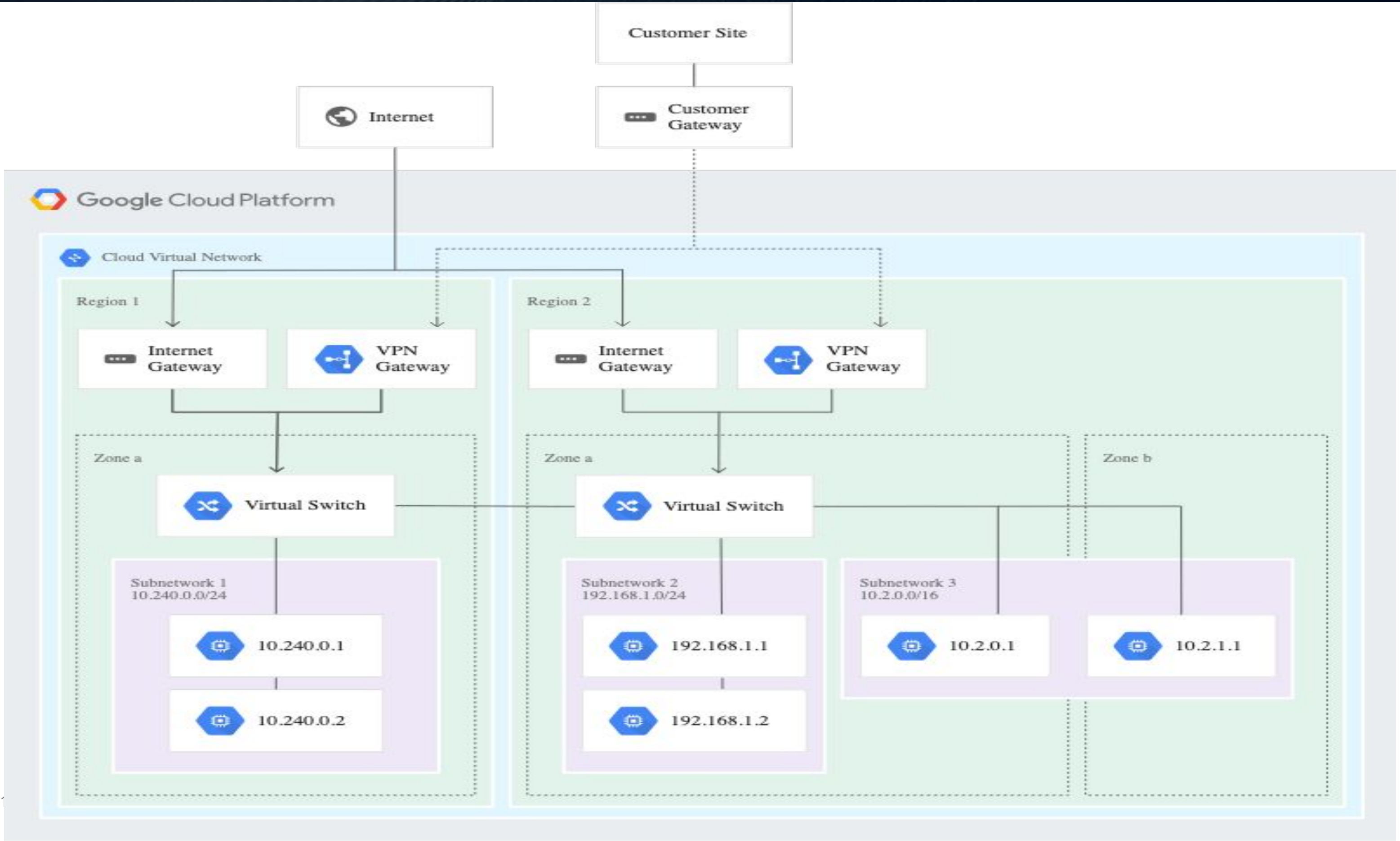
Zone based architecture - in the cloud



Zone based architecture - in the cloud - continued



Zone based architecture - in the cloud - continued

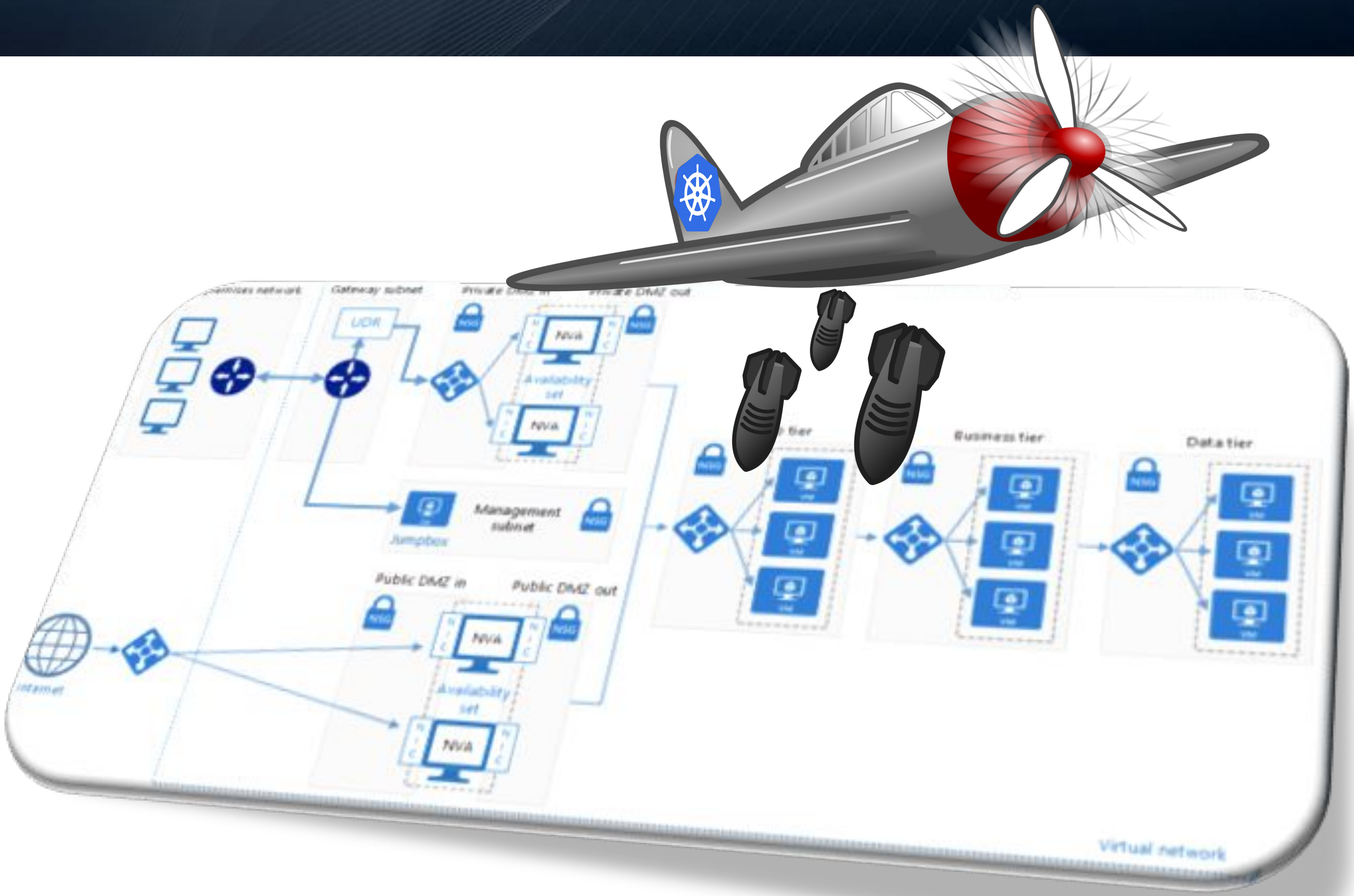


Benefits to Security Team

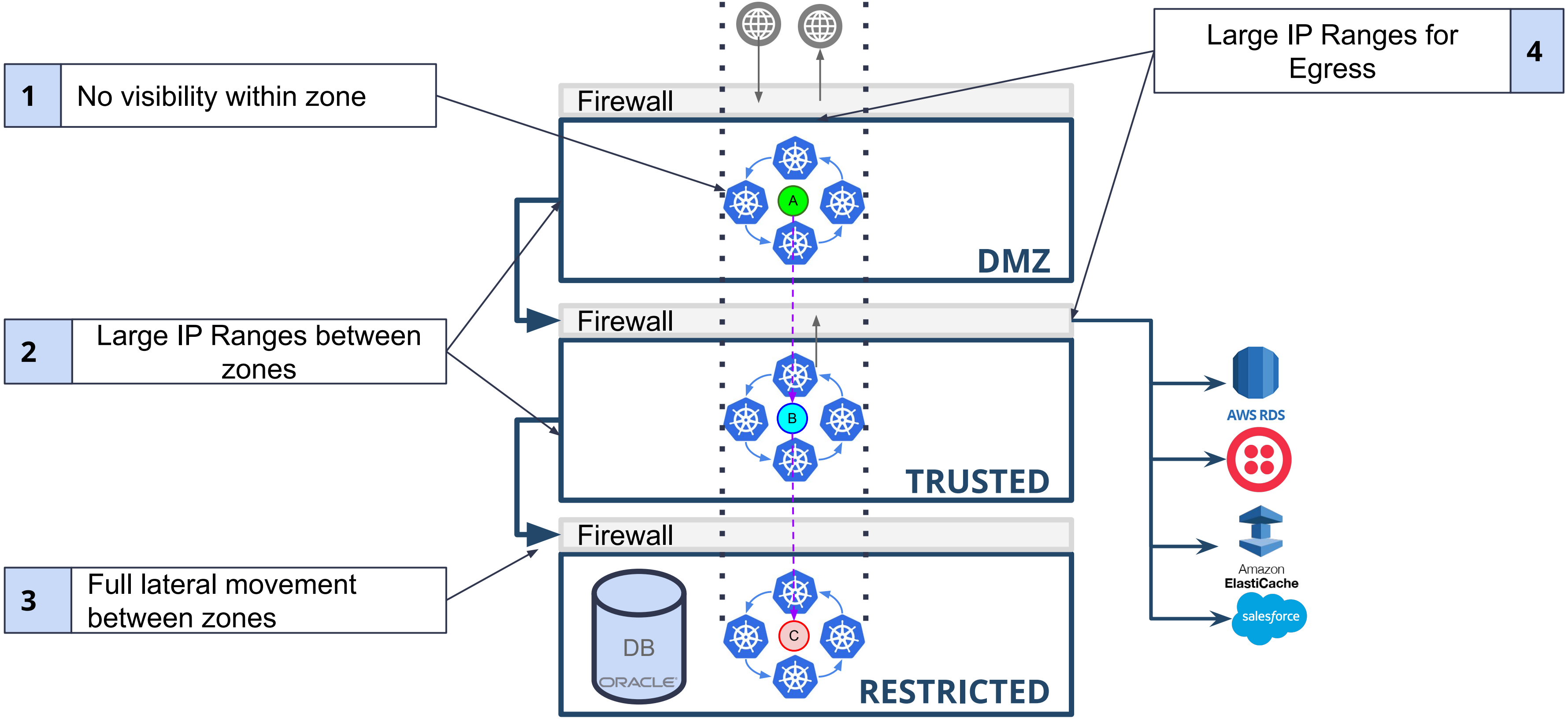
Key Benefits of network segmentation and zones

- Limit exposure & breadth of impact
- Group assets - prioritize focus & spend to high-risk areas
- Visibility and the ability to detect suspicious activity
  - Anomaly detection at each zone firewall
- Containment of an attack to a single zone
- Expedite eradication and recovery - smaller blast radius

Guess What?


premises network
Gateway subnet
Private DMZ in
Private DMZ out
UDR
NSG
NVA
Availability set
NVA
Management subnet
Jumpbox
tier
Business tier
Data tier
VM
Public DMZ in
Public DMZ out
NVA
Availability set
NVA
Internet
Virtual network

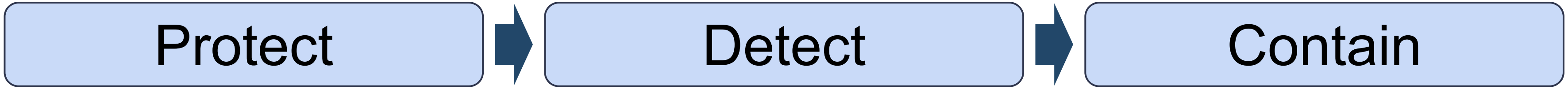
Firewalls for Kubernetes: Key Challenges



1
No visibility within zone
2
Large IP Ranges between zones
3
Full lateral movement between zones
Large IP Ranges for Egress
4
Firewall
DMZ
A
Firewall
B
TRUSTED
Firewall
DB
ORACLE
C
RESTRICTED
AWS RDS
Amazon ElastiCache
salesforce

Zone Based Architecture Overview

Security Zones are put in place to



Establish boundaries and manage access to applications & data

Manage inter-zone communication

Enforce policies and regulations requiring network isolation

Monitor inter-zone communication

Visibility of traffic, users, and assets

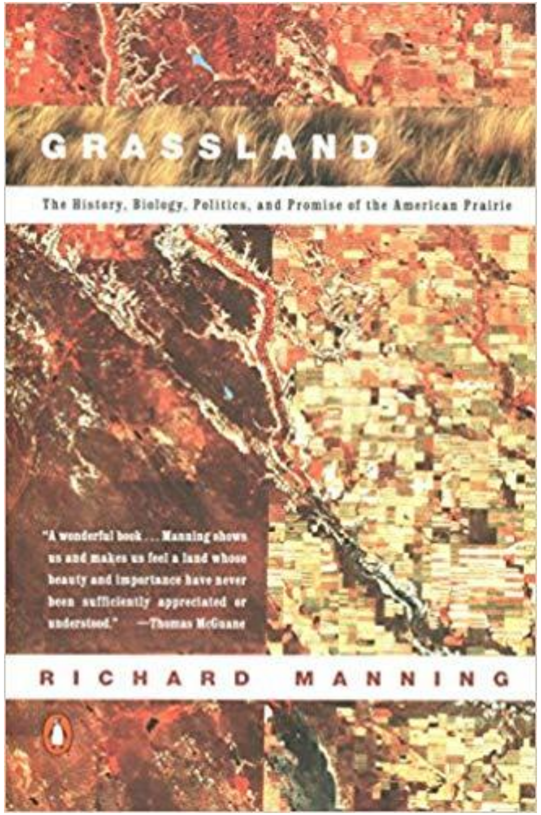
Logging and event correlation

Prevent inter-zone data leakage

Control communications inbound/outbound

Only allow legitimate inter-zone connections

A better approach?



