

# Kubernetes Networking – Getting Started

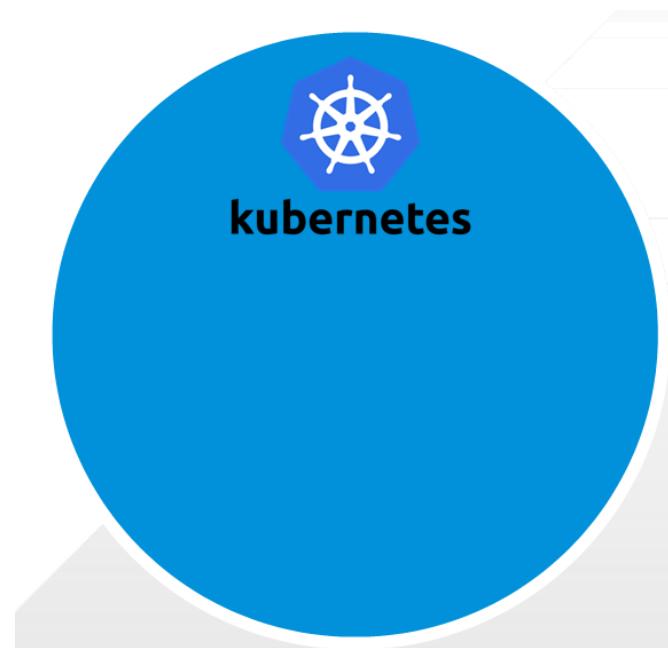
1



Develop a passion for  
learning.

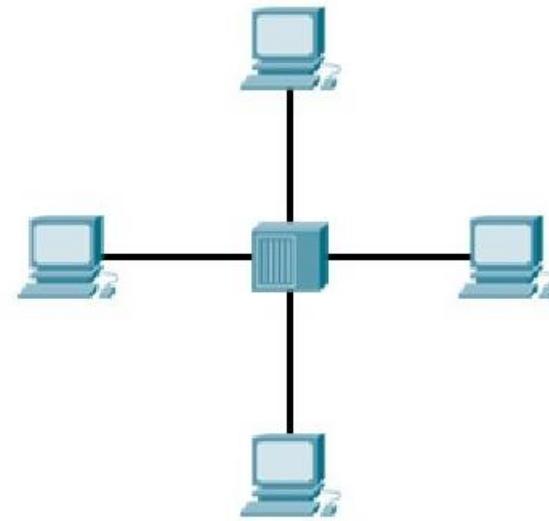
# What is Kubernetes?

Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure.



# What is Networking?

Networking is the exchange of information or services among individuals, groups, and institutions via computers, wired and wireless equipment, and devices



# Challenges of Kubernetes Networking?

Networking is a central part of Kubernetes, but it can be challenging to understand exactly how it is expected to work. There are 4 distinct networking problems addressed by Kubernetes Networking.

# Kubernetes Networking Building Blocks

- **Highly-coupled container-to-container communications:** this is solved by Pods and localhost communications.
- **Pod-to-Pod communications:** this is solved by network namespaces and the IP-per-Pod model.
- **Pod-to-Service communications:** this is covered by services.
- **External-to-Service communications:** this is covered by services.

# Kubernetes Networking Service Models

- In-house Kubernetes deployment
- SaaS Solutions for Kubernetes
- Fully outsourced (managed) Kubernetes services

All of these options have different limitations and options for networking

# Overview



Kubernetes Networking are just small data pipelines with massive amounts of data/information/knowledge/insight being processed. Big data pipeline engineering can become way more complex. There are lots of sources, lots of sinks (places where we put our data), lots of complex transformations, and lots of data. Just imagine having dozens of operational databases, a clickstream from your site coming through Kafka, hundreds of reports, OLAP cubes, and A/B experiments. Imagine, as well, having to store all the data in several ways, starting with raw data and ending with a layer of aggregated, cleaned, verified data suitable for building reports.

# LOGISTICS

## Class Hours:

- Start time is 9:30am PST
- End time is X:XXpm
- Class times may vary slightly for specific classes
- Breaks mid-morning and afternoon (15 minutes)



## Lunch:

- Lunch is
- Yes, 1 hour and 15 minutes
- Extra time for email, phone calls, or simply a walk.



## Telecommunication:

- Turn off or set electronic devices to vibrate
- Reading or attending to devices can be distracting to other students
- Try to delay until breaks or after class

## Miscellaneous

- Courseware
- Bathroom
- Fire drills

# DAY 1

# Agility



- K8S Foundation
  - Kubernetes stuff
- East-West Communication
  - Kube-proxy communication
  - Intra-pod communication (container to container)
  - Inter-pod communication (pod to pod)
  - Service mesh networking Pod to service communication
- North-south communication
  - Kubernetes ingress
  - Multi-cluster communication
- Kubernetes networking versus Docker networking

# DAY 2

# Velocity



- Lookup and discovery
  - Self-registration
  - Services and endpoints
  - Connectivity with queues
  - Loosely coupled connectivity with data stores
- Kubernetes networking
  - IP addresses and ports
  - Network namespaces
- Virtual Network devices
  - Bridges
  - Routing
  - Maximum transmission unit (MTU)
- Pod networking

DAY 3

# Observability



1. Networking Plugins
  - o CNI
  - o Kubenet
  - o CNI plugin – Antrea
  - o Caching proxies for k8s
2. Troubleshooting networking
  - o Underlay
  - o Overlay
  - o Pods
3. Monitoring
  - o Prometheus metrics & dashboard
  - o Grafana visualization

# Meet The Instructor

## George Niece AKA geo

Digital Transformation, DevSecOps, IoT Consultant with a Software Engineering, Digital Commerce, and IT operations background. Focused on cloud-native application modernization, datacenter divestiture, and cloud adoption.



Twitter  
[@georgeniece](https://twitter.com/georgeniece)

LinkedIn  
[Linkedin.com/in/GeorgeNiece](https://www.linkedin.com/in/GeorgeNiece)

mail  
[George.Niece@DigitalTransformationStrategies.net](mailto:George.Niece@DigitalTransformationStrategies.net)

### Expertise

- Cloud-native
- XaaS
- AppDev
- IoT
- Automation
- CI/CD
- Microservices
- Agile

# GTKY

Please let me know your

Name

Where you work from?

Experience with

K8S

Kind/K3D

Kubernetes Networking

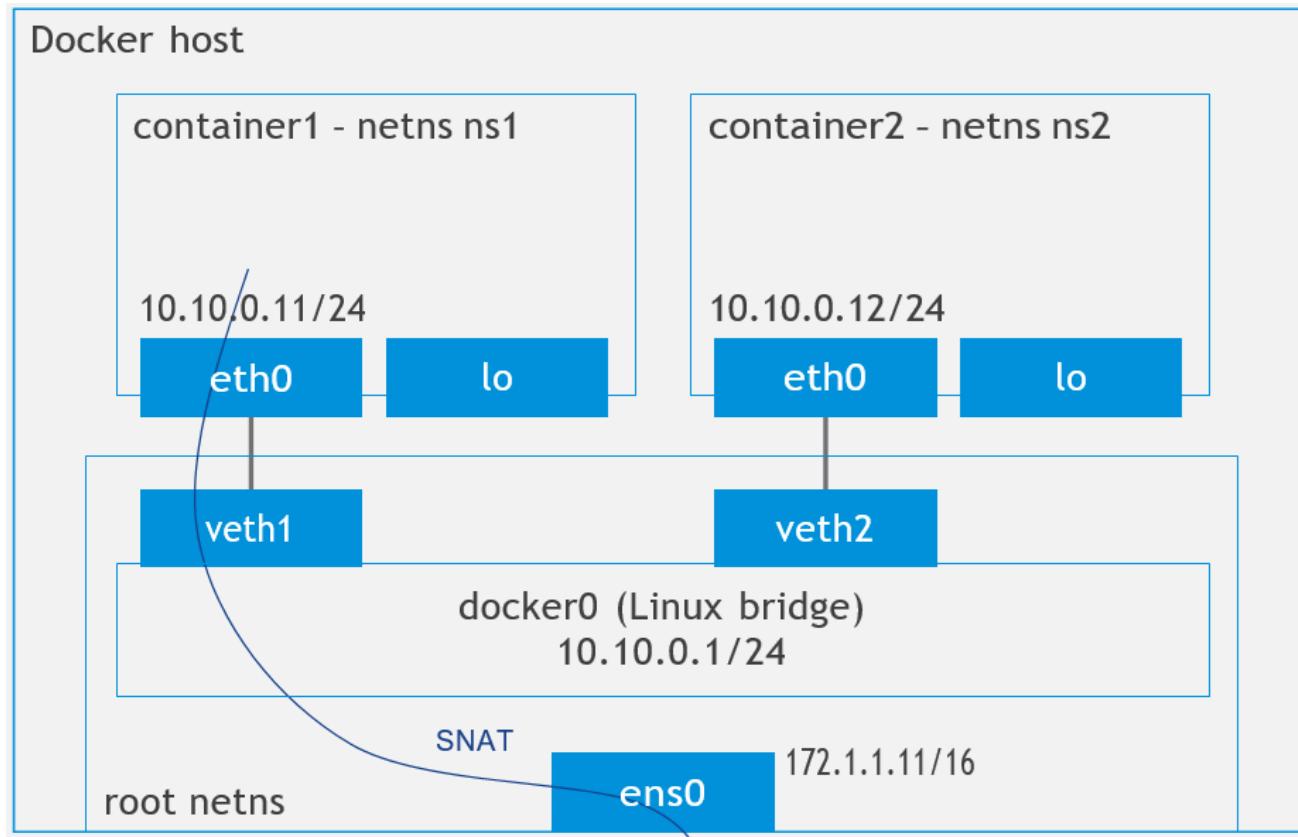
Please rate yourself (1-5) 1 = new joiner, 5 = SME

What is your favorite beverage?

# Basics of Container Networking



# Docker Bridge Network on Linux



# Docker Bridge Network on Linux

## **Network Namespace**

- Isolated network environment provided by Linux kernel

## **Interconnect**

- A simple way:  
veth devices & Linux bridge

## **Communication across hosts**

- Network address translation and port mapping

# Container veth Networking

The **veth networking interface** is commonly used in container virtualization. It is **simple** to configure and **flexible** to accommodate a wide variety of configuration options.

# Bridged Networking

There are four importance concepts about bridged networking:

- Docker0 Bridge
- Network Namespace
- Veth Pair
- External Communication

# K8S aint Docker

Kubernetes has its own very specialized network implementation. It can't easily assign a unique externally accessible IP address to each process the way the Docker MacVLAN setup can. Kubernetes also can't reuse the Docker networking infrastructure. Generally, the K8S cluster takes responsibility for assigning IP addresses to pods and services, and you can't specify them yourself.

# In Kubernetes ...

- You can't manually assign IP addresses to things.
- The cluster-internal IP addresses aren't directly accessible from outside the cluster.
- The Kubernetes constructs can only launch containers on arbitrarily chosen nodes (possibly with some constraints; possibly on every node).
- You don't usually launch a container on a single specific node.
- You can't run a non-container command on a node.

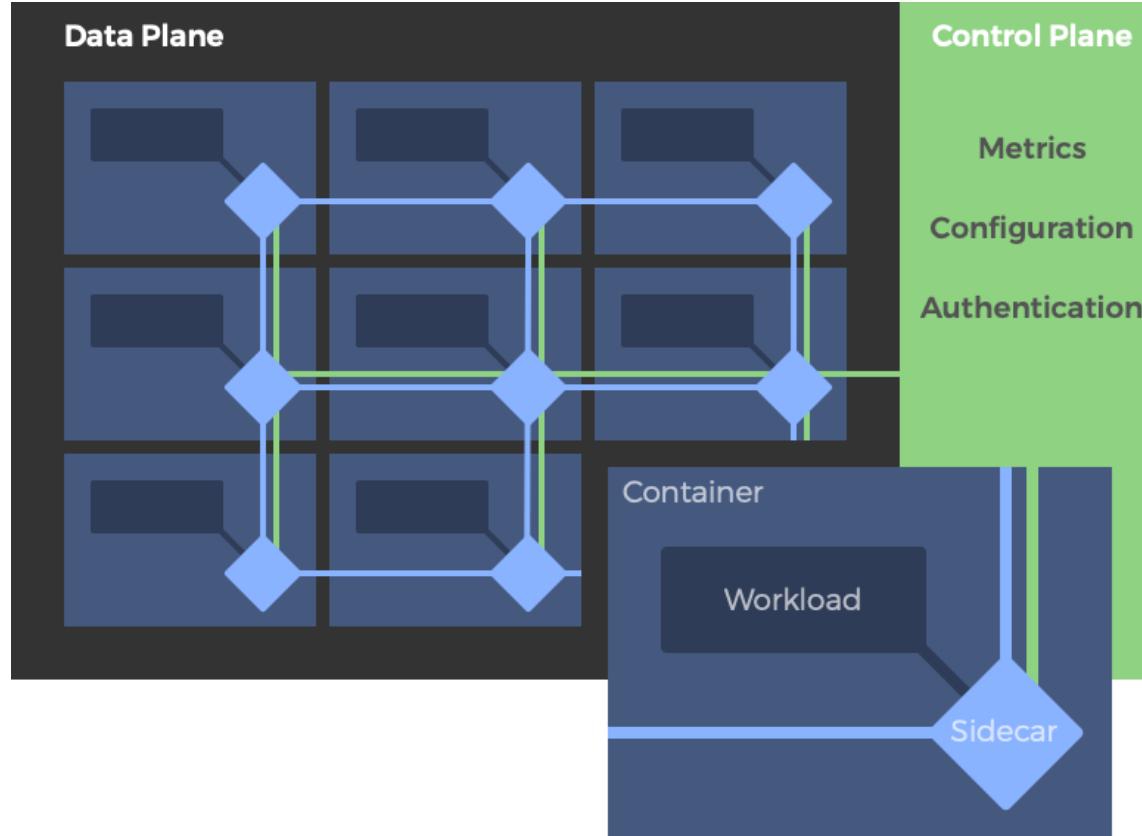
# K8S and Configuration Management

- General-purpose cluster automation tools like Salt Stack, Ansible, or Chef are really useful in some situations.
- CM tools will let you launch processes directly on managed nodes.
- Server-type processes can make use of the node's host's IP addresses as normal to automate operating system maintenance for your cluster.

# K8S Networking Microservice Considerations

- How will we manage the actual communication between the services?
- How will we manage the configuration and policies of service2service communication?

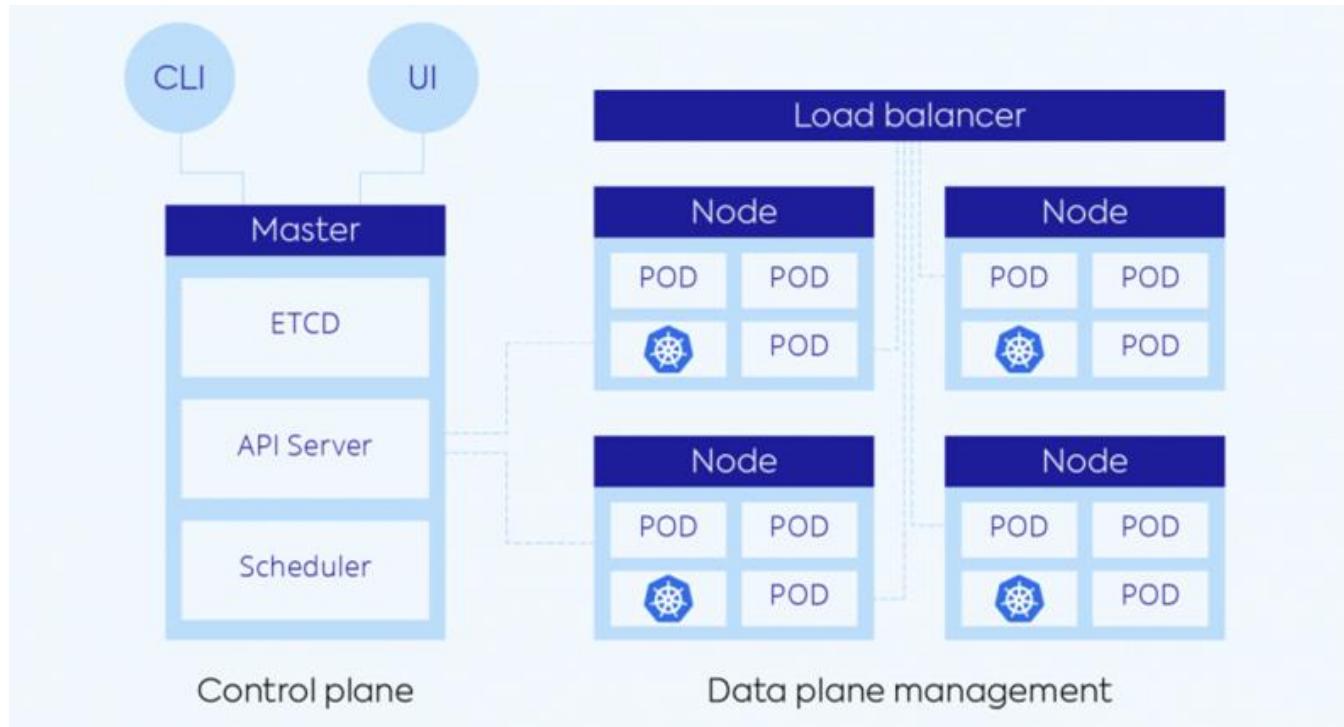
# K8S Networking Planes



# K8S Networking Planes

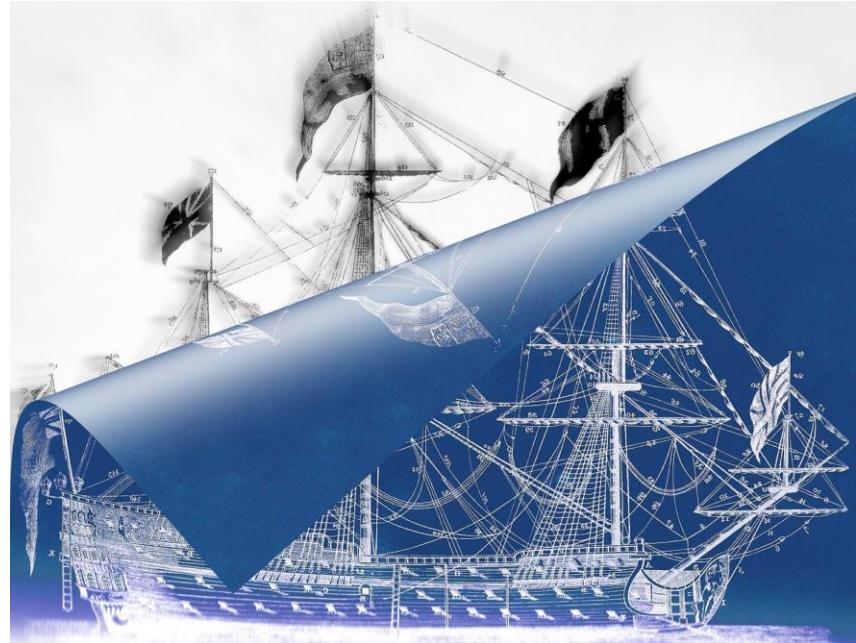
- Control Plane – the “**brains**”, consists of the manager nodes in a Kubernetes cluster. Kubernetes carries out communications internally, and where all the connections from outside— via the API—come into the cluster to tell it what to do.
- Data Plane - If the control plane is the brains of Kubernetes, where all the decisions are made, then the data plane is the “**body**”. Worker nodes (i.e. VMs) on the data plane carry out commands from the control plane and can communicate with each other via the kubelet, while the kube-proxy handles the networking layer.

# K8S Networking Planes



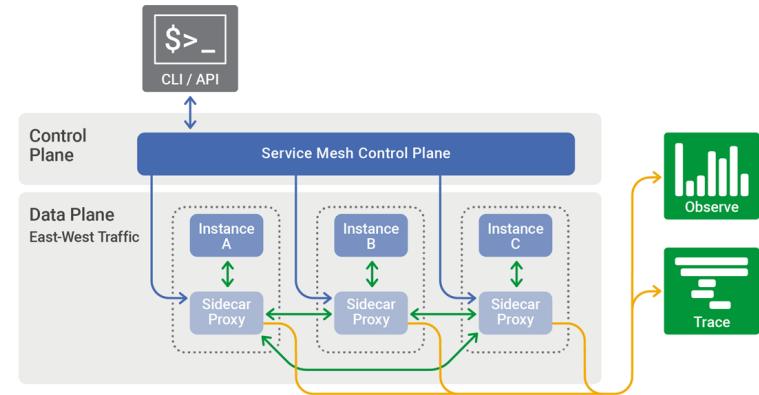
# Service Mesh

*A service mesh is not a “mesh of services.” It is a mesh of Layer 7 proxies that microservices can use to completely abstract the network away. Service meshes are designed to solve the many challenges developers face when talking to remote endpoints.*

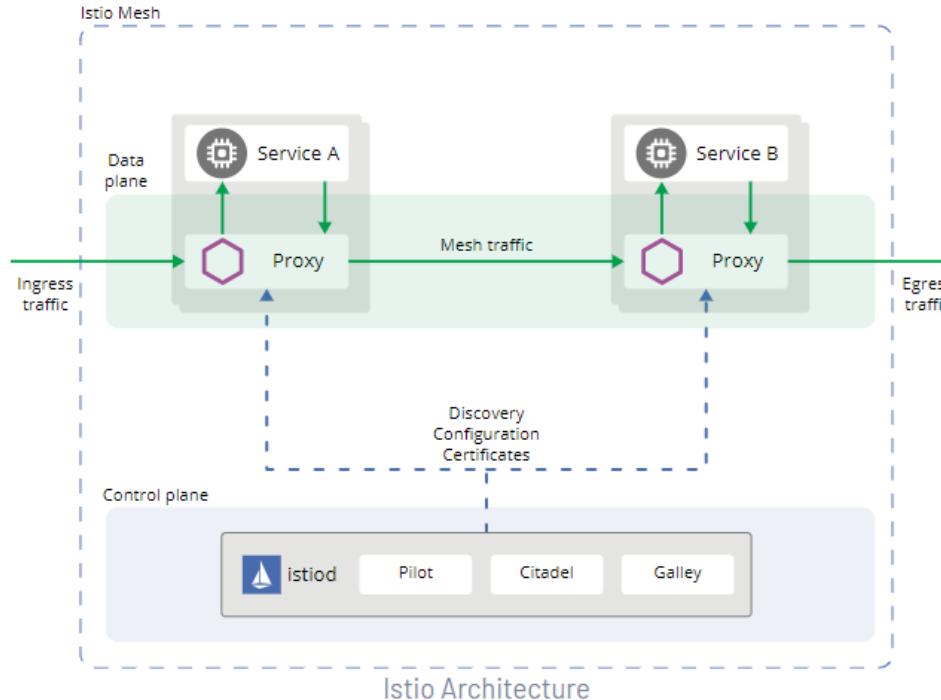


# Service Mesh – Istio & Telemetry

Istio is an open source service mesh designed to make it easier to connect, manage and secure traffic between, and obtain telemetry about microservices running in containers. Istio is a collaboration between IBM, Google and Lyft. It was originally announced in May 2017, with a 1.0 version released in July of 2018.



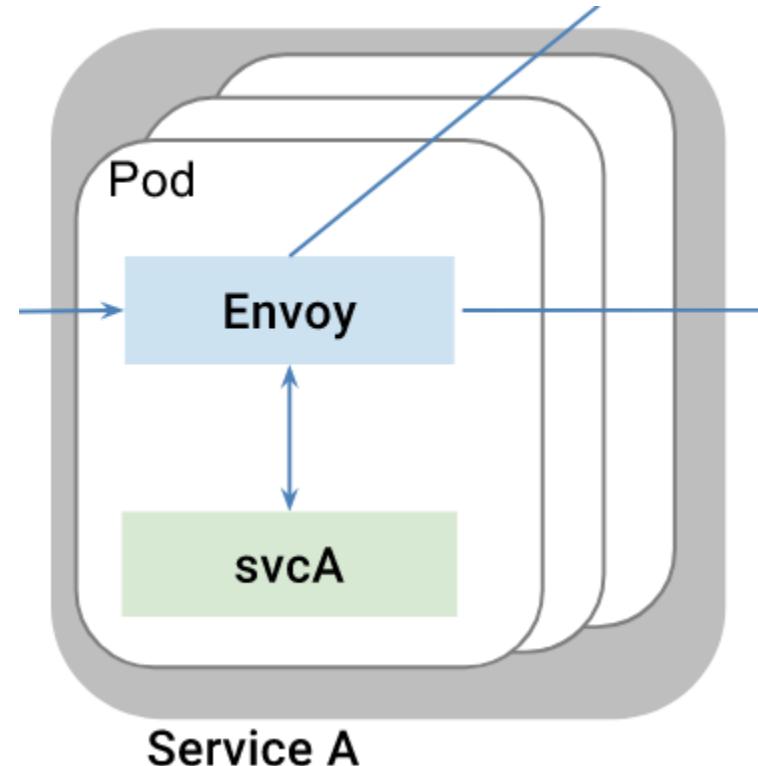
# Service Mesh – Istio Architecture



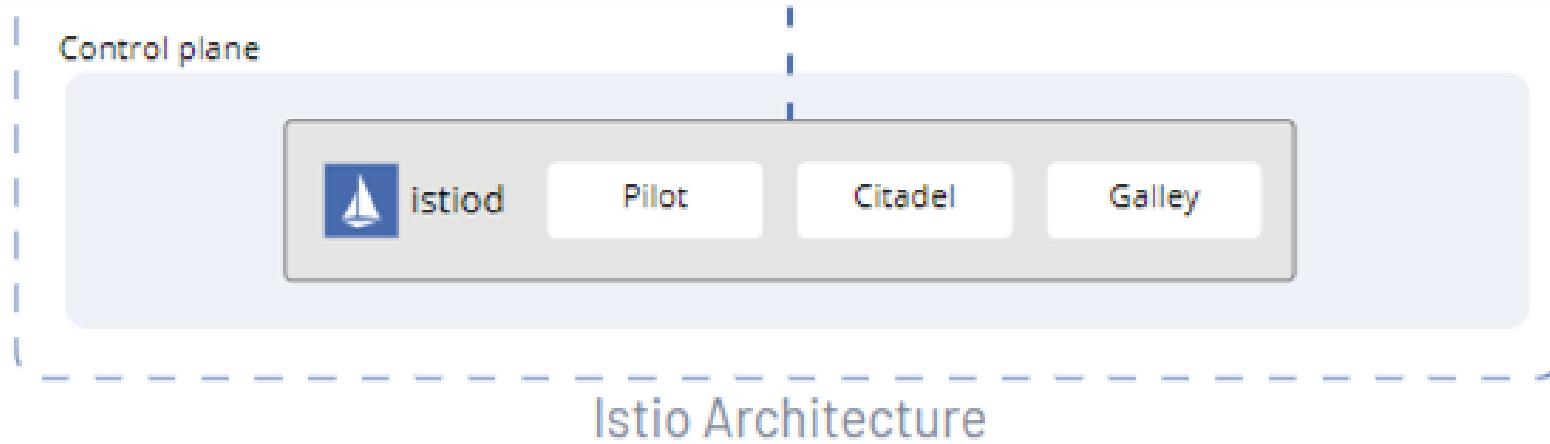
Like all service meshes, an Istio service mesh consists of a data plane and a control plane.

# Service Mesh – Istio Data Plane

The Istio data plane is typically composed of Envoy proxies that are deployed as sidecars within each container on the Kubernetes pod. These proxies take on the task of establishing connections to other services and managing the communication between them.



# Service Mesh – Istio Control Plane



The control plane for Istio is composed of Pilot, Mixer and Citadel

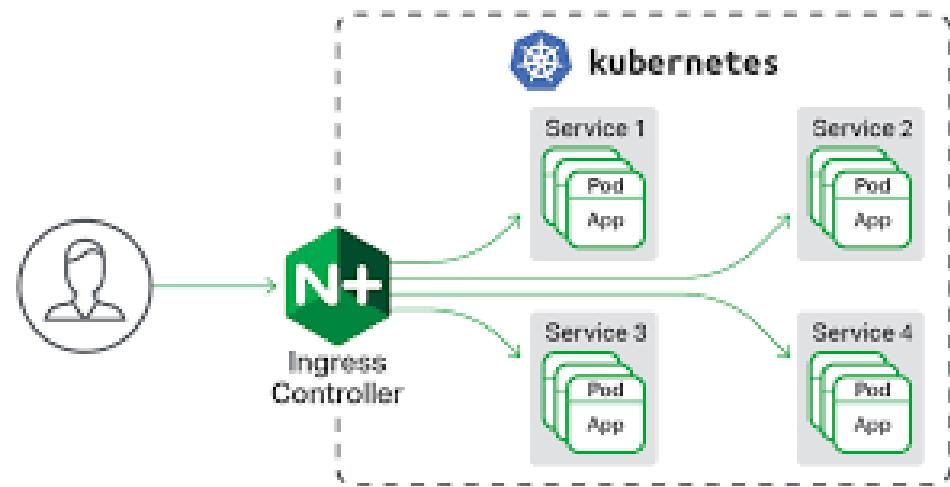
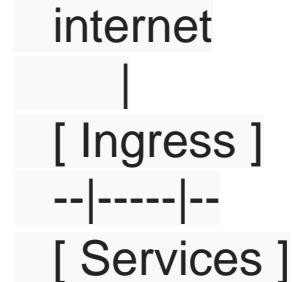
# SM - Istio CP Components

- **Pilot:** responsible for configuring the data plane. Defines proxy behaviours, routing rule between proxies and failure recovery.
- **Mixer:** collects traffic metrics and responds to data plane queries for authorization, access control or quota checks. May interact with logging and monitoring depending on adapter configuration.
- **Citadel:** builds zero-trust environments based on service identity. Works with certificate management



# Ingress

Ingress exposes HTTP and HTTPS routes from outside the cluster to **services** within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

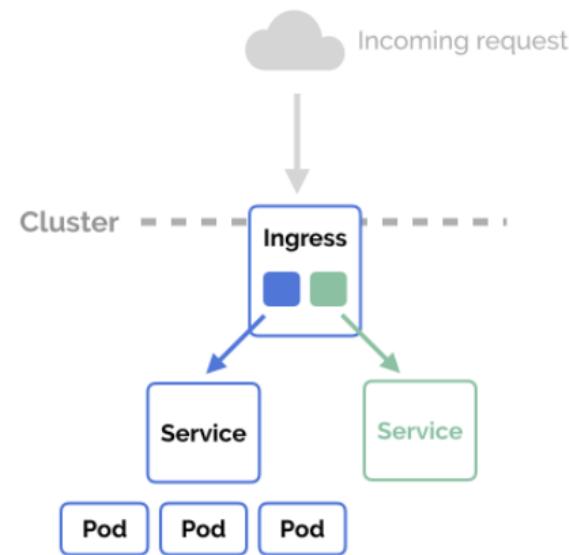


# Kind Ingress

## Setting Up An Ingress Controller

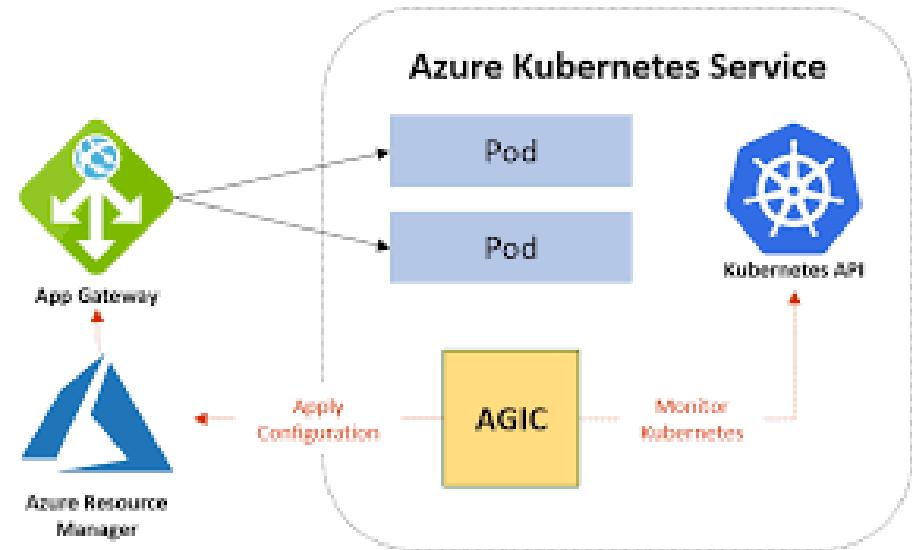
We can leverage KIND's `extraPortMapping` config option when creating a cluster to forward ports from the host to an ingress controller running on a node.

We can also setup a custom node label by using `node-labels` in the `kubeadm InitConfiguration`, to be used by the ingress controller `nodeSelector`.



# Creating a Kubernetes Ingress

To create the ingress controller, normally we use Helm to install a controller like nginx-ingress. For added redundancy, two replicas of the NGINX ingress controllers are deployed. To fully benefit from running replicas of the ingress controller, make sure there's more than one node in your cluster.



# Multi-Cluster Communication

- There are several multicluster service mesh approaches, such as common management, cluster-aware service routing through gateways, flat network and split-horizon endpoints discovery service (EDS).
- Istio has existing multicluster support, additional new functionality in 1.1, and even more appearing in the future.
- External DNS can be used to implement a multi-cluster approach.
- Identify what you want from a solution (single pane of glass observability, unified trust domain etc), and then create a plan on how you implement this.

# POP QUIZ:

## Kubernetes



At its core, Kubernetes is a platform for:

- A: Packaging software containers
- B: Provisioning Machines
- C: Running and scheduling container applications on a cluster



# POP QUIZ:

## Kubernetes



At its core, Kubernetes is a platform for:

- A: Packaging software containers
- B: Provisioning Machines
- C: Running and scheduling container applications on a cluster



# POP QUIZ:

## Kubernetes



In Kubernetes, a node is:

- A: A tool for starting a Kubernetes cluster
- B: A worker machine
- C: A machine that coordinates scheduling container applications on a cluster



# POP QUIZ:

## Kubernetes



In Kubernetes, a node is:

- A: A tool for starting a Kubernetes cluster
- B: A worker machine
- C: A machine that coordinates scheduling container applications on a cluster



# POP QUIZ:

## Kubernetes



What can you deploy on Kubernetes?

- A: Containerized Applications
- B: Virtual Machines
- C: System Processes



# POP QUIZ:

## Kubernetes



What can you deploy on Kubernetes?

- A: Containerized Applications
- B: Virtual Machines
- C: System Processes



# POP QUIZ:

## Kubernetes



What is a Deployment?

- A: A Deployment is responsible for managing the desired state of your applications
- B: A type of container
- C: A type of Kubernetes host



# POP QUIZ:

## Kubernetes



What is a Deployment?

A: A Deployment is responsible for managing the desired state of your applications

B: A type of container

C: A type of Kubernetes host



# POP QUIZ:

## Kubernetes



If a Deployment is exposed publicly, what happens with the network traffic during an update?

- A: Is dropped
- B: Is load-balanced only to the old instances
- C:: Is load-balanced only to available instances (old and new)

# POP QUIZ:

## Kubernetes



If a Deployment is exposed publicly, what happens with the network traffic during an update?

- A: Is dropped
- B: Is load-balanced only to the old instances
- C: Is load-balanced only to available instances (old and new)

# POP QUIZ:

## Kubernetes



Kubernetes is written in which language

- A: Go
- B: C++
- C:: Python.
- D: Java spring framework



# POP QUIZ:

## Kubernetes



Kubernetes is written in which language

- A: Go
- B: C++
- C:: Python.
- D: Java spring framework



# Experiment - Foundation



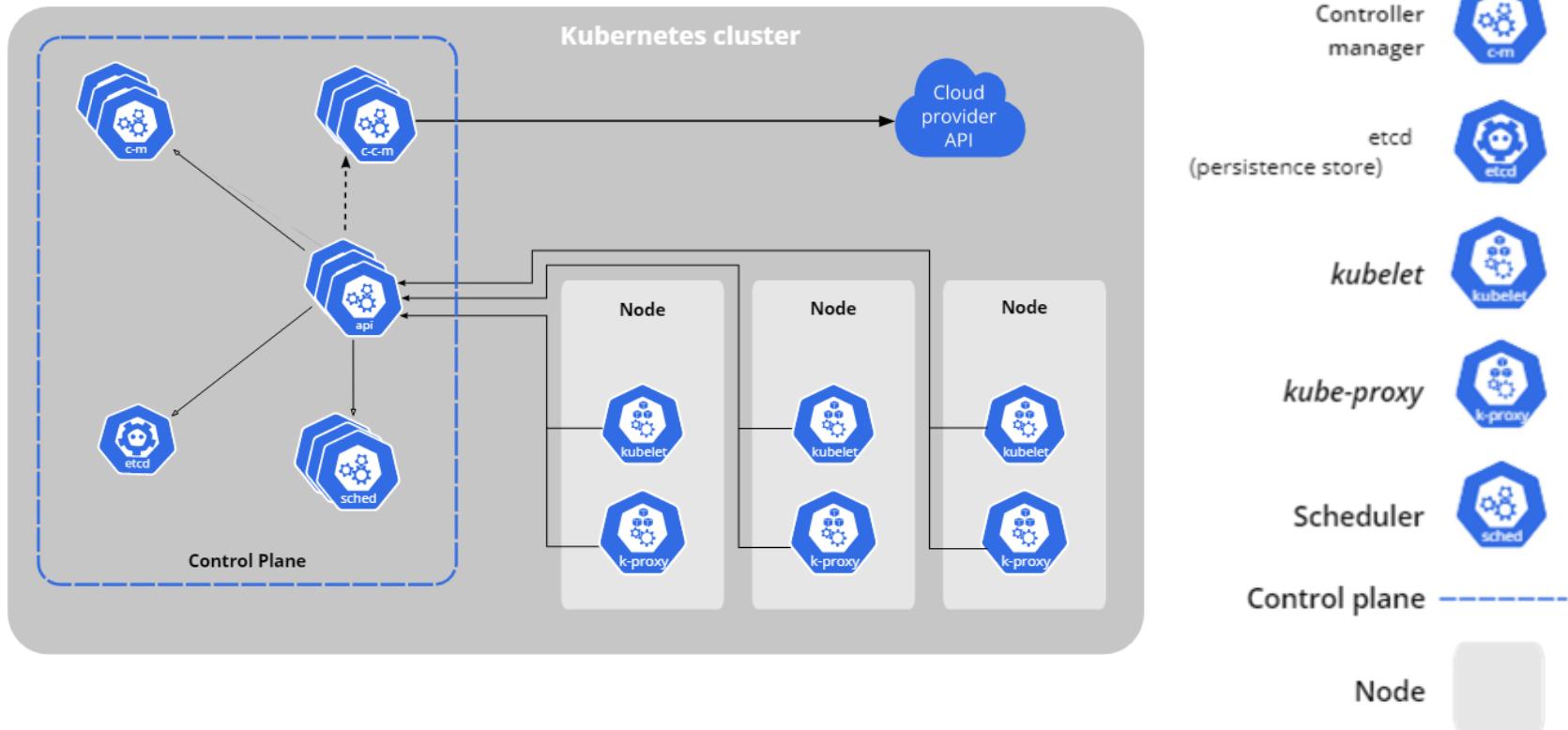
# Experiment – Working with Kind



# East- West Communication

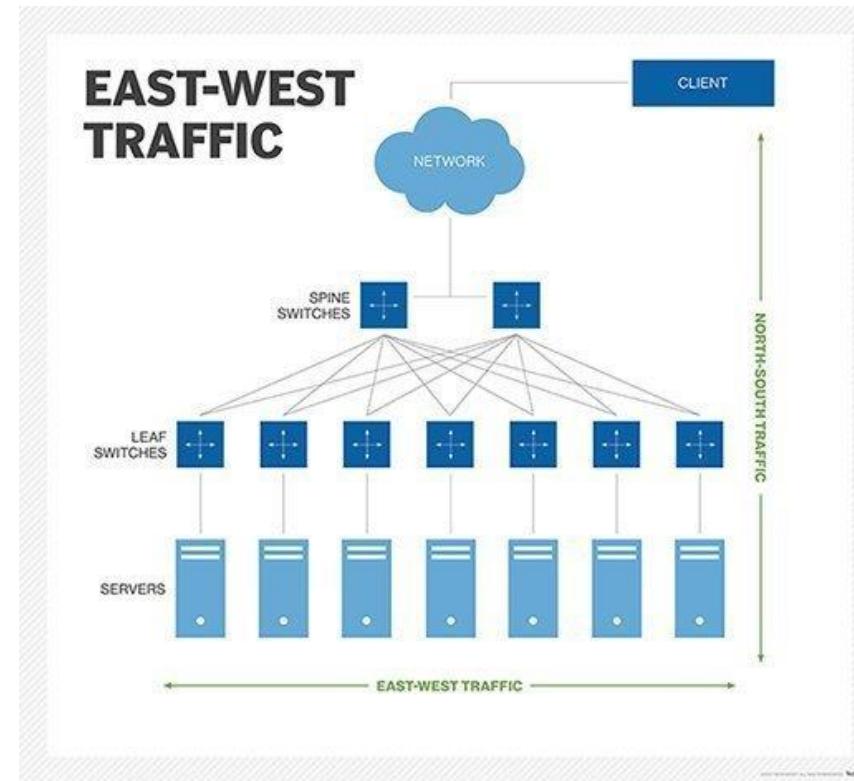


# Refresher

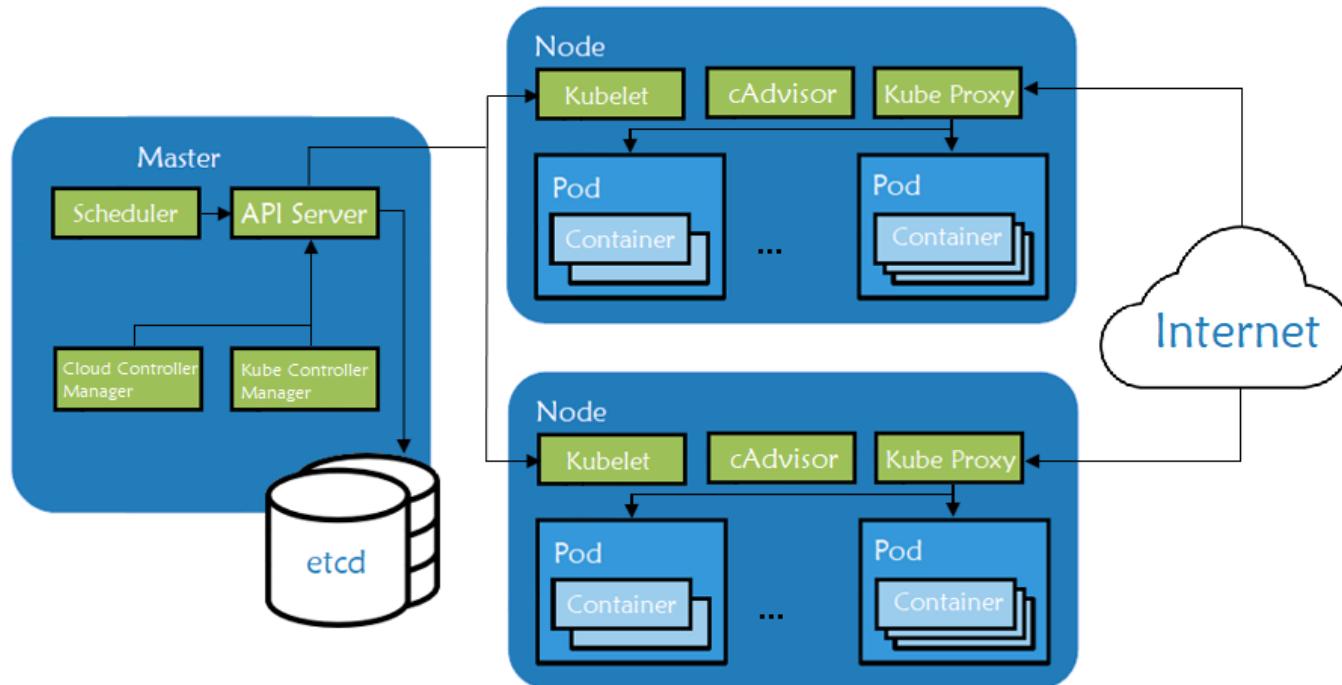


# East West Communication

East-west communication is when services/pods/containers communicate with each other inside the cluster. As you may recall, Kubernetes exposes all the services inside the cluster via both DNS and environment variables.

