# Kubernetes Networking Deep-Dive

Stuart Charlton
January-February 2021

# About the Presenter


**Stuart Charlton**

scharlton@vmware.com

github.com/svrc

twitter.com/svrc


Principal Solution Engineer
Office of the CTO, Global Field

Formerly Pivotal Software, BMC Software,
IT operations executive, consulting & training for 20+ years

REST, SOA, Java/Spring, DevOps, IT Architecture, NSX, Kubernetes, Cloud Foundry

Calgary, Canada based.

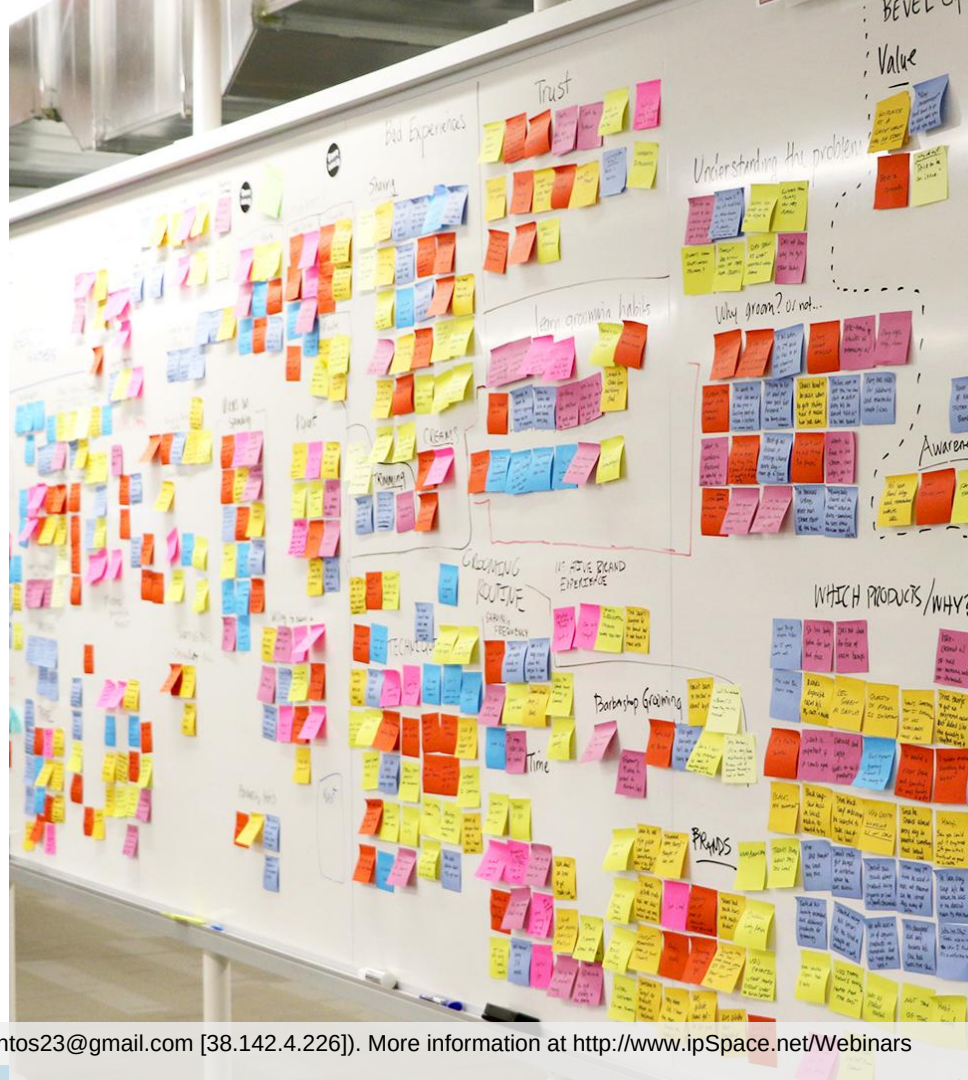
Any opinions expressed in this presentation are solely my own.

# Three Part Series

**Part 1:**
**Why Kubernetes?**
Controllers, Architecture
And the Kubernetes Network Model
(Using kubenet & flannel as examples)

**Part 2:**
**A Dive into the Container Native Interface:**
**Survey of many CNI Plugins &**
**Public/Private Cloud Provider Modules**
(NSX, Tanzu, OpenShift, Azure, AWS, GCP, Calico, Cilium, Antrea, etc.)

**Part 3:**
**Ingress Controllers and Load Balancers**
(NSX, Contour, Istio Gateway, NGINX, AWS, Azure, GCP**,** Knative, Cloud Foundry for K8s)



# Audience Assumptions

- Technical understanding of Linux, networks (routing, L2/L3), and basic container concepts (Docker)

- Prefer less of a full introduction on how to use Kubernetes, but **more** of understanding **how it interacts with the network**, and why there's so much hype

- Errata will be mentioned in future segments

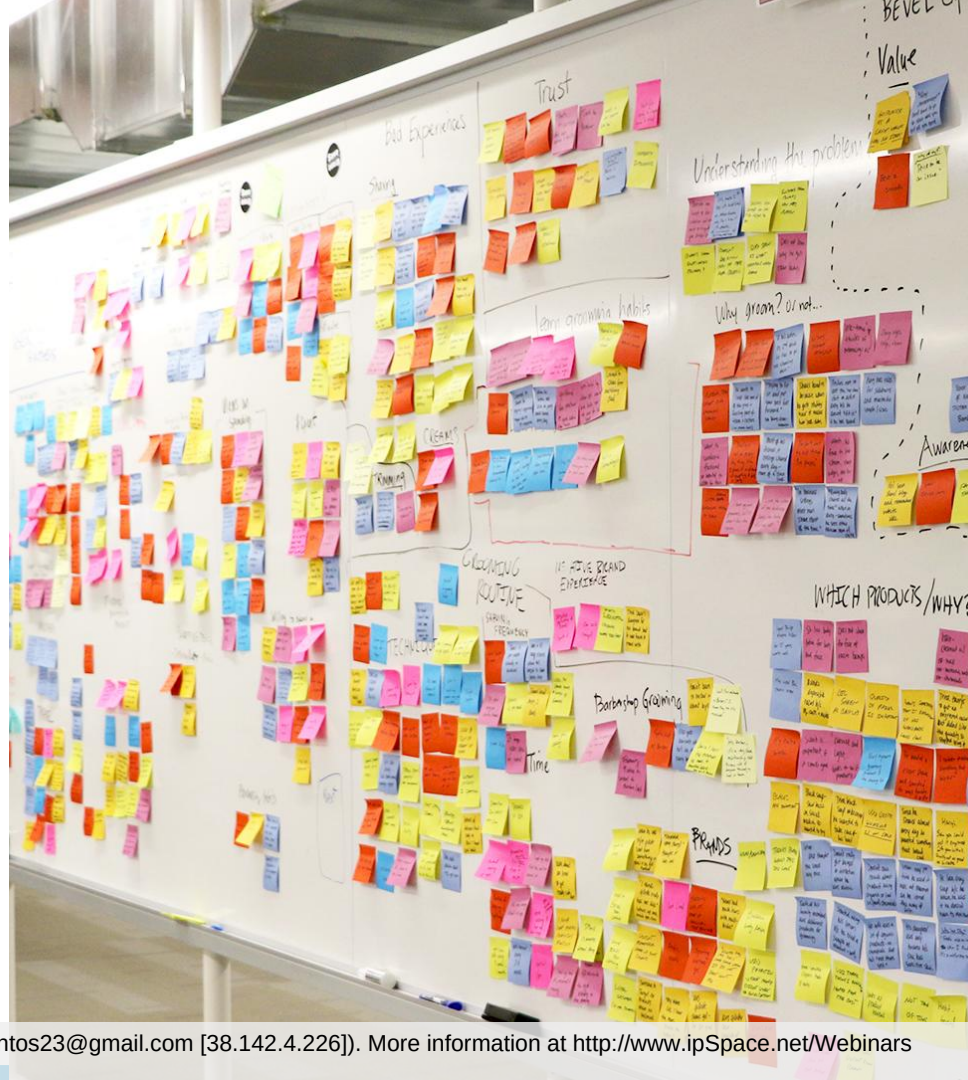
# Part 1

**Why Kubernetes?**

Controllers, Architecture

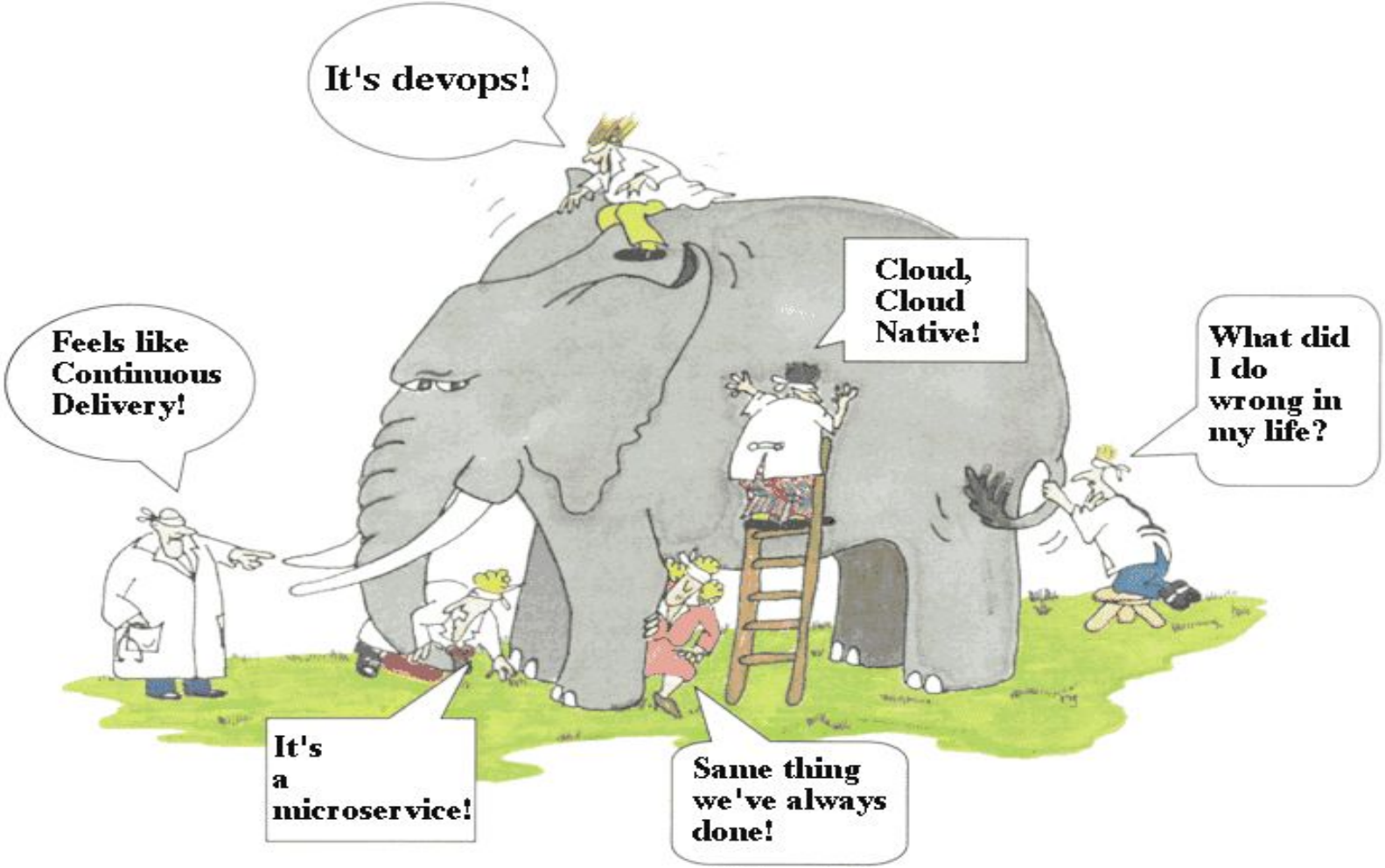
And the Kubernetes Network Model

(Using kubenet & flannel as examples)



# Why Kubernetes?

- Understanding Modern Applications
- What really is Kubernetes?




Source: @littleidea / Andrew Shafer

# Why Kubernetes?

Definition 1:

# Why Kubernetes?

Definition 2:

Allow COTS vendors and/or Enterprise IT to hide their legacy architecture under a veneer of respectability and YAML templates

# Why Kubernetes?

Definition 3:

Kubernetes provides a **balanced** set of abstractions and opinionated **patterns** for building and operating **modern applications**.

(A balance of expressivity, flexibility, modularity and complexity.)

# What's (technically) a Modern Application?


Source: @codinghorror / Jeff Atwood

# What's a Modern Application?

## Services-Oriented , Autonomous Software Runtimes

- Software that has a published API
- Software whose runtimes have independent life cycles
    - No massive recompile/relink - loosely coupled
    - Subsystem upgrades can be performed independently without "regression testing the world"

# What's a Modern Application?

## Intrinsic End-to-End Release Engineering & Automation

- "Move fast without breaking things"

- **Continuous Integration** and **Deployment**
- **Broad Automated Testing Coverage**
- **Soft launches** via feature flags / canary / blue-green
- **Testing** is conducted both **before** and **in** production (debatably indistinguishable from synthetic monitoring)

# What's a Modern Application?

**Infrastructure specification & configuration is intrinsically part of the software**

- Compute
- Network
- Storage
- OS
- Software packages & configuration
- Processes & health checks

# What's a Modern Application?

**Corollary:**
**Infrastructure largely consists of ephemeral resources**

aka. elastic, fungible, temporary, disposable

No more triple redundant big iron

# What's a Modern Application?

**The application owner is responsible for its performance and availability.**

No finger pointing or blaming the storage or network team.  (One can dream)

"Reliable systems made from unreliable parts".

Subsystems are Resilient & Self-Healing

# How does Kubernetes help achieve these goals?

1. **Change the industry's fundamental unit of compute**

   From **(Virtual) Machine** to **Pod (Container)**

# How does Kubernetes help achieve these goals?

## 2. Open Information, Modular Closed Control Loops

All information is **transparent** and **declarative**

All effects are **only** executed in **reactive** control loops

**Abstractions** to build **resilience** in your application

Similar approach as behavior-based robotics
(aka. The “subsumption architecture”
of Rodney Brooks / iRobot fame)

# How does Kubernetes help achieve these goals?

## 3. New abstractions for defining

- Network connectivity
- Resource capacity, scheduling, & consumption
- Software release lifecycle

# What about…?

Terraform
Ansible
AWS CloudFormation

Those tools all remain useful, but Kubernetes has subsumed a portion of what they do in a **standard, multi-cloud** manner.

# Practically speaking....

## In Kubernetes YAML, an application can define

- Network ingress/egress configuration
- Firewall policy
- Layer 4 or 7 Load balancer rules
- CPU and memory reservations
- File and block storage requirements
- OS images and initialization scripts
- Healthchecks
- Cluster & machine specification
- Orchestration to external APIs for my needs

# What really is Kubernetes?

1. **The industry's closest thing to a standard cloud API and control plane:  compute, network, storage (and more)**

   Not everything needs to be or should be Kubernetes resource, but it appears we're going to try

# What really is Kubernetes?

**2. An extensible set of reconciliation engines that encourages emergent behavior: self-healing, extensibility, autonomy**

Approximately 30 out of the box, and you can add more

Reactions cascade in a loosely coupled,  transparent manner

# What really is Kubernetes?

**3. A set of abstractions with opinions on how to build and operate modern applications**

- **Pods**
- Pod Controllers (e.g. ReplicaSet, StatefulSet)
- Volumes
- **The Kubernetes Network Model**

- Questions from the audience

# Kubernetes Architecture

- Intro to Controllers
- Component Topology, and how they interact
- The Kubernetes Networking Model

# Understanding Controllers

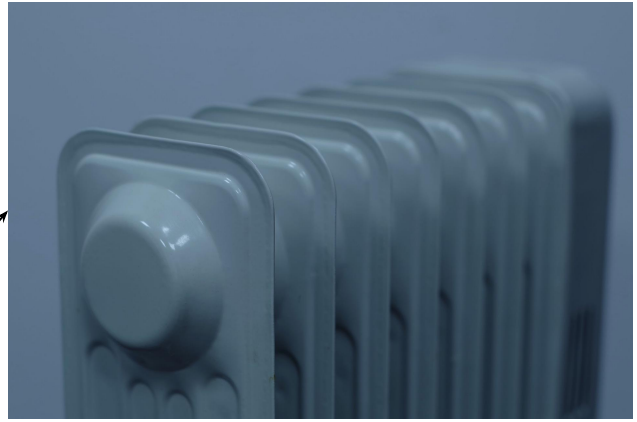
- “Control theory in control systems engineering is a subfield of mathematics that deals with the control of continuously operating dynamical systems in engineered processes and machines.” -- wikipedia

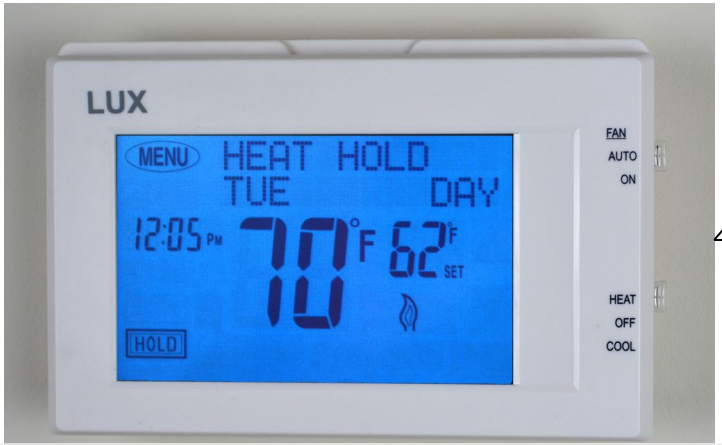
- Field formally launched in 1868 by James Clerk Maxwell in his study of centrifugal governor which is used to control the speed of an engine

- For our purposes:
  is the idea of **controllers** that continuously compare **current state vs. actual state,** then act to **converge** the actual state to the desired state with **idempotent** operations

# Controllers are everywhere

```
while(true) {
  if( sensor.currentTemp() < plan.expectedTemp()){
      heater.on();
      airConditioner.off();
   } else if( sensor.currentTemp() > plan.expected()){
    heater.off();
    airConditioner.on();
   }
   Thread.sleep(30 seconds);
}
```





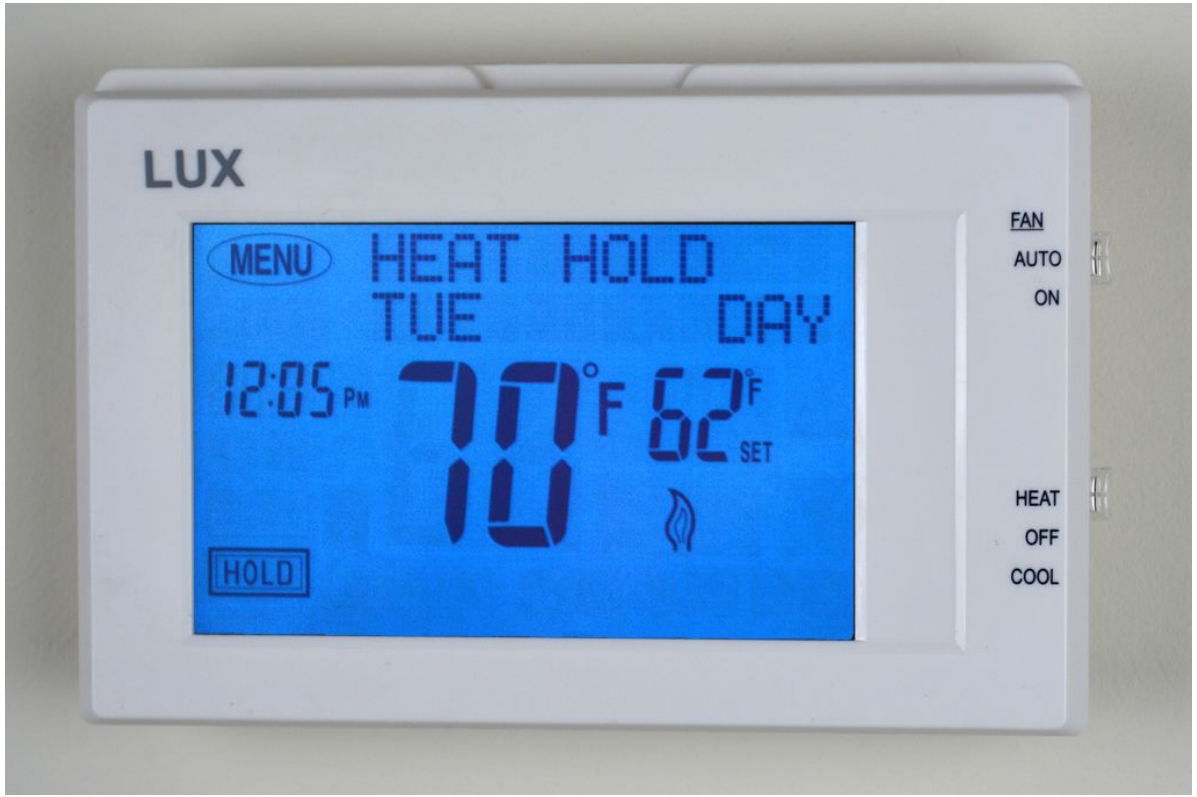

# Kubernetes & Control Theory

- A cluster of Kubernetes bare metal or virtual machines running containers is a **dynamic system**
  - VMs can die and come back
  - New VMs can be added to the cluster
  - VM's can be removed from the cluster
  - Containers crash and need to restarted
  - New container might be launched or killed for scale-out or scale-down
  - Containers might be stopped due to error
  - Containers might be upgraded by user
  - Network configuration may drift and need updates.

- Kubernetes is an example of a ***choreographed system***
- No central orchestrator or "brain"
- 30+ modular controllers are always taking action to **converge** current state to desired state

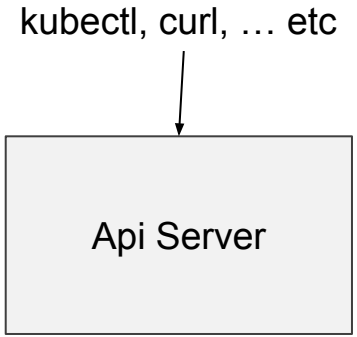
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 10
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80

