Container Networking

Gaetano Borgione
Sr. Staff Engineer @ VMware

Gaetano Borgione
Senior Staff Engineer
Cloud Native Applications
VMWare

SDN Technologies @ PLUMgrid
Data Center Networking @ Cisco

Passionate Engineer with special interests on:

Networking Architecture
Engineering Leadership
Product Management
Customer Advocacy
+
...new Networking / Virtualization ideas !!!

Agenda

Agenda

- Containers, Microservices
- Container Interfaces, Network Connectivity
- Service Discovery, Load Balancing
- Multi-Tenancy, Container Isolation, Micro-Segmentation
- On-Premise Private Cloud design

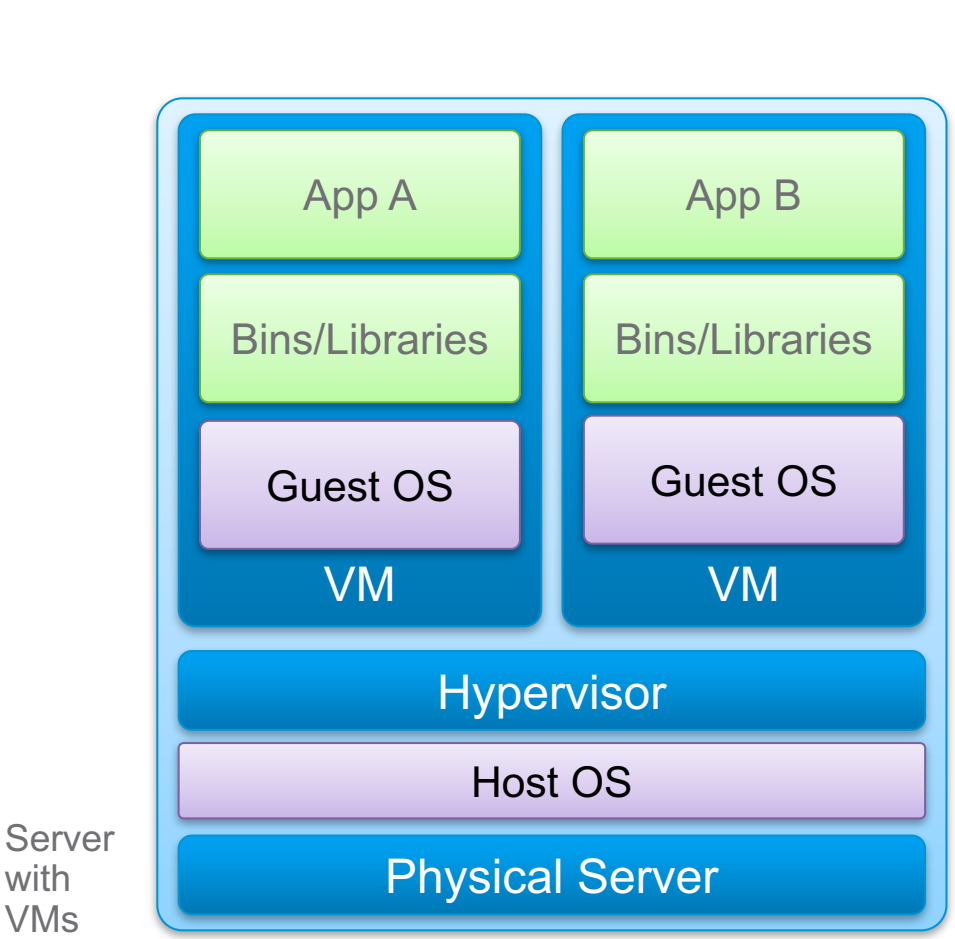
Containers && Microservices

Containers

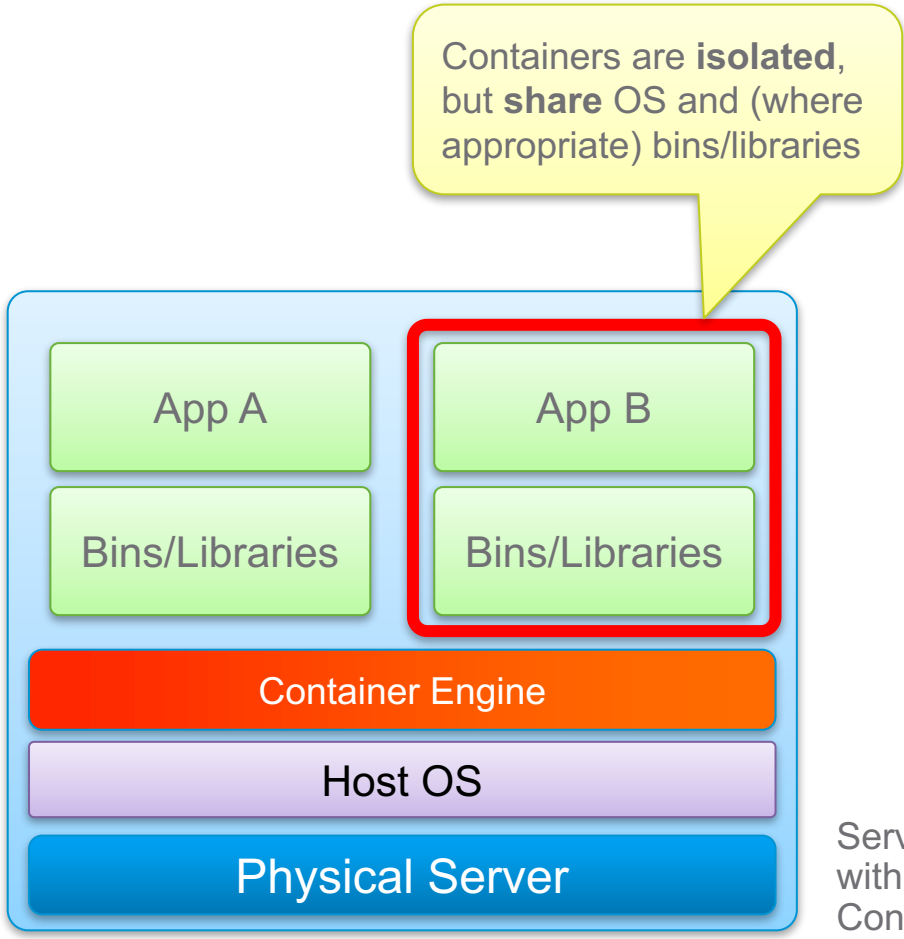
- A container image is a lightweight, stand-alone, executable unit of software
- Includes everything needed to run it: code, runtime, system tools, system libraries, settings
- Containerized software run regardless of the environment (i.e. Host OS distro)
- Containers isolate software from its surroundings
  - "smooth out" differences between development and staging environments
- Help reduce conflicts between teams running different software on the same infrastructure

Containers “at-a-glance”

Abstraction at the OS layer rather than hardware layer

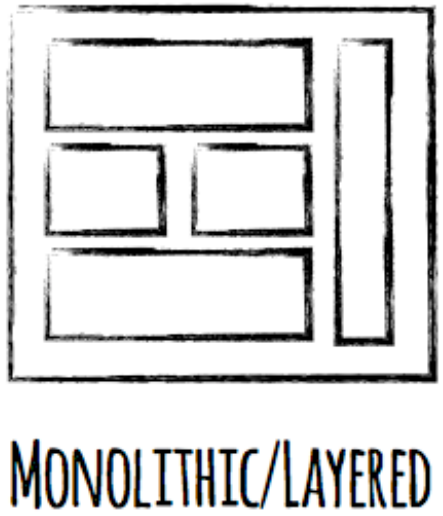


Server with VMs



Server with Containers

Microservices: Application Design is changing !!!



Properties of a Microservice

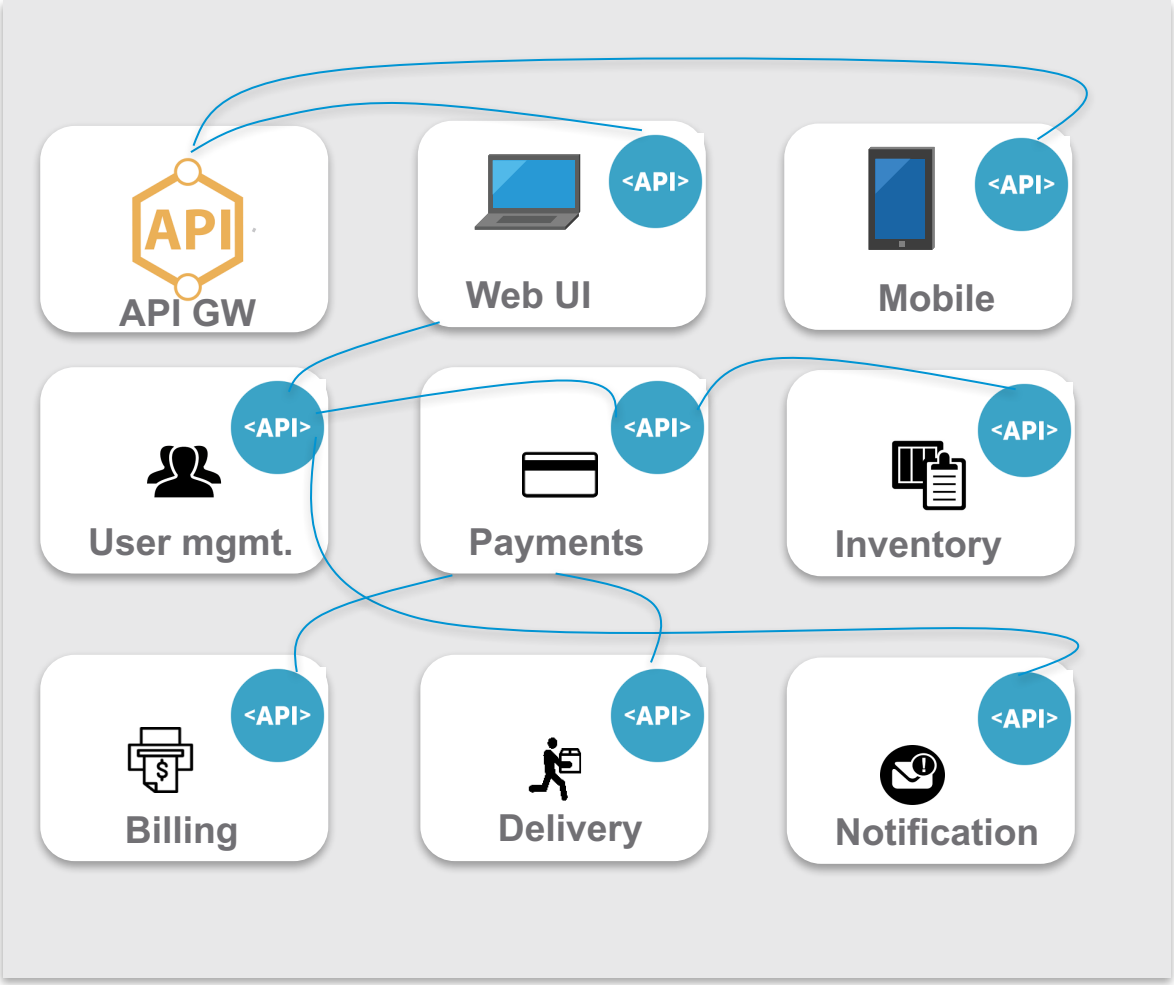
- ✓ Small code base
- ✓ Easy to scale, deploy and throw away
- ✓ Autonomous
- ✓ Resilient

Benefits of a Microservices Architecture

- ✓ A highly resilient, scalable and resource efficient application
- ✓ Enables smaller development teams
- ✓ Teams free to use the right languages and tools for the job
- ✓ Rapid application development

Cloud Native Application

Applications built using the “Microservices” architecture pattern



- **Loosely coupled distributed application**
  Application tier is decomposed into multiple web services
- **Datastore**
  Each micro service typically has its own datastore
- **Packaging**
  Each microservice is typically packaged in a “Container” image
- **Teams**
  Typically a team owns one or more Microservices

More on Microservices….