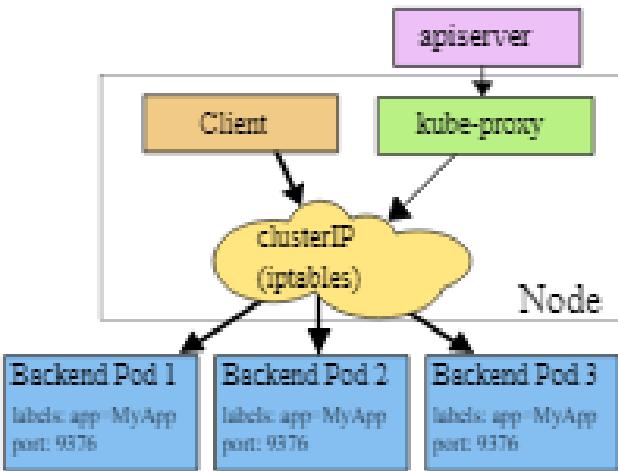


# Kube-proxy Communication



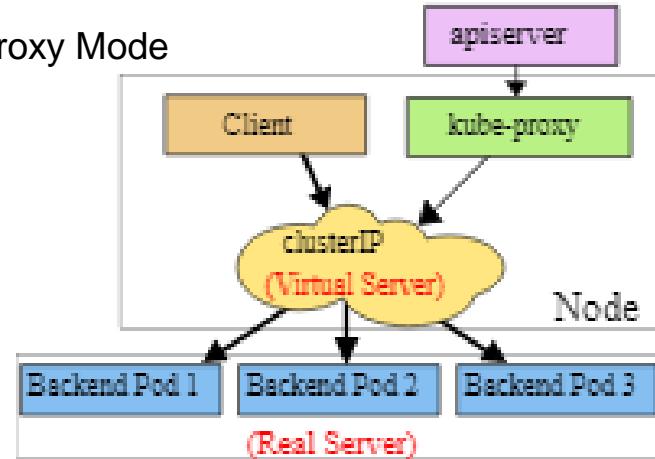
kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.

# Proxy Modes

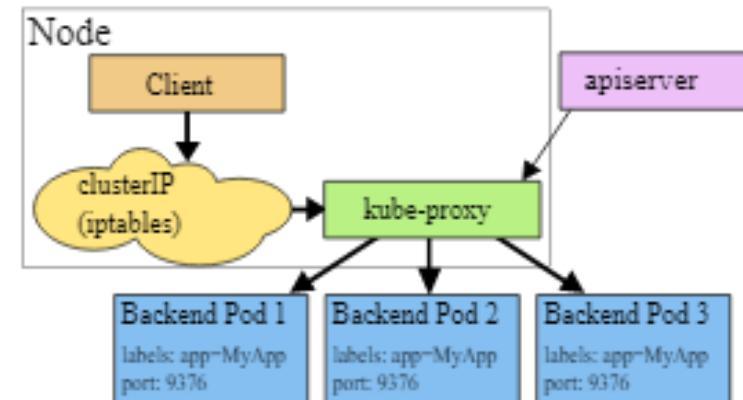


Iptables Proxy

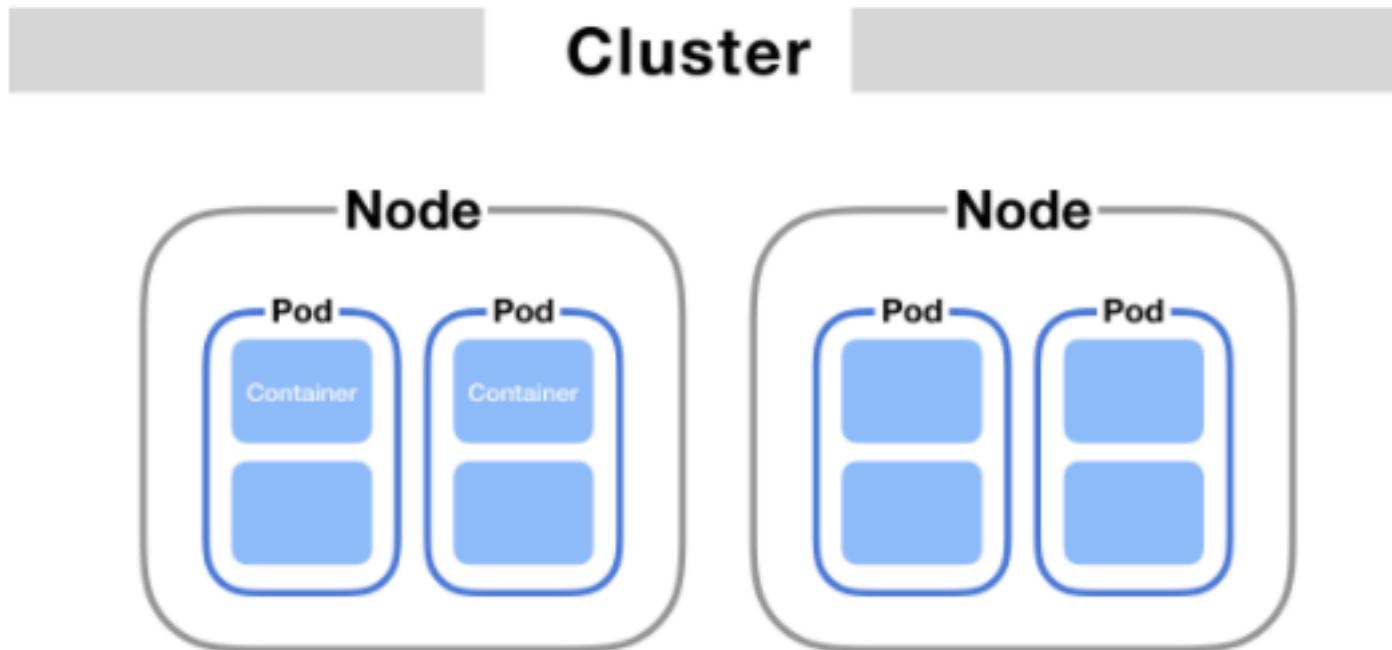
Ipvs Proxy Mode



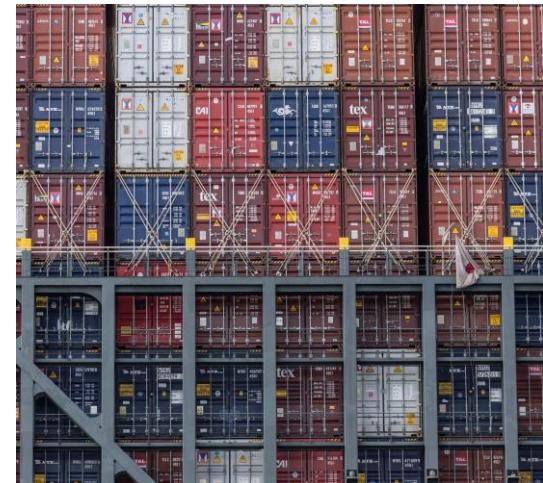
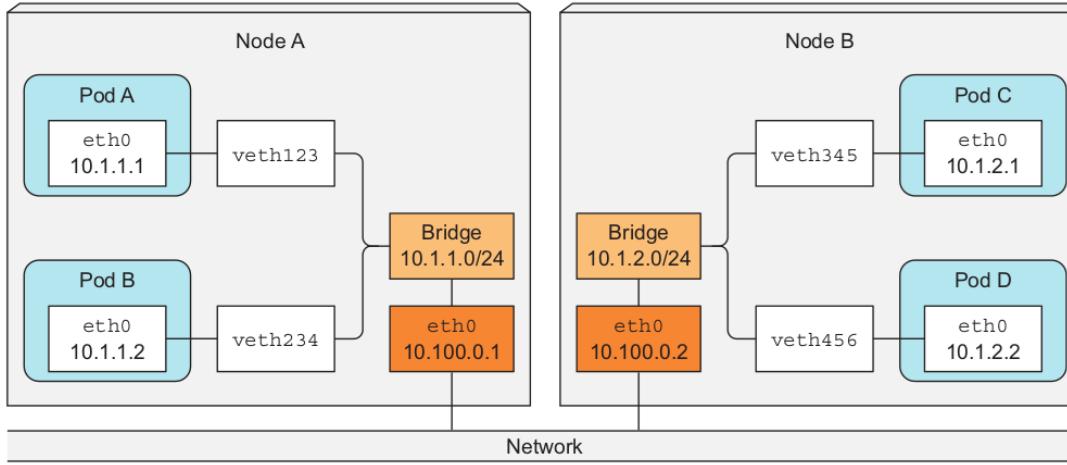
User Space Proxy Mode



# Intra-pod communication (Container to Container)



# Intra-pod communication (Pod to Pod)



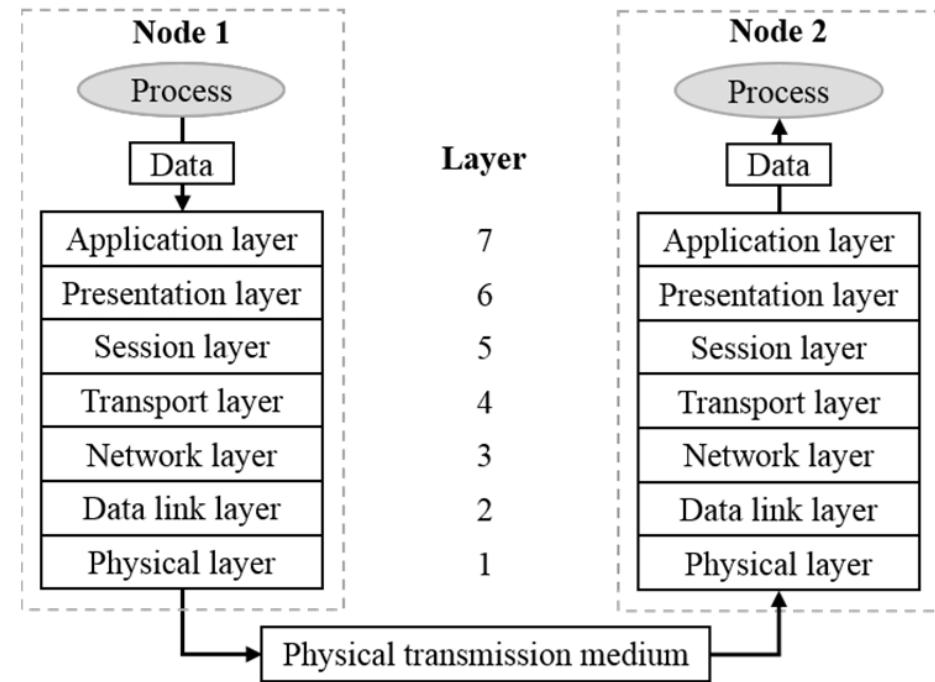
# Intra-pod communication (Pod to Pod)

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- All pods can communicate with all other pods without NAT
- IP that a pod sees itself as is the same IP that other pods see it as

# Node communication

- All nodes running pods can communicate with all pods (and vice-versa) without NAT
- Nodes should be able to talk to all pods.
- Pods should be able to communicate with all nodes.



# What is a Service Mesh

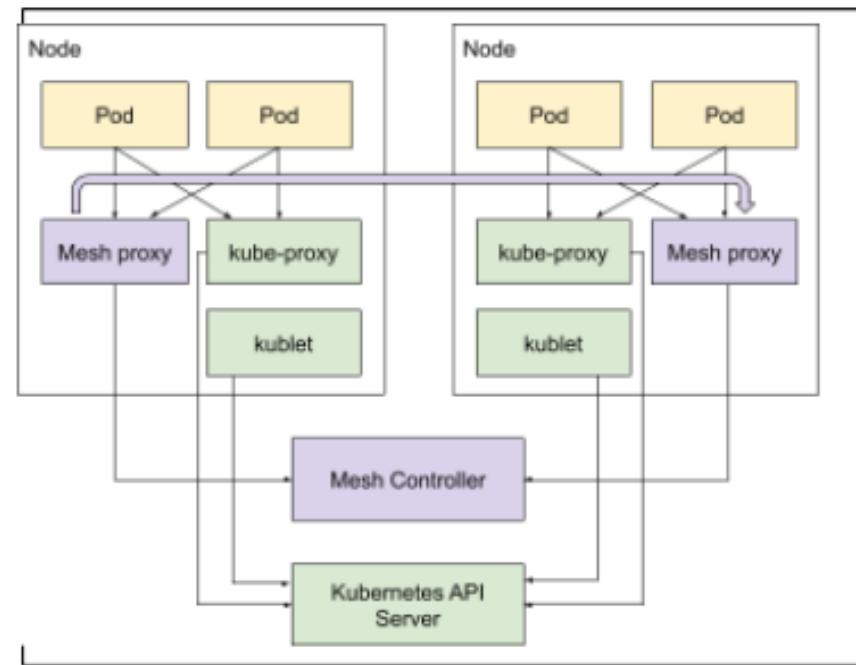
A service mesh is a dedicated infrastructure layer for handling service-to-service communication.

The components:

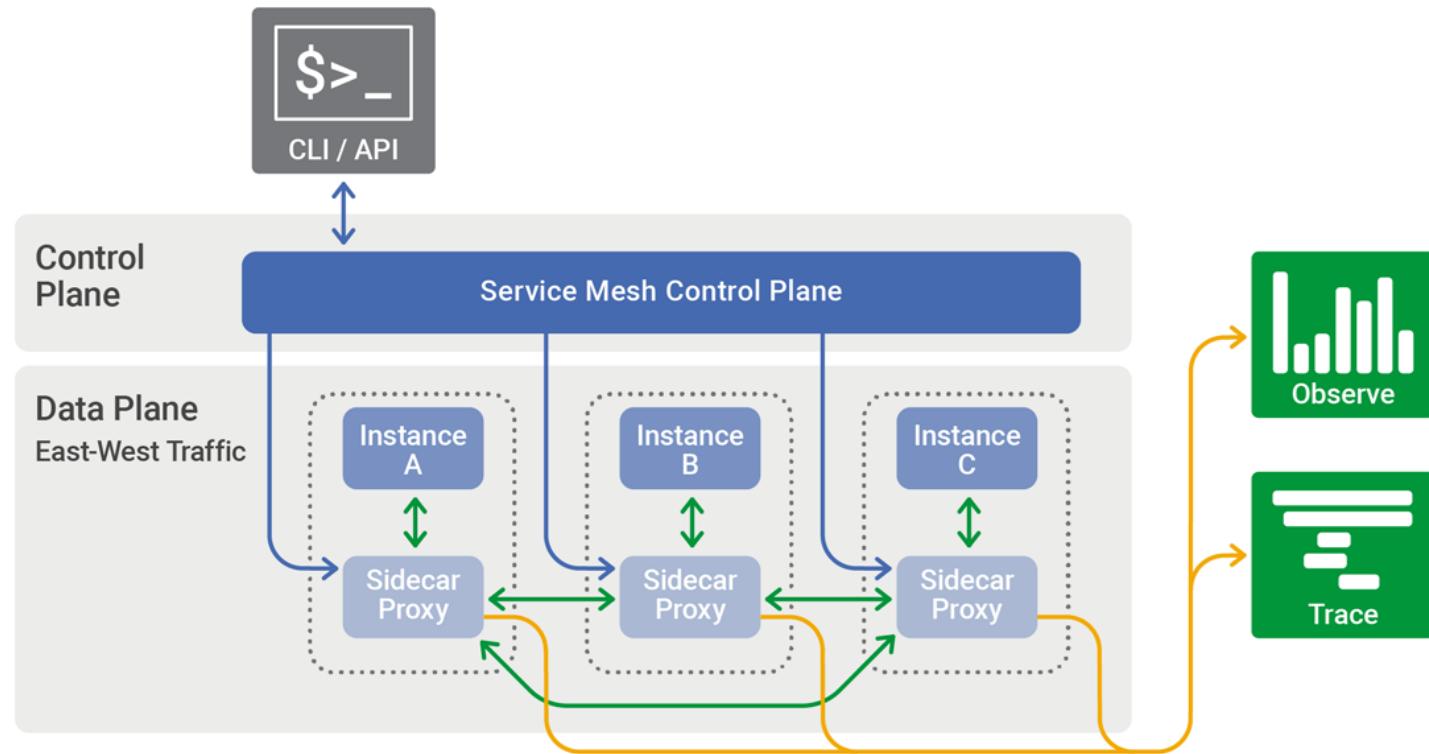
**Data plane** - lightweight proxies that are distributed as sidecars. Proxies include NGINX, or envoy

**Control plane** - provides the configuration for the proxies, issues the TLS certificates authority, and contain the policy managers.

The following diagram illustrates how a service mesh is embedded into a Kubernetes cluster:



# What is a Service Mesh



# Why Service Mesh?

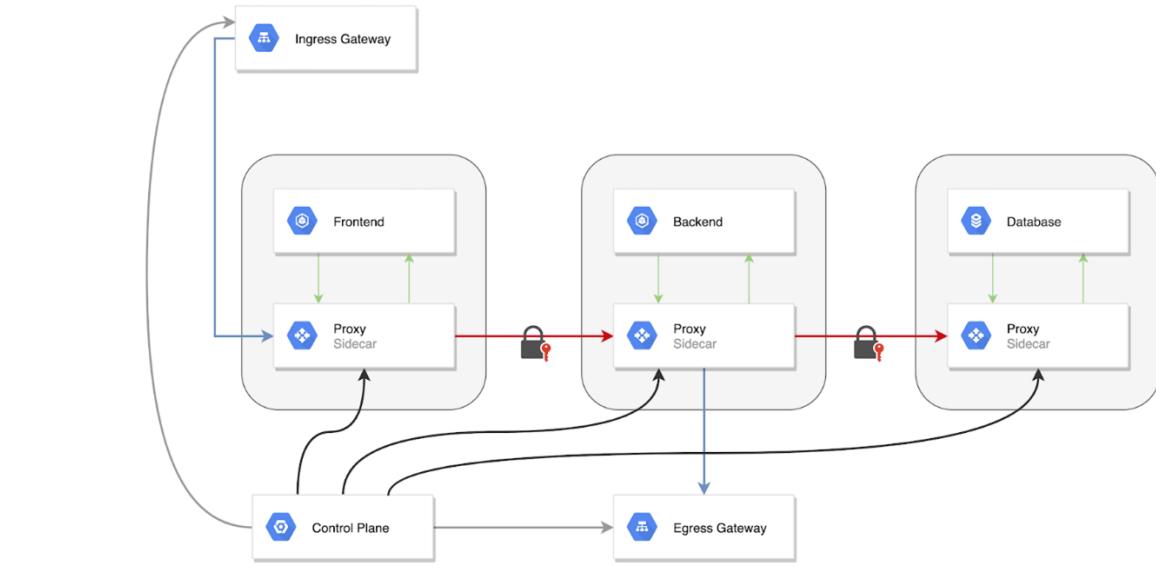
- Advanced load balancing
- Service discovery
- Support canary deployments
- Caching
- Tracing a request across multiple microservices
- Authentication between services
- Throttling the number requests a service handles at a given time
- Automatically retrying failed requests
- Failing over to an alternative component when a component fails consistently
- Collecting metrics

# Service mesh networking

---

Service mesh allows you to separate the business logic of the application from observability, and network and security policies. It allows you to connect, secure, and monitor your microservices.

## Service Mesh Traffic overview



# Optimizing Communication

a service mesh also captures every aspect of service-to-service communication as performance metrics. Over time, data made visible by the service mesh can be applied to the rules for interservice communication, resulting in more efficient and reliable service requests.

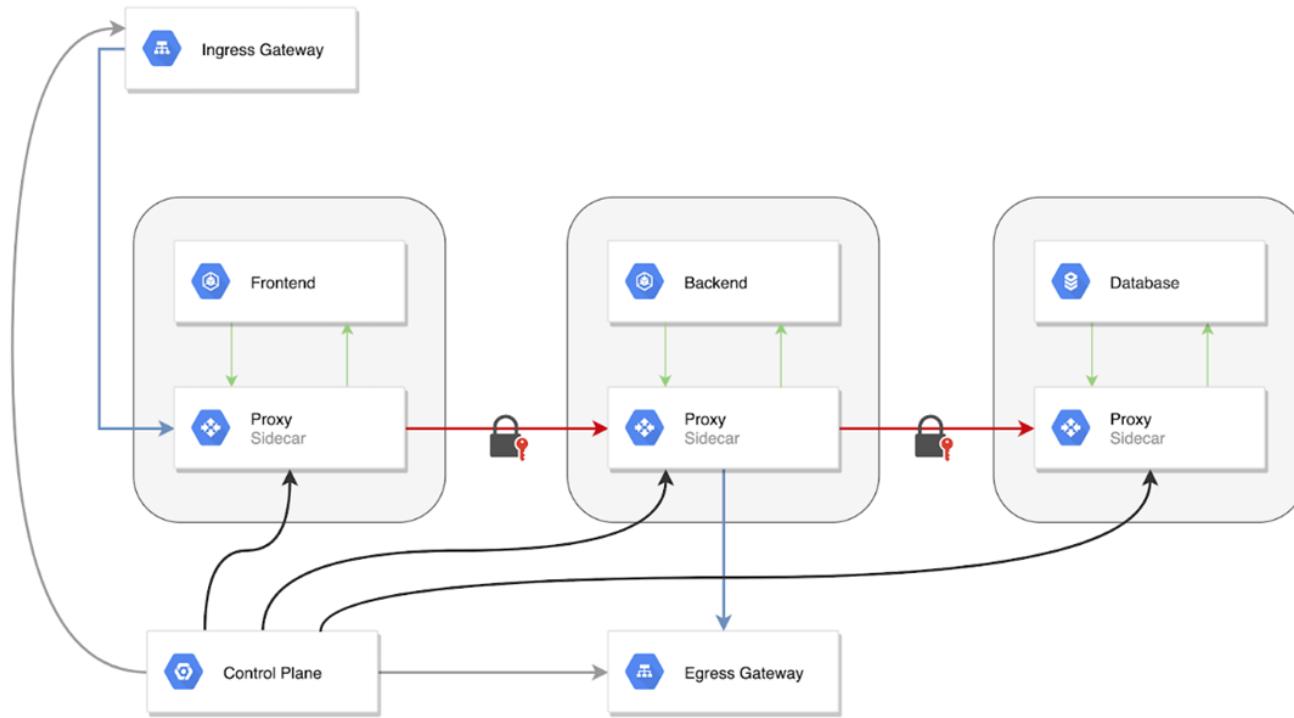


# Observability

Many organizations are initially attracted to the uniform observability that service meshes provide. No complex system is ever fully healthy. Service-level telemetry illuminates where your system is behaving sickly, illuminating difficult-to-answer questions like why your requests are slow to respond



# Kubernetes and Service Mesh



# Implementations

**Istio**

**AWS App Mesh**

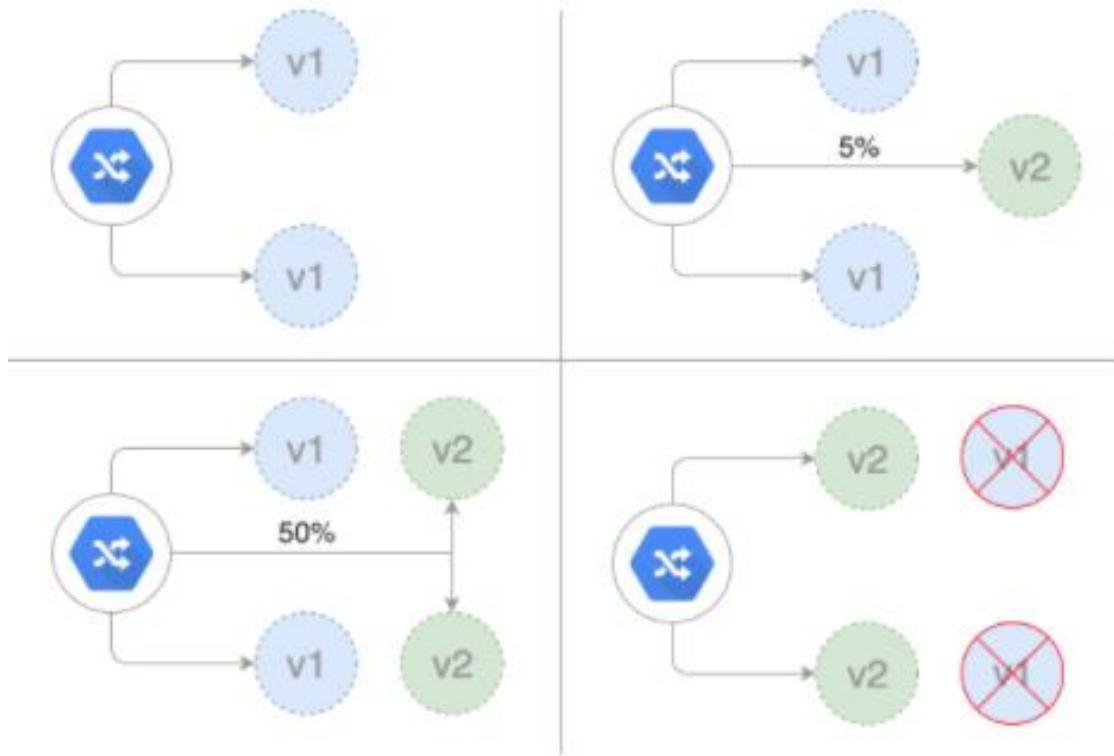
**Linkerd v2**

**Consul Connect**

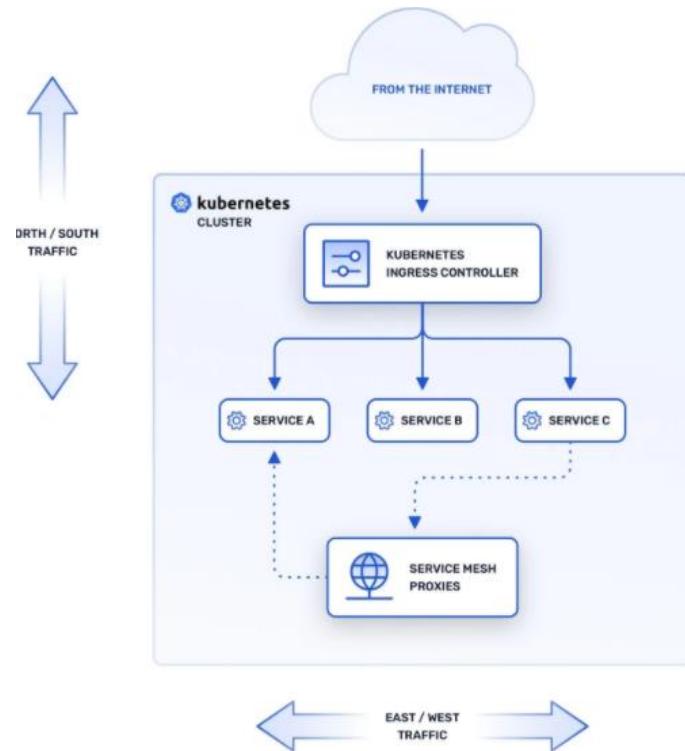


# Canary

A canary is used for when you want to test some new functionality typically on the backend of your application.



# Problem Statement



# Architecture



# Best Practices

- Auto-inject proxies
- Focus on automation
- Monitor and Trace everything



# Implementing Retry Logic

**Implement retry logic** only where the full context of a failing operation is understood. For example, if a task that contains a **retry** policy invokes another task that also contains a **retry** policy, this extra layer of **retries** can add long delays to the processing

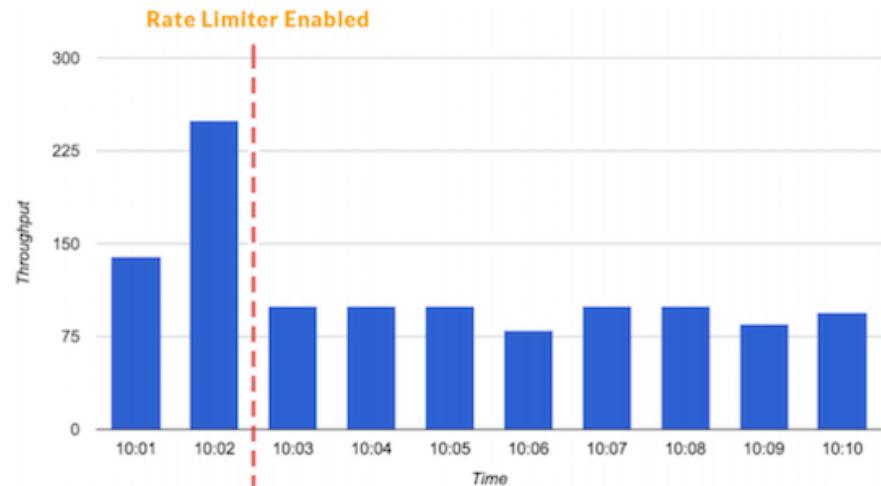
Object	API	Version
VirtualService	networking.istio.io	v1alpha3



# Rate limiters & Load Shredders

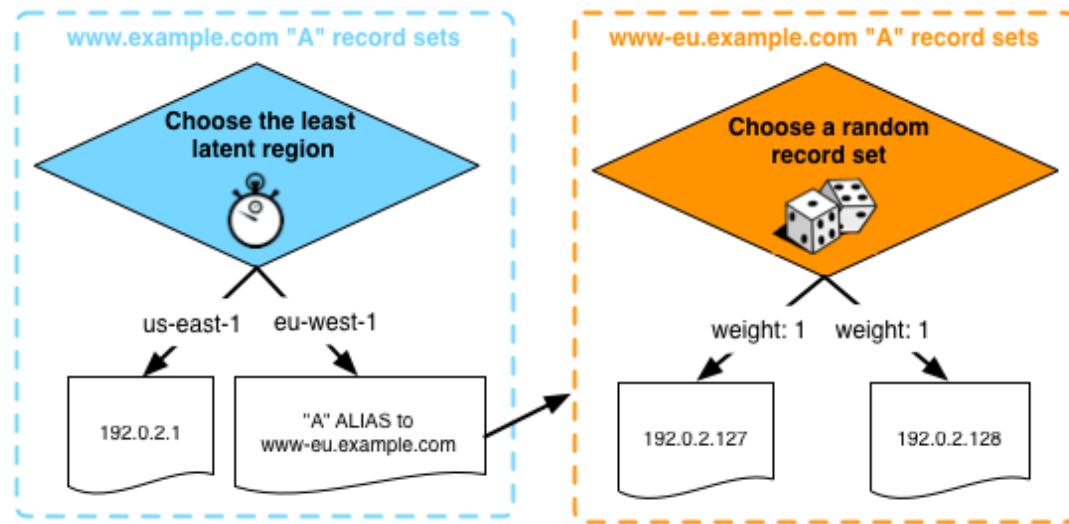
Rate limiting is the technique of defining how many requests can be received or processed by a particular customer or application during a timeframe

A *fleet usage load shredder* can ensure that there are always enough resources available to serve critical transactions. It keeps some resources for high priority requests and doesn't allow for low priority transactions to use all of them



# Implementing Weighted Routing

Weighted routing lets you associate multiple resources with a single domain name (example.com) or subdomain name (acme.example.com) and choose how much traffic is routed to each resource.



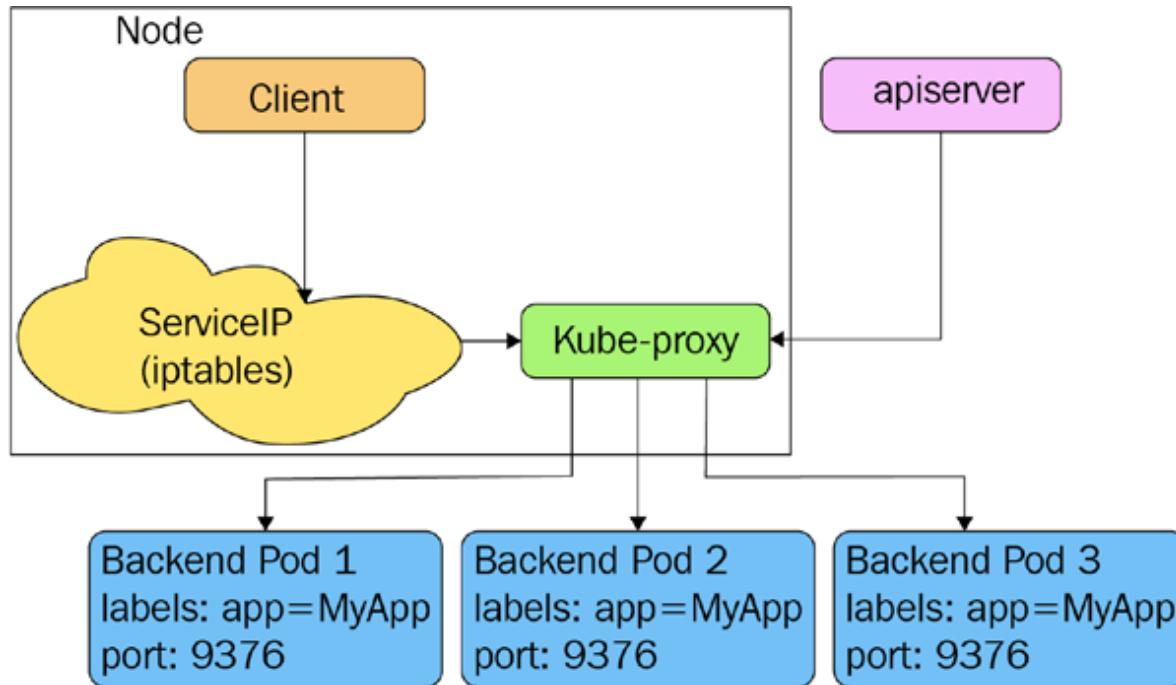
# Implementing Access Controls

tips for implementing access control systems:

1. Implement a central repository with well-defined whitelisting policies. ...
2. Solve self-generated scripts. ...
3. Withdraw your departing employees' digital rights. ...
4. Adapt your **access control**. ...
5. Create consistent processes to whitelist new cloud applications



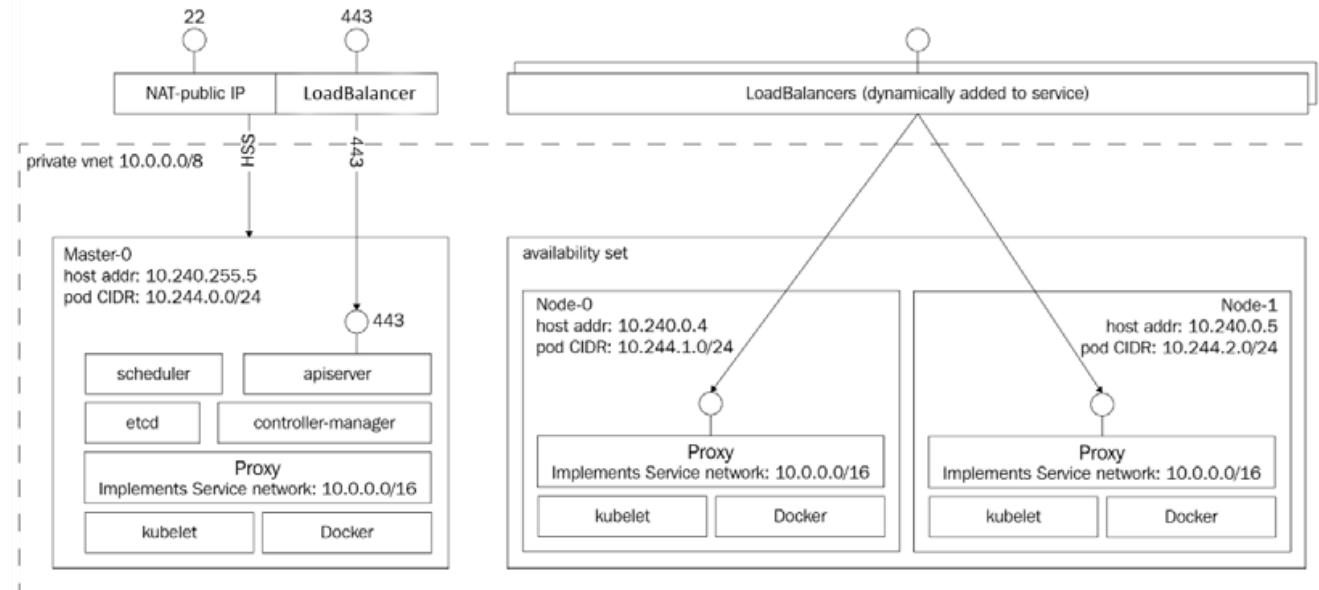
# Pod to Service Communication



Pods can talk to one another directly using their IP addresses and well-known ports, but that requires the pods to know each other's IP addresses.

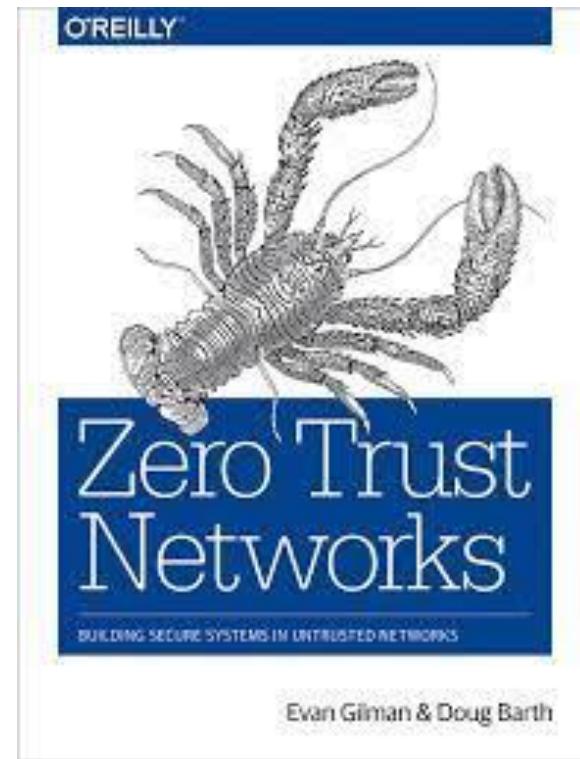
# External Access

some containers need be accessible from the outside world. The pod IP addresses are not visible externally. The service is the right vehicle, but external access typically requires two redirects.



# Zero Trust Networks

The Zero Trust Model is simple: cybersecurity professionals must stop trusting packets as if they were people. Instead, they must eliminate the idea of a trusted network (usually the internal network) and an untrusted network (external networks). In Zero Trust, all network traffic is untrusted. - Forrester



Evan Gilman & Doug Barth

# POP QUIZ:

## KUBERNETES DEVOPS



What is service mesh?

- A: a dedicated infrastructure layer for handling service-to-service communication.
- B: A lightweight proxy that is distributed as sidecars.
- C: lets you associate multiple resources with a single domain name



# POP QUIZ:

## KUBERNETES DEVOPS



What is service mesh?

A: a dedicated infrastructure layer for handling service-to-service communication.

B: A lightweight proxy that is distributed as sidecars.

C: lets you associate multiple resources with a single domain name



# POP QUIZ:

## KUBERNETES DEVOPS



What two fields configure a retry?

- A: Attempts & perTryTimeout
- B: Rate Limiter & Load shedders
- C: NameSpace & ReplicaSet



# POP QUIZ:

## KUBERNETES DEVOPS



What two fields configure a retry?

- A: Attempts & perTryTimeout
- B: Rate Limiter & Load shedders
- C: NameSpace & ReplicaSet

# POP QUIZ:

## Kubernetes DEVOPS



Which of the following process runs on Kubernetes non-master node?

- A: Kube-proxy
- B: Kube-apiserver
- C:: Both
- D: Neither



# POP QUIZ:

## Kubernetes DEVOPS



Which of the following process runs on Kubernetes non-master node?

- A: Kube-proxy
- B: Kube-apiserver
- C:: Both
- D: Neither



# POP QUIZ:

## Kubernetes DEVOPS



Which of the following is a kubernetes controller?

- A: Replicaset
- B: Deployment
- C:: Namespace
- D: Both Replicaset & Deployment



# POP QUIZ:

## Kubernetes DEVOPS



Which of the following is a kubernetes controller?

- A: Replicaset
- B: Deployment
- C:: Namespace
- D: Both Replicaset & Deployment



# Experiment – Kind Ingress



# Experiment – Building Troubleshooting Images

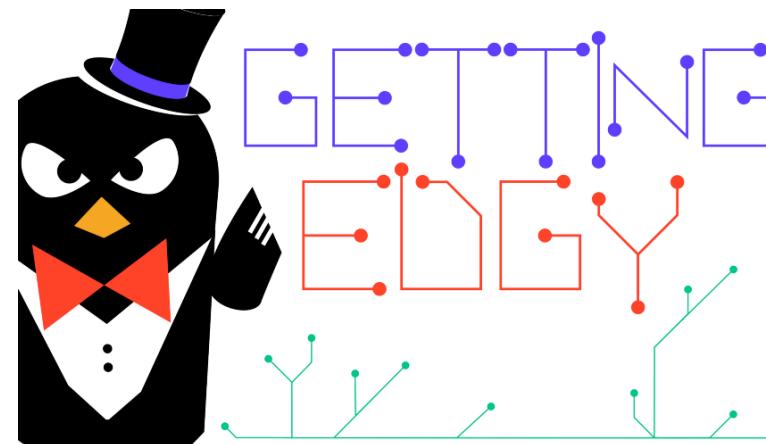


# North- South Communication



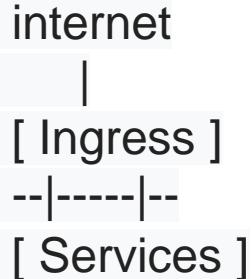
# North South Communication

North/south traffic is traffic flowing from the user into the cluster, through the ingress.