kubectl apply -f deployment.yml
```

# What does a controller need?



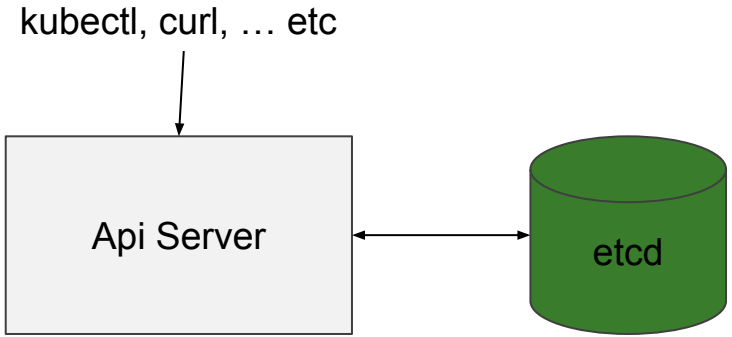

- Actuators to set desired state
- A place to store desired state
- Sensors to measure actual state
- Control loop with the logic to compare desired and actual state determine what action needs to be taken
- Ability to interact with the controlled devices

# What is the interface for setting desired state in K8s?



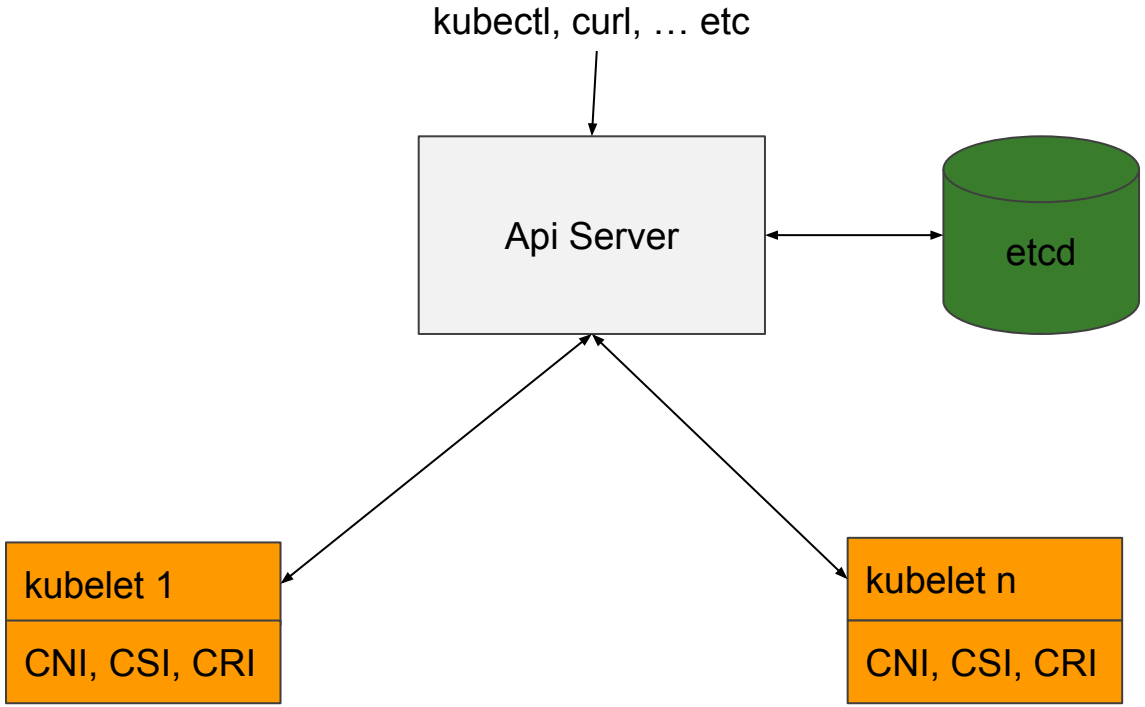

- A REST API server acts as the interface to the K8s cluster
- A collection of cli clients and libraries can interact with k8s api server

# What is the place where desired state is stored in k8s?



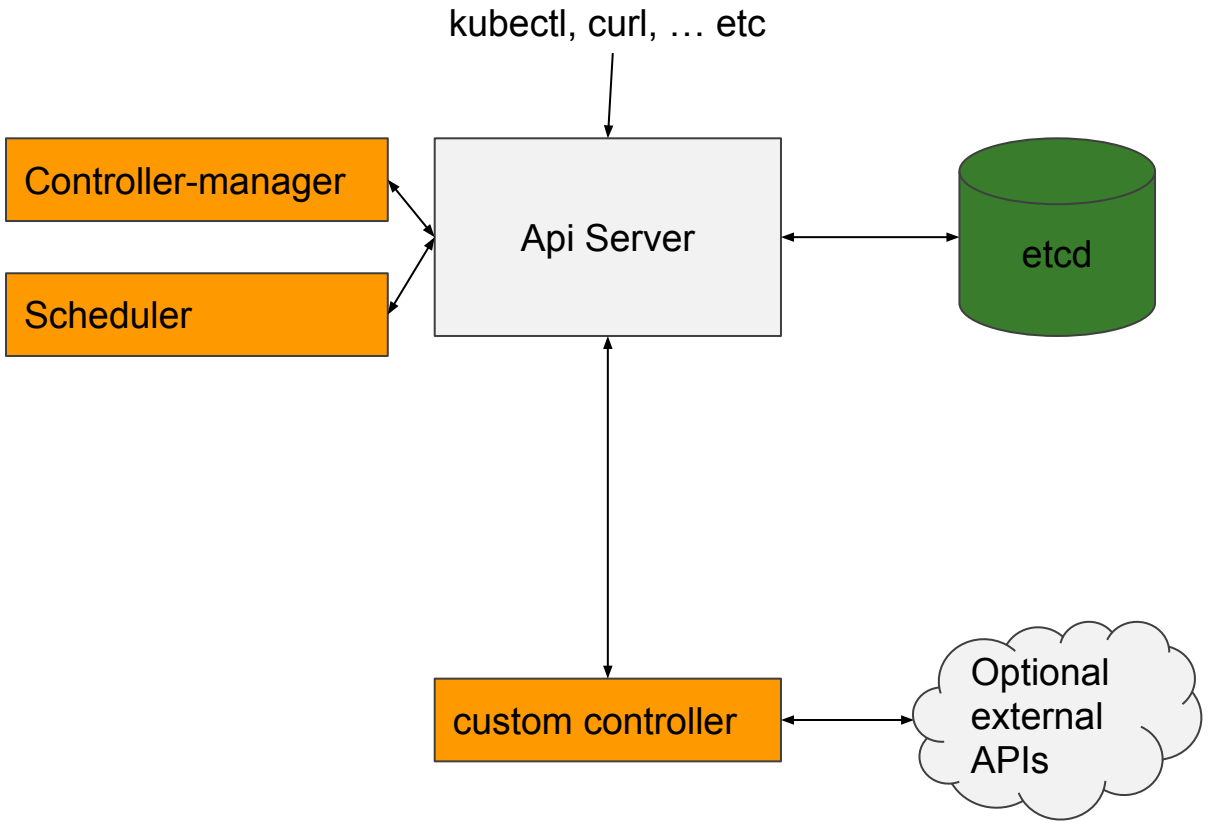

- Etcd is used as the desired state store

# How does k8s watch & control the state of running containers?




- The kubelet is a process that launches & keeps track of what containers are running on a worker node
- Connects with the container runtime using the **Container Runtime Interface (CRI)** API
- Connects with the SDN controller with the **Container Network Interface (CNI)** CLI+API
- Connects with the SDS (storage) controller with the **Container Storage Interface (CSI)** API

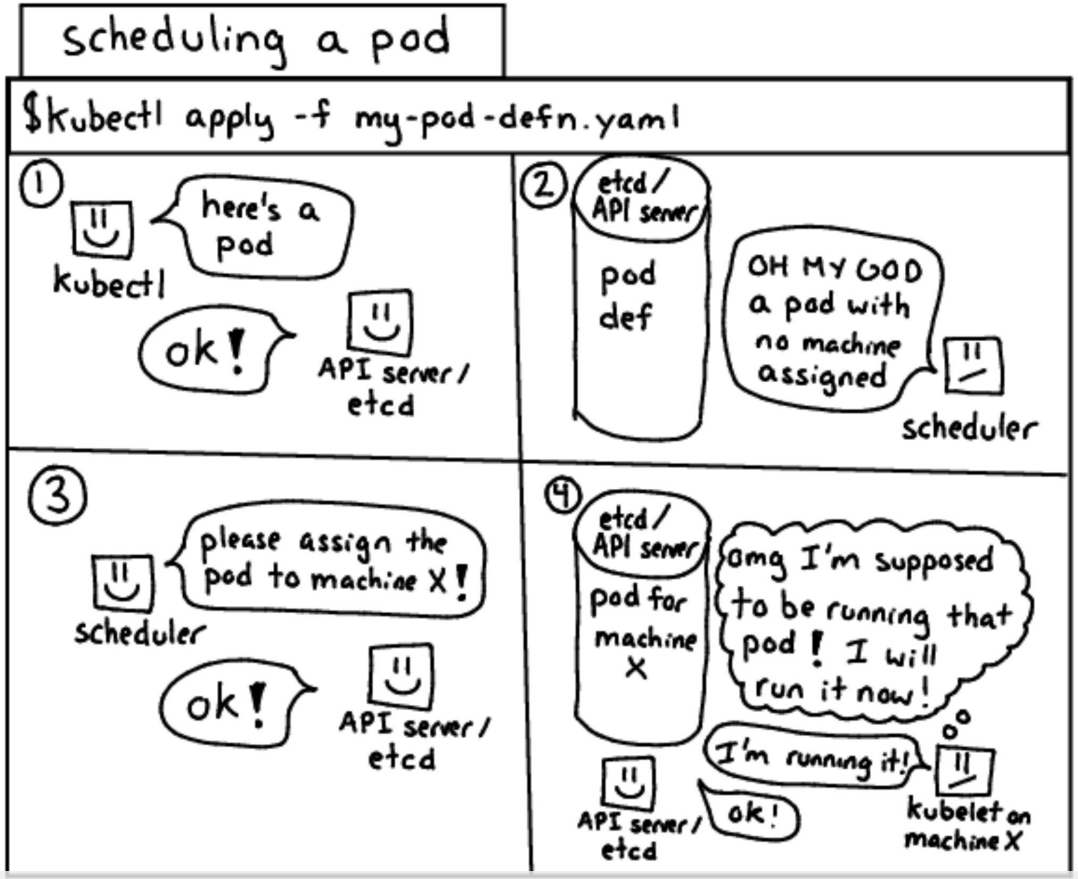
# Where is the control logic kept is k8s?



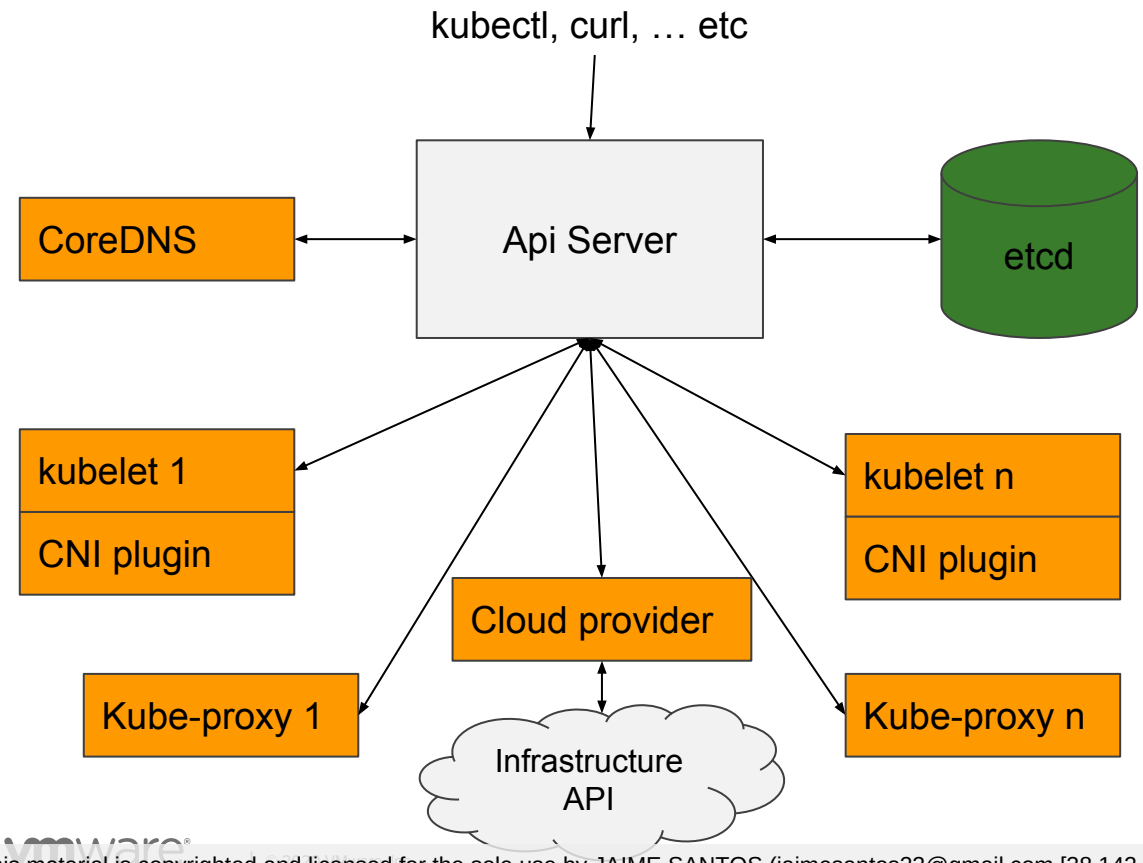

- The control logic in spread a across a variety of processes that connect to the api server
- **Kube-controller-manager** contains most out of the box control loops as threads in a single process/binary
- **Scheduler** is a controller responsible for allocating Pods to Hosts
- Extensions for k8s are typically built as **controllers** running as processes inside Pods. Also known as "**operators**"

# Scenes from Kubernetes

Source: Julia Evans
https://drawings.jvns.ca/scenes-from-kubernetes/

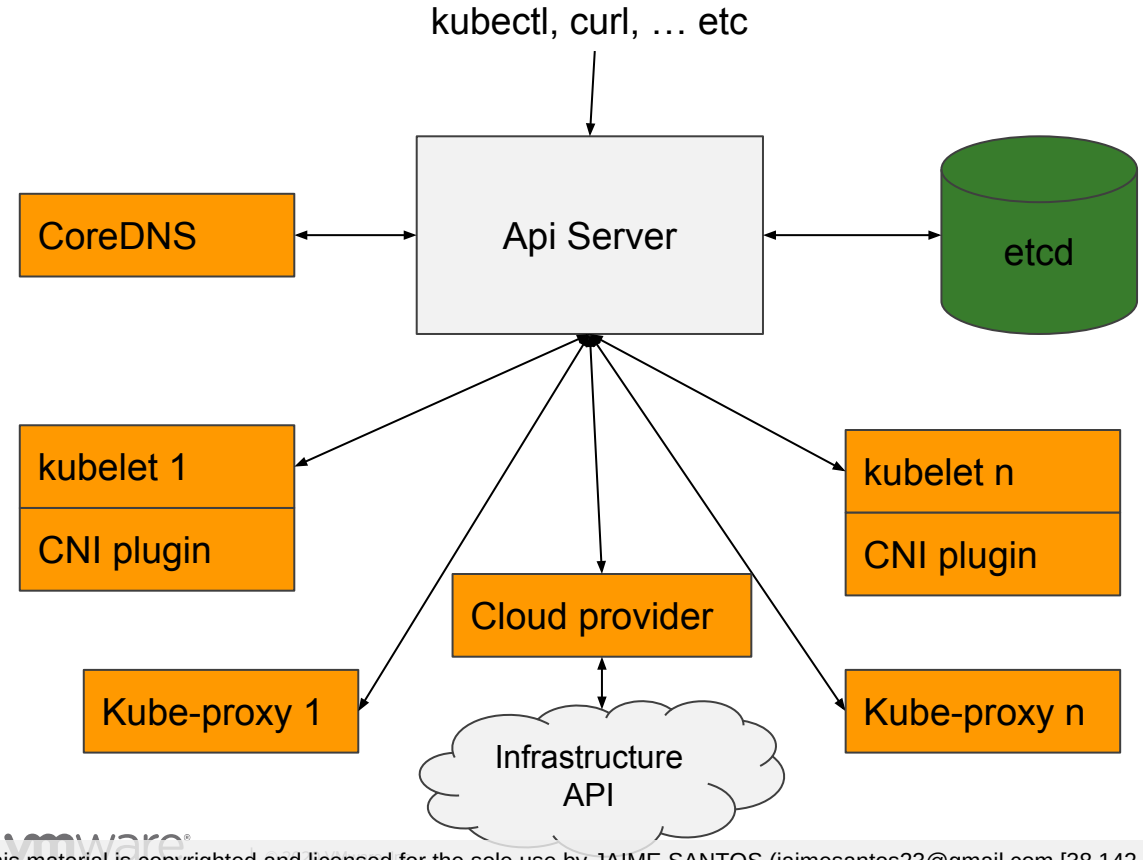


# How does k8s control the network?



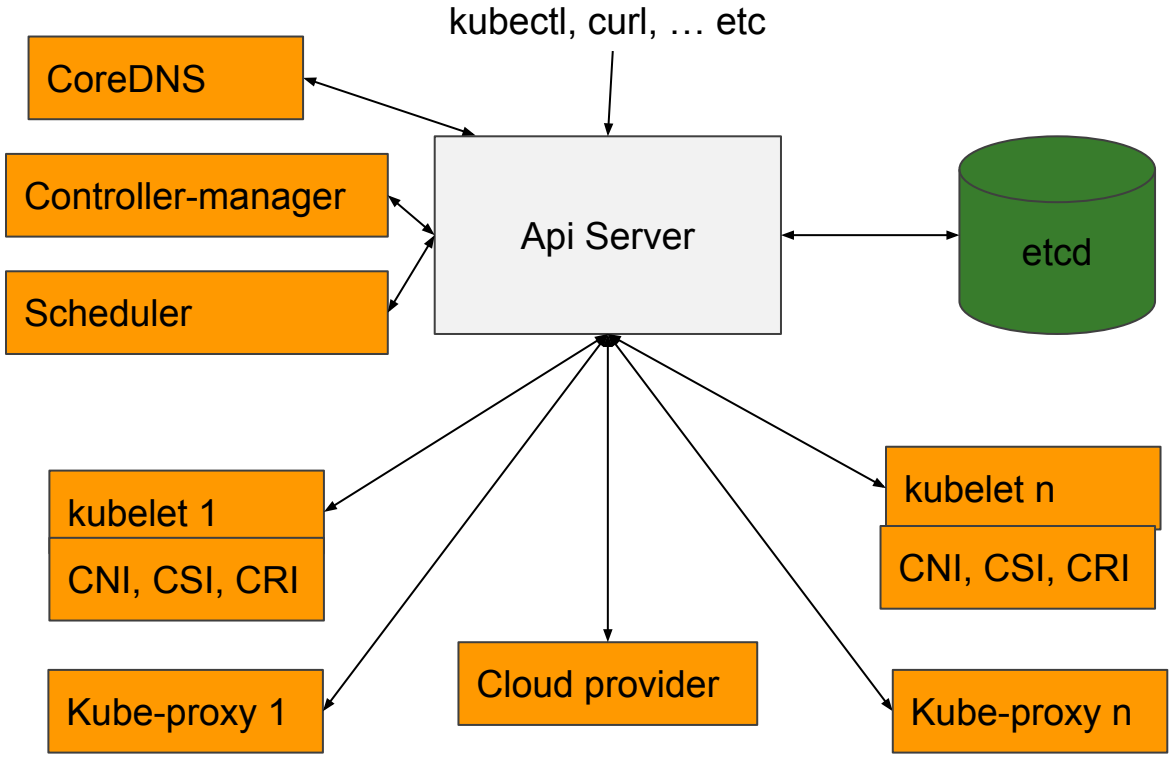

- **Kubelet** talks to a CNI plugin (CLI) which interfaces with the OS and any SDN controllers (if any)
- **Kube-proxy** (optionally) configures Kubernetes services: configuring SNAT, DNAT, load balancing in the OS
- The **cloud provider** is a controller that can orchestrates cluster-wide network routes, load balancers, firewall rules, etc.

# How does k8s control the network?



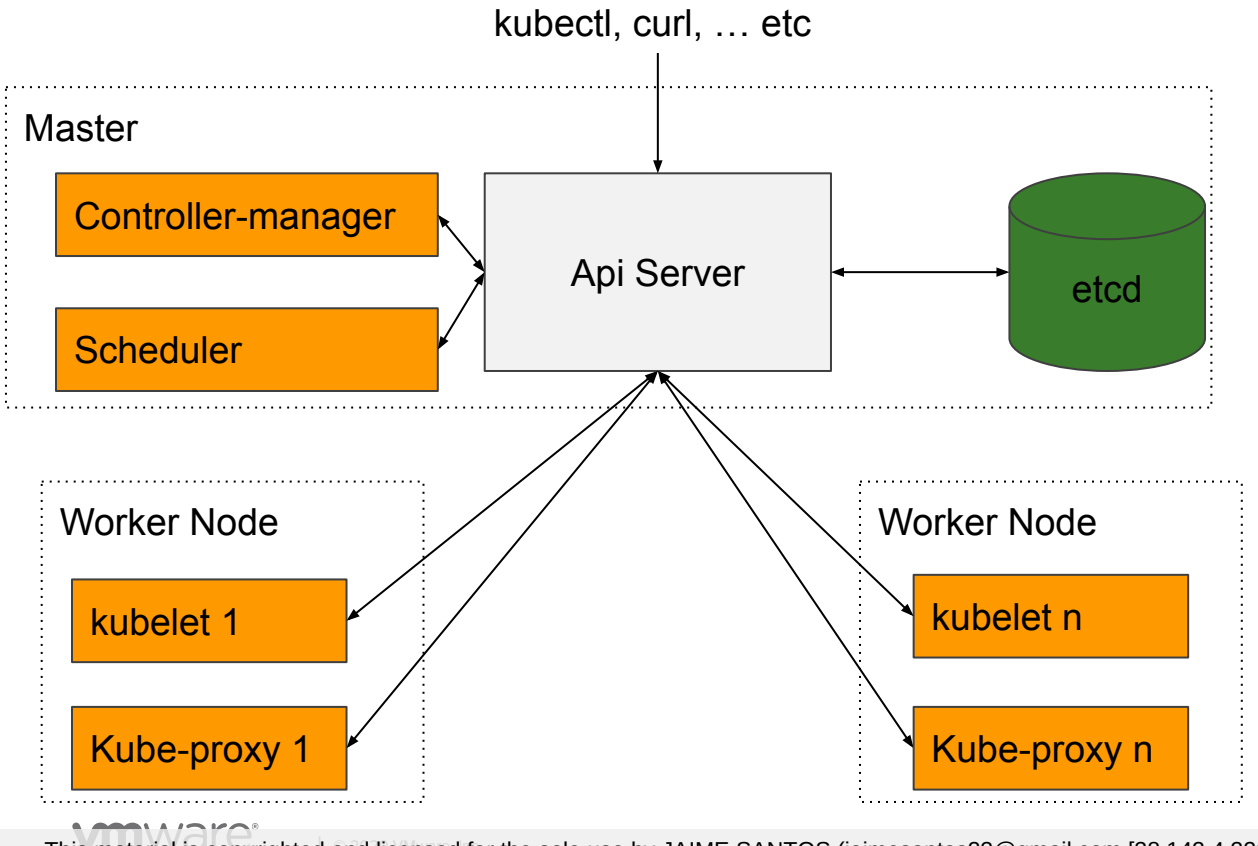

- The **CNI plugin** orchestrates node-level network interfaces, routes, encapsulation, and firewall policy.
- CoreDNS offers **cluster-local DNS zones** for service discovery, and acts as a caching resolver to an upstream

# The Kubernetes Control Plane



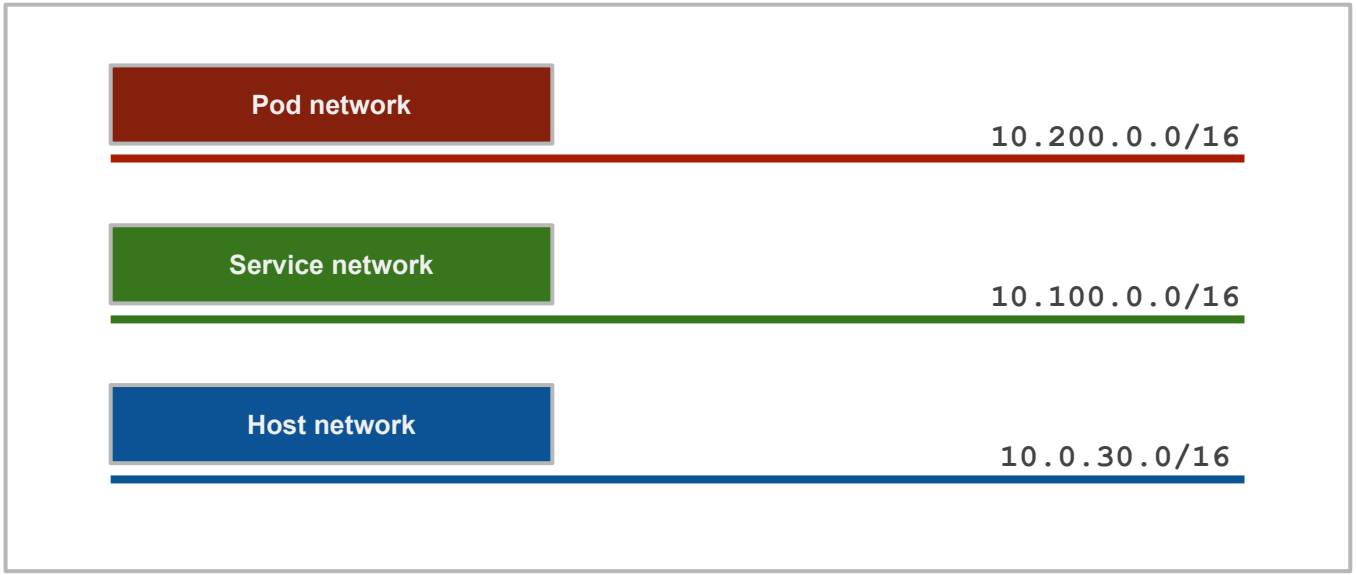

- The K8s control plane consists of all components that are used to track the state or change the state of the compute, network, or storage
- All the components of K8s can be run on a single machine or VM, or multiple (with some restrictions)

# Master Nodes vs. Worker Nodes



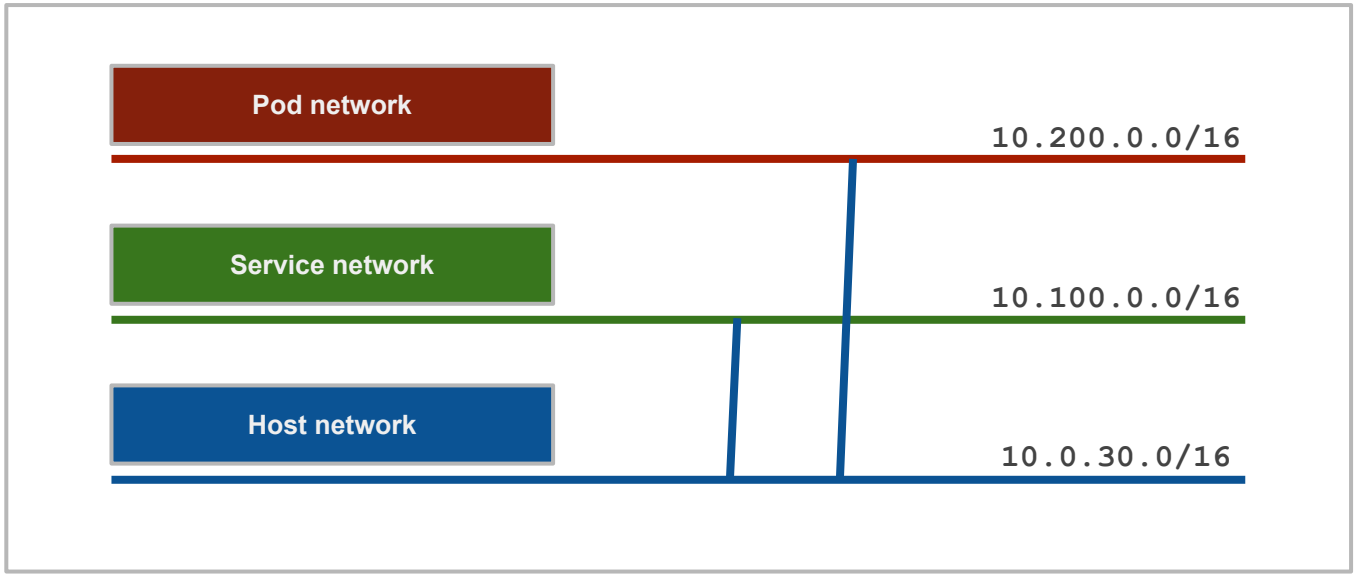

- A k8s master typically runs at minimum:
  - etcd
  - kube-apiserver
  - kube-controller-manager
  - Kube-scheduler
  - Cloud provider controller
  - SDN controller(s)
  - Storage controller(s)
- A K8s worker node runs
  - kubelet
  - Kube-proxy
  - Container runtime
  - SDN node agent(s)
  - Storage node agent(s)
- Master/worker can often be combined for self hosting
  - e.g. “Static pods” for the k8s master

# The Kubernetes Networking Model




- Three Layer 3 Forwarding Domains on each Worker Node
- Each Worker node is a Layer 3 Router, pure IPv4 or IPv6
- Dual stack is currently [December 2020] alpha

**Host network must have connectivity to Service and Pod Networks**
**No requirement** that Pod and Service networks have cluster-external connectivity

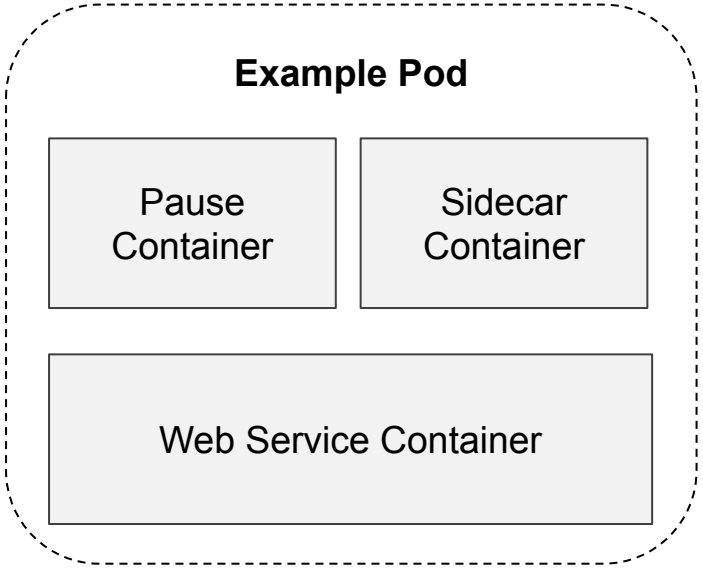


# Questions from the audience

# Understanding Pods

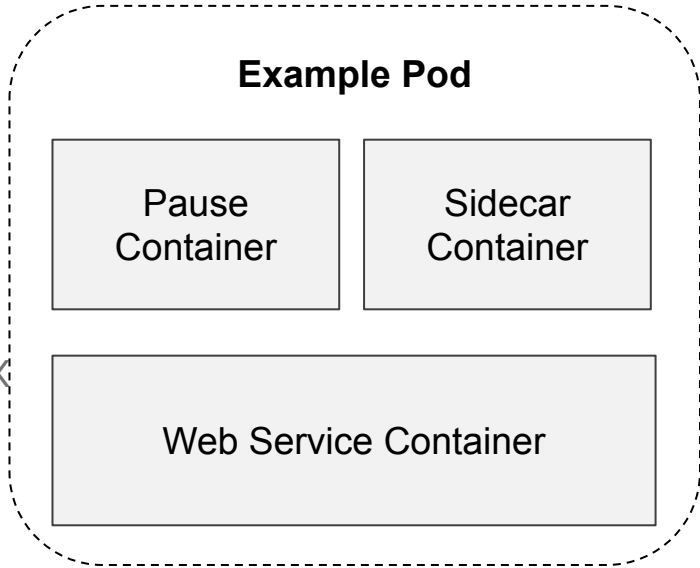
# What is a pod?

- A **set of containers** with the following Linux namespace & cgroup configuration:
- All containers share
  - network namespace (like a VRF)
  - ipc namespace
  - (optionally) selected volumes
- Each container has seperate
  - pid namespace
  - uts namespace
  - mnt namespace
  - user namespace
  - cgroup namespace
- Each container resource consumption is controlled via a cgroup with specifiable requests / limits
- The pause container owns the namespaces that are shared across other containers



# What is the pause container?

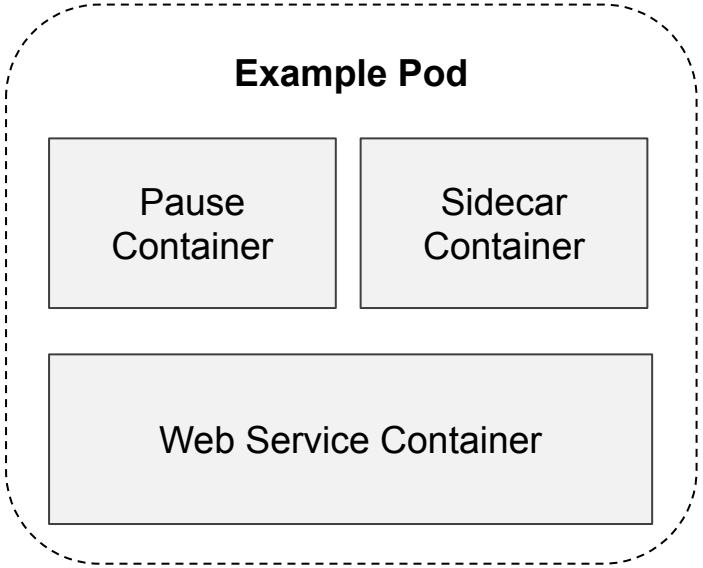
- The pause container is the first container launch in all pods
- Pause container holds the shared network and ipc namespaces
- Pause holds the ip address of the pod