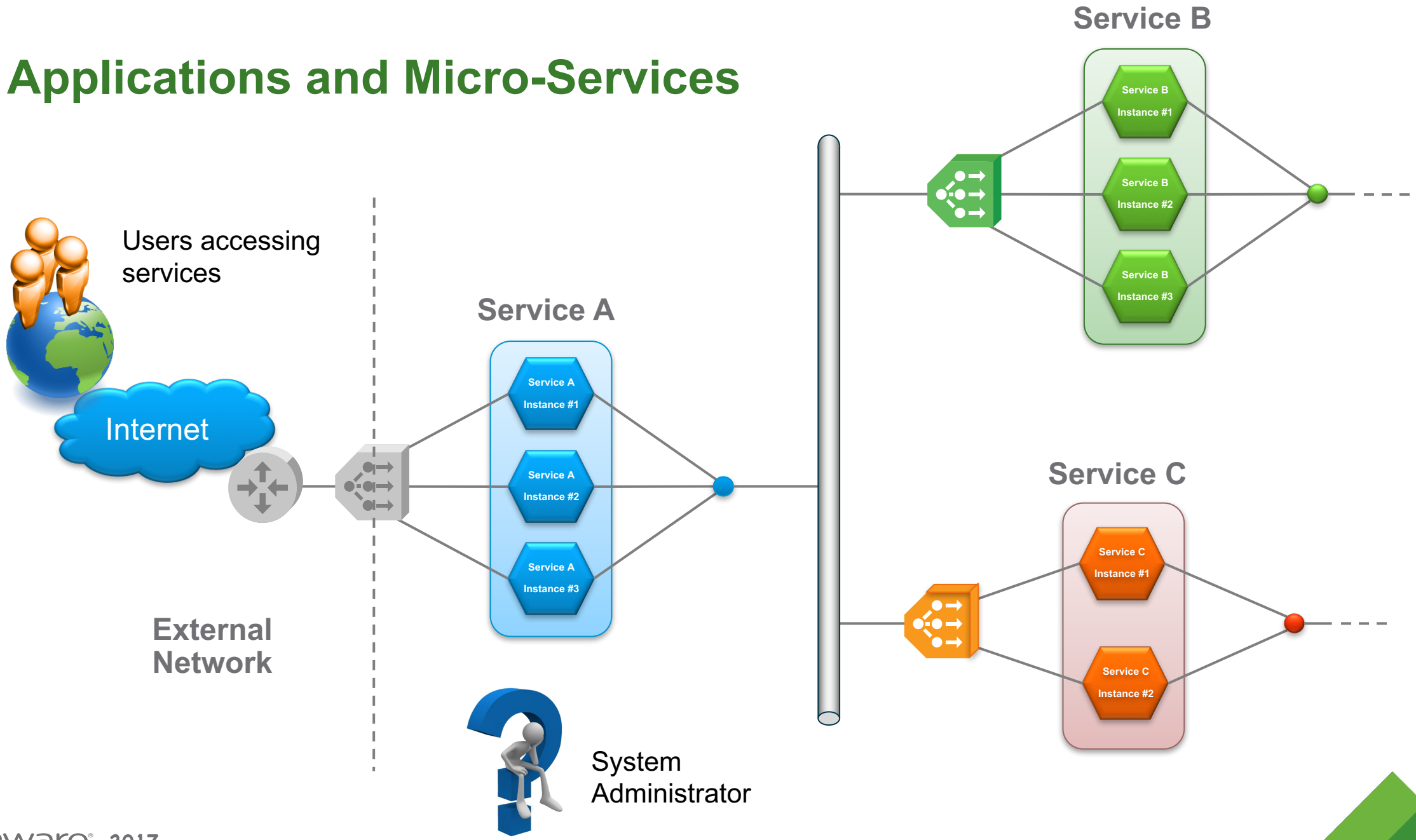
- Microservices != Containers
- The idea behind Microservices is to separate functionality into small parts that are created independently, by different teams, and possibly even in very different languages
- Microservices communicate with each other using language-agnostic APIs (e.g. REST)
- The host for each Microservice could be a VM, but containers are seen are ideal packaging unit to deploy a Microservice => low footprint

https://upload.wikimedia.org/wikipedia/commons/9/9b/Social_Network_Analysis_Visualization.png

Challenges of running Microservices…

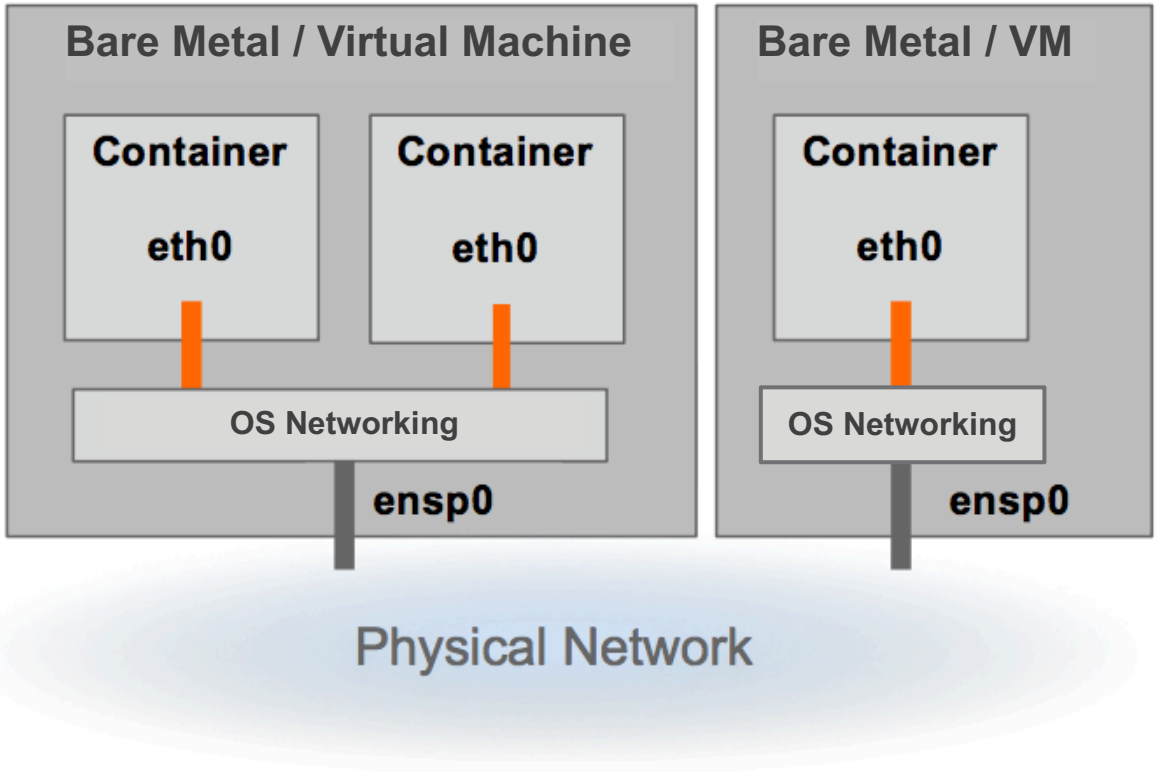
- Service Discovery
- Operational Overhead (100s+ of Services !!!)
- Distributed System... inherently complex
- Service Dependencies
  - service fan-out
  - dependency services running “hot”
- Traffic / Load each service can handle
- Service Health / Fault Tolerance
- Auto-Scale

Applications and Micro-Services



Container Interfaces && Network Connectivity

Basics of Container Networking

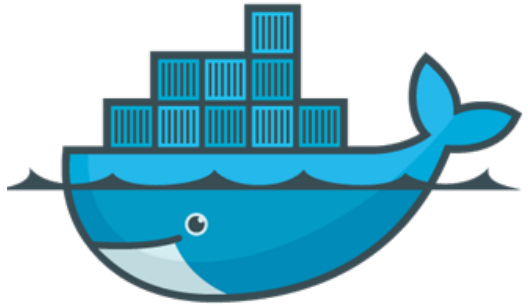


Minimalist Networking requirements:

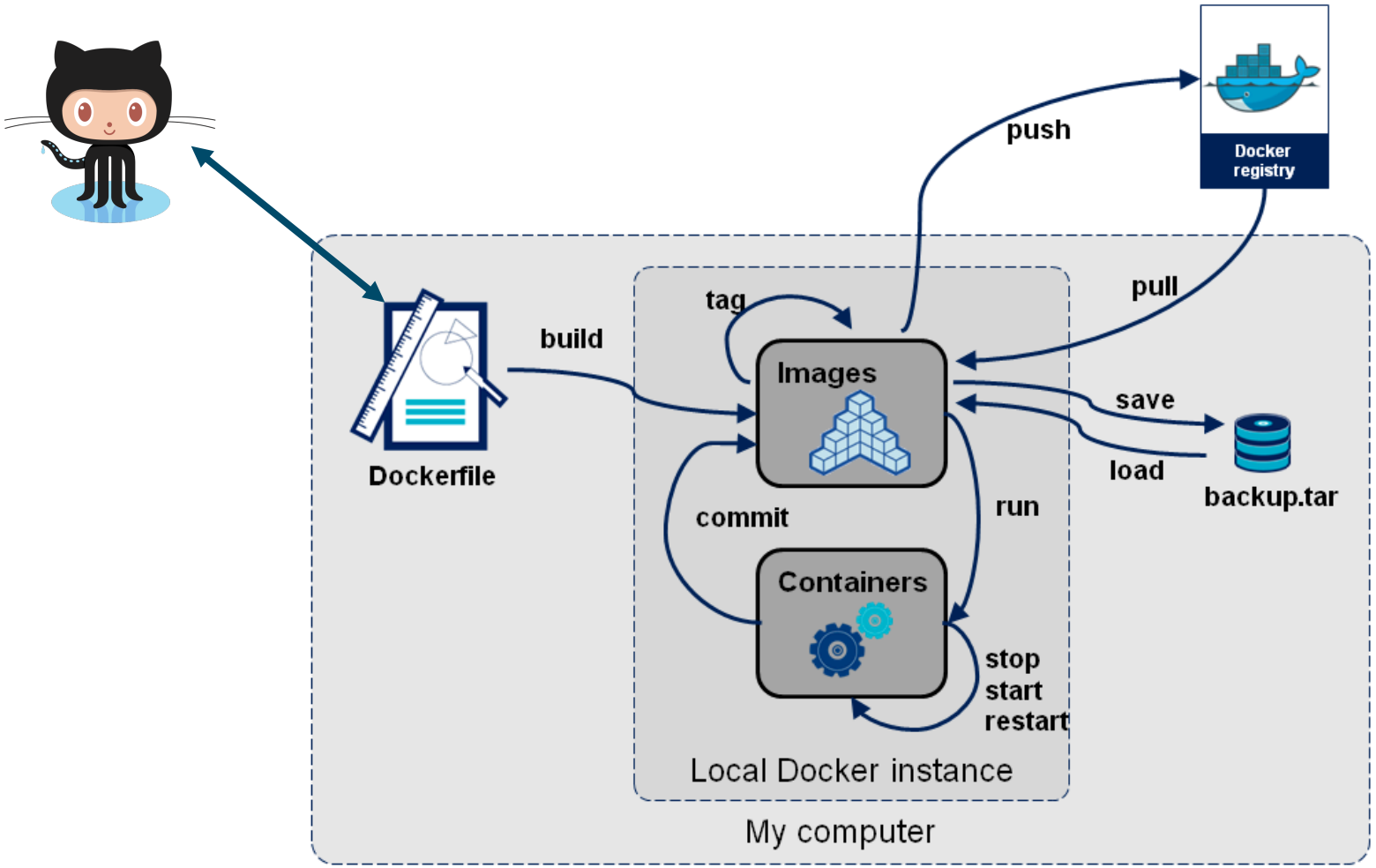
- IP Connectivity in Container's Network
- IP Address Management (IPAM) and Network Device Creation
- External Connectivity via Host NAT or Route Advertisement