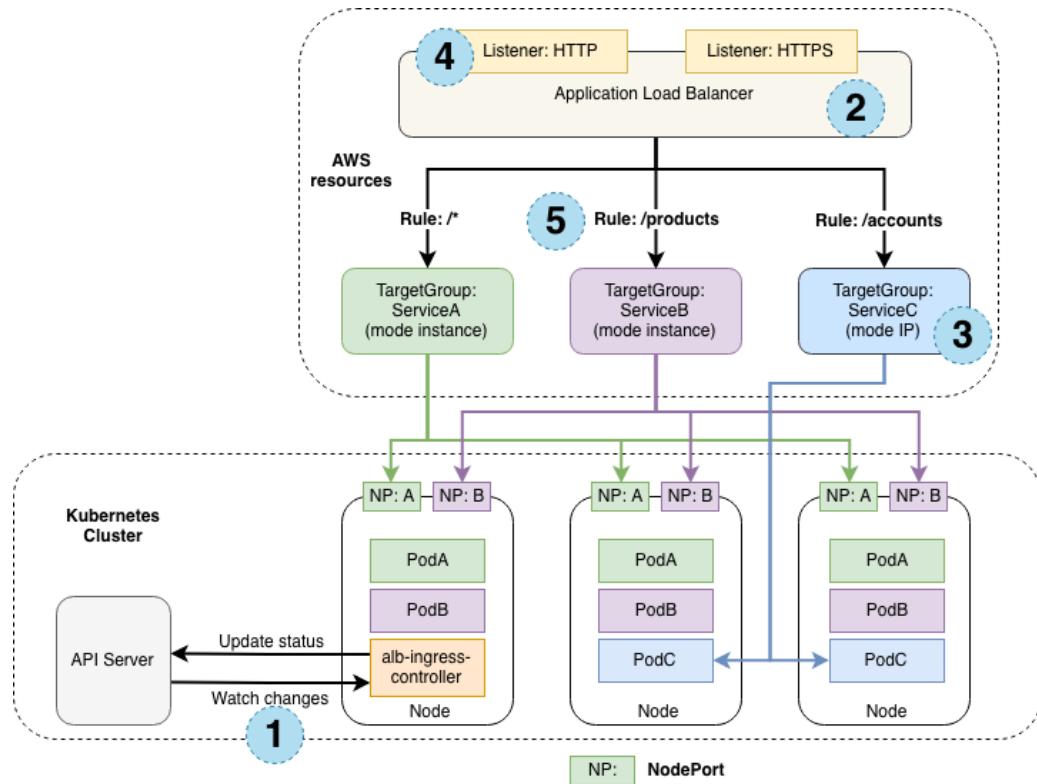
# Ingress

Ingress exposes HTTP and HTTPS routes from outside the cluster to `services` within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.



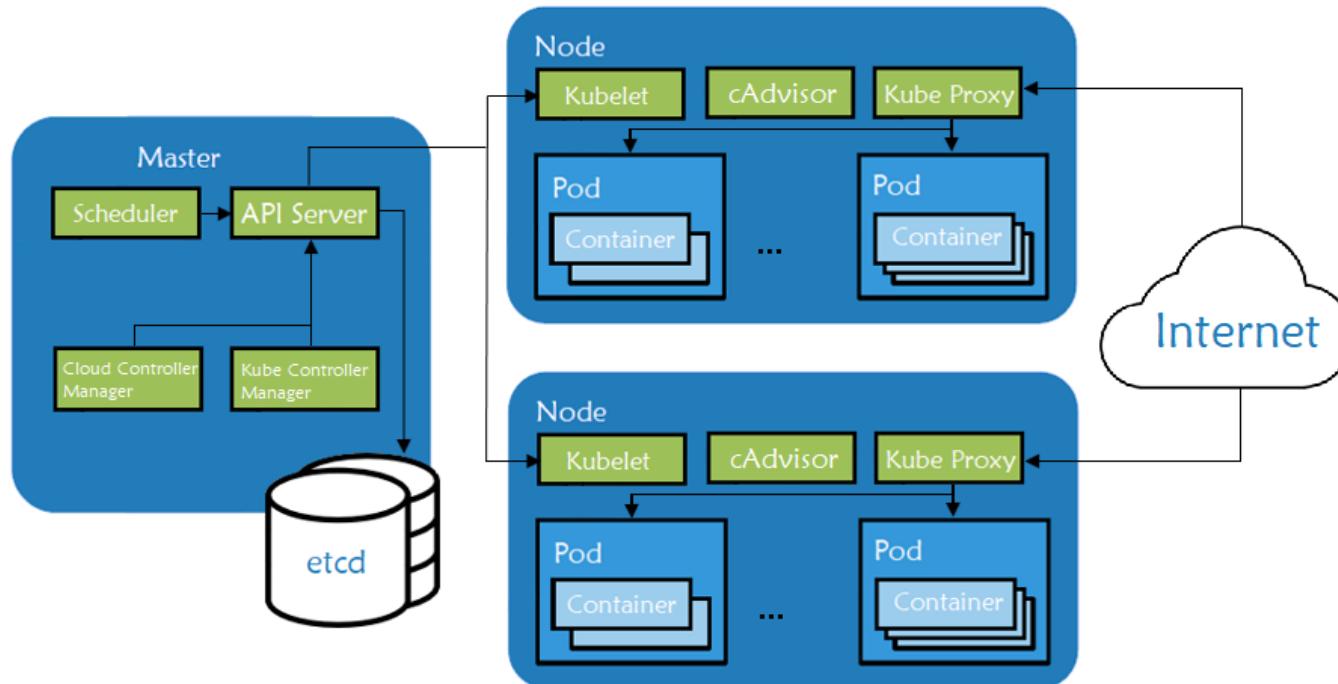
```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        backend:
          serviceName: test
          servicePort: 80
```

# Ingress Controllers



In order for the Ingress resource to work, the cluster must have an ingress controller running.

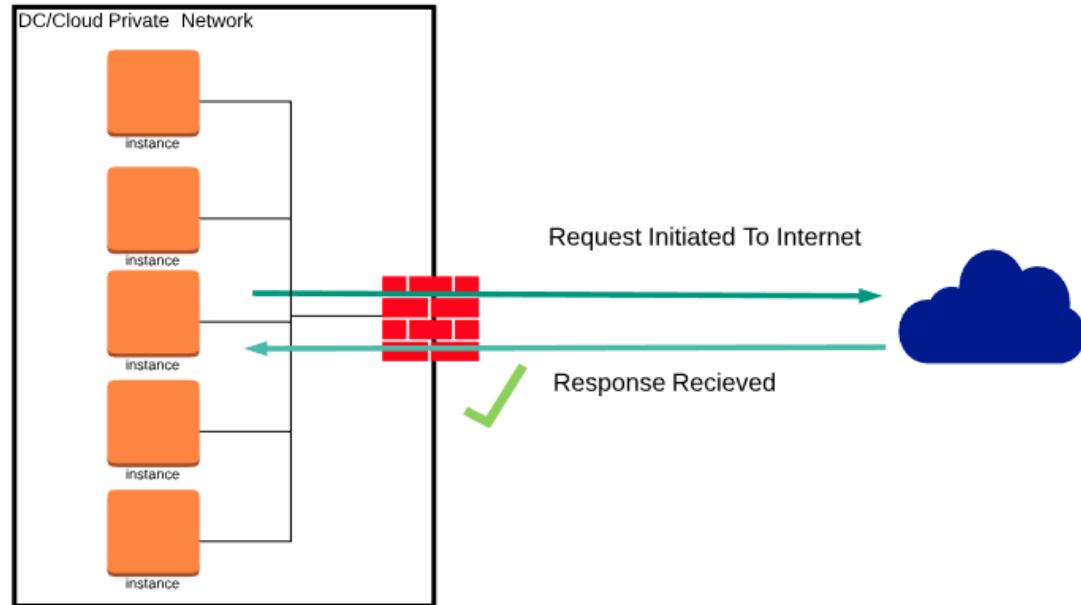
# Kube-proxy Communication



kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.

# Egress

From the point of view of a Kubernetes pod, **egress** is outgoing traffic from the pod.



# Egress Network Policies

Denotes any network policy that applies to egress  
(irrespective of whether the policy also applies to ingress).

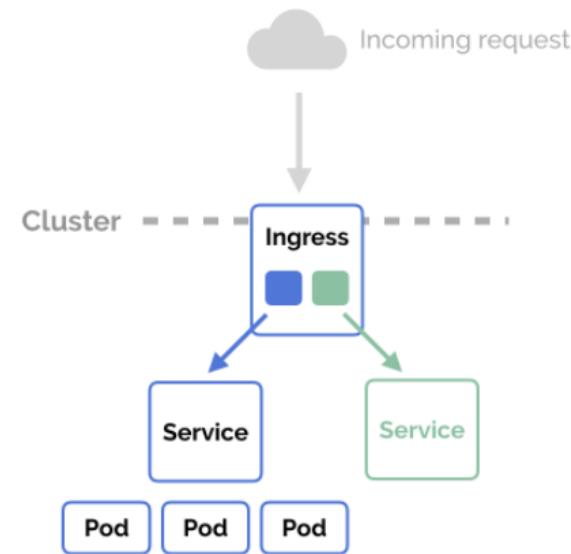
By default, if *no* egress network policy applies to a pod, it is *non-isolated* for egress. (Note that isolation is evaluated independently for ingress and egress; it is possible for a pod to be isolated for neither, for exactly one, or for both). When a pod is non-isolated for egress, all network egress is allowed from the pod.

# Creating a Kubernetes Ingress

## Setting Up An Ingress Controller

We can leverage KIND's `extraPortMapping` config option when creating a cluster to forward ports from the host to an ingress controller running on a node.

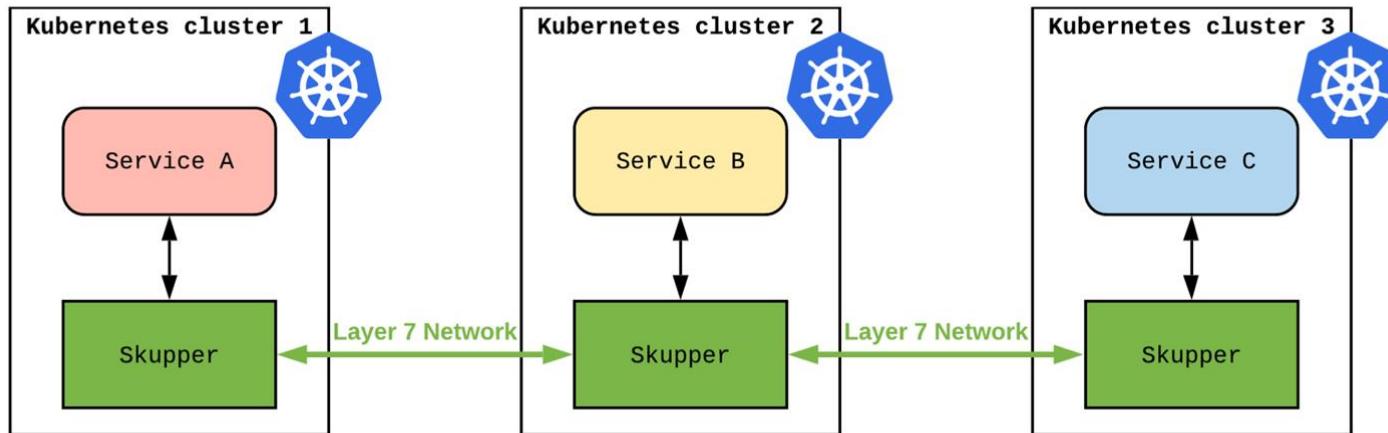
We can also setup a custom node label by using `node-labels` in the `kubeadm InitConfiguration`, to be used by the ingress controller `nodeSelector`.



# Creating a Kubernetes Egress

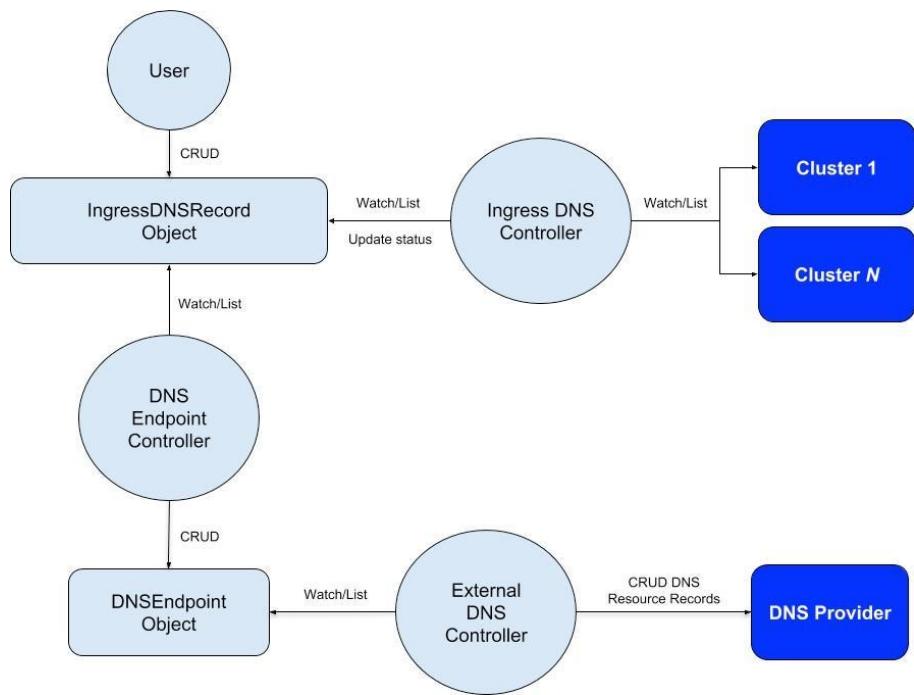
# Multi-Cluster Communication

Multi-cluster is a strategy for deploying an application on or across multiple Kubernetes clusters with the goal of improving availability, isolation, and scalability. Multi-cluster can be important to ensure compliance with different and conflicting regulations, as individual clusters can be adapted to comply with geographic- or certification-specific regulations.



# Utilizing Multi-Cluster Ingress DNS

Ingress in a single cluster is done at the edge of the cluster and forwards traffic into the cluster. However, in the multi-cluster world, the situation is different. The cluster may be needed to send requests from the receiving cluster to a different cluster. Finding the correct destination relies on an external DNS, which is used in addition to the in-cluster CoreDNS. The primary idea is that the endpoints from all of the clusters are managed by a DNS endpoint controller and an **Ingress DNS controller**.



# POP QUIZ:

## Kubernetes



Replication Controllers and Deployment Controllers are part of

- A: API Controller Manager
- B: Etcd Manager
- C: Master Controller Manager
- D: Pod Networking



# POP QUIZ:

## Kubernetes



Replication Controllers and Deployment Controllers are part of

- A: API Controller Manager
- B: Etcd Manager
- C: Master Controller Manager
- D: Pod Networking



# POP QUIZ:

## Kubernetes



If a Deployment is exposed publicly, what happens with the network traffic during an update?

- A: Is dropped
- B: Is load-balanced only to the old instances
- C:: Is load-balanced only to available instances (old and new)



# POP QUIZ:

## Kubernetes



What is the basic operational unit of Kubernetes?

- A: Task
- B: Pod
- C:: Names
- D: Container



# POP QUIZ:

## Kubernetes



What is the basic operational unit of Kubernetes?

- A: Task
- B: Pod
- C:: Names
- D: Container



# POP QUIZ:

## Kubernetes



Which of the following runs on each node and insures containers are running a pod?

- A: Pod
- B: Kubelet
- C:: Ect.
- D: Scheduler

# POP QUIZ:

## Kubernetes



Which of the following runs on each node and insures containers are running a pod?

- A: Pod
- B: Kubelet
- C:: Ect.
- D: Scheduler



# POP QUIZ:

## Kubernetes



Which one of the following can be considered as the primary data store of Kubernetes?

- A: Pod
- B: Node
- C:: Ect.
- D: Container

# POP QUIZ:

## Kubernetes



Which one of the following can be considered as the primary data store of Kubernetes?

- A: Pod
- B: Node
- C:: Ect.
- D: Container



# Experiment – Container Communication

## Eavesdropping



# Traffic Flow & Security



# Requirements

- Keeping the load balancer (LB) configuration in sync with the infrastructure
- Security for north-south traffic
- Central controller for large deployments
- Access control between microservices
- Encryption for east-west traffic
- Application traffic analytics
- Application delivery controller

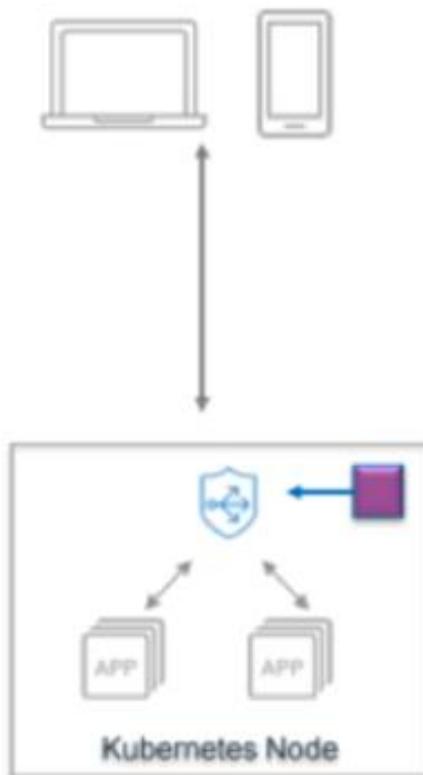
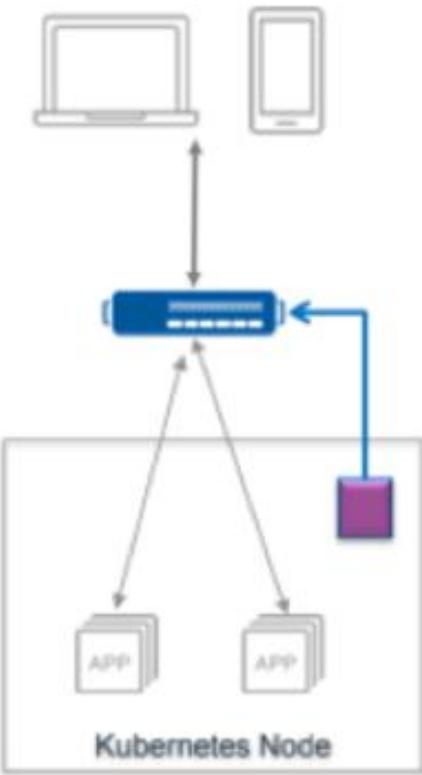


# Traffic Flow & Security Needs

- Keeping the load balancer (LB) configuration in sync with the infrastructure
- Security for north-south traffic
- Central controller for large deployments
- Access control between microservices
- Encryption for east-west traffic
- Application traffic analytics
- Application delivery controller



# Synching the Load Balancer

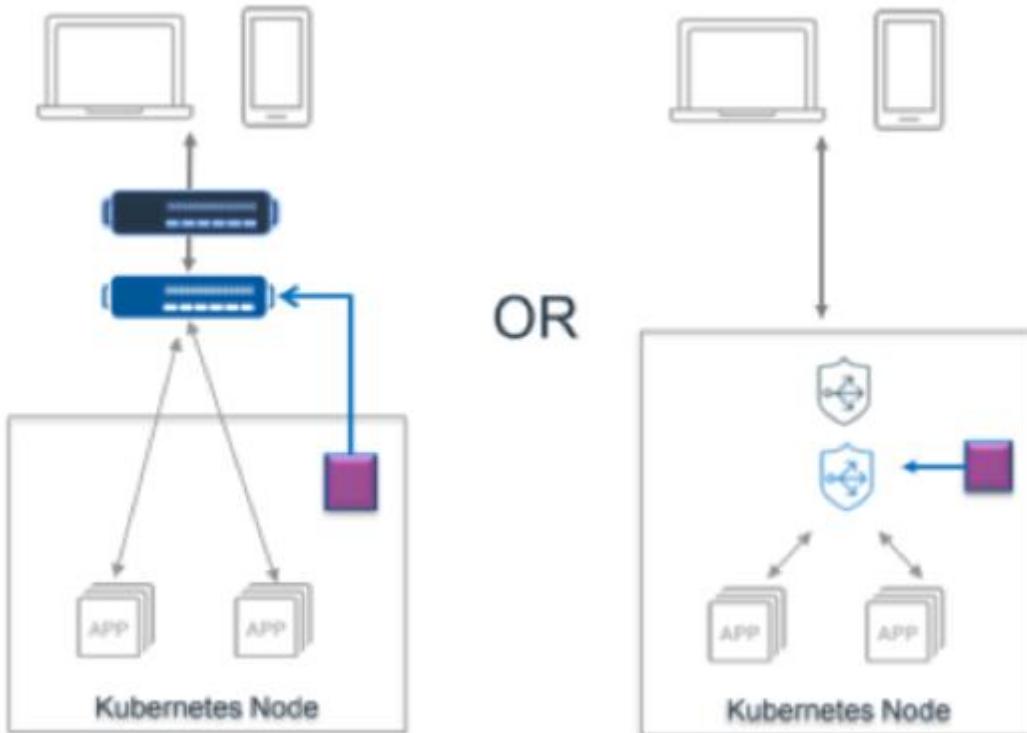


Kubernetes replaces pods as they  
are ill

IP address of the pod  
changes

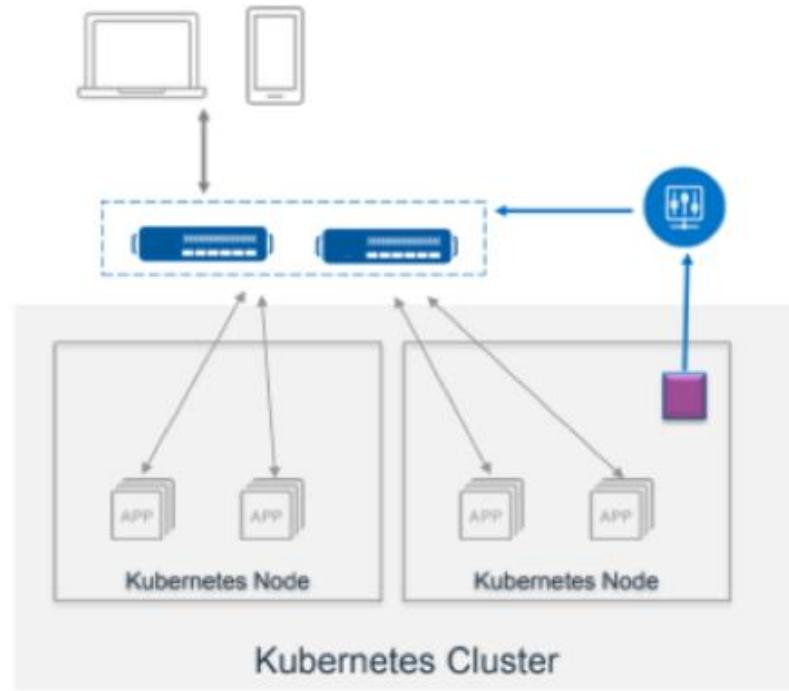
Ingress Controller is defined by  
Kubernetes but not implemented

# Security for North-South Traffic

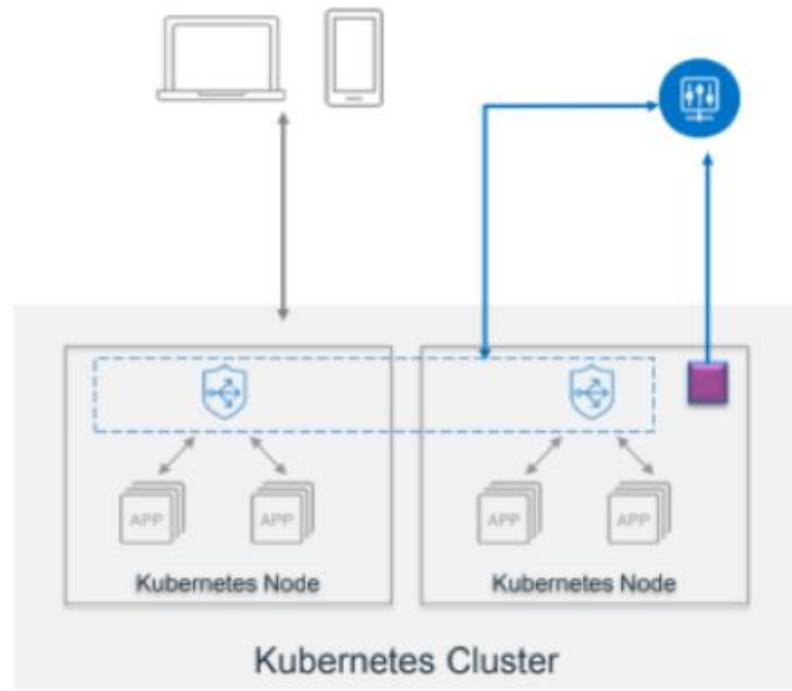


- SSL Offload
- HTTP 2.0
- OWASP Top 10
- Malware, Bad BOTs
- Application DDoS attack
- Traffic Management/ Observability

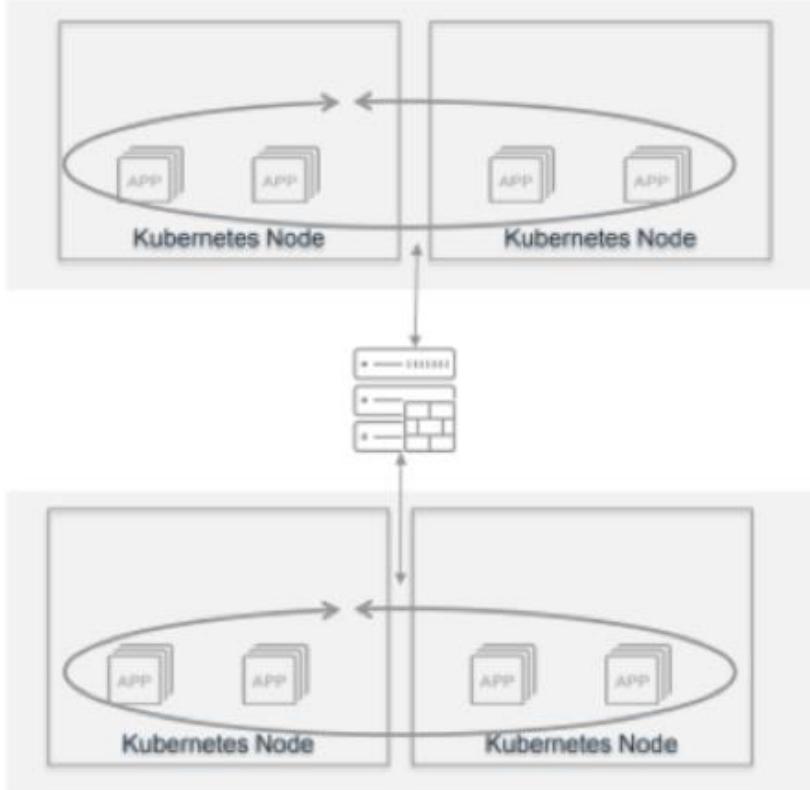
# Central Controller for Large Deployments



OR

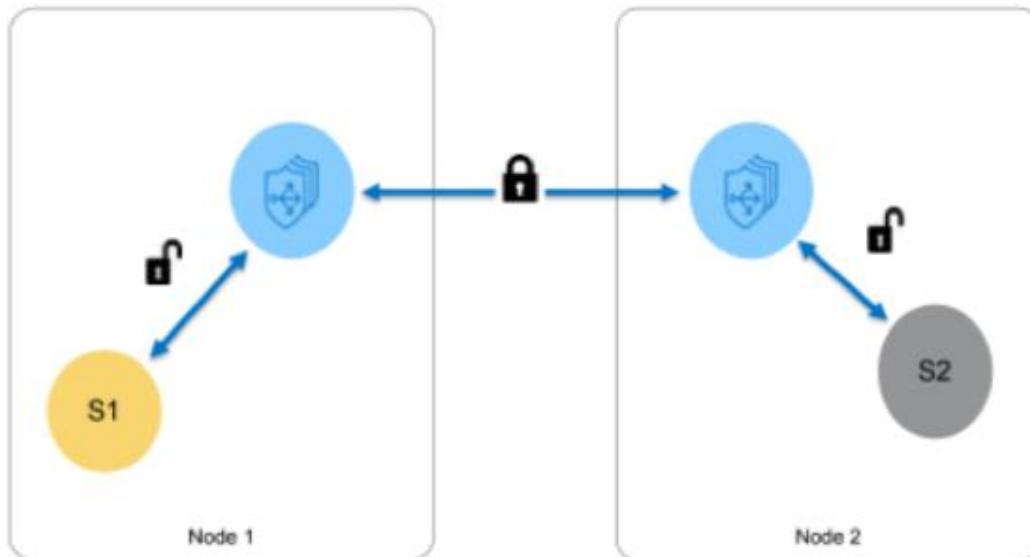


# Microservices Access Control



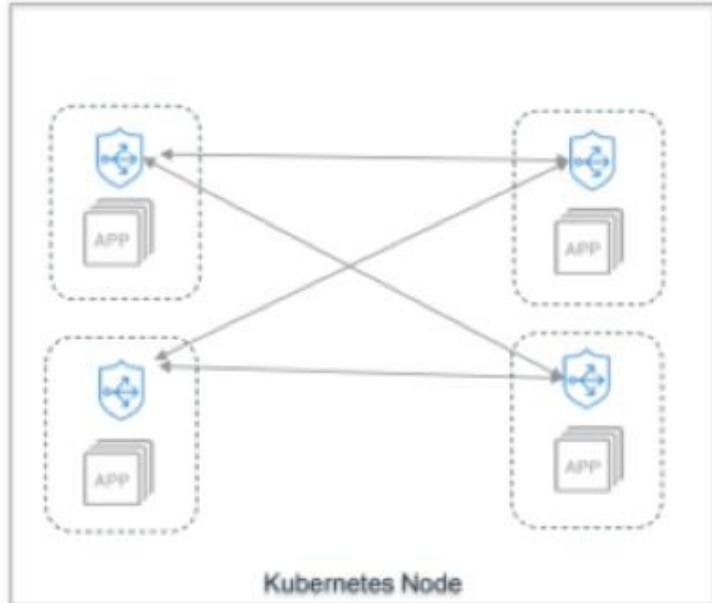
- For Security and Compliance reasons, communication between microservices must be controlled
- In absence of logical policy enforcement, organizations are isolating clusters

# Encryption for East-West Traffic



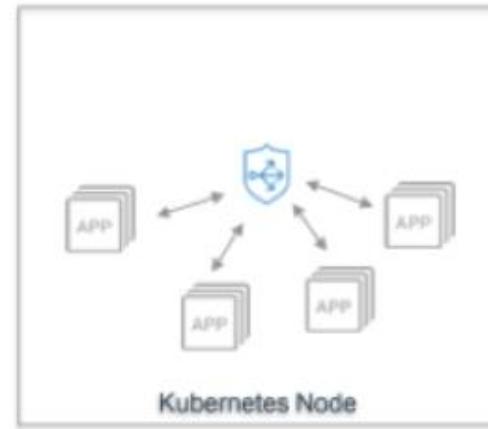
- Traffic when flowing between nodes should be encrypted
- Load balancer should take care of SSL encryption/decryption transparently

# Encryption Patterns



Sidecar Proxy Deployment

OR

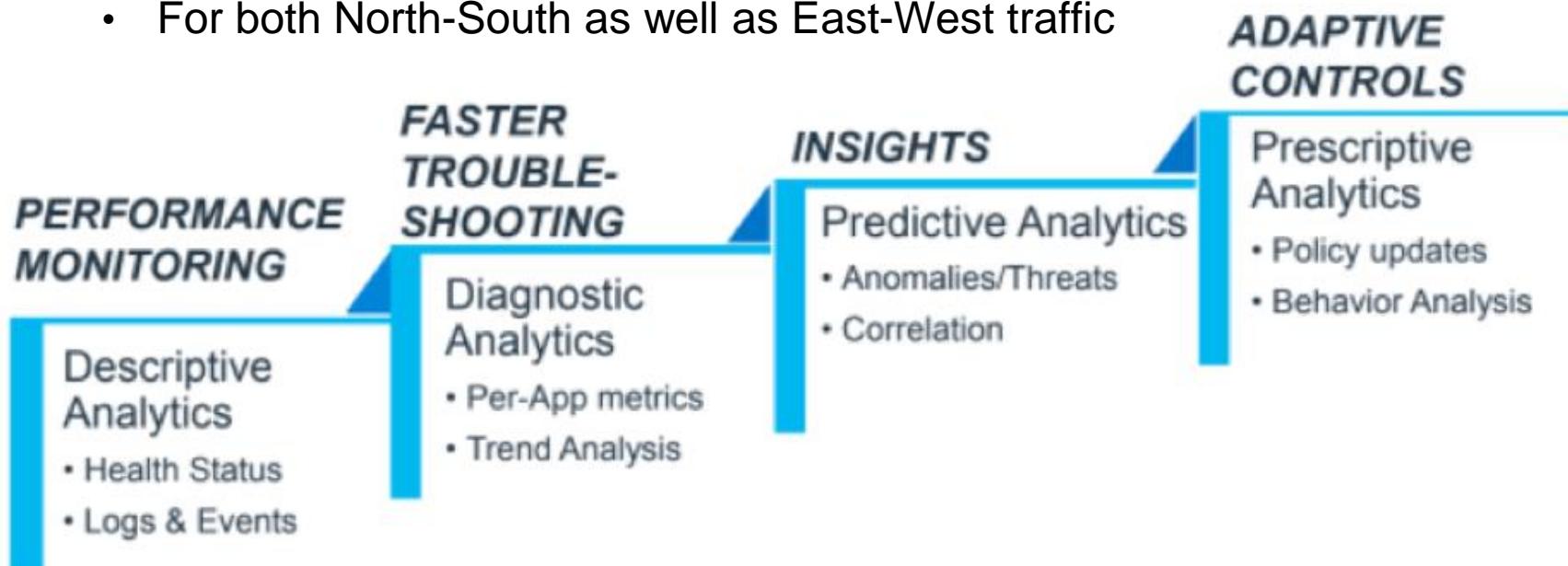


Hub-Spoke Proxy Deployment

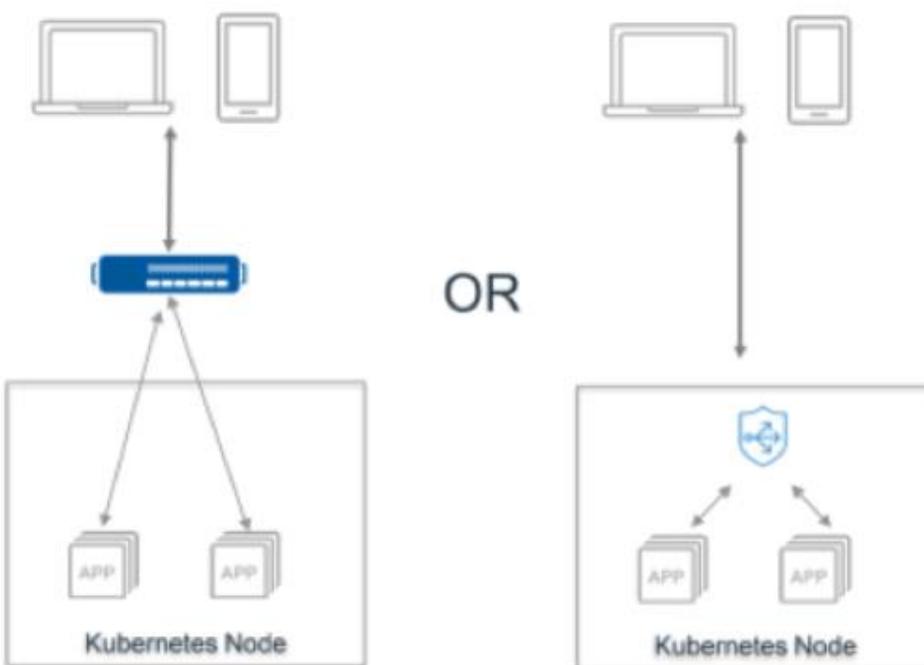
# Traffic Flow & Security Needs

Knowing details of traffic at application layer is important

- For both application optimization and security
- For both North-South as well as East-West traffic



# Application Delivery Controller (ADC)



- Kubernetes provides Kubeproxy for simple usage
  - Works by adjusting iptables rules (Layer 3)
- Implementation of Advance Traffic Ingress is left to ecosystem vendors

# Additional DevOps Considerations

These considerations simplify DevOps for flow and security:

- A simple architecture with a unified solution.
- Central management and control for easy analytics and troubleshooting.
- Common configuration formats like YAML and JSON, which Kubernetes also uses.
- No change in application code to implement security.
- Automated application of security policies.

# Experiment – Kind Load Balancing



# Services Networking



# Service Networking Concerns

Kubernetes service networking addresses four concerns:

- Containers within a Pod use networking to communicate via loopback.
- Cluster networking provides communication between different Pods.
- The Service resource lets you expose an application running in Pods to be reachable from outside your cluster.
- You can also use Services to publish services only for consumption inside your cluster.

# Requirements

- Service
- Topology-aware traffic routing with topology keys
- DNS for Services and Pods
- Connecting Applications with Services
- Ingress
- Ingress Controllers
- EndpointSlices
- Service Internal Traffic Policy
- Topology Aware Hints
- Network Policies
- IPv4/IPv6 dual-stack



# Service

An abstract way to expose an application running on a set of Pods as a network service. With Kubernetes you don't need to modify your application to use an unfamiliar service discovery mechanism.

Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

# Services: Supported Protocols

- TCP
- UDP
- SCTP
- HTTP
- PROXY

# Network Services without Selectors

- Services most commonly abstract access to Kubernetes Pods, but they can also abstract other kinds of network backends. For example:
- You want to have an external database cluster in production, but in your test environment you use your own databases.
- You want to point your Service to a Service in a different Namespace or on another cluster.
- You are migrating a workload to Kubernetes. While evaluating the approach, you run only a portion of your backends in Kubernetes.

# DNS for Services

- Service
- Topology-aware traffic routing with topology keys
- DNS for Services and Pods
- Connecting Applications with Services
- Ingress



Kubernetes DNS for Services and Pods

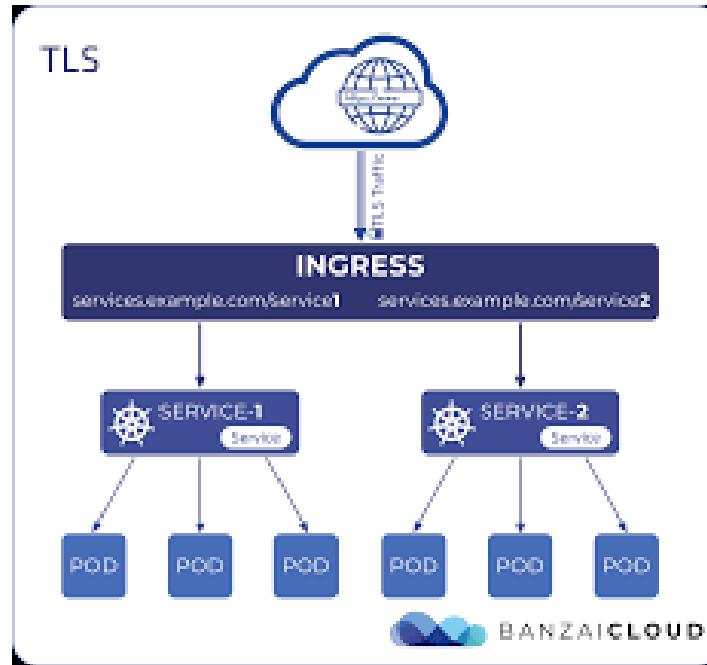
# Connecting Applications with Services

- Service
- Topology-aware traffic routing with topology keys
- DNS for Services and Pods
- Connecting Applications with Services
- Ingress



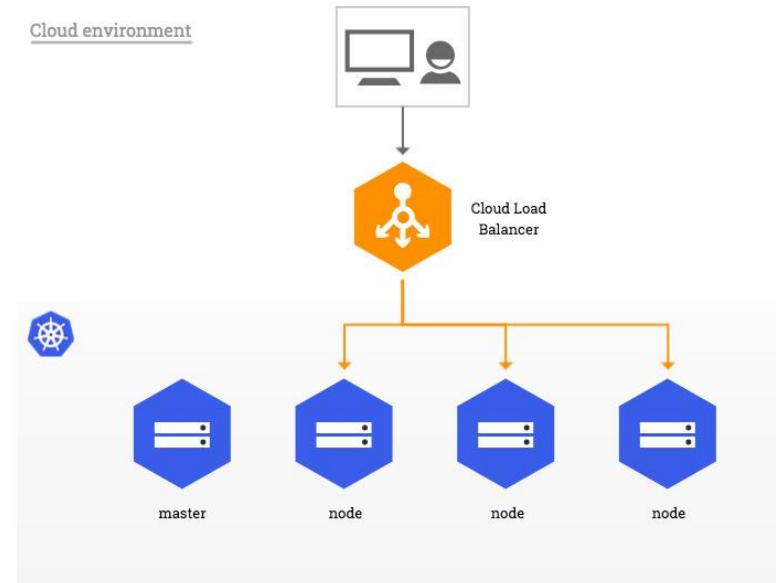
# Ingress

- Service
- Topology-aware traffic routing with topology keys
- DNS for Services and Pods
- Connecting Applications with Services
- Ingress



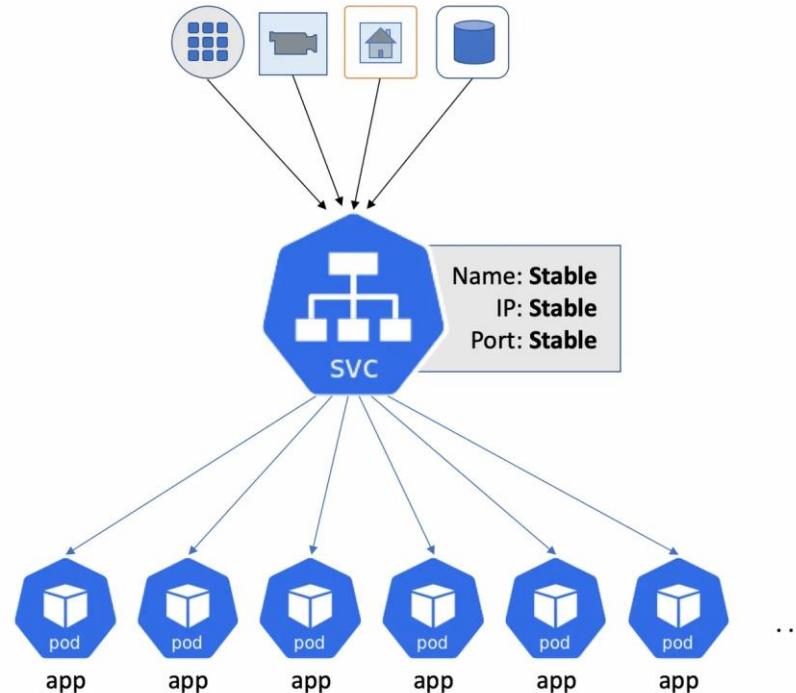
# Ingress Controllers

- Ingress Controllers
- EndpointSlices
- Service Internal Traffic Policy
- Topology Aware Hints
- Network Policies
- IPv4/IPv6 dual-stack



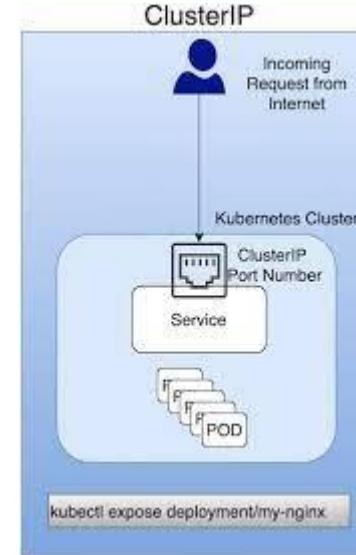
# Endpoint Slices

- Ingress Controllers
- Endpoint Slices
- Service Internal Traffic Policy
- Topology Aware Hints
- Network Policies
- IPv4/IPv6 dual-stack



# Service Internal Traffic Policy

- Ingress Controllers
- Endpoint Slices
- Service Internal Traffic Policy
- Topology Aware Hints
- Network Policies
- IPv4/IPv6 dual-stack



# Topology Aware Hints

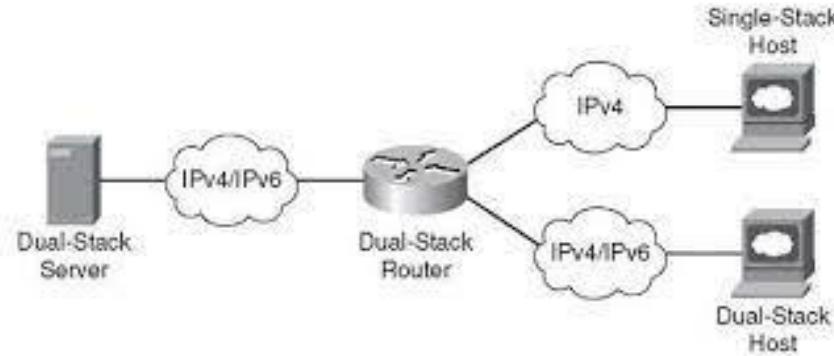
- Ingress Controllers
- Endpoint Slices
- Service Internal Traffic Policy
- Topology Aware Hints
- Network Policies
- IPv4/IPv6 dual-stack



**kubernetes**

# Dual Stack

- Ingress Controllers
- Endpoint Slices
- Service Internal Traffic Policy
- Topology Aware Hints
- Network Policies
- IPv4/IPv6 dual-stack



# POP QUIZ:

## Kubernetes



Which of the following commands is used to create Kubernetes service?