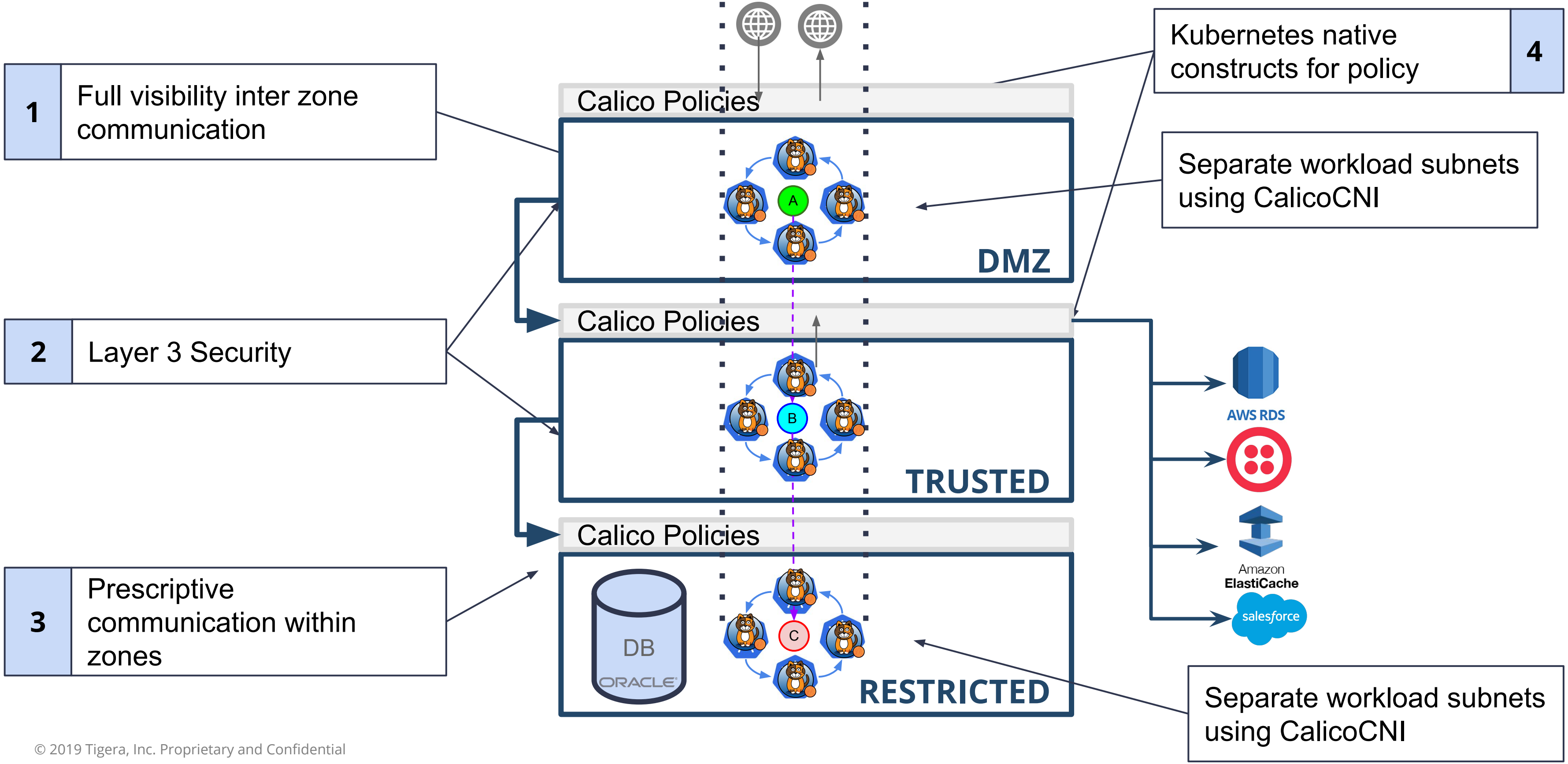
GRASSLAND

The History, Biology, Politics, and Promise of the American Prairie

"A wonderful book . . . Manning shows us and makes us feel a land whose beauty and importance have never been sufficiently appreciated or understood." —Thomas McGuane

RICHARD MANNING

Calico to the rescue!



1 Full visibility inter zone communication
2 Layer 3 Security
3 Prescriptive communication within zones
Kubernetes native constructs for policy 4
Separate workload subnets using CalicoCNI
Calico Policies
A
DMZ
Calico Policies
B
TRUSTED
Calico Policies
DB
ORACLE
C
RESTRICTED
Separate workload subnets using CalicoCNI
AWS RDS
Amazon ElastiCache
salesforce

Using the CalicoCNI to Build Zones

Where to run the Calico CNI?