Container Interfaces && Network Connectivity

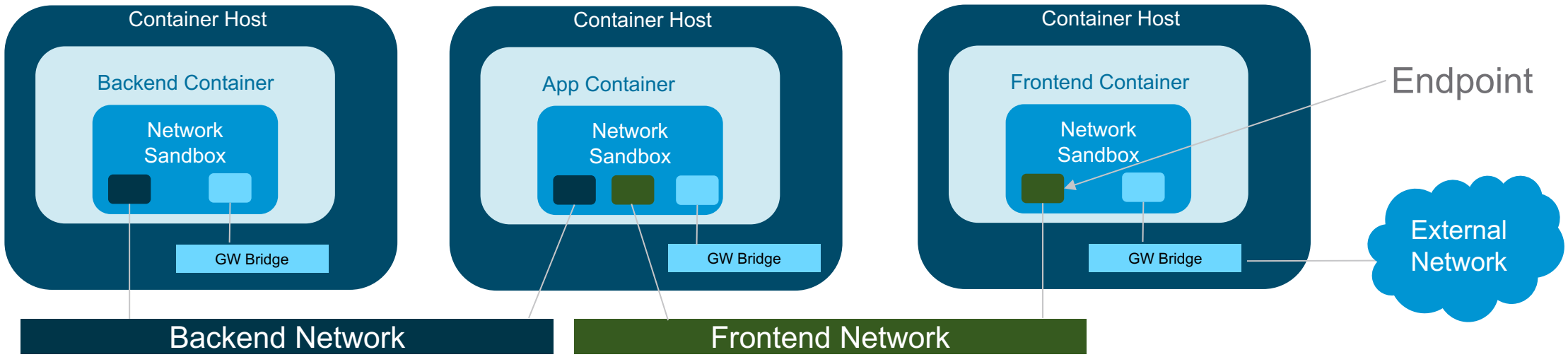
Docker



Docker is a “Shipping Container” for Code



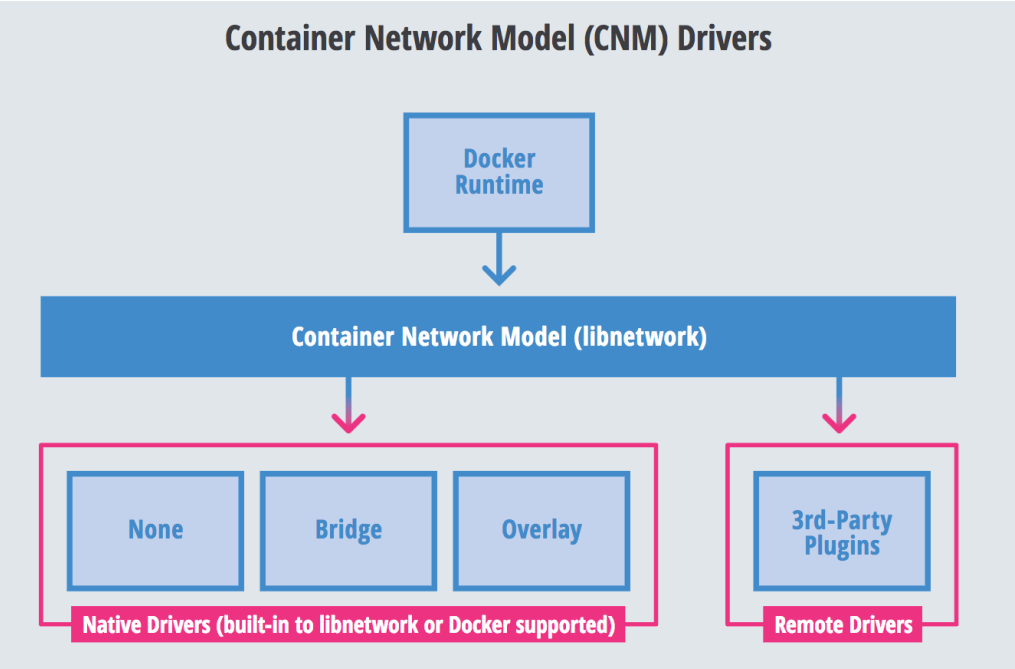
Docker: The Container Network Model (CNM) Interfacing



- **Sandbox**
  - A Sandbox contains the configuration of a container's network stack. This includes management of the container's interfaces, routing table and DNS settings. An implementation of a Sandbox could be a Linux Network Namespace, a FreeBSD Jail or other similar concept.
- **Endpoint**
  - An Endpoint joins a Sandbox to a Network. An implementation of an Endpoint could be a veth pair, an Open vSwitch internal port or similar
- **Network**
  - A Network is a group of Endpoints that are able to communicate with each-other directly. An implementation of a Network could be a VXLAN Segment, a Linux bridge, a VLAN, etc.

Container Network Model (CNM)

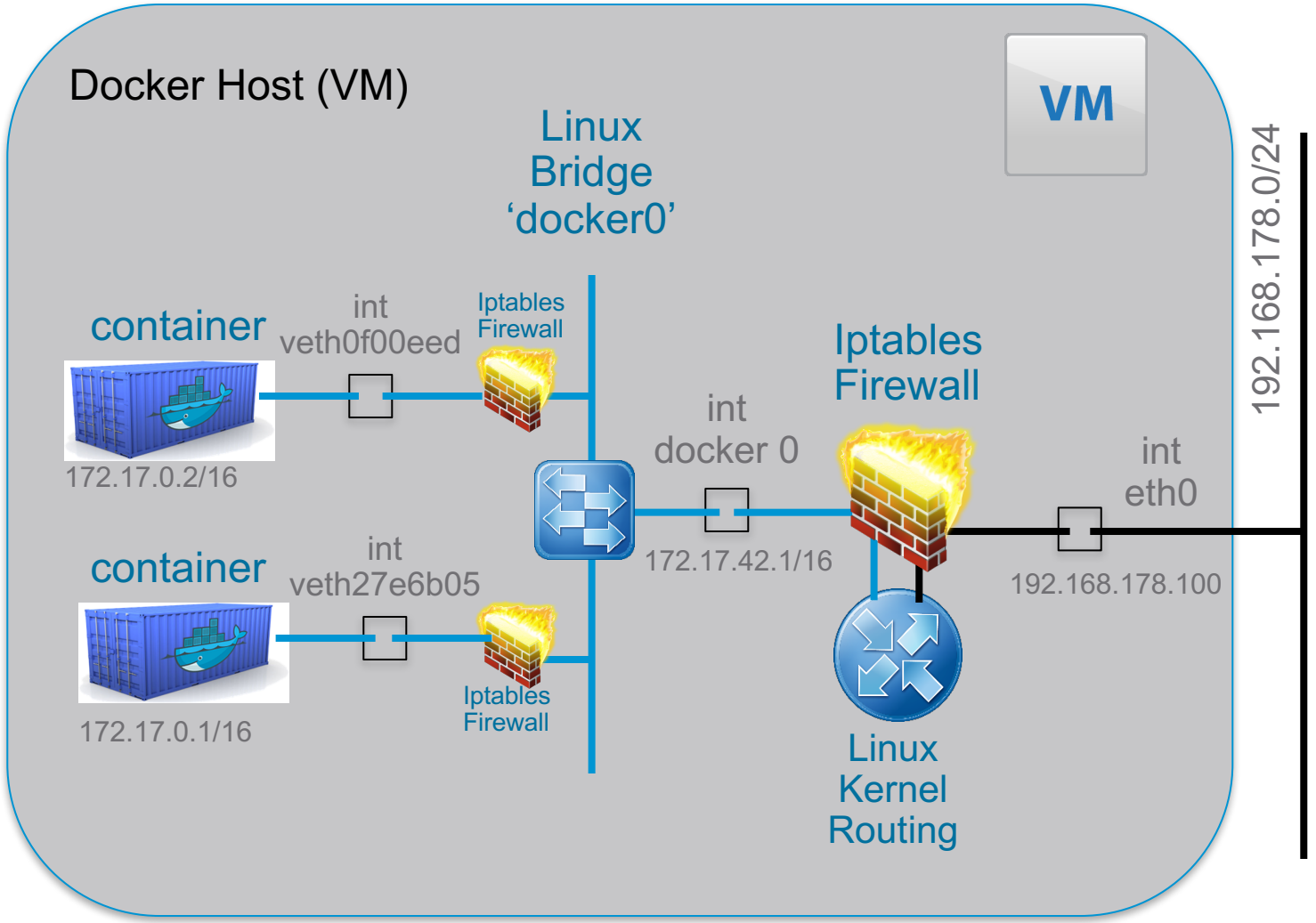
- The intention is for CNM (aka libnetwork) to implement and use any kind of networking technology to connect and discover containers
- Partitioning, Isolation, and Traffic Segmentation are achieved by dividing network addresses
- CNM does not specify one preferred methodology for any network overlay scheme



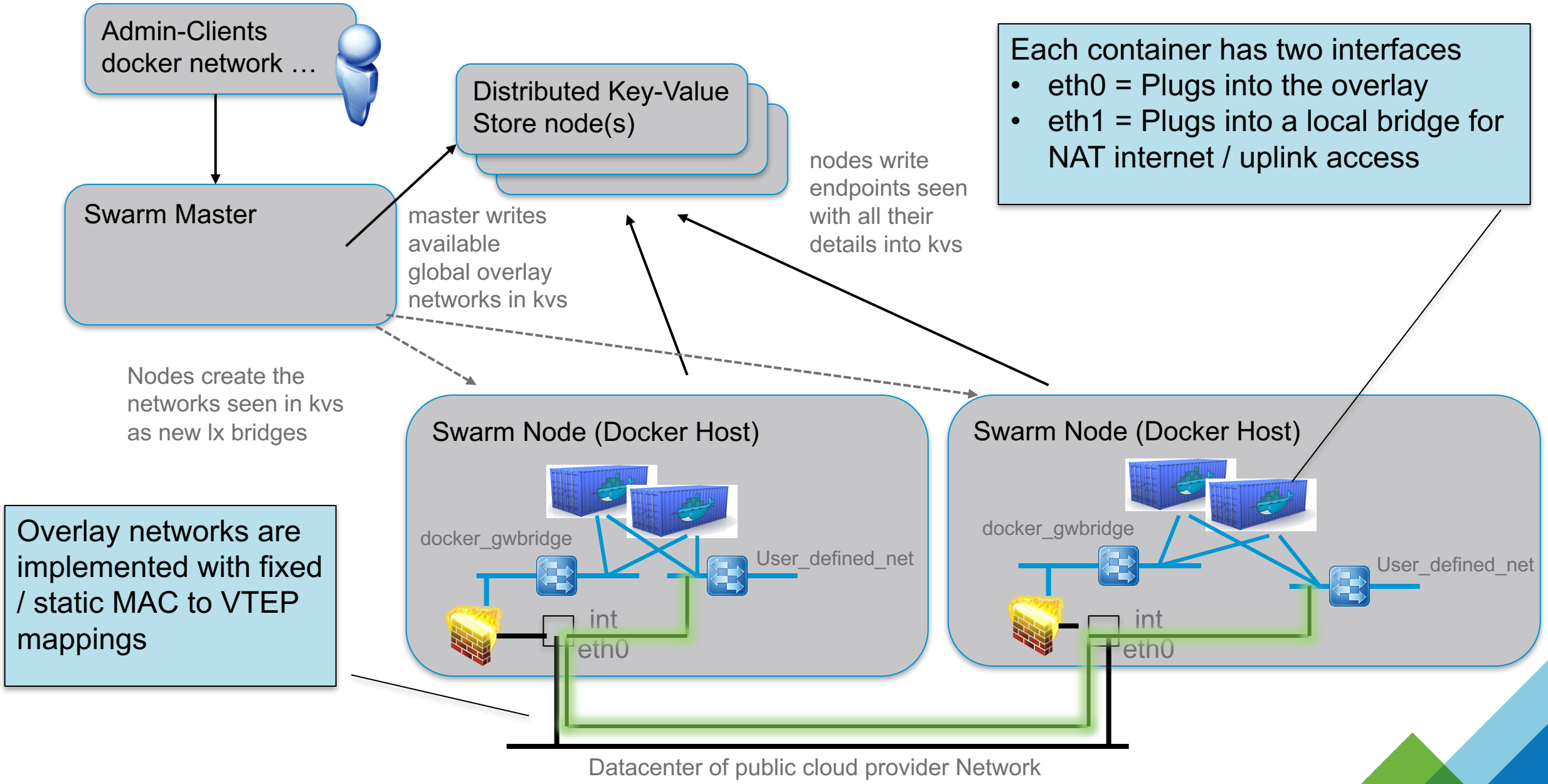

```
~/ docker network create \
   --driver overlay \
   --subnet 192.168.1.0/24 \
   multi-host-network
```

```
~/ docker network connect multi-host-network container1
```

Docker networking – Using the defaults



Docker Swarm && libnetwork – Built-In Overlay model



Docker Networking – key points

- Docker adopts the Container Network Model (CNM), providing the following contract between networks and containers:
  - All containers on the same network can communicate freely with each other
  - Multiple networks are the way to segment traffic between containers and should be supported by all drivers
  - Multiple endpoints per container are the way to join a container to multiple networks
  - An endpoint is added to a network sandbox to provide it with network connectivity
- Docker Engine can create overlay networks on a single host. Docker Swarm can create overlay networks that span hosts in the cluster
- A container can be assigned an IP on an overlay network. Containers that use the same overlay network can communicate, even if they are running on different hosts
- By default, nodes in the swarm encrypt traffic between themselves and other nodes. Connections between nodes are automatically secured through TLS authentication with certificates

Container Interfaces && Network Connectivity

Kubernetes

Kubernetes Architectural overview

Users accessing services

Admin-Clients (kubectl, ..)