- A: Kubectl run
- B: Kubectl deploy
- C: Kubectl expose
- D: Java spring framework



# POP QUIZ:

## Kubernetes



Which of the following commands is used to create Kubernetes service?

- A: Kubectl run
- B: Kubectl deploy
- C:: **Kubectl expose**
- D: Java spring framework



# Experiment – K3D Getting Started

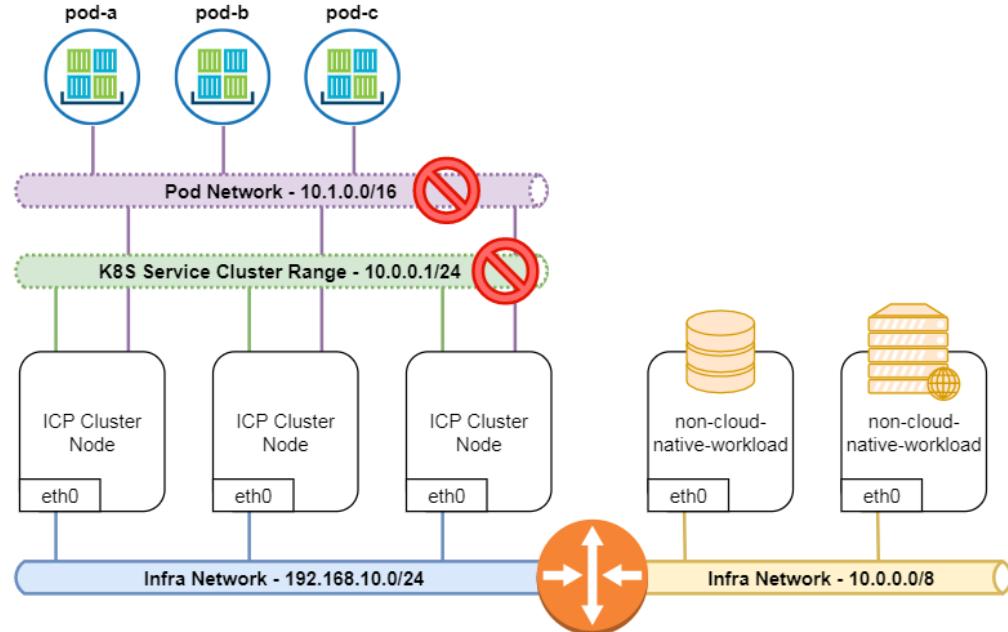


# Kubernetes Networking



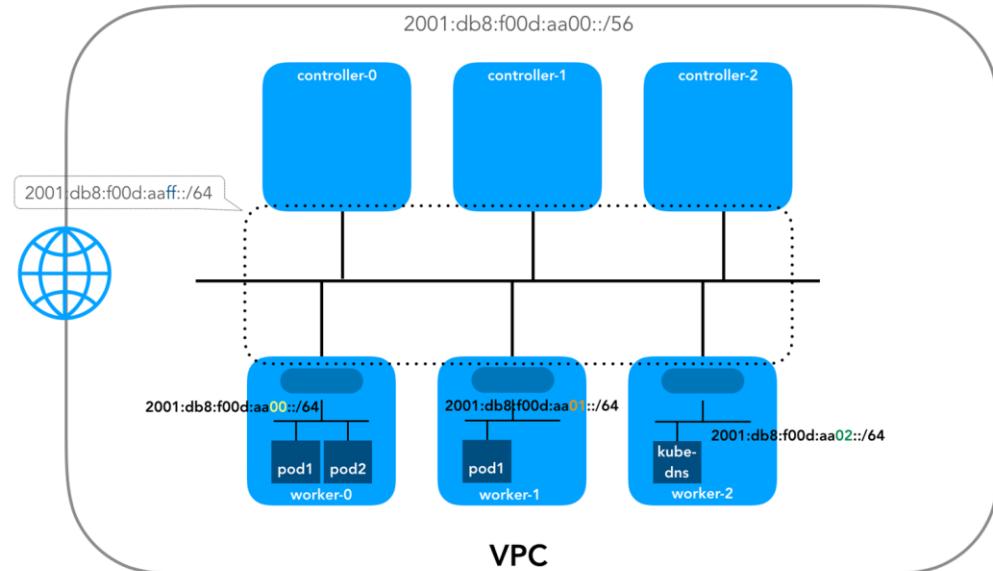
# Kubernetes Networking Model

Every Pod gets its own IP address. This means you do not need to explicitly create links between Pods and you almost never need to deal with mapping container ports to host ports. This creates a clean, backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.



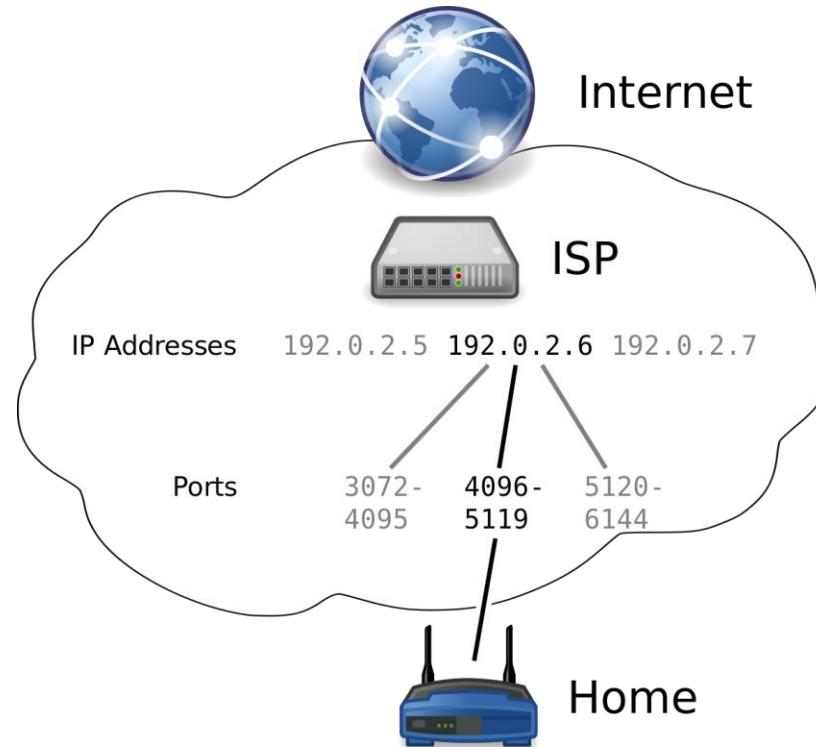
# Cluster Networking

Network entities are identified by their IP address. Servers can listen to incoming connections on multiple ports. Clients can connect to (TCP)s or send/receive data from (UDP), servers within their network.



# IP Addresses and Ports

Network entities are identified by their IP address. Servers can listen to incoming connections on multiple ports. Clients can connect to (TCP)s or send/receive data from (UDP), servers within their network.



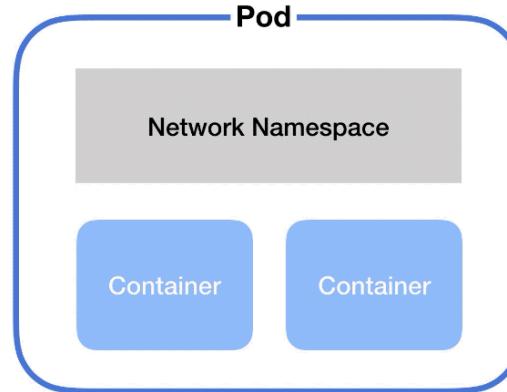
# IP Addresses and Endpoints

Network entities are identified by their IP address. Servers can listen to incoming connections on multiple ports. Clients can connect to (TCP)s or send/receive data from (UDP), servers within their network.

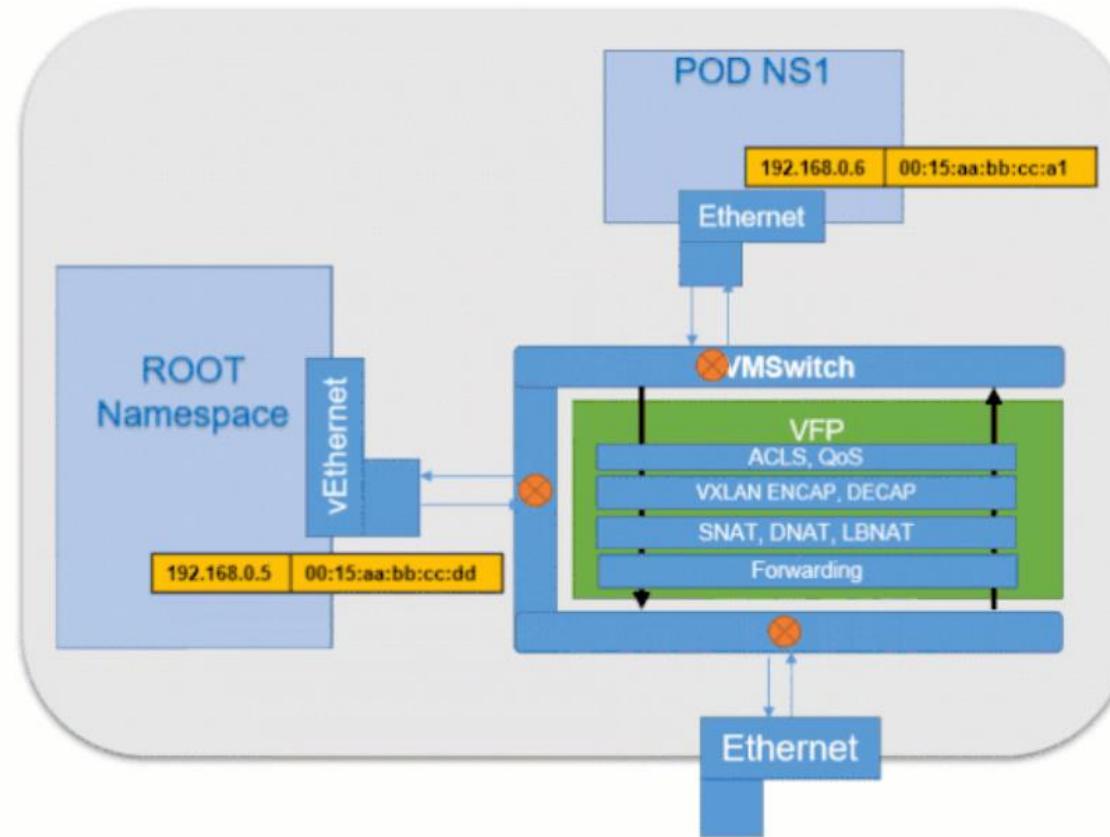


# Networking Namespaces

A namespace is a kind of virtual cluster. You can have a single physical cluster that contains multiple virtual clusters segregated by namespaces. By default, pods in one namespace can access pods and services in other namespaces



# Windows Container Networking



# POP QUIZ:

## Kubernetes



Which of the following is true of namespaces?

- A: Namespaces group varying data types
- B: Namespaces must all have the same name
- C: Namespaces organize elements that have unique names
- D: Namespaces are allowed only in object-oriented design

# POP QUIZ:

## Kubernetes



Which of the following is true of namespaces?

- A: Namespaces group varying data types
- B: Namespaces must all have the same name
- C: Namespaces organize elements that have unique names
- D: Namespaces are allowed only in object-oriented design

# POP QUIZ:

## Kubernetes



If a Deployment is exposed publicly, what happens with the network traffic during an update?

- A: Is dropped
- B: Is load-balanced only to the old instances
- C:: Is load-balanced only to available instances (ld and new)

# POP QUIZ:

## Kubernetes



What is the basic operational unit of Kubernetes?

- A: Task
- B: Pod
- C:: INames
- D: Container



# Experiment – K3D Simple NodePort

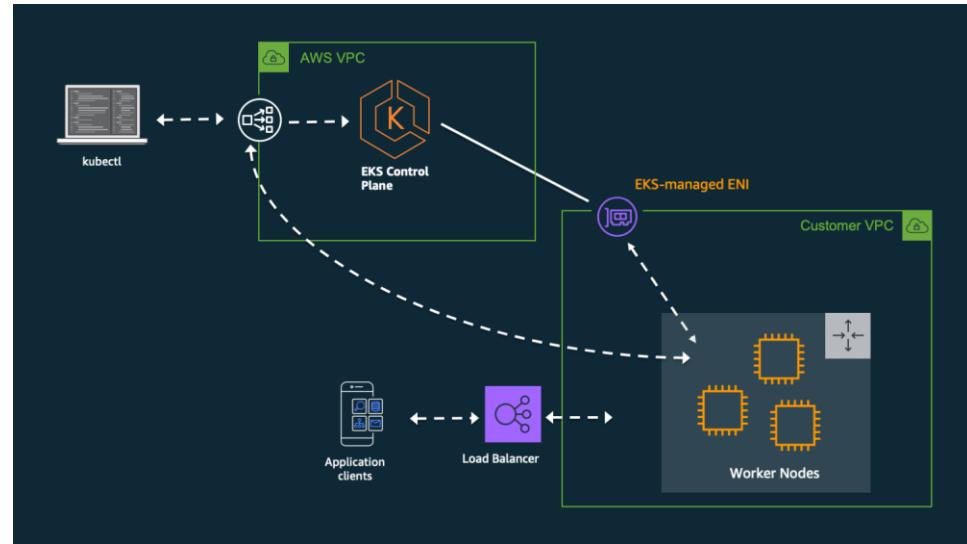


# Cloud Kubernetes Networking



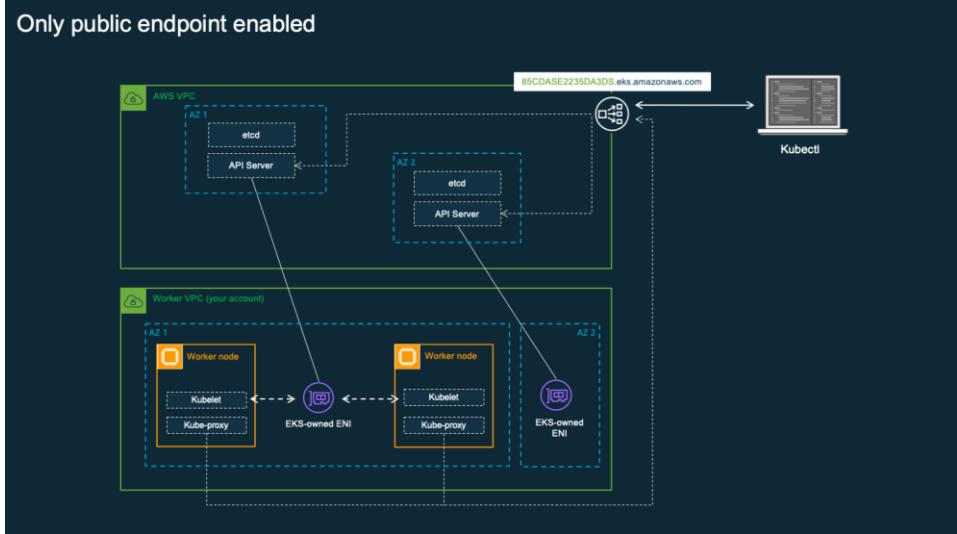
# De-mystifying Cluster Networking

An EKS cluster consists of two VPCs: one VPC managed by AWS that hosts the Kubernetes control plane and a second VPC managed by customers that hosts the Kubernetes worker nodes (EC2 instances) where containers run, as well as other AWS infrastructure (like load balancers) used by the cluster. All worker nodes need the ability to connect to the managed API server endpoint. This connection allows the worker node to register itself with the Kubernetes control plane and to receive requests to run application pods.



# Networking Modes

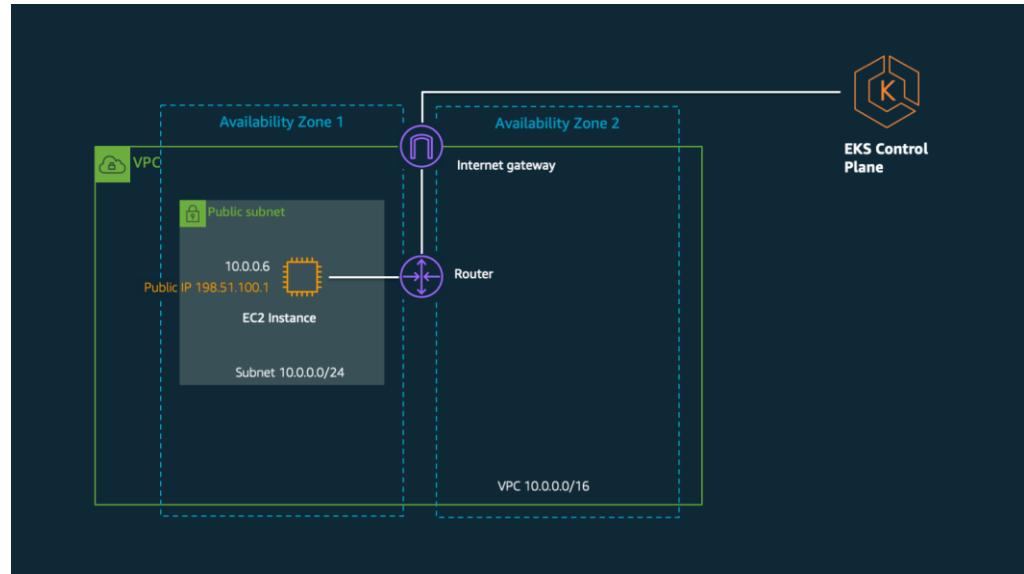
EKS has two ways of [controlling access to the cluster endpoint](#). Endpoint access control lets you configure whether the endpoint is reachable from the public internet or through your VPC. You can enable the public endpoint (default), private endpoint, or both endpoints at the same time. When the public endpoint is enabled, you can also add CIDR restrictions, which allow you to limit the client IP addresses that can connect to the public endpoint.



# VPC Configurations

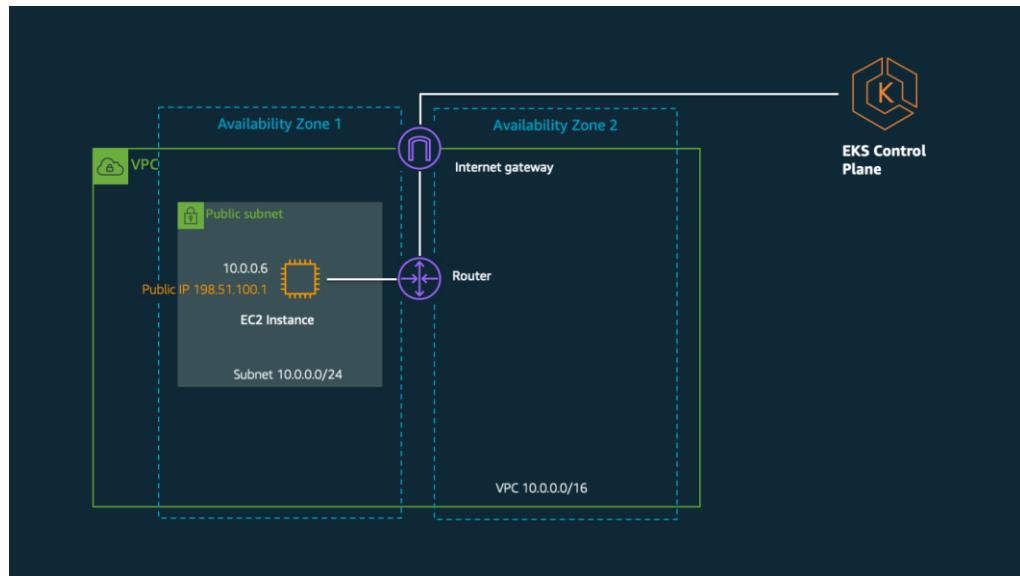
There are three typical ways to configure the VPC for your Amazon EKS cluster:

1. Using only public subnets.
2. Using public and private subnets.
3. Using only private subnets.



# Pod Networking CNI

Amazon EKS supports native VPC networking with the Amazon VPC Container Network Interface (CNI) plugin for Kubernetes. This plugin assigns an IP address from your VPC to each pod. The Amazon VPC CNI plugin is fully supported for use on Amazon EKS and self-managed Kubernetes clusters on AWS.



# Experiment – K3D Simple Ingress

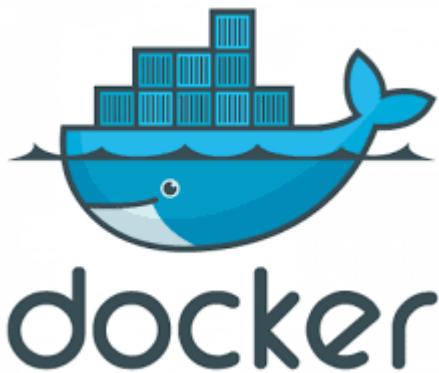


# Kubernetes vs. Docker Networking



# What is Docker

Docker is an open source standalone application which works as an engine used to run containerized applications. It is installed on your operating system (OS), preferably on [Linux](#), but can be also installed on Windows and [macOS](#), which in turn runs on a physical or virtual machine.



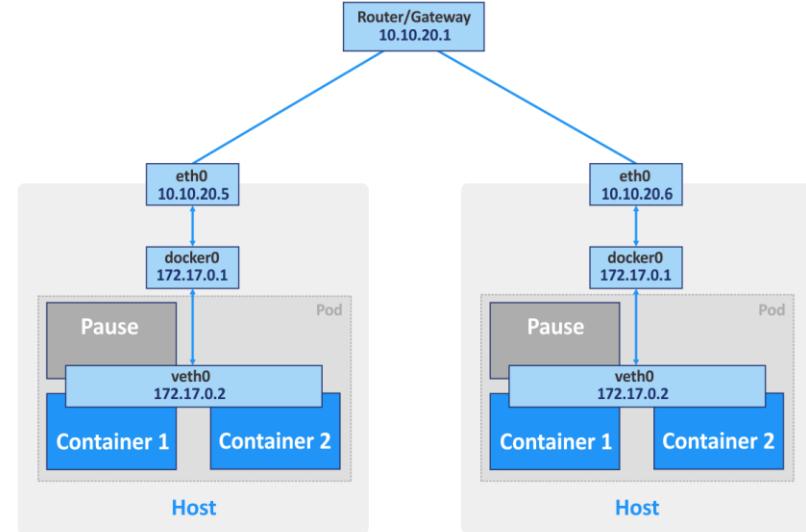
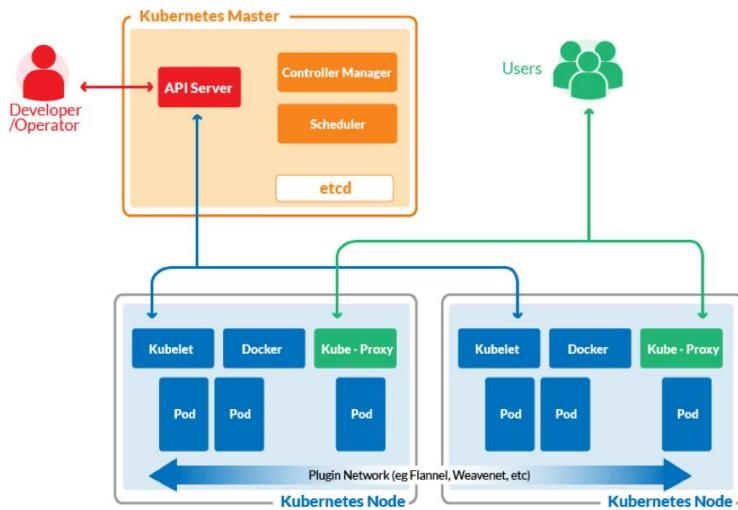
## Docker Swarm

- 1 No Auto Scaling
- 2 Good community
- 3 Easy to start a cluster
- 4 Limited to the Docker API's capabilities
- 5 Does not have as much experience with production deployments at scale

## Kubernetes

- 1 Auto Scaling
- 2 Great active community
- 3 Difficult to start a cluster
- 4 Can overcome constraints of Docker and Docker API
- 5 Deployed at scale more often among organizations

# Networking



# Networking

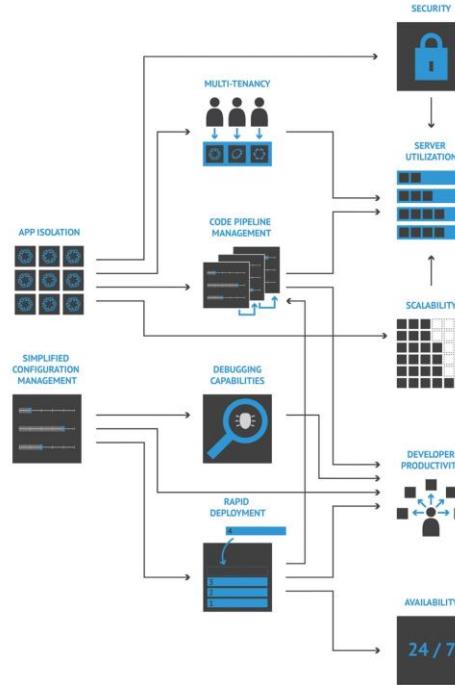
**Docker Swarm.** There is one default internal network for communication of containers within a cluster, onto which more networks can be added if needed. Manual certificate generation is supported for the encryption of container data traffic.

**Kubernetes.** The network model of Kubernetes is quite different and is implemented by using plugins, one of which is Flannel. Pods interact with each other, and this interaction can be restricted by policies. There is an internal network managed by the etcd service.

# Use Cases

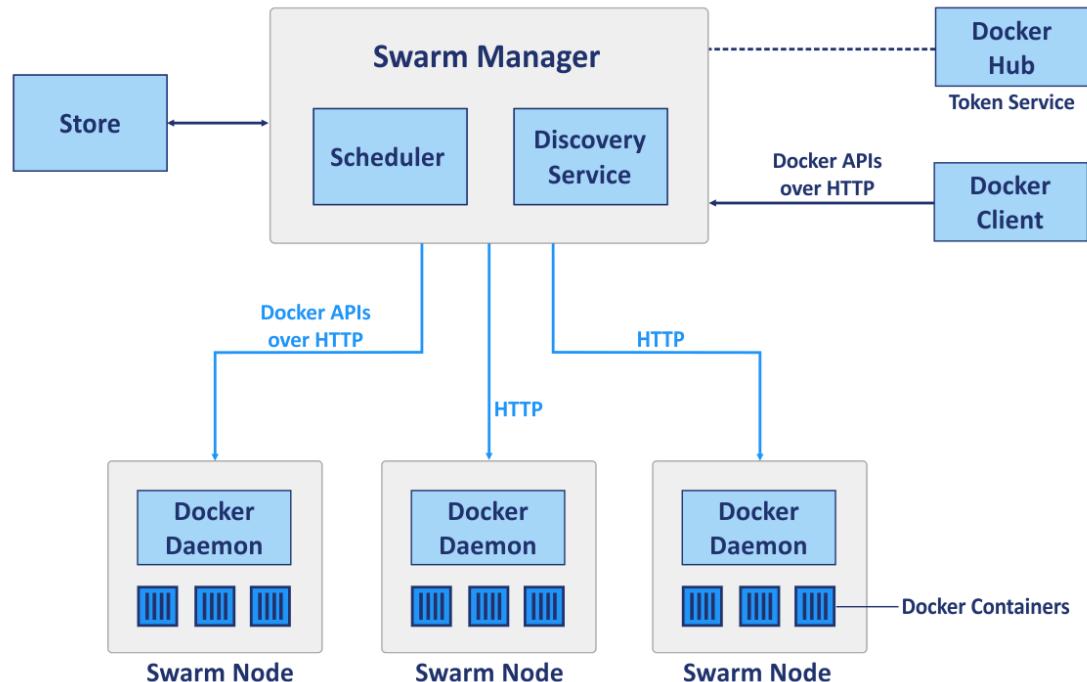
Using Docker as a standalone software is good for development of applications, as developers can run their applications in isolated environments. What's more, testers can also use Docker to run applications in sandbox environments. If you wish to use Docker to run a high number of containers in the production environment you may encounter some complications along the way. For example, some containers can easily be overloaded or fail. You can manually restart the container on the appropriate machine, but manual management can take a lot of your valuable time and energy.

## DOCKER USE CASES



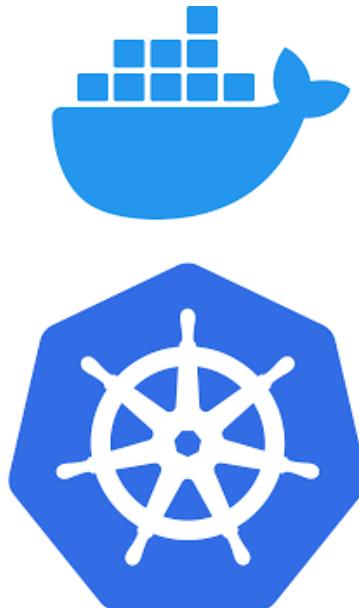
# Docker Swarm

*Docker Swarm* is a native clustering tool for Docker that can turn a pool of Docker hosts into a single virtual host. Docker Swarm is fully integrated with the Docker Engine and allows you to use standard APIs and networking processes; it is intended to deploy, manage and scale Docker containers.



# Cluster Deployment

**Docker Swarm.** A Standard Docker API allows you to deploy a cluster with Swarm by using a standard Docker CLI (command line interface) that makes deployment easier, especially when used for the first time



**Docker Swarm.** A Standard Docker API allows you to deploy a cluster with Swarm by using a standard Docker CLI (command line interface) that makes deployment easier, especially when used for the first time

# Scalability

**Docker Swarm.** A command line interface (CLI) is quite simple and easy to understand. In terms of simplicity, Swarm can be considered a more scalable solution when compared with Kubernetes.

**Kubernetes:** When it comes to auto-scale, Kubernetes is preferable due to its ability to analyze server loads, and scale up and down automatically in accordance with the given requirements. Kubernetes is the optimal choice for large distributed networks and complex systems.

# High Availability

The two solution options each possess similar service replication and redundancy mechanisms, and in both cases the system is self-regulated and does not require manual reconfiguration after a failed node.

**Docker Swarm.** Manager nodes handle the resources of worker nodes and the entire cluster. The Swarm cluster nodes participate in replication of services.

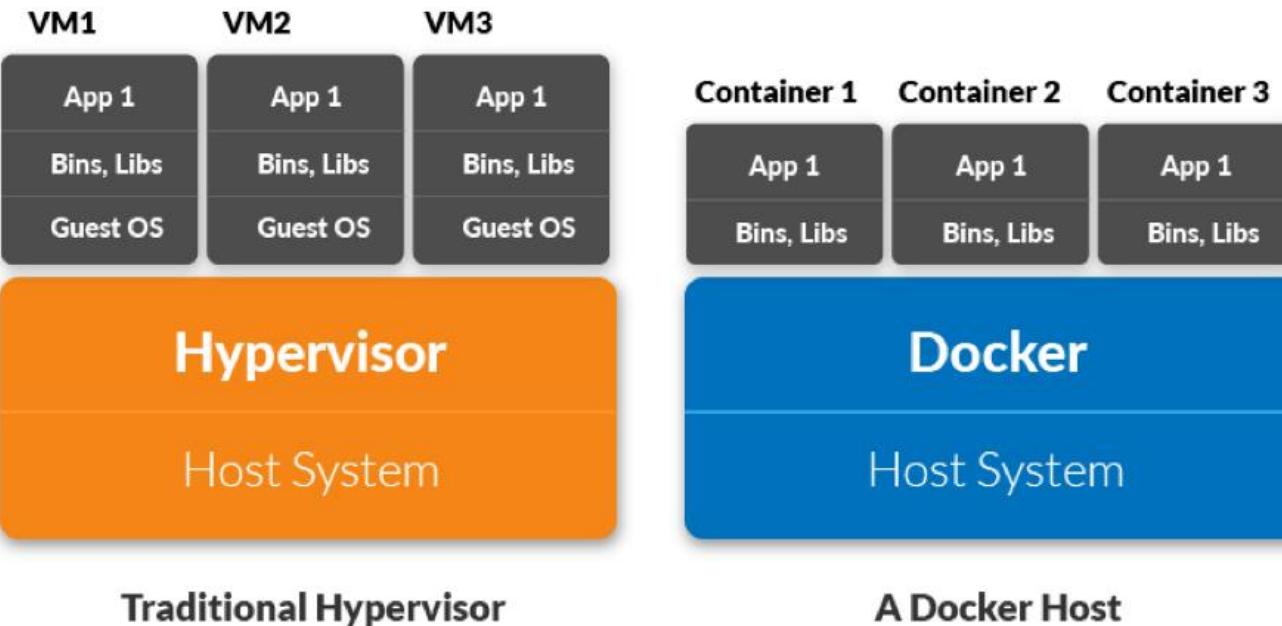
**Kubernetes.** Unhealthy nodes are detected by load balancing services of Kubernetes, and are eliminated from the cluster. All Pods are distributed among nodes, thereby providing high availability, should a node on which a containerized application is running fail.

# Load Balancing

**Docker Swarm.** Load balancing is a built-in feature and can be performed automatically by using the internal Swarm network.

**Kubernetes.** Policies defined in Pods are used for load balancing in Kubernetes. In this case, container Pods must be defined as services. You have to configure load balancing settings manually, while Ingress can be utilized for load balancing.

# Containers



# Monitoring

**Docker Swarm:** Unlike Kubernetes, Docker Swarm does not offer a monitoring solution out-of-the-box. As a result, you have to rely on third-party applications to support monitoring of Docker Swarm.

Typically, monitoring a Docker Swarm is considered to be more complex due to its sheer volume of cross-node objects and services, relative to a K8s cluster.

**Kubernetes:** offers multiple native logging and monitoring solutions for deployed services within a cluster. These solutions monitor application performance by:

- Inspecting services, pods, and containers

# POP QUIZ:

## Kubernetes: Local Development, Test, and Debug



How are Kubernetes and Docker Related?

- A: Docker can't deploy automatic rollbacks; Kubernetes can deploy rolling updates as well as automatic rollbacks
- B: Docker allows for auto-scaling when using Kubernetes
- C: Kubernetes allows for the manual linking and orchestration of several containers, running on multiple hosts that have been created using Docker.

# POP QUIZ:



## Kubernetes: Local Development, Test, and Debug



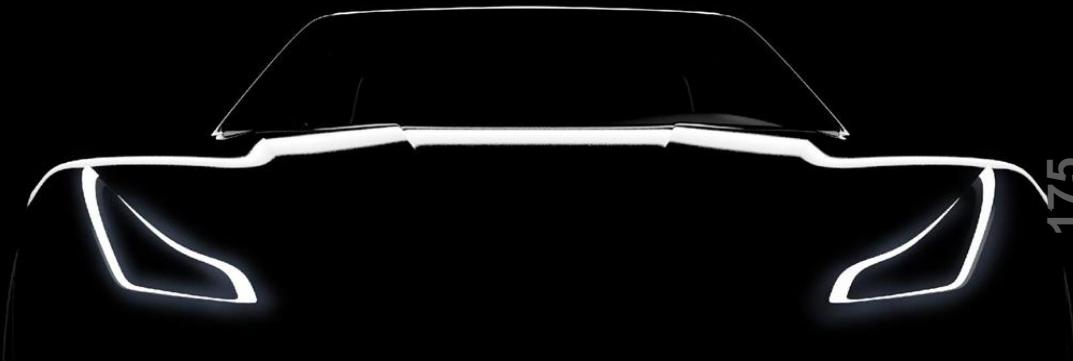
How are Kubernetes and Docker Related?

- A: Docker can't deploy automatic rollbacks; Kubernetes can deploy rolling updates as well as automatic rollbacks
- B: Docker allows for auto-scaling when using Kubernetes
- C: Kubernetes allows for the manual linking and orchestration of several containers, running on multiple hosts that have been created using Docker.

# Experiment – K3D & Istio



# Lookup and Discovery



# Lookup and Discovery

In order for pods and containers to communicate with each other, they need to find one another. There are several ways for containers to locate other containers or announce themselves. There are also some architectural patterns that allow containers to interact indirectly. Each approach has its own pros and cons.



# Service

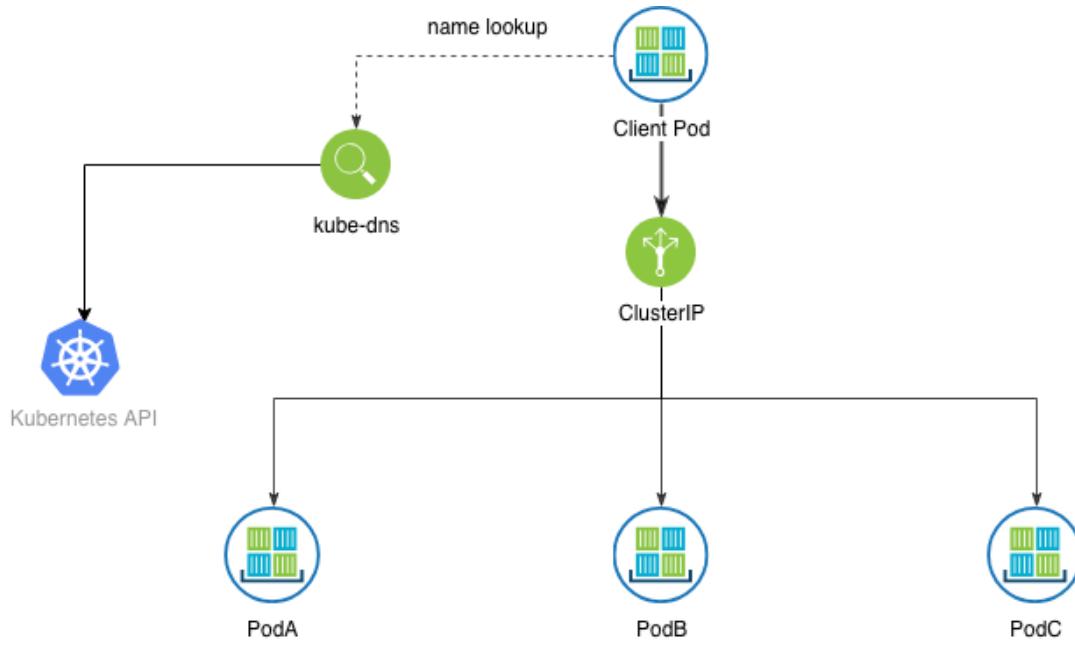
An abstract way to expose an application running on a set of Pods as a network service.

With Kubernetes you don't need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

# Service Discovery

Kubernetes expects that a service is running within the pod network mesh that performs name resolution and acts as the primary name server within the cluster.