- Very simple C program (68 lines of C code) that register UNIX signal handlers then goes into an infinite loop



# The pod network namespace

- Since all the containers in the pod share the network namespace
  - Pod IP is allocated via a pluggable CNI IPAM module
  - All IPs are ephemerally allocated
  - Ingress/Egress Static IPs require other constructs

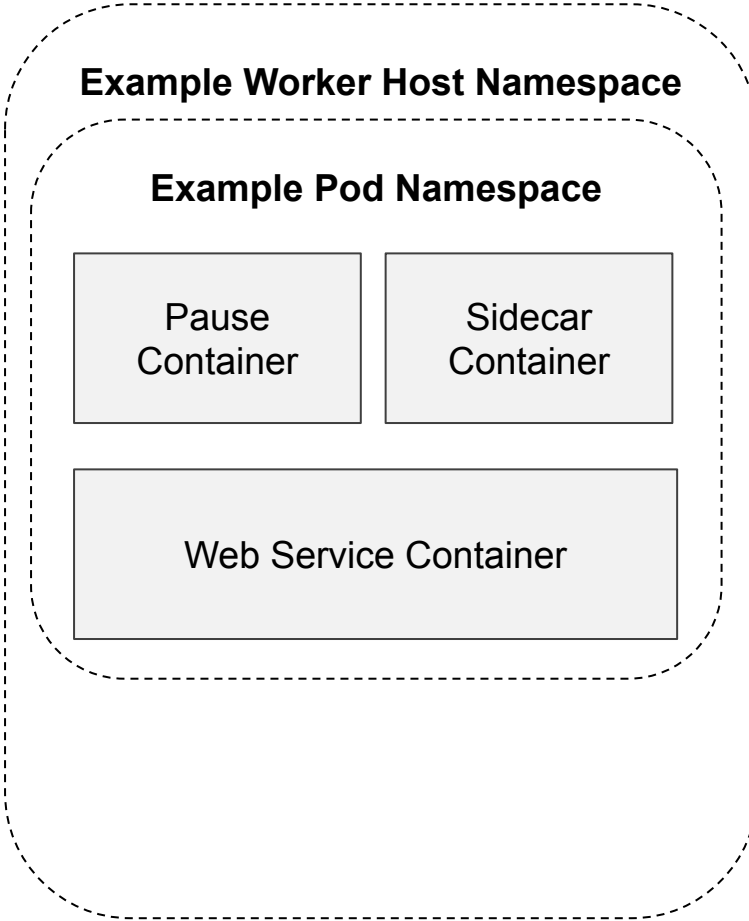
  - **1 IP address shared by all the containers (each pod is like having a private VRF)**
  - Containers inside a pod can communicate with each other over the localhost interface



# Customizing the pod

- Pods can escape its namespaces
  - **hostNetwork**: places containers in the host network namespace
  - **hostPort**: DNATs a worker port to a pod namespace port using a **CNI plugin** (be careful: Services are probably better)
  - **hostPath**: mount worker subdirectories as volumes

- These can be secured for only privileged use
  - Pod Security Policies admission controller
  - Open Policy Agent admission controller



# The Pod Network Model Constraints

- **Mandatory** constraints
  - The host network must have **routable connectivity** to the pod network
    - If a process outside of a pod needs to talk to a pod, it can
  - All pods across all worker nodes are **routable** to each other
  - All pods across all worker nodes are **routable** to the **service network**
  - All containers within a pod **share** a networking namespace
    - and thus IP address, routing table, ARP table, etc.
  - Pod IP address allocation is **dynamic** and **never** static

- **Optional**, varies depends on your specific cluster configuration
  - Pods are routable to the host network
  - L2 adjacency
  - Unknown Unicast/Broadcast/Multicast
  - Subnet allocation

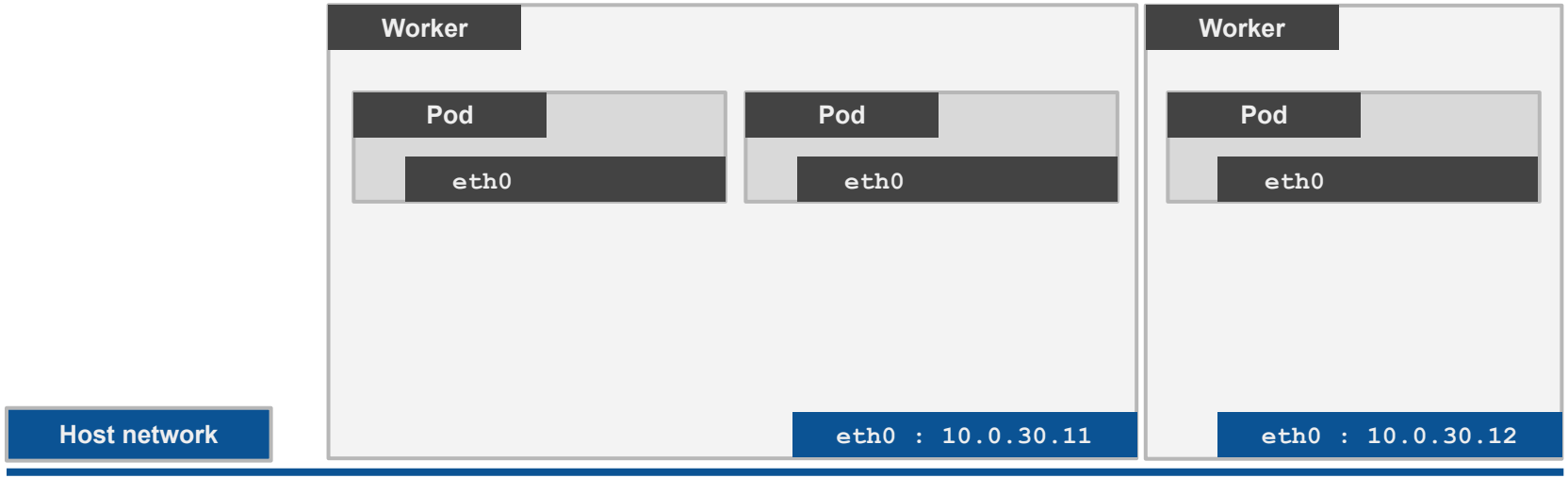
# Pod Network Considerations

- **Interfaces -** Typically a **Linux virtual ethernet pair** (veth) with one side in the pod namespace, the other in the host namespace

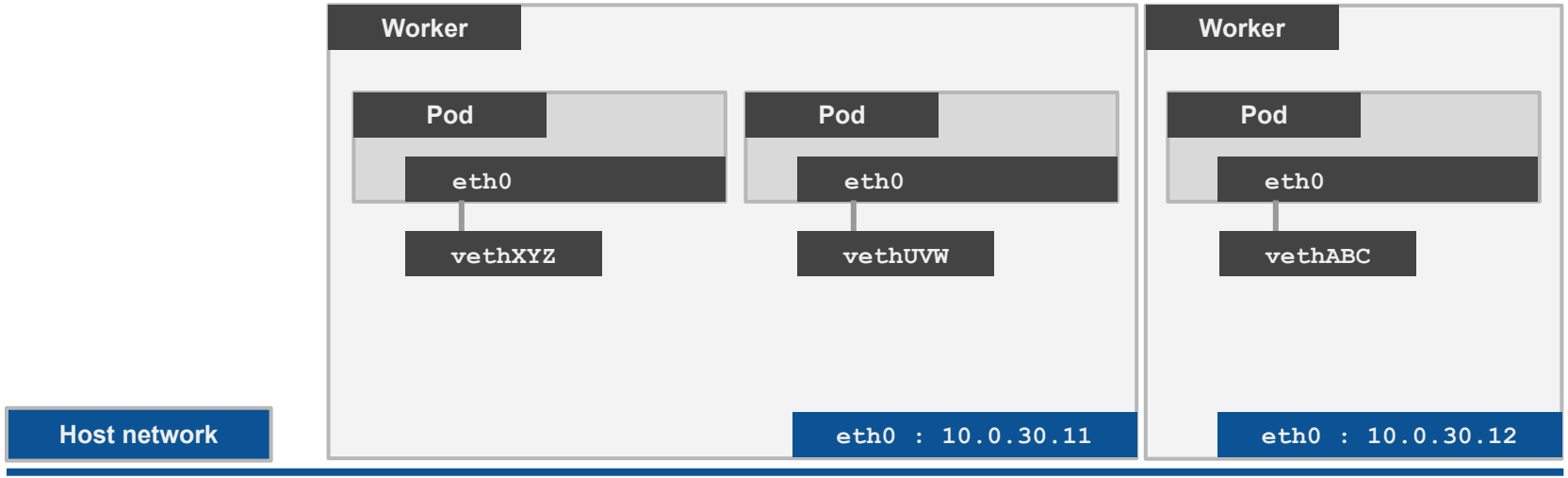
- **Routing** - Lots of options!
  - L2 Bridge on the worker node (Linux native bridge or Open vSwitch)
  - Link local L3 routing (e.g. 169.254.1.1 default gw in the pod)
  - Pure L3 to the host network
    - The default assumption
    - Assumes L2 adjacency among worker nodes,
      or multiple subnets allowed on a single VLAN (i.e. IP aliasing)
  - Encapsulated virtual overlay on the host network
    - E.g. VXLAN, GENEVE, IPsec over GRE
  - Host route or Pod subnet advertisements:
    SDN controller in-cluster, SDN controller external to cluster, BGP

# Kubernetes Network Model
# Traffic Walk

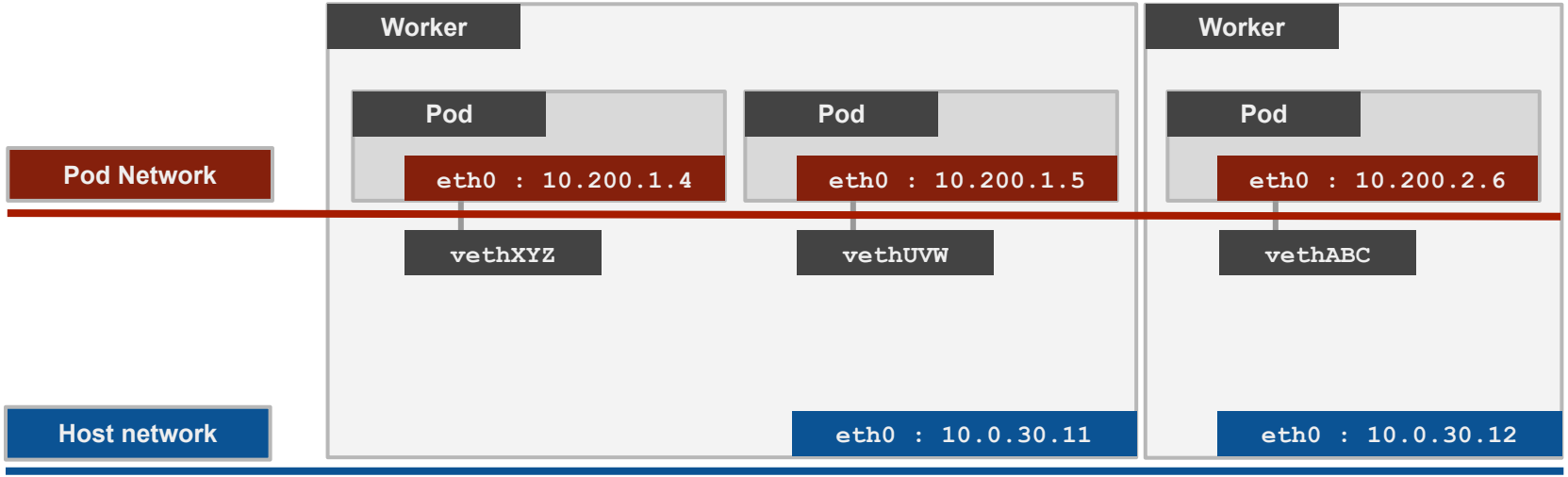
# Kubernetes Network Model
# Traffic Walk



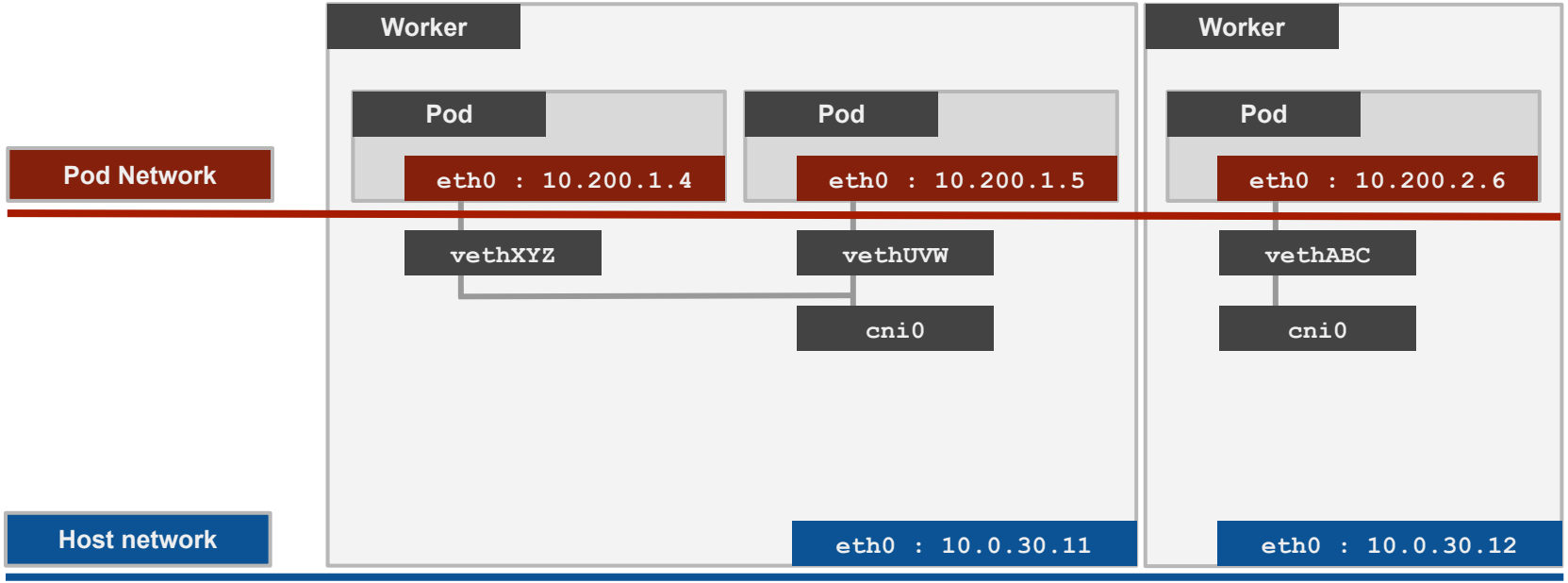
# Kubernetes Network Model
# Traffic Walk



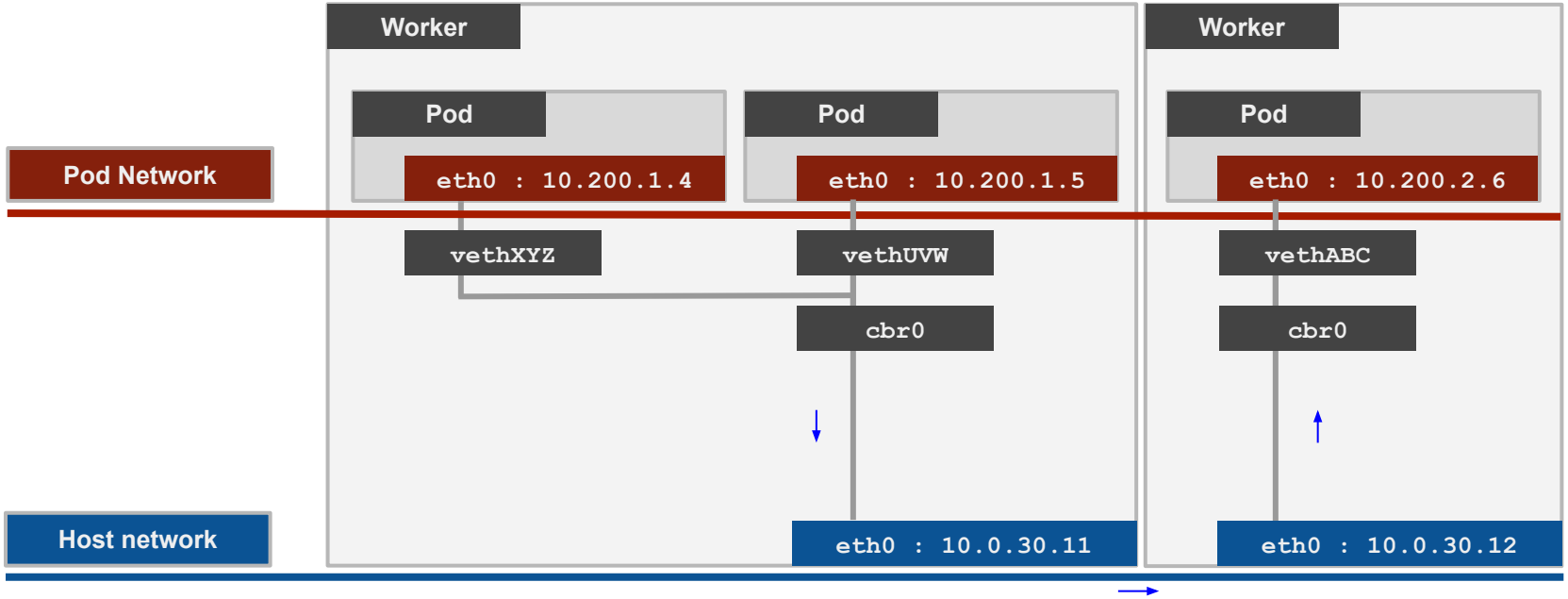
# Kubernetes Network Model
# Traffic Walk



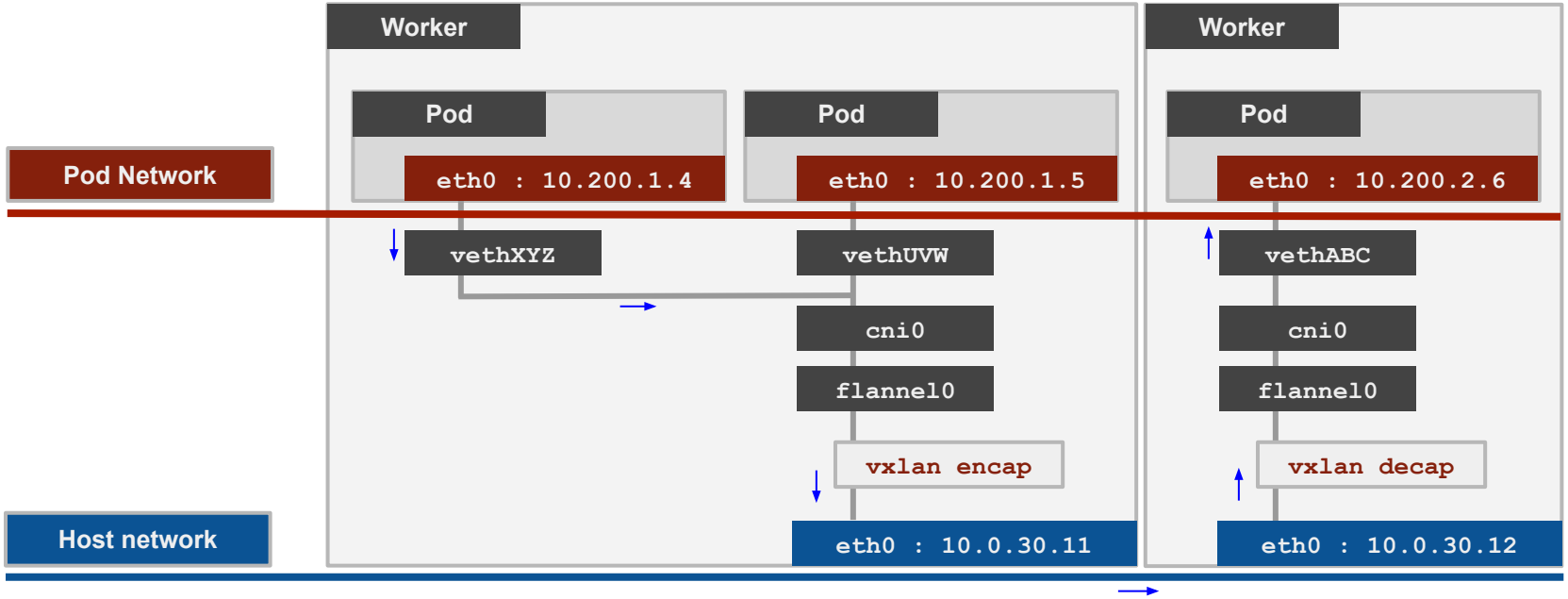
# Kubernetes Network Model
# Traffic Walk



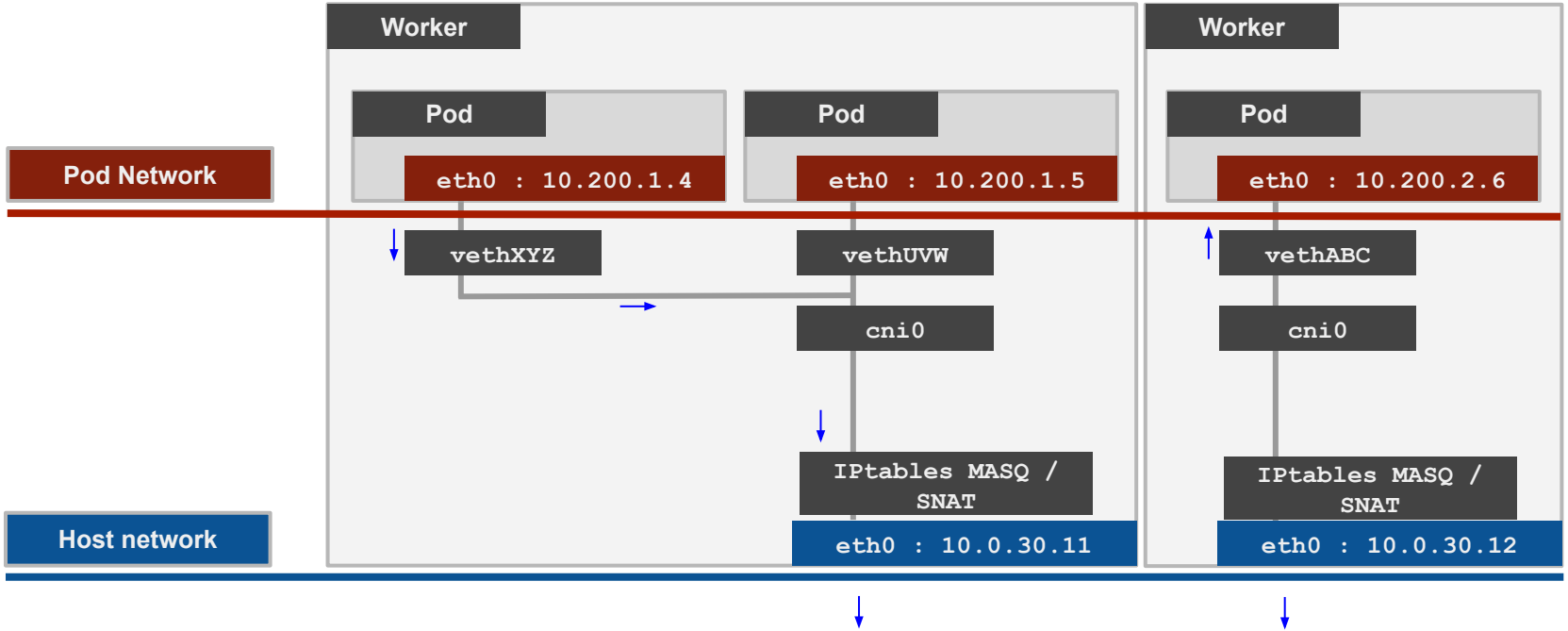
# Kubernetes Network Model Traffic Walk
# Kubenet (batteries included networking)



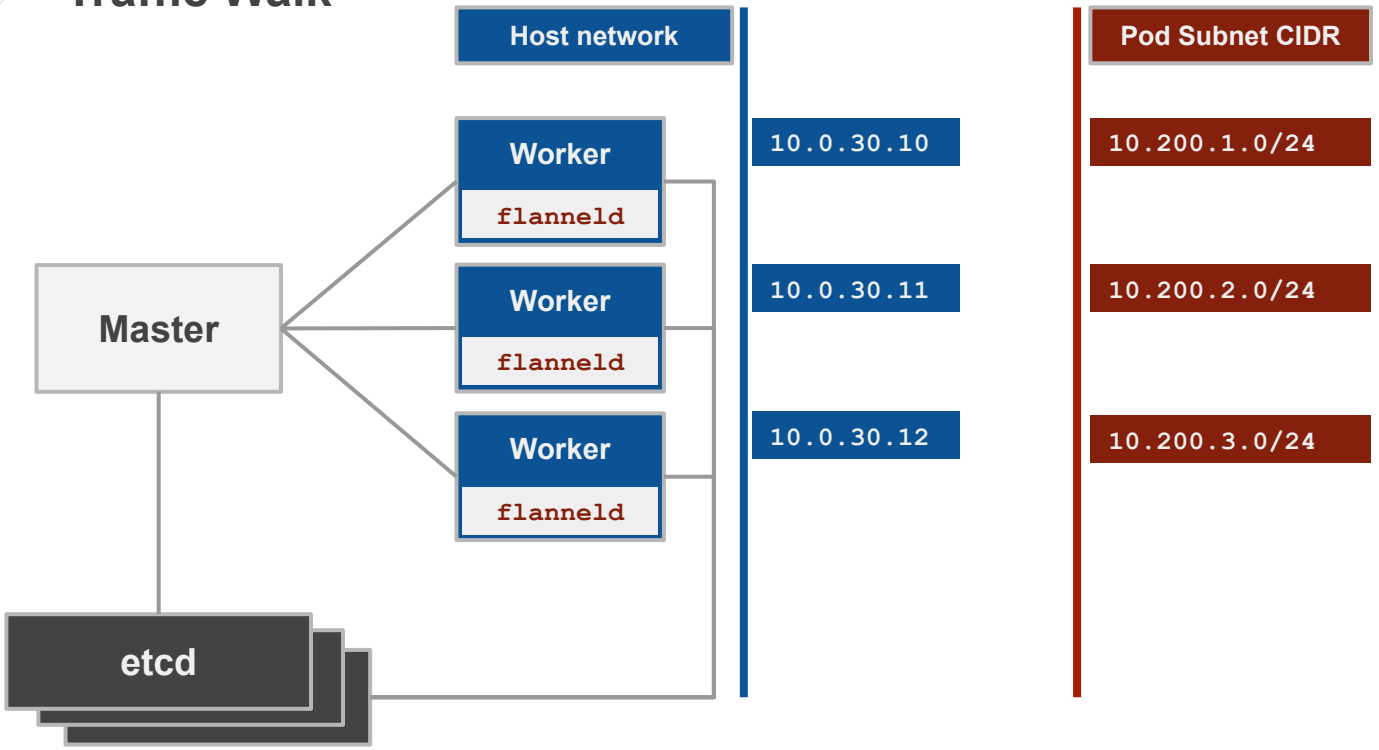
# Kubernetes Network Model Traffic Walk
# Flannel (VXLAN encapsulation)



# Kubernetes Network Model Traffic Walk
# Egress outside of cluster flow



# Kubernetes Network Model
# Traffic Walk



# Pod Network Considerations

- **IPAM -** Host local subnets allocated per-worker-node;
  or custom topologies (e.g. subnet per Kubernetes namespace) if your CNI supports
- **CNI Plugin** - Configures host-level connectivity;  potentially external SDN controller routes to pods
- **Cloud Provider** - Configures cluster-wide routes and/or firewall rules in external SDN controller
- **Policy** - Kubernetes NetworkPolicy (optional!) configures an east-west distributed firewall on the cluster via OVS flows, IPtables, eBPF program
- **Ingress to the cluster**
  - Typically handled through the service network.  But if pods externally routable….
- **Egress from the cluster**
  - If Pods are not externally routable, how is the traffic SNATed?
    Per worker node?  Some other topology (e.g. per namespace?)

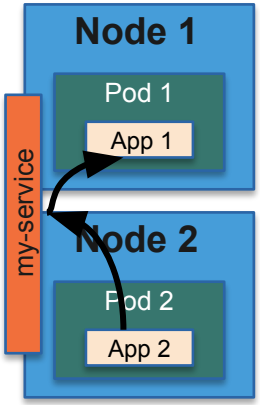
# Questions from the audience

# Understanding Services

# What is a Kubernetes Service?

- Persistent, stable **identifier** for a networked service
- Select pods, or alias to an external DNS entry
- 100% logical concept (except ext. LB)
- Target pods identified using ***selectors***
- Spec also includes
  - List of ports objects
  - Service type
  - Any attributes associated with Service type

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
  selector:
    app: app1
```