Kubernetes Master

Authentication / Authorization

APIs

Scheduling actuator

REST interface (pods, services, rep. controllers)

scheduler

Controller Manager (replication controller, etc)

Master components are colocated or spread across machines

Distributed Key-Value Store node(s) (etcd)

Kubernetes Nodes (Minions)

Kubelet

Kube-Proxy

Docker engine

Control Pod

cadvisor

skyDNS

Pod

Pause

Pod

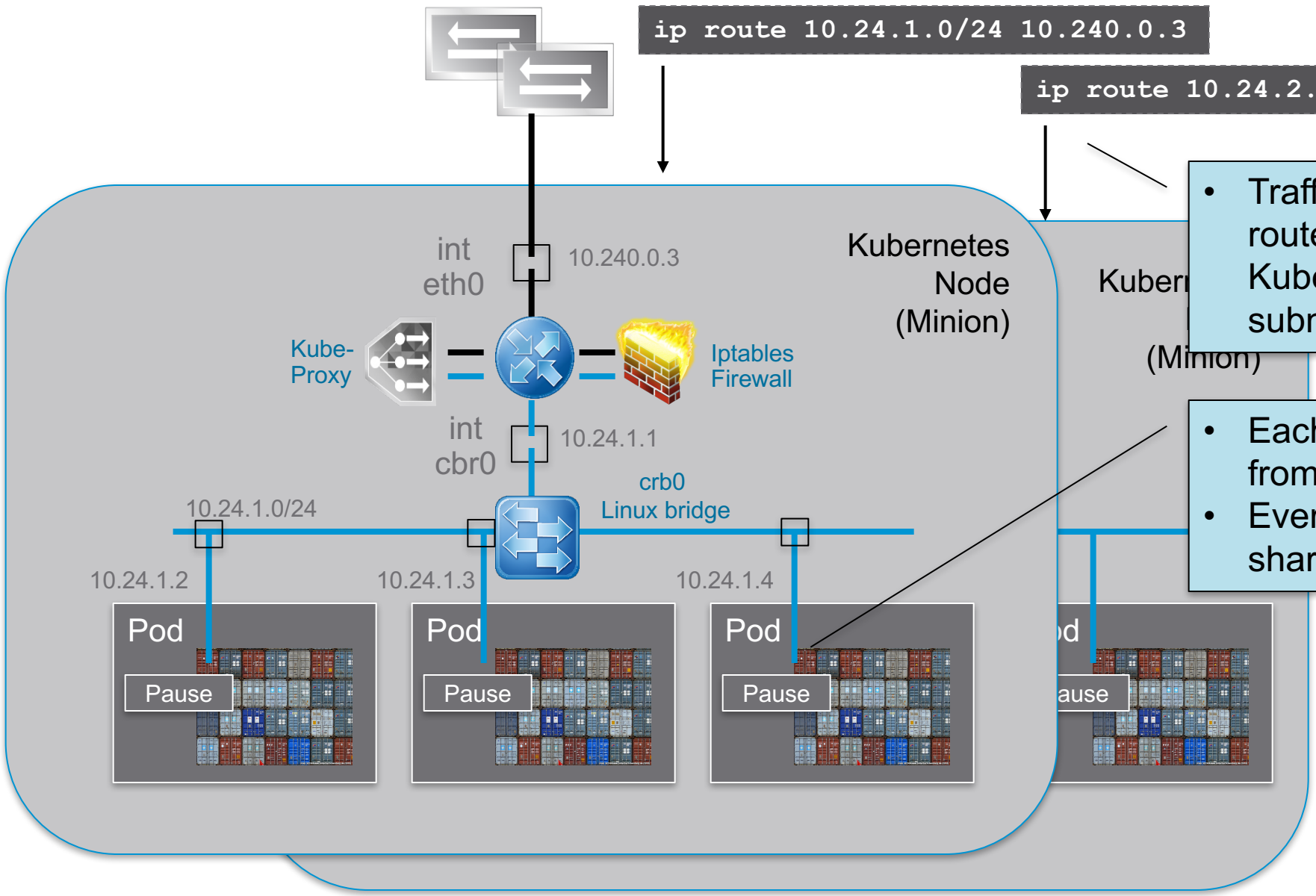
Quick Overview of Kubernetes

Kubernetes (k8s) = Open Source Container Cluster Manager

- **Pods:** tightly coupled group of containers
- **Replication controller:** ensures that a specified number of pod "replicas" are running at any one time.
- **Networking:** Each pod gets its own IP address
- **Service:** Load balanced endpoint for a set of pods with internal and external IP endpoints
- **Service Discovery**: Using env variable injection or SkyDNS with the Service

- Uses etcd as distributed key-value store
- Has its roots in ‘borg’, Google’s internal container cluster management

Kubernetes Node (Minion) – networking details

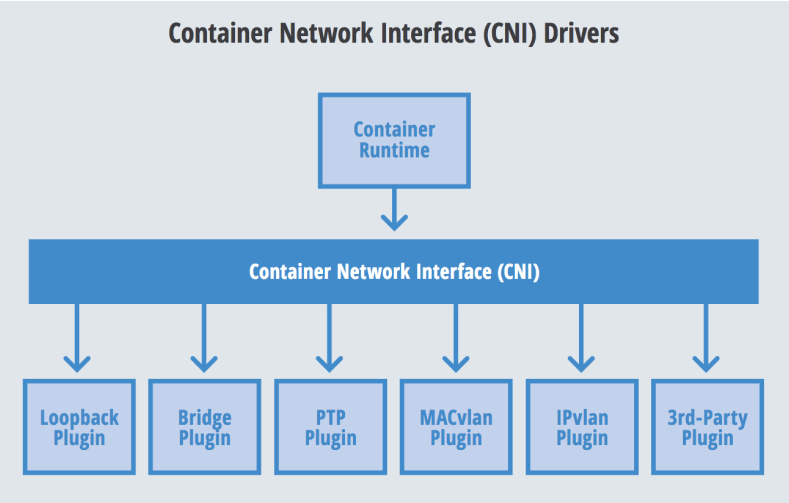


- Traffic destined to a POD is routed by the IaaS network to the Kubernetes node that 'owns' the subnet

- Each POD uses one single IP from the nodes IP range
- Every container in the POD shares the same IP

Container Network Interface (CNI)

- Kubernetes uses the Container Network Interface (CNI) specification and plug-ins to orchestrate networking
- Very differently from CNM, CNI is capable of addressing other containers' IP addresses without resorting to network address translation (NAT)
- Every time a POD is initialized or removed, the default CNI plug-in is called with the default configuration
- This CNI plug-in creates a pseudo interface, attaches it to the relevant underlay network, sets IP Address / Routes and maps it to the POD namespace




```
{
    "name": "net",
    "type": "bridge",
    "bridge": "br-int",
    "isGateway": true,
    "ipMasq": false,
    "ipam": {
        "type": "host-local",
        "subnet": "10.96.0.64/26"
    }
}
```

/etc/cni/net.d/10-bridge.conf

Kubernetes Networking – key points

- Kubernets adopts the Container Network Interface (CNI) model to provide a contract between networks and containers

- From a user perspective, provisioning networking for a container involves two steps:
  - Define the network JSON
  - Connect container to the network

- Internally, CNI provisioning involves three steps:
  - Runtime create a network namespace and gives it a name
  - Invokes the CNI plugin specified in the “type” field of the network JSON. Type field refers to the plugin being used and so CNI invokes the corresponding binary
  - Plugin code in turn will create a veth pair, check the IPAM type and data in the JSON, invoke the IPAM plugin, get the available IP, and finally assign the IP address to the interface

Container Interfaces && Network Connectivity

Summary

Container Networking Specifications

Container Networking Model

CNM

- Specification proposed by Docker, adopted by projects such as libnetwork
- Plugins built by projects such as Weave, Project Calico and Kuryr
- Supports only Docker runtime

Container Networking Interface

CNI

- Specification proposed by CoreOS and adopted by projects such as Kubernetes, Cloud Foundry and Apache Mesos
- Plugins built by projects such as Weave, Project Calico, Contiv Networking
- Supports any container runtime

CNI and CNM commonalities…

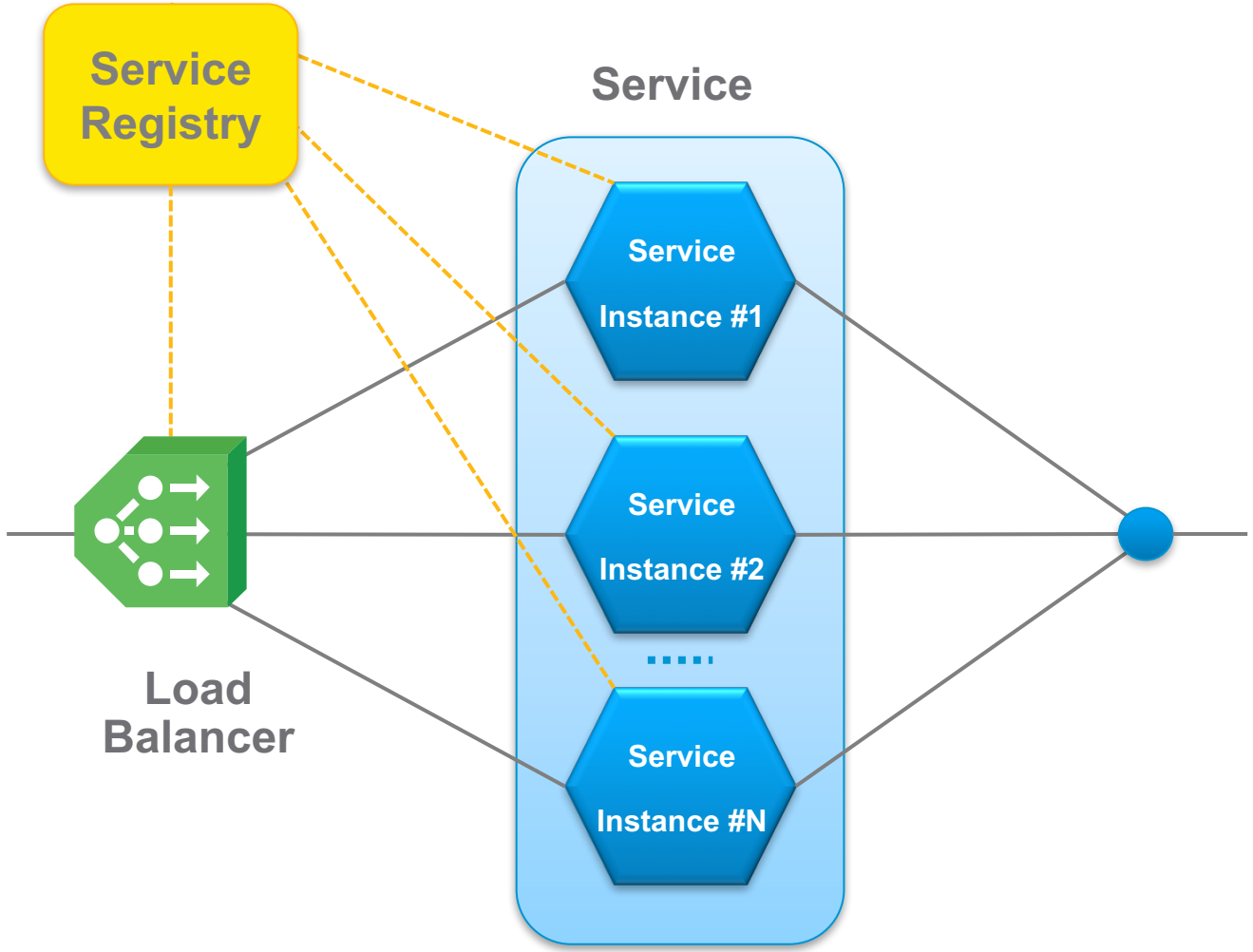
- CNI and CNM models are both driver-based
  - provide “freedom of selection” for a specific type of container networking
- Multiple Network drivers can be active and used concurrently
  - 1-1 mapping among network type and network driver
- Containers are allowed to join one or more networks
- Container runtime can lunch network in its own namespace
  - delegate to the network driver the responsibility of connecting the container to the network

Container Networking Specifications (cont.)