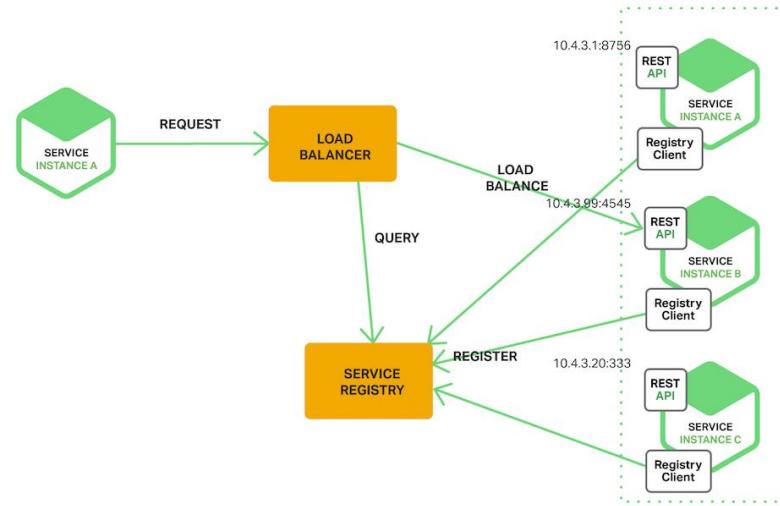
```
# cat /etc/resolv.conf
nameserver <kube-dns ClusterIP>
search
<namespace>.svc.<cluster_domain>
> svc.<cluster_domain>
<cluster_domain> <additional ...>
options ndots:5
```



# Cloud-Native Service Discovery

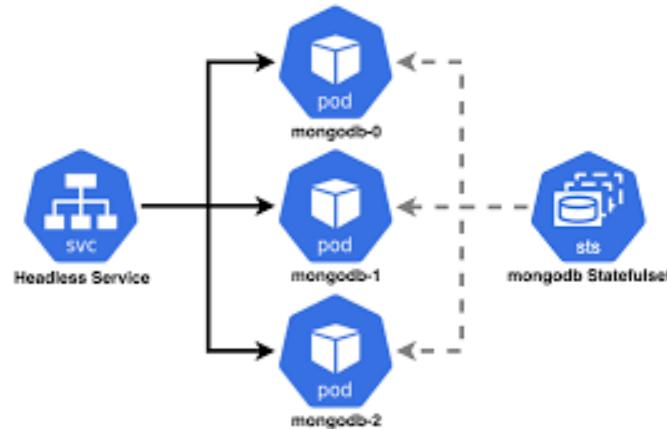
Given the demands of microservices architecture, you can probably guess the benefits cloud native coordination & service discovery platforms bring. These services help you shift away from manual processes and get the most out of cloud native. Their benefits include:

- Performance
- Simple coordination & health checks
- Scalability



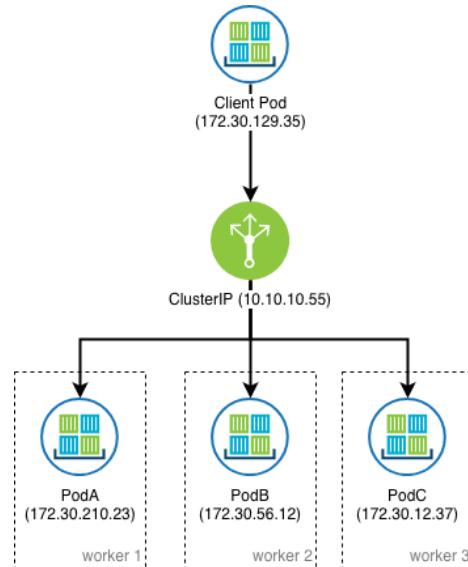
# Headless Services

A headless service is a service with a service IP but instead of load-balancing it will return the IPs of our associated Pods. This allows us to interact directly with the Pods instead of a proxy. It's as simple as specifying `None` for `.spec.clusterIP` and can be utilized with or without selectors - you'll see an example with selectors in a moment.



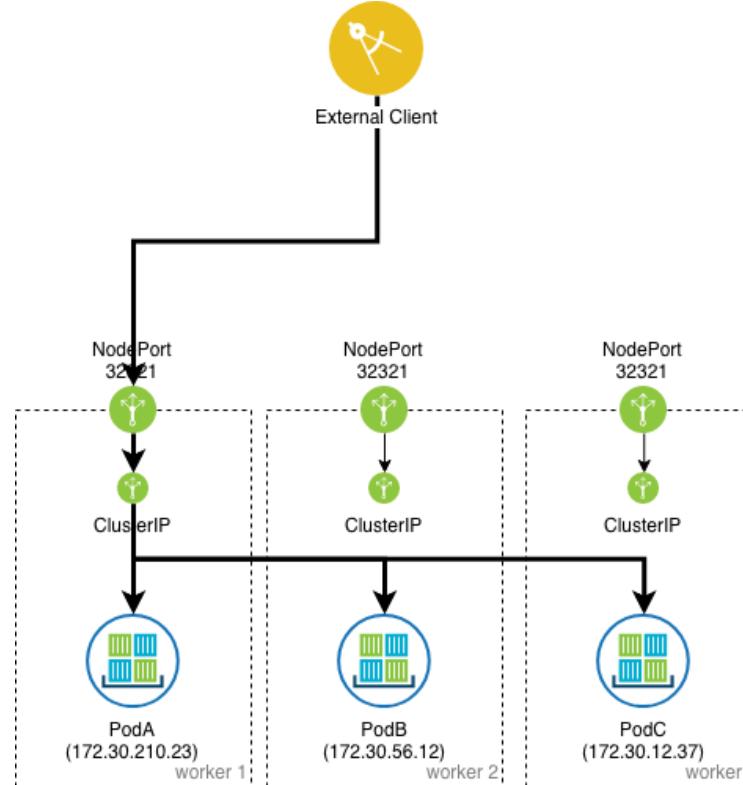
# Cluster IP

A **cluster IP** is the virtual IP that represents your clustered service. Typically this is the [IP address](#) assigned to your clustered service on your [load balancer](#). An internal fixed IP known as a `ClusterIP` can be created in front of a pod or a replica as necessary



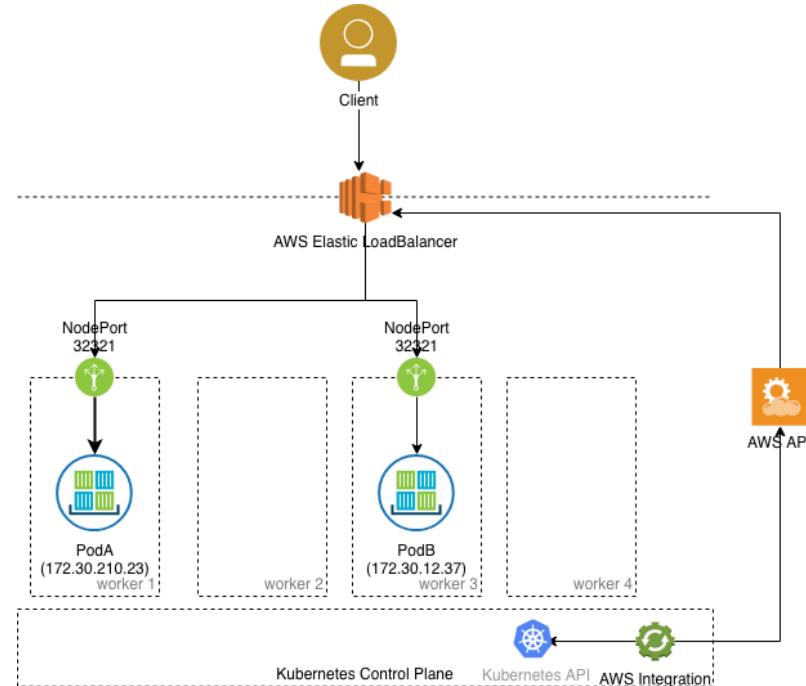
# NodePort

Services of type NodePort build on top of ClusterIP type services by exposing the ClusterIP service outside of the cluster on high ports (default 30000-32767). If no port number is specified then Kubernetes automatically selects a free port. The local kube-proxy is responsible for listening to the port on the node and forwarding client traffic on the NodePort to the ClusterIP.



# LoadBalancer

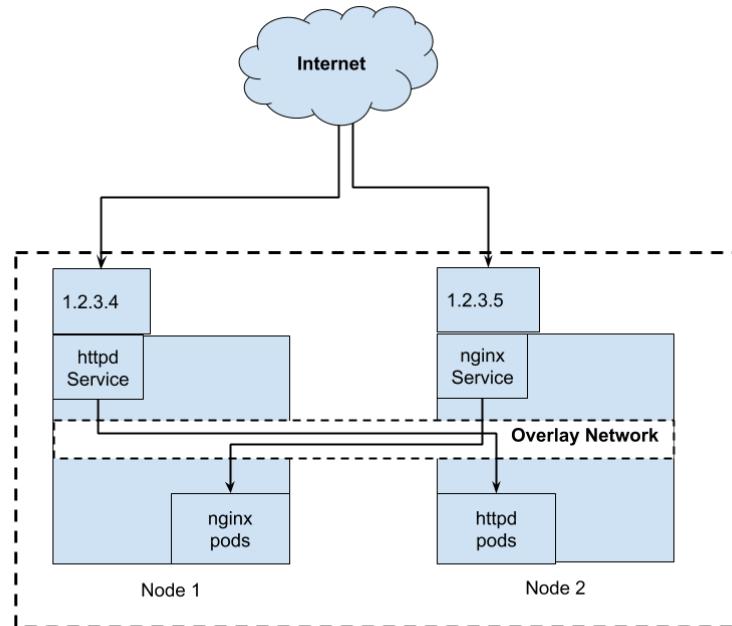
The LoadBalancer service type is built on top of NodePort service types by provisioning and configuring external load balancers from public and private cloud providers. It exposes services that are running in the cluster by forwarding layer 4 traffic to worker nodes.



# External Services

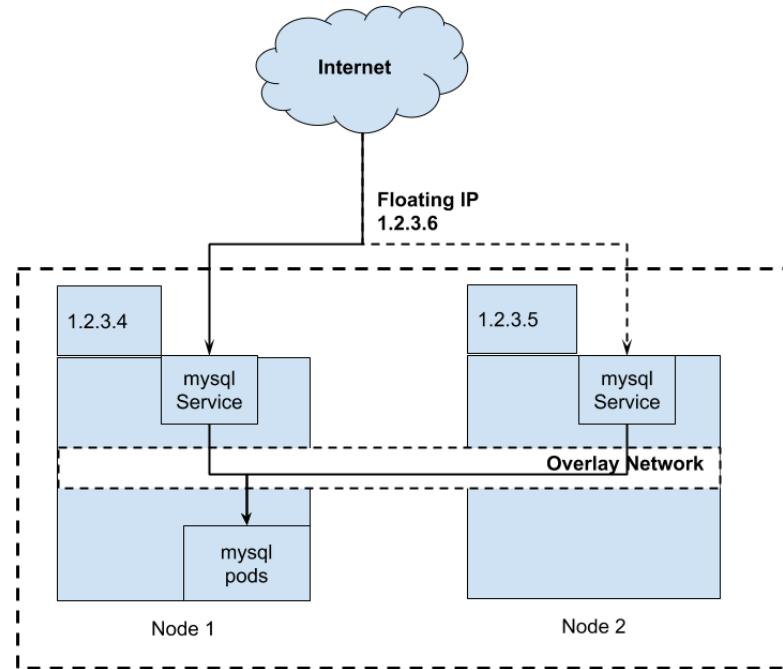
If there are external IPs that route to one or more cluster nodes, Kubernetes Services can be exposed on those

externalIPs. Traffic that ingresses into the cluster with the external IP (as destination IP), on the Service port, will be routed to one of the Service endpoints. externalIPs are not managed by Kubernetes and are the responsibility of the cluster administrator.



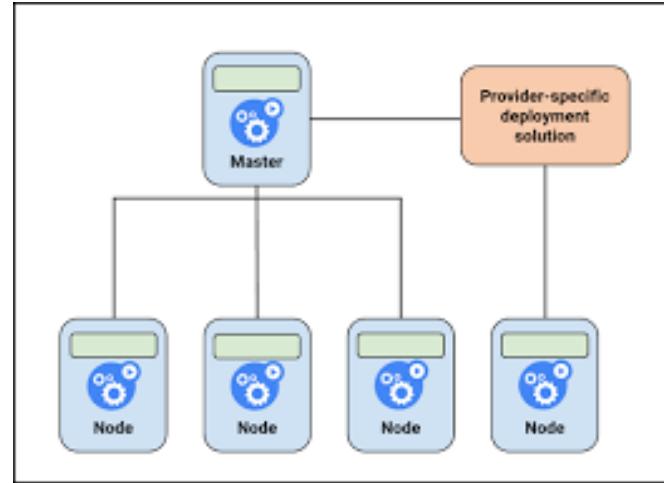
# Floating IP Service

Most cloud providers nowadays offer Floating IP service. Floating IP allows you to have 1 IP and assign that IP dynamically to any IP that you want. In this case, the IP can be assigned to any worker node in the Kubernetes cluster.



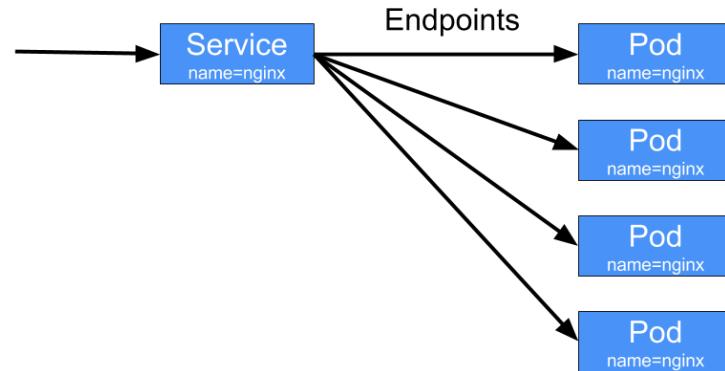
# Self-registration

When a container runs, it knows its pod's IP address. Each container that wants to be accessible to other containers in the cluster can connect to a registration service and register its IP address and port. Other containers can query the registration service for the IP addresses and ports of all registered containers and connect to them. When a container is destroyed (gracefully), it will unregister itself. If a container dies ungracefully, then a mechanism needs to be established to detect that.



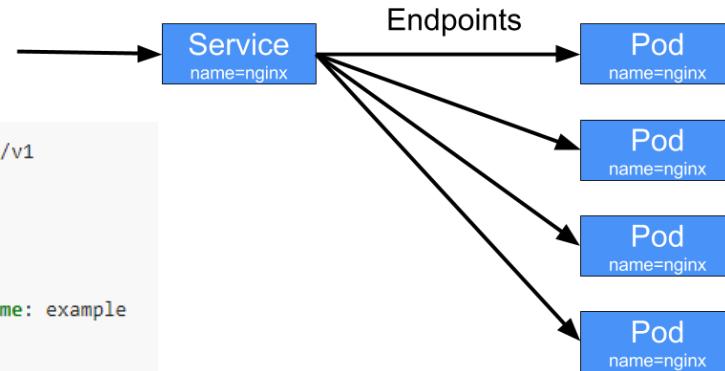
# Services and endpoints

Kubernetes services can be regarded as a registration service. Pods that belong to a service are registered automatically based on their labels. Other pods can look up the endpoints to find all the service pods or take advantage of the service itself and directly send a message to the service that will get routed to one of the backend pods, although, most of the time, pods will just send their message to the service itself that will forward it to one of the backing pods.



# Endpoint Slices

EndpointSlices are an API resource that can provide a more scalable alternative to Endpoints.



```
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: example-abc
  labels:
    kubernetes.io/service-name: example
addressType: IPv4
ports:
  - name: http
    protocol: TCP
    port: 80
endpoints:
  - addresses:
      - "10.1.2.3"
    conditions:
      ready: true
    hostname: pod-1
    nodeName: node-1
    zone: us-west2-a
```

# Loose Coupling

```
class CustomerRepository
{
    private readonly IDatabase database;

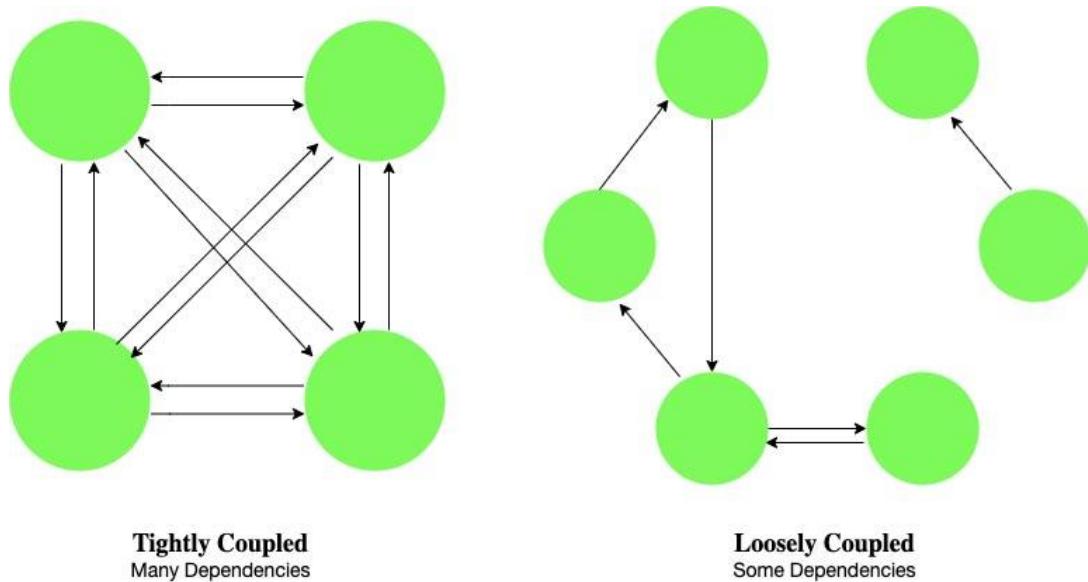
    public CustomerRepository(IDatabase database)
    {
        this.database = database;
    }

    public void Add(string CustomerName)
    {
        database.AddRow("Customer", CustomerName);
    }
}

interface IDatabase
{
    void AddRow(string Table, string Value);
}

class Database implements IDatabase
{
    public void AddRow(string Table, string Value)
    {
    }
}
```

loose coupling in microservice architecture means microservices should know little about each other, and any change to one service should not affect the others.



# Tight Coupling

```
class CustomerRepository
{
    private readonly Database database;

    public CustomerRepository(Database database)
    {
        this.database = database;
    }

    public void Add(string CustomerName)
    {
        database.AddRow("Customer", CustomerName);
    }
}

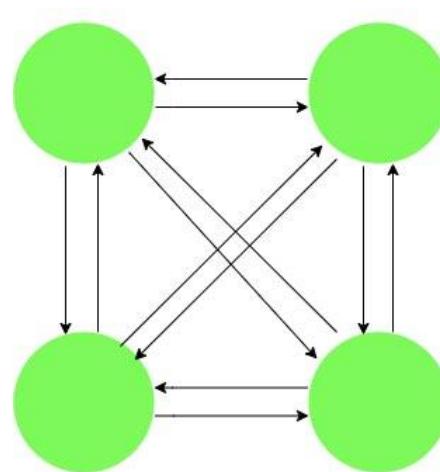
class Database
{
    public void AddRow(string Table, string Value)
    {
    }
}
```

Example of loose coupling:

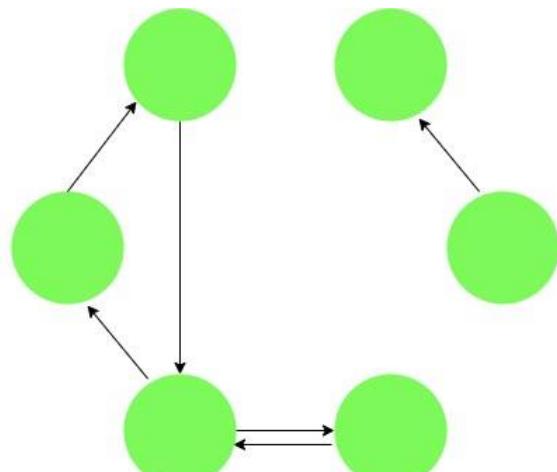
```
class CustomerRepository
{
    private readonly IDatabase database;
```

Tight coupling is when a group of classes are highly dependent on one another.

This scenario arises when a class assumes too many responsibilities, or when one concern is spread over many classes rather than having its own class.



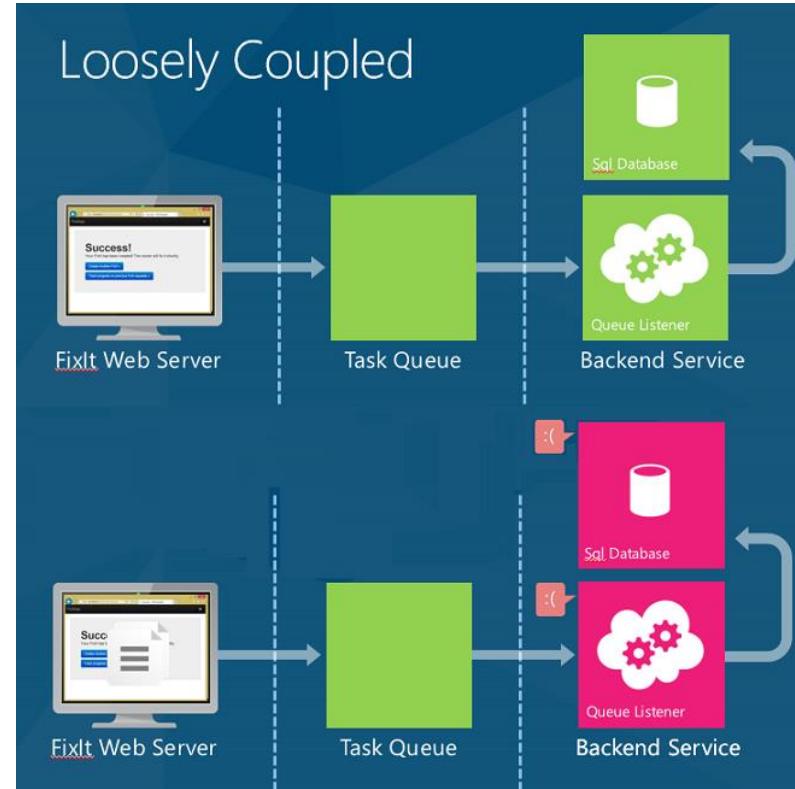
**Tightly Coupled**  
Many Dependencies



**Loosely Coupled**  
Some Dependencies

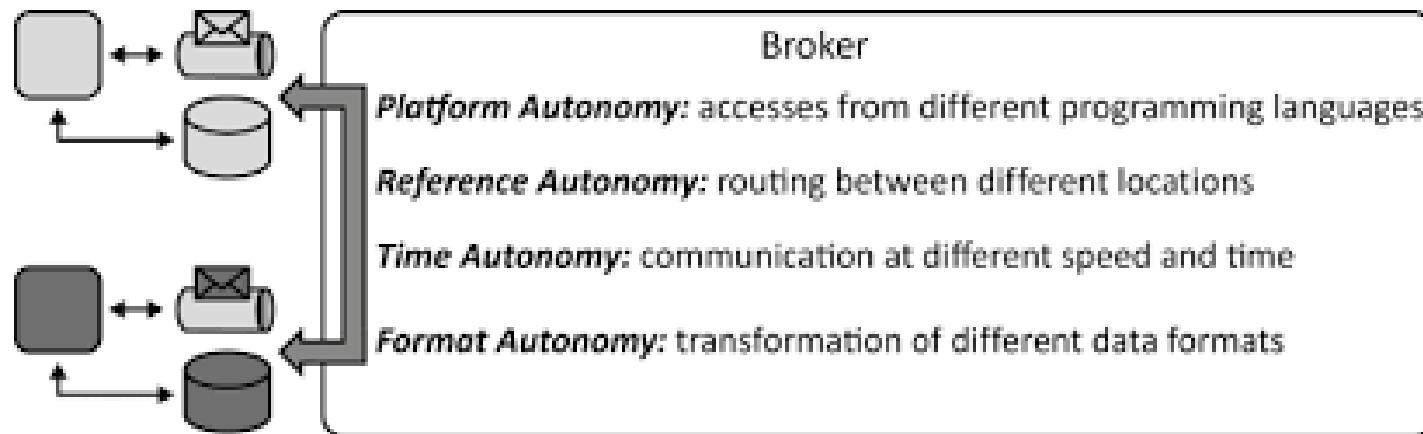
# Loosely Coupled Connectivity with Queues

Queues facilitate such loosely coupled systems. Components (containers) listen to messages from the queue, respond to messages, perform their jobs, and post messages to the queue regarding progress, completion status, and errors.



# Loosely Coupled Connectivity with Data Stores

Another loosely coupled method is to use a data store (for example, Redis) to store messages and then other containers can read them. While possible, this is not the design objective of data stores and the result is often cumbersome, fragile, and doesn't have the best performance. Data stores are optimized for data storage and not for communication.



# POP QUIZ:

## Kubernetes



What is service mesh?

- A: a dedicated infrastructure layer for handling service-to-service communication.
- B: A lightweight proxy that is distributed as sidecars.
- C: lets you associate multiple resources with a single domain name

# POP QUIZ:

## Kubernetes



What is a Service?

- A: A co-located and co-scheduled group of one or more containers that share volume and IP address
- B: A service is responsible for creating and updating instances of your containerization applications
- C: A Kubernetes Service is an abstraction layer which defines a logical set of Pods

# POP QUIZ:

## Kubernetes



What is a Service?

A: A co-located and co-scheduled group of one or more containers that share volume and IP address

B: A service is responsible for creating and updating instances of your containerization applications

C: A Kubernetes Service is an abstraction layer which defines a logical set of Pods

# POP QUIZ:

## Kubernetes



What is the scope of a rolling update?

- A: To update a Service
- B: To update a Deployment
- C: To scale an app

# POP QUIZ:

## Kubernetes



What is the scope of a rolling update?

- A: To update a Service
- B: To update a Deployment
- C: To scale an app



# Experiment – Kind & Nginx Ingress

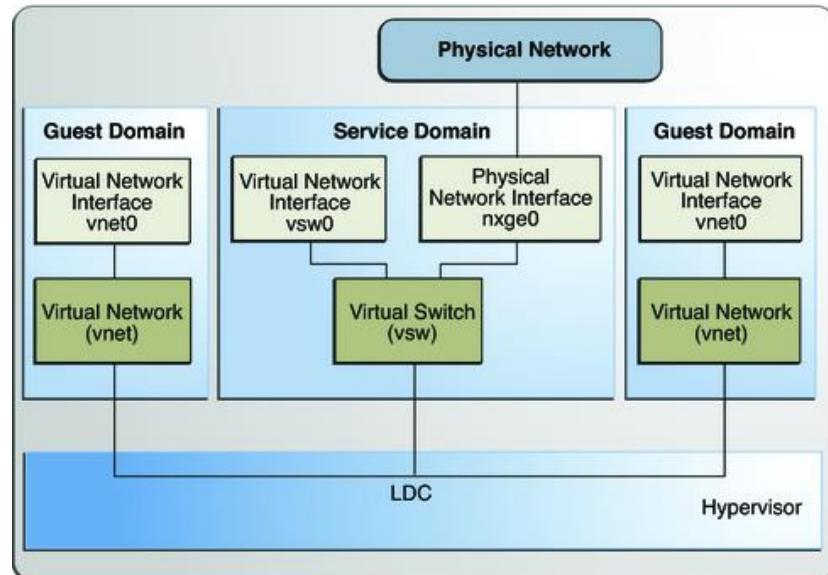


# Virtual Network Device & Optimization

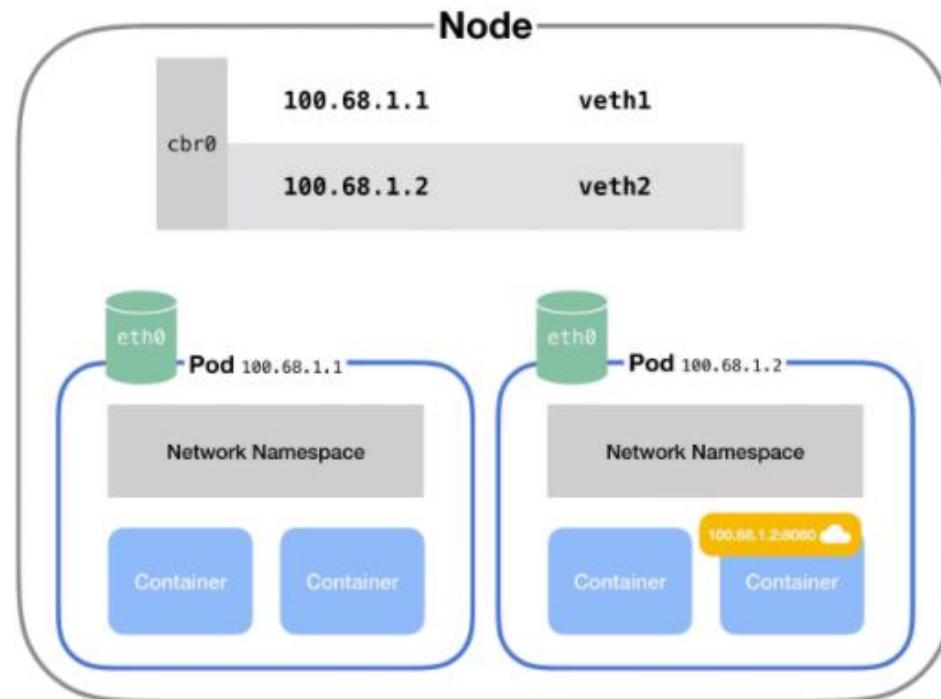


# Virtual Network Devices

A virtual network (`vnet`) device is a virtual device that is defined in a domain connected to a virtual switch. A virtual network device is managed by the virtual network driver, and it is connected to a virtual network through the hypervisor using logical domain channels (LDCs).



# Bridges



# Routing



# Maximum Transmission unit (MTU)

MTU is critical for network performance. Kubernetes network plugins such as Kubenet make their best efforts to deduce optimal MTU, but sometimes they need help.

If an existing network interface (for example, the Docker docker0 bridge) sets a small MTU, then Kubenet will reuse it.

Another example is IPSEC, which requires a lowering of the MTU due to the extra overhead from IPSEC encapsulation, but the Kubenet network plugin doesn't take it into consideration.

MTU can be further optimized by turning off unneeded encapsulations in network plugins.

# Experiment – Kind & Calico



# Pod Networking

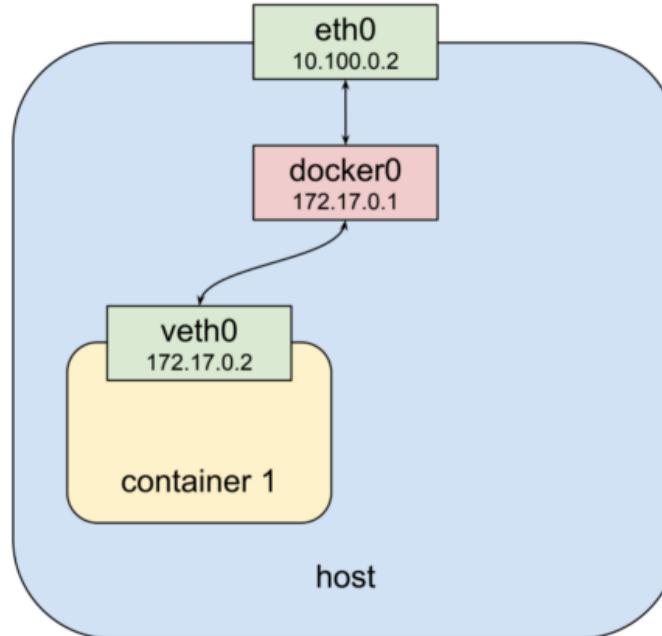


12

2017

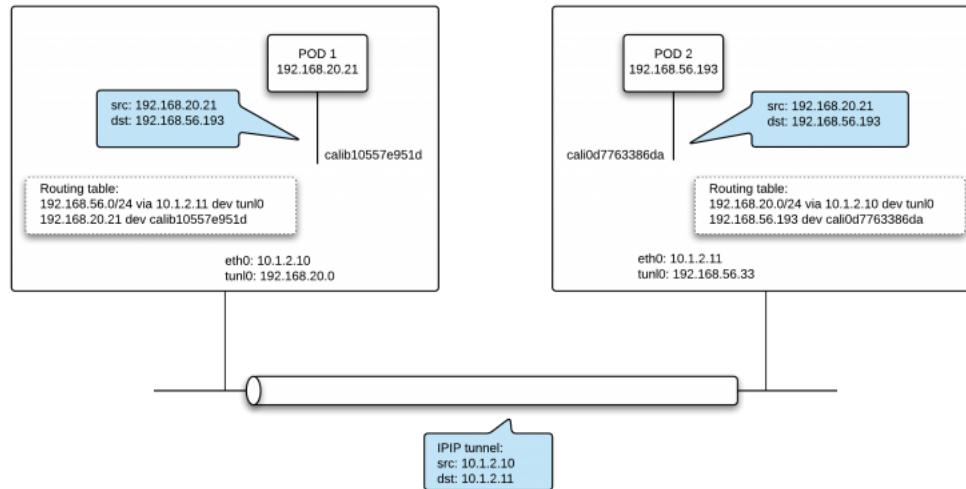
# Pod Refresher

**Pod.** The basic object for deployments in Kubernetes. Each pod has its own IP address and can contain one or more containers



# Pod Networking

In Kubernetes, every pod has its own routable IP address. Kubernetes networking takes care of routing all requests internally between hosts to the appropriate pod. External access is provided through a service, load balancer, or ingress controller, which Kubernetes routes to the appropriate pod.



# Pod Networking Simplified

A pod consists of one or more containers that are collocated on the same host, and are configured to share a network stack and other resources such as volumes.

Pods are the basic unit kubernetes applications are built from. What does “share a network stack” actually mean?

In practical terms it means that all the containers in a pod can reach each other on localhost.

If I have a container running nginx and listening on port 80 and another container running scrapyd the second container can connect to the first as `http://localhost:80`.

# Pod Lifecycle

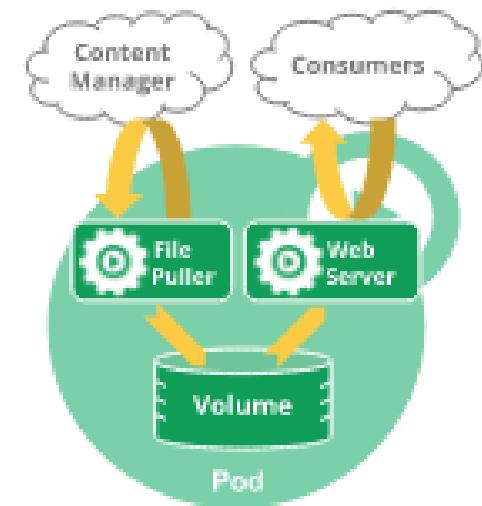
**Pending:** The pod is accepted by the Kubernetes system but its container(s) is/are not created yet.

**Running:** The pod is scheduled on a node and all its containers are created and at-least one container is in Running state.

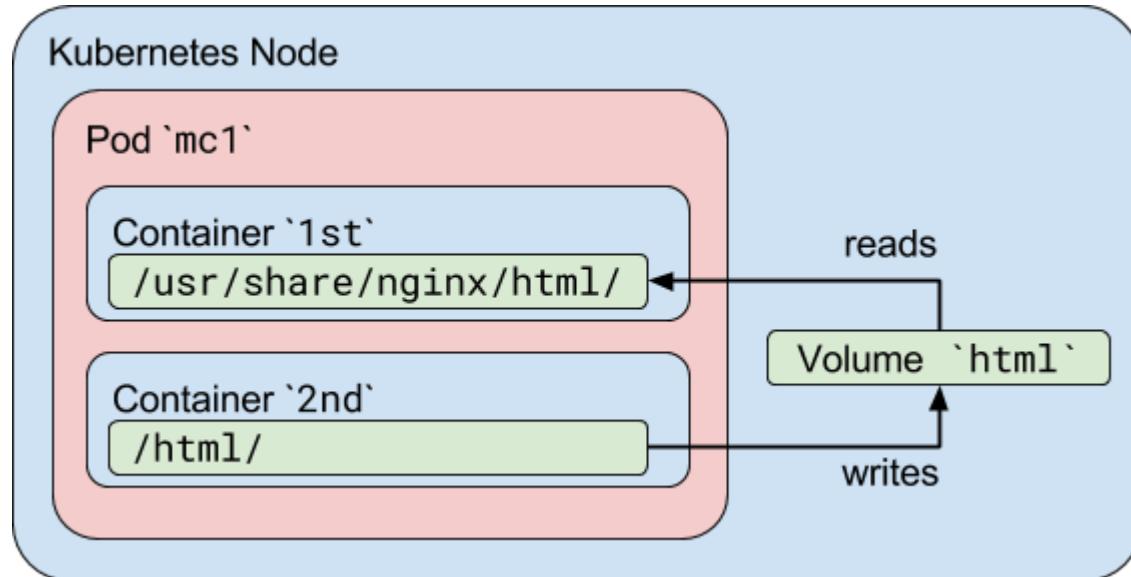
**Succeeded:** All container(s) in the Pod have exited with status 0 and will not be restarted.

**Failed:** All container(s) of the Pod have exited and at least one container has returned a non-zero status.

**CrashLoopBackoff:** The container fails to start and is tried again and again.



# Pods managing Multiple Containers

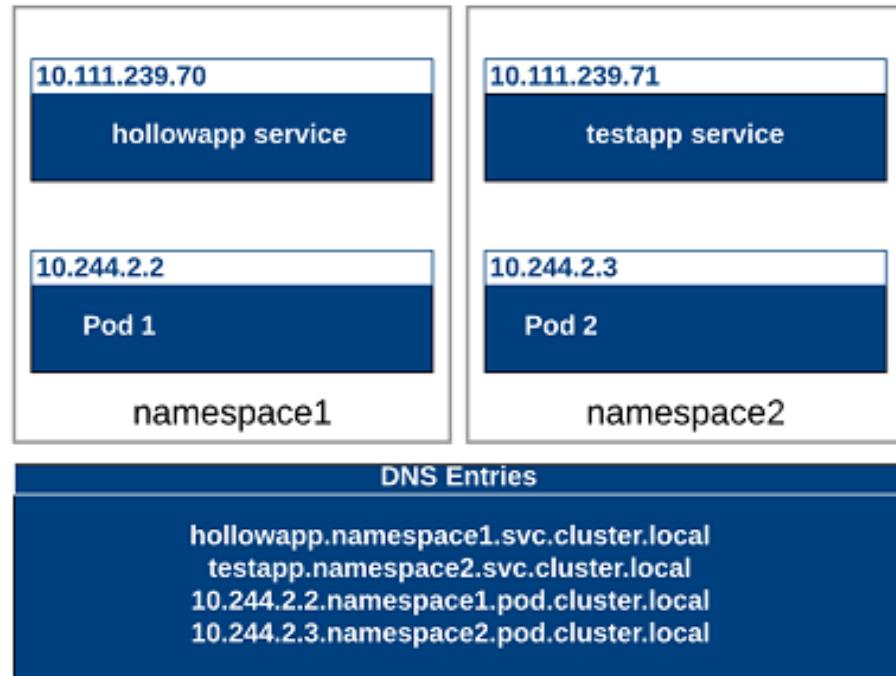


Pods are designed to support multiple cooperating processes (as containers) that form a cohesive unit of service. The containers in a Pod are automatically co-located and co-scheduled on the same physical or virtual machine in the cluster. The containers can share resources and dependencies, communicate with one another, and coordinate when and how they are terminated.

# DNS for Services and Pods

Kubernetes enables efficient service discovery with its built-in DNS add-ons: Kube-DNS or CoreDNS.

Kubernetes DNS system assigns domain and sub-domain names to pods, ports, and services, which allows them to be discoverable by other components inside your Kubernetes cluster.



# Utilizing multi-cluster Ingress DNS

Ingress in a single cluster is done at the edge of the cluster and forwards traffic into the cluster.

In multi-cluster we may need to send requests from the receiving cluster to a different cluster.

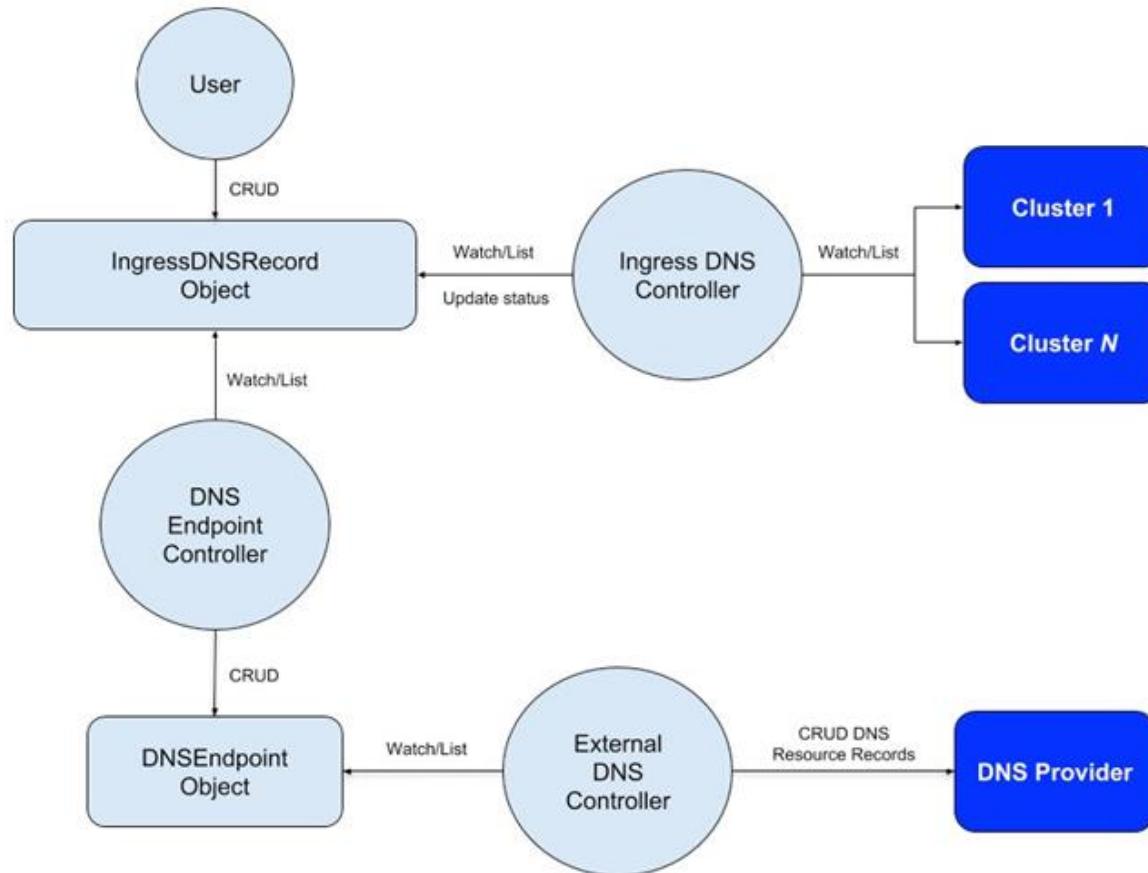
Finding the correct destination relies on an external DNS, which is used in addition to the **in-cluster CoreDNS**.

The primary idea is that the endpoints from all of the clusters are managed by a **DNS endpoint controller** and an **Ingress DNS controller**.

They watch all the clusters and update the multi-cluster **IngressDNSRecord** and domain names.

An **External DNS Controller** interacts with the external **DNS Provider** to assign external names that are valid across all the clusters and allow you to locate endpoints across clusters.