Upstream K8s - bare metal or cloud

OPENSHIFT

docker

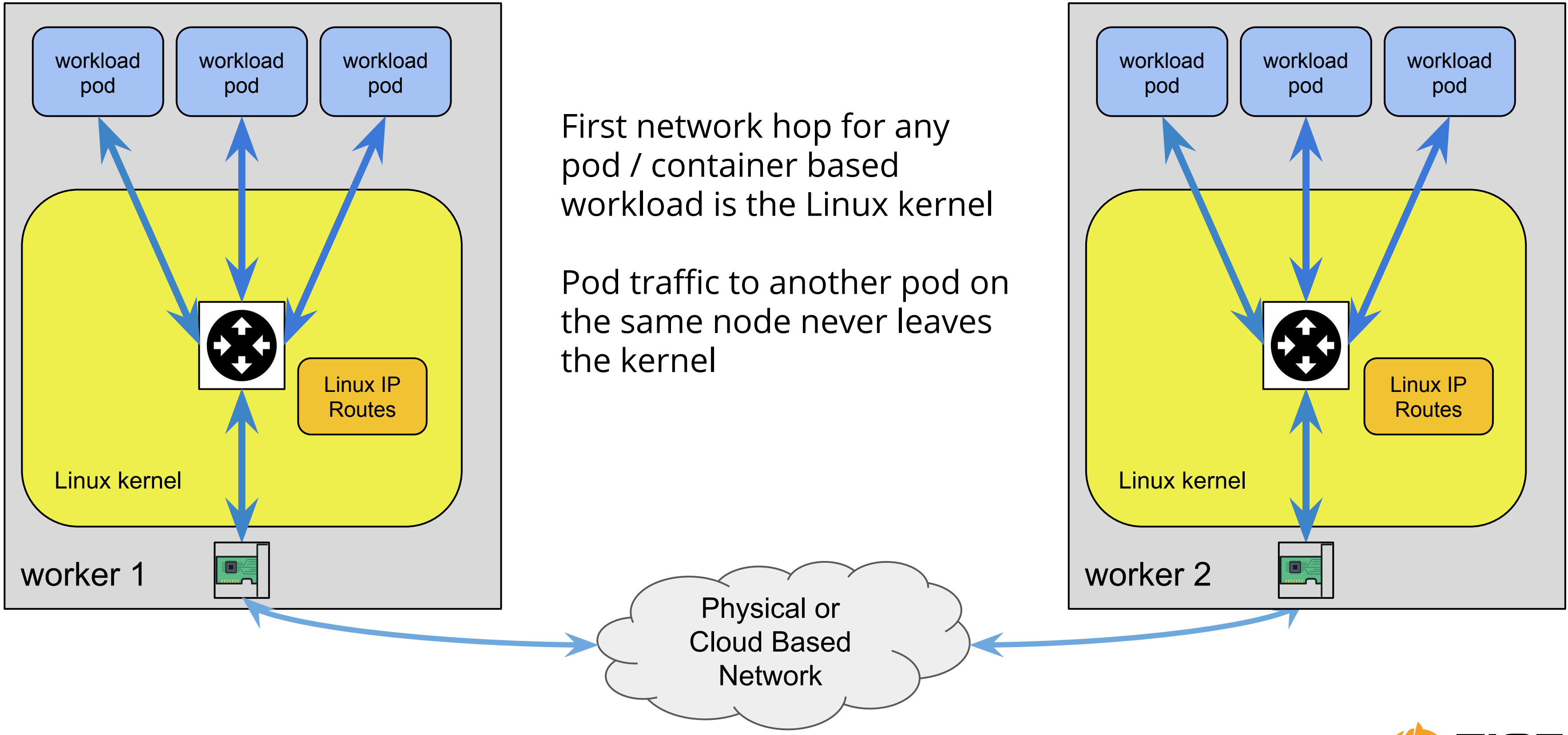
D2 IQ

The artist formerly known as Mesosphere

RANCHER®

Anthos

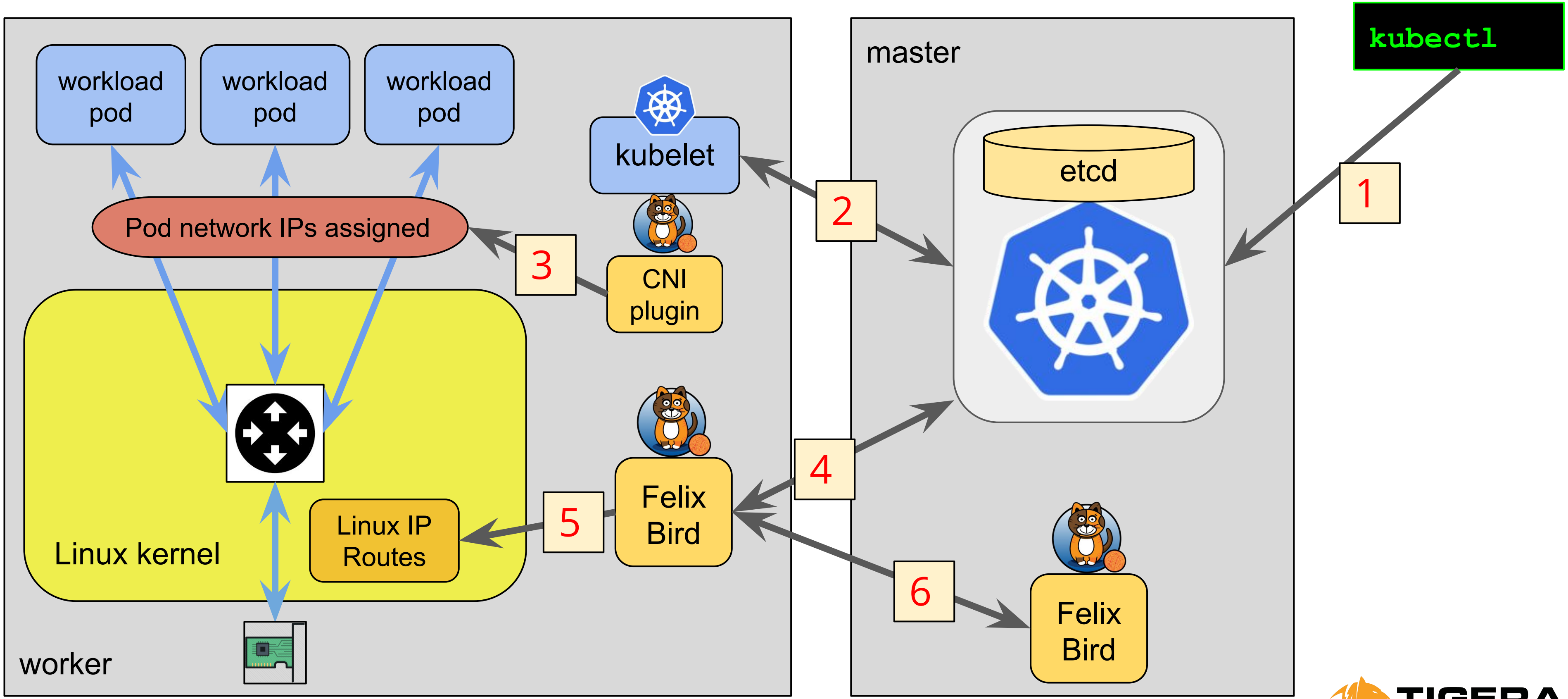
Intra Pod and Intra Node Networking



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

High Level Calico CNI Architecture



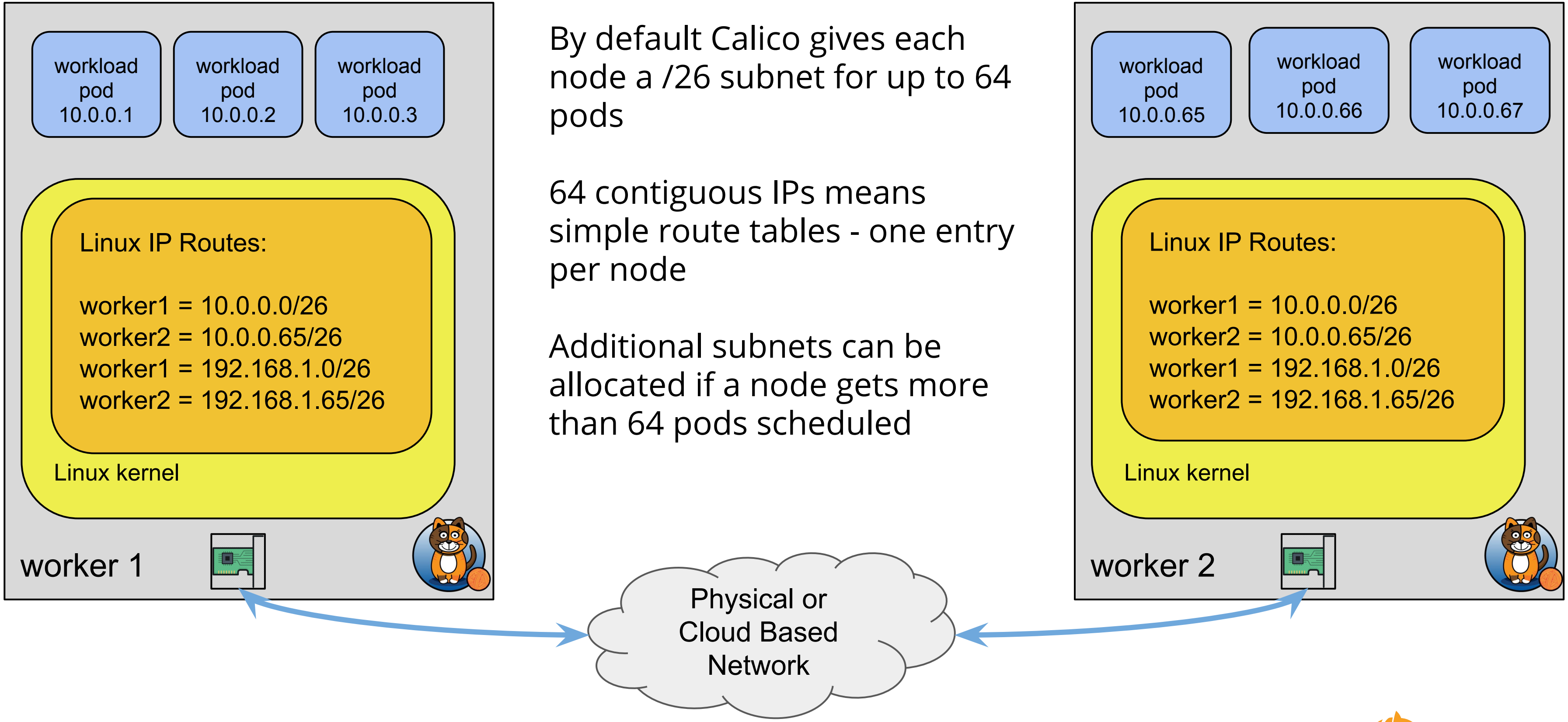
TIGERA

Calico Gives Each Node a Subnet

By default Calico gives each node a /26 subnet for up to 64 pods

64 contiguous IPs means simple route tables - one entry per node

Additional subnets can be allocated if a node gets more than 64 pods scheduled



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 Pools for Topology - Types of Topology

Physical location: Kubernetes clusters can be segregated by racks or even datacenter “rooms”

Multi-tenancy: Designate certain nodes as “staging nodes” and other as “dev nodes” OR “customer A nodes” and “customer B nodes”

Hardware type: Nodes with fast disk for persistent data workloads or nodes with GPUs for GPU-dependant workloads

Cloud-based topology: High availability K8s clusters often are spread across “availability zones” within cloud regions

Building a Pool Around Topology

Step 1: Label the nodes based on your topology use-case