Container Networking Models	Container Network Interface (CNI)	Docker Libnetwork
Container Platform	kubernetes, OPENSHIFT ENTERPRISE by Red Hat, Pivotal Cloud Foundry, MESOSPHERE	docker SWARM
Pluggable Network Stack	YES, YES, WORK IN PROGRESS, WORK IN PROGRESS	YES

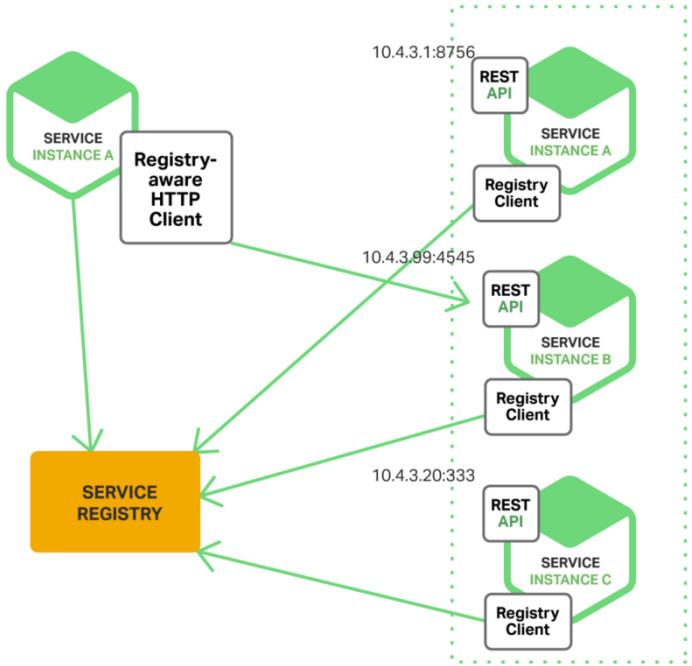
Service Discovery && Load Balancing

Service Anatomy



Client vs Server side Service discovery

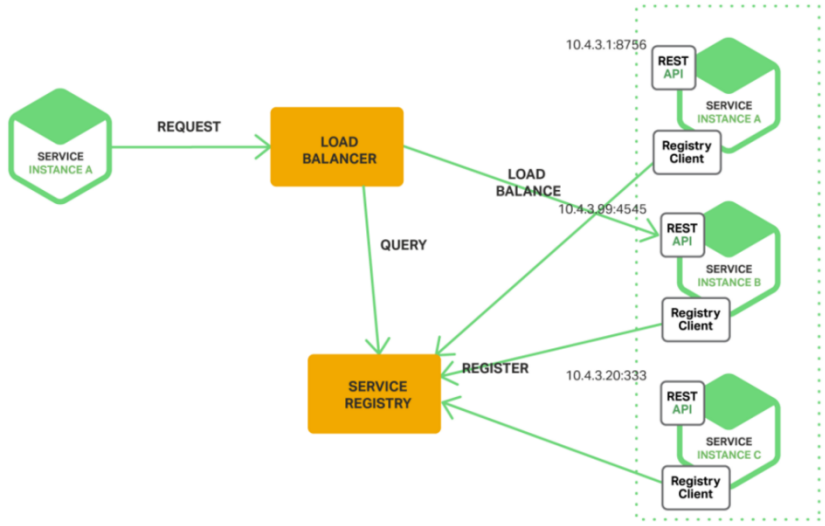
Client Discovery



- Client talks to Service registry and does load balancing.
- Client service needs to be Service registry aware.

eg: Netflix OSS

Server Discovery



- Client talks to load balancer and load balancer talks to Service registry.
- Client service need not be Service registry aware

eg: Consul, AWS ELB, K8s, Docker

What should Service Discovery provide ?

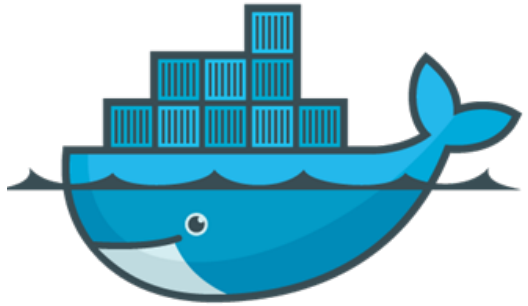
- Discovery
  - Services need to discover each other dynamically, to get IP address and port detail to communicate with other services in the cluster
  - Service Registry maintains a database of services and provides an external API (HTTP/DNS). Typically implemented as a distributed key, value store
  - Registrator registers services dynamically to Service registry by listening to Service creation and deletion events
- Health check
  - Monitoring Service Instance health dynamically and updates Service registry appropriately
- Load balancing
  - Traffic destined to a particular service should be dynamically load balanced to “healthy” instances providing that service

Health Check options…

- Script based check
  - User provided script is run periodically to verify health of the service.
- HTTP based check
  - Periodic HTTP based check is done to the service IP and endpoint address.
- TCP based check
  - Periodic TCP based check is done to the service IP and specified port.
- Container based check
  - Health check application is available as a Container. Health Check Manager invokes the Container periodically to do the health-check.

Service Discovery && Load Balancing

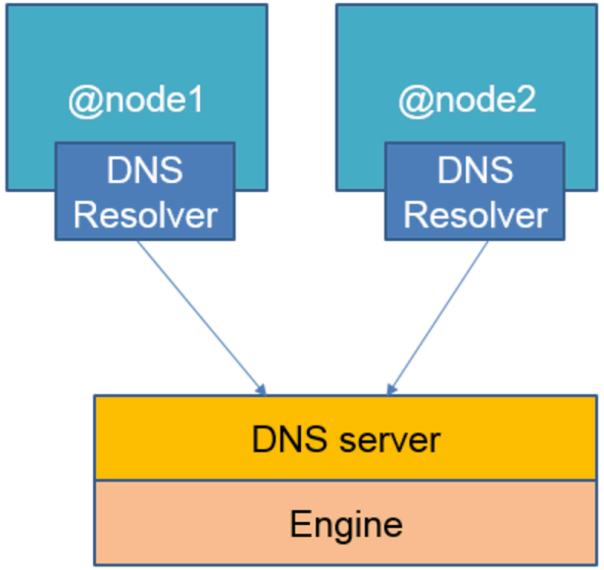
Docker

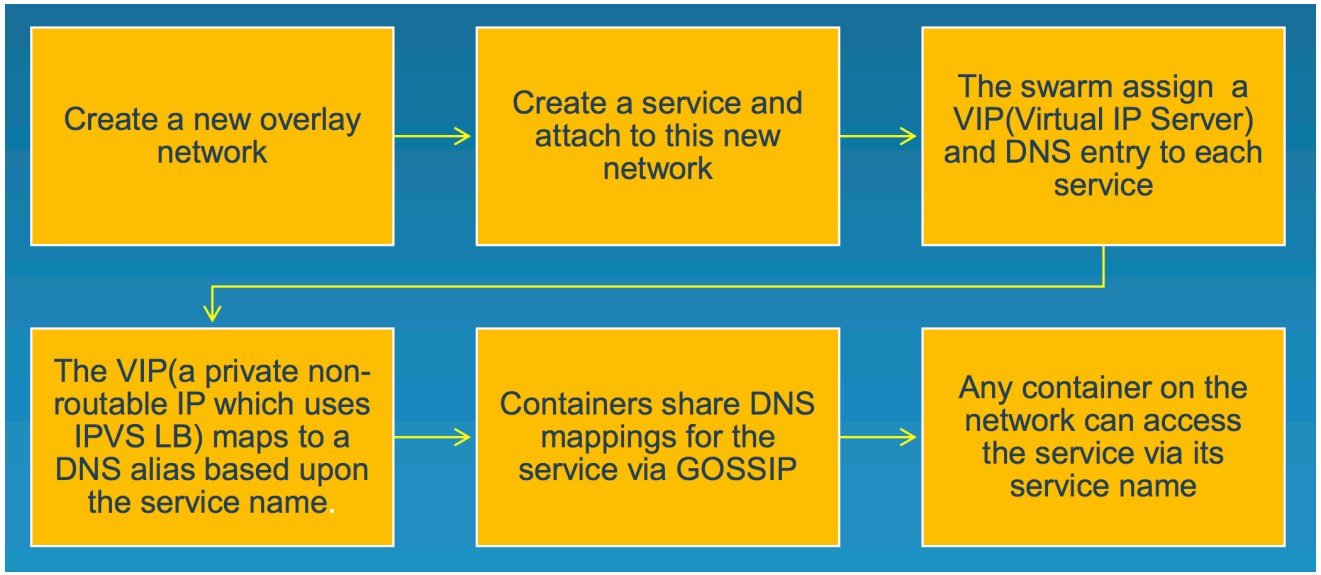


Service Discovery

Service Discovery:

- Provided by Embedded DNS
- Highly Available
- Uses Network Control Plane to learn state
- Can be used to discover services and containers

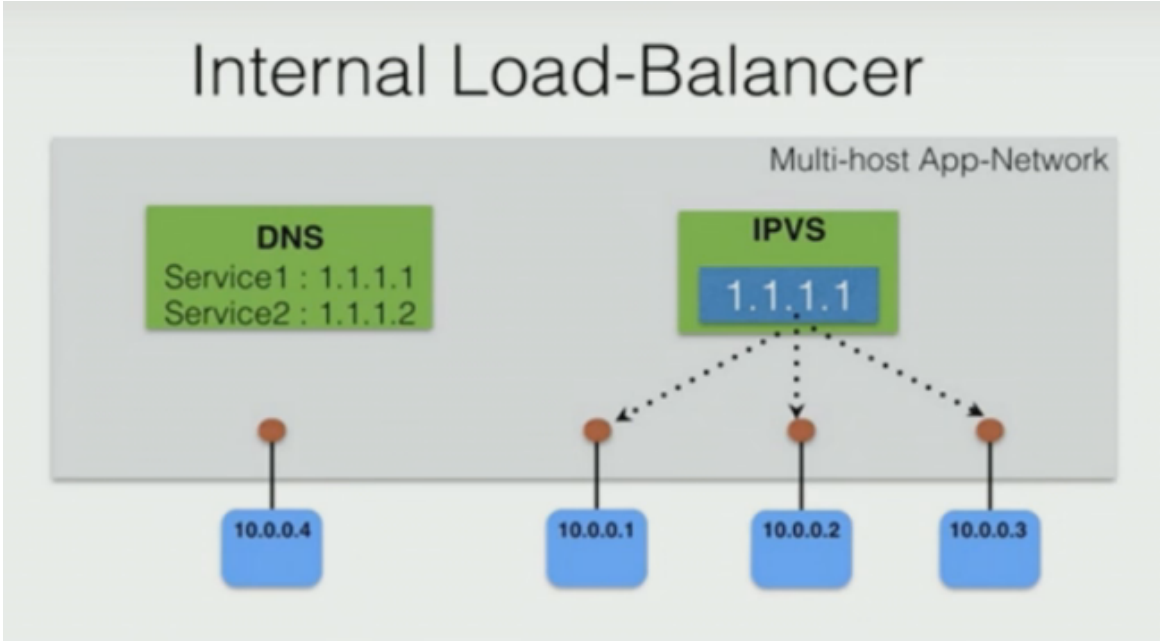




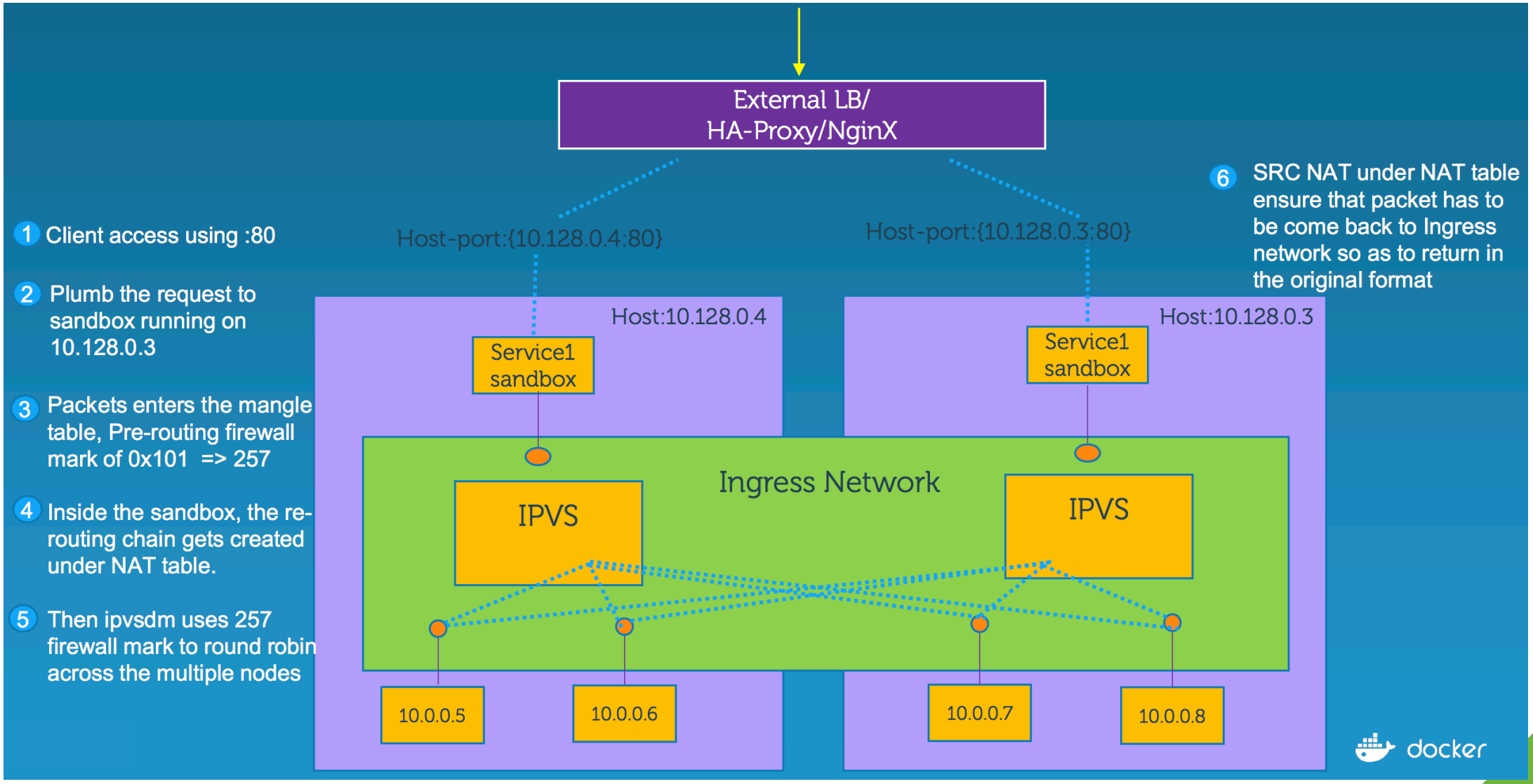
Service Discovery in a nutshell

Internal Load Balancer - IPVS

- IPVS (IP Virtual Server) implements transport-layer load balancing inside the Linux kernel, so called Layer-4 switching
- It's based on Netfilter and supports TCP, SCTP & UDP, v4 and v7
- IPVS is dynamically configurable, supports 8+ balancing methods, provides health checking