# Multi-cluster Ingress DNS



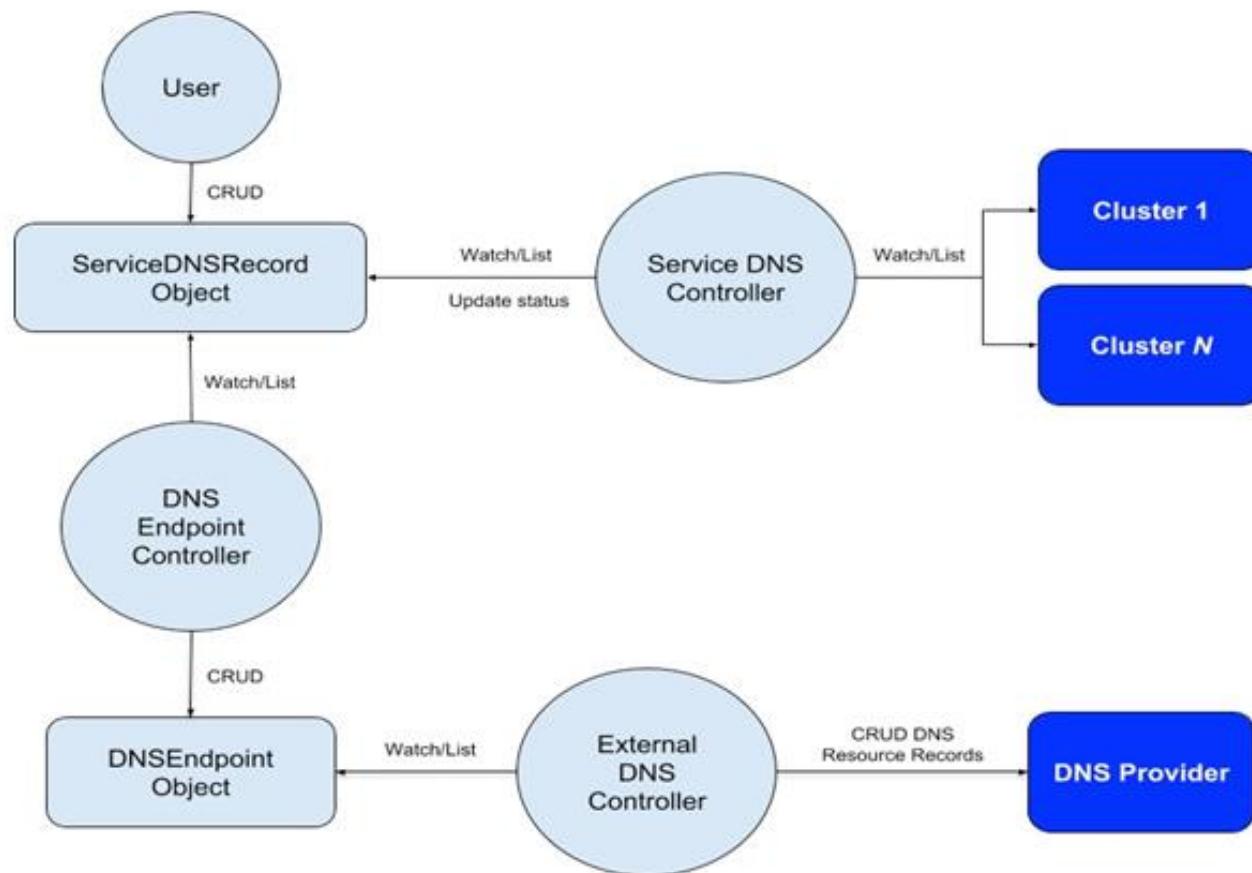
# Utilizing multi-cluster Service DNS

In a Kubernetes federation, services need to be federated, which means their backing pods may be federated across multiple clusters.

The typical workflow is:

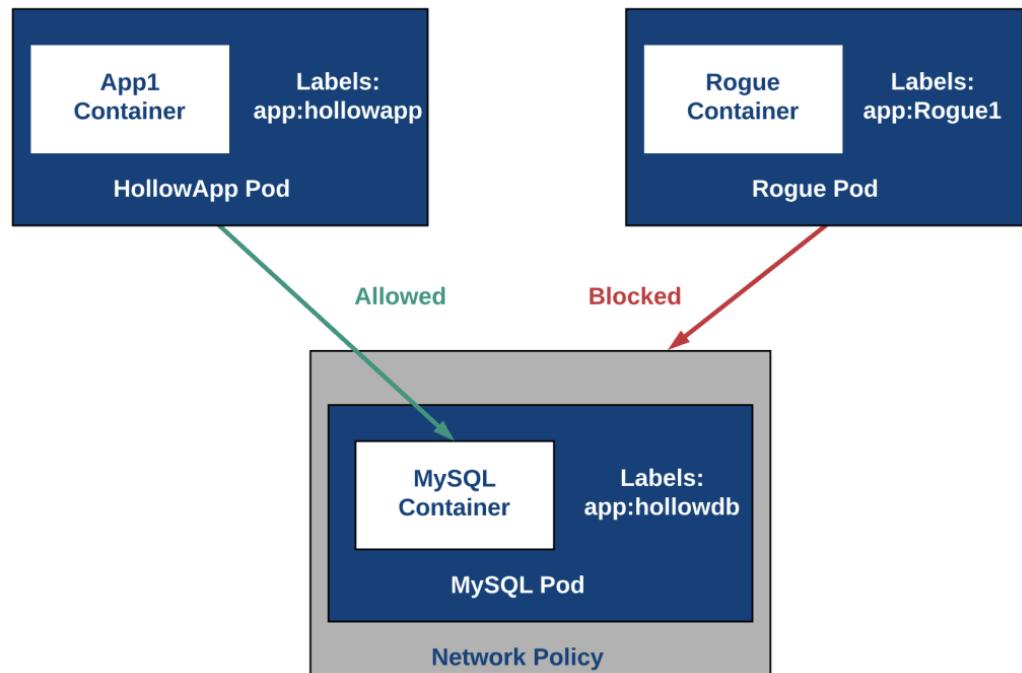
- Create deployment and service objects.
- Create a domain object that associates a DNS zone and an authoritative nameserver for the control plane.
- Create a record that identifies the intended domain name of a multi-cluster Service object.
- The DNS endpoint controller creates an endpoint for the service record.
- An external DNS system watches and lists endpoint objects and creates DNS resource records in external DNS providers.

# Multi-cluster Service DNS

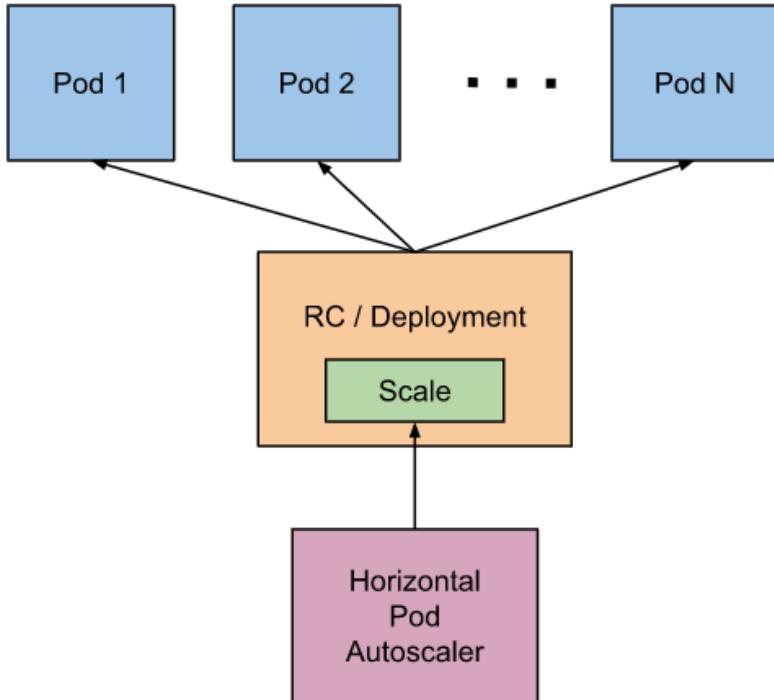


# Network Policies

Kubernetes clusters allow traffic between all pods by default, but if you've got a network plugin capable of using Network Policies, then you can start locking down that traffic. Build your Network Policy manifests and start including them with your application manifests.

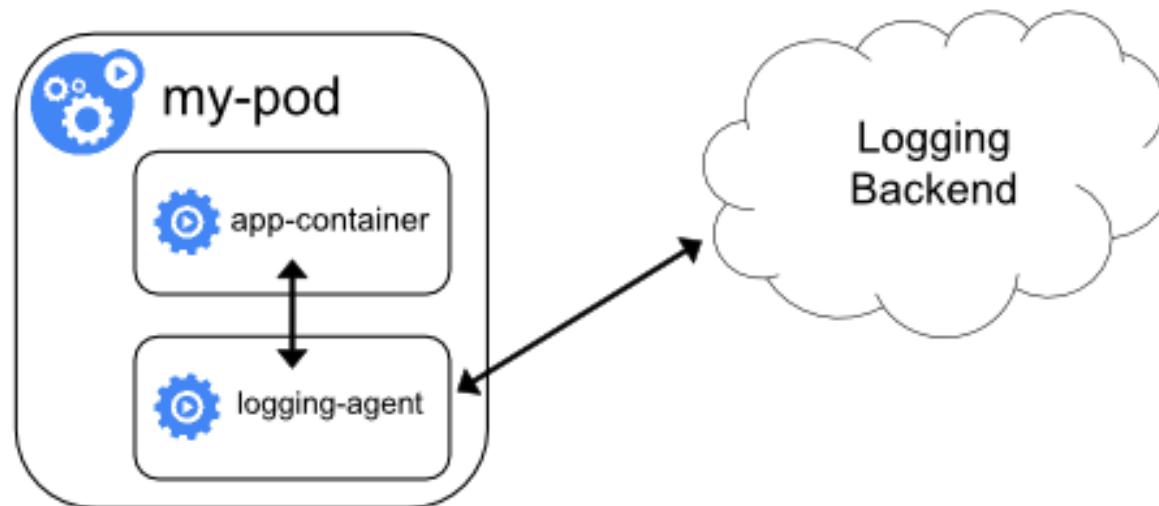


# Pod Scaling - Horizontal Pod Autoscaling



The Horizontal Pod Autoscaler automatically scales the number of Pods in a replication controller, deployment, replica set or stateful set based on observed CPU utilization (or, with [custom metrics](#) support, on some other application-provided metrics). Note that Horizontal Pod Autoscaling does not apply to objects that can't be scaled, for example, DaemonSets.

# Pod Logging



# Cluster level logging

Cluster-level logging architectures require a separate backend to store, analyze, and query logs. Kubernetes does not provide a native storage solution for log data.

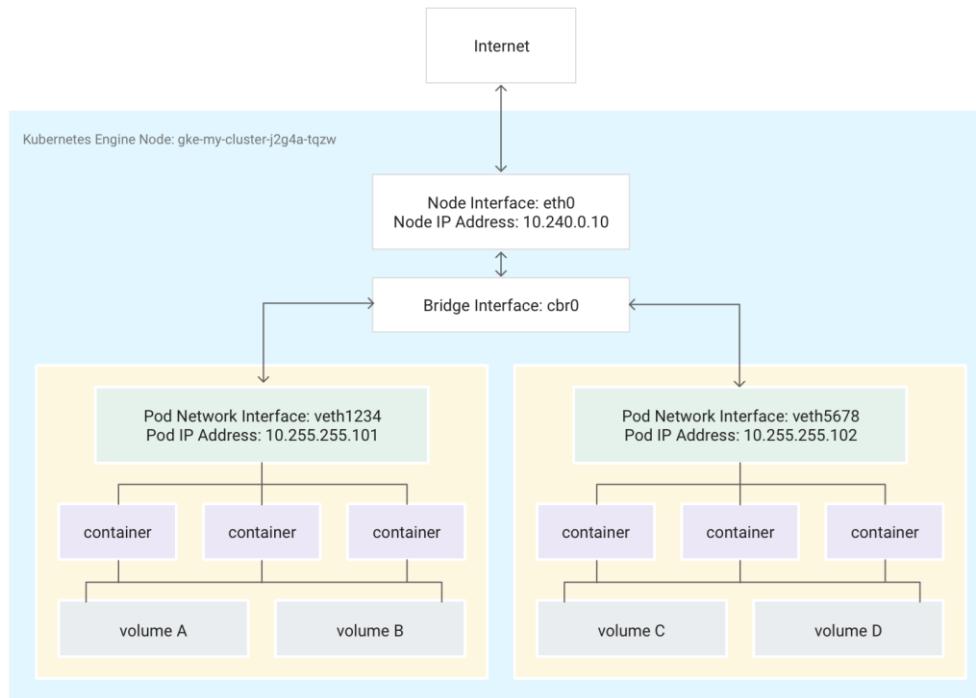
There are many logging solutions that integrate with Kubernetes.

# Pod Service Discovery

If you're able to use Kubernetes APIs for service discovery in your application, you can query the API server for Endpoints, that get updated whenever the set of Pods in a Service changes.

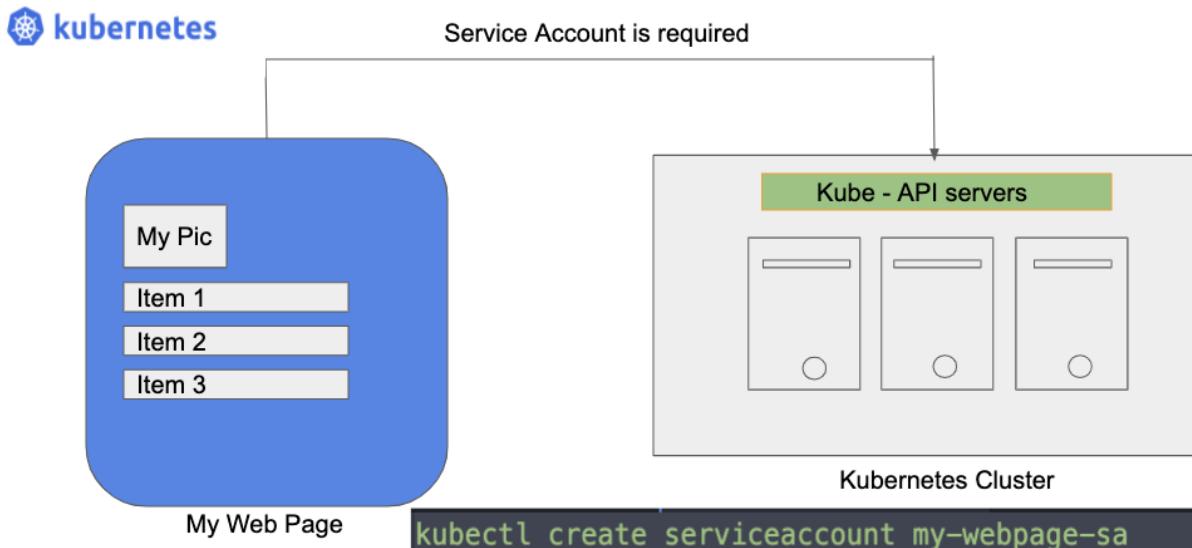
For non-native applications, Kubernetes offers ways to place a network port or load balancer in between your application and the backend Pods

# Pod Ip Addressing



A virtual network (**vnet**) device is a virtual device that is defined in a domain connected to a virtual switch. A virtual network device is managed by the virtual network driver, and it is connected to a virtual network through the hypervisor using logical domain channels (LDCs).

# Service Accounts/ Identity



```
kubectl describe secret <token-name>
```

```
kubectl describe secret my-webpage-sa-token-zngkh
```

# Pod and Containers

A *Pod* (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context.

A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled.

As well as application containers, a Pod can contain init containers that run during Pod startup. You can also inject ephemeral containers for debugging if your cluster offers this.

# Pod and Controllers

The Job controller is an example of a Kubernetes built-in controller. Built-in controllers manage state by interacting with the cluster API server.

Job is a Kubernetes resource that runs a Pod, or perhaps several Pods, to carry out a task and then stop.

Once [scheduled](#), Pod objects become part of the desired state for a kubelet.

When the Job controller sees a new task it makes sure that, somewhere in your cluster, the kubelets on a set of Nodes are running the right number of Pods to get the work done.

The Job controller does not run any Pods or containers itself. Instead, the Job controller tells the API server to create or remove Pods.

# Connect applications and services

Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on.

Kubernetes gives every pod its own cluster-private IP address, so you do not need to explicitly create links between pods or map container ports to host ports.

This means that containers within a Pod can all reach each other's ports on localhost, and all pods in a cluster can see each other without NAT.

The rest of this document elaborates on how you can run reliable services on such a networking model.

# POP QUIZ:

## Kubernetes



What is a Pod

- A: A host machine where containers are deployed
- B: A kubernetes primitive responsible for deploying and scheduling application containers
- C: A group of one or more application containers that include shared volume and IP address

# POP QUIZ:

## Kubernetes



What is a Pod

A: A host machine where containers are deployed

B: A kubernetes primitive responsible for deploying and scheduling application containers

C: A group of one or more application containers that include shared volume and IP address

# POP QUIZ:

## Kubernetes



What is a Node?

- A: A group of one or more containers deployed on Kubernetes
- B: A node is a worker machine in Kubernetes
- C: A node is a machine that coordinates the cluster



# POP QUIZ:

## Kubernetes



What is a Node?

A: A group of one or more containers deployed on Kubernetes

B: A node is a worker machine in Kubernetes

C: A node is a machine that coordinates the cluster

# POP QUIZ:

## Kubernetes



Kube-apiserver on Kubernetes master is design to scale?

- A: Vertically
- B: Horizontally
- C:: Both
- D: Neither



# POP QUIZ:

## Kubernetes



Kube-apiserver on Kubernetes master is design to scale?

- A: Vertically
- B: Horizontally
- C:: Both
- D: Neither



# Experiment – AWS EKS Networking



# Networking Plugins



# Network Plugins in K8S

Network plugins in Kubernetes come in a few flavors:

- CNI plugins: adhere to the [Container Network Interface](#) (CNI) specification, designed for interoperability.
  - Kubernetes follows the [v0.4.0](#) release of the CNI specification.
- Kubenet plugin: implements basic cbr0 using the bridge and host-local CNI plugins

# Network Plugin Requirements

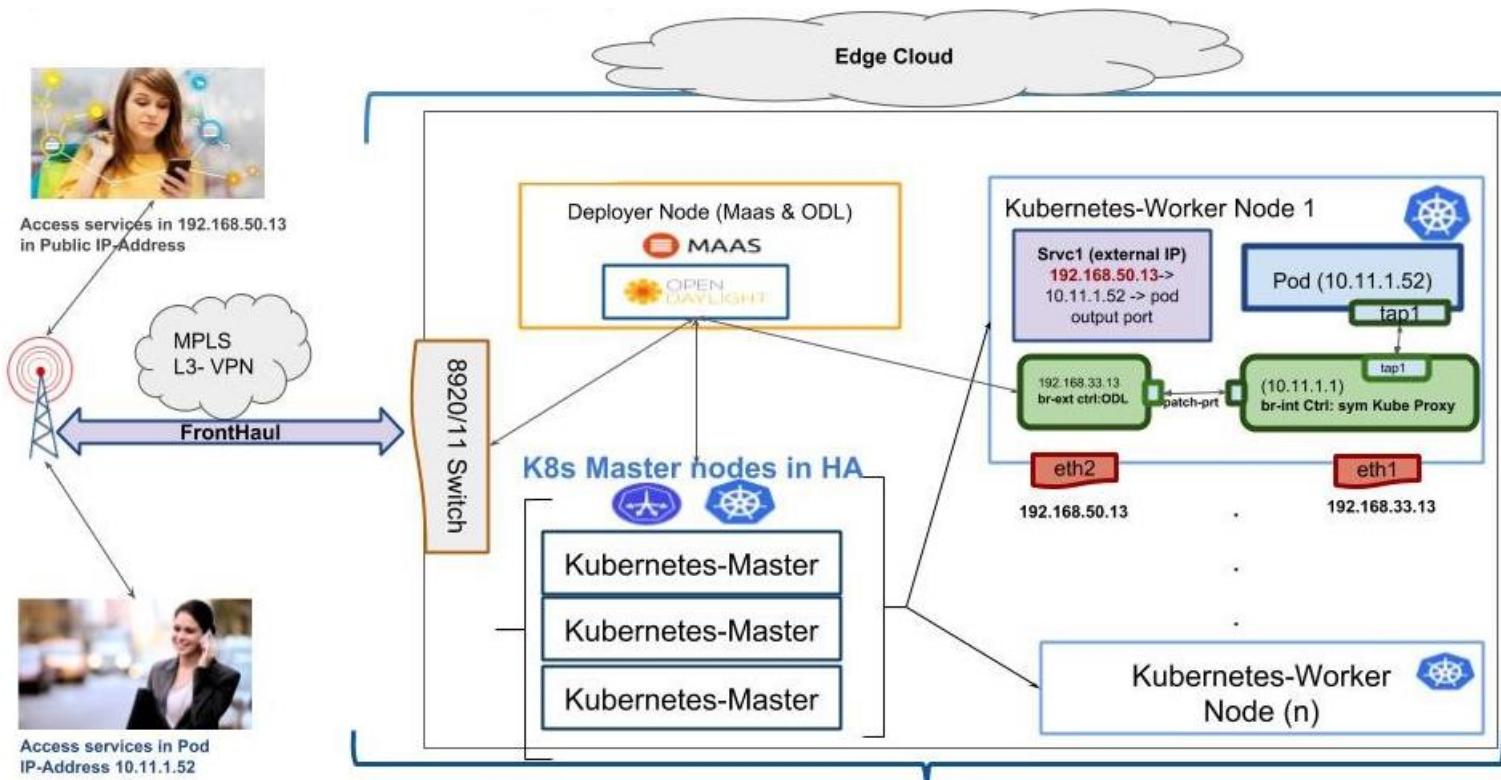
Network plugins in Kubernetes come in a few flavors:

- CNI plugins: adhere to the [Container Network Interface](#) (CNI) specification, designed for interoperability.
  - Kubernetes follows the [v0.4.0](#) release of the CNI specification.
- Kubenet plugin: implements basic `cbr0` using the bridge and host-local CNI plugins

# Network Plugin Interface

- Maintains compatibility with older k8s versions with the IPTables bridge
- Newer plugins are responsible for setting the NPI correctly depending on whether or not they connect containers to Linux bridges or some over mechanism, i.e. SDN vswitch

# SDN Adaptor



# Container Network Interface (CNI)

Network plugins in Kubernetes come in a few flavors:

- CNI plugins: adhere to the [Container Network Interface \(CNI\)](#) specification, designed for interoperability.
  - Kubernetes follows the [v0.4.0](#) release of the CNI specification.
- Kubenet plugin: implements basic cbr0 using the bridge and host-local CNI plugins

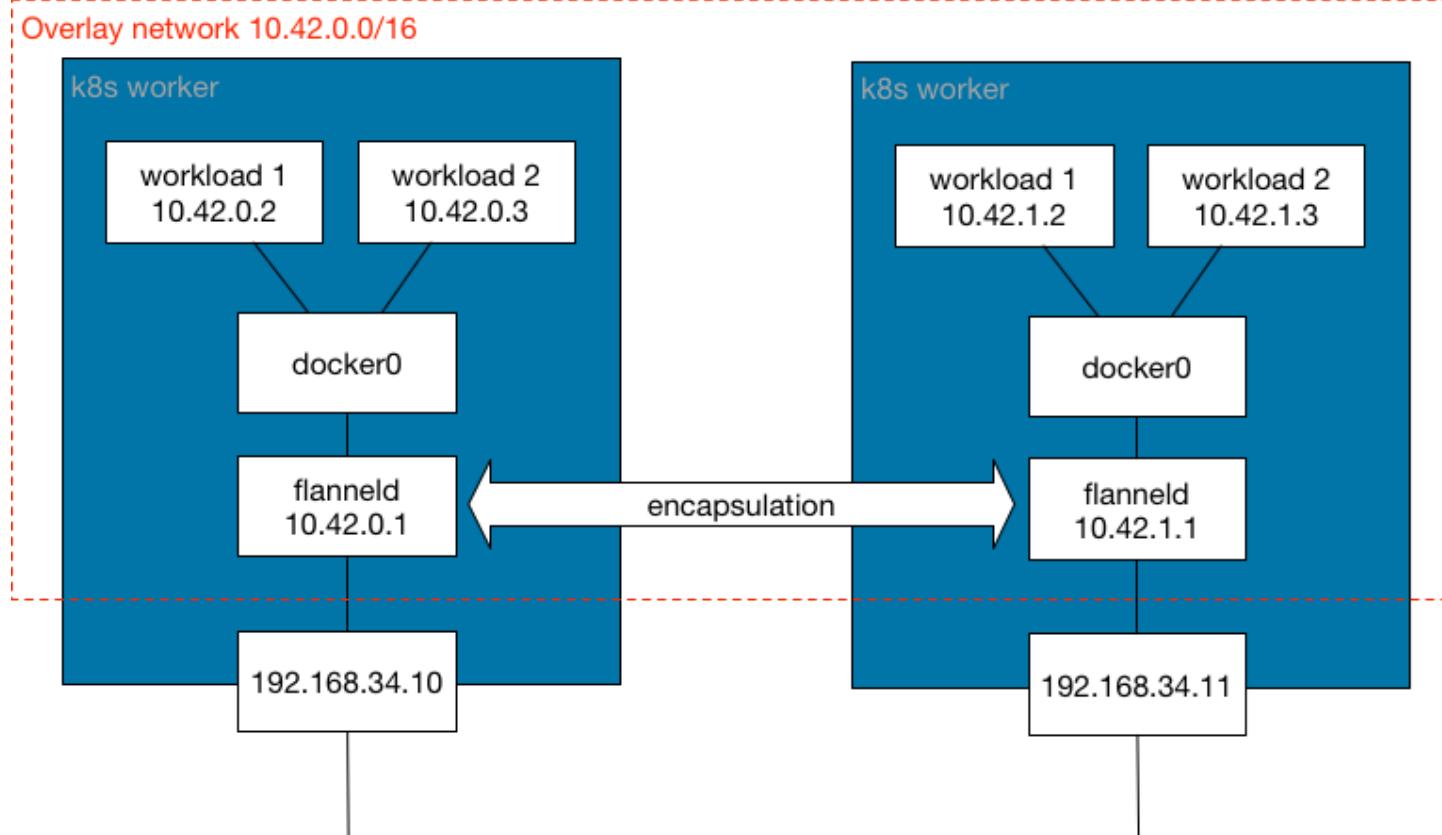
# CNI Network Models

CNI network providers implement their network fabric using either an encapsulated network model such as Virtual Extensible Lan ([VXLAN](#)) or an unencapsulated network model such as Border Gateway Protocol ([BGP](#)).

# Encapsulated Network Model

- Encapsulated networks generate a kind of **bridge** extended between Kubernetes workers, where pods are connected.
- This network model is used when an **extended L2 bridge** is preferred.
- Sensitive to L3 network latencies of the Kubernetes workers.
- Require **low latencies between data centers** (zones) to avoid eventual network segmentation.
- CNI network providers using this network model include **Flannel, Canal, and Weave**.

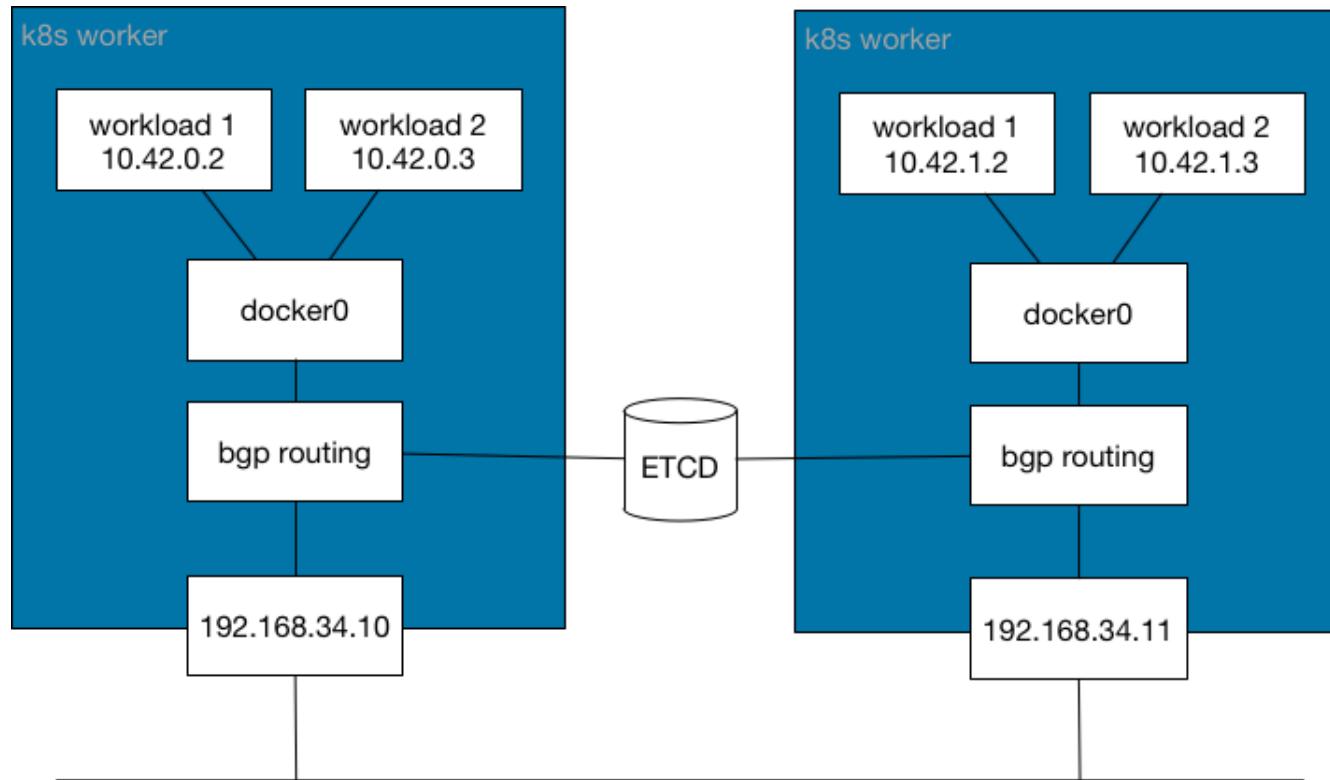
# Encapsulated Network Model



# Unencapsulated Network Model

- Unencapsulated networks generate a kind of **network router** extended between Kubernetes workers.
- This network model is used when a **routed L3 network** is preferred.
- This mode dynamically updates routes at the OS level for Kubernetes workers. It's **less sensitive to latency**.
- CNI network providers using this network model include **Calico** and **Romana**.

# Unencapsulated Network Model



# CNI Providers

<https://rancher.com/docs/rancher/v2.x/en/faq/networking/cni-providers/>

# Well Known CNI Plugins

Calico

Flannel -

Antrea – Vmware

ACI – Cisco

AWS VPC CNI

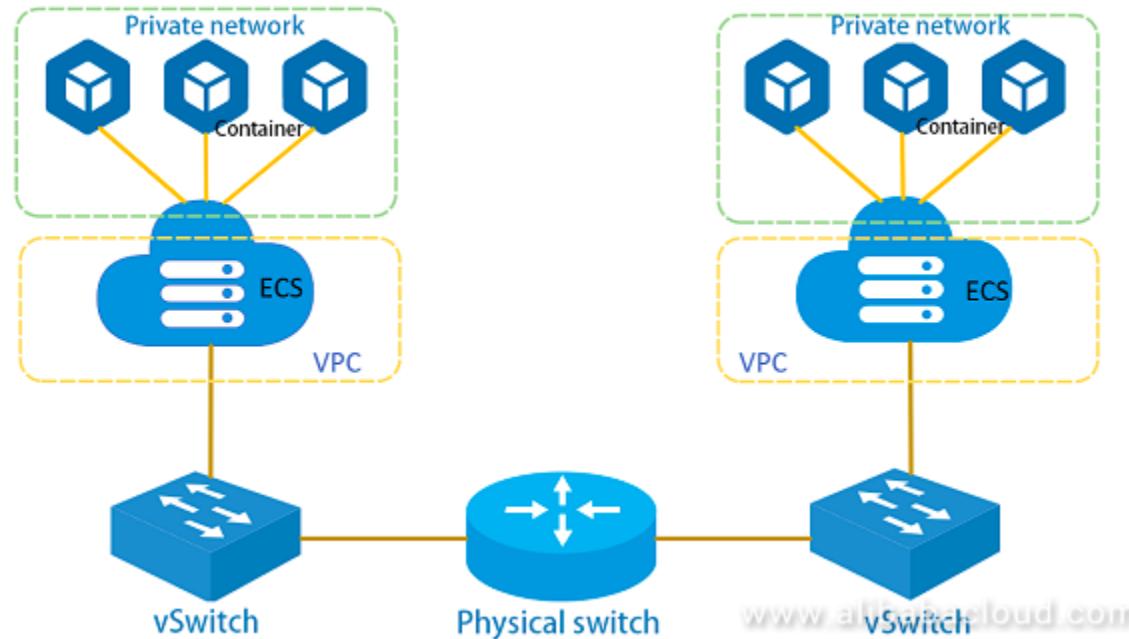
Azure CNI

Big Cloud Fabric – Big Switch Networks

Weave

Canal

# Mainstream Container Network



# Multi-NIC Container Networking

VMs with multiple Network Interface Cards (NICs) that are dynamically hot swappable can be used on a container network. With these the container network don't need to utilize technologies such as Linux VETH and Bridge.

# Kubelet – Default Networking Plugin

The kubelet has a single default network plugin, and a default network common to the entire cluster. It probes for plugins when it starts up, remembers what it finds, and executes the selected plugin at appropriate times in the pod lifecycle (this is only true for Docker, as CRI manages its own CNI plugins).

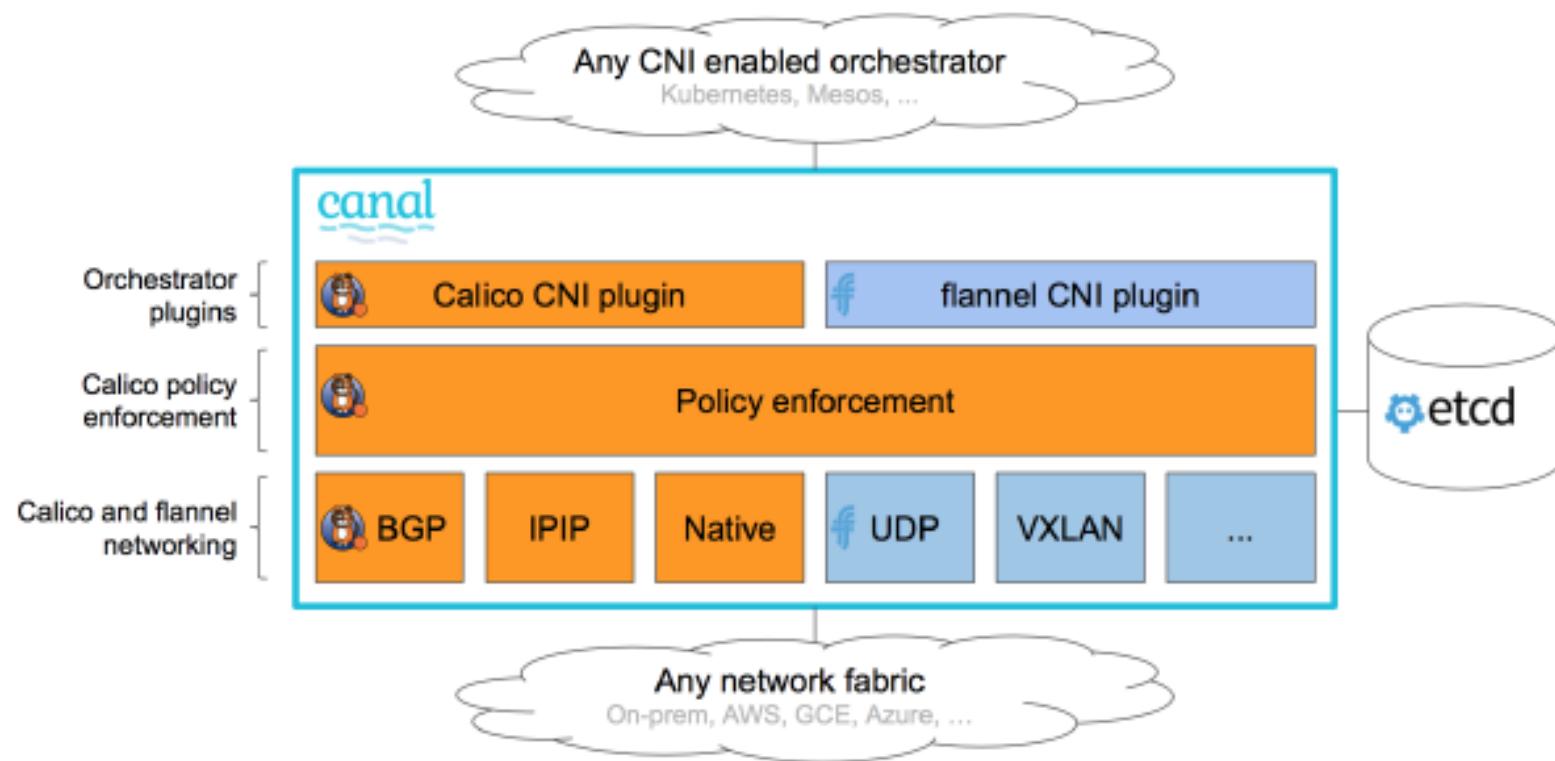
# Kubelet – Default Plugin Commands

There are two Kubelet command line parameters to keep in mind when using plugins:

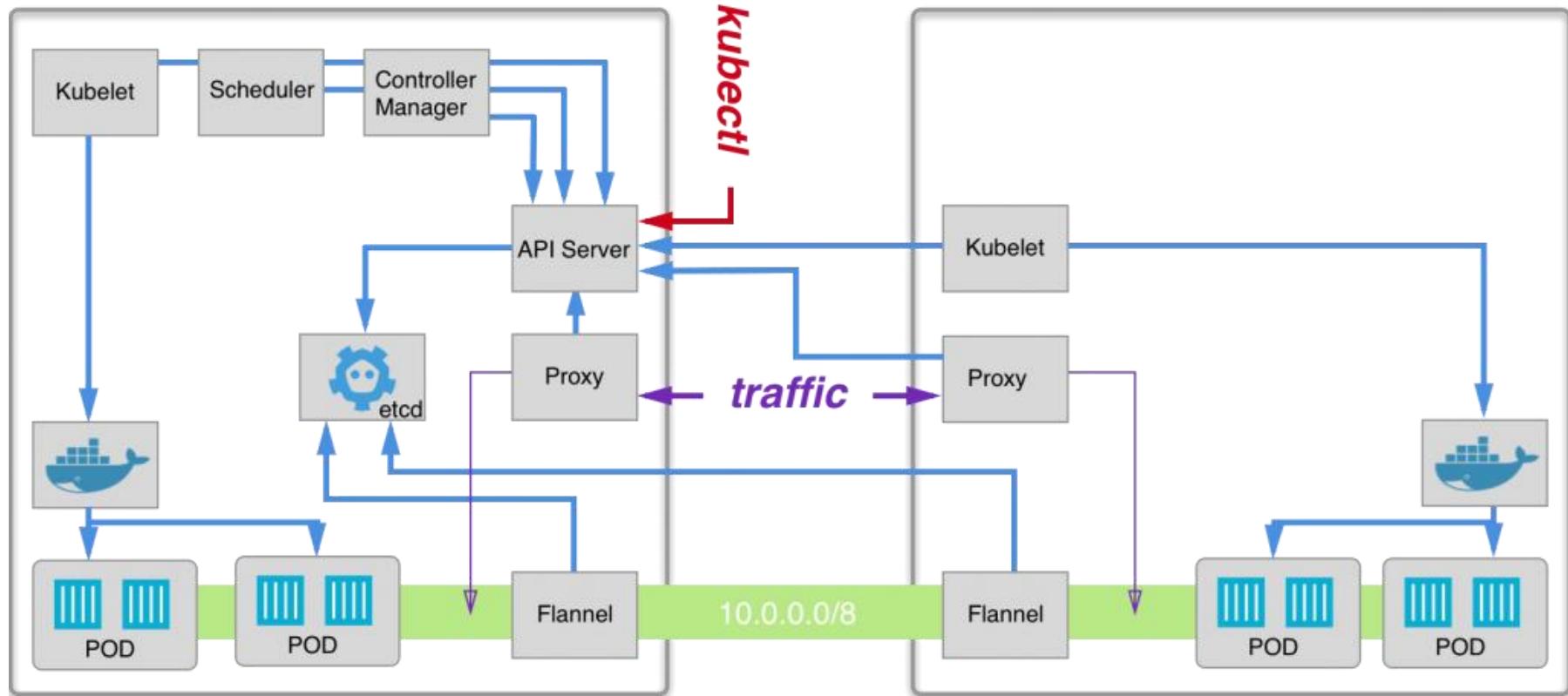
cni-bin-dir: Kubelet probes this directory for plugins on startup

network-plugin: The network plugin to use from cni-bin-dir. It must match the name reported by a plugin probed from the plugin directory. For CNI plugins, this is cni.