# Service Discovery

- ***Services*** discovered via DNS or env vars
- Every Pod receives a set of env vars for each created service

```
MY_SERVICE_HOST=10.100.200.106
MY_SERVICE_PORT=80
MY_SERVICE_PORT=tcp://10.100.200.106:80
MY_SERVICE_PORT_80_TCP=tcp://10.100.200.106:80
MY_SERVICE_PORT_80_TCP_PROTO=tcp
MY_SERVICE_PORT_80_TCP_PORT=80
MY_SERVICE_PORT_80_TCP_ADDR=10.100.200.106
```

- DNS publishes A records and SRV records of all created services to all pods
  - Client DNS search automatically includes own namespace and default cluster domain

```
Name:      my-service
Address 1: 10.100.200.106 my-service.default.svc.cluster.local

Same namespace:          "my-service"
Different namespace:     "my-service.default"
Or Different namespace: "my-service.default.svc.cluster.local"
```

# Services mapped

- Services published externally by ***Service Types***

- type: ClusterIP (default)
  Service assigned service-network VIP that DNATs to one or more Pods.

- type: Headless (type: none)
  DNS-only registration, per-pod and per-service. No VIP

- type: NodePort
  Static DNAT port mapped on each node's IP. ClusterIP automatically created. Access by requesting <NodeIP>:<NodePort>

- type: LoadBalancer
  Exposes service externally using Cloud Provider's LB. A NodePort and ClusterIP automatically is created. Access by requesting <ExternalIP>:<ServicePort>

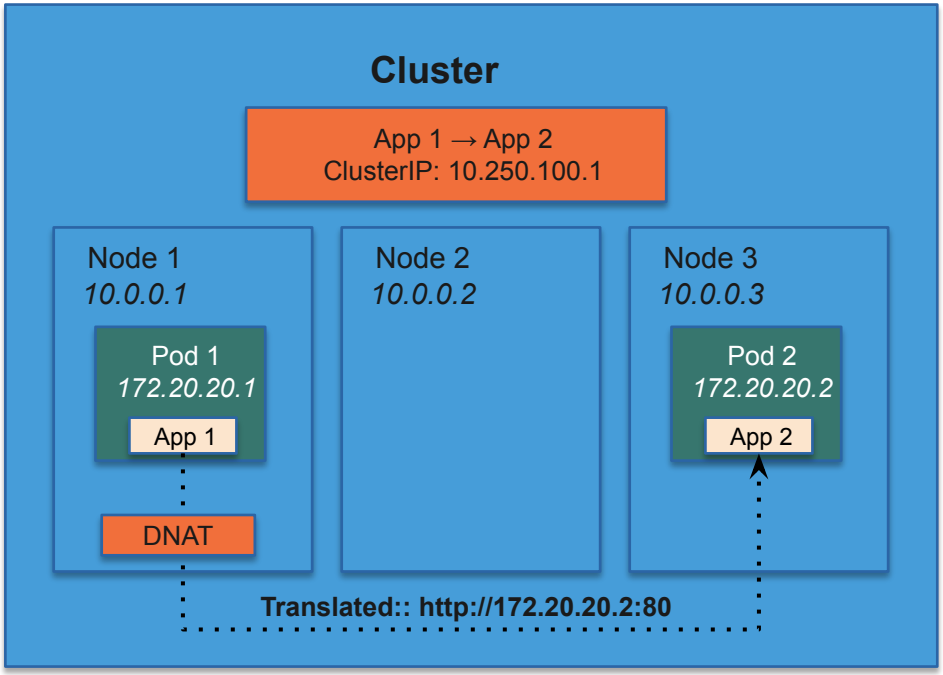
- type: ExternalName
  Maps service to the IP or DNS specified in ExternalName field. CNAME record created. Access by requesting Service DNS A Record

# Services - ClusterIP

- Service assigned cluster-internal VIP via **Kube-Proxy** (IPVS or IPtables) or **custom SDN controller agent**
- IPtables uses randomized load balancing with “recent match” tracking to improve balancing
- Optional **externalIP**: is available for adding externally routable aliases (if your CNI supports it)

```
apiVersion: v1
kind: Service
metadata:
 name: my-service
spec:
 ports:
 - targetPort: 80
   port: 8080
 selector:
   app: app2
```

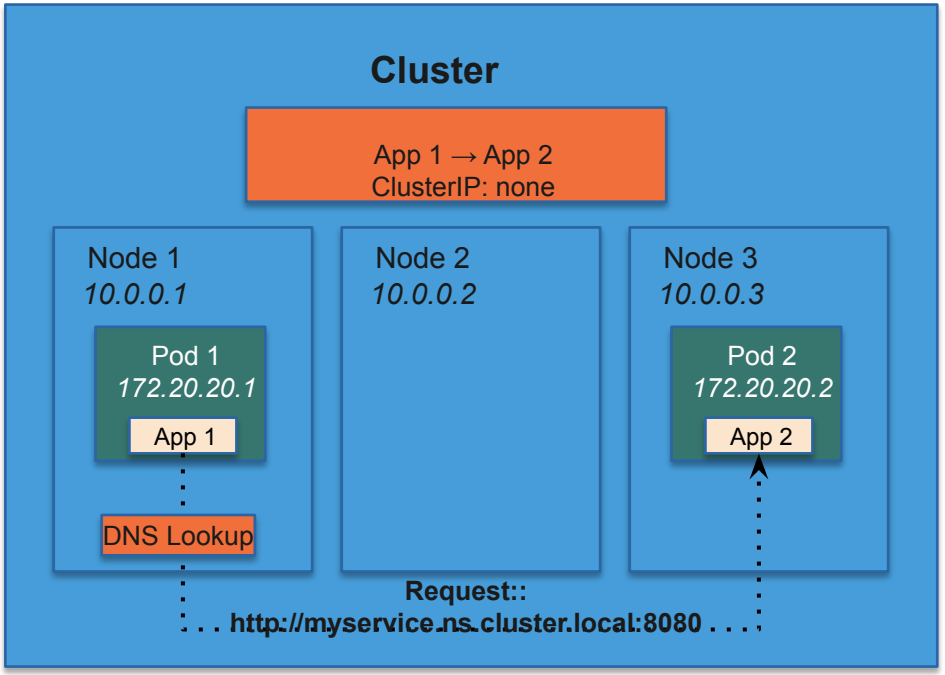
**Request::**
**http://10.250.100.1:8080**



# Services - Headless

- DNS entries only  (and also the only way to get per-pod DNS)
- Useful for data services & database clusters (e.g. Kafka, Cassandra, etc.)

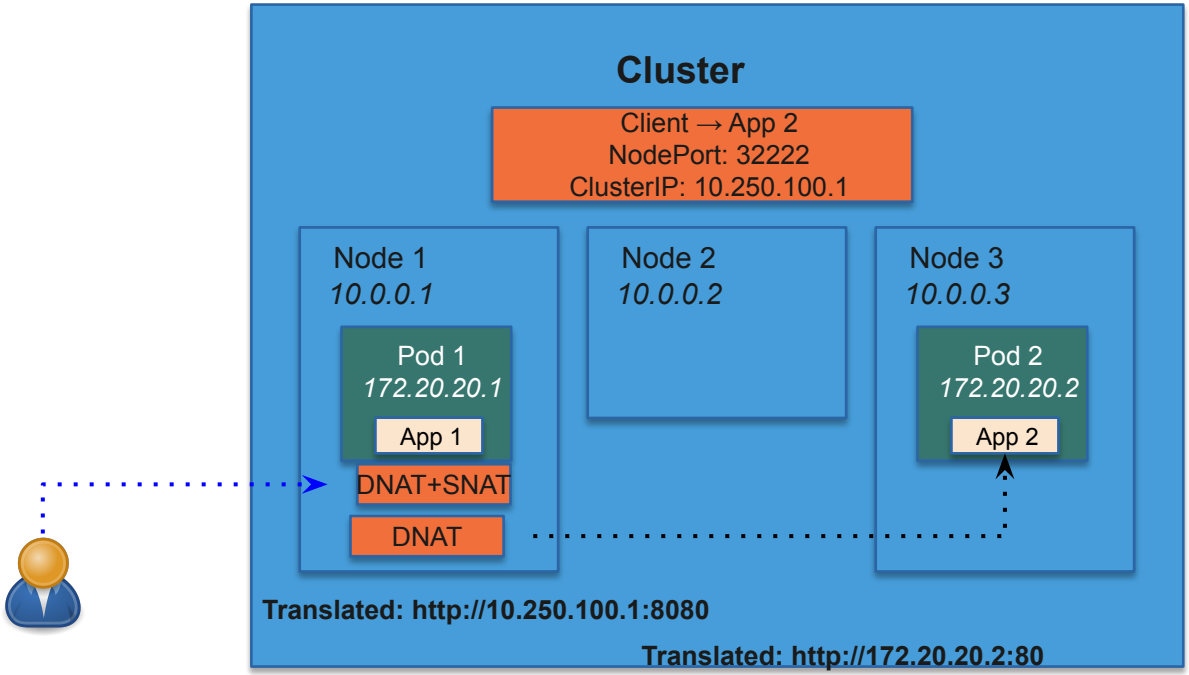
```yaml
apiVersion: v1
kind: Service
metadata:
 name: my-service
spec:
 clusterIP: none
 ports:
 - targetPort: 80
   port: 8080
 selector:
   app: app2
```



# Service - NodePort

- Static port mapped on each node's IP.  ClusterIP automatically created.
- Access by requesting <NodeIP>:<NodePort>
- **externalTrafficPolicy: Cluster** (default) SNATs to the node that is hit

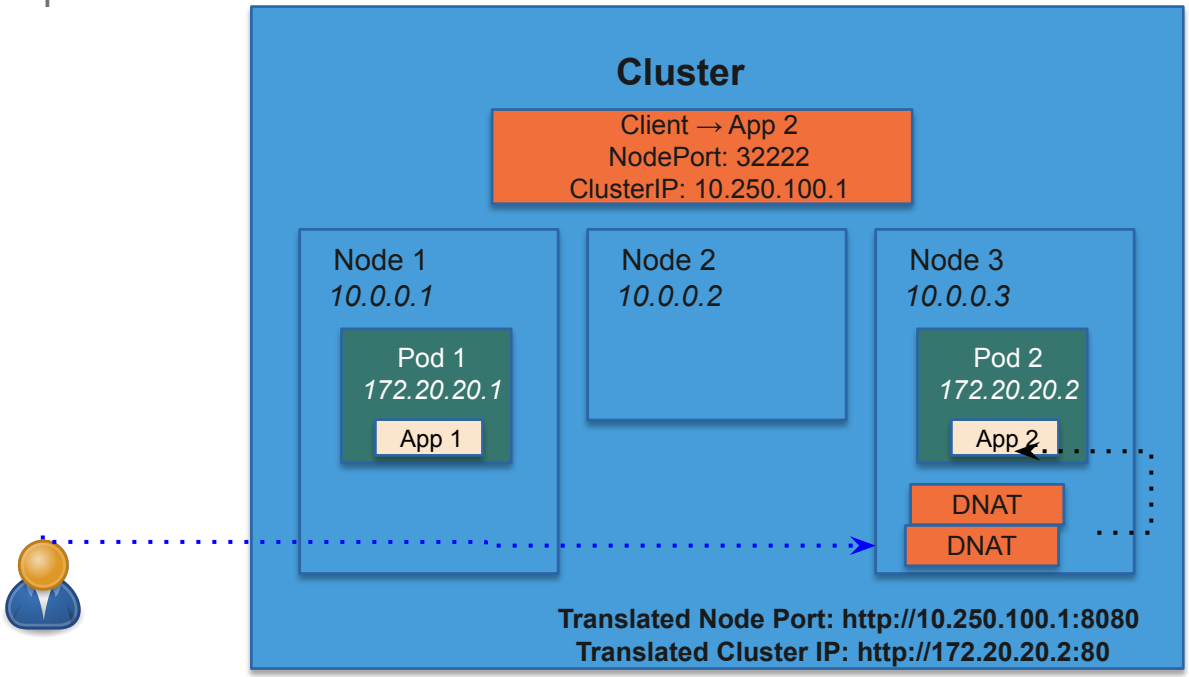
```
apiVersion: v1
kind: Service
metadata:
 name: my-service
spec:
 ports:
 - targetPort: 80
   port: 8080
   nodePort: 32222
 type: NodePort
 selector:
   app: app2
```



# Service - NodePort

- **externalTrafficPolicy: Local** ensures only local pods are hit
- Preserves source IP addresses
- Requires there to be a local pod!

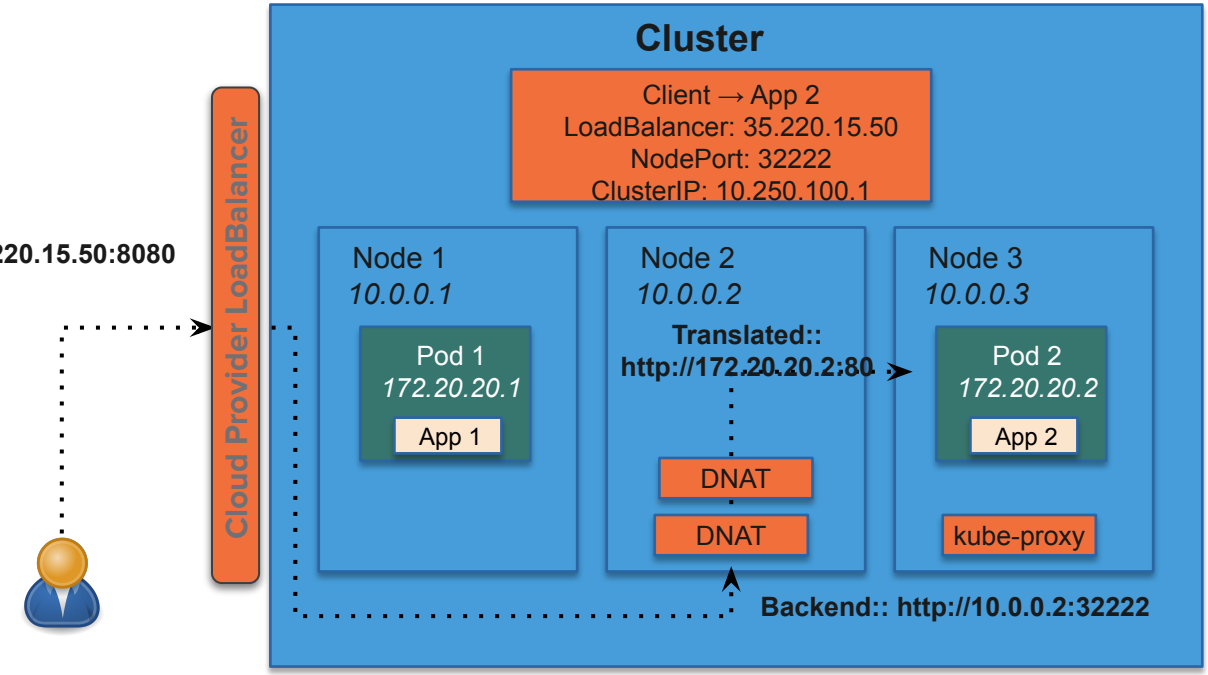
```
apiVersion: v1
kind: Service
metadata:
 name: my-service
spec:
 ports:
 - targetPort: 80
   port: 8080
   nodePort: 30008
 type: NodePort
 selector:
   app: app2
```



# Service - LoadBalancer

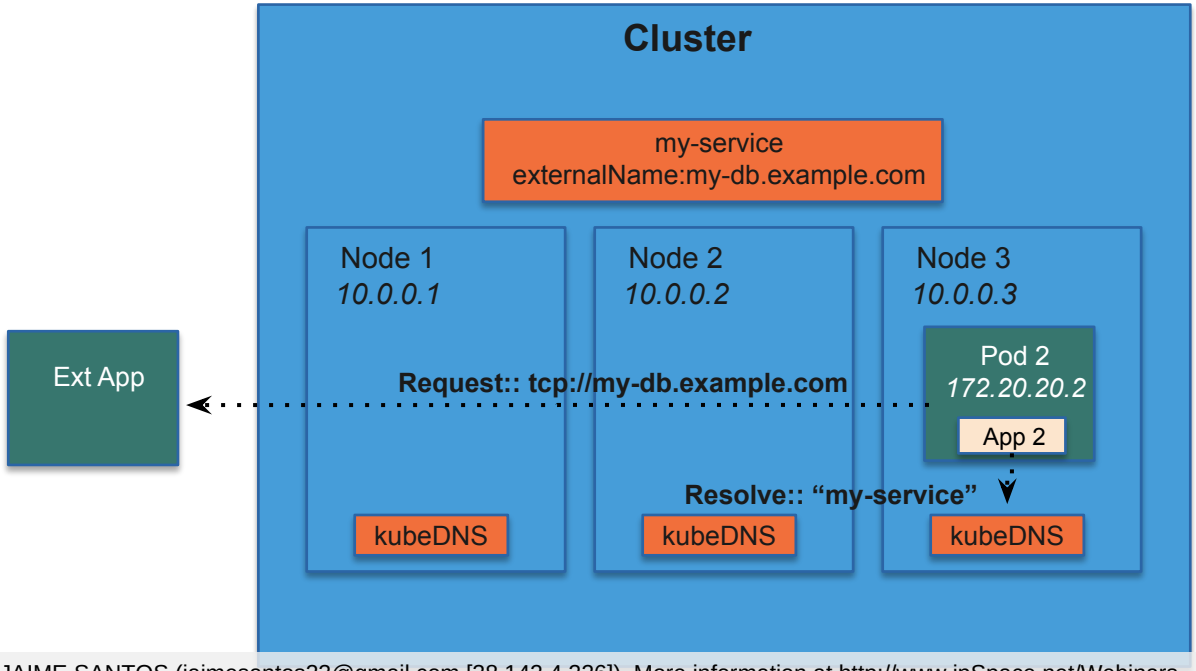
- type: LoadBalancer
  Exposes service externally using Cloud Provider's LB (AWS ELB). A NodePort and ClusterIP automatically create. Access by requesting <ExternalIP>:<ServicePort>
- ***Could go direct to pods and avoid the double DNAT if the Pod network is externally routable***

```
apiVersion: v1
kind: Service
metadata:
 name: my-service
spec:
 ports:
 - port: 8080
   targetPort: 80
 type: LoadBalancer
 selector:
   app: app2
```



# Services - External Aliases

- type: ExternalName
  Maps service to the IP or DNS specified in externalName field. CName record created. Access by requesting Service DNS A Record
- A way to return an alias to an external service residing outside the cluster.



# Services internals: considerations

- The service network is largely realized on the worker nodes only via IPtables DNAT or IPVS virtual IPs (or custom replacement kube-proxy mechanisms)
- Typically routable only within the Pod and Host network
- **Can be advertised / made routable externally** with some CNIs (e.g. via BGP host routes , or via advertising the entire service network)
    - E.g. All worker nodes with a selected pod for a service would be advertised as the ECMP next hop

More on these variants in the section on Ingress and Load Balancing

# Questions from the audience

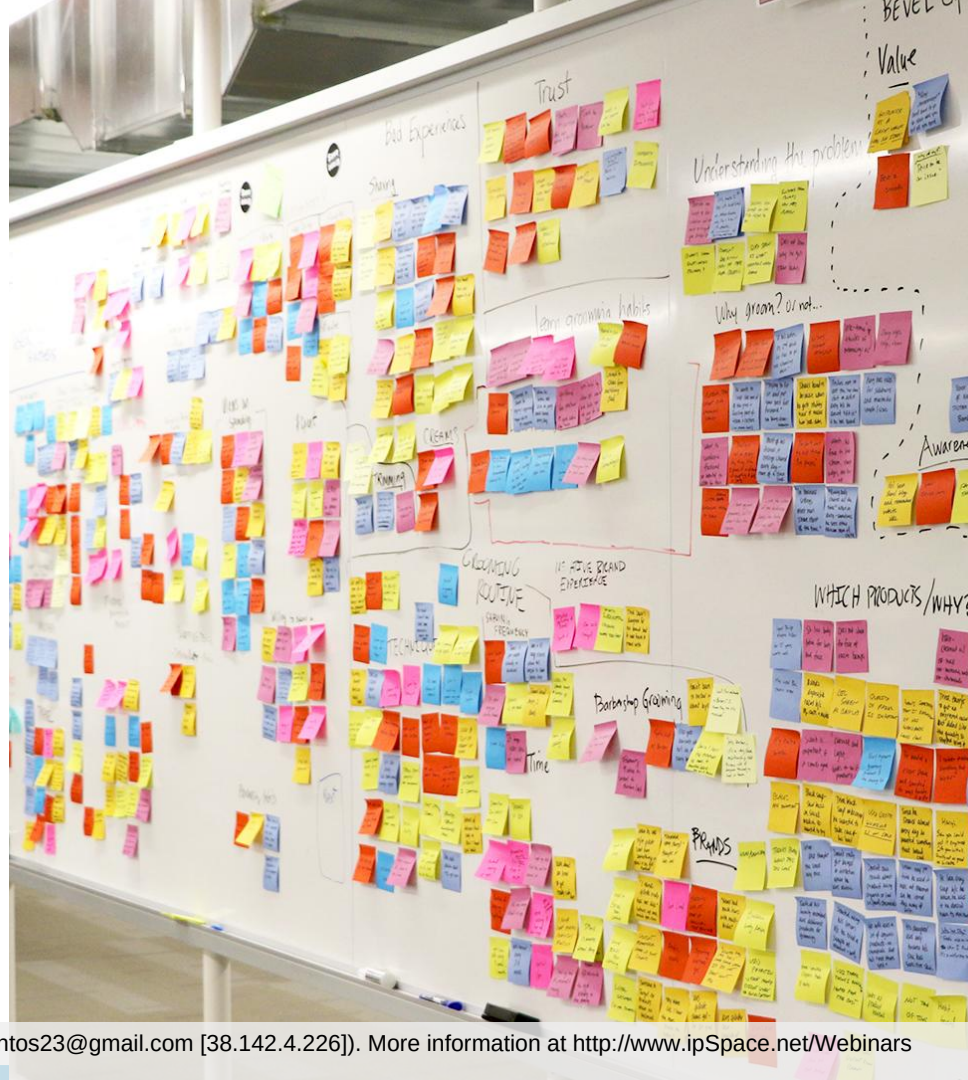
# Kubernetes Networking Deep-Dive

Stuart Charlton
January-February 2021

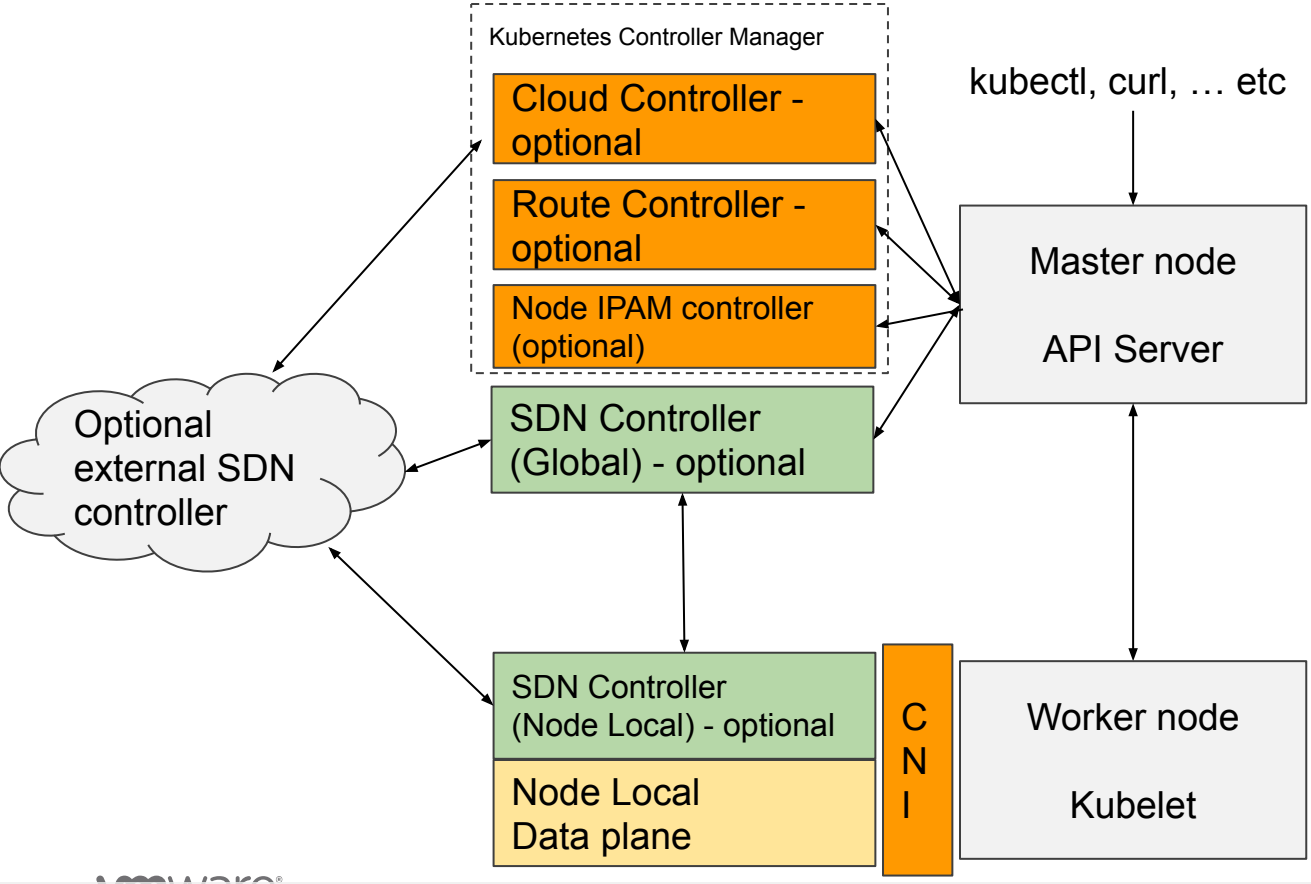
# Part 3

**A Dive into Kubernetes Container Networking internals:**

- An overview of Kubernetes SDN abstractions and Container Networking Interface (CNI) standard
- Exploring how a pod gets its network in Flannel
- Examining  Calico
- Overview of Kubernetes Network Policy
- Survey of many Public/Private Kubernetes SDN configurations

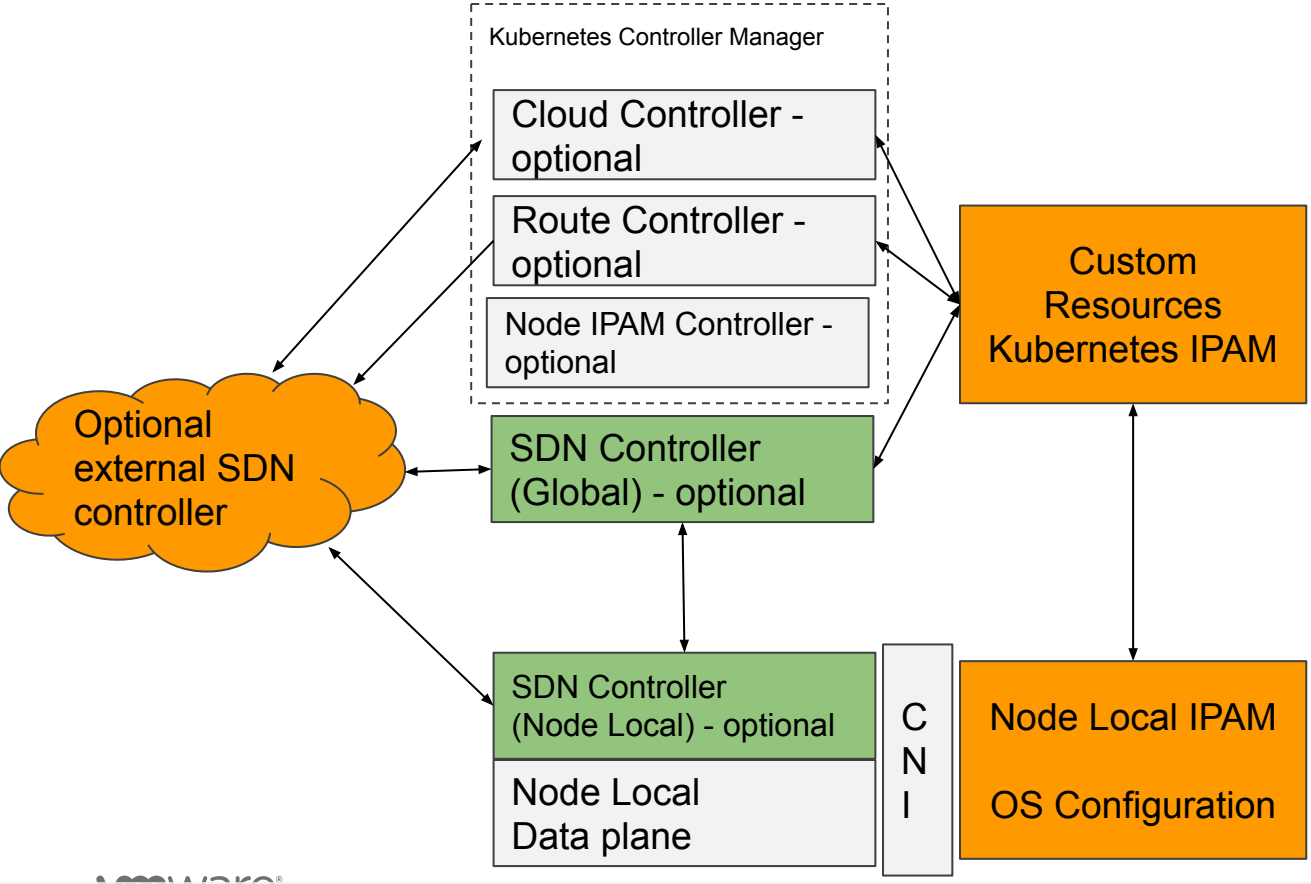


# Understanding Kubernetes SDN



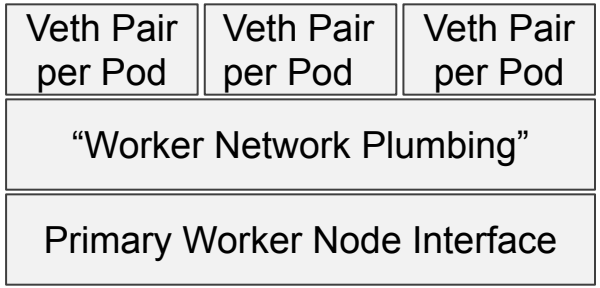

- Kubernetes networking requires interaction with software-defined networking (SDN) functions & abstractions
  - No flooding & learning, routing protocols optional
  - Not necessarily OpenFlow or any particular standard
- Kubernetes includes standard interfaces to SDN
  - **CNI**: Container Networking Interface, a CLI-based interface from Kubelet to node-local SDN and data plane
  - **Route Controller:** Optionally plumbs external SDN routes for pod network
  - **Cloud Controller:** Typically only for network services (firewalls, LBs)