Ingress Load Balancing



Service Discovery && Load Balancing

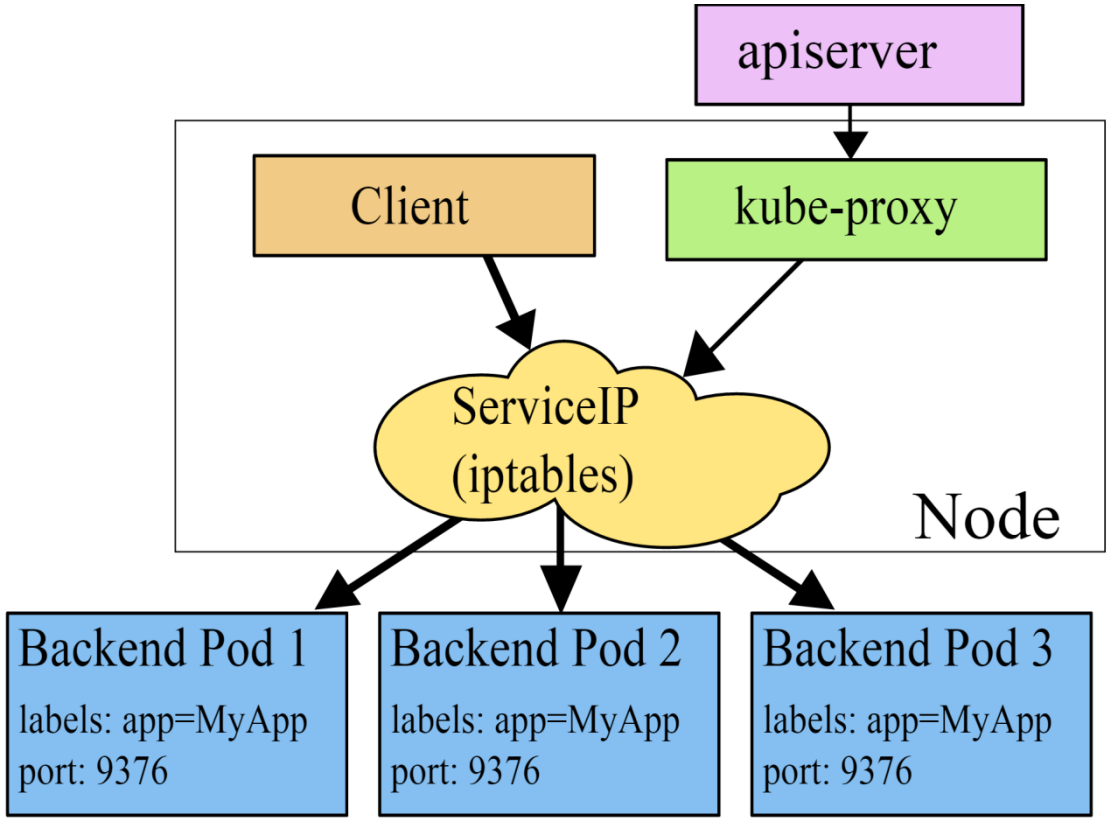
Kubernetes

Service Discovery

- Kubernetes provides two options for internal service discovery :
  - **Environment variable**: When a new Pod is created, environment variables from older services can be imported. This allows services to talk to each other. This approach enforces ordering in service creation.
  - **DNS**: Every service registers to the DNS service; using this, new services can find and talk to other services. Kubernetes provides the kube-dns service for this.

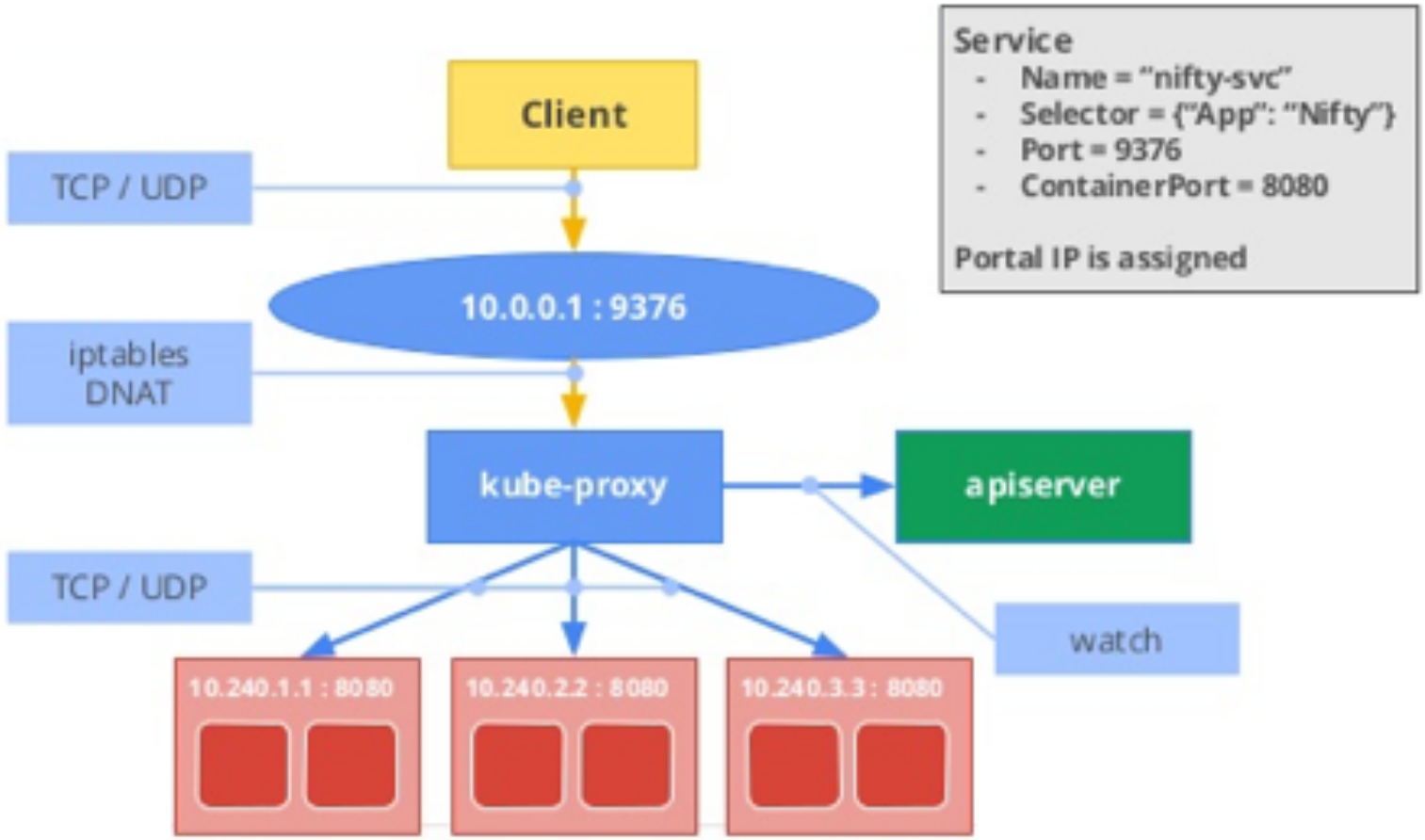
- Kubernetes provides several ways to expose services to the outside:
  - **HostNetwork / HostPort / NodePort**: In these methods, Kubernetes exposes the service through special ports (30000-32767) of the node IP address.
  - **Loadbalancer**: In this method, Kubernetes interacts with the cloud provider to create a load balancer that redirects the traffic to the Pods. This approach is currently available with GCE
  - **Ingress Controller** : Since Kubernetes v1.2.0 it's possible to use Kubernetes ingress which includes support for TLS and L7 http-based traffic routing

Internal Load Balancing



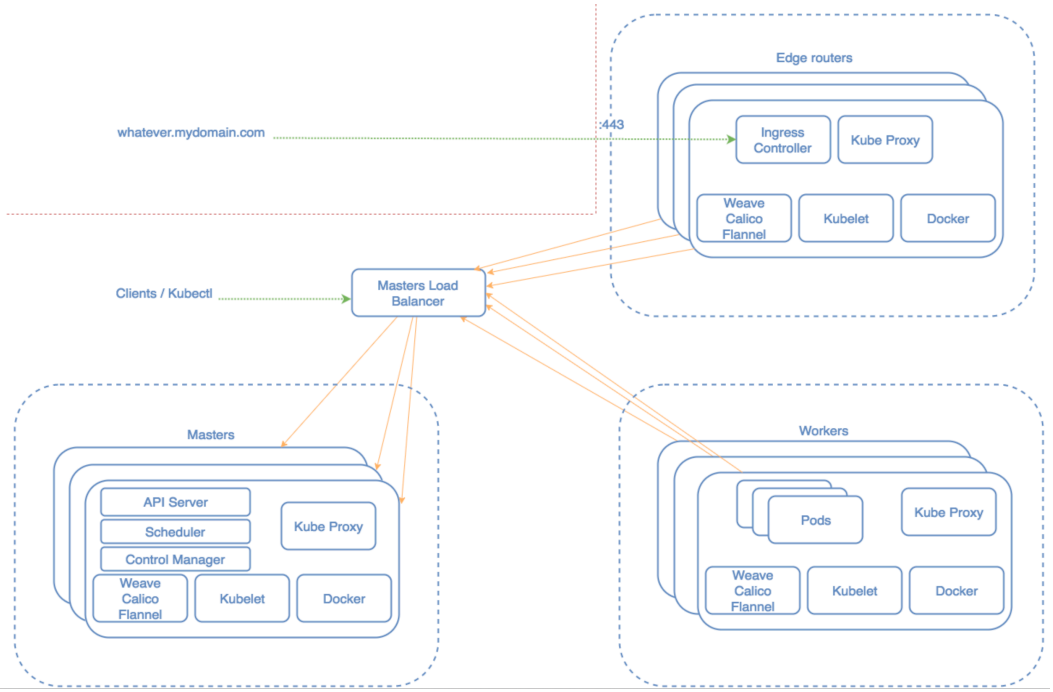
- Service name gets mapped to Virtual IP and port using Skydns
- Kube-proxy watches Service changes and updates IPtables. Virtual IP to Service IP, port remapping is achieved using IP tables
- Kubernetes does not use DNS based load balancing to avoid some of the known issues associated with it

Internal Load Balancing (cont.)