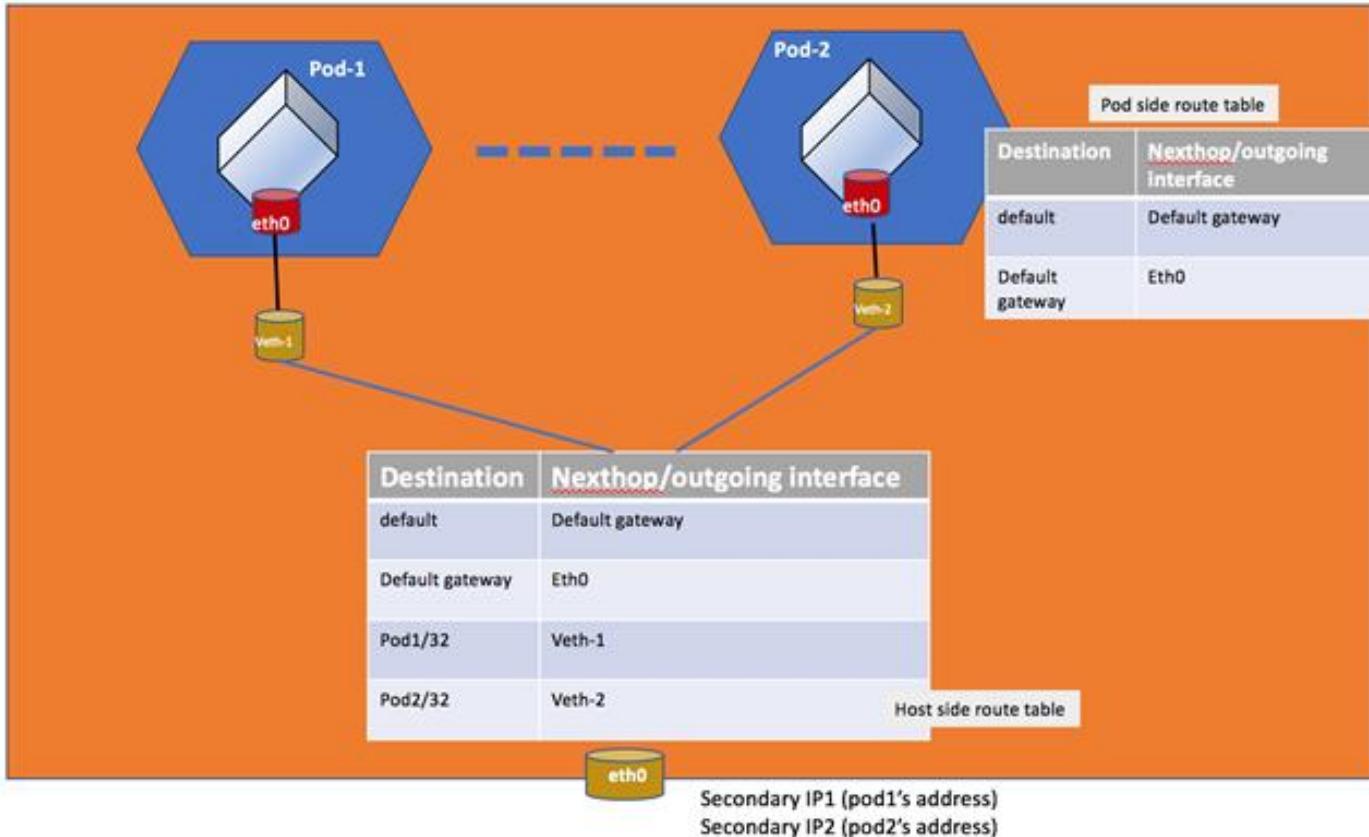
# CNI plugin - Canal



# CNI plugin - Flannel



# AWS EKS CNI



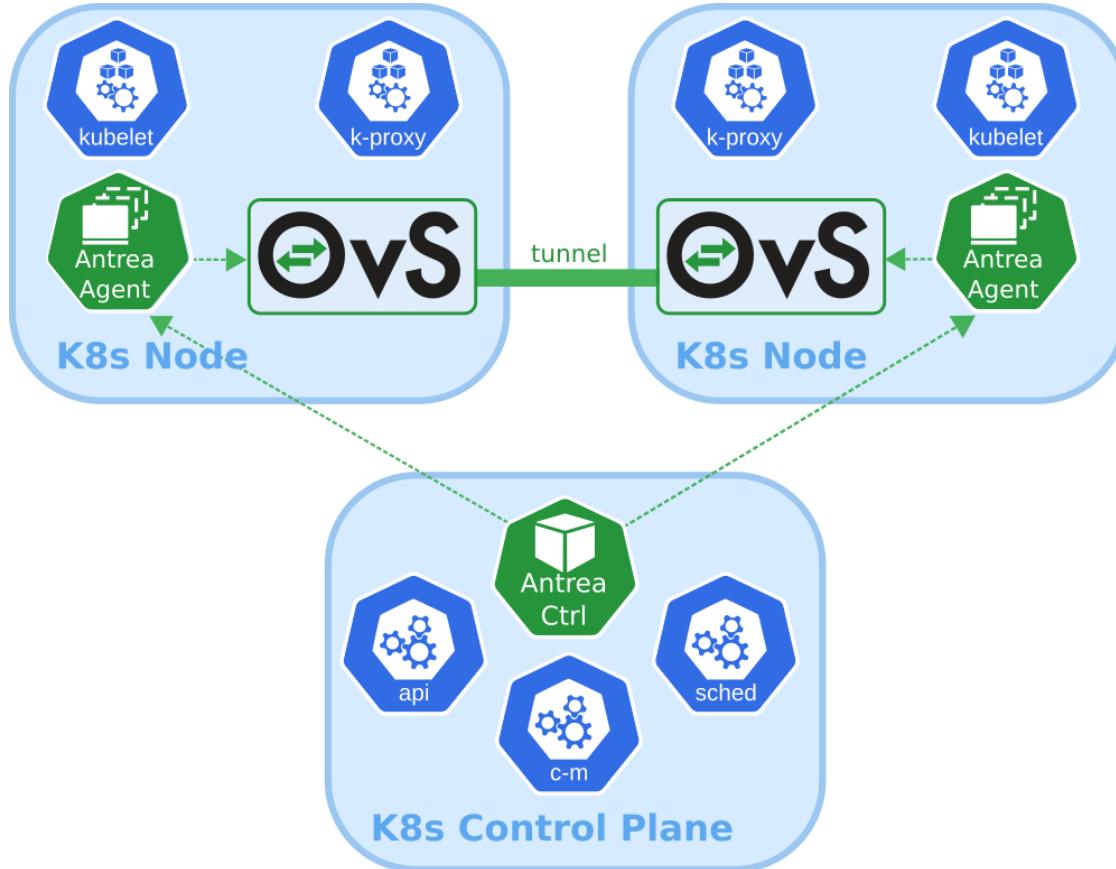
# CNI plugin - Antrea

- **VMware Container Networking™ with Antrea™** offers users signed images and binaries, along with full enterprise support for Project Antrea.
- VMware Container Networking integrates with managed Kubernetes services to further enhance Kubernetes network policies.
- Antrea also supports Windows and Linux workloads on Kubernetes across multiple clouds.

# What is Project Antrea

- Antrea is a Kubernetes-native project that implements the CNI and Kubernetes NetworkPolicy to provide network connectivity and security for pod workloads.
- Antrea uses Open vSwitch as the networking data plane in every Kubernetes node.
- Due to the programmable characteristic of Open vSwitch, Antrea extends the benefits of programmable networks and performance from Open vSwitch to Kubernetes

# Antrea in Action



# Key Benefits

- Simplified K8S networking
- Support for CPU-intensive workloads via SmartNICs
- Searmless integration with Prometheus and Traceflow
- Encrypted traffic between pods even on untrusted fabric
- VMware supported

# Proxies for K8S

- kubectl proxy
- apiserver proxy
- kube proxy
- Proxy/Load-balancer in front of apiserver(s)
- Cloud Load Balancers on external services

Kubernetes users will typically not need to worry about anything other than the first two types. The cluster admin will typically ensure that the latter types are setup correctly.

# Kubectl Proxv

- runs on a user's desktop or in a pod
- proxies from a localhost address to the Kubernetes apiserver
- client to proxy uses HTTP
- proxy to apiserver uses HTTPS
- locates apiserver
- adds authentication headers

# APIServer Proxy

- is a bastion built into the apiserver
- connects a user outside of the cluster to cluster IPs which otherwise might not be reachable
- runs in the apiserver processes
- client to proxy uses HTTPS (or http if apiserver so configured)
- proxy to target may use HTTP or HTTPS as chosen by proxy using available information
- can be used to reach a Node, Pod, or Service
- does load balancing when used to reach a Service

# Kube Proxy

- runs on each node
- proxies UDP, TCP and SCTP
- does not understand HTTP
- provides load balancing
- is only used to reach services

# Proxy/Load Balancer w/ APIServer

- existence and implementation varies from cluster to cluster (e.g. nginx)
- sits between all clients and one or more apiservers
- acts as load balancer if there are several apiservers.

# Cloud Load Balancers

- are provided by some cloud providers (e.g. AWS ELB, Google Cloud Load Balancer)
- are created automatically when the Kubernetes service has type LoadBalancer
- usually supports UDP/TCP only
- SCTP support is up to the load balancer implementation of the cloud provider
- implementation varies by cloud provider.

# Troubleshooting Networking



# CNI Configuration Errors

```
kubelet: E0714 12:45:30.541001 7263 kubelet.go:2  
136] Container runtime network not ready: Networ  
kReady=false reason:NetworkPluginNotReady messag  
e: network plugin is not ready: cni config unini  
tialized
```

# CNI Configuration

cni config is installed  
to `/etc/cni/net.d` and `/opt/cni/bin` w  
here cni's are typically installed if  
you use kubeadm:

# Ensure Kubernetes is installed and running correctly

There are **a lot** of different platform and open-source actors that are needed to operate a Kubernetes cluster. It can be hard to keep track of all of them - especially given that they release at a different cadence. Scripting to validate the components is an important part of CI/CD pipelines.

# Component Validation Scripting

```
PS C:\k\yaml> .\Debug-WindowsNode.ps1
Checking for common problems with Windows Kubernetes nodes
Container Host OS Product Name: Windows Server 2019 Datacenter
Container Host OS Build Label: 17763.1.amd64fre.rs5_release.180914-1434
Describing Windows Version and Prerequisites
[+] Is Windows Server 2019 108ms
[+] Has 'Containers' feature installed 3.31s
[+] Has HNS running 77ms
Describing Docker is installed
[+] A Docker service is installed - 'Docker' or 'com.Docker.Service' 74ms
[+] Service is running 47ms
[+] Docker.exe is in path 251ms
[+] Should be a supported version 61ms
Docker version: 18.09.5
Describing Kubernetes processes are running
[+] There is 1 running kubelet.exe process 70ms
[+] There is 1 running kube-proxy.exe process 54ms
PS C:\k\yaml> ■
```

# Component Listing

```
Administrator: Windows PowerShell
PS C:\Users\Administrator> ps |findstr "kube flanneld"
 276      16    24148      27392      0.39    1568    2 flanneld
 347      21    42164      56220      1.86    4708    2 kubelet
 239      18    23320      28436      0.41    8136    2 kube-proxy
PS C:\Users\Administrator> -
```

Any piece of software can crash or enter a deadlock-like state, including host-agent processes such as kubelet.exe or kube-proxy.exe. Running a simple ps command may be sufficient to identify those issues.

# Validate basic cluster connectivity

There are test suites which can validate basic connectivity scenarios and report on success/failure. There may be configuration and cluster requirements.

# Validate basic cluster connectivity

There are test suites which can validate basic connectivity scenarios and report on success/failure. There may be configuration and cluster requirements.

# Event logs

After verifying that all the processes are running as expected, the next step is to query the built-in Kubernetes event logs and see what the basic built-in health-checks that ship with K8s using the “kubectl describe” command.

```
PS C:\k\yaml> kubectl get pods -o wide
NAME                  READY   STATUS            RESTARTS   AGE     IP           NODE
win-webserver-69495c7694-76nxd  0/1    ContainerCreating  0          5s    <none>      win-24elfepi8d9
win-webserver-69495c7694-tjf7f  0/1    ContainerCreating  0          6s    <none>      win-24elfepi8d9
PS C:\k\yaml> kubectl describe po/win-webserver-69495c7694-76nxd
```

# Analyze Container Creation Logs

Another useful source of information that can be leveraged to perform root-cause analysis for failing container creations is the kubelet, FlannelD, and kube-proxy logs. These components all have different responsibilities, and your configuration may have additional logging that should be monitored.

# Problem Areas for Investigation

Component	Responsibility	When to inspect?
Kubelet	Interacts with container runtime (e.g. Dockershim) to bring up containers and pods.	Erroneous pod creations/configurations
Kube-proxy	Manages network connectivity for containers (programming policies used for NAT'ing or load balancing).	Mysterious network glitches, in particular for service discovery and communication
FlannelID	Responsible for keeping all the nodes in sync with the rest of the cluster for events such as node removal/addition. This consists of assigning IP blocks (pod subnets) to nodes as well as plumbing routes for inter-node connectivity.	Failing inter-node connectivity

# Kubernetes Troubleshooting Tools

PowerfulSeal – [A powerful testing tool for Kubernetes clusters](#)

Crash-diagnostic – [Crash-Diagnostics is a tool to help investigate, analyze, and troubleshoot unresponsive or crashed Kubernetes clusters](#)

K9s – [Kubernetes CLI To Manage Your Clusters In Style!](#)

Kubernetes CLI Plugin – Doctor – [kubectl cluster triage plugin for k8s](#)

Knative Inspect – [A light-weight debugging tool for Knative's system components](#)

Kubeman – [To find information from Kubernetes clusters, and to investigate issues related to Kubernetes and Istio](#)

Kubectl-debug – [Debug your pod by a new container with every troubleshooting tools pre-installed](#)

ksniff – [Kubectl plugin to ease sniffing on kubernetes pods using tcpdump and wireshark](#)

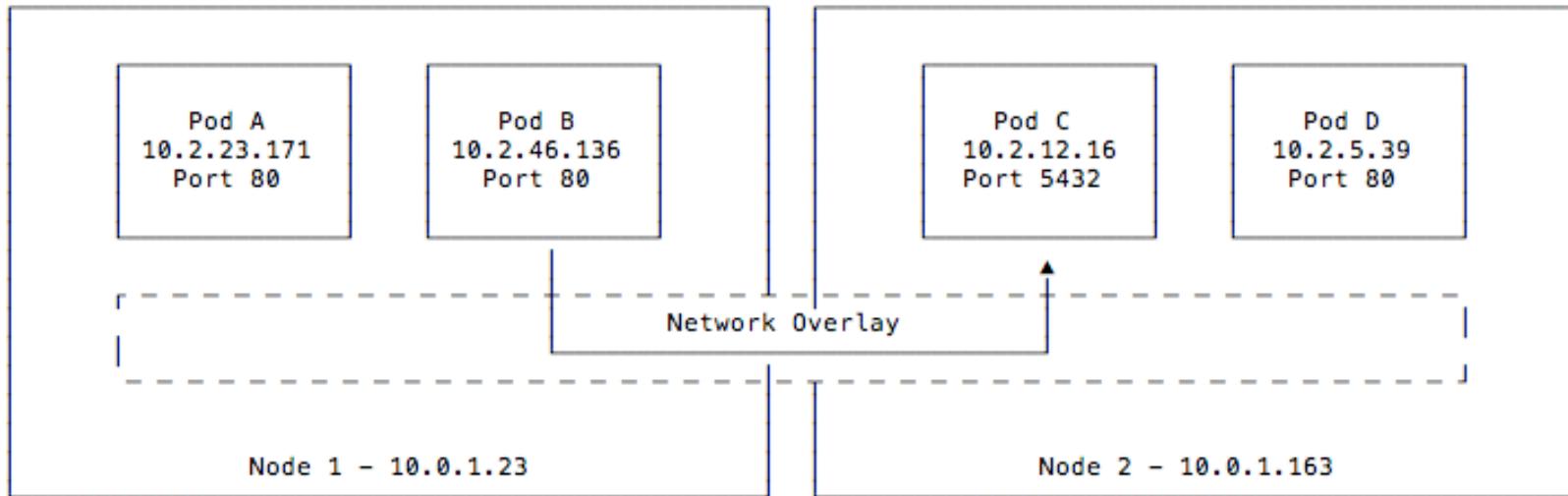
# Kubernetes Network Overlay

Pods are the scheduling primitives in Kubernetes. Each pod is composed of multiple containers that can optionally expose ports.

However, because pods may share the same host on the same ports, workloads must be scheduled in a way that ensures ports do not conflict with each other on a single machine.

To solve this problem, Kubernetes uses a network overlay. In this model, pods get their own virtual IP addresses to allow different pods to listen to the same port on the same machine.

# Kubernetes Network Overlay



# Monitoring



# Observability & Monitoring

**Observability** is the property of the system that defines what we can tell about the state and behavior of the system, right now and historically.

- Gain understanding actively
- Ask questions based on hypotheses
- Build to tame dynamic environments with changing complexity
- Preferred by developers of systems with variability and unknown permutations

**Monitoring** is the collection of tools, processes, and techniques we use to increase the observability of the system.

- Consume information passively
- Ask questions based on dashboards
- Built to maintain based on dashboards

Built to maintain static environments with little variation

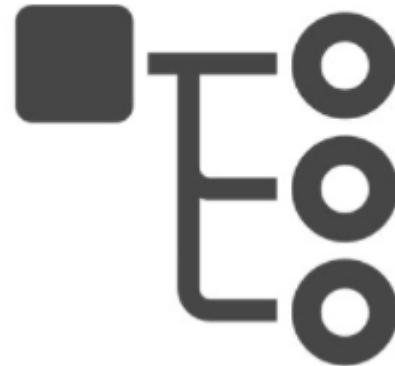
Used by developers of systems with little change and known permutations

# Observability

Three pillars of observability



Metrics



Traces



Logs



# Why?



# Rise of Prometheus

The following diagram illustrates the entire system:

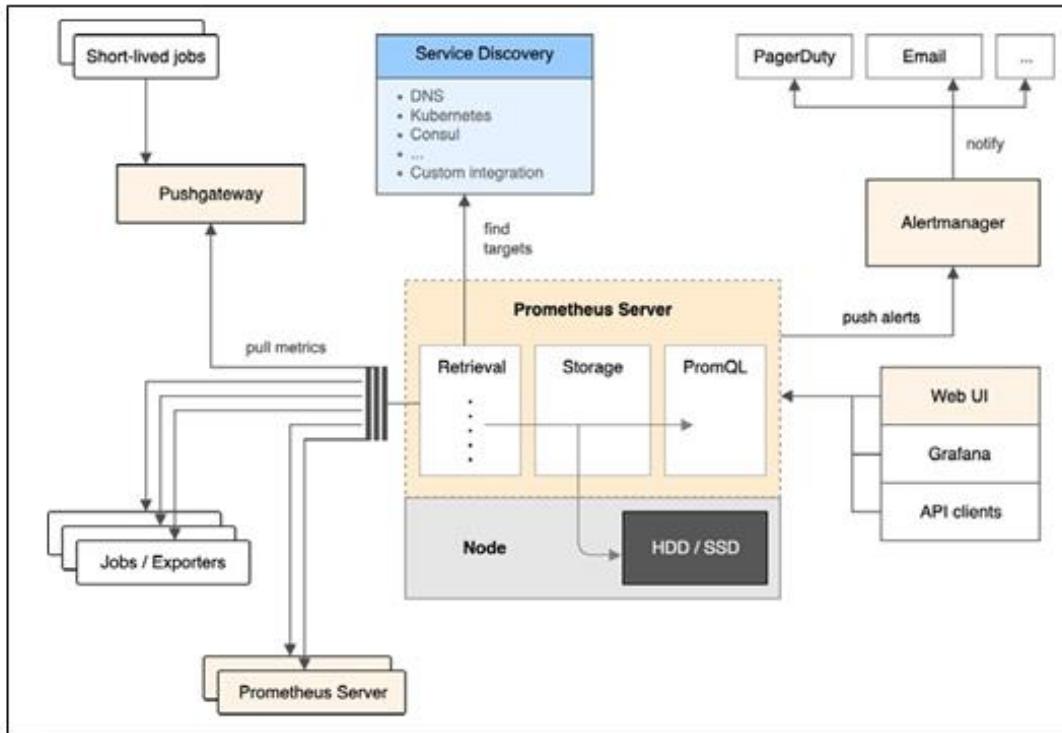
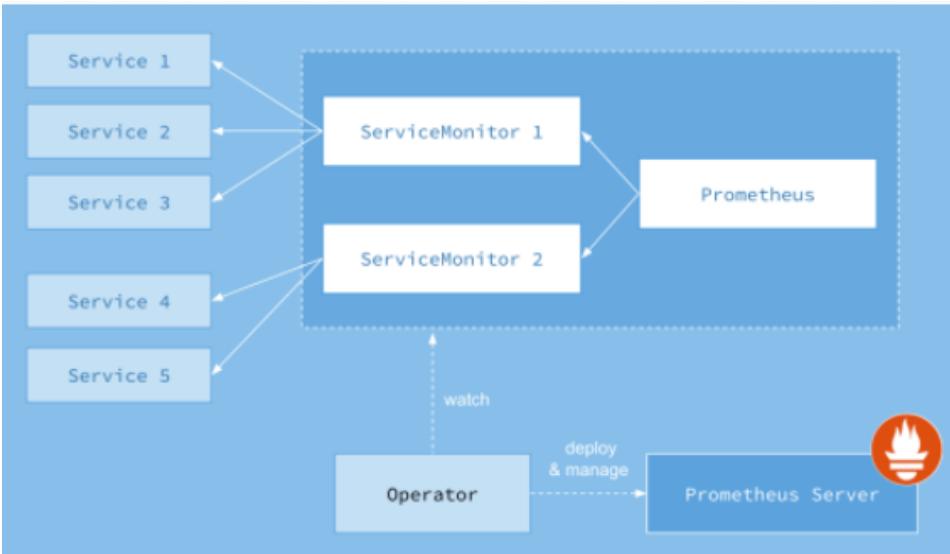


Figure 13.7: the Prometheus system

# Installing Prometheus

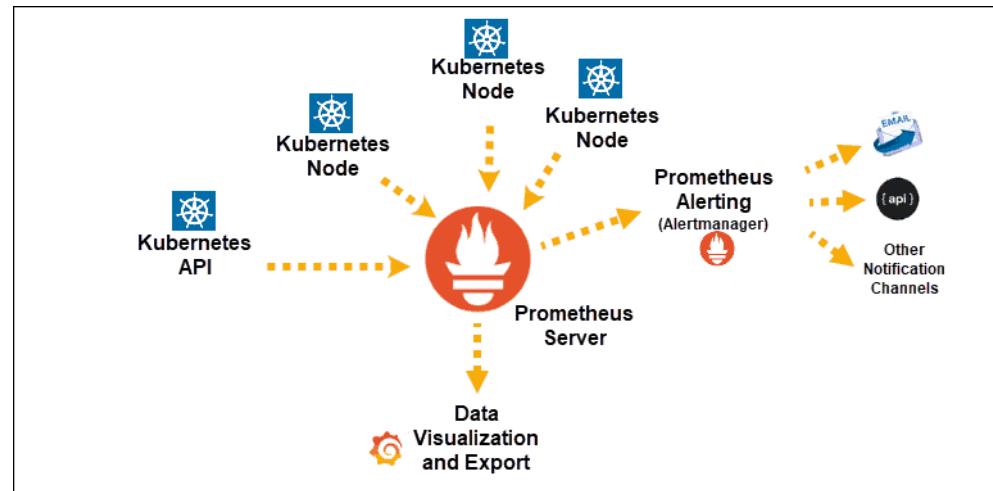


# Monitoring with Prometheus

## Why Use Prometheus for Kubernetes Monitoring

Two technology shifts took place that created a need for a new monitoring framework:

1. DevOps culture
2. Containers and Kubernetes



# Interacting With Prometheus

Prometheus has a basic web UI that you can use to explore its metrics. Let's do port forwarding to localhost:

```
$ POD_NAME=$(kubectl get pods -n monitoring  
-l "app=prometheus" \  
-o  
jsonpath="{.items[0].metadata.na  
me}")  
  
$ kubectl port-forward -n monitoring  
$POD_NAME 9090
```

Then, you can browse to <http://localhost:9090>, where you can select different metrics and view raw data or graphs:

Prometheus records an outstanding number of metrics (990, in my current setup). The most relevant metrics on Kubernetes are the metrics exposed by kube-state-metrics and node exporters.

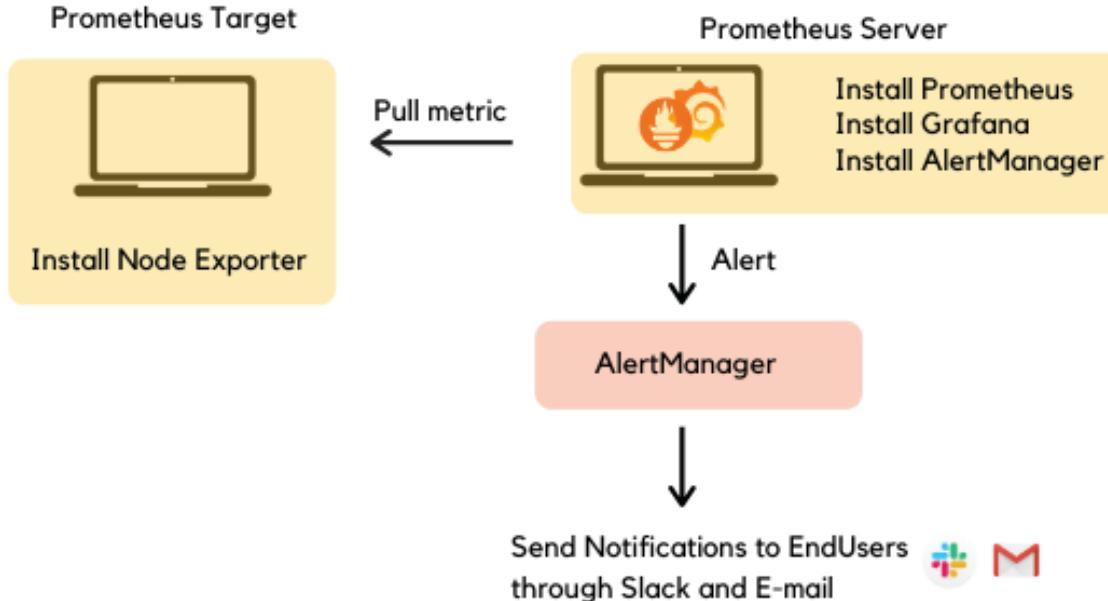


# Incorporating Kube-state-metrics

The Prometheus operator already installs kube-state-metrics. It is a service that listens to Kubernetes events and exposes them through a /metrics HTTP endpoint in the format that Prometheus expects.



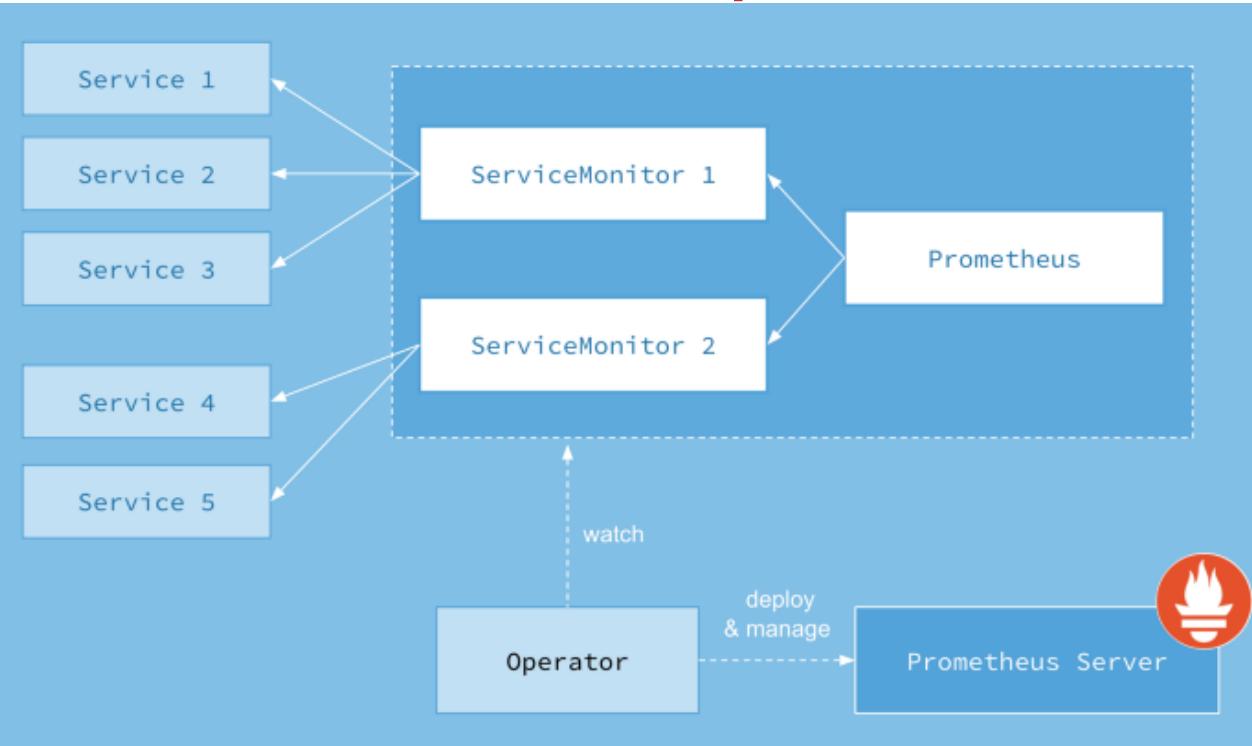
# Triggering Alerts with AlertManager





# Prometheus

# Prometheus Operator



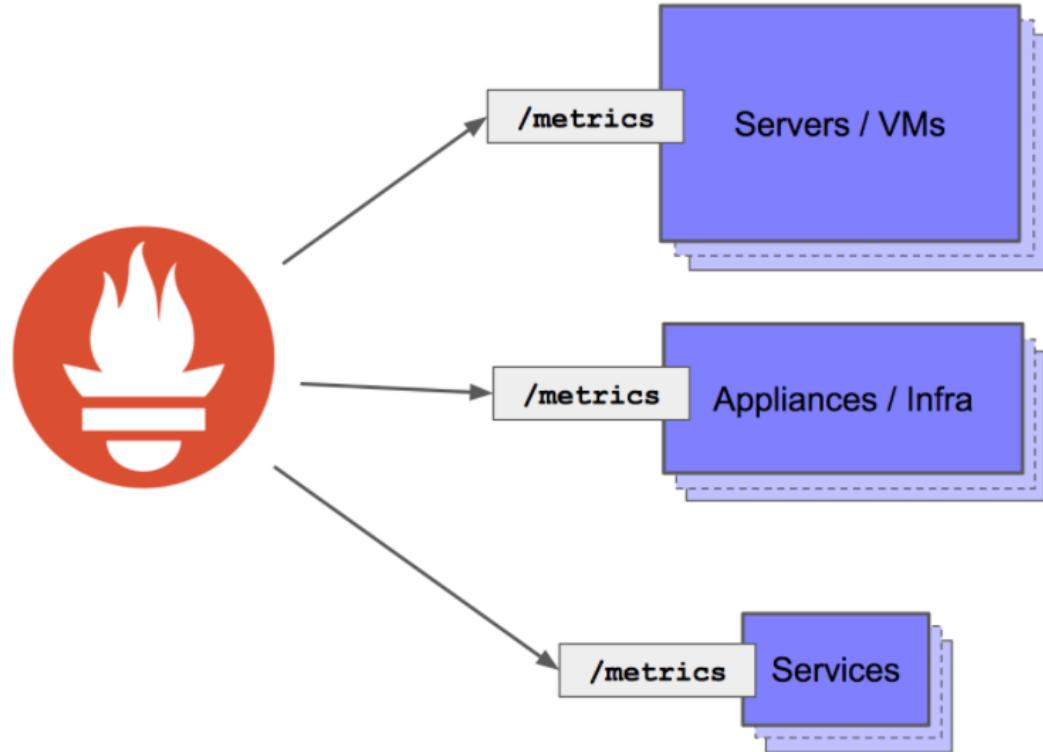
The Prometheus Operator provides easy monitoring for k8s services and deployments besides managing Prometheus, Alertmanager and Grafana configuration.



# Prometheus Operator

- **Kubernetes Custom Resources:** Use Kubernetes custom resources to deploy and manage Prometheus
- **Simplified Deployment Configuration:** Configure the fundamentals of Prometheus like versions, persistence, retention policies, and replicas
- **Prometheus Target Configuration:** Automatically generate monitoring target configurations based on Kubernetes label queries

# Exposing Metrics with Prometheus



# Deploying Prometheus Using Helm

1. Update the Helm repository. This command will fetch up-to-date charts locally from public chart repositories:

```
$ helm repo update
```

2. Deploy Prometheus Operator in the monitoring namespace using the `helm install` command. This command will deploy Prometheus along with the Alertmanager, Grafana, the node-exporter and kube-state-metrics addon; basically, a bundle of the components needed to use Prometheus on a Kubernetes cluster:

```
$ helm install stable/prometheus-operator --name prometheus \
--namespace monitoring
```

3. Verify the status of the pods deployed in the monitoring namespace:

```
$ kubectl get pods -n monitoring
NAME READY STATUS RESTARTS AGE
alertmanager-prometheus-prometheus-oper-alertmanager-0 2/2 Running
0 88s
prometheus-grafana-6c6f7586b6-f9jbr 2/2 Running 0 98s
prometheus-kube-state-metrics-57d6c55b56-wf4mc 1/1 Running 0 98s
prometheus-prometheus-node-exporter-8drg7 1/1 Running 0 98s
prometheus-prometheus-node-exporter-lb715 1/1 Running 0 98s
prometheus-prometheus-node-exporter-vx7w2 1/1 Running 0 98s
prometheus-prometheus-oper-operator-86c9c956dd-88p82 2/2 Running 0
98s
prometheus-prometheus-prometheus-oper-prometheus-0 3/3 Running 1
78s
```



Prometheus

# Node Exporter

`kube-state-metrics` collects node information from the Kubernetes API server, but this information is pretty limited.

Prometheus comes with its own node exporter, which collects tons of low-level information about the nodes.

Remember that Prometheus may be the de facto standard metrics platform on Kubernetes, but it is not Kubernetes-specific.

Here is a small subset of the metrics exposed by the node exporter:

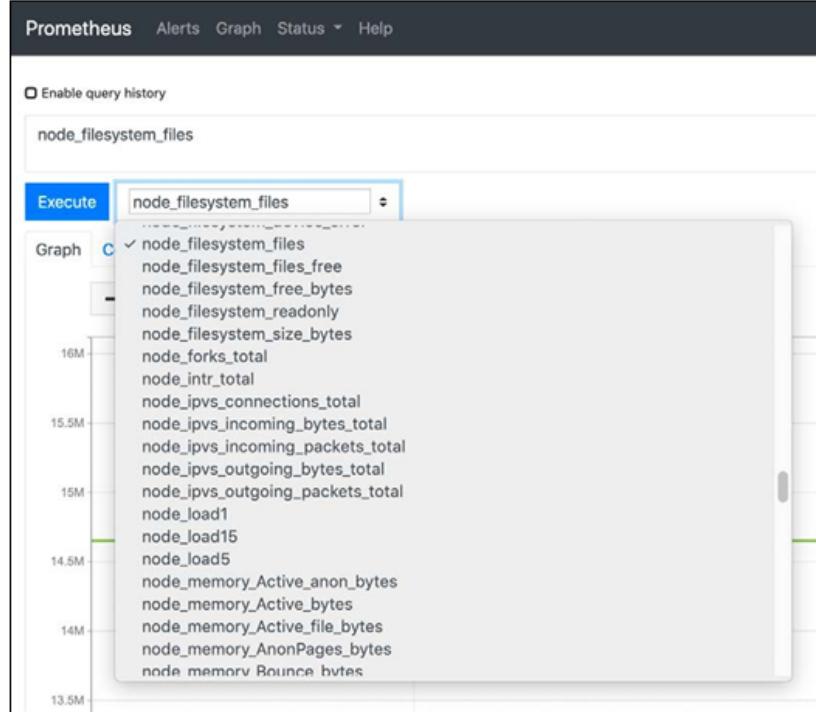


Figure 13.9: Metrics exposed by the node exporter

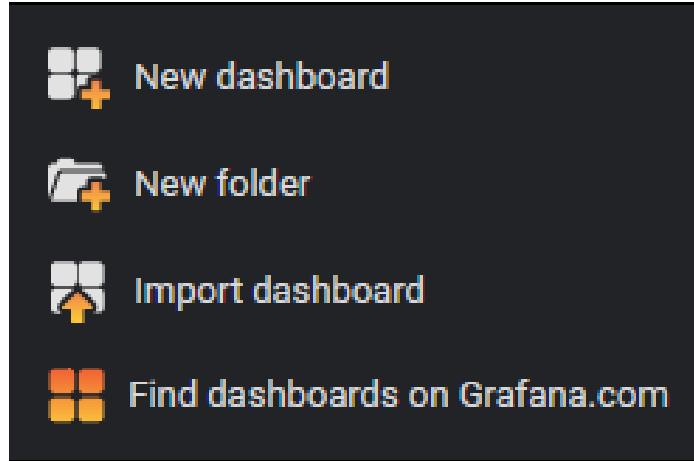
# Grafana

Grafana is an open source analytics and monitoring solution. By default, Grafana is used for querying Prometheus. Follow these instructions to expose the included Grafana service instance and access it through your web browser:

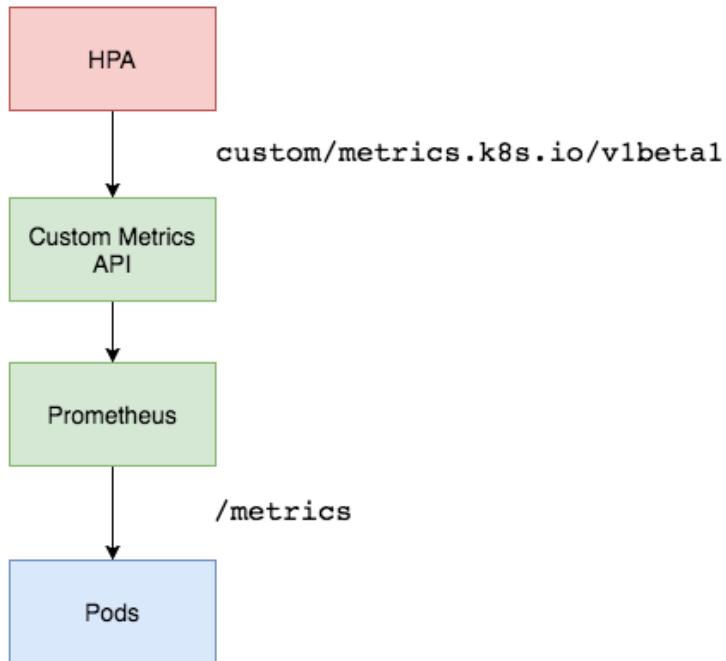


# Grafana Dashboard

Grafana is an open sourced monitor used to visualize the metrics stored on Prometheus. It offers dynamic and reusable dashboards with template variables. In this recipe, we will learn how to add a new dashboard from the library of pre-built dashboards to monitor an application deployed on Kubernetes.



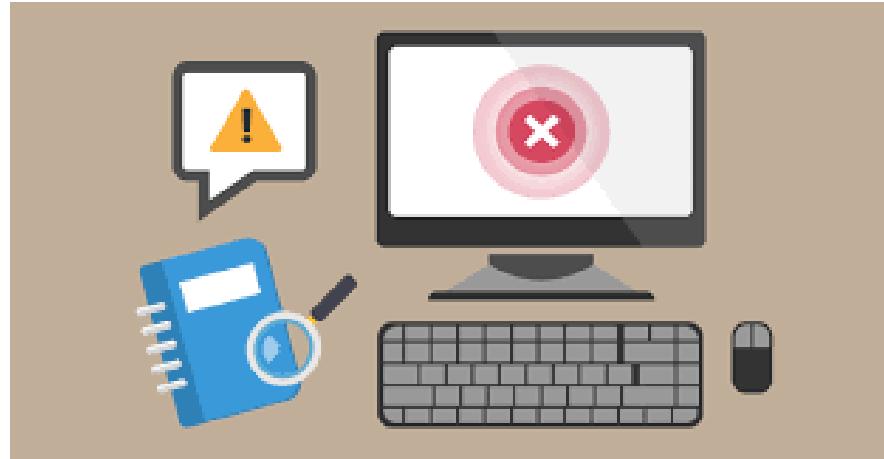
# Creating Custom Metrics to Trigger Cluster Autoscaling



# Logging

Logging is a key monitoring tool. Every self-respecting long-running software must have logs. Logs capture timestamped events. They are critical for many

applications, like business intelligence, secu



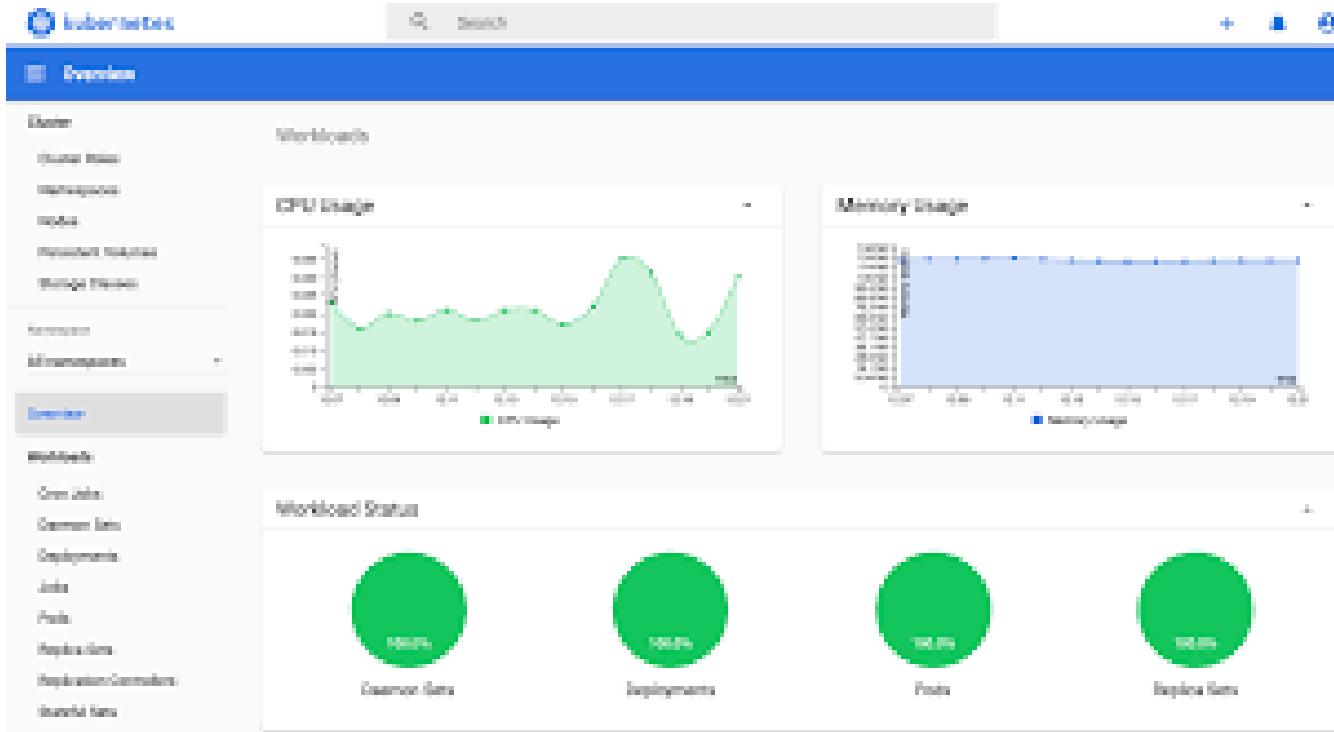
# Metrics

Metrics measure the same aspect of the system over time. Metrics are time series of numerical values (typically, floating-point numbers). Each metric has a name and often a set of labels that help later in slicing and dicing. For example, the CPU utilization of a node or the error rate of a service are metrics.