# How Kubernetes SDN Stores Control Data




- Management plane state is typically managed by
  - Kubernetes API custom resources
  - Optionally an **External SDN**
  - Optionally **custom static SDN / CNI** configuration
- Control plane state is managed by
  - Kubernetes controller manager has an optional **IPAM interface** for allocating **pod subnets** to **nodes**
  - Optionally **SDN brings its own database** (e.g. etcd)
  - CNI has optional **node local IPAM**
  - Optionally an External SDN

# Typical Network Data Path



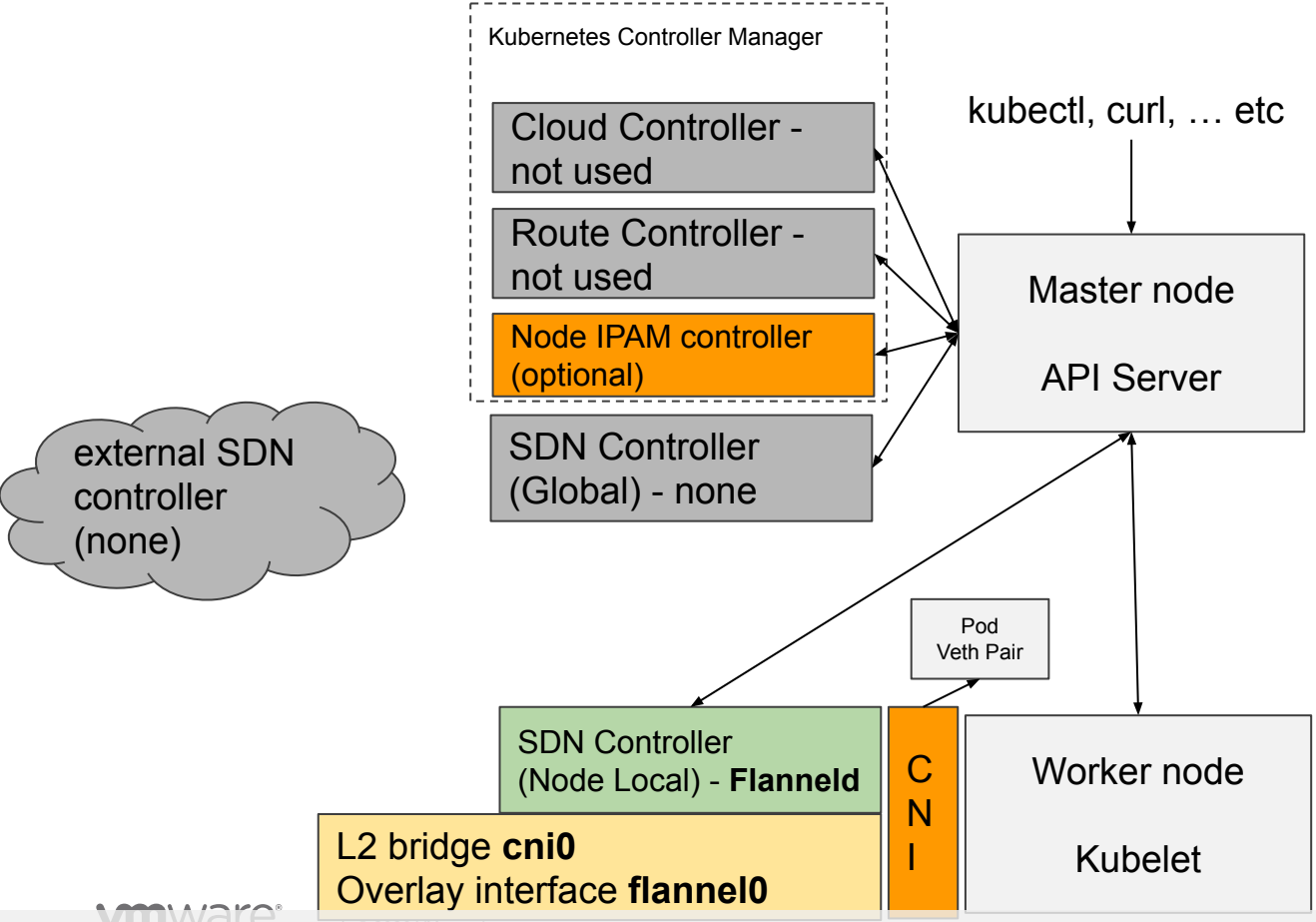

- Typically pods have a single interface, and worker nodes have a single interface
- Worker network plumbing varies depends on your SDN abstraction
  - L2 **bridged** interface node-local
  - L3 **routed** interface node-local or external
  - **Open vSwitch (OVS)** interfaces
  - **eBPF** (extended Berkeley Packet Filter)
  - Overlay networking encapsulation (**GENEVE, VXLAN)** for external pod network communication (or not)
  - **SNAT/Masquerade** for external egress (or not) via IPTables, or eBPF, or OVS
- Pure IPv6 (beta) or Pure IPv4 interfaces supported in Kubernetes 1.20+
- Dual-stack IPv4/IPv6 in alpha as of 1.20

# Questions from the audience

# Example Kubernetes SDN Configurations

1. **Flannel** (node-local SDN controller & CNI plugin)
2. **Calico** (optional global + node-local SDN controller & CNI plugin)
3. **Cilium** (global + node-local SDN controller & CNI plugin)
4. **Weave** (global + node-local SDN controller & CNI plugin)
5. **Azure** (route tables and/or Azure-native CNI plugin and/or Calico, Cilium, etc.)
6. **Amazon Web Services** (route tables and Amazon-native CNI plugin, or Calico/Antrea)
7. **Google Cloud** (route tables or IP aliases, optional Calico or Cilium)
8. **Google Anthos** (Calico, eventually Cilium)
9. **NSX-T Container Plugin (NCP)** (Open vSwitch node-local + external SDN + CNI plugin)
10. **Antrea** (Open vSwitch node-local + global/node-local SDN controllers + CNI plugin)
11. **vSphere 7 with Tanzu** (Calico or Antrea w/ optional NSX integration)
12. **Red Hat OpenShift OVN** (Open vSwitch node-local + global SDN controller + CNI plugin)

# Examining Flannel as the SDN model




- Flanneld runs on nodes (as a hostNetwork pod or native daemon). Needs **Kubernetes API server** or **etcd** access for global database
- Flanneld allocates pod subnets from **host-local** config or **node IPAM controller**
- **Static routing and bridging** driven by the **flanneld** SDN controller **watching** node lifecycle events
  - Static IP Routes
  - Static ARP Entries
  - Static MAC table entries

# Flannel plumbing - Route table

ip route (on node with 10.200.35.0/24 pod network)

default via 10.0.16.1 dev eth0  proto dhcp  src 10.0.16.14  metric 1024
default via 10.0.16.1 dev eth0  proto dhcp  metric 1024
10.0.16.0/20 dev eth0  proto kernel  scope link  src 10.0.16.14
10.0.16.1 dev eth0  proto dhcp  scope link  src 10.0.16.14  metric 1024
**10.200.35.0/24** dev **cni0**  proto kernel  scope link  src 10.200.35.1
**10.200.48.0/24 via 10.200.48.0 dev flannel.1 onlink**
**10.200.75.0/24 via 10.200.75.0 dev flannel.1 onlink**
**10.200.78.0/24 via 10.200.78.0 dev flannel.1 onlink**

# Flannel plumbing - ARP table

ip neighbor (on node with 10.200.35.0/24 pod network)

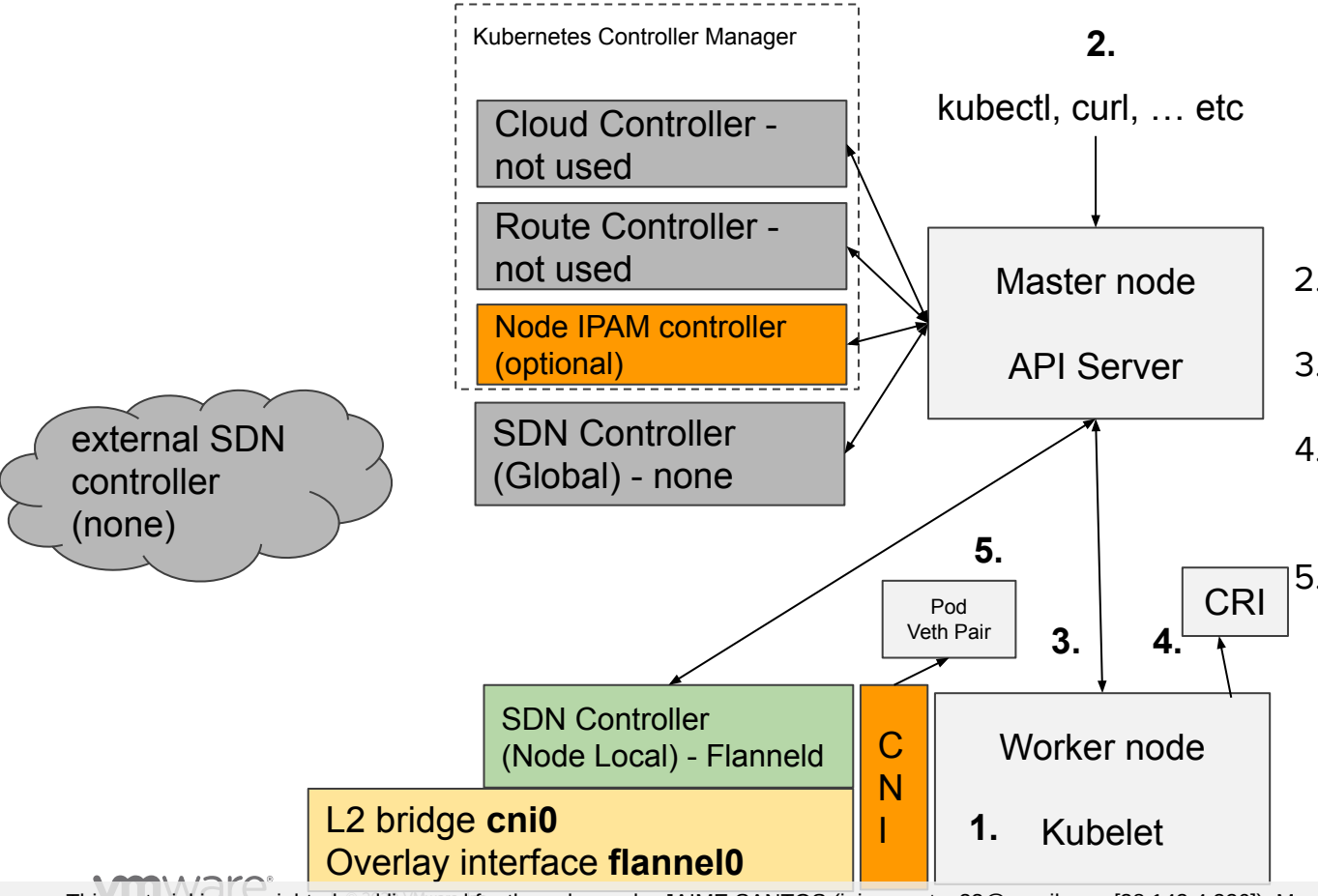
**10.200.78.0 dev flannel.1 lladdr ba:d8:c7:23:92:d9 PERMANENT**
10.200.35.24 dev **cni0** lladdr aa:a4:3b:e9:a9:78 REACHABLE
10.200.35.23 dev **cni0** lladdr 1a:96:1c:17:64:d5 REACHABLE
10.200.35.18 dev **cni0** lladdr f2:03:f4:21:70:e2 REACHABLE
**10.200.48.0 dev flannel.1 lladdr 92:5b:65:01:ef:09 PERMANENT**
10.200.35.28 dev **cni0** lladdr 7a:a1:93:5c:bc:ea REACHABLE
10.200.35.27 dev **cni0** lladdr 56:c6:7a:31:f4:52 DELAY
**10.200.75.0 dev flannel.1 lladdr 5a:f1:a5:73:e4:b8 PERMANENT**
10.0.16.6 dev eth0 lladdr 12:34:56:78:9a:bc REACHABLE
10.0.16.1 dev eth0 lladdr 12:34:56:78:9a:bc REACHABLE
10.0.16.12 dev eth0 lladdr 12:34:56:78:9a:bc REACHABLE
10.0.16.15 dev eth0 lladdr 12:34:56:78:9a:bc REACHABLE
10.0.16.4 dev eth0 lladdr 12:34:56:78:9a:bc REACHABLE

# Flannel plumbing - MAC Forwarding table

bridge fdb show

01:00:5e:00:00:01 dev eth0 self permanent
**# These are static MAC entries that point to VXLAN VTEP IPs**
**2a:24:3e:dd:58:86 dev flannel.1 dst 10.0.16.15 self permanent**
**ba:d8:c7:23:92:d9 dev flannel.1 dst 10.0.16.13 self permanent**
**5a:f1:a5:73:e4:b8 dev flannel.1 dst 10.0.16.15 self permanent**
**92:5b:65:01:ef:09 dev flannel.1 dst 10.0.16.12 self permanent**
01:00:5e:00:00:01 dev **cni0** self permanent
5e:08:f6:a1:40:91 dev **cni0** master cni0 permanent
5e:08:f6:a1:40:91 dev **cni0** vlan 1 master cni0 permanent
f2:03:f4:21:70:e2 dev veth664b0743 master cni0
66:3d:64:01:b7:59 dev veth664b0743 master cni0 permanent
66:3d:64:01:b7:59 dev veth664b0743 vlan 1 master cni0 permanent
01:00:5e:00:00:01 dev veth664b0743 self permanent

# How a Flannel Pod gets its Network

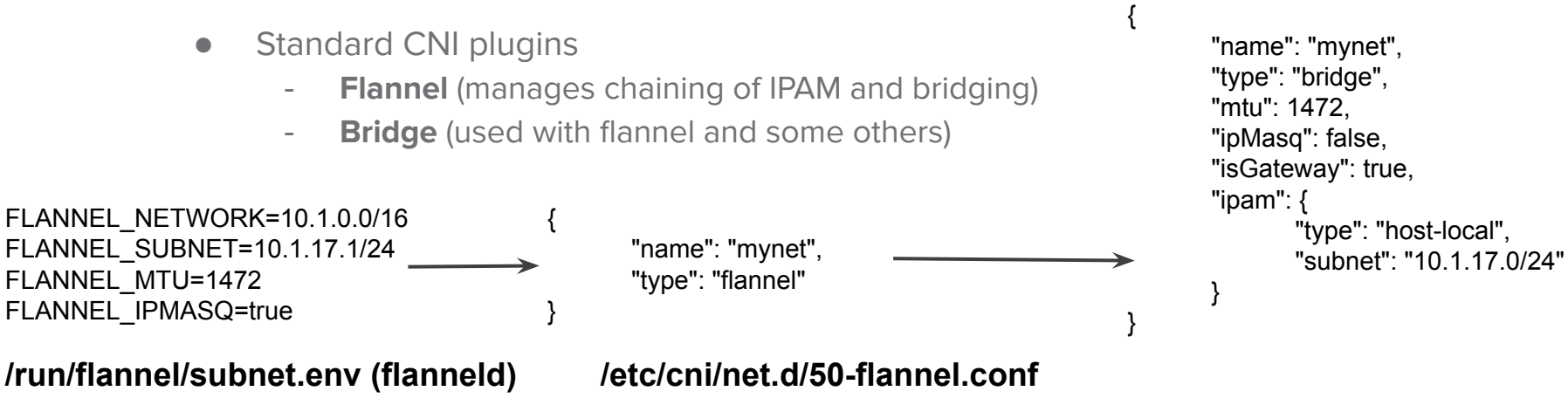



1. Node appears or disappears; **flanneld** on all other nodes reconfigure the Route, ARP and MAC tables. Flanneld on new nodes creates tunnel interface **flannel0** and plumbs its tables.
2. A **Pod** is created
3. **Pod** is scheduled and Kubelet reacts
4. Kubelet calls the CRI to **create the container**
5. Kubelet calls CNI **flannel** plugin to query IPAM, then flannel **calls** the **bridge** CNI plugin which creates the **veth pair** with one half in the network namespace; **bridges** the other half to **cni0** (which it also creates if missing)

# Questions from the audience

# What does the Container Networking Interface consist of?

- CNI plugins are **command-line executables** invoked with stdin/stdout JSON
- Must be installed on nodes via Pod **hostPath** or external package management
- CNI plugins are **chained**, i.e. they can call each other, or be **invoked in sequence** by Kubelet
- Standard CNI plugins
  - **Flannel** (manages chaining of IPAM and bridging)
  - **Bridge** (used with flannel and some others)

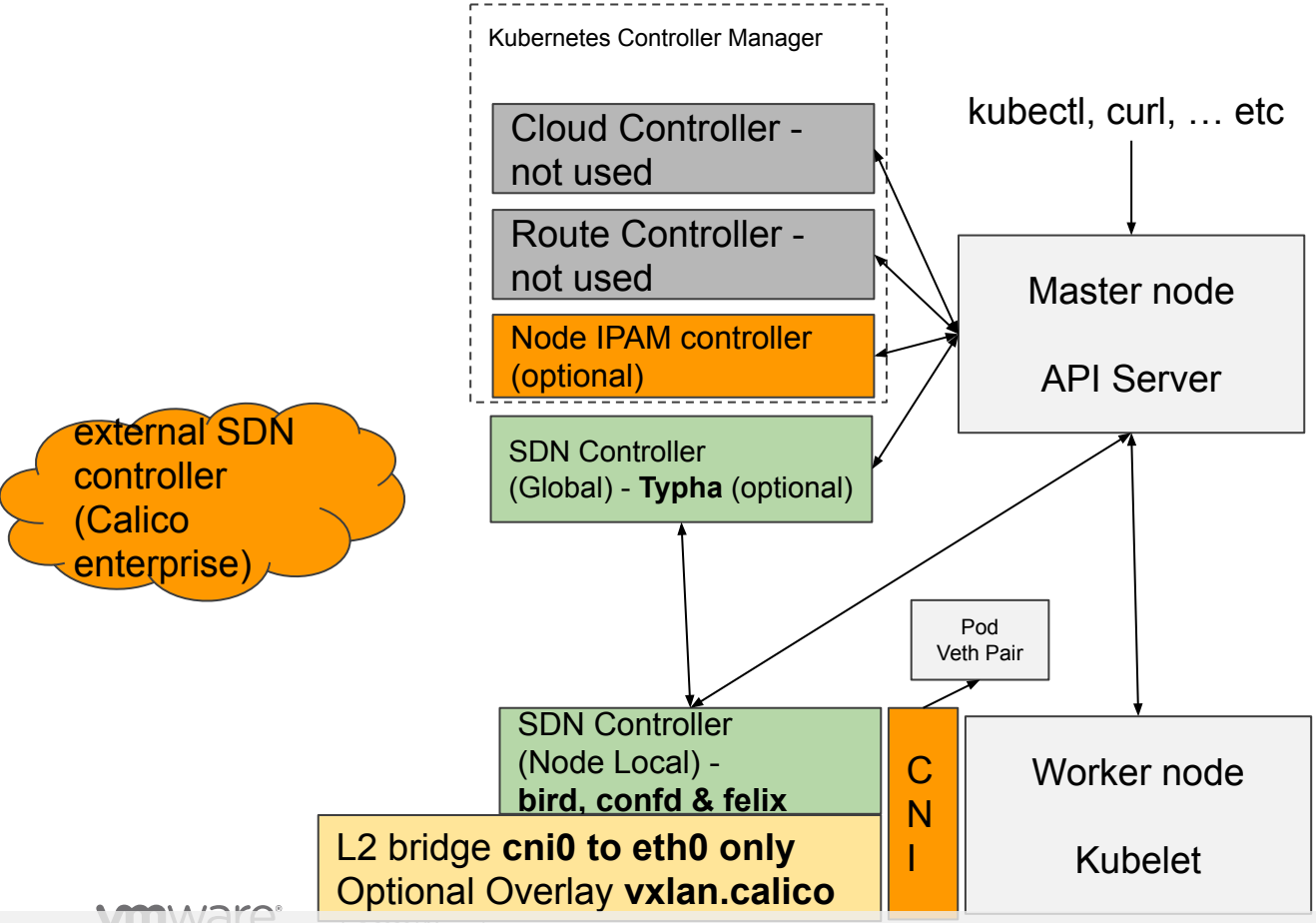


# Further CNI plugins

- Special purpose CNI plugins
  a. **Ipvlan** (Linux kernel feature, inspects IP in packet to select interface)
  b. **Macvlan** (Linux kernel feature, virtual MAC addresses bridged to primary MAC)
  c. **Portmap** (supports **hostPort** services and **static port forwarding** via iptables)
- IPAM CNI plugins
  a. **Host-local** (static IPAM configuration per node)
  b. **Dhcp** (for use with e.g. macvlan or ipvlan)
- 3rd party CNI plugins
  **a. Calico, Cilium, Weave, Azure, Amazon, OVN, Antrea, NCP**
- **Meta CNI plugins (compose many together)**
  a. **Multus CNI** (multi-NIC pods, e.g. macvlan, ipvlan, SR-IOV, DPDK, etc.)
  b. **Istio CNI** (optional use of CNI to use iptables to reroute Pod network traffic to Envoy sidecar container, avoids use of NET_ADMIN privileges required in user pod)

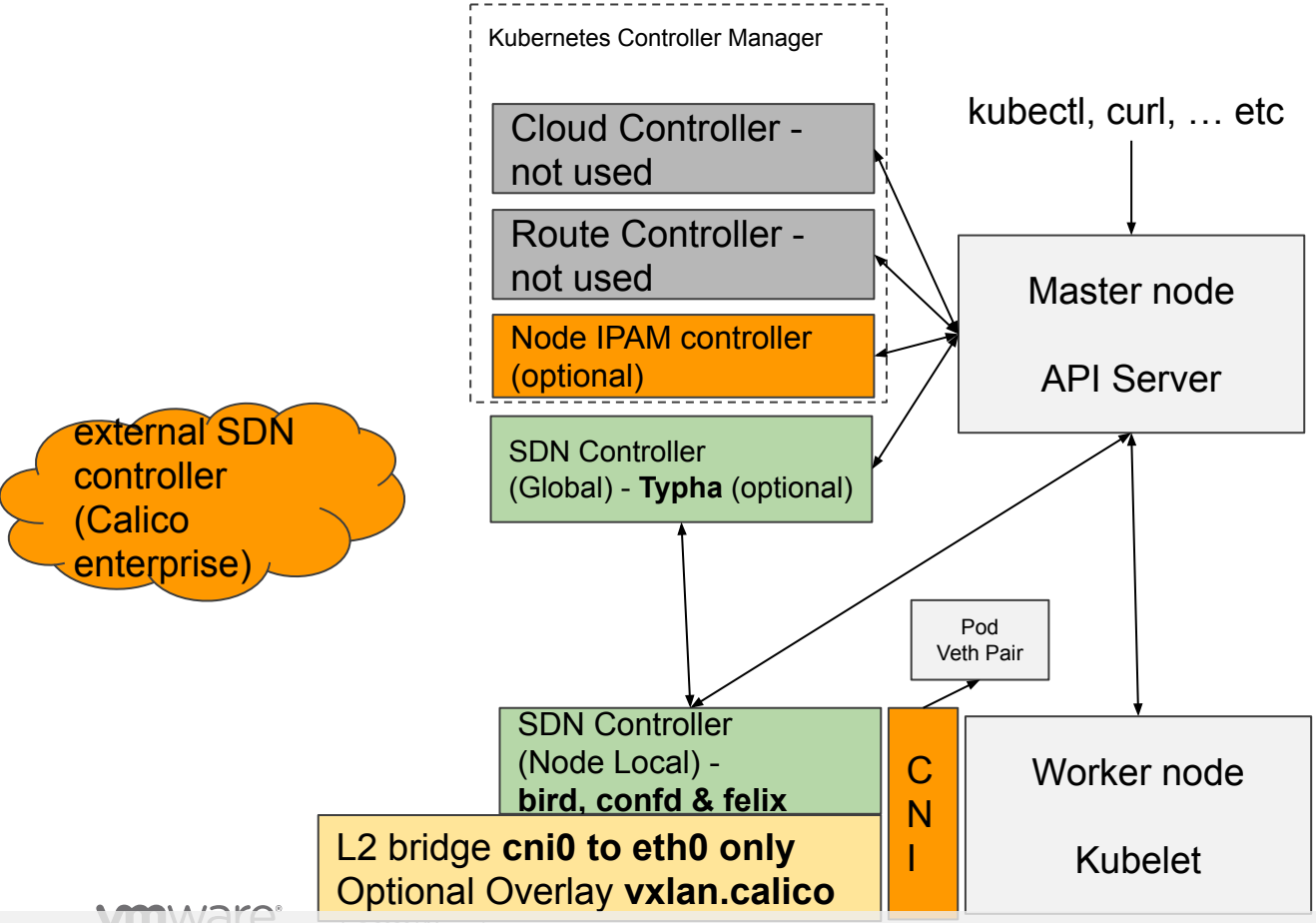
# Questions from the audience

# Examining Calico, part 1




- **Calico-node** agent (**bird, confd, and felix**) runs on nodes (as a **hostNetwork** pod), optionally creates vxlan.calico overlay inteface
- Needs **Kubernetes API server** or **etcd** access for global database. **Typha** acts as a **cache** to offload Kubernetes API server
- Calico allocates pod subnets from **host-local** config or **node IPAM controller**

# Examining Calico, part 2




- Calico CNI plugin **creates its own veth pair** to do **L3 routing only** to cni0 (no bridging), which it also creates if missing.
- CNI also plumbs network namespace **IPtables** or **eBPF** for **network policy** enforcement
- **Routes** driven by the **felix** SDN controller **watching** node lifecycle events,

  optionally **bird (BGP)** if using pure IP pod networking**,** with optional IPIP encapsulation

  optionally **VTEP entries** in **MAC and ARP** tables if using VXLAN overlay

# Questions from the audience

# Network Policy

- Ingress/egress stateful distributed firewall for pods within a namespace
- Can secure TCP and UDP ports using the standard Kubernetes API
- Somewhat application-aware, as you can define the following targets
  a. **Mandatory: Pod labels** as the **source** for the egress or **destination** for the ingress policy (within a namespace only)
  b. **Optional: Pod and/**or **namespace** labels as the other side of the ingress/egress rule
  c. **Optional: IP blocks** as the other side of the ingress/egress rule
- Cannot block a pod's traffic to itself
- **Data plane** is IPtables (Calico traditionally), OVS (NSX, Antrea, OVN), or eBPF (Cilium, Calico)
- Many CNI plugins support custom Kubernetes resources for further features
  a. Global rules across namespaces
  b. ICMP rules
  c. Richer querying languages & criteria