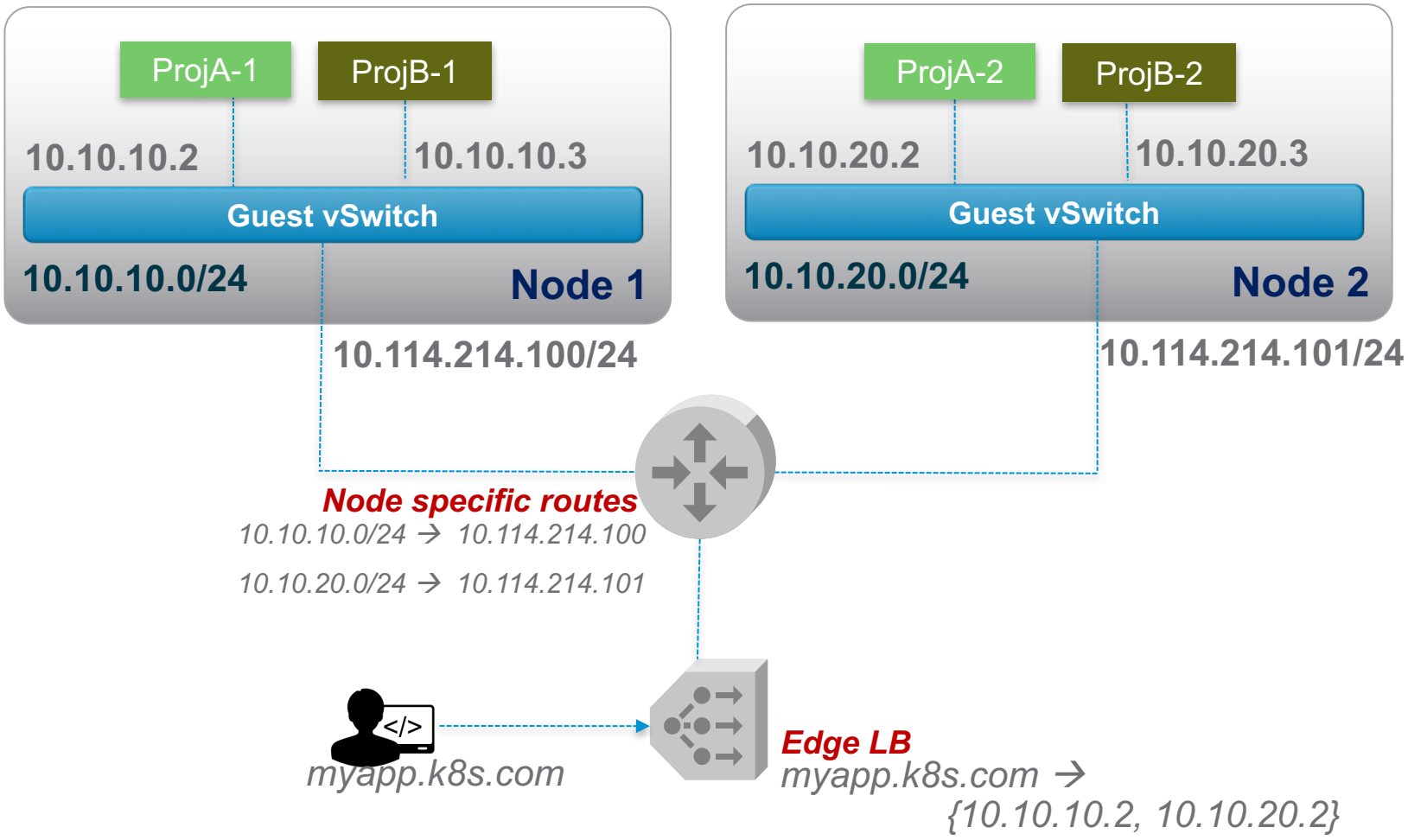


Ingress Load Balancing w/t Ingress Controller

- An Ingress is a collection of rules that allow inbound connections to reach the cluster services.
- It can be configured to give services externally-reachable urls, load balance traffic, terminate SSL, offer name based virtual hosting etc
  - Users request ingress by POSTing the Ingress resource to the API server.
- In order for the Ingress resource to work, the cluster must have an Ingress controller running. The Ingress controller is responsible for fulfilling the Ingress dynamically by watching the ApiServer's /ingresses endpoint.



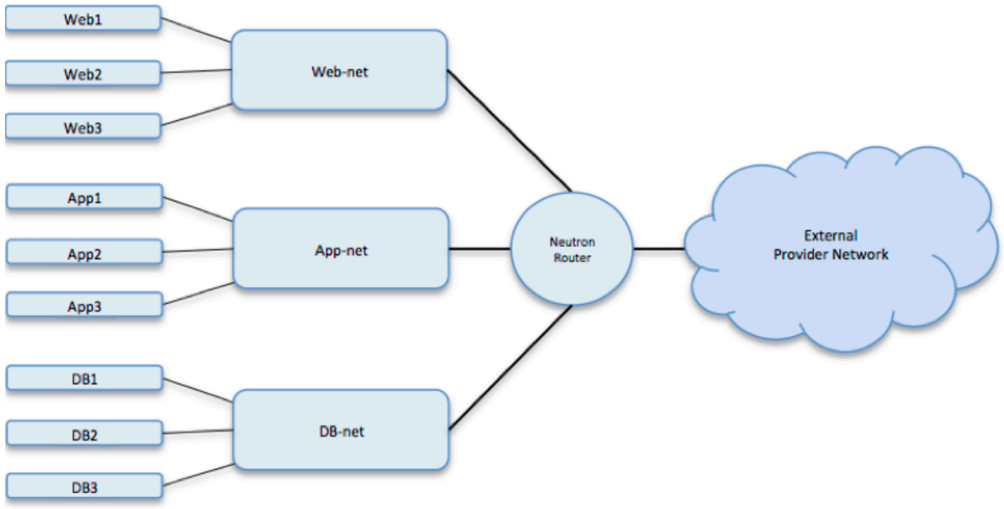
Networking for Services



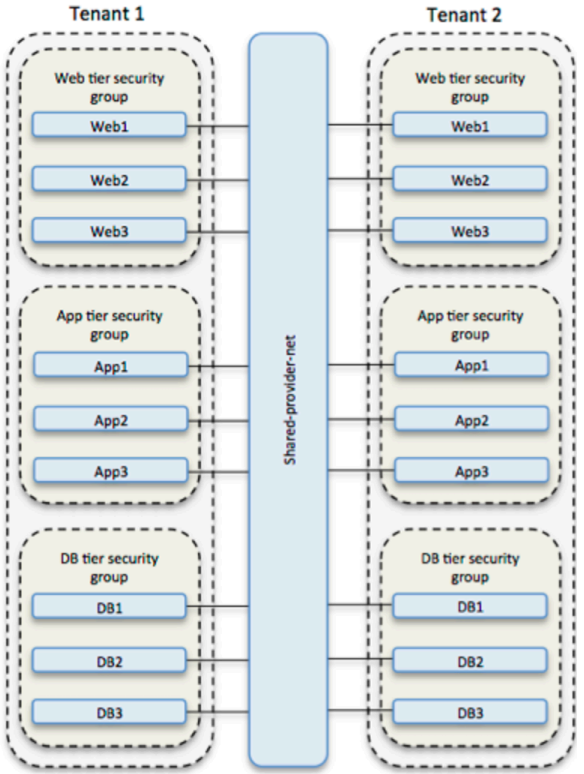
- K8s default networking configures
  - Routable IP per POD
  - Subnet per node / minion
- K8s **Service** provides East-West Load Balancing
- Provides DNS based service discovery – Service Name to IP
- Network Security Policy – in beta
- Not in K8s scope
  - Edge LB – e.g. external to frontend pods
  - Routing of a subnet to k8s node

Multi-Tenancy
Container Isolation
Micro-Segmentation

Multi-Tenancy and Application tiering



Traditional application tiering

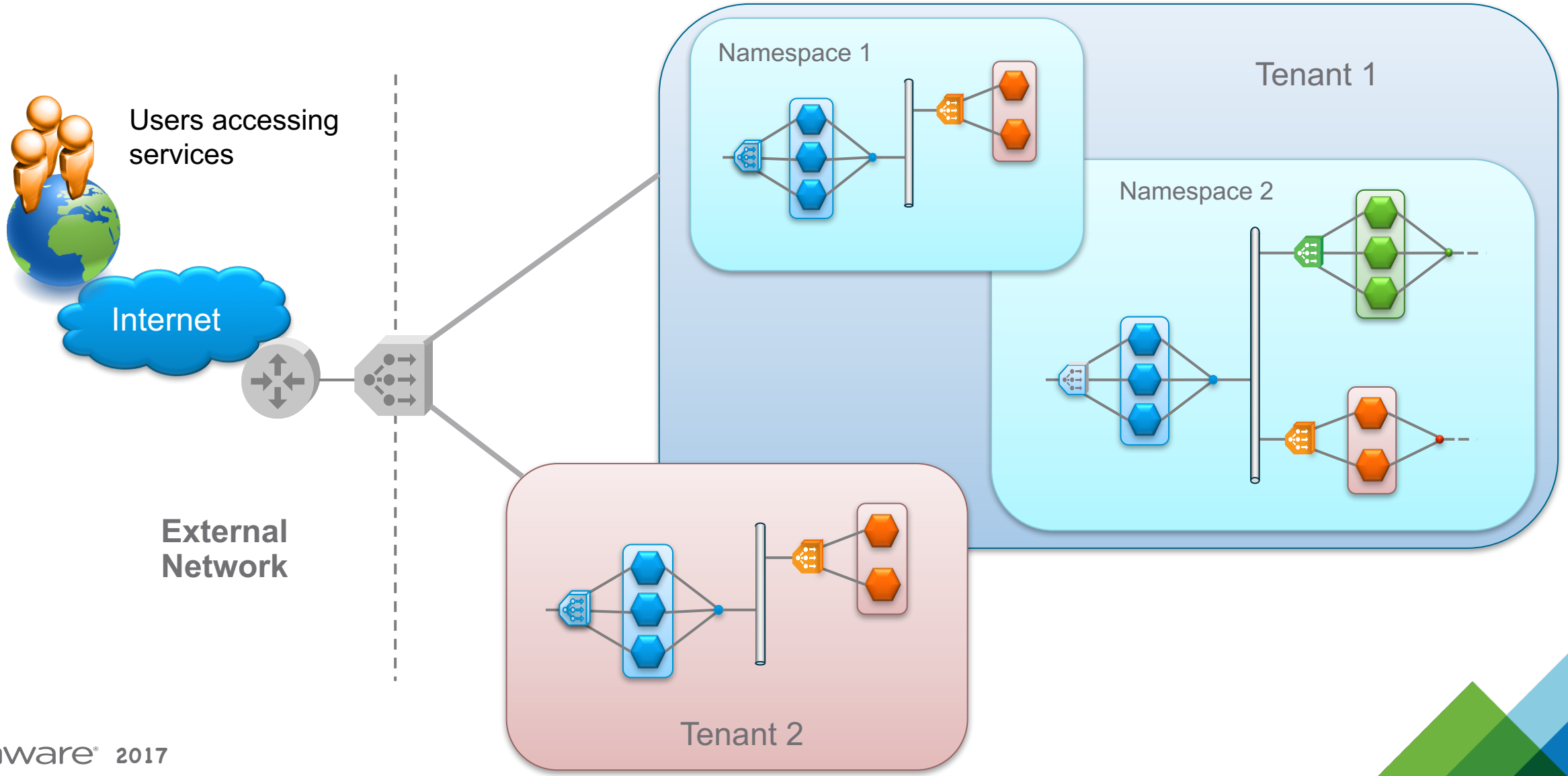


Application tiering/security zones using security groups

Multi-Tenancy and Application tiering (cont.)