```
kubectl label nodes node17.mycompany.com type=staging
```

```
kubectl label nodes node17-rack4 disk=ssd
```

```
Kubectl label nodes node13 az=useast1a
```

Step 2: Create an IP pool that uses a node selection to specify which nodes should pull IPs from this pool (see next slide for details)

Step 3: Create a deployment that specifies the node selector for your topology see: https://kubernetes.io/docs/concepts/configuration/assign-pod-node/

IP Pool Definition For Topology Use Case

```
staging_pool.yaml
```

```
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
   name: staging
spec:
   cidr: 10.0.1.0/24
   ipipMode: Always
    natOutgoing: true
      nodeSelector: type: “staging”
```

This will define an IP Pool named staging.

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

Building a Pool For Workloads or Namespaces

Step 1: Create the pool(s) required for your use case

Step 2: Add the pool annotation to the yaml / json defining the workload or namespace that will use the pool

Note: Namespace IP pool associations take precedence over workload IP Pool associations

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.

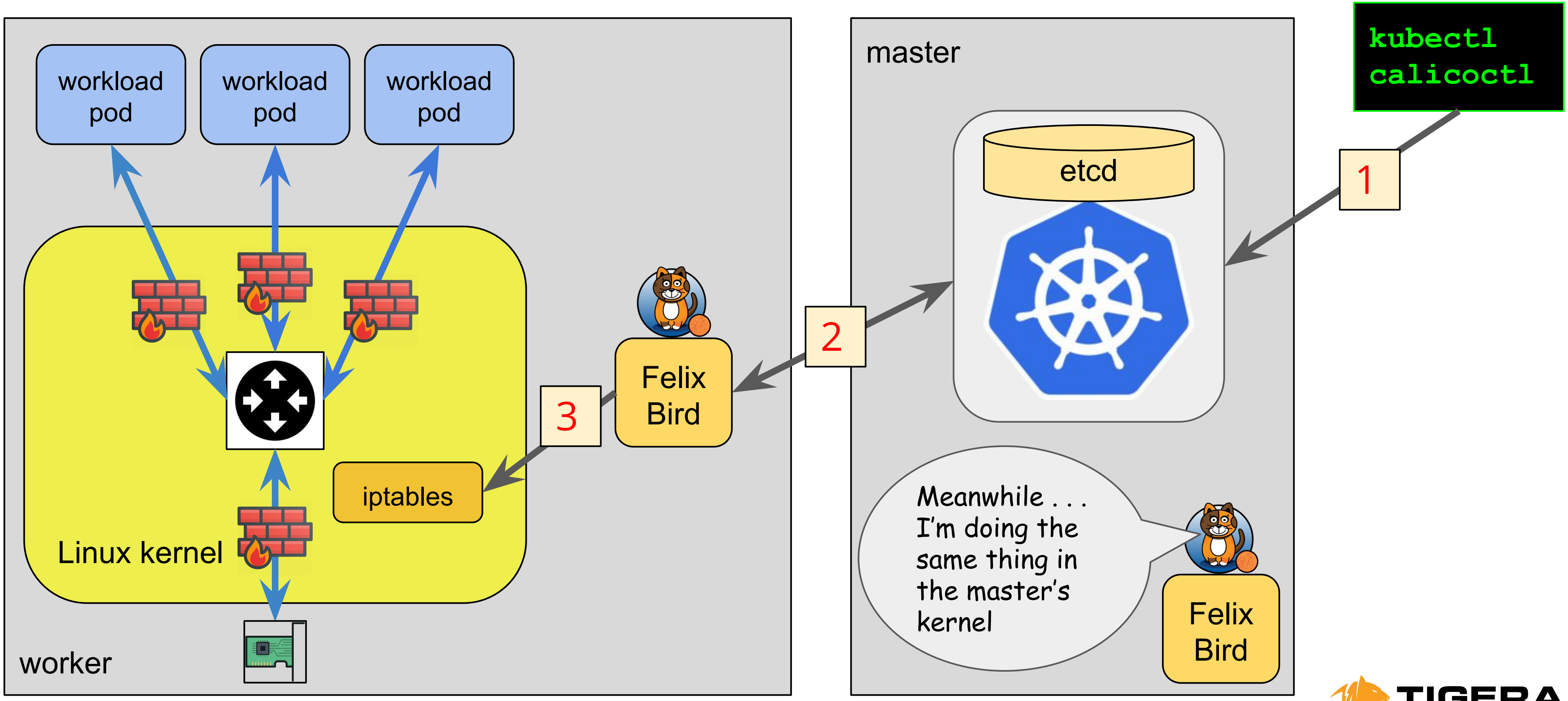
IP Pool definitions must be applied using calicoctl