# Example Standard Network Policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
```

```
ingress:
 - from:
   - ipBlock:
       cidr: 172.17.0.0/16
       except:
       - 172.17.1.0/24
   - namespaceSelector:
       matchLabels:
         project: myproject

   - podSelector:
       matchLabels:
         role: frontend

   ports:
   - protocol: TCP
     port: 6379
```

```
egress:
 - to:
   - ipBlock:
       cidr: 10.0.0.0/24
   ports:
   - protocol: TCP
     port: 5978
```

# Example “Deny All” Network Policy

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

# Default “Allow All” Network Policy

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-ingress
spec:
  podSelector: {}
  ingress:
  - {}
  policyTypes:
  - Ingress
```

# Example Calico Global Network Policy

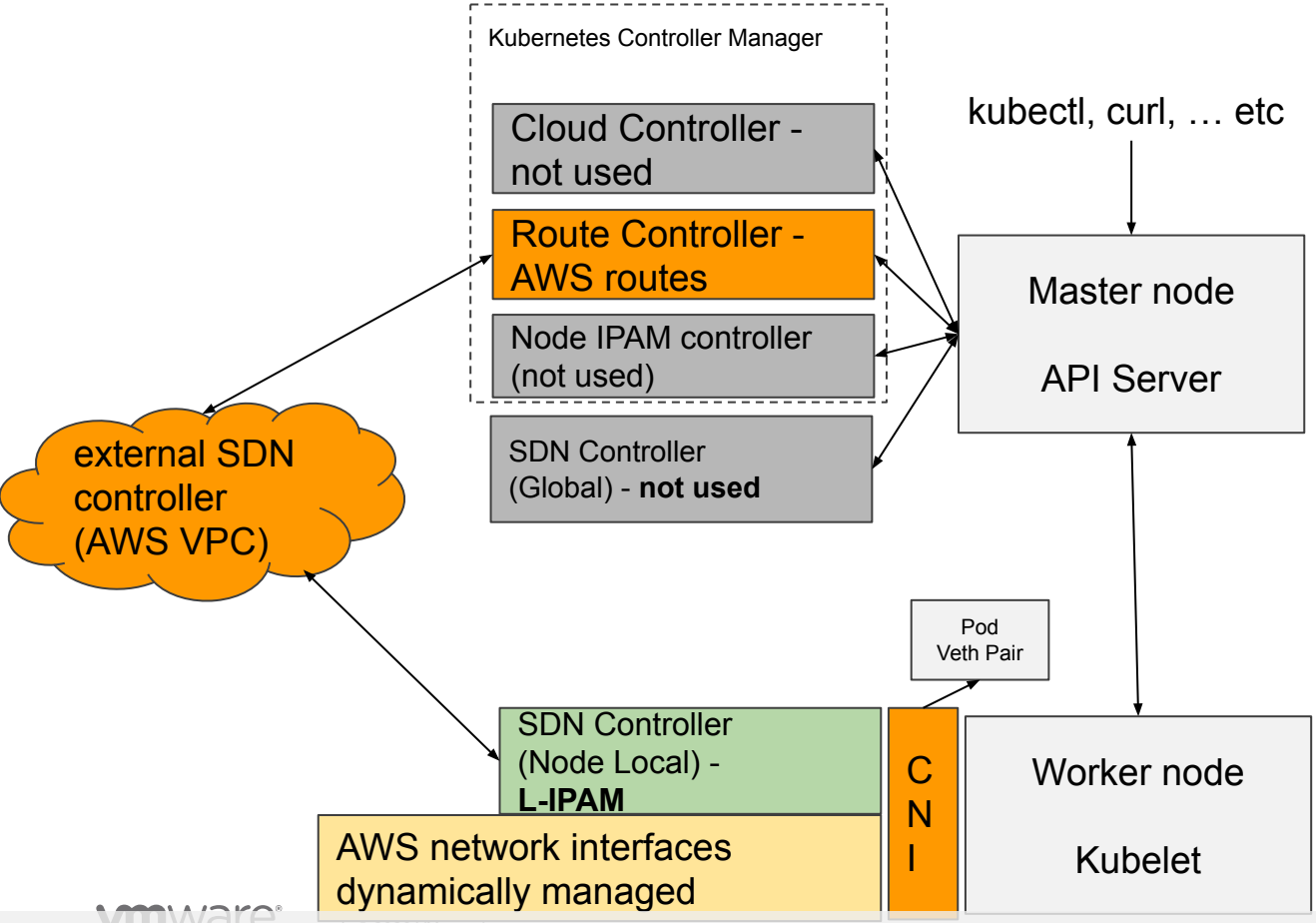
```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: allow-tcp-6379
spec:
  selector: role == 'database'
  types:
  - Ingress
  - Egress
  ingress:
  - action: Allow
    metadata:
      annotations:
        from: frontend
        to: database
    protocol: TCP
    source:
      selector: role == 'frontend'
    destination:
      ports:
      - 6379
  egress:
  - action: Allow
```

# Network Policy Support in the CNI plugins

| **Network Policy Not Supported**<br>(Requires chaining to Calico, Antrea, Cilium, OVN) | **Network Policy supported** |
|---|---|
| Amazon VPC CNI | Calico |
| Flannel | Cilium |
| Kubenet (Azure AKS default) | NSX Container Plugin (NCP) |
| | Antrea |
| | OpenShift OVN |
| | Azure CNI |
| | |

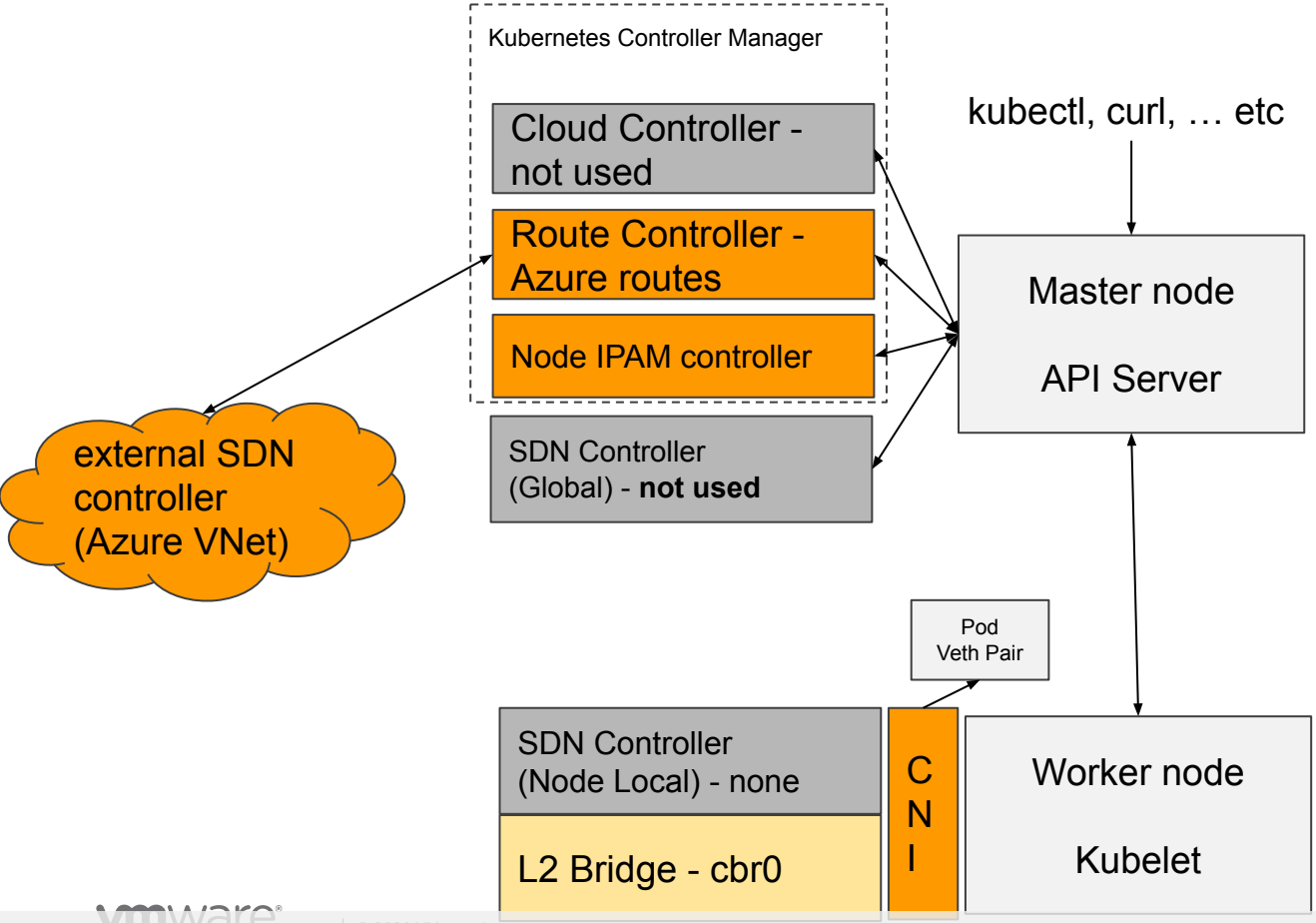
# Questions from the audience

# Examining Amazon EKS CNI



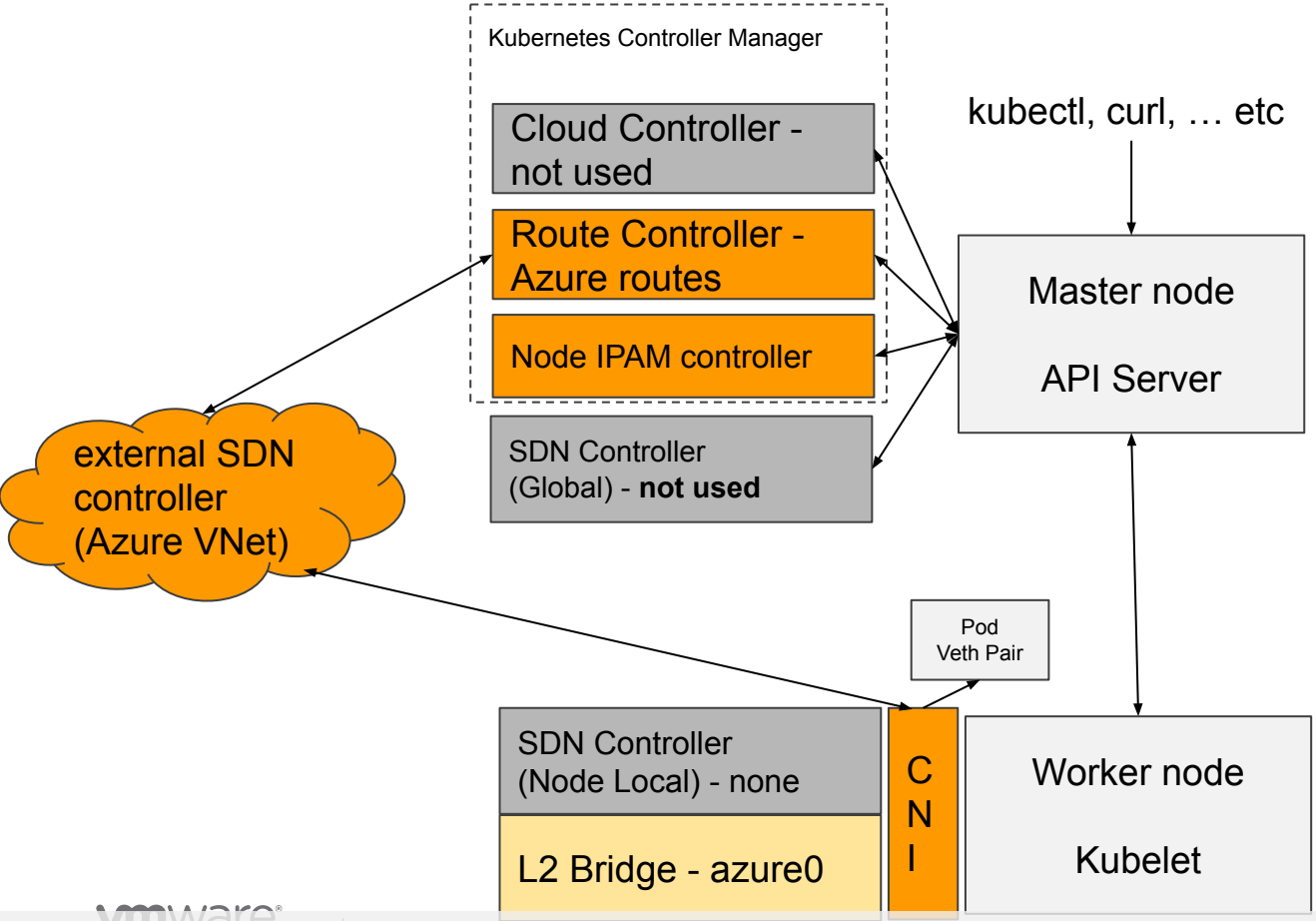

- **L-IPAM** agent runs on nodes (as a **hostNetwork** pod), dynamically creates & associates AWS network interfaces with node
- # of pods limited by number of interfaces x # of IP addresses per interface for a given instance type
- IPAM managed by AWS VPC controller
- Route controller ensures VPC sees which pods have which IPs
- Can be chained with Calico, or Antrea for policy only, or fully in overlay mode

# Examining Azure AKS Default



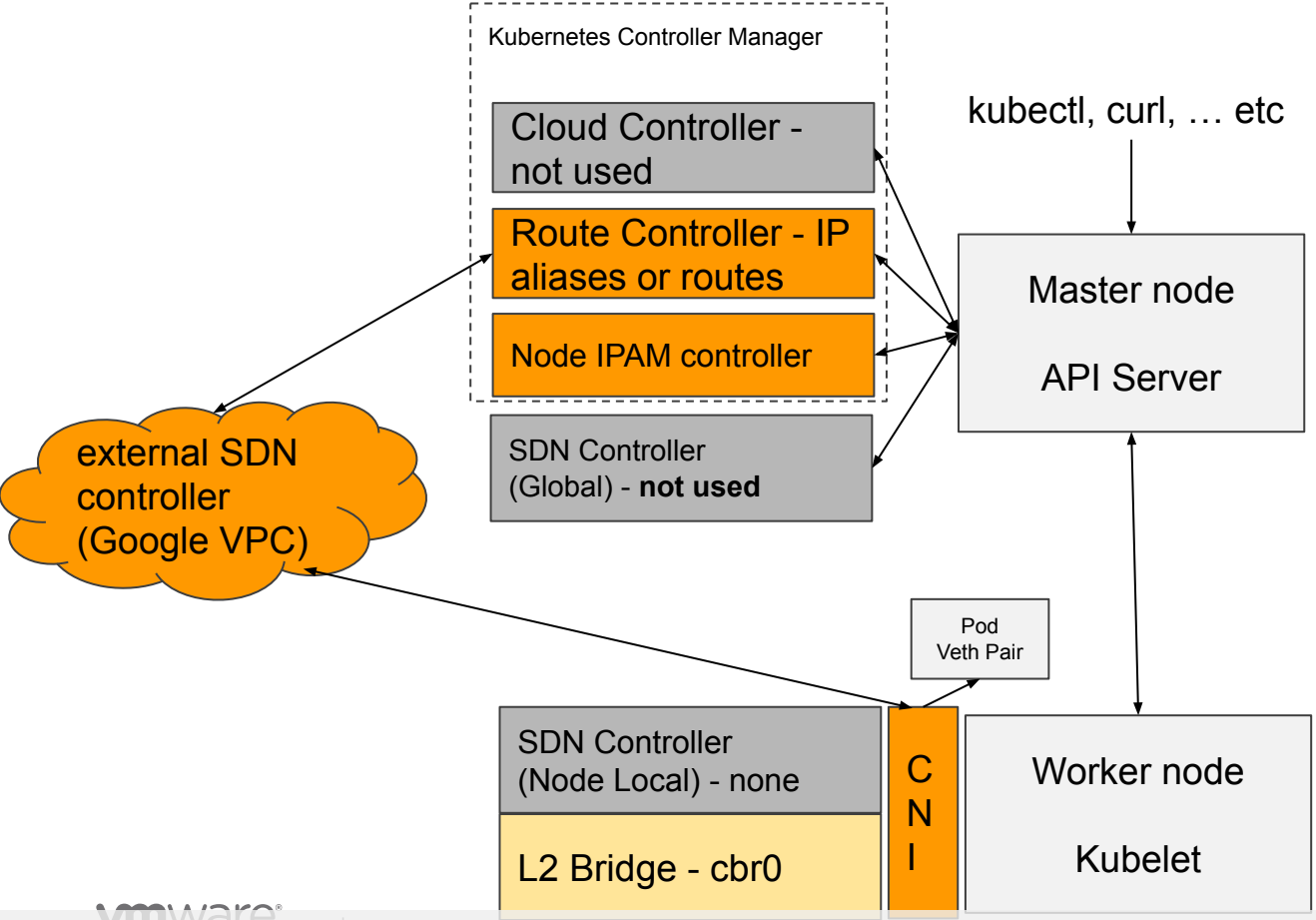

- Default model uses “old school” Kubernetes networking, Kubenet, which assumes the network “just works” when routing pod network
- Kubernetes IPAM maps to subnets of **private (not routable) pod IP ranges**
- CNI **Bridge** plugin only; IPtables SNAT for egress outside of pod network
- **Route controller** configures an Azure Route Table with user-defined routes for the pod subnets

# Examining Azure CNI




- Pods are **routable on the Azure VNet**; can use Azure security constructs (e.g. security groups)
- Azure-VNet-IPAM CNI plugin maps pod subnets per-node to subnets of routable Azure IPs
- CNI **Azure** plugin; IPtables SNAT for egress outside of pod network
- Assumes all L3 Routing for pod network is handled externally
- **Route controller** configures an Azure Route Table with user-defined routes for the pod subnets

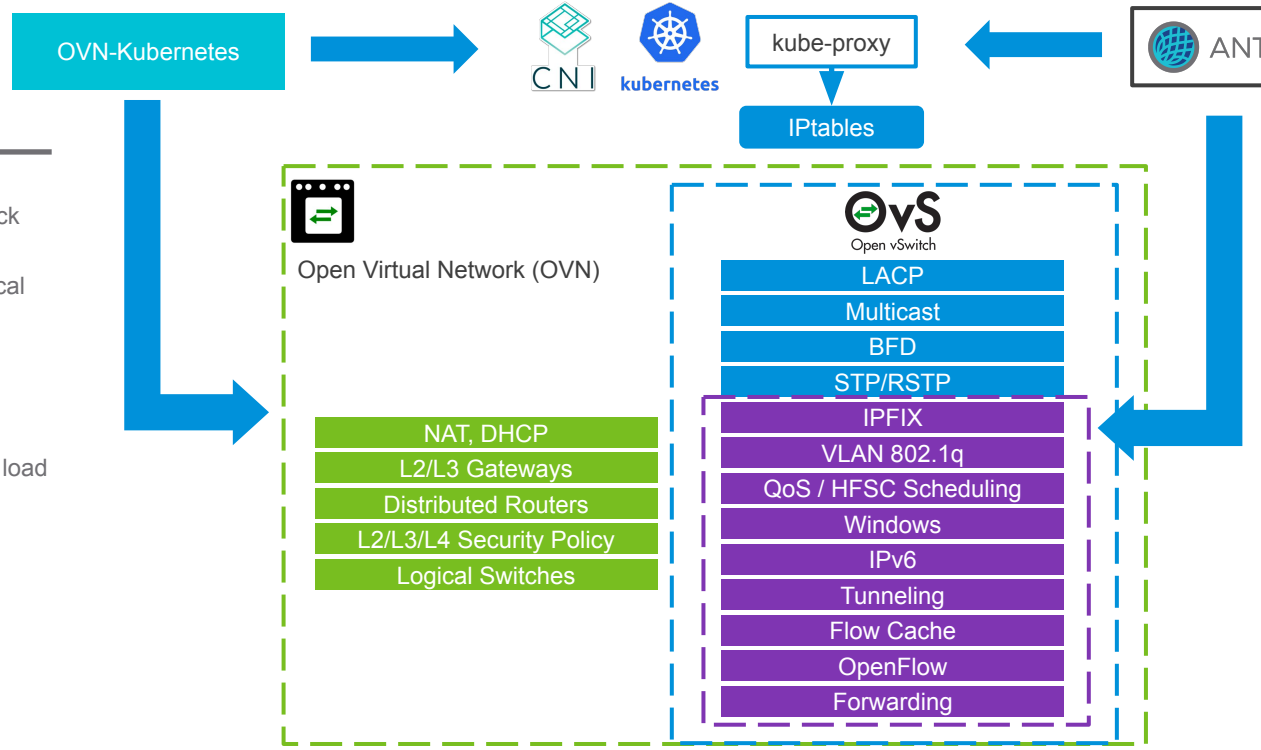
# Examining Google networking




- Kubenet by default: Pods are **routable on the Google cloud network**; can use GCP security constructs (e.g. security groups)
- CNI **Bridge** plugin only; IPtables SNAT for egress outside of pod network.
- Alternatively chain with **Calico** for policy
- Assumes all L3 Routing for pod network is handled externally
- **Route controller** configures GCP IP aliases associated with nodes (preferred), or GCP Routes in the VPC

# Open Source OVS-based CNIs

Antrea is a lightweight utilization of OVS for Kubernetes pod networking; OVN heralds from OpenStack days as full-featured SDN




Full NSX-like SDN Originally for OpenStack

OVN adapts K8S resources to OVN logical network

Control-plane is OVN

Geneve Encapsulation

Relies exclusively on kube-proxy for service load balancing

Multicast support

K8S network policy

Multus support

Light-weight

Uses only OVS bridge and flows on nodes

Control-plane is native Kubernetes controller

Multiple transit options including encapsulated and non-encapsulated

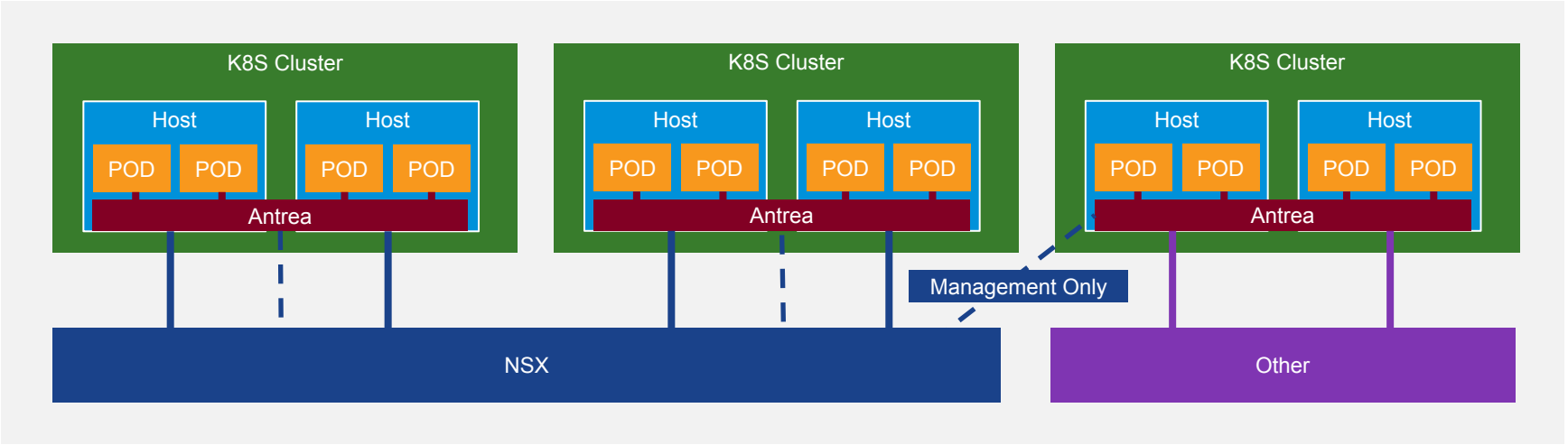
Implements service load-balancing in-line in OVS, only relies on kube-proxy for externally originated traffic

Supports policy-only chaining and Multus

K8S and advanced native network policy

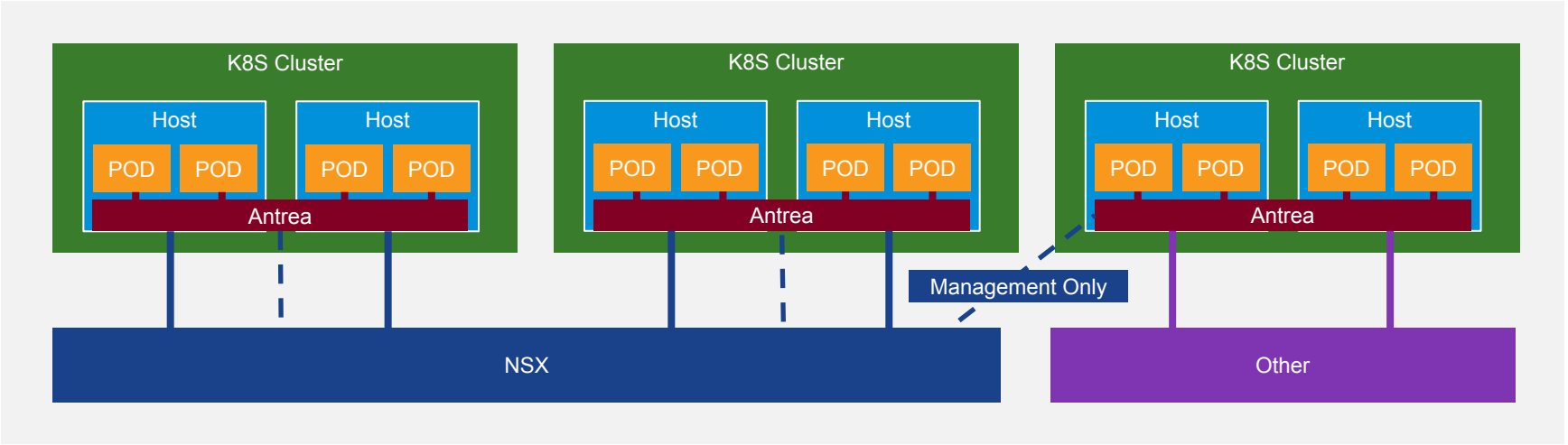
# Antrea + NSX

Antrea creates autonomous data plane for K8S clusters – only host is connected to NSX
NSX not responsible for policy enforcement or transit – efficient scale out
Antrea also works in non-NSX environments (public cloud, on premise, etc.)
NSX will be able to manage policy distribution and visibility in future NSX release



# Antrea + NSX

Antrea creates autonomous data plane for K8S clusters – only host is connected to NSX
NSX not responsible for policy enforcement or transit – efficient scale out
Antrea also works in non-NSX environments (public cloud, on premise, etc.)
NSX will be able to manage policy distribution and visibility in future NSX release



# Kubernetes Networking Deep-Dive

Stuart Charlton
May 2021

# About the Presenter


**Stuart Charlton**

scharlton@vmware.com

github.com/svrc

twitter.com/svrc


Principal Solution Engineer
Office of the CTO, Global Field

Formerly Pivotal Software, BMC Software,
IT operations & enterprise architecture executive,
consulting & training for 20+ years

REST, SOA, Java/Spring, DevOps, IT Architecture,
NSX, Kubernetes, Cloud Foundry

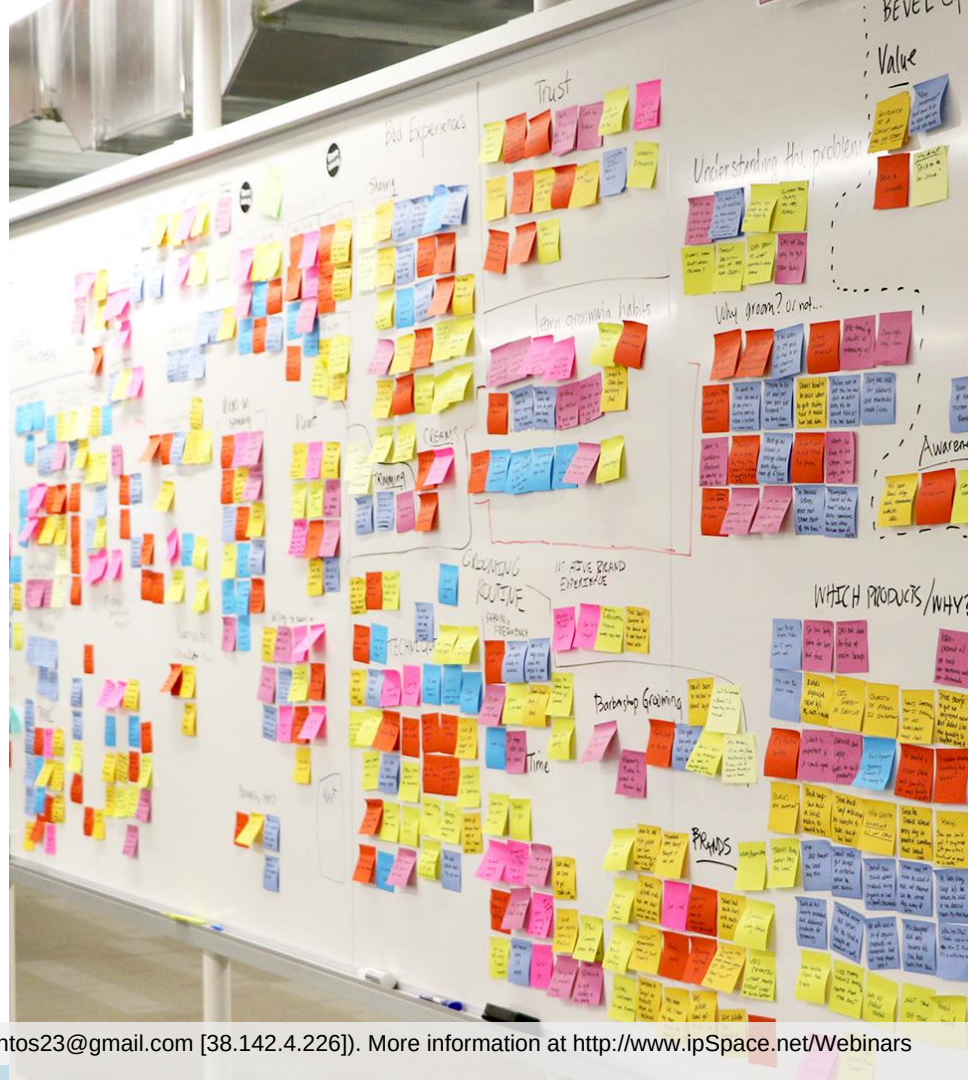
Calgary, Canada based.