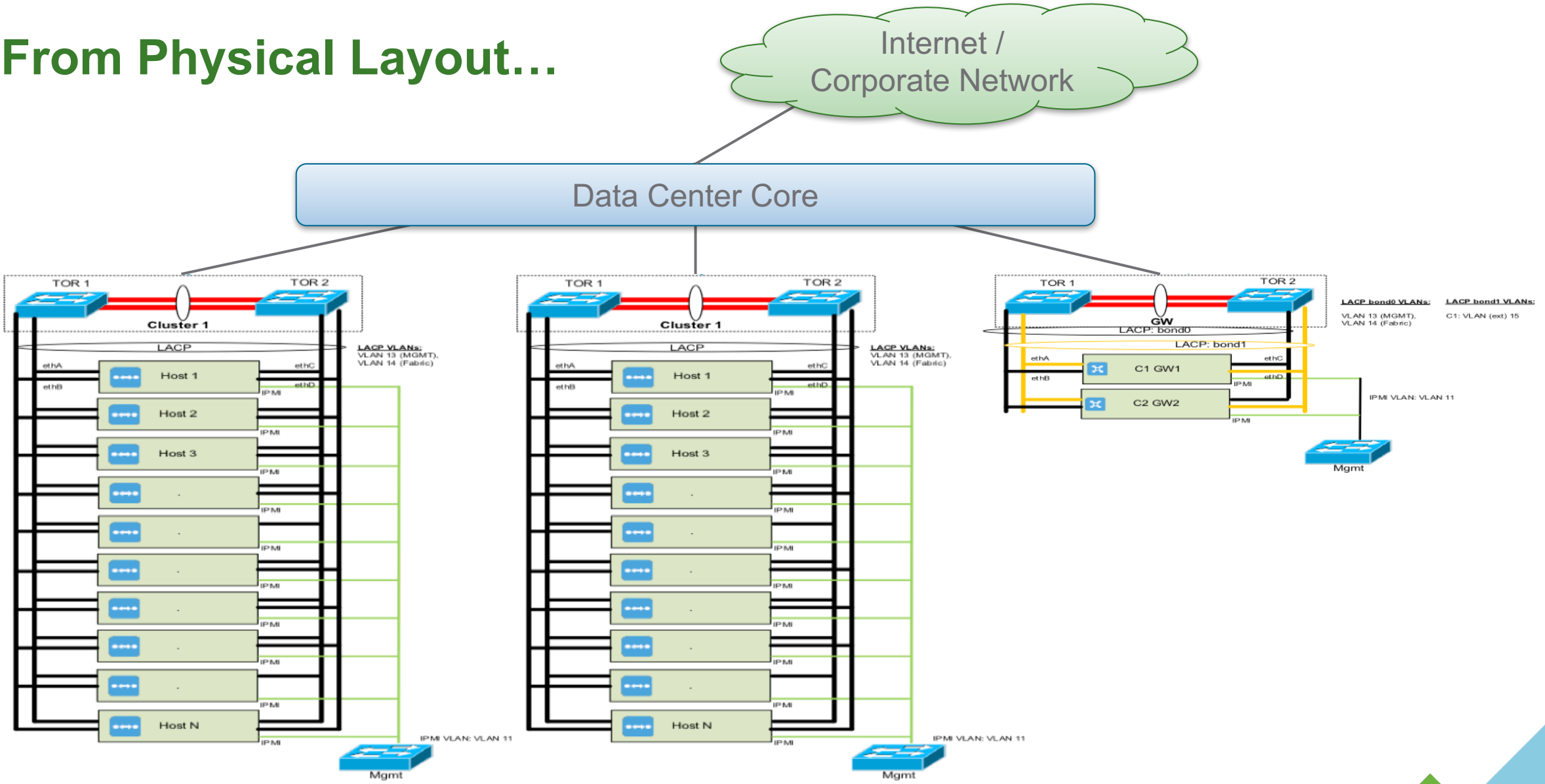
Example of Multi-Tenancy Model

Multi-Tenancy, Namespaces && Microsegmentation



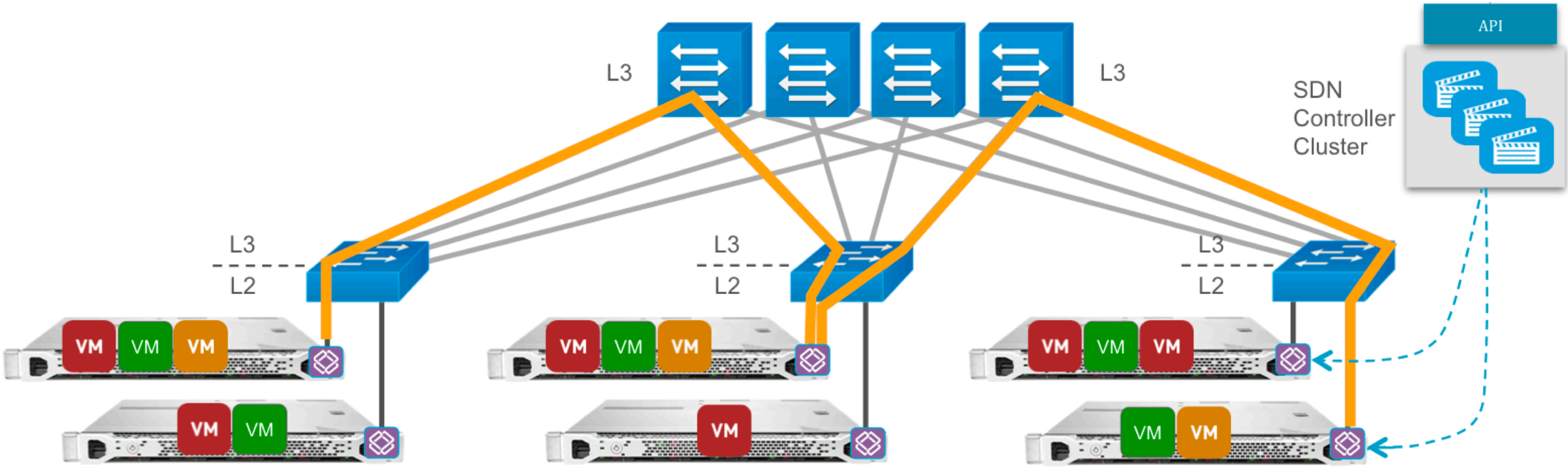
On-Premise Private Cloud design

From Physical Layout…

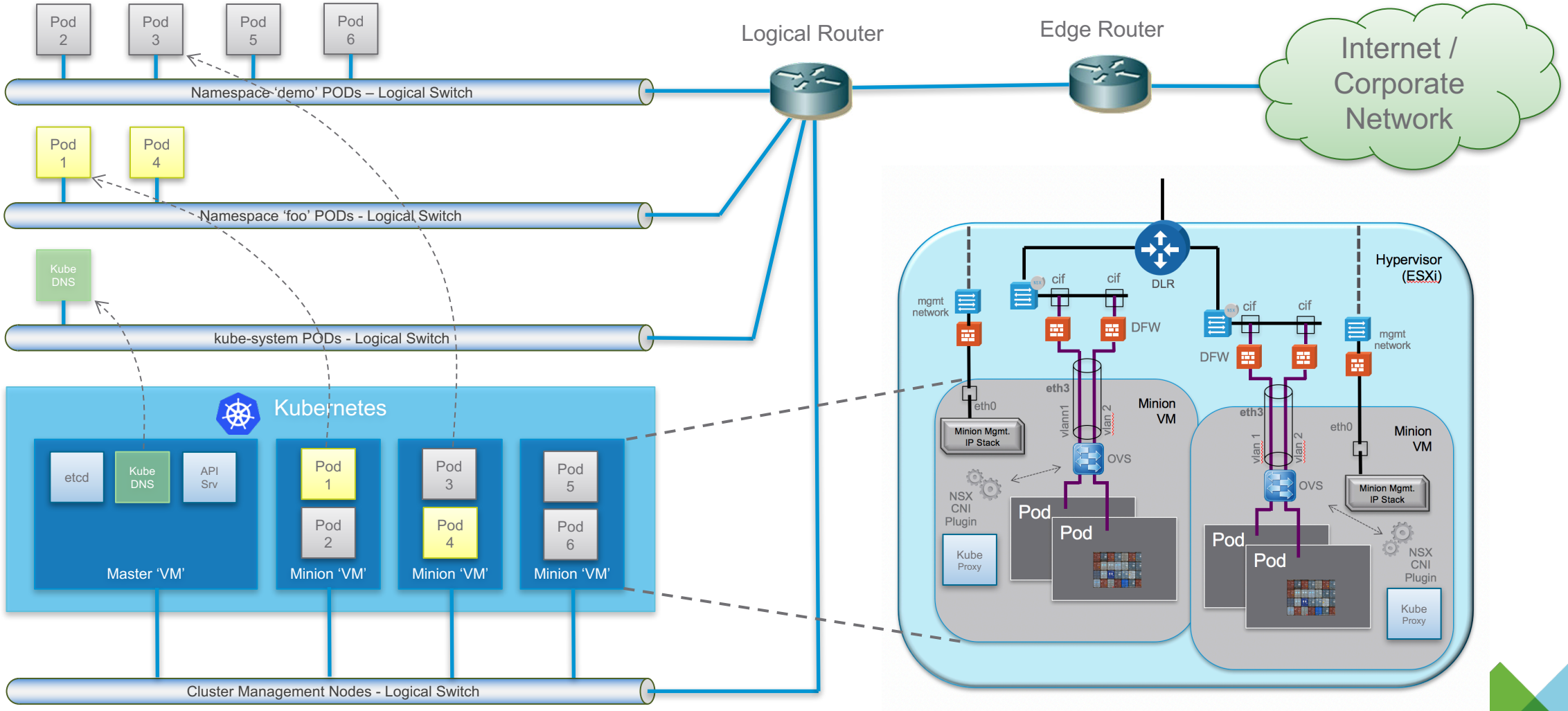


…to Overlay-based Networking Model…

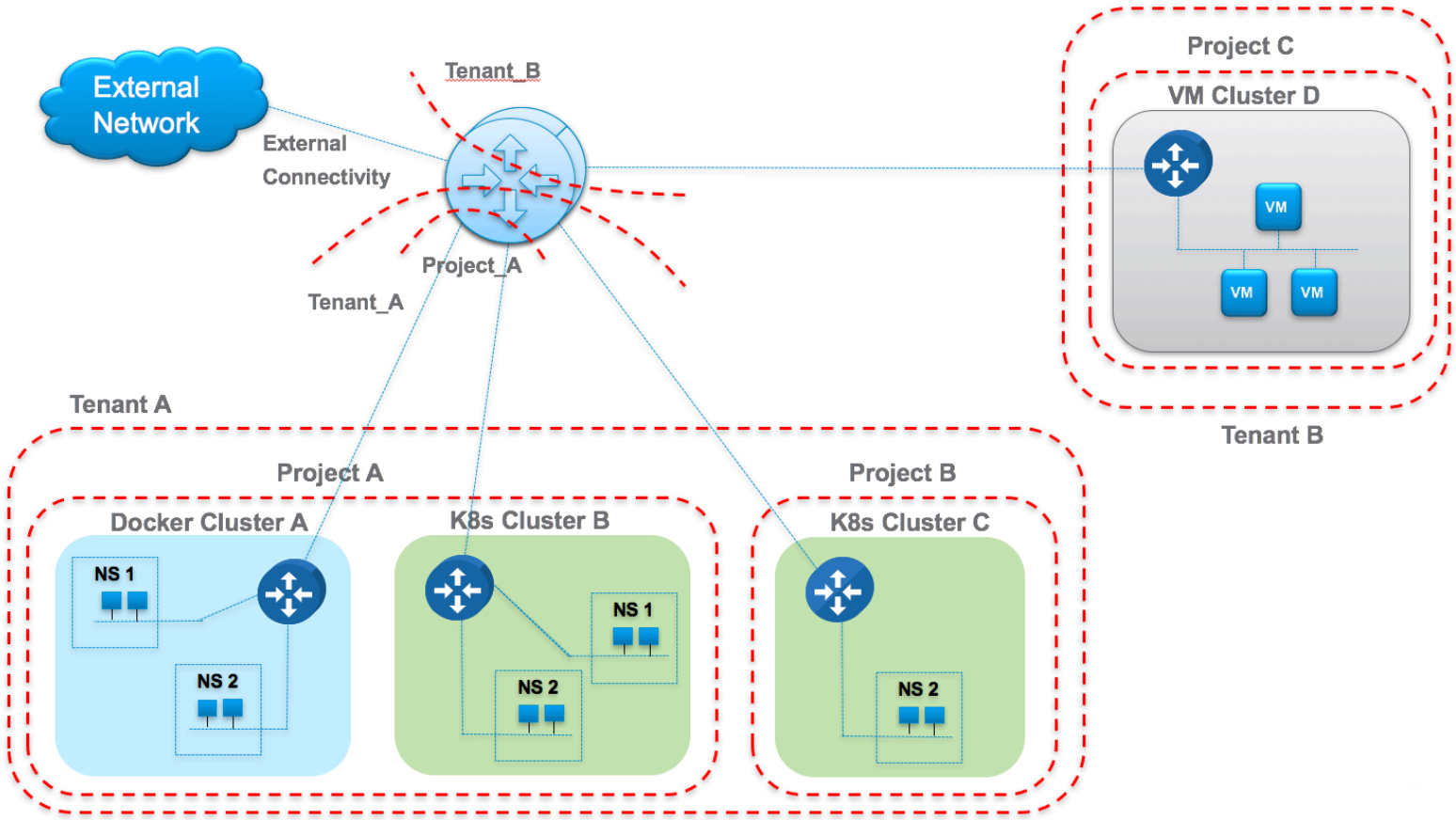
- CMS to communicate with SDN Controller via vendor-specific APIs
- SDN Controller manages vSwitches in Hypervisors
- Vmware NSX, Contrail, Nuage, Midokura, …



…to Cluster Deployment on Logical Networks…



…to Multi-Cluster / Multi-Tenancy deployments



Multi-Tenancy deployment and Networking constrains

Sample of Provisioning Workflow…

- **tenant create** tano-tenant -l 'vm 200, cont 1000, memory 2000 GB, ephemeral-disk.capacity 20000 GB, persistent-disk.capacity 20000 GB, sdn.floatingips 100'
- **tenant set** tano-tenant
- **project create** tano-project
- **project set** tano-project
- **cluster create** -n tano-kube1 -k KUBERNETES --vm_flavor vm-sm --disk_flavor disk-sm --number-of masters 1 --number-of-etcds 1 --number-of-workers 10
- **tenant apply-policy** -f network-policy.yml

```
apiVersion: extensions/v1
kind: NetworkPolicy
metadata:
  name: network-policy
spec:
  egress:
    TenantSelector: tano-tenant
    ProjectSelector: tano-project
    ClusterSelector: *
  ingress:
    from:
      TenantSelector: *
      ProjectSelector: *
      ClusterSelector: *
    ports:
      protocol: tcp
      port: 6379
    expression:
      MatchExpression: "src_project==front-end ||
                        src_tenant==external"
```

network_policy.yml

Q & A

Thank You!

@cloudnativeapps
#vmwcna

vmware.github.io

blogs.vmware.com/cloudnative

microservices@vmware.com