Ok That’s Cool and All . . . But Why Would I Want This??

- IP Pools can help identify workloads in their logs
  - Teach Splunk or Elastic which subnet is which
- IP Pools can help identify workloads in external networking devices for security and network shaping
  - Prioritize network for certain workloads over others
  - Control traffic passing through external firewalls
- IP Pools can help establish firm boundaries between workloads for auditors and security teams
  - Use the flow logs from underlying networking to help build a compliance / security report

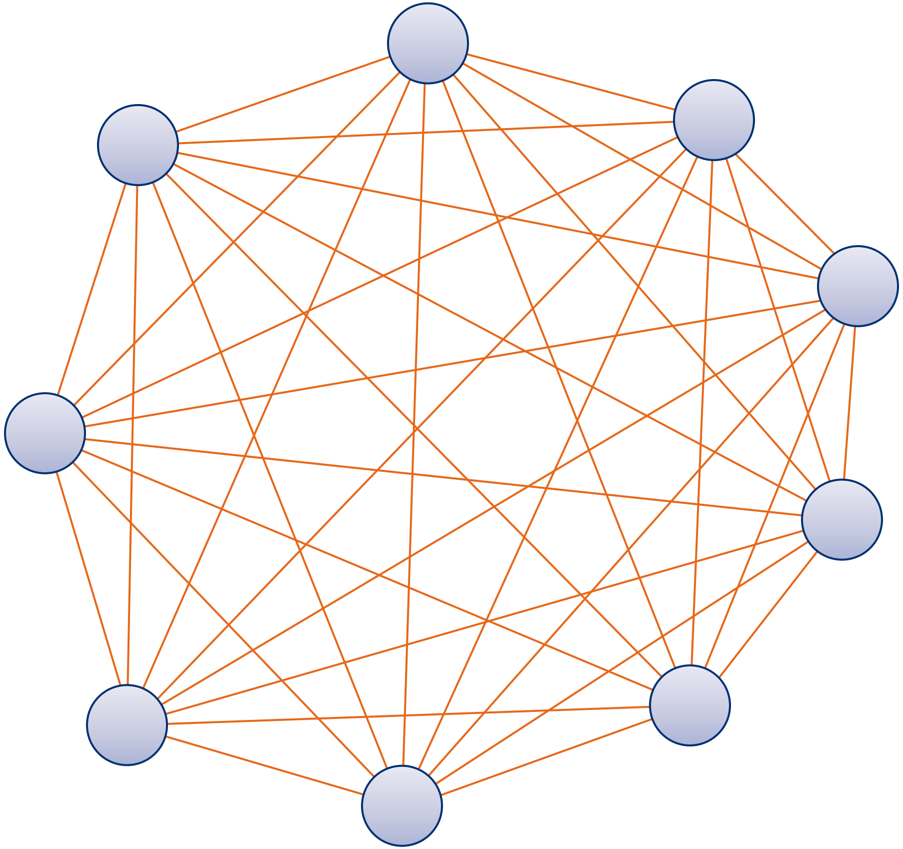
Calico Network Policy

Calico High Level Policy Architecture

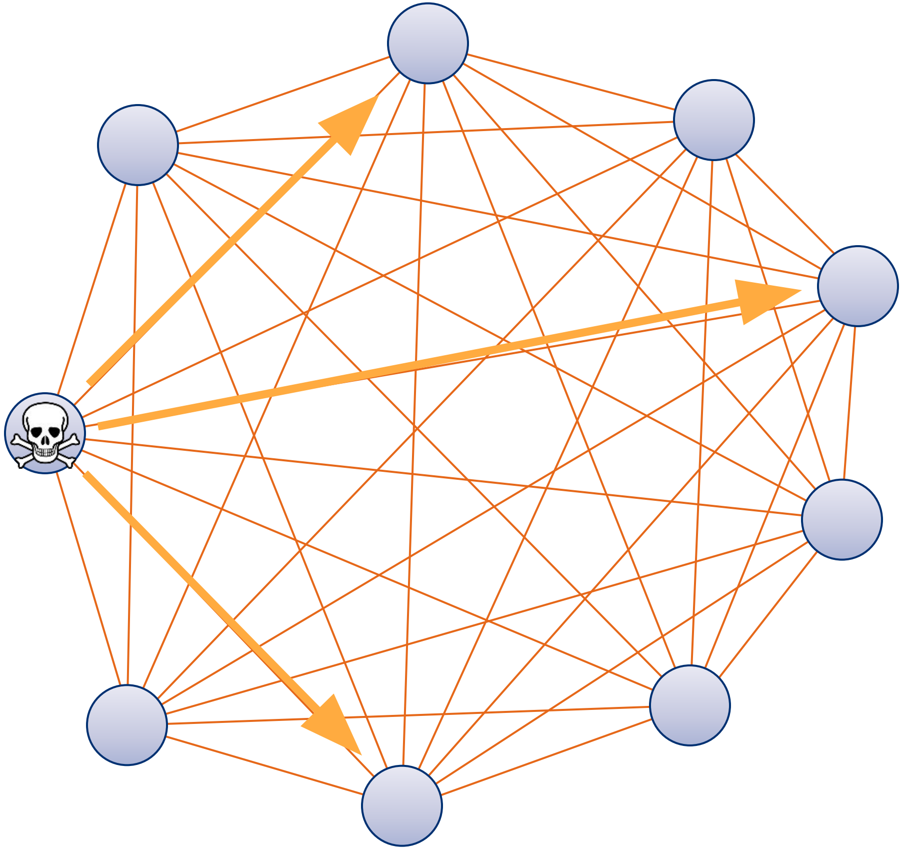


MOTIVATION FOR NETWORK POLICY

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.

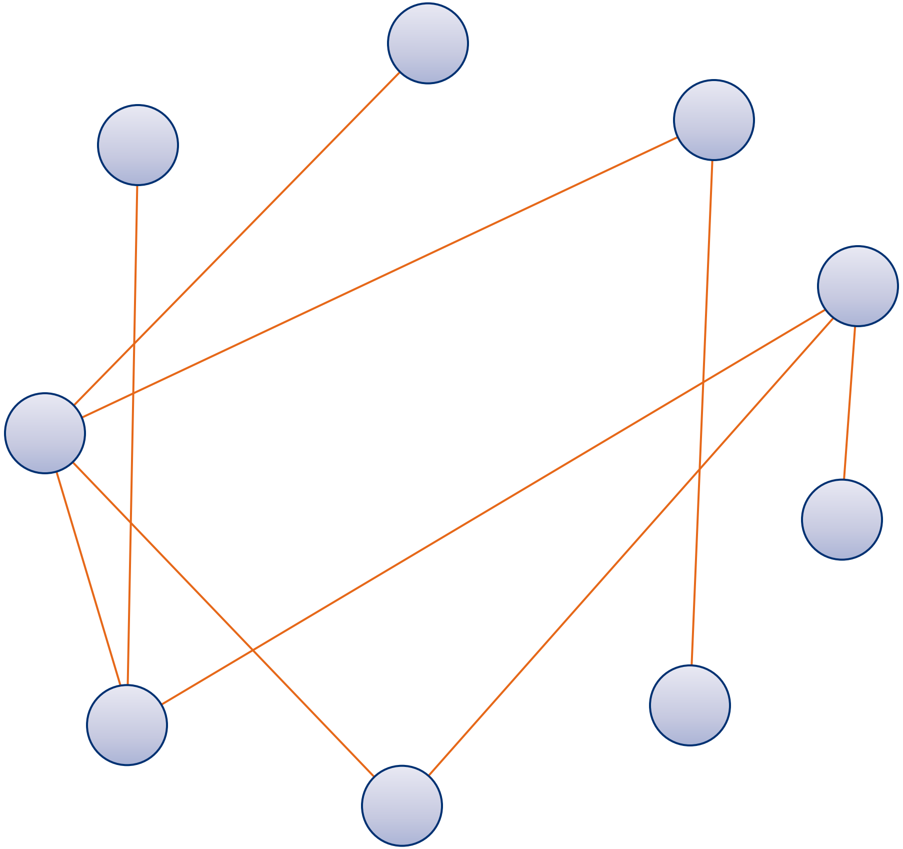


The rest are only useful for an attacker.