Any opinions expressed in this presentation are solely my own.

# Part 4

**A Survey of Load balancing, Ingress and Service Mesh**

- The Problem Space
- A look at Layer 4 vs. Layer 7 load balancers
- Survey of Ingress Controllers capabilities
  - NGINX, Public Cloud, Contour, Istio, NSX ALB
- Introduction to Service Mesh



# The Problem Space

## Modern Application Load Balancing

1. **Allocating** application traffic efficiently / resiliently **to** regions or datacenters

Requirements:

**Spread** traffic across regions / datacenters based on load, geographical location, availability
**Protect** from Denial of Service (DoS) attacks

Typical Solutions:

**IP Anycast** network routing
**Global** server load balancing (GSLB) via DNS & HTTP Redirects

# The Problem Space

## Modern Application Load Balancing

2. **Allocating** application traffic efficiently / resiliently **within** regions / data centers

Requirements:

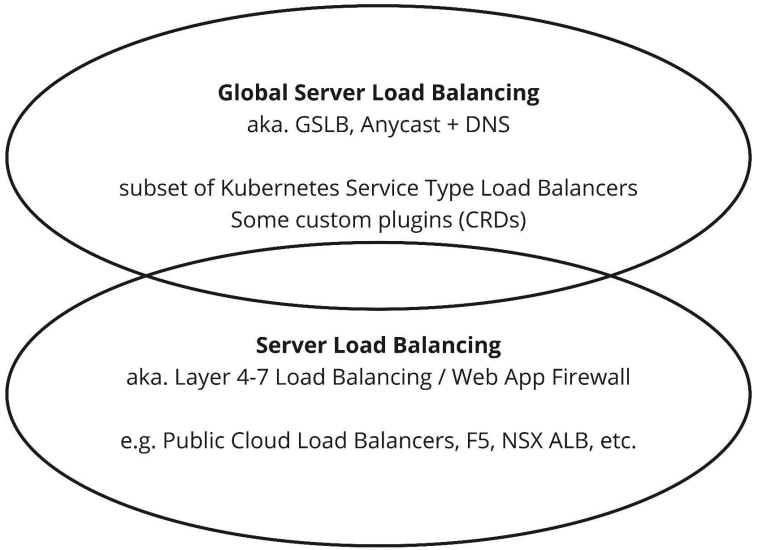
**Direct** traffic to the healthiest servers (load balancing)
**Tolerate** network or server failures
**Scale** to needs of users
**Handle** bursty , unpredictable traffic
**Configure** via Kubernetes standard (preferred) or custom resources

Typical Solutions:

**Network load balancing** (Layer 4) TCP/UDP, Service Type Load Balancers
**Server load balancing** (Layer 4-7) HTTP(S), Ingress Controllers

# Visualizing the Problem Space

## Overlapping Solutions



# The Problem Space

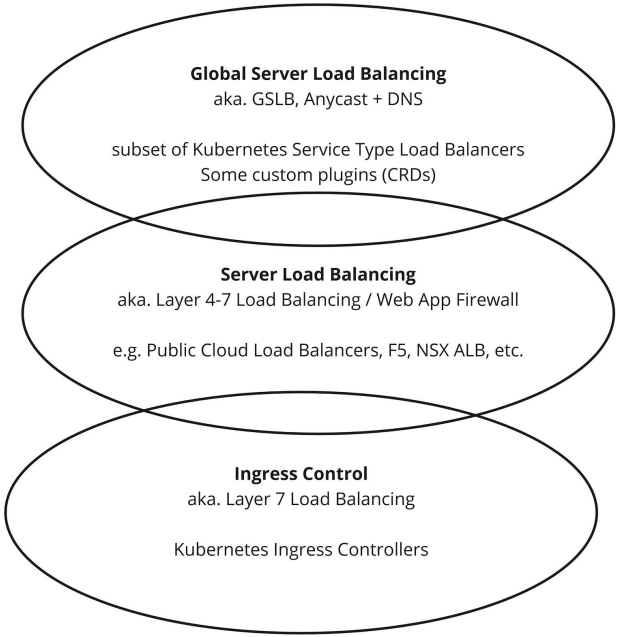
## Modern Application Load Balancing

3. **Managing, routing, transforming** application traffic **sent** to our clusters

**Routing** - where is the traffic destination, by hostname, by web path, etc?
**Traffic Management** - what happens when things go wrong? how should sessions be handled?
**Encryption** - do we decrypt and re-encrypt? do we pass it through? what can we validate?
**Transformation** - how does the app expect the traffic to look? what metadata do they need?
**Security bridging** - should the traffic be authenticated, authorized?
**Control plane** - can I use familiar tools to manage this for the audience? e.g. CLI, API or code?
**Service Discovery** - how do I know what back-end app services exist?
**Observability** - how can we visualize, diagnose, troubleshoot our applications/API net performance?
**Configure** via Kubernetes standard (preferred) or custom resources

# Visualizing the Problem Space

## Overlapping Solutions



# Layer 4 Load Balancing vs. Layer 7 Load Balancing

## Why you need both

**Layer 4 Load Balancing**

The goal:

Choosing a **network path** for a given session

Routing to **physical/virtual machines**

**Minimal state** maintained by the network (just the network flow)

**Changes slowly** by network topology needs

**Layer 7 Load Balancing**

The goal:

Choosing an **application path** for a given session

Routing to **software processes**

Multiplexing and **shared state** (TLS, HTTP/2, Sticky Sessions, etc.)

**Changes frequently** by app development needs

**Overall point: Almost all policy, architecture, and leverage in modern application networking is shifting heavily towards a new generation of distributed Layer 7 data planes & control planes. We still need the lower layers, but they're expected to be "stable abstractions".**

Great discussion of the architectural considerations:

https://blog.envoyproxy.io/introduction-to-modern-network-load-balancing-and-proxying-a57f6ff80236

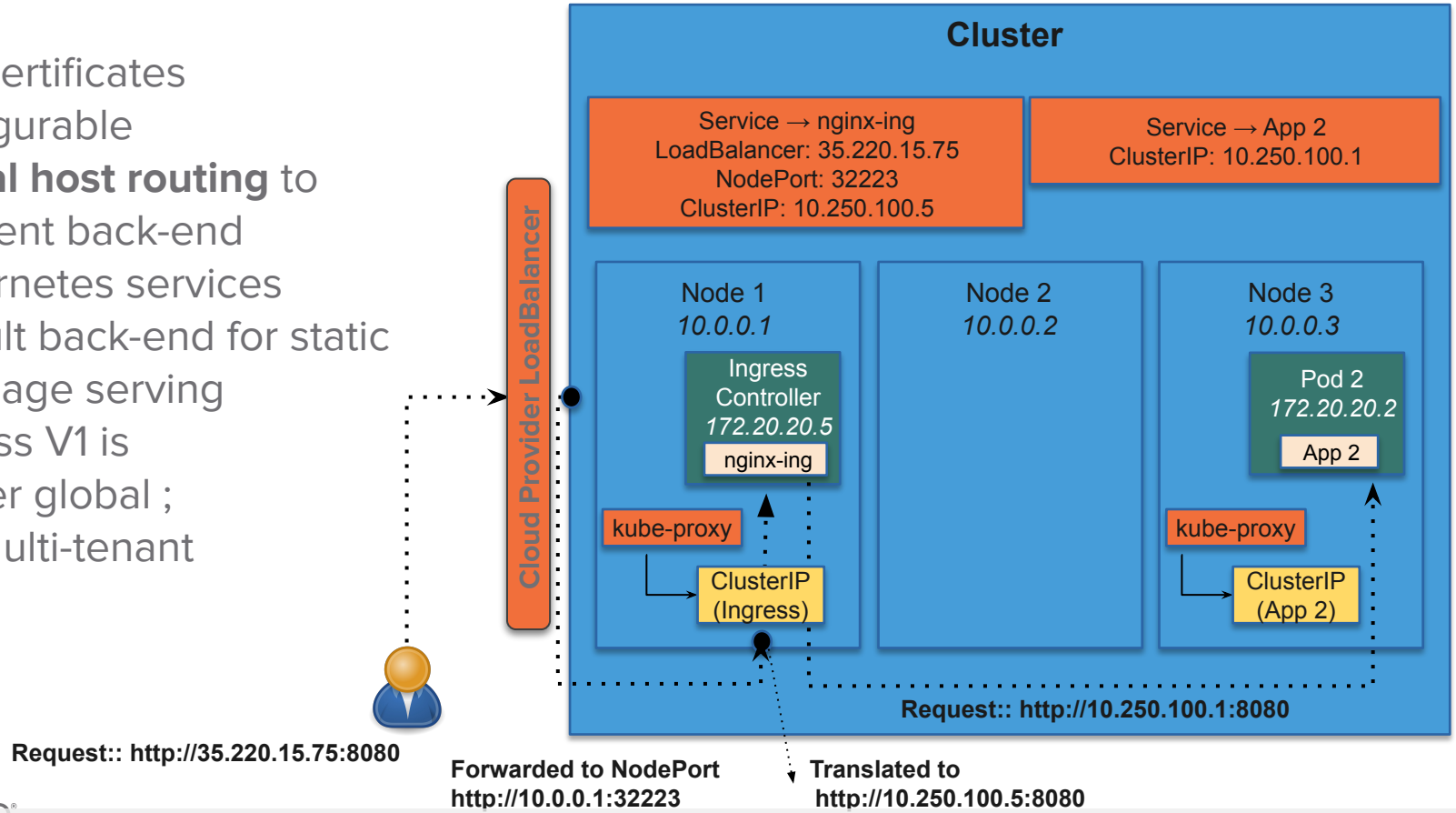
# Ingress and IngressControllers

- ***Ingress***: An API object that manages external access to the services in a cluster
- ***Ingress Controller***: An implementation of Ingress (there are many)
  - e.g. NGINX, HA Proxy, Envoy-based, cloud or commercial LBs
- Traffic routing is controlled by rules defined on the Ingress resource
- Ingress can provide load balancing, SSL termination and name-based virtual hosting