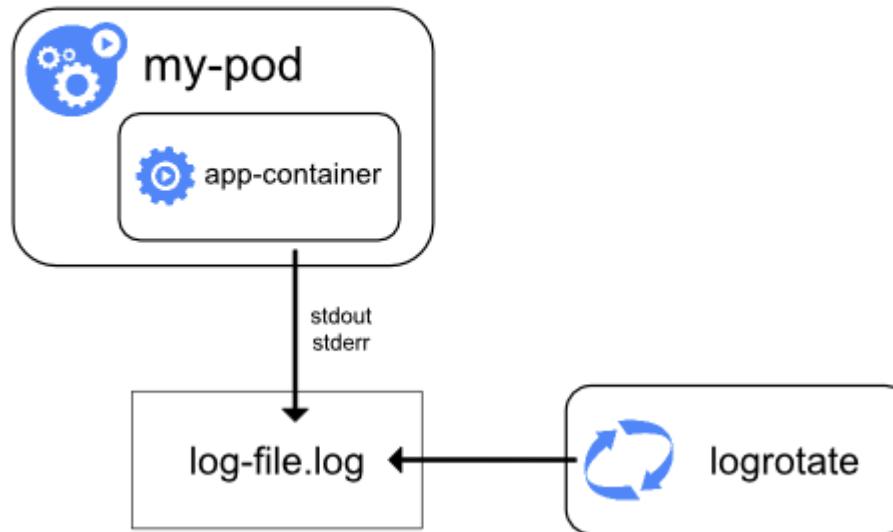
# Monitoring Metrics using Kubernetes Dashboard

Kubernetes Dashboard doesn't display detailed metrics unless Kubernetes Metrics Server is installed and the kubernetes-metrics-scraper sidecar container is running



# Logging Solution

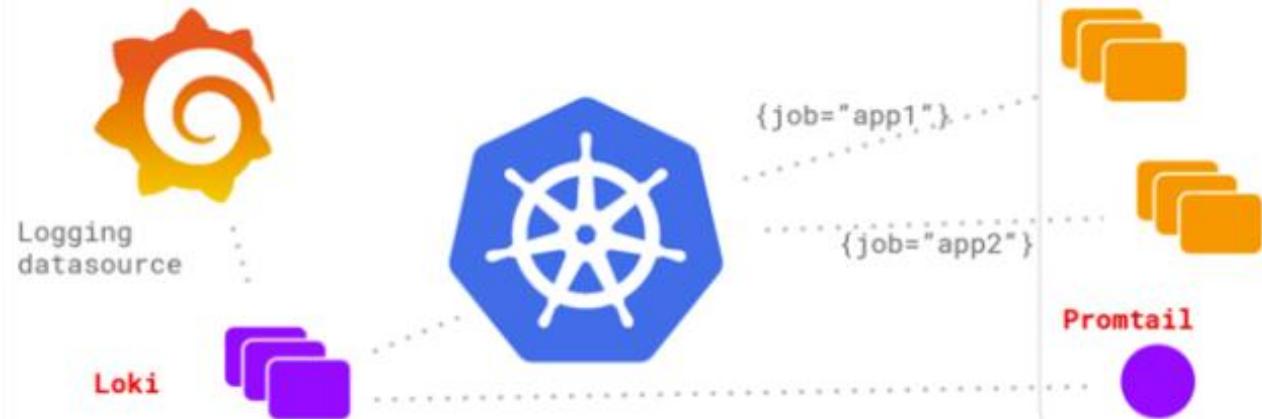
Logging at the node level



# Loki

Loki is a logging backend optimized for users running Prometheus and Kubernetes.

## Logging architecture



Loki was built for efficiency alongside the following goals:

- Logs should be cheap. Nobody should be asked to log less.
- Easy to operate and scale.
- Metrics, logs (and traces later) need to work together.

# Logging with Loki

2019-12-11T10:01:02.123456789Z {app="nginx", instance="1.1.1.1"} GET /about

**Timestamp**

with nanosecond precision

Prometheus-style **Labels**

key-value pairs

**Content**

log line

indexed

unindexed

# Selecting Log Streams

Selecting logstreams, is done by using **label matchers** and **filter expressions**

## Log stream

{app="nginx", instance="1.1.1.1"}

Query: {app="nginx"} start=T5 end=T7

{app="nginx", instance="2.2.2.2"}

## Chunks

T1-T5

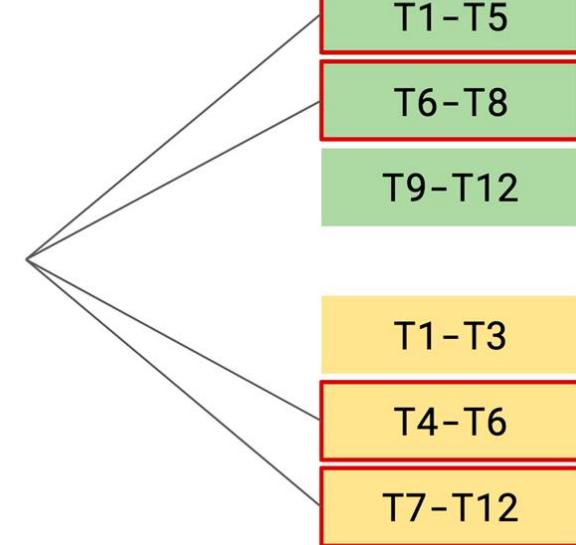
T6-T8

T9-T12

T1-T3

T4-T6

T7-T12



# POP QUIZ:

## Kubernetes: Controller, Operator Patterns, Creation



What are the three pillars of observability?

- A: Metrics, Traces, Logs
- B: Technique, Process, Metrics
- C: Efficiency, Metrics, Monitoring

# POP QUIZ:

## Kubernetes: Controller, Operator Patterns, Creation



What are the three pillars of observability?

- A: Metrics, Traces, Logs
- B: Technique, Process, Metrics
- C: Efficiency, Metrics, Monitoring



# POP QUIZ:

## Kubernetes: Controller, Operator Patterns, Creation



What is the difference between Grafana and Prometheus?

A: Grafana is an open-source visualization software, which helps the users to understand the complex data with the help of data metrics. Prometheus is an open-source event monitoring and alerting tool. It stores the majority of the data locally after scrapping metrics

B: Prometheus Is an open-source visualization software, which helps the users to understand the complex data with the help of data metrics. Grafana is an open-source event monitoring and alerting tool. It stores the majority of the data locally after scrapping metrics

# POP QUIZ:

## Kubernetes: Controller, Operator Patterns, Creation



What is the difference between Grafana and Prometheus?

A: Grafana is an open-source visualization software, which helps the users to understand the complex data with the help of data metrics. Prometheus is an open-source event monitoring and alerting tool. It stores the majority of the data locally after scrapping metrics

B: Prometheus is an open-source visualization software, which helps the users to understand the complex data with the help of data metrics. Grafana is an open-source event monitoring and alerting tool. It stores the majority of the data locally after scrapping metrics

# POP QUIZ:

## Kubernetes: Controller, Operator Patterns, Creation



What is the mission of Prometheus Operator?

- A: To make running prometheus as efficient as possible
- B: To make running prometheus as fast as possible
- C: To make running Prometheus on top of Kubernetes as easy as possible

# POP QUIZ:

## Kubernetes: Controller, Operator Patterns, Creation



What is the mission of Prometheus Operator?

- A: To make running prometheus as efficient as possible
- B: To make running prometheus as fast as possible

C: To make running Prometheus on top of Kubernetes as easy as possible

# Experiment – Prometheus



# Day in the Life of K8S API Request



**Let's take a look under the hood!**

```
$ kubectl create -f rs.yaml
```

## Overview

- kubectl
- conversion / serialization / defaulting
- apiserver
- storage (i.e. etcd)

## **Warning: code ahead**

- Code on these slides is minimalist!

# kubectl

- CLI setup code

```
func NewCmdCreate(f *cmdutil.Factory, out io.Writer) *cobra.Command {
    options := &CreateOptions{}
    cmd := &cobra.Command{
        Use:     "create -f FILENAME",
        Short:   "Create a resource by filename or
stdin",
        Long:    "create_long",
        Example: "create_example",
        Run: func(cmd *cobra.Command, args []string) {
            if len(options.Filenames) ==
                cmd.Help()
            return
        }

        cmdutil.CheckErr(ValidateArgs(cmd, args))

        cmdutil.CheckErr(cmdutil.ValidateOutputArgs(cmd))
        cmdutil.CheckErr(RunCreate(f,
            cmd, out, options))
    },
}
...

```

# kubectl

- Command execution time!
- Factory? Builder? Helper?

```
func RunCreate(f *cmdutil.Factory, cmd *cobra.Command, out io.Writer, ...) error {
    mapper, typer, err := f.UnstructuredObject()
    r := resource.NewBuilder(mapper, typer, ...).
        Schema(schema).
        ContinueOnError().
        NamespaceParam(cmdNamespace).DefaultNamespace().
        FilenameParam(enforceNamespace,
options.Recursive, options.Filenames...).
        Flatten().
        Do()
    err = r.Visit(func(info *resource.Info, err error) error {
        if err := createAndRefresh(info); err != nil {
            return
        }
        cmdutil.AddSourceToErr("creating", info.Source, err)
    })
}

// createAndRefresh creates an object from input info and refreshes that object
func createAndRefresh(info *resource.Info) error {
    obj, err := resource.NewHelper(info.Client, info.Mapping).
        Create(info.Namespace, true, info.Object)
    info.Refresh(obj, true)
}
```

# kubectl

## Factory - replaceable logic repository

```
// Factory provides abstractions that allow the Kubectl command to be extended across
// multiple types of resources and different API sets.
type Factory struct {
    // Returns interfaces for dealing with arbitrary runtime.Objects.
    Object func(thirdPartyDiscovery bool) (meta.RESTMapper,
        runtime.ObjectTyper)
    // Returns interfaces for dealing with arbitrary
    // runtime.Unstructured. This performs API calls to discover types.
    UnstructuredObject func() (meta.RESTMapper, runtime.ObjectTyper,
        error)
    // Returns interfaces for decoding objects - toInternal is
    deprecated
    Decoder func(toInternal bool) runtime.Decoder
    // Returns an encoder capable of encoding a provided object into
    JSON
    JSONEncoder func() runtime.Encoder
    // Returns a client for accessing Kubernetes resources or an error.
    Client func() (*client.Client, error)
    // Returns a client.Config for accessing the Kubernetes server.
    ClientConfig func() (*restclient.Config, error)
    // Returns a RESTClient for working with the specified RESTMapping
    ClientForMapping func(mapping *meta.RESTMapping)
    (resource.RESTClient, error)
    // Returns a RESTClient for working with Unstructured objects.
    UnstructuredClientForMapping func(mapping *meta.RESTMapping)
    (RESTClient, error)
    // Returns a Describer for displaying the specified RESTMapping
    type
    Describer func(mapping *meta.RESTMapping) (kubectl.Describer,
        error)
    // Returns a Printer for formatting objects of the given type
    Printer func(mapping *meta.RESTMapping, options
        kubectl.PrintOptions) (kubectl.ResourcePrinter, error)
    // Returns a Scaler for changing the size of the specified
    RESTMapping type
    Scaler func(mapping *meta.RESTMapping) (kubectl.Scaler, error)
    // Returns a Reaper for gracefully shutting down resources.
    Reaper func(mapping *meta.RESTMapping) (kubectl.Reaper, error)
    ...
}
```

# kubectl

- This is the thing that makes the Factory
- 600 lines of factory factory in one function!
- OpenShift has a customization

```
// NewFactory creates a factory with the default Kubernetes resources defined
func NewFactory(optionalClientConfig clientcmd.ClientConfig) *Factory {
    mapper := kubectl.ShortcutExpander{RESTMapper:
        registered.RESTMapper()}
    clients := NewClientCache(clientConfig)
    return &Factory{
        UnstructuredObject: func() (meta.RESTMapper,
            runtme.ObjectTyper, error) {
            cfg, err :=
        clients.ClientConfigForVersion(nil)
            dc, err :=
        discovery.NewDiscoveryClientForConfig(cfg)
            mapper :=
        discovery.NewRESTMapper(
            groupResources, meta.InterfacesForUnstructured)
            typer :=
        discovery.NewUnstructuredObjectTyper(groupResources)
            return
        kubectl.ShortcutExpander{RESTMapper: mapper}, typer, nil
    },
}
}
```

# kubectl

**Builder - builds the list of resources to operate on**

```
// Builder provides convenience functions for taking arguments and parameters
// from the command line and converting them to a list of resources to iterate
// over using the Visitor interface.
type Builder struct {
    mapper *Mapper
    ...
}

// NewBuilder creates a builder that operates on generic objects.
func NewBuilder(mapper meta.RESTMapper, typer runtime.ObjectTyper, clientMapper
ClientMapper, decoder runtime.Decoder) *Builder {
    return &Builder{
        mapper:      &Mapper{typer, mapper,
clientMapper, decoder},
        requireObject: true,
    }
}

// Do returns a Result object with a Visitor for resources identified by the Builder.
// The visitor will respect the error behavior specified by ContinueOnError.
func (b *Builder) Do() *Result {
    r := b.visitorResult()
    if b.flatten {
        r.visitor = NewFlattenListVisitor(r.visitor,
b.mapper)
    }
    helpers := []VisitorFunc{}
    if b.defaultNamespace { helpers = append(helpers,
SetNamespace(b.namespace)) }
    helpers = append(helpers, FilterNamespace)
    if b.requireObject { helpers = append(helpers, RetrieveLazy) }
    r.visitor = NewDecoratedVisitor(r.visitor, helpers...)
    return r
}
```

# kubectl

## Helper - Generic REST Now we're getting somewhere!

```
// Helper provides methods for retrieving or mutating a RESTful
// resource.
type Helper struct {
    // The name of this resource as the server would recognize it
    Resource string
    // An interface for reading or writing the resource version of this
    // type.
    Versioner runtime.ResourceVersioner
    // True if the resource type is scoped to namespaces
    NamespaceScoped bool
}

func (m *Helper) Create(namespace string, modify bool, obj runtime.Object)
    (runtime.Object, error) {
    if modify {
        // Attempt to version the object based on client
        logic.
            version, err := m.Versioner.ResourceVersion(obj)
    }
    return m.createResource(m.RESTClient, m.Resource, namespace, obj)
}

func (m *Helper) createResource(c RESTClient, resource, namespace string,
    obj runtime.Object) (runtime.Object, error) {
    return c.Post().NamespaceIfScoped(namespace, m.NamespaceScoped).
        Resource(resource).Body(obj).Do().Get()
}
```

# kubectl

- **RESTClient**
- **Now published separately in the client-go repository!**

```
// RESTClient imposes common Kubernetes API conventions on a set of resource paths.
// The baseURL is expected to point to an HTTP or HTTPS path that is the parent
// of one or more resources. The server should return a decodable API resource
// object, or an api.Status object which contains information about the reason for
// any failure.
//
// Most consumers should use client.New() to get a Kubernetes API client.
type RESTClient struct { ... }

// Post begins a POST request. Short for c.Verb("POST").
func (c *RESTClient) Post() *Request {
    return c.Verb("POST")
}

// Verb begins a request with a verb (GET, POST, PUT, DELETE).
//
// Example usage of RESTClient's request building interface:
// c, err := NewRESTClient(...)
// if err != nil { ... }
// resp, err := c.Verb("GET").
// Path("pods").
// SelectorParam("labels", "area=staging").
// Timeout(10*time.Second).
// Do()
// if err != nil { ... }
// list, ok := resp.(*api.PodList)
func (c *RESTClient) Verb(verb string) *Request {
    backoff := c.createBackoffMgr()
    return NewRequest(c.Client, verb, c.base, c.versionedAPIPath,
c.contentConfig, c.serializers, backoff, c.Throttle)
}
```

# kubectl

- RESTClient - Request builder
- Also in the client-go repository!

```
// Request allows for building up a request to a server in a chained fashion.  
// Any errors are stored until the end of your call, so you only have to  
// check once.  
type Request struct { ... }  
  
// Prefix adds segments to the relative beginning to the request path. These  
// items will be placed before the optional Namespace, Resource, or Name sections.  
// Setting AbsPath will clear any previously set Prefix segments  
func (r *Request) Prefix(segments ...string) *Request  
  
// Suffix appends segments to the end of the path. These items will be placed  
// after the prefix and optional Namespace, Resource, or Name sections.  
func (r *Request) Suffix(segments ...string) *Request  
  
// Resource sets the resource to access (<resource>/[ns/<namespace>/]<name>)  
func (r *Request) Resource(resource string) *Request  
  
// SubResource sets a sub-resource path which can be multiple segments segment  
// after the resource name but before the suffix.  
func (r *Request) SubResource(subresources ...string) *Request  
  
// Name sets the name of a resource to access (<resource>/[ns/<namespace>/]<name>)  
func (r *Request) Name(resourceName string) *Request  
  
// Namespace applies the namespace scope to a request  
// (<resource>/[ns/<namespace>/]<name>)  
func (r *Request) Namespace(namespace string) *Request  
  
// NamespaceIfScoped is a convenience function to set a namespace if scoped is true  
func (r *Request) NamespaceIfScoped(namespace string, scoped bool) *Request
```

# kubectl

- RESTClient - Request body builder
- Now published separately in the client-go repository!

```
// Body makes the request use obj as the body. Optional.  
// If obj is a string, try to read a file of that name.  
// If obj is a []byte, send it directly.  
// If obj is an io.Reader, use it directly.  
// If obj is a runtime.Object, marshal it correctly, and set Content-Type header.  
// If obj is a runtime.Object and nil, do nothing.  
// Otherwise, set an error.  
func (r *Request) Body(obj interface{}) *Request {  
    switch t := obj.(type) {  
        case string:  
            data, err := ioutil.ReadFile(t)  
            glog.V(8).Infof("Request Body: %#v",  
                string(data))  
            r.body = bytes.NewReader(data)  
        case []byte:  
            glog.V(8).Infof("Request Body: %#v", string(t))  
            r.body = bytes.NewReader(t)  
        case io.Reader:  
            r.body = t  
        case runtime.Object:  
            data, err :=  
                runtime.Encode(r.serializers.Encoder, t)  
                glog.V(8).Infof("Request Body: %#v",  
                    string(data))  
            r.body = bytes.NewReader(data)  
            r.SetHeader("Content-Type",  
                r.content.ContentType)  
        default:  
            r.err = fmt.Errorf("unknown type used for body:  
                %+v", obj)  
    }  
    return r  
}
```

# kubectl

- **RESTClient - Request execution**

```
// Do formats and executes the request. Returns a Result object for easy response
// processing.
//
// Error type:
//  * If the request can't be constructed, or an error happened while building its
//    arguments: *RequestConstructionError
//  * If the server responds with a status:
//    *errors.StatusError or
//    *errors.UnexpectedObjectError
//  * http.Client.Do errors are returned directly.
func (r *Request) Do() Result {
    r.tryThrottle()

    var result Result
    err := r.request(func(req *http.Request, resp *http.Response) {
        result = r.transformResponse(resp, req)
    })
    if err != nil {
        return Result{err: err}
    }
    return result
}
```

## Serialize, Convert, Default

- Auto-generated machinery to make your life “easier”
  - Problem: map memory structures to some canonical wire format
  - Problem: clients may be written against old API versions
  - Problem: clients may not understand new API fields

# Serialize

- Problem: map memory structures to some canonical wire format
- Solution: JSON, with proto option
  - nonstandard proto generation since we started with go structs and needed to maintain compatibility
  - Lots of generated code

# Serialize

- **API objects must conform to this interface!**

```
package runtime

// All API types registered with Scheme must support the Object interface. Since
// objects in a scheme are expected to be serialized to the wire, the interface an
// Object must provide to the Scheme allows serializers to set the kind, version, and
// group the object is represented as. An Object may choose to return a no-op
// ObjectKindAccessor in cases where it is not expected to be serialized.
type Object interface {
    GetObjectKind() unversioned.ObjectKind
}
```

# Serialize

- **Serialization Interfaces**

```
// Encoders write objects to a serialized form
type Encoder interface {
    // Encode writes an object to a stream.
    Encode(obj Object, w io.Writer) error
}

// Decoders attempt to load an object from data.
type Decoder interface {
    // Decode attempts to deserialize the provided data using either
    // the innate
    // typing of the scheme or the default kind, group, and version provided. It
    // returns a decoded object as well as the kind, group, and version from the
    // serialized data, or an error. If into is non-nil, it will be used as the
    // target type and implementations may choose to use it rather than
    // reallocating
    // an object. However, the object is not guaranteed to be populated. The
    // returned object is not guaranteed to match into. If defaults are provided,
    // they are applied to the data by default. If no defaults or partial defaults
    // are provided, the type of the into may be used to guide conversion
    // decisions.
    Decode(data []byte, defaults *unversioned.GroupVersionKind, into
Object)
        (Object, *unversioned.GroupVersionKind, error)
}

// Serializer is the core interface for transforming objects into a serialized
// format and back. Implementations may choose to perform conversion of the object,
// but no assumptions should be made.
type Serializer interface {
    Encoder
    Decoder
}
```

# Serialize

- **Serialization Interfaces II**
  - **this is how we abstract out JSON vs Proto serialization**

```
// NegotiatedSerializer is an interface used for obtaining encoders, decoders, and
// serializers for multiple supported media types. This would commonly be accepted by
// a
// server component that performs HTTP content negotiation to accept multiple formats.
type NegotiatedSerializer interface {
    // Media types supported for reading and writing single objects.
    SupportedMediaTypes() []string
    // SerializerForMediaType returns a serializer for the provided
    media type.
    SerializerForMediaType(mediaType string, params map[string]string)
        (s SerializerInfo, ok bool)

    // Media types of the supported streaming serializers.
    SupportedStreamingMediaTypes() []string
    // StreamingSerializerForMediaType returns a serializer for the
    provided media
    // type that supports reading and writing multiple objects to a stream.
    StreamingSerializerForMediaType(mediaType string, params
        map[string]string)
        (s StreamSerializerInfo, ok bool)

    // EncoderForVersion returns an encoder that ensures objects being
    written to
    // the provided serializer are in the provided group version.
    EncoderForVersion(serializer Encoder, gv GroupVersioner) Encoder
    // DecoderForVersion returns a decoder that ensures objects being
    read by the
    // provided serializer are in the provided group version by default.
    DecoderToVersion(serializer Decoder, gv GroupVersioner) Decoder
}
```

# Convert

- Problem: clients may be written against old API versions
- Solution: Keep all supported versions of the API in source tree.
  - auto-convert structures that haven't changed but
  - provide an escape hatch for humans to patch up items that have.

# Conversion

- **Serialization Interfaces**

```
// Codec is a Serializer that deals with the details of versioning objects.  
// It offers the same interface as Serializer, so this is a marker to consumers that  
// care about the version of the objects they receive.  
type Codec Serializer
```

## Default

- Problem: clients may not understand new API fields
- Solution: server does a defaulting pass, to fill omitted fields
  - the machinery calls defaulting methods if provided.

# apiserver

- ListenAndServeTLS

```
func (s *GenericAPIServer) Run(options *options.ServerRunOptions) {
    if s.enableSwaggerSupport { s.InstallSwaggerAPI() }
    if s.enableOpenAPISupport { s.InstallOpenAPI() }
    secureLocation = net.JoinHostPort(options.BindAddress,
options.SecurePort)
    sem := make(chan bool, options.MaxRequestsInFlight)
    longRunningTimeout := func(req *http.Request) (<-chan time.Time,
string) {
        // TODO unify this with
        apiserver.MaxInFlightLimit
    }
    if longRunningRequestCheck(req) { return nil, "" }
    return time.After(globalTimeout), ""
}

handler := apiserver.TimeoutHandler(
    apiserver.RecoverPanics(s.Handler), longRunningTimeout)

secureServer := &http.Server{
    Addr:      secureLocation,
    Handler:   apiserver.MaxInFlightLimit(sem, longRunningRequestCheck, handler),
}

glog.Infof("Serving securely on %s", secureLocation)
go func() {
    defer utilruntime.HandleCrash()
    for {
        if err :=
            secureServer.ListenAndServeTLS(...); err != nil {
            glog.Errorf("Unable to listen for secure (%v)", err)
        }
        time.Sleep(15 * time.Second)
    }()
}

select {}
```

# apiserver - routing

- **timeout / max in flight**
- **add context**
- **authenticator**
- **CORS**
- **Logs**
- **Dispatch (go mux)**
- **Dispatch (go-restful)**

```
// init initializes GenericAPIServer.  
func (s *GenericAPIServer) init(c *Config) {  
    // Register root handler.  
    if c.EnableIndex {  
        s.mux.HandleFunc("/",  
            apiserver.IndexHandler(s.HandlerContainer, s.MuxHelper))  
    }  
    if c.EnableLogsSupport {  
        apiserver.InstallLogsSupport(s.MuxHelper,  
            s.HandlerContainer)  
    }  
    if len(c.CorsAllowedOriginList) > 0 {  
        handler = apiserver.CORS(handler,  
            allowedOriginRegexp, nil, nil, "true")  
    }  
  
    // Install Authenticator  
    if c.Authenticator != nil {  
        authenticatedHandler, err :=  
            handlers.NewRequestAuthenticator(  
                s.RequestContextMapper, c.Authenticator, ..., handler)  
        handler = authenticatedHandler  
    }  
  
    // After all wrapping is done, put a context filter around both  
    var err error  
    handler, err = api.NewRequestContextFilter(s.RequestContextMapper,  
        s.Handler)  
    s.Handler = handler  
  
    s.installGroupsDiscoveryHandler()  
}
```

# apiserver - dispatch

- We have to program go-restful
- Our API is divided by group
- This is the group definition consumed by the api installer

```
// APIGroupVersion is a helper for exposing rest.Storage objects as http.Handlers
// via go-restful. It handles URLs of the form:
// ${storage_key}/${object_name}
// Where 'storage_key' points to a rest.Storage object stored in storage. This object
// should contain all parameterization necessary for running a particular API version
type APIGroupVersion struct {
    Storage map[string]rest.Storage

    // GroupVersion is the external group version
    GroupVersion unversioned.GroupVersion
    // Serializer is used to determine how to convert responses from
    API methods
    // into bytes to send over the wire.
    Serializer     runtime.NegotiatedSerializer

    Typer      runtime.ObjectTyper
    Creator    runtime.ObjectCreator
    Convertor  runtime.ObjectConvertor
    Copier     runtime.ObjectCopier
    Linker    runtime.ObjectLinker

    // SubresourceGroupVersionKind contains the GroupVersionKind
    overrides for each
    // subresource that is accessible from this API group version.
    SubresourceGroupVersionKind map[string]unversioned.GroupVersionKind

    // ResourceLister is an interface that knows how to list resources
    // for this API Group.
    ResourceLister APIResourceLister
}
```

# apiserver - dispatch II

- We have to program go-restful
- The APIInstaller sets up the Swagger (& OpenAPI) handler as it installs entry points into the go-restful mux.

```
type APIInstaller struct {
    group          *APIGroupVersion
    info           *RequestInfoResolver
}

// Installs handlers for API resources.
func (a *APIInstaller) Install(ws *restful.WebService)
    (apiResources []unversioned.APIResource, errors []error) {
    // Register paths in a deterministic order to get a deterministic
    swagger spec.
    for _, path := range paths {
        apiResource, err := a.registerResourceHandlers(
            path, a.group.Storage[path], ws, proxyHandler)
        apiResources = append(apiResources,
            *apiResource)
    }
    return apiResources, errors
}
```

# apiserver - dispatch III

- We have to program go-restful
- The APIInstaller is not pretty :(
- Here's just the bits that set up POST handlers

```
func (a *APIInstaller) registerResourceHandlers(path string, storage rest.Storage, ws
*restful.WebService, proxyHandler http.Handler) (*unversioned.APIResource, error) {
    creater, isCreater := storage.(rest.Creater)

    // Get the list of actions for the given scope.
    switch scope.Name() {
    case meta.RESTScopeNameNamespace:
        actions = appendIf(actions,
            action{"POST", resourcePath, resourceParams, namer}, isCreater)
    }

    for _, action := range actions {
        // case "POST": Create a resource.
        handler := CreateResource(creater, reqScope,
a.group.Typer, admit)
        route := ws.POST(action.Path).To(handler).
Doc(doc).

        Param(ws.QueryParameter("pretty", "If 'true', then the output is
pretty printed.")).

        Operation("create"+namespaced+kind+strings.Title(subresource)).

        Produces(append(storageMeta.ProducesMIMETypes(action.Verb),
a.group.Serializer SupportedMediaTypes()....)).
        Returns(http.StatusOK, "OK",

versionedObject).

        Reads(versionedObject).
        Writes(versionedObject)
        addParams(route,
ws.Route(route)

    }
    return &apiResource, nil
}
```

# apiserver - Handlers

- Finally we get to something that does something!
  - negotiation
  - read body
  - decode / convert
  - call to registry
  - set self link
  - return to user

```
// CreateResource returns a function that will handle a resource creation.
func CreateResource(r rest.Creater, scope RequestScope, typer runtime.ObjectTyper,
admit admission.Interface) restful.RouteFunction {
    return createHandler(&namedCreaterAdapter{r}, scope, typer, admit,
false)
}

func createHandler(r rest.NamedCreater, scope RequestScope, typer runtime.ObjectTyper,
admit admission.Interface, includeName bool) restful.RouteFunction {
    return func(req *restful.Request, res *restful.Response) {
        namespace, err = scope.Namer.Namespace(req)
        ctx := scope.ContextFunc(req); ctx =
api.WithNamespace(ctx, namespace)
        gv := scope.Kind.GroupVersion()
        s, err := negotiateInputSerializer(req.Request,
scope.Serializer)
        decoder := scope.Serializer.DecoderToVersion(s,
unversioned.GroupVersion{Group: gv.Group, Version: runtime.APIVersionInternal})
        body, err := readBody(req.Request)

        obj, gvk, err := decoder.Decode(body,
&defaultGVK, original)
        if admit != nil &&
admit.Handles(admission.Create) {

            admit.Admit(admission.NewAttributesRecord(obj, nil, scope.Kind,
namespace, name, scope.Resource, scope.Subresource, admission.Create, userInfo))
        }
        result, err := finishRequest(timeout, func() {
            (runtime.Object, error) {
                name, obj)
                out, err := r.Create(ctx,
                    return out, err
            })
            setSelfLink(result, req, scope.Namer)
            write(http.StatusCreated,
scope.Kind.GroupVersion(), scope.Serializer, result, w, req.Request)
        }
    }
}
```

# apiserver - Interfaces

- REST interfaces
- ~ 18 different interfaces allow resources to implement any subset of functionality

```
// Storage is a generic interface for RESTful storage services. Resources which are
// exported to the RESTful API of apiserver need to implement this interface. It is
// expected that objects may implement any of the below interfaces.
type Storage interface {
    // New returns an empty object that can be used with Create and
    // Update after
    // request data has been put into it. This object must be a pointer type for
    // use
    // with Codec.DecodeInto([]byte, runtime.Object)
    New() runtime.Object
}

// Creater is an object that can create an instance of a RESTful object.
type Creater interface {
    // New returns an empty object that can be used with Create after
    // request data
    // has been put into it. This object must be a pointer type for use with
    // Codec.DecodeInto([]byte, runtime.Object)
    New() runtime.Object

    // Create creates a new version of a resource.
    Create(ctx api.Context, obj runtime.Object) (runtime.Object, error)
}
```

# apiserver - Store

- **Most of the implementation work is done for you!**

```
// Store implements generic.Registry. It's embeddable, so you can implement any
// non-generic functions if needed.
//
// The intended use of this type is embedding within a Kind specific
// RESTStorage implementation. This type provides CRUD semantics on
// a Kube-like resource, handling details like conflict detection with
// ResourceVersion and semantics. The RESTCreateStrategy and
// RESTUpdateStrategy are generic across all backends, and encapsulate
// logic specific to the API.
type Store struct {
    // Called to make a new object, should return e.g., &api.Pod{}
    NewFunc func() runtime.Object
    // Allows extended behavior during creation, required
    CreateStrategy rest.RESTCreateStrategy
    // On create of an object, attempt to run a further operation.
    AfterCreate rest.ObjectFunc

    // Used for all storage access functions
    Storage storage.Interface
}
```

# apiserver - Store II

- **Most of the implementation work is done for you!**

```
// Create inserts a new item according to the unique key from the object.
func (e *Store) Create(ctx api.Context, obj runtime.Object) (runtime.Object, error) {
    rest.BeforeCreate(e.CreateStrategy, ctx, obj)
    name, err := e.ObjectNameFunc(obj)
    key, err := e.KeyFunc(ctx, name)
    ttl, err := e.calculateTTL(obj, 0, false)
    out := e.NewFunc()
    if err := e.Storage.Create(ctx, key, obj, out, ttl); err != nil {
        err = storeerr.InterpretCreateError(err,
            e.QualifiedResource, name)
        err =
    }
    rest.CheckGeneratedNameError(e.CreateStrategy, err, obj)
    if !kubeerr.IsAlreadyExists(err) { return nil, err }
    err }

    errGet := e.Storage.Get(ctx, key, out, false)
    accessor, errGetAcc := meta.Accessor(out)
    if accessor.GetDeletionTimestamp() != nil {
        *msg = fmt.Sprintf("object is
being deleted: %s", *msg)
    }
    return nil, err
}
if e.AfterCreate != nil {
    if err := e.AfterCreate(out); err != nil {
        return nil, err
    }
}
if e.Decorator != nil {
    if err := e.Decorator(obj); err != nil {
        return nil, err
    }
}
return out, nil
}
```

# apiserver - RS strategy

- Actual RS strategy is trivial!

```
// rsStrategy implements verification logic for ReplicaSets.  
type rsStrategy struct { ... }  
  
// PrepareForCreate clears the status of a ReplicaSet before creation.  
func (rsStrategy) PrepareForCreate(ctx api.Context, obj runtime.Object) {  
    rs := obj.(*extensions.ReplicaSet)  
    rs.Status = extensions.ReplicaSetStatus{  
        Generation: 1  
    }  
  
    // Validate validates a new ReplicaSet.  
    func (rsStrategy) Validate(ctx api.Context, obj runtime.Object) field.ErrorList {  
        rs := obj.(*extensions.ReplicaSet)  
        return validation.ValidateReplicaSet(rs)  
    }  
}
```

# apiserver - Storage

- All storage operations done through this go interface

```
package storage

// Interface offers a common interface for object marshaling/unmarshaling operations
// and hides all the storage-related operations behind it.
type Interface interface {

    // Create adds a new object at a key unless it already exists.
    Create(ctx context.Context, key string, obj, out runtime.Object,
           ttl uint64) error

    // Delete removes the 'key' and returns the value that existed at
    // that spot.
    Delete(ctx context.Context, key string, out runtime.Object,
            preconditions *Preconditions) error

    // Watch begins watching the specified key.
    Watch(ctx context.Context, key string, resourceVersion string,
           p SelectionPredicate) (watch.Interface, error)

    // WatchList begins watching the specified key's items.
    WatchList(ctx context.Context, key string, resourceVersion string,
               p SelectionPredicate) (watch.Interface, error)

    // Get unmarshals json found at key into objPtr.
    Get(ctx context.Context, key string, objPtr runtime.Object,
         ignoreNotFound bool) error

    // List unmarshals jsons found at directory defined by key.
    List(ctx context.Context, key string, resourceVersion string,
          p SelectionPredicate, listObj runtime.Object) error
}
```

# apiserver - Storage

- **This is the etcd3 implementation**
- **etcd v2 implementation is also in the source tree**
- **We are open to more storage implementations**
- **...but there is a high bar here due to the critical nature of this area**

```
// New returns an etcd3 implementation of storage.Interface.  
func New(c *clientv3.Client, codec runtime.Codec, prefix string) storage.Interface {  
    return newStore(c, codec, prefix)  
}  
  
func newStore(c *clientv3.Client, codec runtime.Codec, prefix string) *store {  
    versioner := etcd.APIObjectVersioner{}  
    return &store{  
        client:      c,  
        versioner:   versioner,  
        codec:       codec,  
        pathPrefix:  prefix,  
        watcher:    newWatcher(c, codec, versioner),  
    }  
}
```

# apiserver - Storage

- This is the etcd3 implementation of Create

```
/ Create implements storage.Interface.Create.
func (s *store) Create(ctx context.Context, key string, obj, out runtime.Object,
                      ttl uint64) error {
    version, err := s.versioner.ObjectResourceVersion(obj)
    data, err := runtime.Encode(s.codec, obj)
    key = keyWithPrefix(s.pathPrefix, key)

    opts, err := s.ttlOpts(ctx, int64(ttl))

    txnResp, err := s.client.KV.Txn(ctx).If(
        notFound(key),
    ).Then(
        clientv3.OpPut(key, string(data), opts...),
    ).Commit()

    if !txnResp.Succeeded {
        return storage.NewKeyExistsError(key, 0)
    }

    if out != nil {
        putResp := txnResp.Responses[0].GetResponsePut()
        return decode(s.codec, s.versioner, data, out,
                      putResp.Header.Revision)
    }
    return nil
}
```

# apiserver - Storage

- **Command line options determine storage settings**
- **This is the factory that constructs the storage interface**

```
// DefaultStorageFactory takes a GroupResource and returns back its storage interface.  
// This result includes:  
// 1. Merged etcd config, including: auth, server locations, prefixes  
// 2. Resource encodings for storage: group,version,kind to store as  
// 3. Cohabitating default: some resources like hpa are exposed through multiple APIs.  
// They must agree on 1 and 2  
type DefaultStorageFactory struct {  
    // StorageConfig describes how to create a storage backend in  
    general.  
    StorageConfig storagebackend.Config  
  
    Overrides map[unversioned.GroupResource]groupResourceOverrides  
  
    // DefaultMediaType is the media type used to store resources.  
    DefaultMediaType string  
  
    // DefaultSerializer is used to create encoders and decoders for  
    the  
    // storage.Interface.  
    DefaultSerializer runtime.StorageSerializer  
  
    // ResourceEncodingConfig describes how to encode a particular  
    // GroupVersionResource  
    ResourceEncodingConfig ResourceEncodingConfig  
  
    // APIResourceConfigSource indicates whether the *storage* is  
    enabled, NOT  
    // the API. This is discrete from resource enablement because those are separate  
    // concerns. How this source is configured is left to the caller.  
    APIResourceConfigSource APIResourceConfigSource  
}
```

# apiserver - Assembly

- The “glue” that sets up all REST resources with their backing storage

```
func (m *Master) InstallAPIs(c *Config) {
    if c.APIResourceConfigSource.AnyResourcesForVersionEnabled("v1") {
        // Install v1 API.
        m.initV1ResourcesStorage(c)
        apiGroupInfo := genericapiserver.APIGroupInfo{
            GroupMeta: VersionedResourcesStorageMap:
                "v1": {
                    IsLegacyGroup: true,
                    Scheme: api.Scheme,
                    ParameterCodec: api.ParameterCodec,
                    NegotiatedSerializer: api.Codecs,
                    SubresourceGroupVersionKind: map[string]GroupVersionKind{},
                }
        }
        apiGroupsInfo = append(apiGroupsInfo,
        apiGroupInfo)
    }

    for _, group := range
sets.StringKeySet(c.RESTStorageProviders).List() {
        if
!c.APIResourceConfigSource.AnyResourcesForGroupEnabled(group) {
            continue
        }
        restStorageBuilder :=
            apiGroupInfo, enabled :=
                c.APIResourceConfigSource, restOptionsGetter)
        glog.V(1).Infof("Enabling API group %q.", group)
        apiGroupsInfo = append(apiGroupsInfo,
        apiGroupInfo)
    }
    if err := m.InstallAPIGroups(apiGroupsInfo); err != nil {
        glog.Fatalf("Error in registering group
versions: %v", err)
    }
}
```

# apiserver - Assembly

- RS setup code!

```
func (p ExtensionsRESTStorageProvider) v1beta1Storage(apiResourceConfigSource
genericapiserver.APIResourceConfigSource, restOptionsGetter RESTOptionsGetter)
map[string]rest.Storage {
    version := extensionsapiv1beta1.SchemeGroupVersion
    storage := map[string]rest.Storage{}

    if apiResourceConfigSource.ResourceEnabled(
        version.WithResource("replicasets")) {
        replicaSetStorage := replicasetetcd.NewStorage(
            restOptionsGetter(extensions.Resource("replicasets")))
        storage["replicasets"] =
            replicaSetStorage.ReplicaSet
        storage["replicasets/status"] =
            replicaSetStorage.Status
        storage["replicasets/scale"] =
            replicaSetStorage.Scale
    }

    return storage
}
```

# apiserver - Assembly

- RS setup code -- continued

```
func NewStorage(opts generic.RESTOptions) ReplicaSetStorage {
    replicaSetRest, replicasetStatusRest := NewREST(opts)
    replicaSetRegistry := replicaset.NewRegistry(replicaSetRest)

    return ReplicaSetStorage{
        ReplicaSet: replicaSetRest,
        Status:     replicasetStatusRest,
        Scale:      &ScaleREST{registry:
    replicaSetRegistry},
    }
}

// NewREST returns a RESTStorage object that will work against ReplicaSet.
func NewREST(opts generic.RESTOptions) (*REST, *StatusREST) {
    prefix := "/" + opts.ResourcePrefix

    newListFunc := func() runtime.Object { return
    &extensions.ReplicaSetList{} }
    storageInterface, dFunc := opts.Decorator(
        opts.StorageConfig,

        cachesize.GetWatchCacheSizeByResource(cachesize.Replicaset),
        &extensions.ReplicaSet{},
        prefix,
        replicaset.Strategy,
        newListFunc,
        storage.NoTriggerPublisher,
    )

    store := &registry.Store{
        NewFunc: func() runtime.Object { return
    &extensions.ReplicaSet{} },
        ...
    }
```

# apiserver - Assembly

- RS setup code-- final

```
...
NewListFunc: newListFunc,
    // Produces a path that etcd understands, to the
root of the resource
    // by combining the namespace in the context
with the given prefix
KeyRootFunc: func(ctx api.Context) string {
    return
}
registry.NamespaceKeyRootFunc(ctx, prefix)
    },
    // Produces a path that etcd understands, to the
resource by combining
    // the namespace in the context with the given
prefix
KeyFunc: func(ctx api.Context, name string)
    return
}
registry.NamespaceKeyFunc(ctx, prefix, name)
    },
    // Retrieve the name field of a ReplicaSet
ObjectNameFunc: func(obj runtime.Object)
    return
}
obj.(*extensions.ReplicaSet).Name, nil
    },
    // Used to match objects based on labels/fields
for list and watch
PredicateFunc:
replicaset.MatchReplicaSet,
QualifiedResource:
api.Resource("replicasets"),
DeleteCollectionWorkers:
opts.DeleteCollectionWorkers,
CreateStrategy: replicaset.Strategy,
UpdateStrategy: replicaset.Strategy,
DeleteStrategy: replicaset.Strategy,
Storage: storageInterface,
DestroyFunc: dFunc,
}
statusStore := *store
statusStore.UpdateStrategy = replicaset.StatusStrategy
return &REST{store}, &StatusREST{store: &statusStore}
}
```