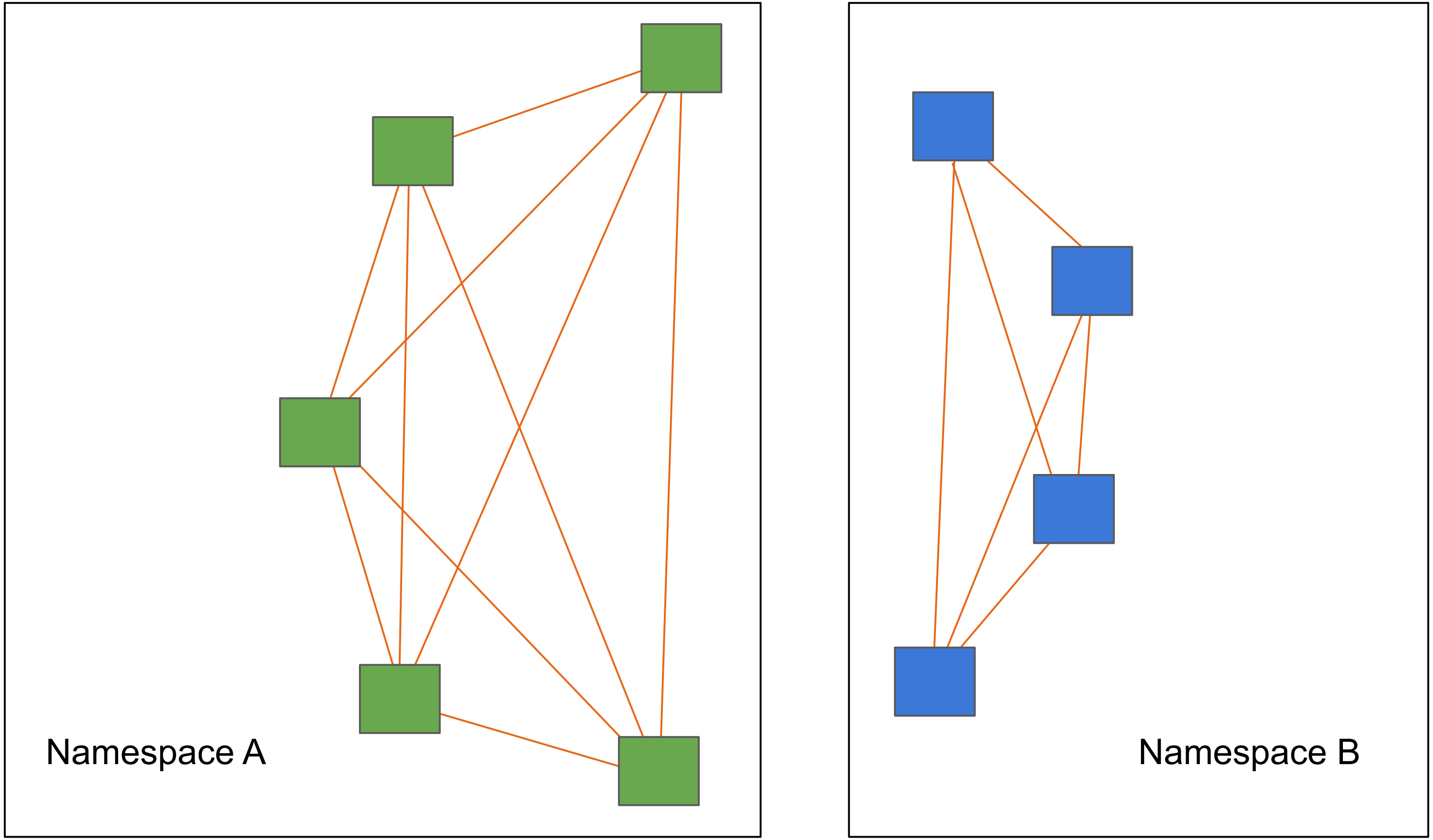


We know that our frontend load balancers don’t connect directly to the backend database. dev containers do not connect to prod, and so on.

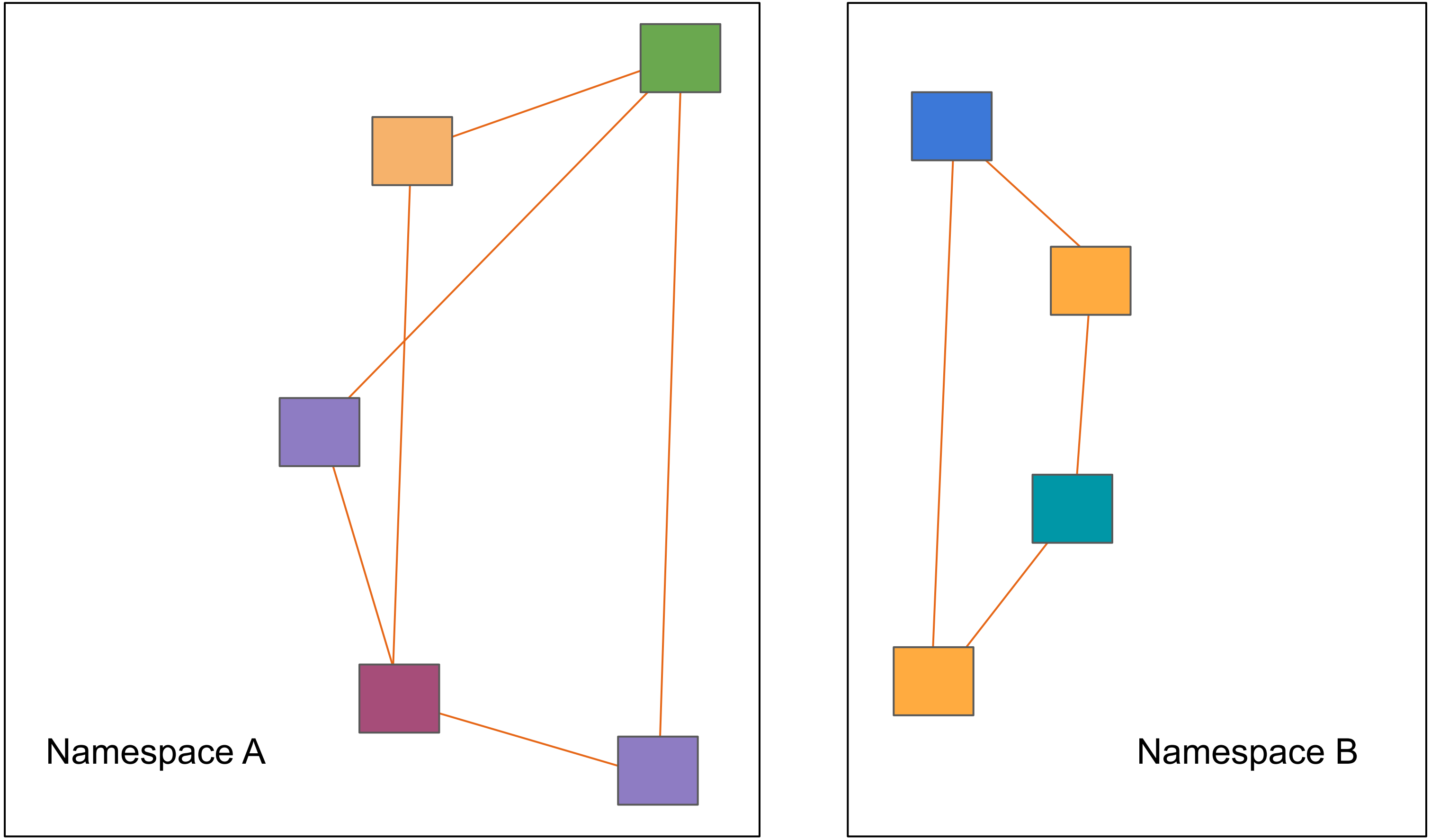
Each connection path that we can eliminate will reduce complexity and improve security.