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: my-subdomain.ingress.cloud
    http:
      paths:
      - backend:
          serviceName: my-service
          servicePort: 80
```

# Ingress and IngressControllers

- TLS certificates configurable
- **virtual host routing** to different back-end Kubernetes services
- default back-end for static webpage serving
- Ingress V1 is cluster global ; not multi-tenant



# Envoy is emerging industry standard Layer 7 Proxy

- Developed at Lyft, contributed to CNCF
- Designed from scratch to be fully API driven
- Low memory footprint, written in C++
- Dynamic endpoint discovery
- Rich telemetry
- Intelligent traffic management
- Distributed security
- Web Assembly (WASM) extensibility

# Where do various Ingress solutions **overlap**?

## Layer 7 Load Balancing / Ingress / Gateways

| Capability | **NGINX** | **Public Cloud Native** | **NSX ALB** | **Contour (Envoy)** | **Istio Gateway (Envoy)** |
|---|---|---|---|---|---|
| Kubernetes Ingress Controller | Yes | Yes | Yes | Yes | Yes |
| HTTP/2 | Yes | Yes | Yes | Yes | Yes |
| Virtual Host Routing | Yes | Yes | Yes | Yes | Yes |
| Wildcard host routing | Yes | Yes | Yes | Yes | Yes |
| Header transformation | Yes (configmap & annotation) | Varies | Yes, custom CRD | Yes (HTTPproxy CRD) | Yes (VirtualService CRD) |
| Source IP preservation | Yes (annotation) | Yes | Yes | Yes | Yes |
| App traffic management (weighted LB, failover, circuit breaking) | NGINX PLUS | Varies | Yes, custom CRD | Yes | Yes |
| (m)TLS Encryption | Yes | Yes | Yes | Yes | Yes |

# Where do the solutions **differ**?

## Layer 7 Load Balancing / Ingress / Gateways

| Capability | **NGINX** | **Public Cloud Native** | **NSX ALB** | **Contour (Envoy)** | **Istio Gateway (Envoy)** |
|---|---|---|---|---|---|
| mTLS x509 Authentication | Limited unless on NGINX Plus | Limited | Limited | Yes | Yes |
| SAML SSO App Authentication | 3rd party auth server | No (varies) | Yes | No | No |
| OAuth2 / OIDC (JWT) SSO App Authentication | 3rd party auth server | No (varies) | No | Yes | Yes |
| HTTP Content Caching | Yes | No (varies) | Yes | No (Possible future) | No (Possible future) |
| HTTP Path / Method RBAC | No | No (varies) | Yes (limited) | No | Yes (Authorization policy) |
| Multi-Cluster Ingress Rules | No | Varies | Yes | No | Partially (Multi-cluster) |
| Modular manifests (via includes) | NGINX Plus | No | No | Yes (HTTPProxy CRD) | No |

# Where do the solutions **differ**?

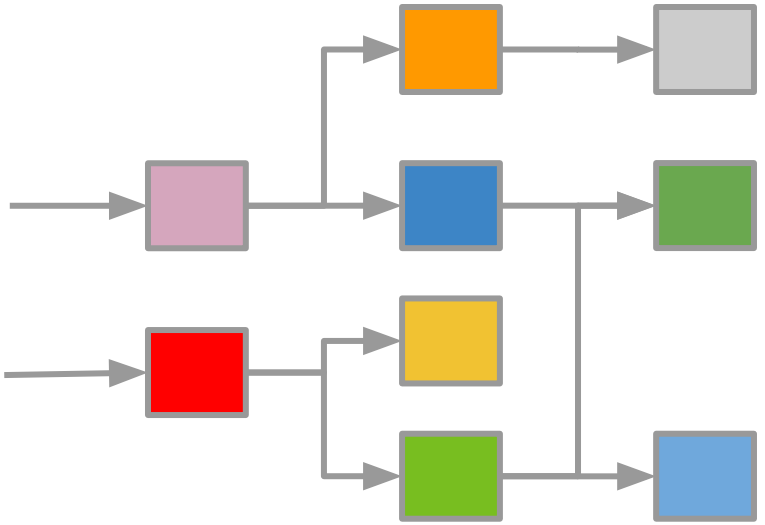
## Layer 7 Load Balancing / Ingress / Gateways

| Capability | **NGINX** | **Public Cloud Native** | **NSX ALB** | **Contour (Envoy)** | **Istio Gateway (Envoy)** |
|---|---|---|---|---|---|
| Multi-Tenant Delegated Proxy Admin | Maybe with NGINX Plus | No | No | Yes (HTTPProxy CRD) | Yes (VirtualService) |
| Multi-Tenant Self-Service Hostnames | Maybe with NGINX Plus | No | No | Yes (HTTPProxy CRD) | Yes (Gateway & VS) |
| Multi-Tenant Hostname conflict detection | Maybe with NGINX Plus | No | No | Yes | Yes |
| Multi-Tenant TLS certificate configuration | NGINX Plus or Cert-Manager | No | No | Yes (HTTPProxy CRD) | Via Cert-Manager (or in-mesh mTLS) |
| Multiple Upstream Services for a Hostname | NGINX Plus | Varies | No | Yes (HTTPProxy CRD) | Yes |
| Friendly Kubernetes Status Reporting | Varies | Varies | Yes | Yes | Yes |

# The Service Mesh: Problem Space

In a world with **many microservices**, in **different languages**, running on **different platforms**...

too many of the **operational requirements** are put on the **individual apps**



# Problems solved by developers vs. operations

Examples…

**Security**:

*all apps must encrypt and authenticate all network calls with mTLS*

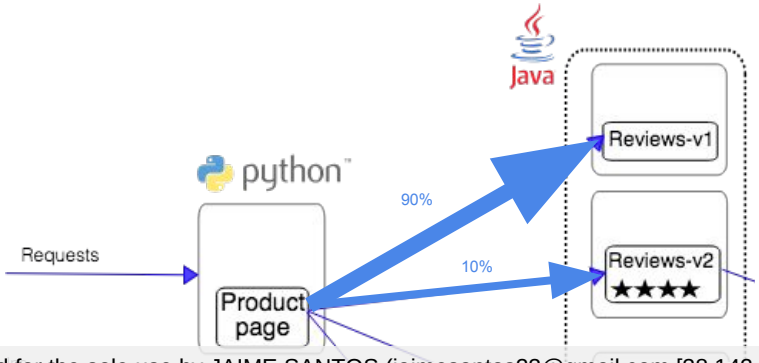
*all apps must include a JWT token denoting the authenticated user*

# Problems solved by developers vs. operations

Examples…

**Traffic management**:

*shift 10% of our mobile traffic to the new version of the Reviews microservice, leave 90% using the old version.*



# Problems solved by developers vs. operations

Examples…

**Observability**:

*How often do our apps encounter a timeout error while uploading to the blobstore?*
*What's the latency between these microservices?*

# Libraries & frameworks help some; commercial LBs have solved for years

- Individual load balancing technologies have solved these problems for years but they're often inaccessible to developer teams

- **Spring** (Java framework) helped standardize; **NetflixOSS** pioneered these capabilities on Java

- App Devs still have to configure the framework
  - operator-concerns (like security) are not separated

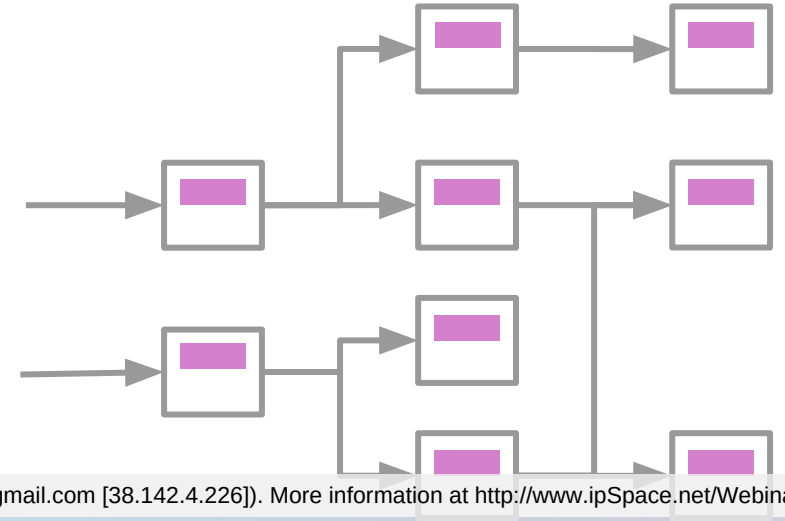
- And what happens when different microservices are written in different languages?

# Problem statement

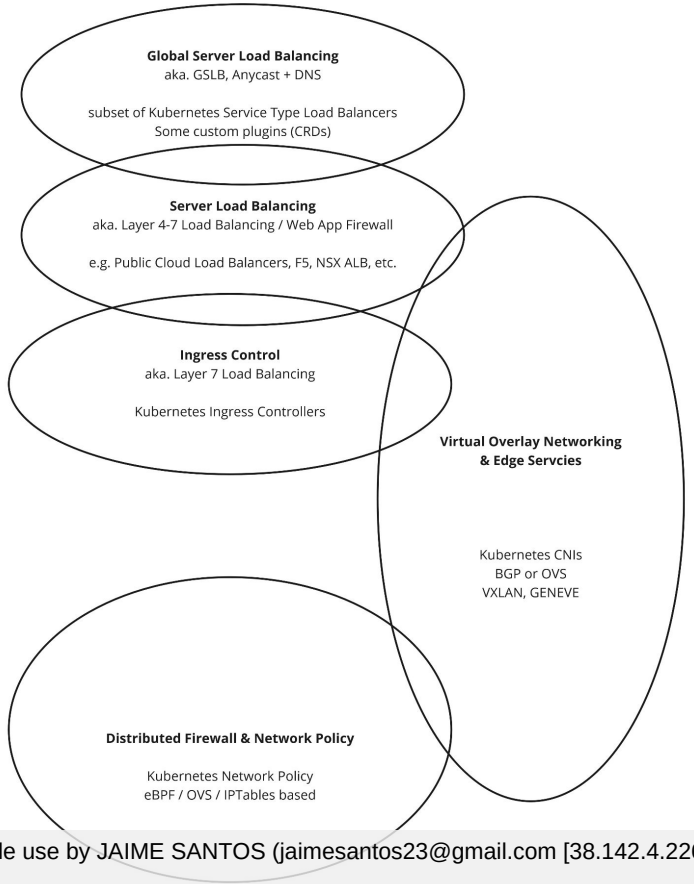
In a world with **many microservices**, in **different languages**, running on **different platforms**...

too many of the **operational requirements** are put on the **individual Apps**

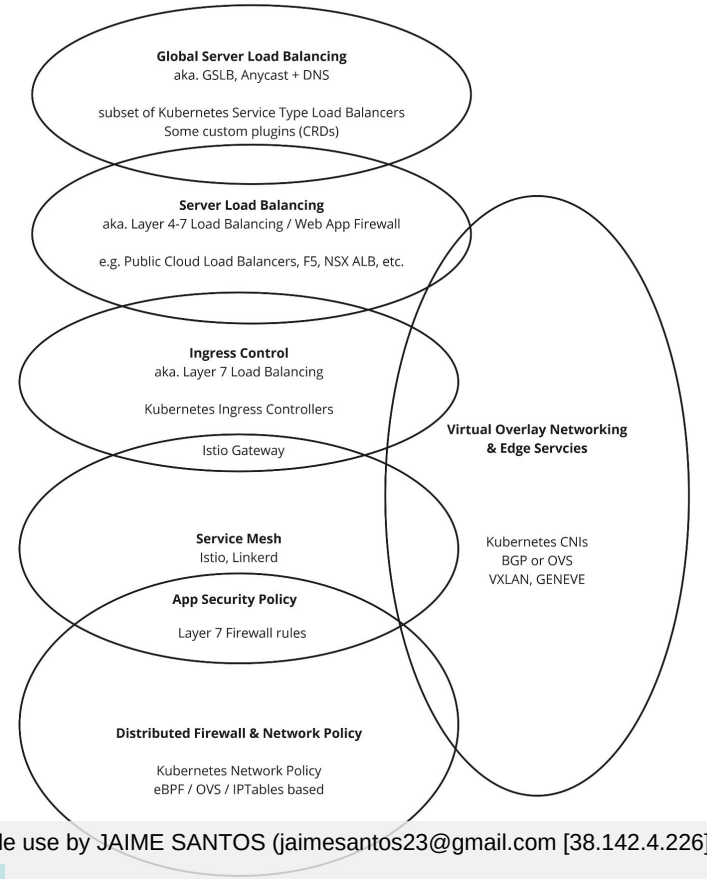
Instead, let's try to **solve** application networking challenges more **in the platform**.



# Visualizing the Problem Space **before Service Mesh**



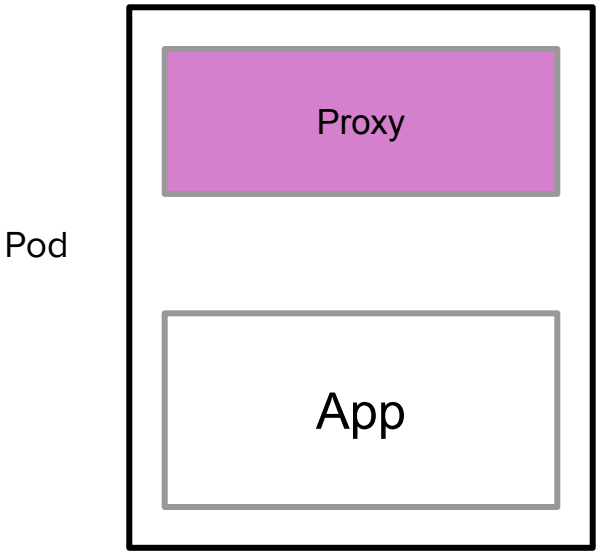
# Visualizing the Problem Space **after Service Mesh**



# What is a service mesh?

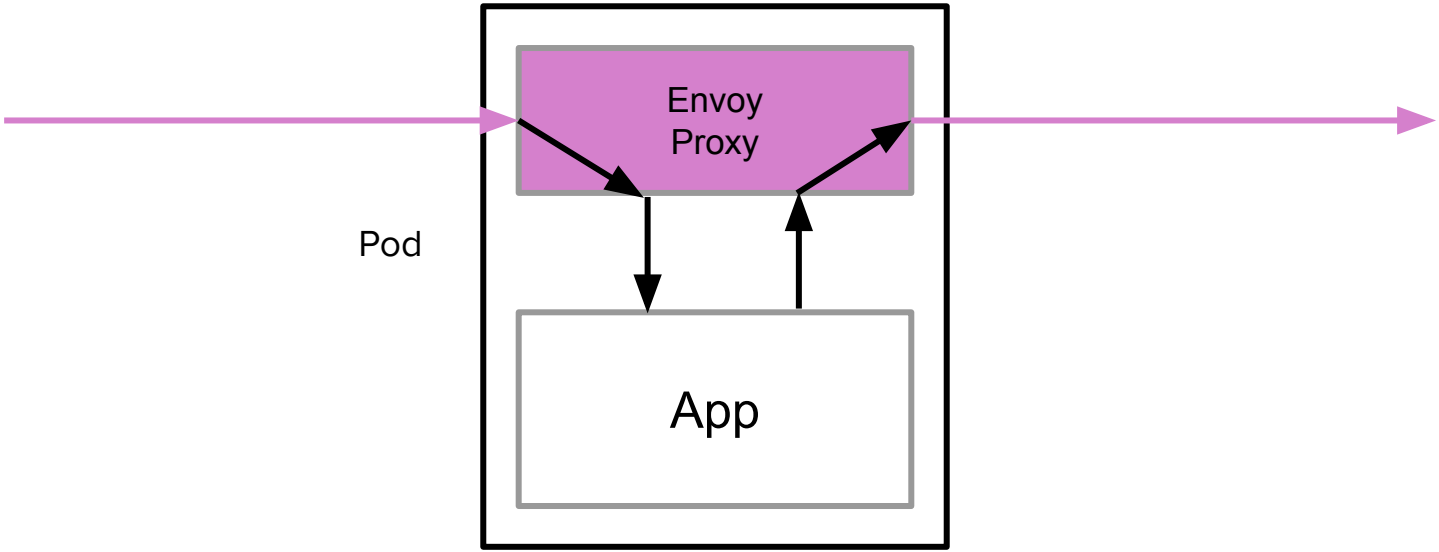
- Every workload (Pod) gets an adjacent **sidecar proxy**

  These are dynamically injected into a Kubernetes manifest using an admission controller



# What is a service mesh?

- **all traffic** in and out of the workload **traverses the proxy**
- Typically **iptables** redirect rules per-pod; some CNIs can plumb sidecards with **eBPF**
- TCP traffic, HTTP traffic, some app-specific L7 protocols (e.g. MySQL, MongoDB, Redis, etc.)

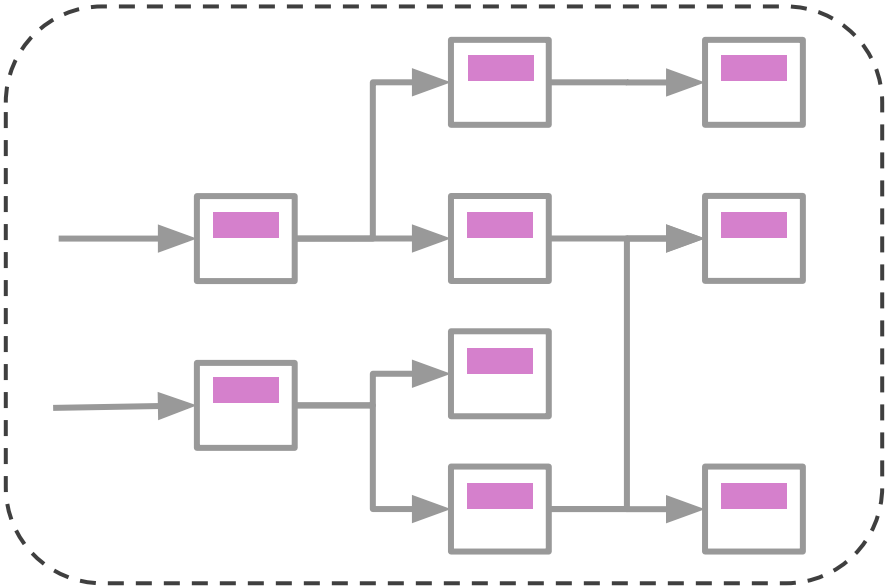


# What is a service mesh?

- Microservice calls go through a proxy on **both sides**
- Uses the Kubernetes service endpoint metadata - but ***bypasses*** all DNAT / ClusterIPs
- Each proxy in the mesh has a map directly to every pod per service selector!

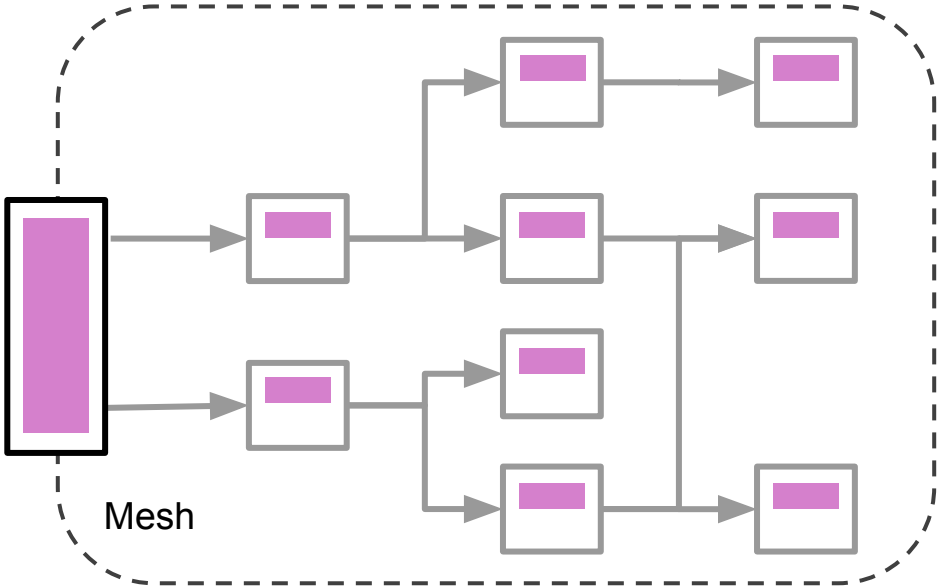
# What is a service mesh?

- *"Service Mesh" = All **proxy**-ified workloads*



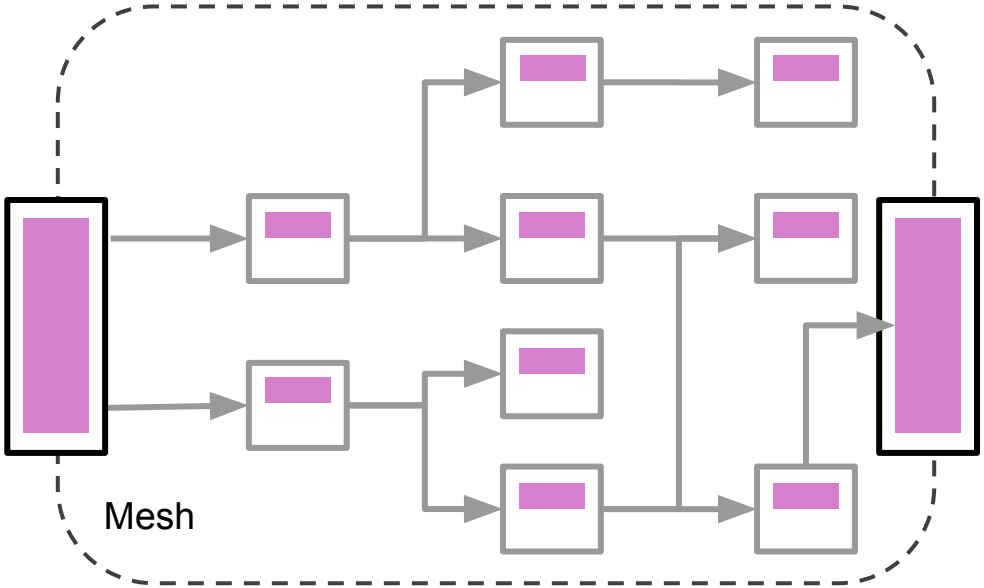
# What is a service mesh?

- *"Service Mesh" = All **proxy**-ified workloads and **ingress proxies** too!*



# What is a service mesh?

- *"Service Mesh" = All **proxy**-ified workloads and **ingress proxies** too! Even **egress**!*

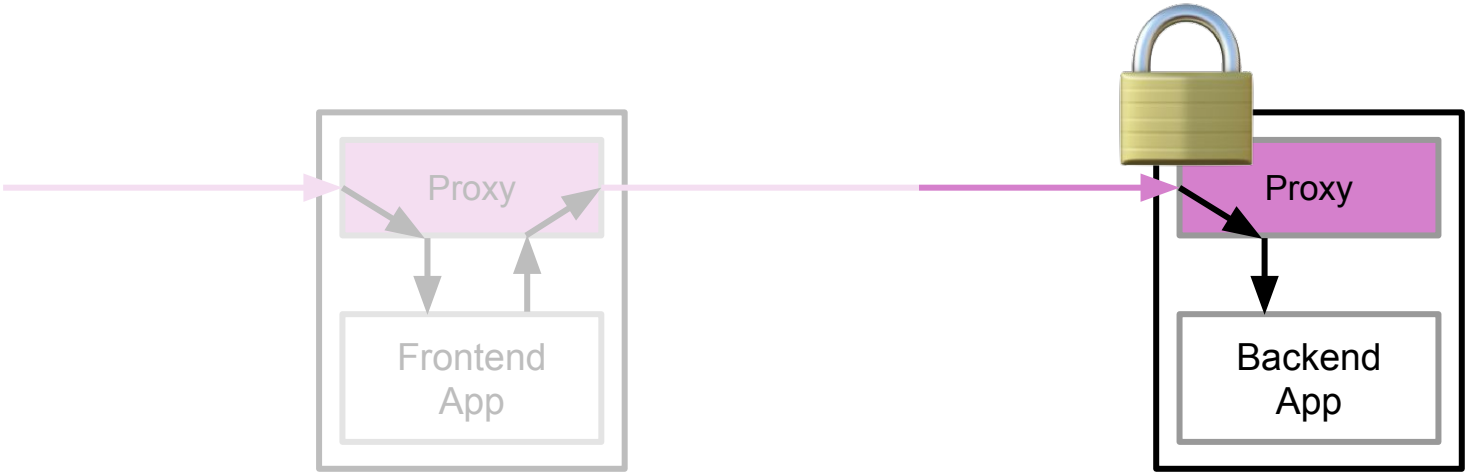


# What can the proxy do?

**Security**: Decrypt incoming (ingress) requests & authenticate clients via mTLS

**SPIFFE**: "Secure Production Identity Framework for Everyone" standardizes x509 subjects
e.g. spiffe://trust.domain/ns/kubernetes-namespace/sa/kubernetes-service-account

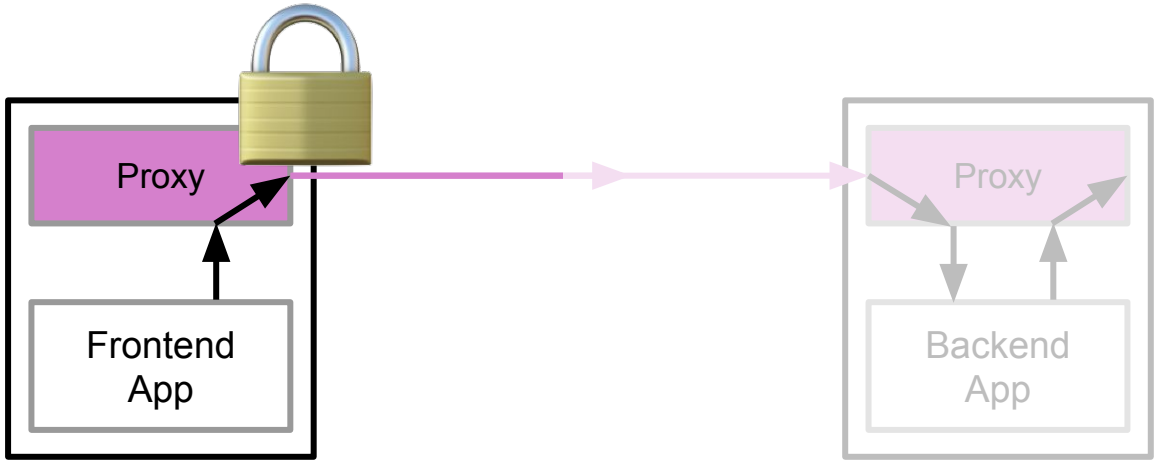
Authorize flows, i.e. Layer 7 segmentation (Service Account A cannot talk to Service Acct B)



# What can the proxy do?

**Security**: Encrypt outgoing (egress) requests & authenticate servers

e.g. mTLS origination; TLS mediation

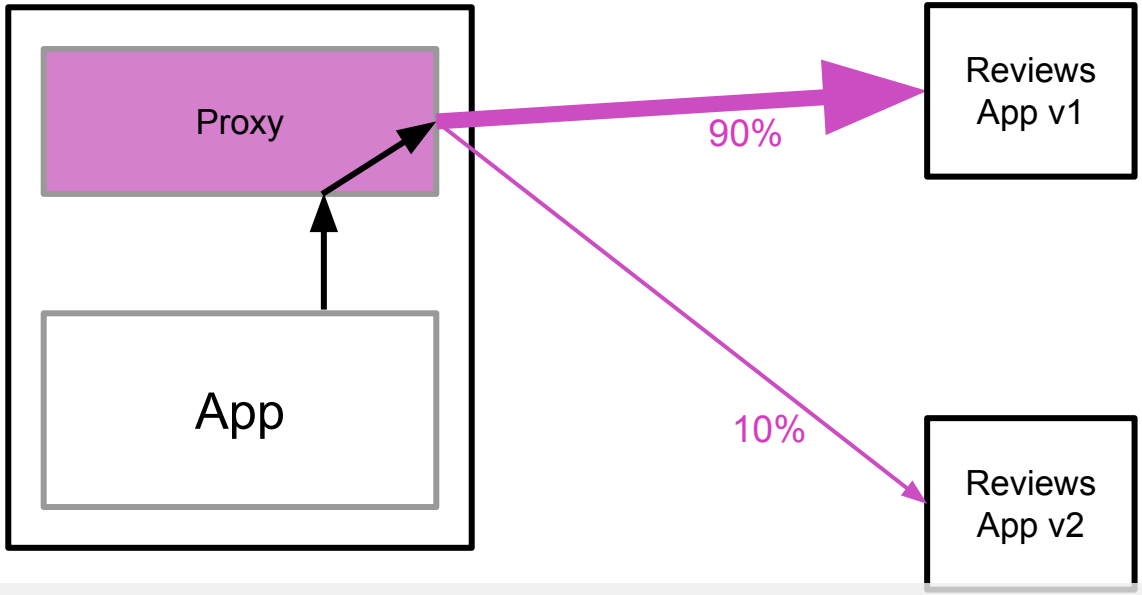


# What can the proxy do?

Traffic management: apply custom load-balancing strategies

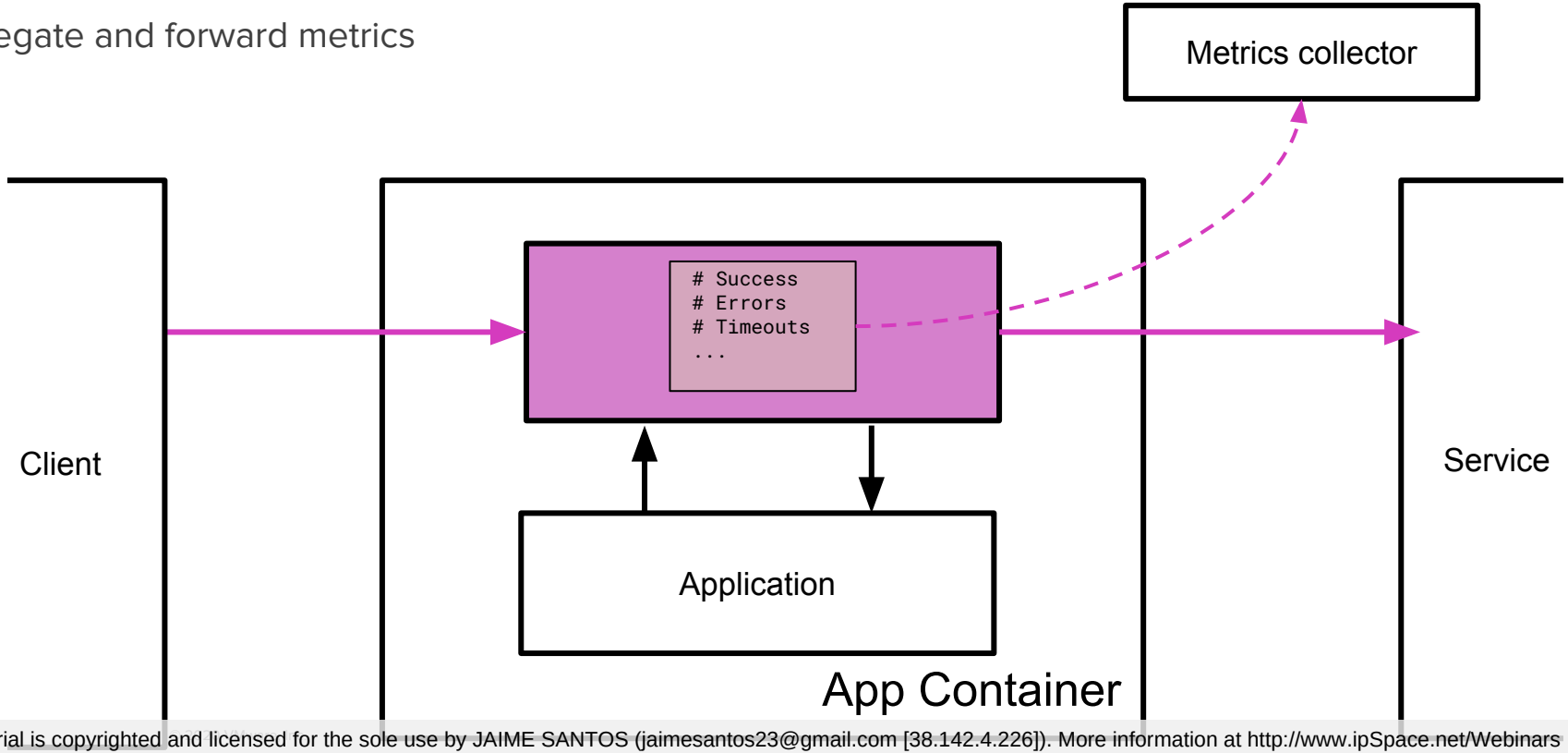
outlier detection

redirect by locality or availability



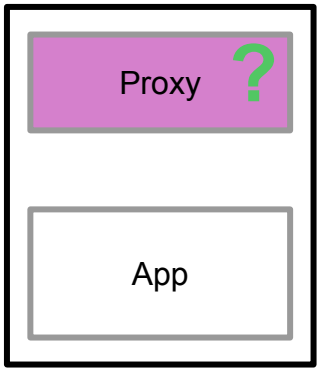
# What can the proxy do?

Aggregate and forward metrics



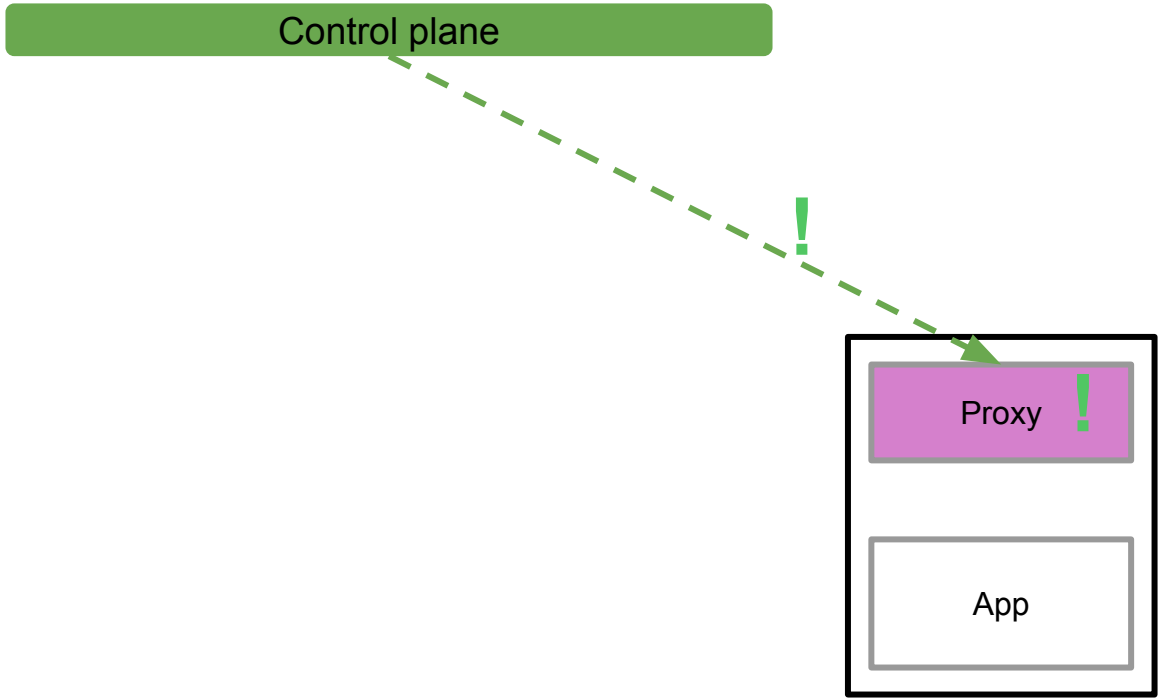
# How does the proxy know what to do?

- **Security:** which clients to allow in? which servers to trust?
- **Traffic management**: what %-weights to use for which backends?
- **Observability**: which metrics to collect?
  and where to send them?

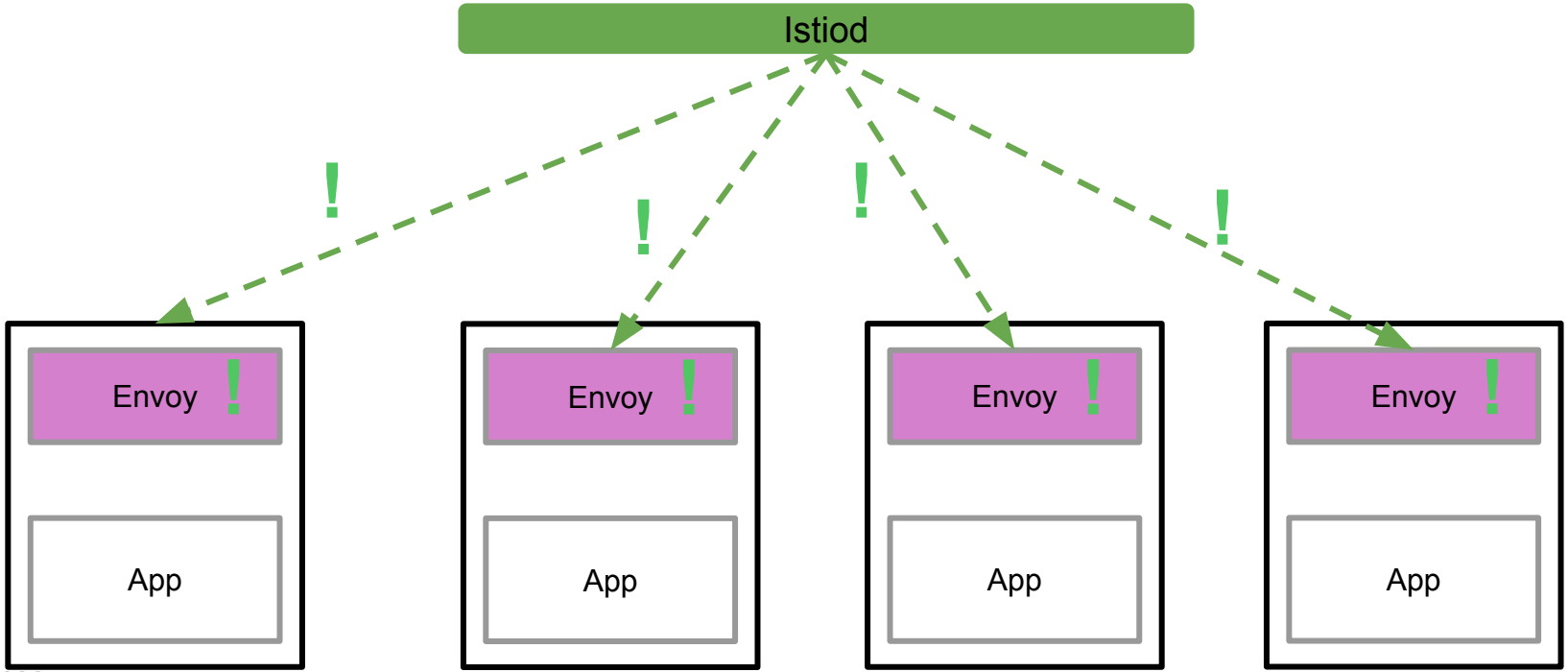


# The proxy is configured by a control plane

(management servers)

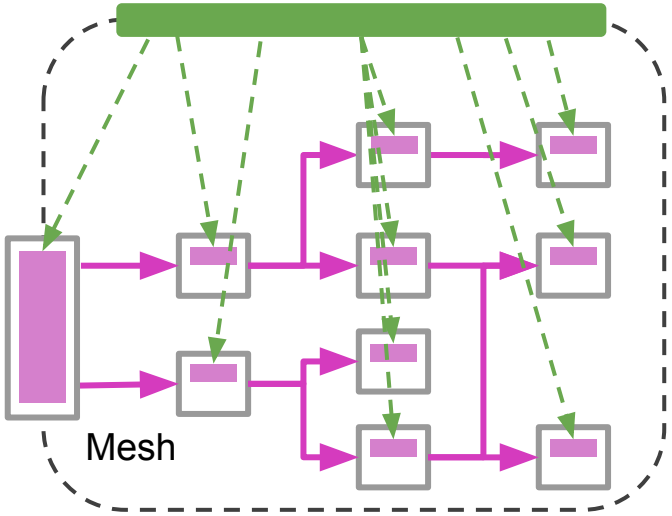


# The **control plane** (e.g. Istio) can manage many proxies



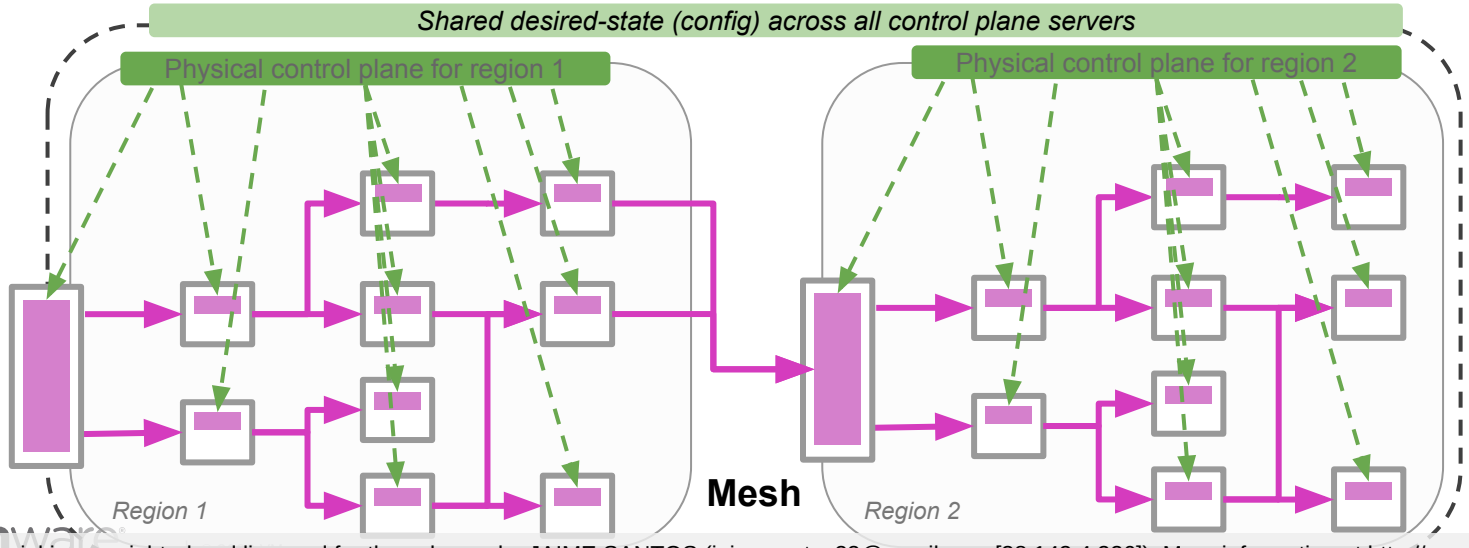
# What is a service mesh?

- *"Service Mesh" = All **proxy**-ified workloads sharing a* logical ***control plane***

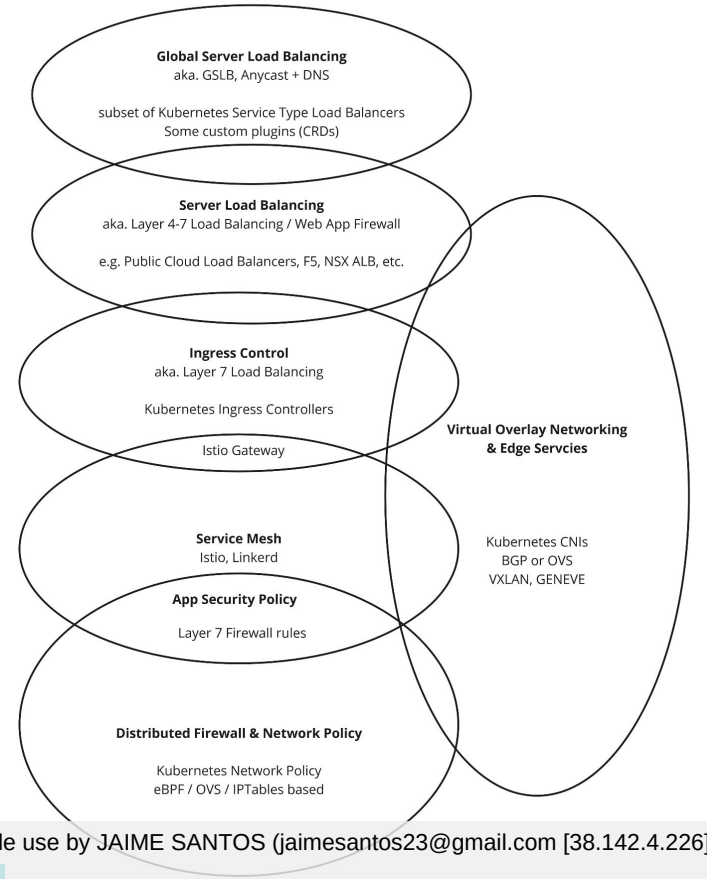


# The Multicluster Service Mesh Present & Future

- *Control plane may be distributed across many servers, in different containers, VMs, datacenters, etc.*
- *Ingress controllers can parse via special SNI rules & dynamic service discovery, split-horizon endpoint resolution*
- ***Virtual circuit routing (TCP or HTTP) across disparate L3 networks via HTTP/2 and mTLS***



# **Summary:** The Kubernetes Networking Problem Space



# Thank You!

Twitter: @svrc
Email: scharlton@vmware.com