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 (==, !=)

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

- Set membership (in, notin, exists)

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

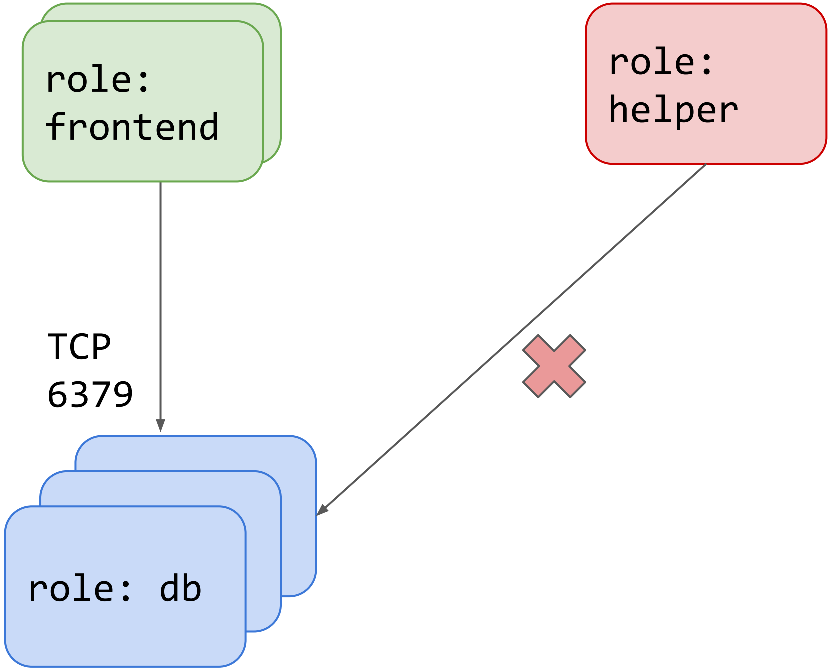
KUBERNETES NetworkPolicy RESOURCE

- 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/

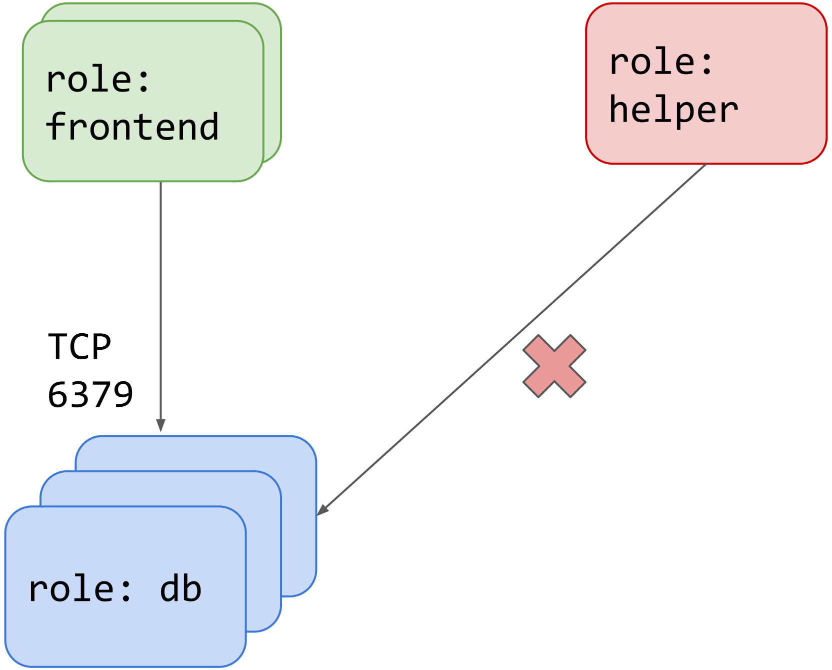
Kubernetes Network Policy Example

```
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
```



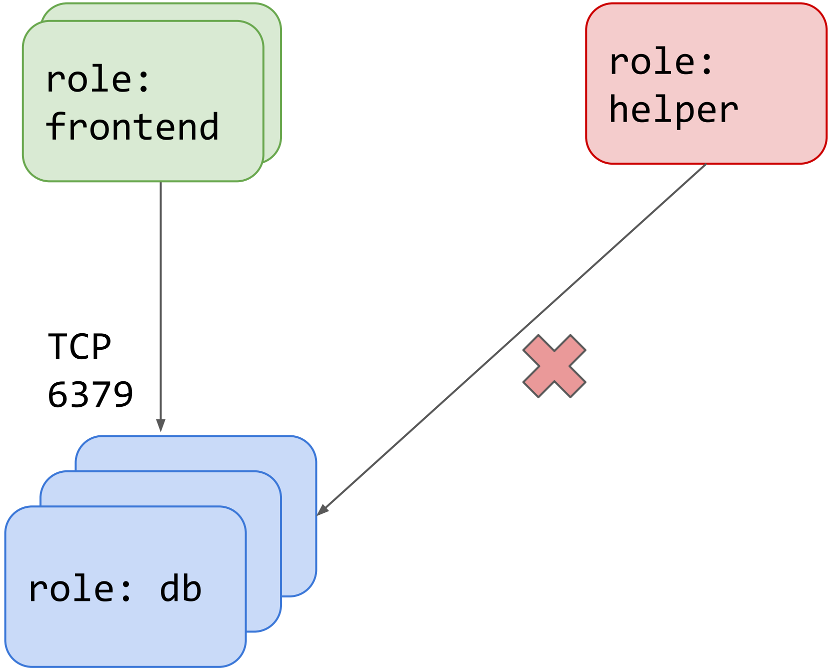
Kubernetes Network Policy Example

```
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
```



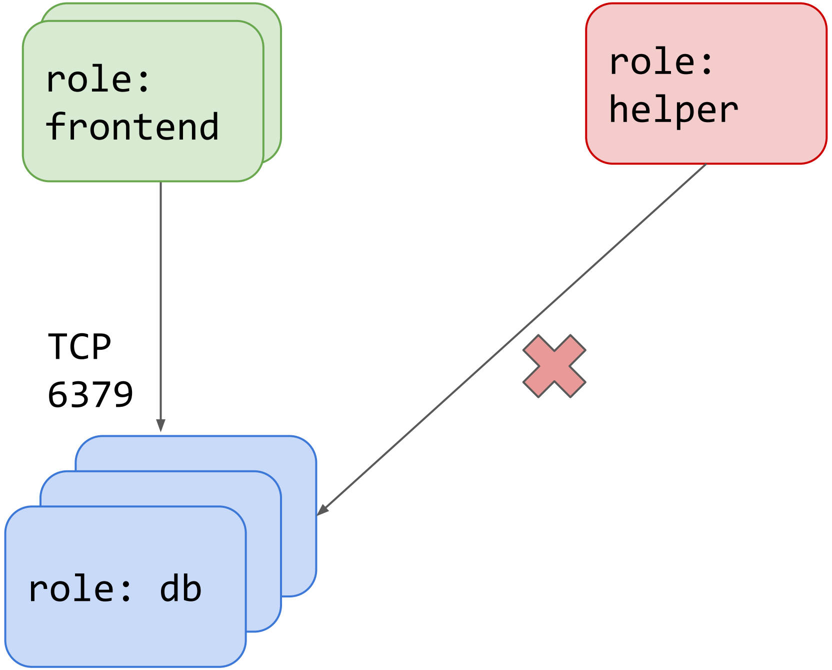
Kubernetes Network Policy Example

```
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
```



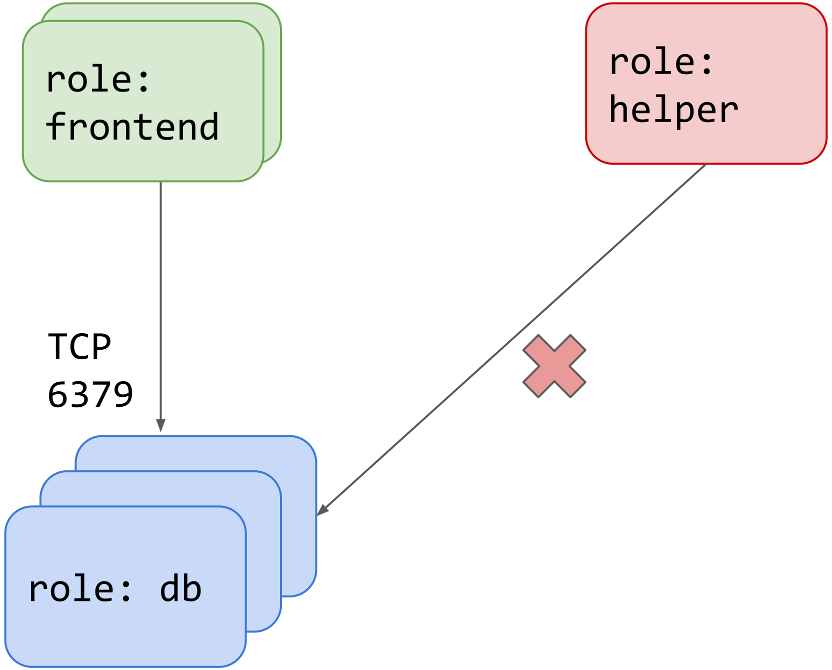
Kubernetes Network Policy Example

```
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
```



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
```

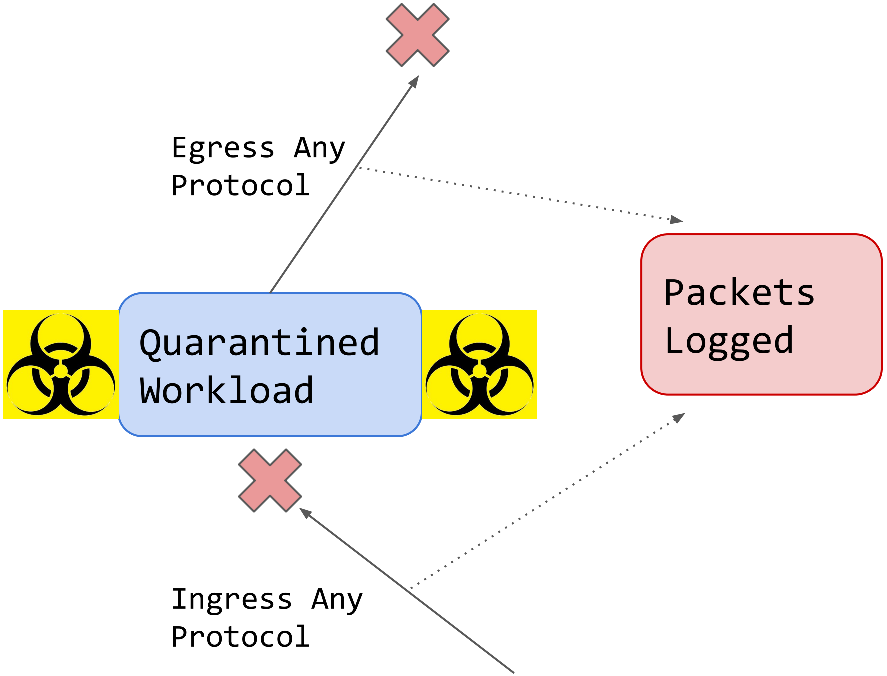


CALICO NETWORK POLICY

- Calico supports more advanced use cases beyond Kubernetes Network Policy
  - **global** network policies (rather than being limited to apply to pods in a single namespace)
  - the 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)
  - the ability to specify policy rules that **reference service account labels** (alpha in 3.0, GA in 3.1)
  - **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

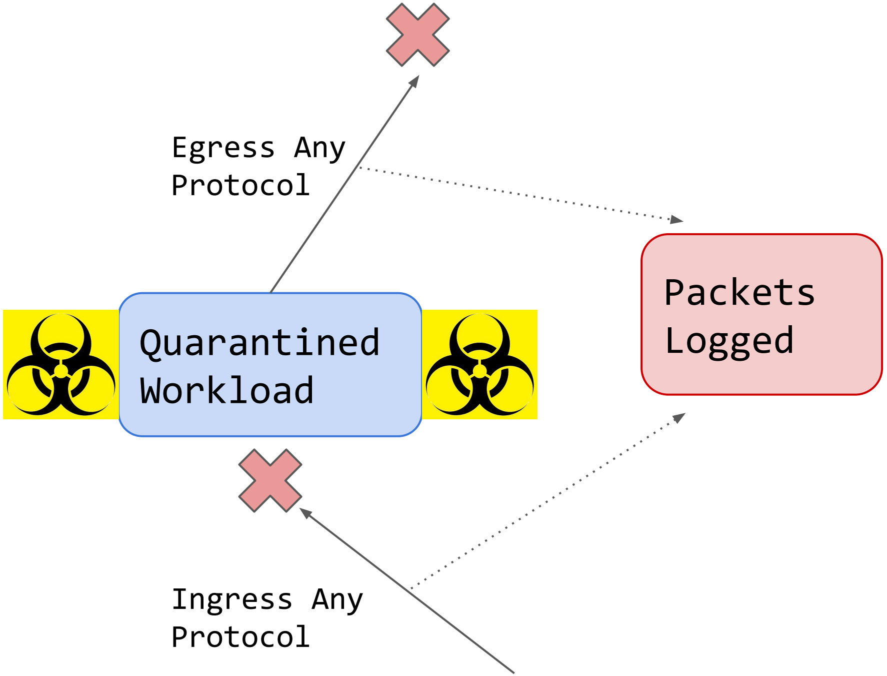
Calico Network Policy Example

```
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
```



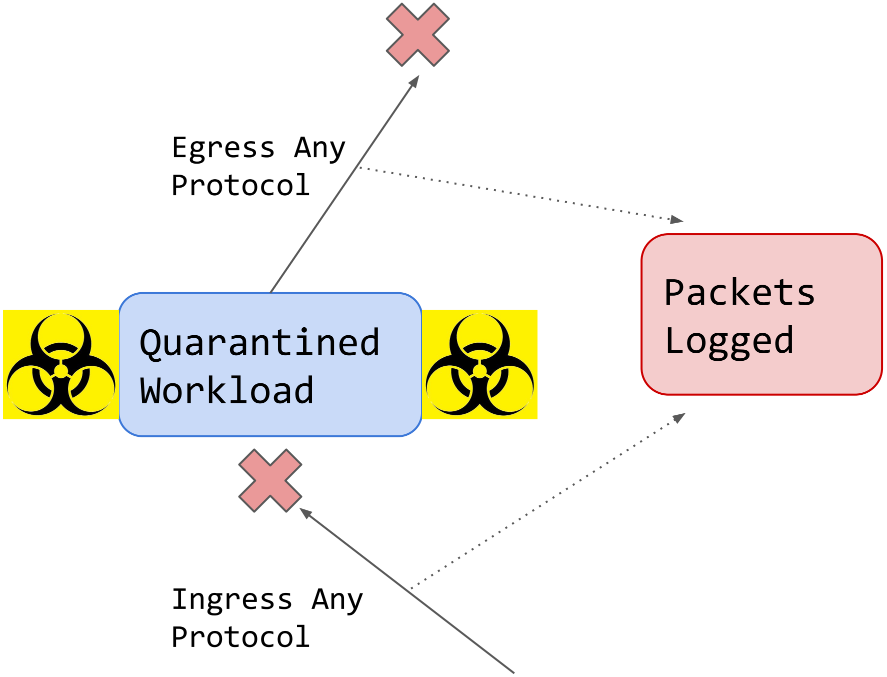
Calico Network Policy Example

```
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
```



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
```

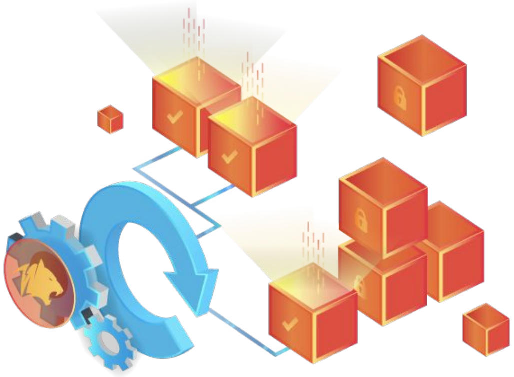


Calicoctl: The Calico Command Line

- 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!

Tigera Secure Enables Key Security Capabilities



Zero Trust Network Security	Visibility & Threat Detection	Continuous Compliance
Extend your security controls to Kubernetes	Monitor traffic, detect and prevent threats	Continuous reporting, alert on non-compliance

Thank You - Please like us at github.com/projectcalico/calico

TIGERA

Follow us on: