Kubernetes Networking @AWS

AWS

Infrastructure constructs

Network Security

Services

AWS Infrastructure Components

Regions - used to manage network latency and regulatory compliance per country. No Data replication outside a region

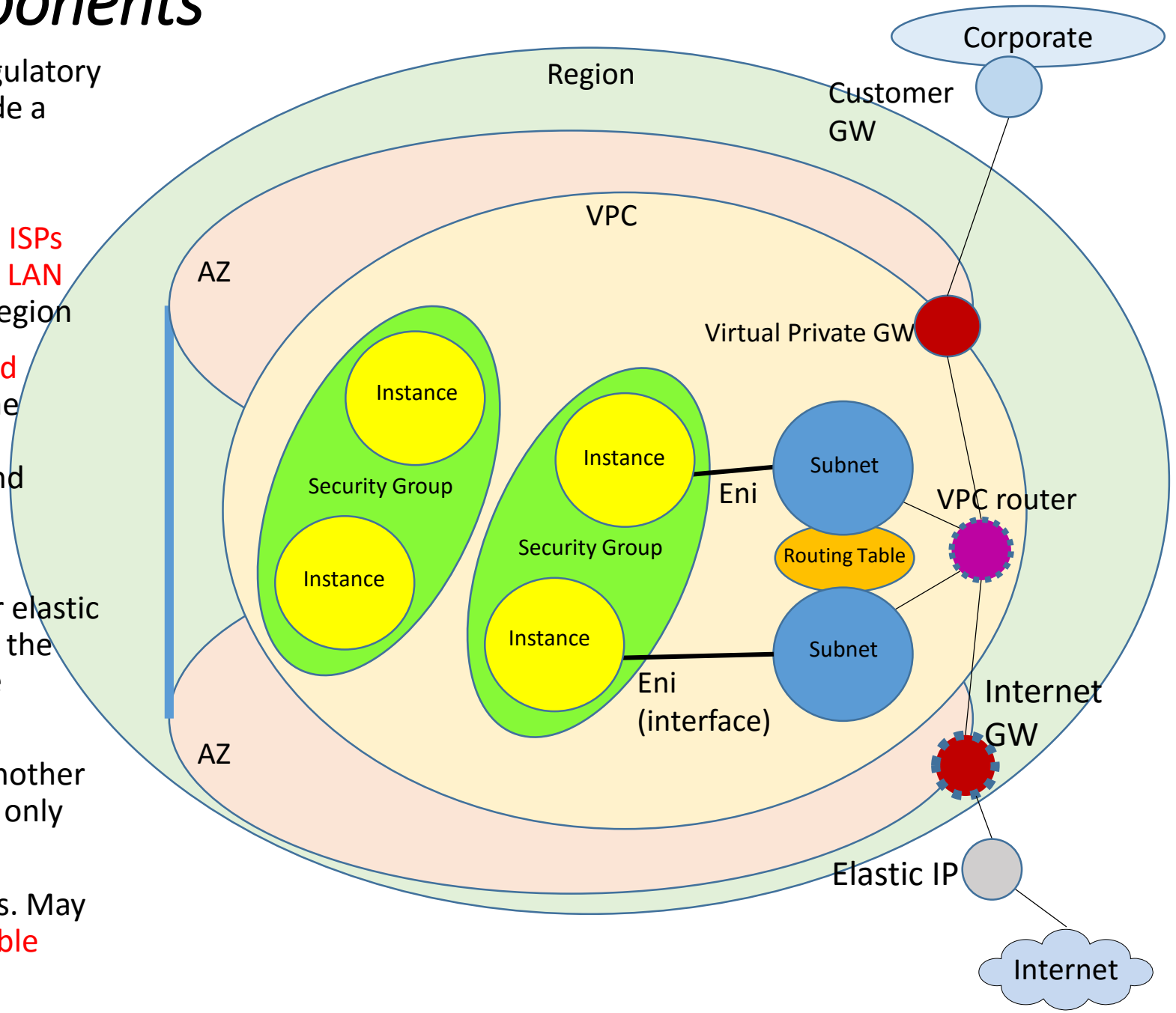
Availability Zones - at least two in a region. Designed for fault isolation. Connected to multiple ISPs and different power sources. Interconnected using LAN speed for inter-communications within the same region

VPC – spans all region's AZs. Used to create isolated private cloud within AWS. IP ranges allocated by the customer.
Networking – interfaces, subnets, routing tables and gateways (Internet, NAT and VPN).
Security – security groups

Interface (ENI) – can include primary, secondary or elastic IP. Security group attaches to it. Independent from the instance (even though primary interface cannot be detached from an instance).

Subnet – connects one or more ENIs, can talk to another subnet only through a L3 router. Can connected to only one routing table. Cannot span AZs

Routing table – decides where network traffic goes. May connect to multiple subnets. 50 routes limit per table



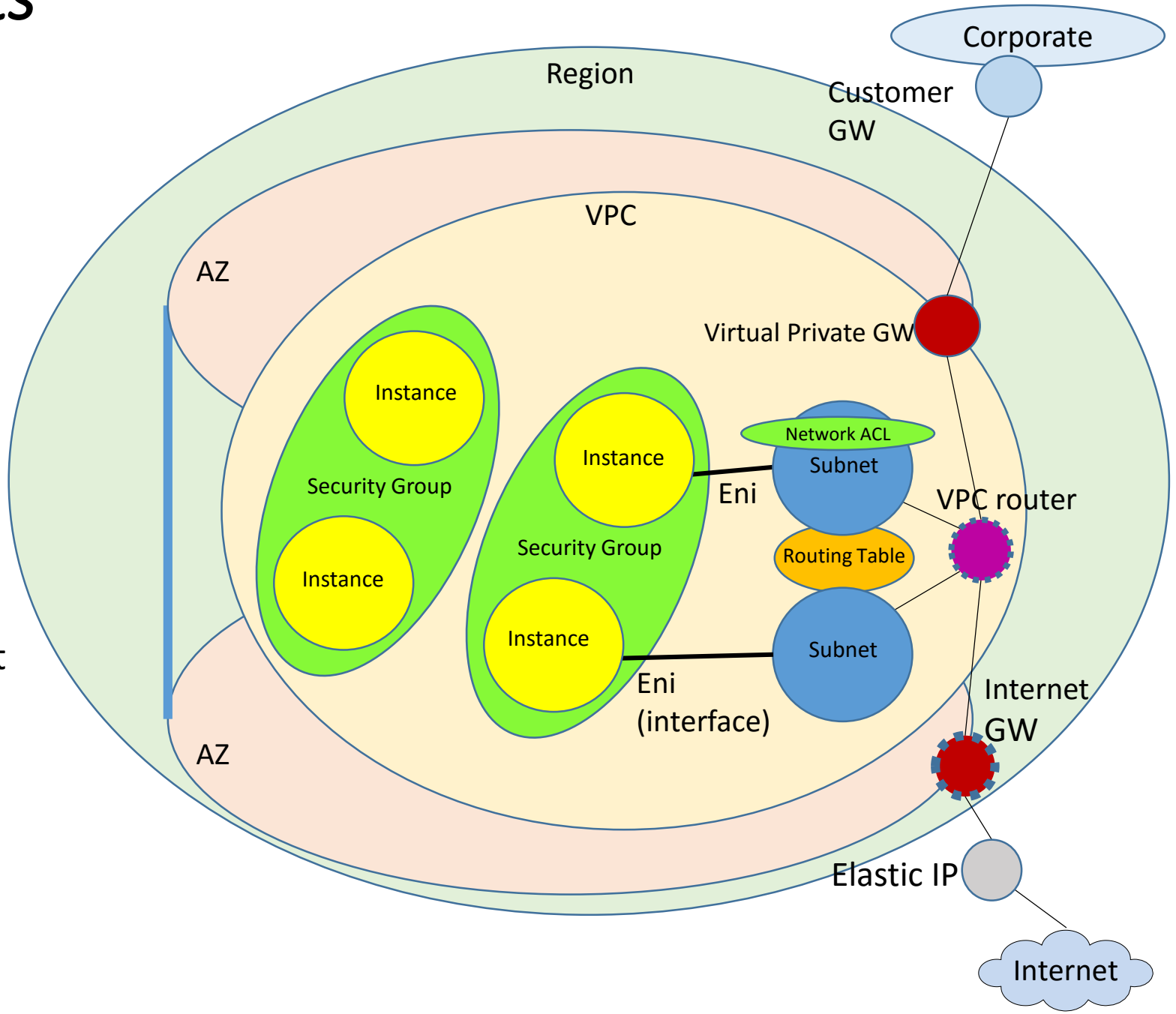
VPC Security Components

Security group - virtual firewall for the instance to control inbound and outbound traffic.

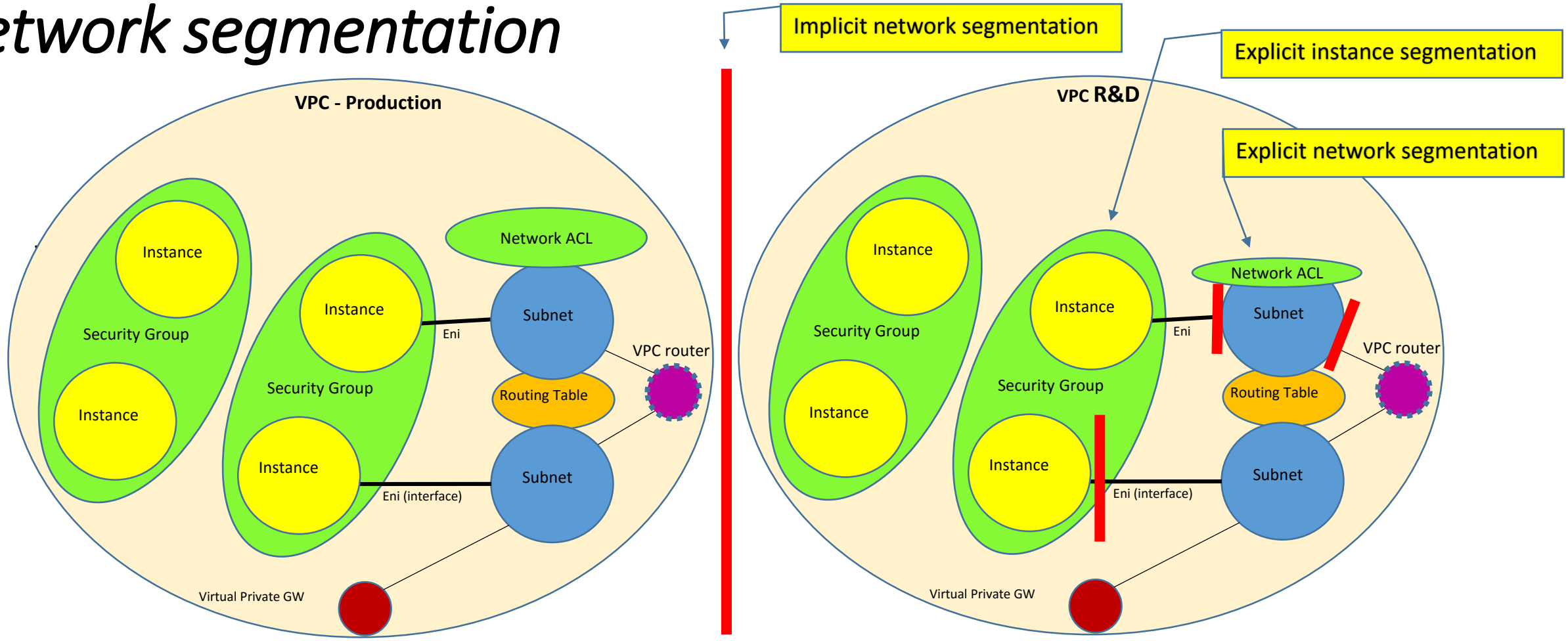
- Applied on ENI (instance) only
- No deny rules
- Stateful – return traffic implicitly allowed
- All rules evaluated before decision
- Up to five per instance

Network ACL- virtual IP filter on the subnet level

- Applied on subnet only
- Allows deny rules
- Stateless – return traffic should be specified
- First match takes



Network segmentation

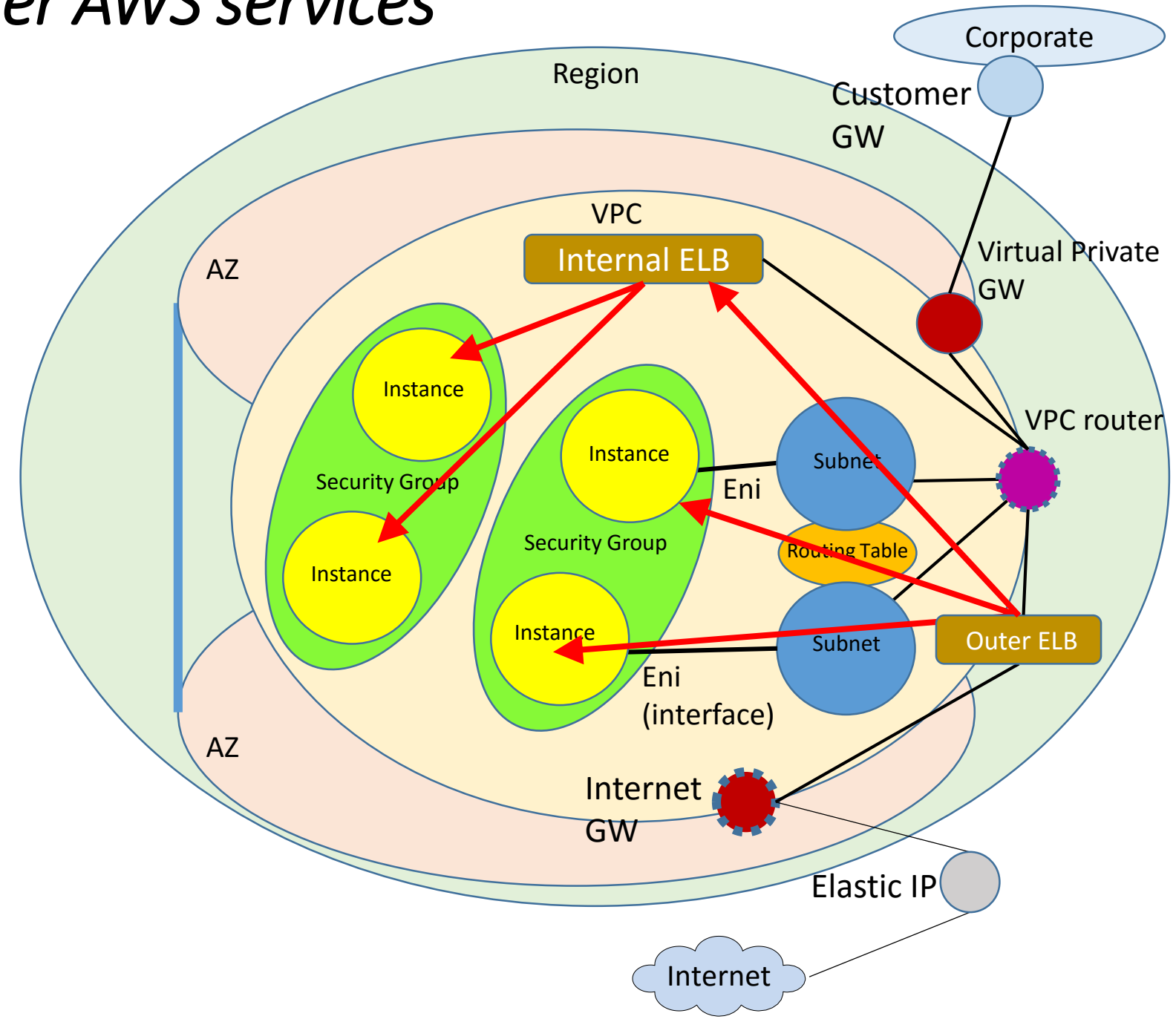


- AWS VPC – Great even for internal zoning. No need for policies
- Security group – Statefull and flexible. Network location agnostic
- Network ACL – good for additional subnet level control

VPC integration with other AWS services

Elastic load balancing –

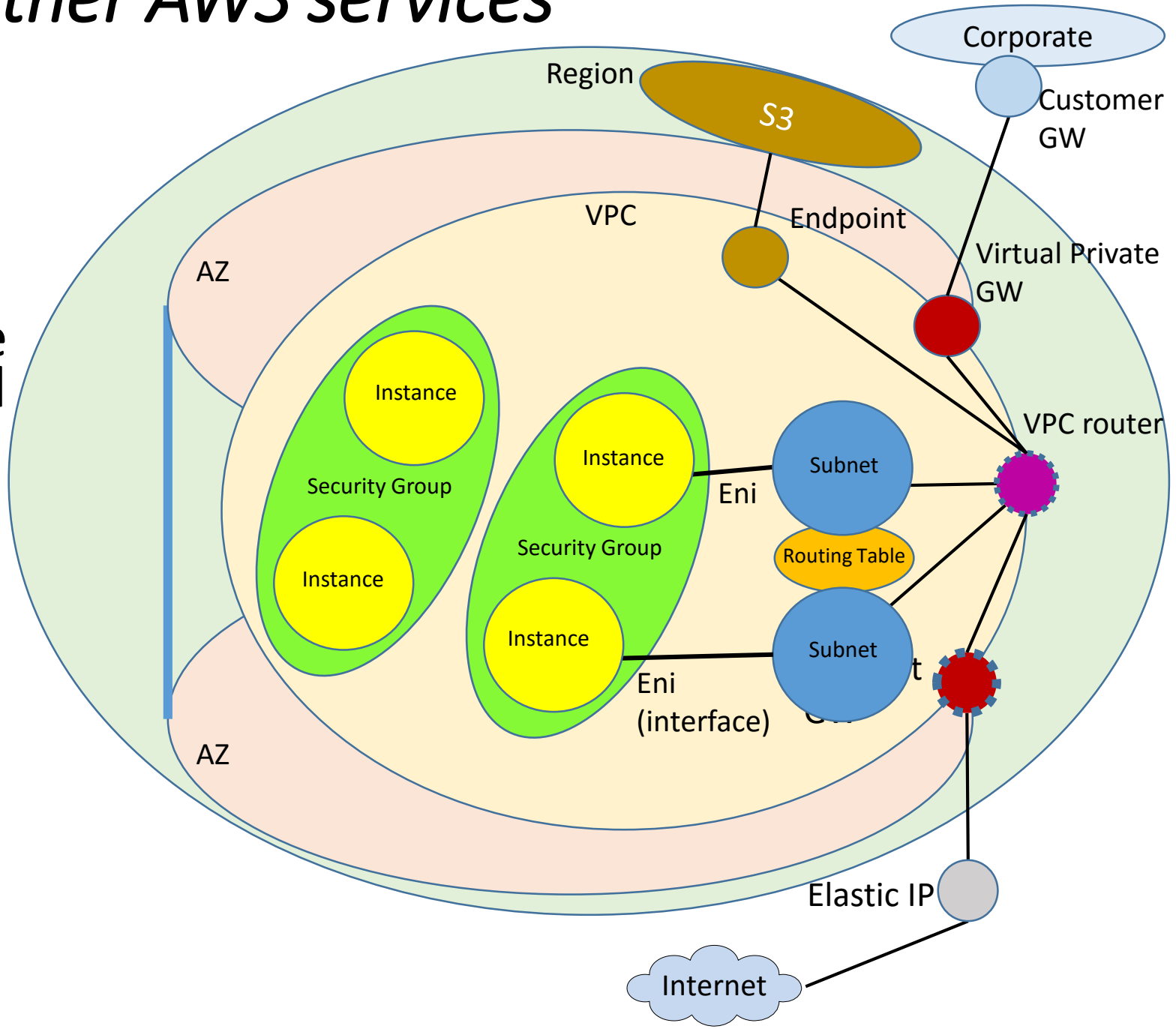
- Types – Classic and application
- Classic is always Internet exposed
- Application LB can be internal
- ELB always sends traffic to private IP backends
- Application ELB can send traffic to containers



VPC integration with other AWS services

AWS Simple Storage Service (S3) -

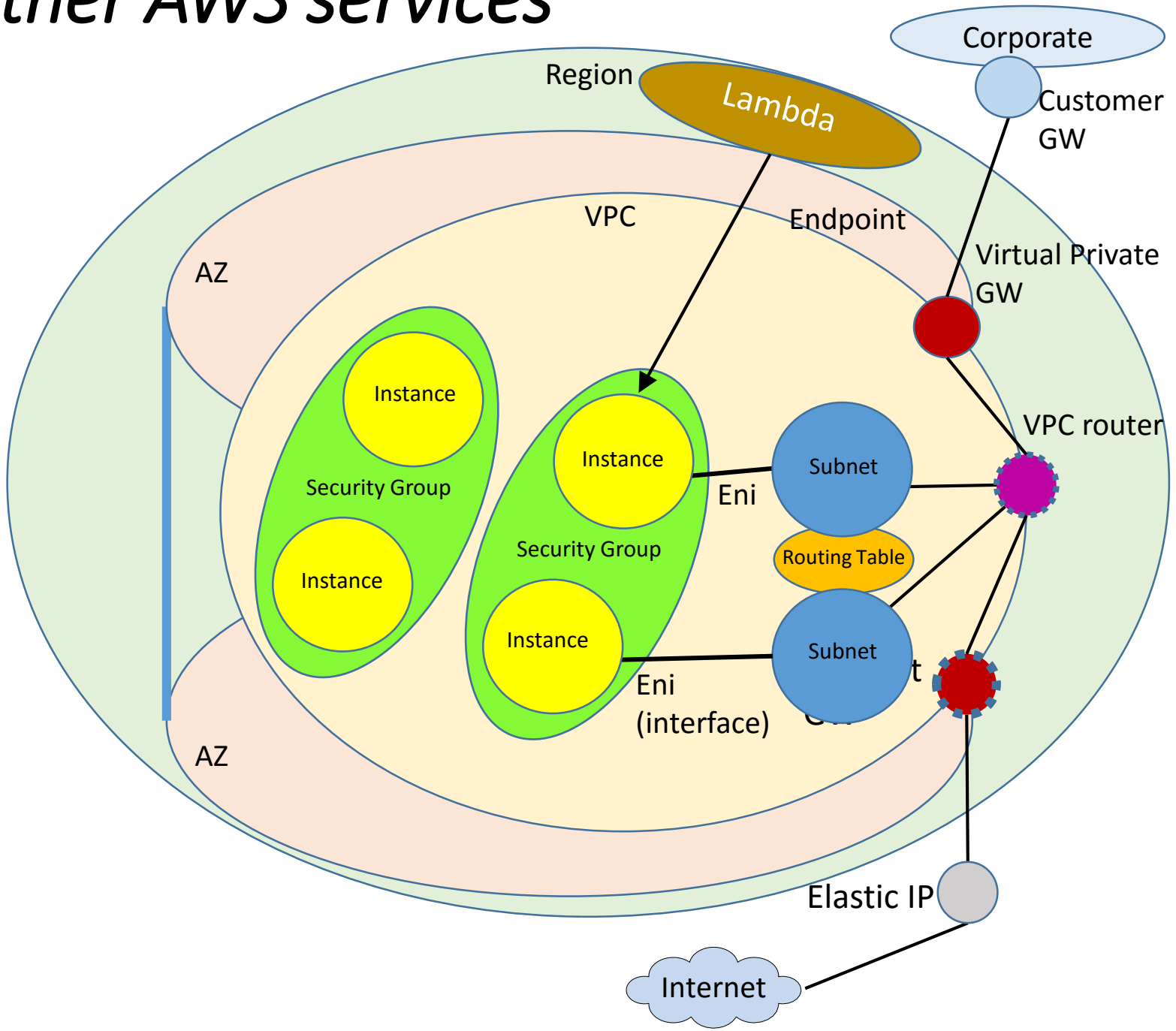
- Opened to the internet
- Data never spans multiple regions unless transferred
- Data spans multiple AZs
- Connected to VPC via a special endpoint
- The endpoint considered and interface in the routing table
- Only subnets connected to the relevant routing table can use the endpoint



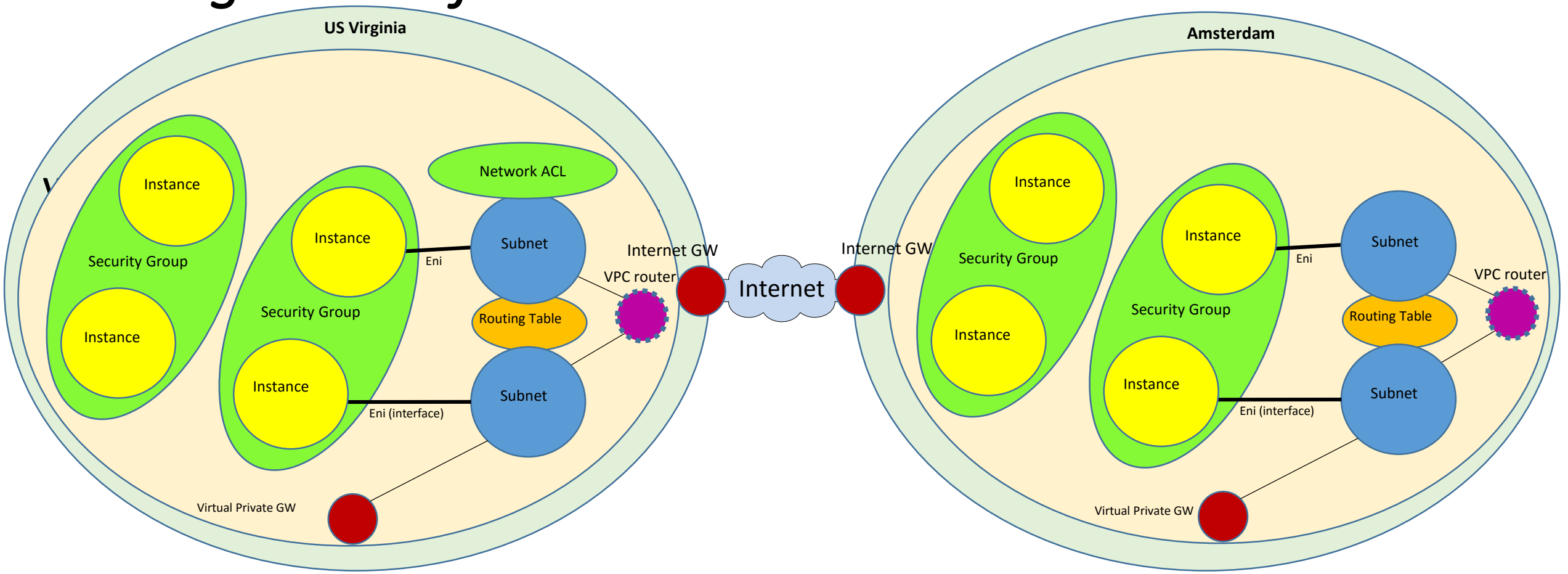
VPC integration with other AWS services

Lambda –

- Service that runs a selected customer's code
- Runs over a container located in AWS own compute resources
- Initiate traffic from an IP outside of the VPC
- Single Lambda function can access only one VPC
- Traffic from Lambda to endpoints outside the VPC should be explicitly allowed on the VPC



Inter-region interface



- Complete isolation
- Only through the internet over VPN connection

Networking the containerized environment

Major Concepts

Containerized applications networking
What are we looking for?

- Service discovery – automated reachability knowledge sharing between networking components
- Deployment – standard and simple. No heavy network experts involvement.
- Data plane – direct access (no port mapping), fast and reliable
- Traffic type agnostic – multicast, IPv6
- Network features - NAT, IPAM, QoS
- Security features - micro-segmentation, access-control, encryption…
- Public cloud ready – Multi VPC and AZs support, overcome route table, costs.
- Public cloud agnostic - dependency on the provider’s services – as minimal as possible

Three concepts around

- **Overlay** – A virtual network decoupled from the underlying physical network using a tunnel (most common - VXLAN)

- **Underlay** – attaching to the physical node's network interfaces

- **Native L3 routing** - L3 routing, advertising containers/pod networks to the network. No overlay

Overlay only approach

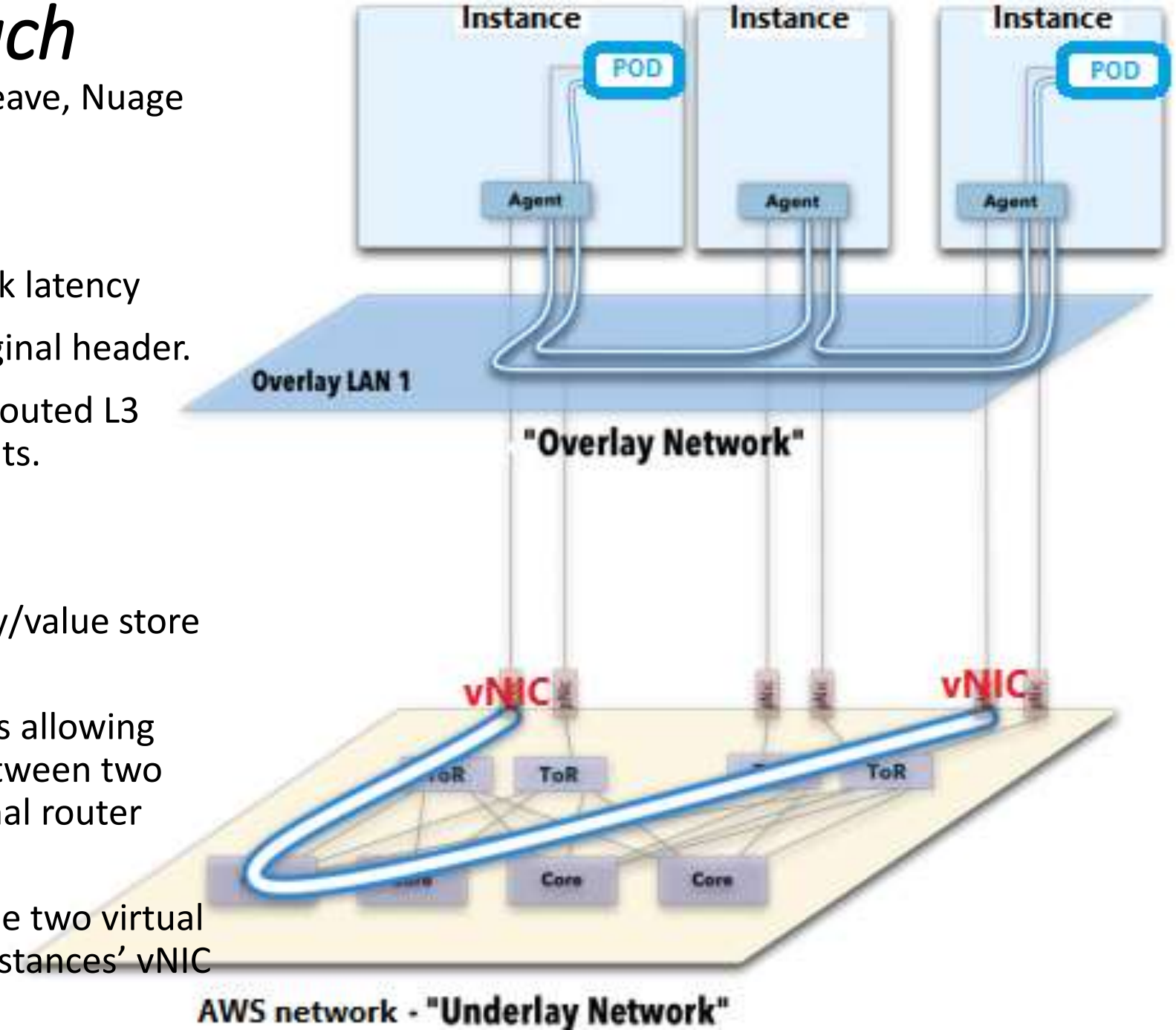
Implementations – Flannel, Contiv, Weave, Nuage

Data Plane –

- Underlying network transparency
- Via kernel space – much less network latency
- **Overhead** - adds 50 bytes to the original header.
- **Traffic** agnostic- Passes direct L2 or routed L3 traffic between two isolated segments. IPv4/IPV6/Multicast

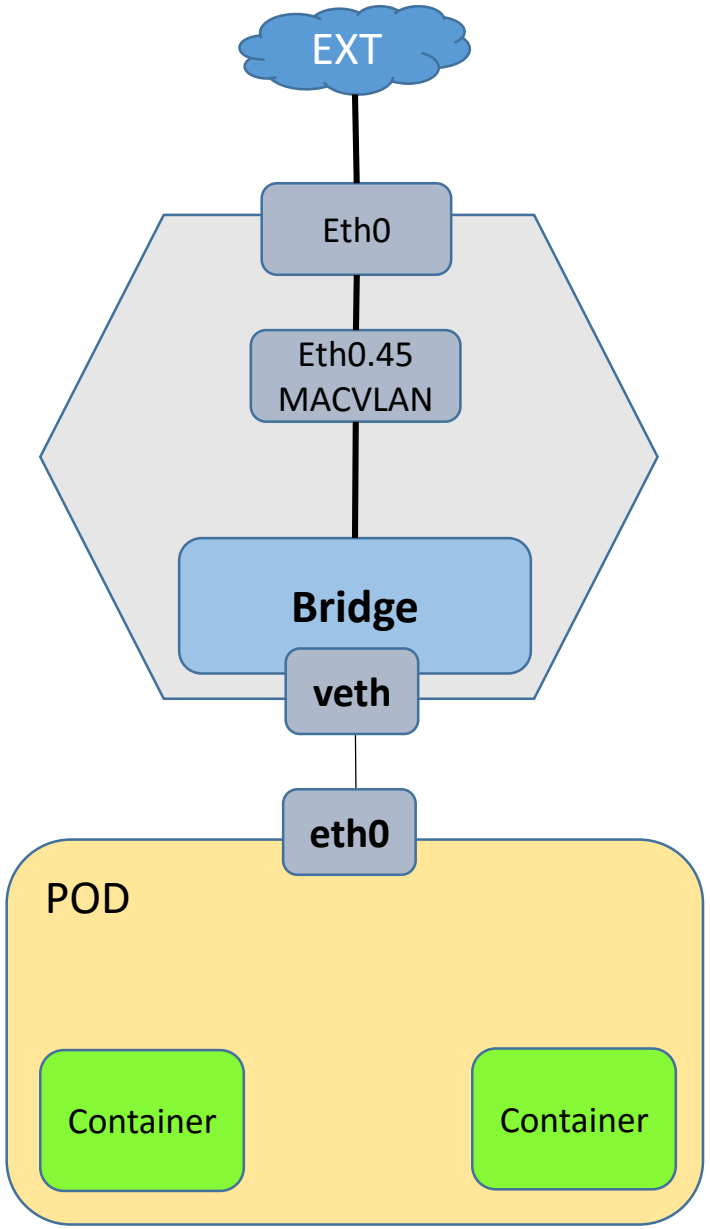
Control plane –

- **Service and network discovery** – key/value store (etcd, Consul …)
- **VNI field** - identifies layer 2 networks allowing isolation between them. Routing between two separate L3 networks – via an external router (VXLAN aware)
- **VTEP** (VXLAN tunnel endpoints) – The two virtual interfaces terminating the tunnel. Instances' vNIC



Underlay - MACVLAN

- Attaches L2 network to the node's physical interface by creating a sub-interface
- Each of the sub-interfaces use different MAC
- Pod belongs to the attached network will be directly exposed to the underlying network w/o port mapping or overlay
- Bridge mode - most commonly used, allows pods, containers or VMs to internally interconnect - traffic doesn't leave the host
- AWS –
  - Disable IP src/dst check
  - Promiscuous mode on the parent nic
  - Verify MAC address per NIC limitation



Native L3 only approach

Implementation – Calico, Romana

Data Plane –

- No overlays – direct container to container (or pod) communications using their real IP addresses, leveraging routing decisions made by container hosts and network routers (AWS route table)

Control plane –

- Containers/Pods/Services IPs being published to the network using routing protocol such as BGP
- Optional BGP peering – between containers nodes for inter-container communications and/or with upstream router for external access
- Large scale – route-reflector implementation may be used
- Due to the L3 nature, native IPv6 is supported
- Optionally NAT is supported for outgoing traffic

Networking models - Comparison

Category\Model	Overlay	L3 routing	Comments
Simple to deploy	Yes	No	L3 BGP requires routing config
Widely used	Yes	No	VXLAN – supported by most plugins
Traffic type agnostic	Yes	Yes*	*Driver support dependent
Allows IP duplication	Yes	No	L3 need address management
Public Cloud friendly	Yes	No	L3 – requires special config on AWS routing tables* HA - two different AZ's subnets still requires tunneling**
Host local routing	No	Yes	Inter-subnet routing on same host goes out External plugins – overcome – split routing
Underlying network independency	Yes	No	L3 needs BGP peering config for external comm.
Performance	Yes*	Yes	*Depends on data path – user or kernel space
Network Efficiency	No	Yes	Overlay adds overhead

Common Implementation Concepts

- The majority of plugins – combine overlay (mostly VXLAN) and L3
- Subnet allocated per node (Nuage is an exception)
- Based on agent installed on the node (project proprietary or Open vSwitch)
- Local routing on the node between different subnets
- Support routing to other nodes (needs L2 networks between nodes)
- Public clouds integration provided for routing table update (limited comparing to standard plugins)
- Performance - Data path in Kernel space
- Distributed or policy based (SDN)

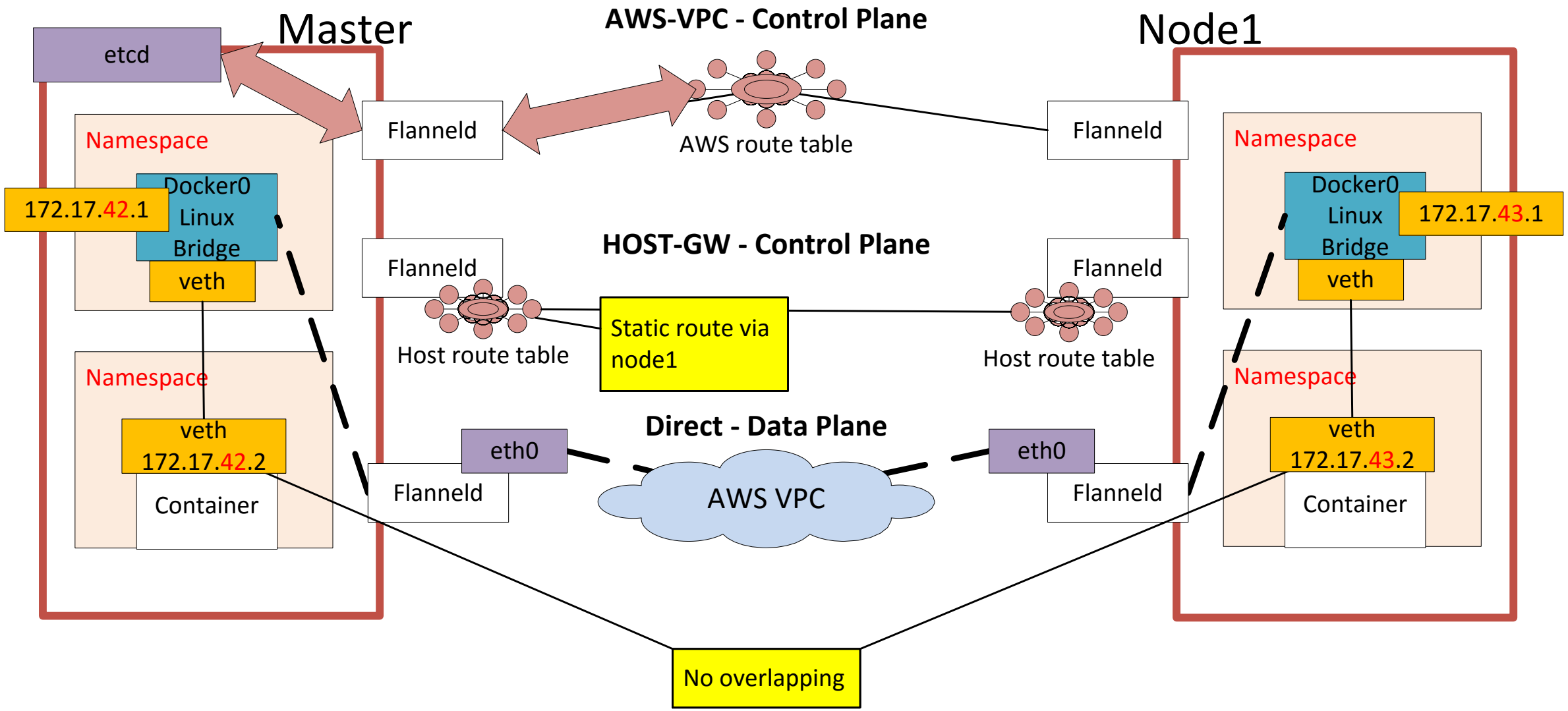
Flannel (CoreOS) – The proprietary example

- **Used for dual OVS scenarios (Openstack)**
- Flanneld agent on the node – allocates a subnet to the node and register it in the etcd store installed on each node
- No Security policy currently supported - A new project, Canal, combines Flannel and Calico for a whole network and security solution
- Subnet cannot span multiple hosts

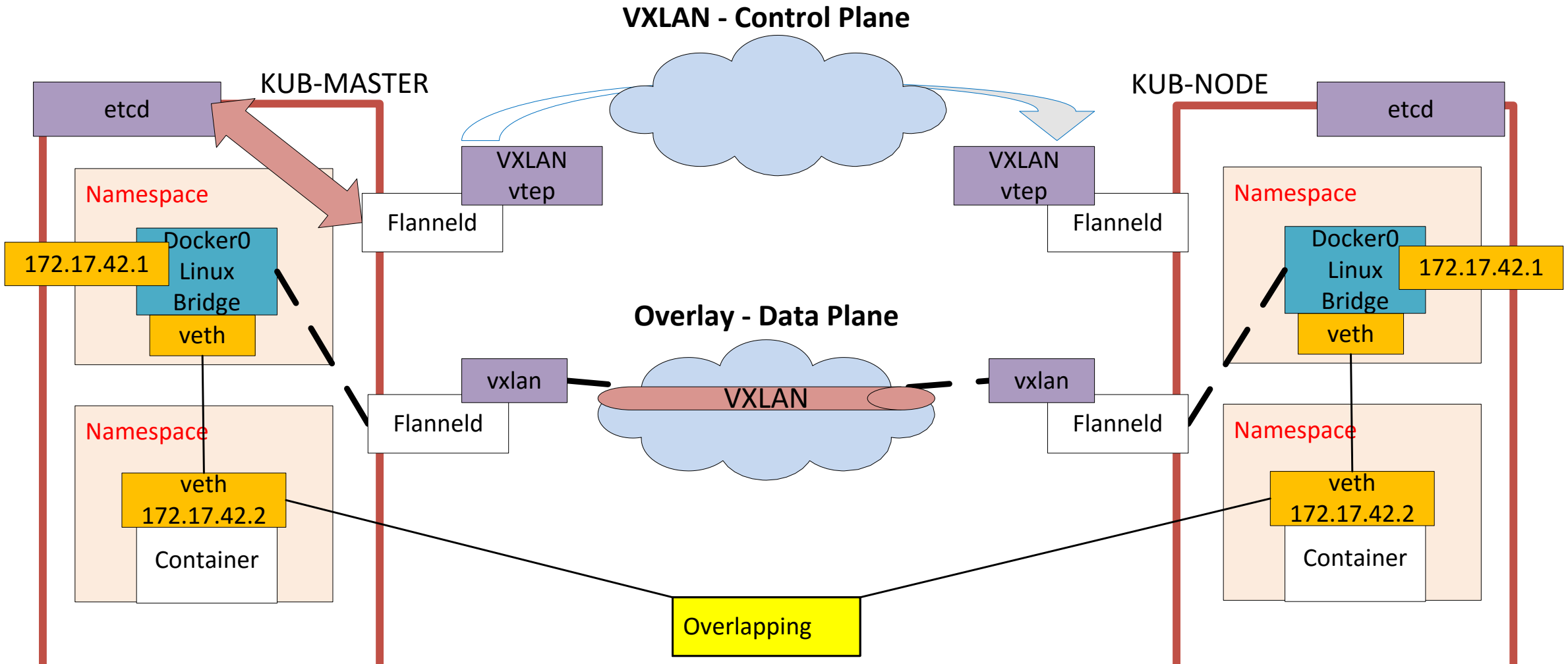
Three implementations:

- Overlay – UDP/VXLAN – etcd used for control plane
- Host-gw – direct routing over L2 network with routing on the node's routing table – Can be used in AWS also – preforms faster
- AWS-VPC – direct routing over AWS VPC routing tables. Dynamically updates the AWS routing table (50 route entries limits in a routing table. If more needed, VXLAN can be used).

Flannel **Direct**



Flannel **OVERLAY**



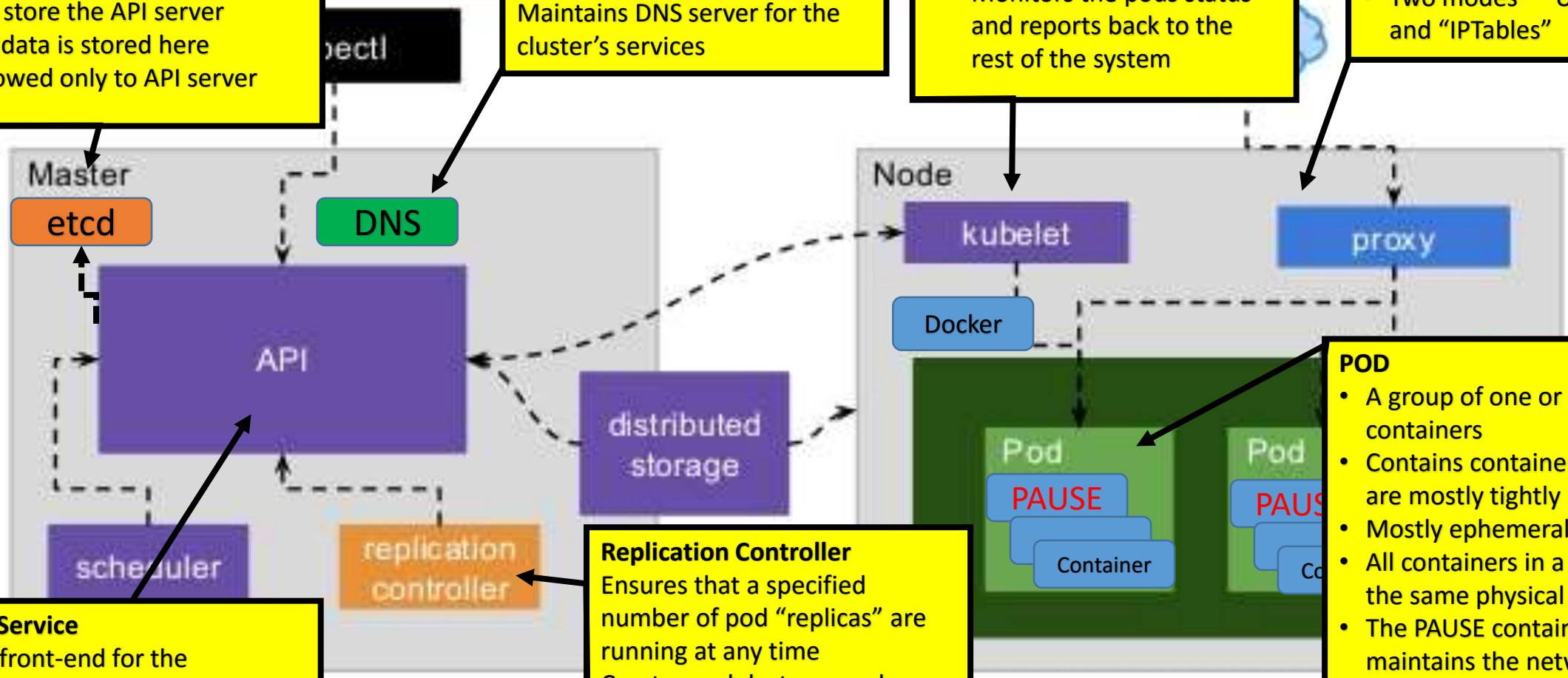
OpenShift Networking over Kubernetes

OVS-based solution

Concepts

Implementation alternatives

Kubernetes architectu



etcd
Key/Value store the API server
All cluster data is stored here
Access allowed only to API server

DNS
Maintains DNS server for the cluster's services

Kubelet
- Watches for pods scheduled to node and mounts the required volumes
- Manages the containers via Docker
- Monitors the pods status and reports back to the rest of the system

KUBE-PROXY
- Assigns a listening port for a service
- Listen to connections targeted to services and forwards them to the backend pod
- Two modes – "Userspace" and "IPTables"

API Service
The front-end for the Kubernetes control plane. It is designed to scale horizontally

Replication Controller
Ensures that a specified number of pod "replicas" are running at any time
Creates and destroys pods dynamically

POD
- A group of one or more containers
- Contains containers which are mostly tightly related
- Mostly ephemeral in nature
- All containers in a pod are in the same physical node
- The PAUSE container maintains the networking

So why not Docker's default networking?

- **Non-networking reason** - drivers integration issues and low level built-in drivers (at least initially)
- **Scalability (horizontality)** – Docker's approach to assign IPs directly to containers limits scalability for production environment with thousands of containers. Containers network footprint should be abstracted
- **Complexity** – Docker's port mapping/NAT requires messing with configuration, IP addressing management and applications' external port coordination
- **Nodes resource and performance limitation** – Docker's port mapping might suffer from ports resource limitations. In addition, extra processing required on the node
- **CNI model** was preferred over the CNM, because of the container access limitation

Kubernetes native networking

- IP address allocation – IP give to pods rather that to containers
- Intra-pod containers share the same IP
- Intra-pod containers use localhost to inter-communicate
- Requires direct multi-host networking without NAT/Port mapping
- Kubernetes doesn't natively give any solution for multi-host networking. Relies on third party plugins: Flannel, Weave, Calico, Nuage, OVS etc.
- Flannel was already discussed previously as an example to overlay networking approach
- OVS will be discussed later as OVS based networking plugins
- Nuage solution will be separately dicussed

Kubernetes - Pod

When POD is created with containers, the following happens:

- "PAUSE" container created –
  - "pod infrastructure" container – minimal config
  - Handles the networking by holding the networking namespace, ports and IP address for the containers on that pod
  - The one that actually listens to the application requests
  - When traffic hits, it's redirected by IPTABLES to the container that listens to this port
- "User defined" containers created –
  - Each use "mapped container" mode to be linked to the PAUSE container
  - Share the PAUSE's IP address

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    deployment: docker-registry-1
    deploymentconfig: docker-registry
    docker-registry: default
  generateName: docker-registry-1-
spec:
  containers:
  - env:
    - name: OPENSHIFT_CA_DATA
      value: ...
    - name: OPENSHIFT_MASTER
      value: https://master.example.com:8443
  ports:
    - containerPort: 5000
      protocol: TCP
    resources: {}
    securityContext: { ... }
  dnsPolicy: ClusterFirst
```

Kubernetes - Service

- Abstraction which defines a logical set of pods and a policy by which to access them
- Mostly permanent in nature
- Holds a virtual IP/ports used for client requests (internal or external)
- Updated whenever the set of pods changes
- Use labels and selector for choosing the backend pods to forward the traffic to

When a service is created the following happens:

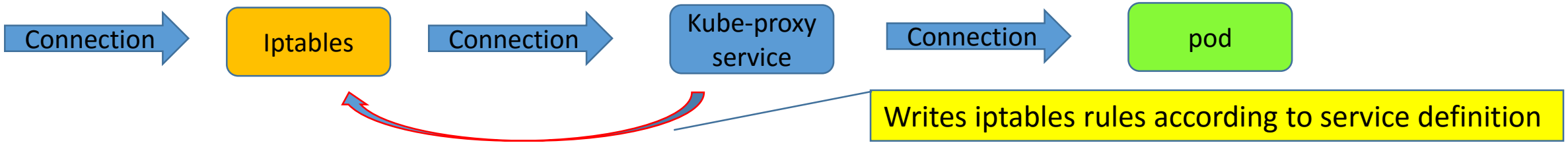
- IP address assigned by the IPAM service
- The kube-proxy service on the worker node, assigns a port to the new service
- Kube-proxy generates iptables rules for forwarding the connection to the backend pods
- Two Kube-proxy modes….

```
apiVersion: v1
kind: Service
metadata:
  name: docker-registry
spec:
  selector:
    docker-registry: default
  portalIP: 172.30.136.123
  ports:
  - nodePort: 0
    port: 5000
    protocol: TCP
    targetPort: 5000
```

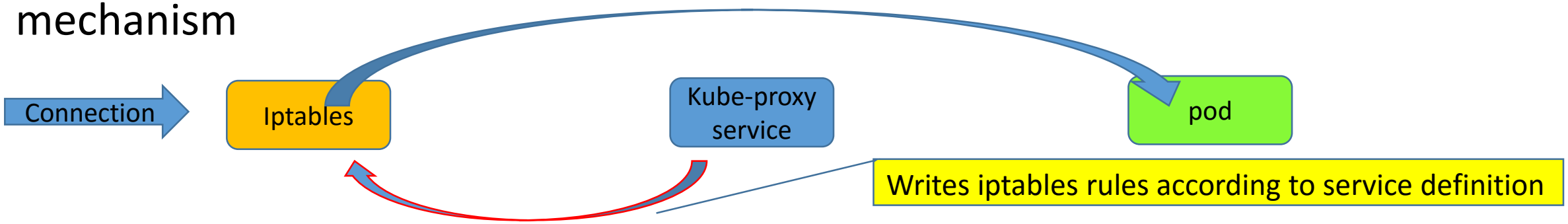
Kube-Proxy Modes

Kube-Proxy always writes the Iptables rules, but what actually handles the connection?

Userspace mode – Kube-proxy is the one that forwards connections to backend pods. Packets move between user and kernel space which adds latency, but the application continues to try till it finds a listening backend pod. Also debug is easier



Iptables mode – Iptables from within the kernel, directly forwards the connection to the pod. Fast and efficient but harder to debug and no retry mechanism

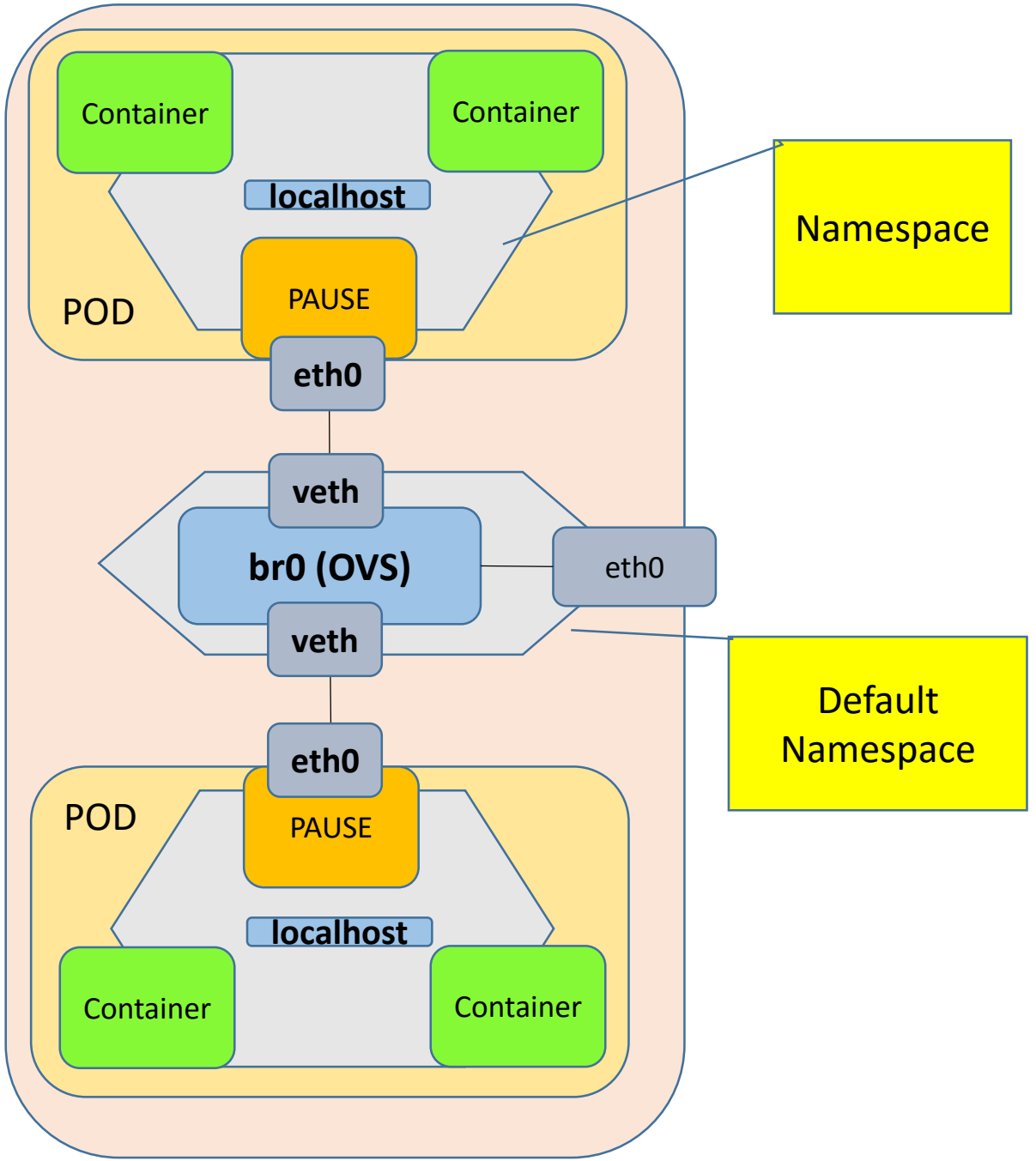


OpenShift SDN - Open Vswitch (OVS) – The foundation

- A multi-Layer open source virtual switch
- Doesn't support native L3 routing need the Linux kernel or external componenet
- Allows network automation through various programmatic interfaces as well as built-in CLI tools
- Supports:
- Port mirroring
- LACP port channeling
- Standard 802.1Q VLAN trunking
- IGMP v1/2/3
- Spanning Tree Protocol, and RTSP
- QoS control for different applications, users, or data flows
- port level traffic policing
- NIC bonding, source MAC addresses LB, active backups, and layer 4 hashing
- OpenFlow
- Full IPv6 support
- kernel space forwarding
- GRE, VXLAN and other tunneling protocols with additional support for outer IPsec

Linux Network Namespace

- Logically another copy of the **network** stack, with its own routes, firewall rules, and **network** devices
- Initially all the processes share the same default **network namespace** from the parent host (init process)
- A pod is created with a “host container” which gets its own network namespace and maintains it
- “User containers” within that pod join that namespace

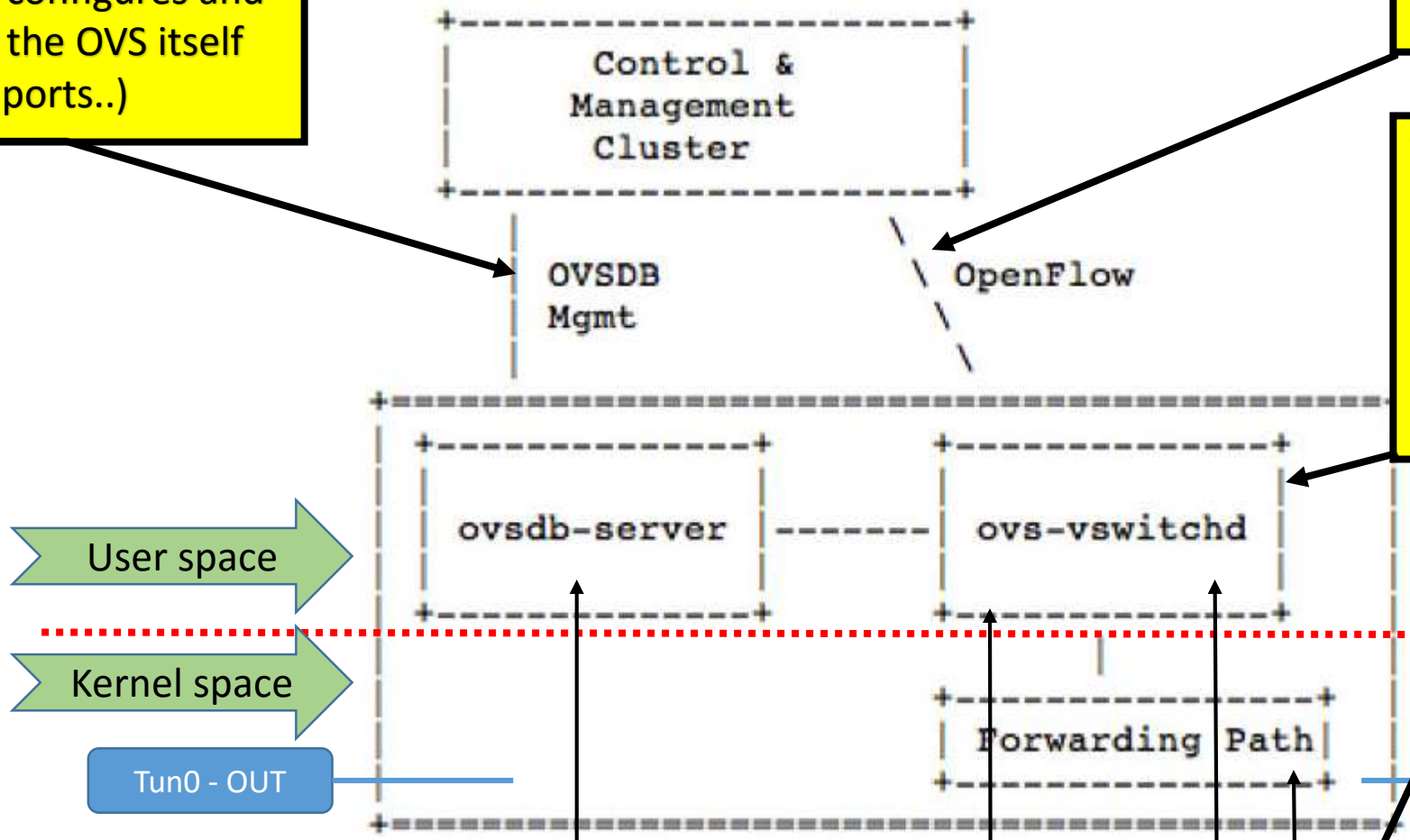


OpenShift SDN - OVS Management

OVSDB – configures and monitors the OVS itself (bridges, ports..)

OpenFlow – programs the OVS daemon with flow entries for flow-based forwarding

The actual OVS daemon
- Process openflow messages
- Manage the datapath (which actually in kernel space)
- Maintain two flow table (exactly flow & wildcard flow)



Configures and monitors the OVS kernel module

Configures and monitors the ovsdb: bridges, ports, flows

ovs-vsctl

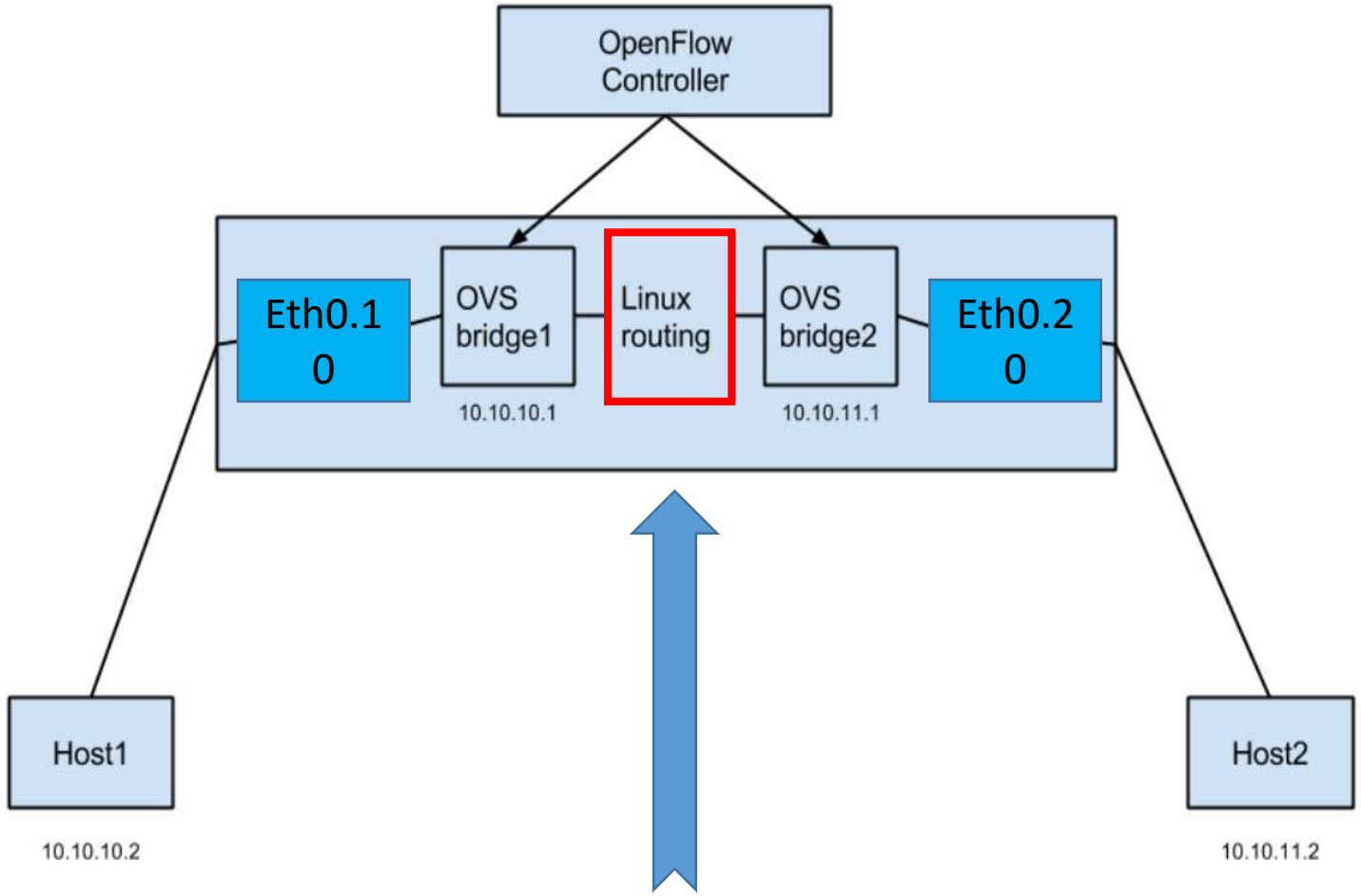
ovs-appctl

ovs-dpctl

Sends commands to OVS daemon
Example: MAC tables

OpenShift SDN – L3 routing

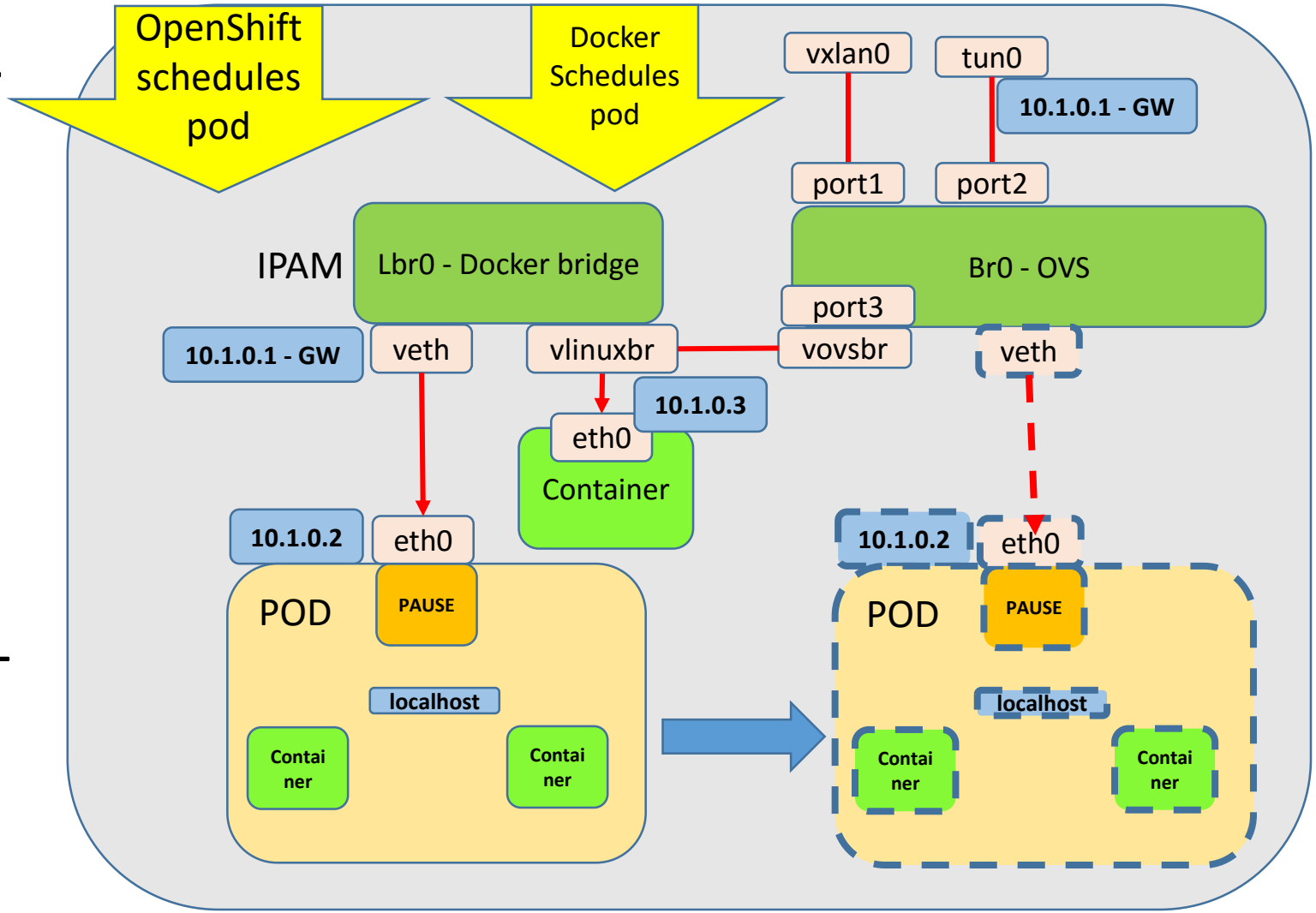
1. OVS doesn't support native L3 routing
2. L3 routing between two subnets done by the parent host's networking stack
3. Steps (one alternative)
   1. Creating two per-VLAN OVS
   2. Creating two L3 sub-interfaces on the parent host
   3. Bridging the two sub-interfaces to both OVS bridges
   4. Activating IP forwarding on the parent host



Note: The L3 routing can be done using plugins such as Flannel, Weave and others

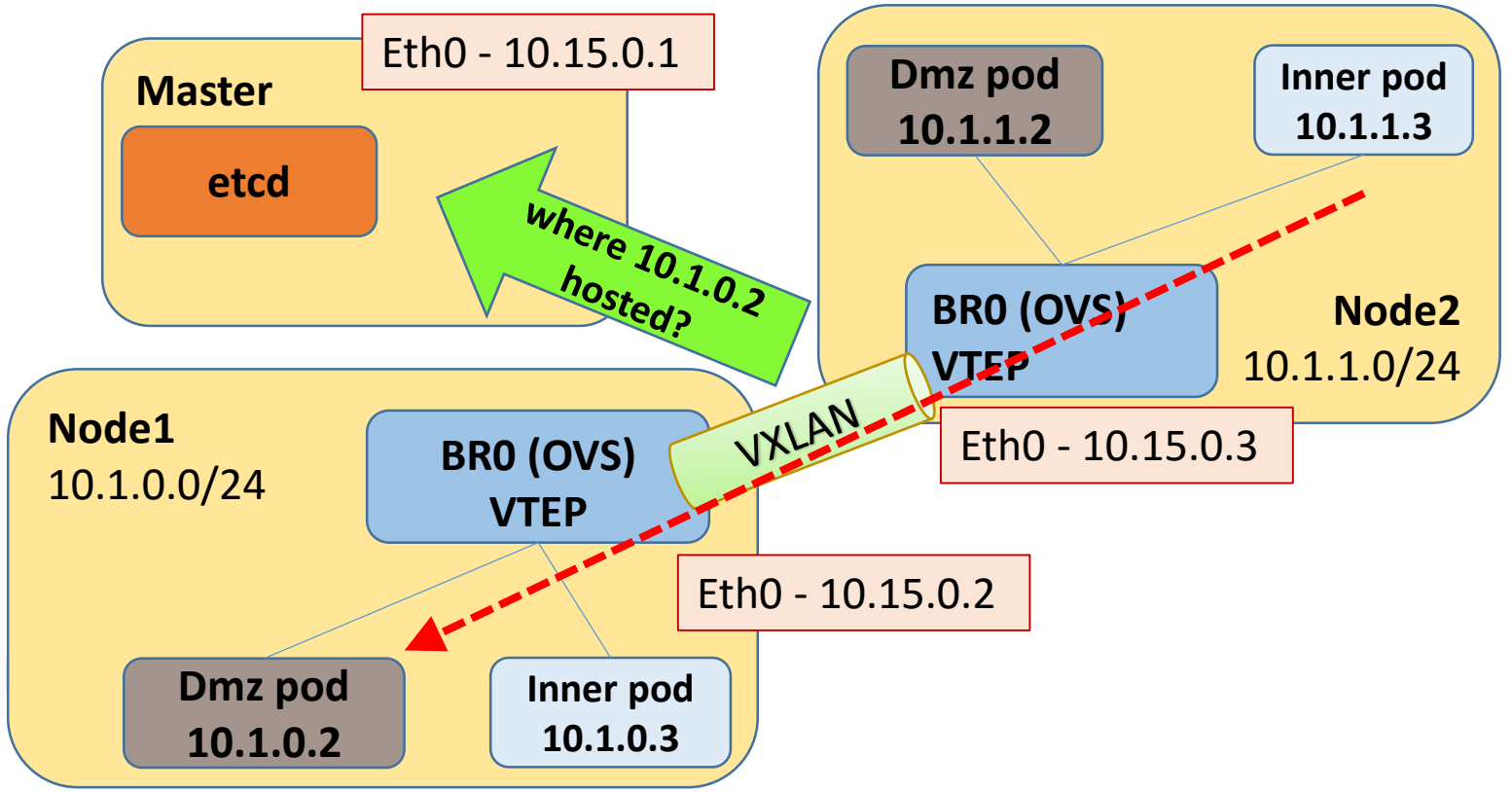
OpenShift SDN – local bridges and interfaces on the host

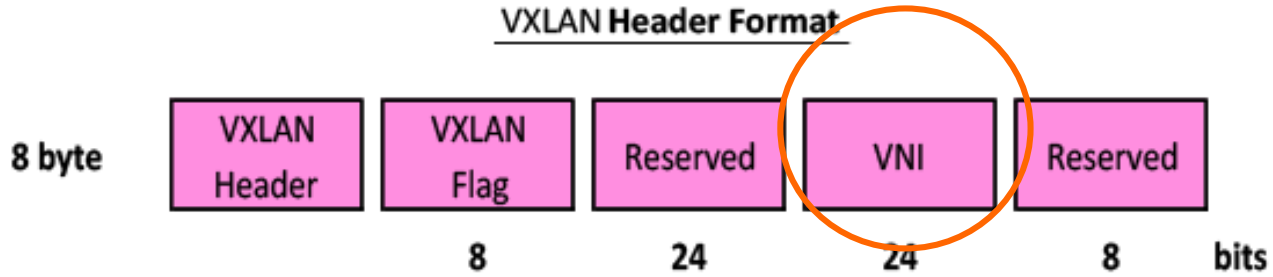
1. Node is registered and given a subnet
2. Pod crated by OpenShift and given IP from Docker bridge
3. Then moved to OVS
4. Container created by Docker engine given IP from Docker bridge
5. Stays connected to lbr0
6. No network duplication – Docker bridge only for IPAM



OpenShift SDN – Overlay

- Control plane – etcd stores information related to host subnets
- Initiated from the node's OVS via the nodes NIC (vtep – lbr0)
- Traffic encapsulated into OVS's VXLAN interface
- When ovs-multitenant driver used – projects can be identified by VNIDs
- Adds 50 bytes to the original frame

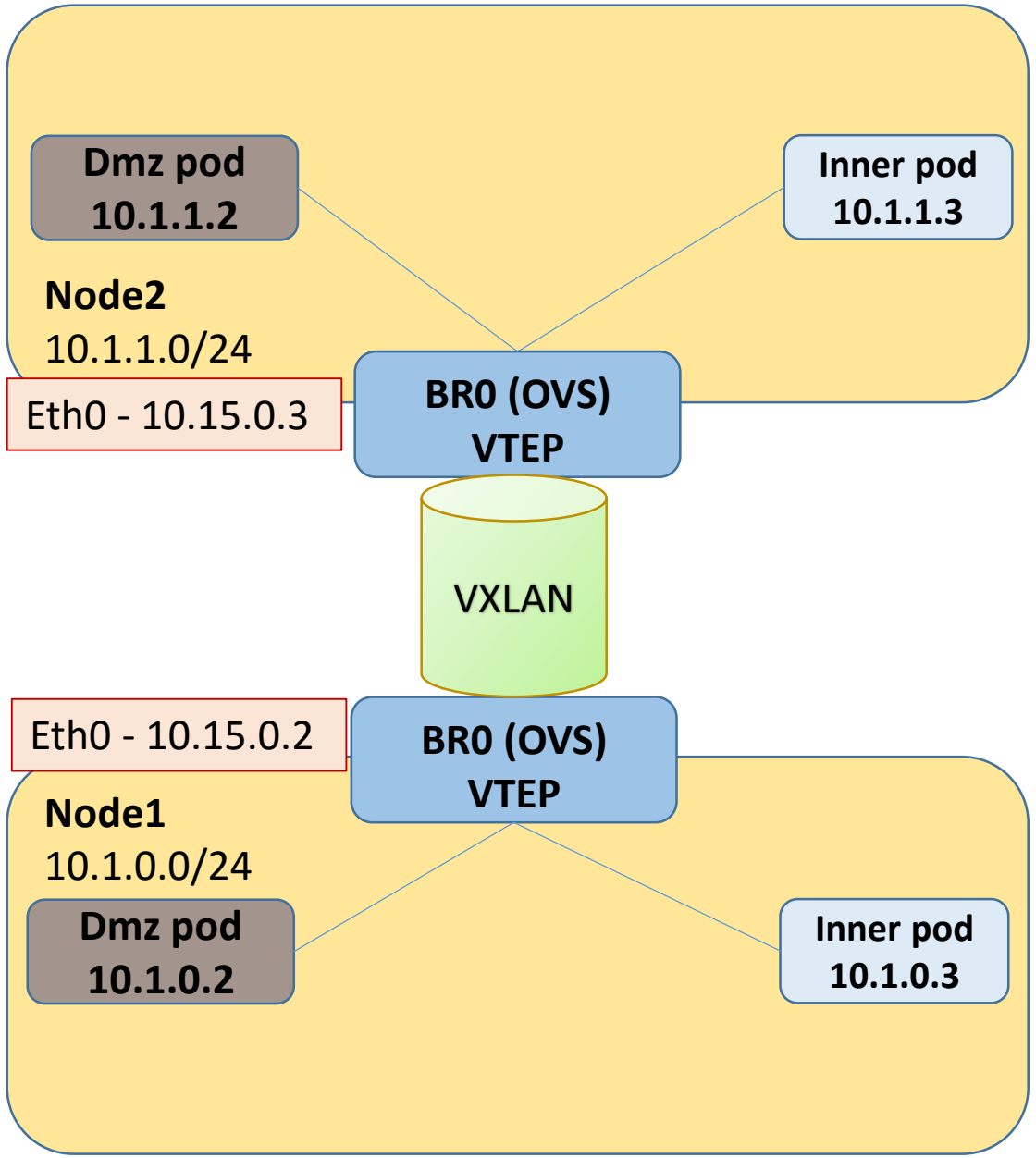




OpenShift SDN – plugin option1

OVS-Subnet – the original driver

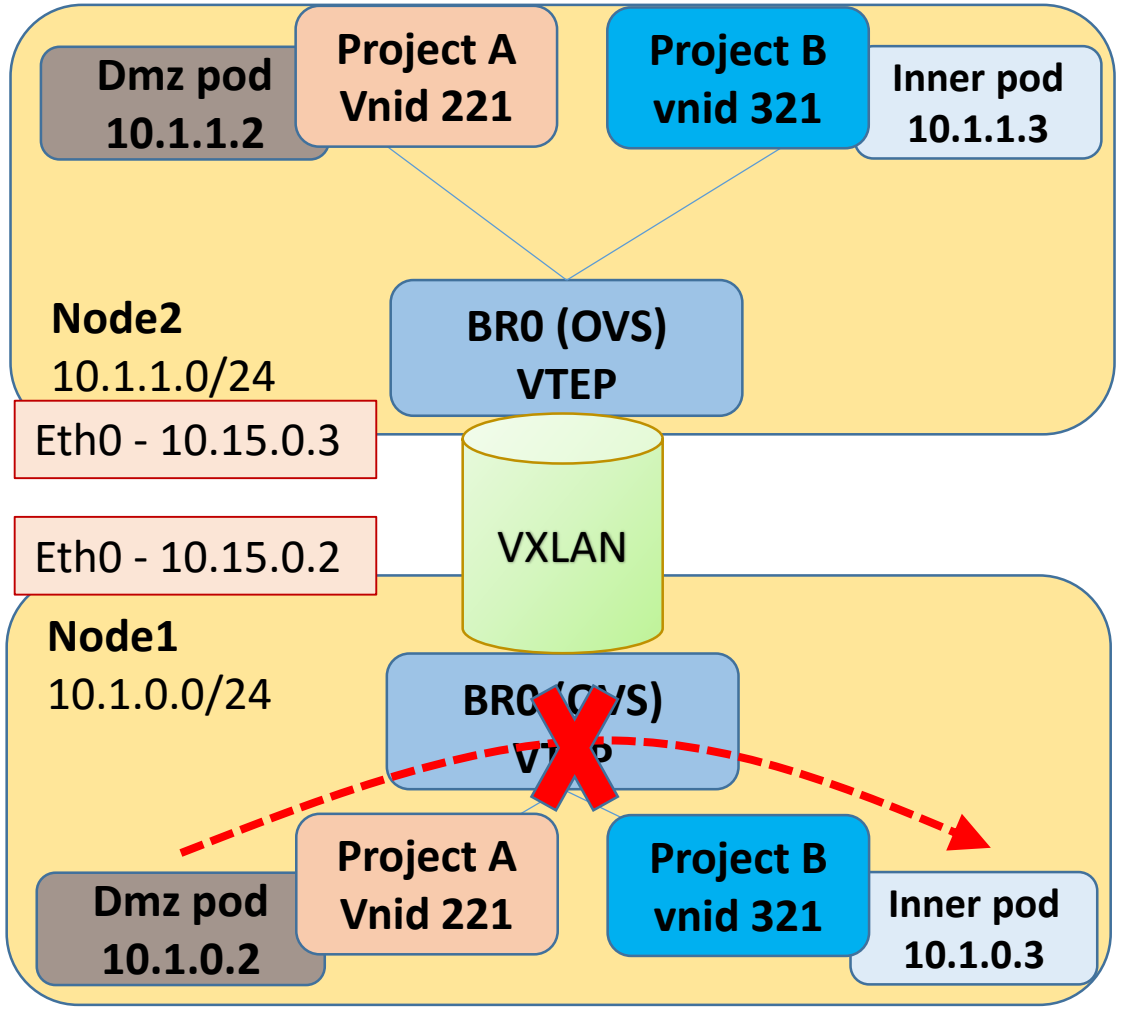
- Creates flat network allows all pod to inter-communicate
- No network segmentation
- Policy applied on the OVS
- No significance to project membership



OpenShift SDN – plugin option 2

OVS-Multitenant –

- Each projects gets a unique VNID – identifies pods in that project
- Default projects – VNID 0 – communicate with all others (Shared services)
- Pods' traffic inspected according to its project membership



OpenShift – service discovery - alternatives

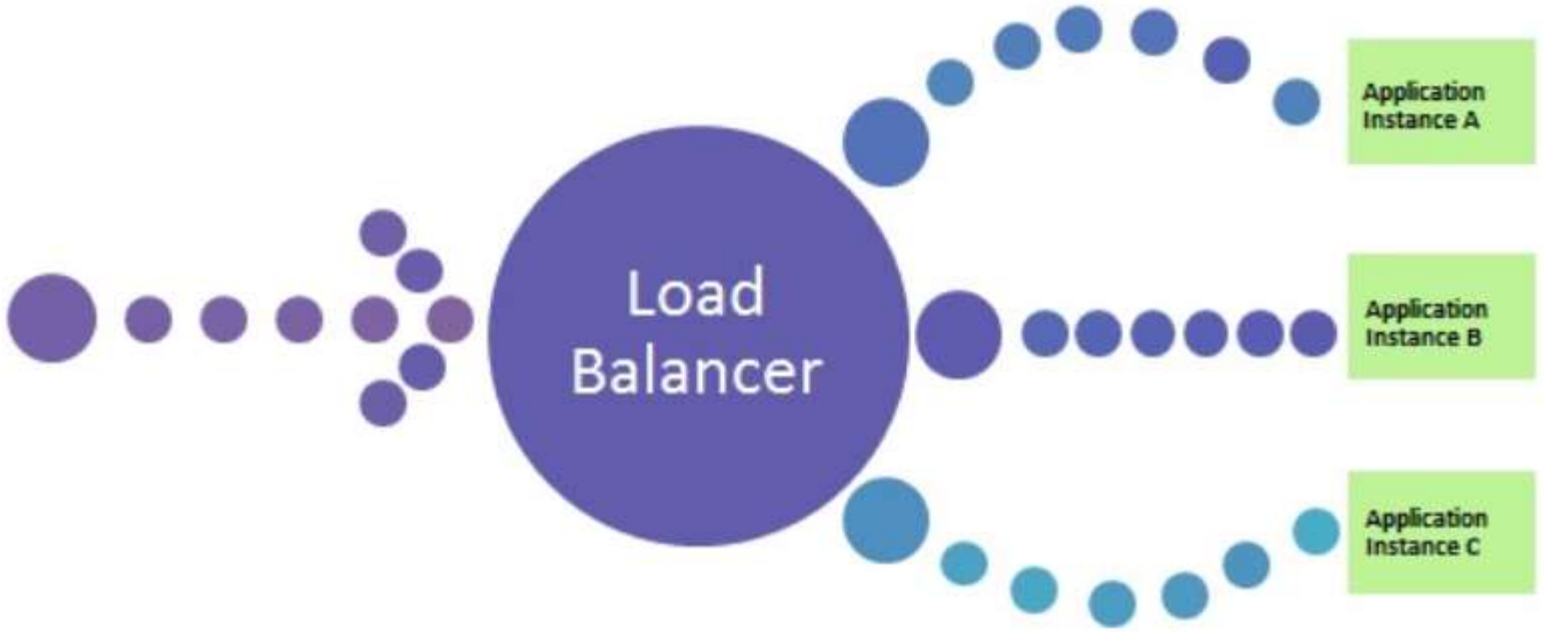
App to App – preferably using Pod-to-Service, avoid Pod-to-Pod

Environment variables –

- Injected to the pod with connectivity info (user names, service IP..)
- For updates, pod recreation is needed
- Destination service must first created (or restarted in case they were created before the pod)
- Not a real dynamic discovery…

DNS – SkyDNZ – serving <*cluster*>*.local* suffixes

- Split DNS - supports different resolution for internal and external
- SkyDNS installed on master and pods are configured by default to use it first
- Dynamic - no need to recreate the pods for any service update
- Newly created services being detected automatically by the DNS
- For direct pod-to-pod connection (no service) – DNS round robin can be used



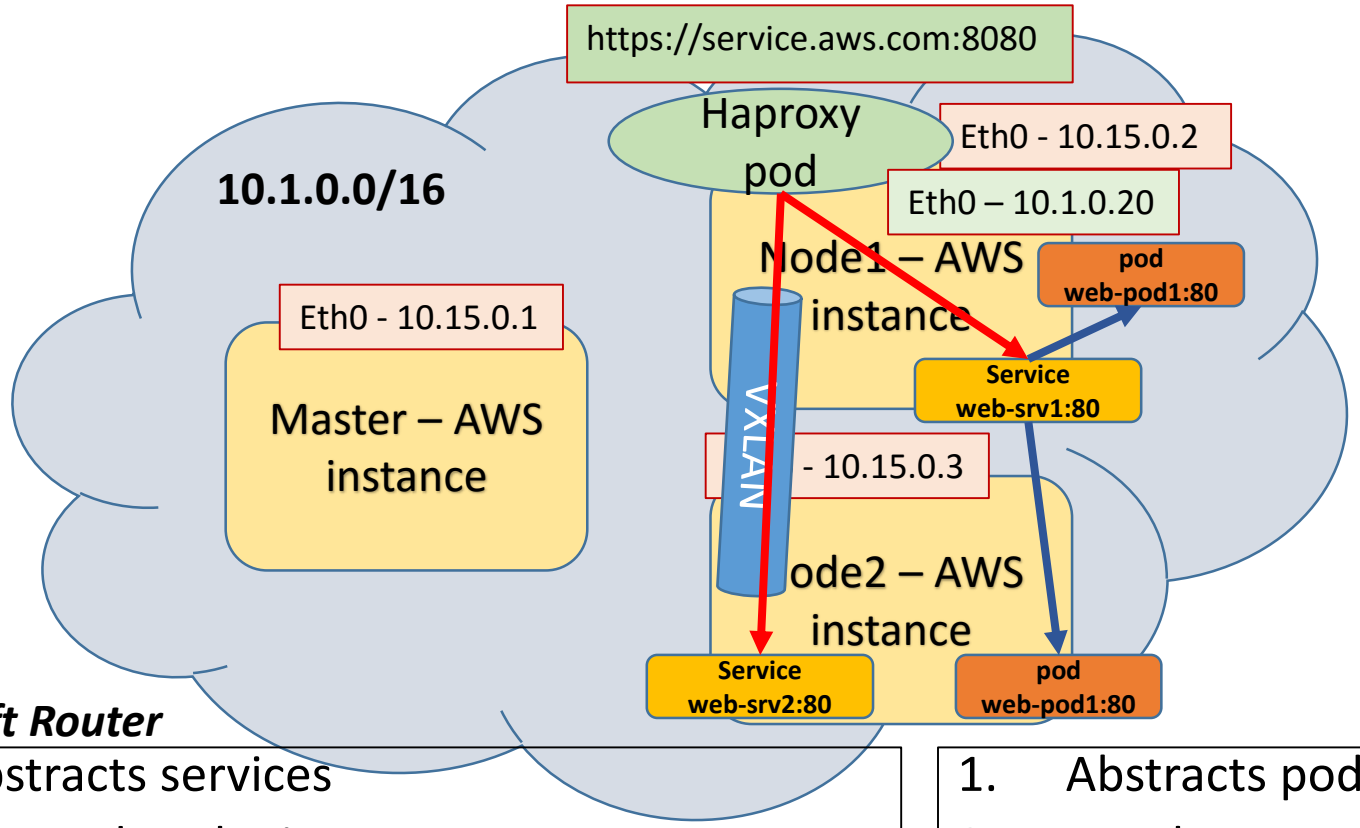
Routing and Load balancing

Requirements

Network discovery

Alternatives

OpenShift Router vs. Kubernetes Service



OpenShift Router

1. Abstracts services
2. Exposed to the internet
3. Kubernetes construct
4. Built-in Haproxy Container or external LB
5. Built-in router supports http/https
6. Independent app
7. Uses its own routing/LB rules for backend-side traffic

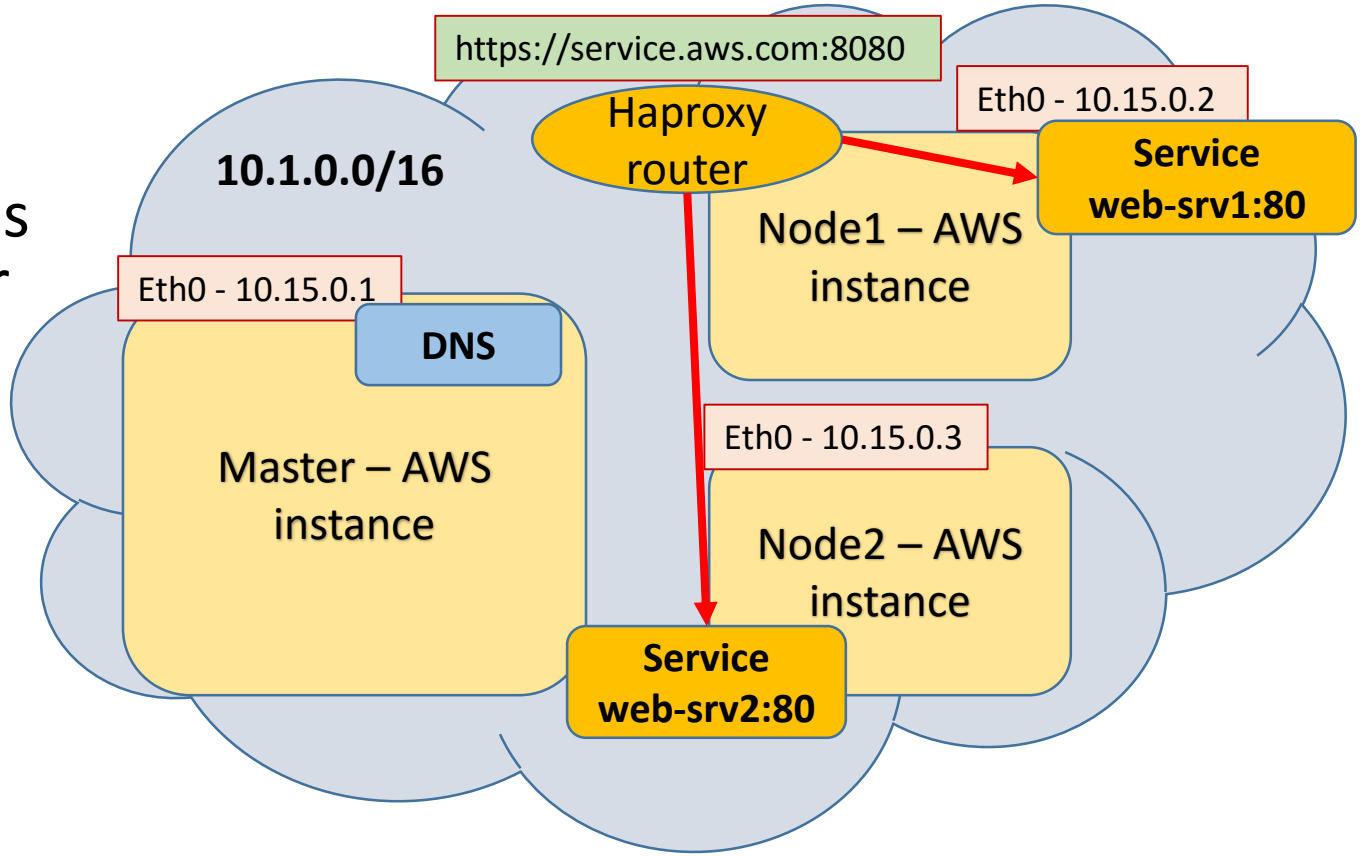
Kubernetes Service

1. Abstracts pods
2. May be exposed to the internet
3. OpenShift construct
4. Installed Kubernetes service on each node
5. Supports all kind of traffic
6. Kubernetes service
7. Uses node's IPTables for destination NAT for backend-side traffic

Routing – alternative 1

OpenShift router –

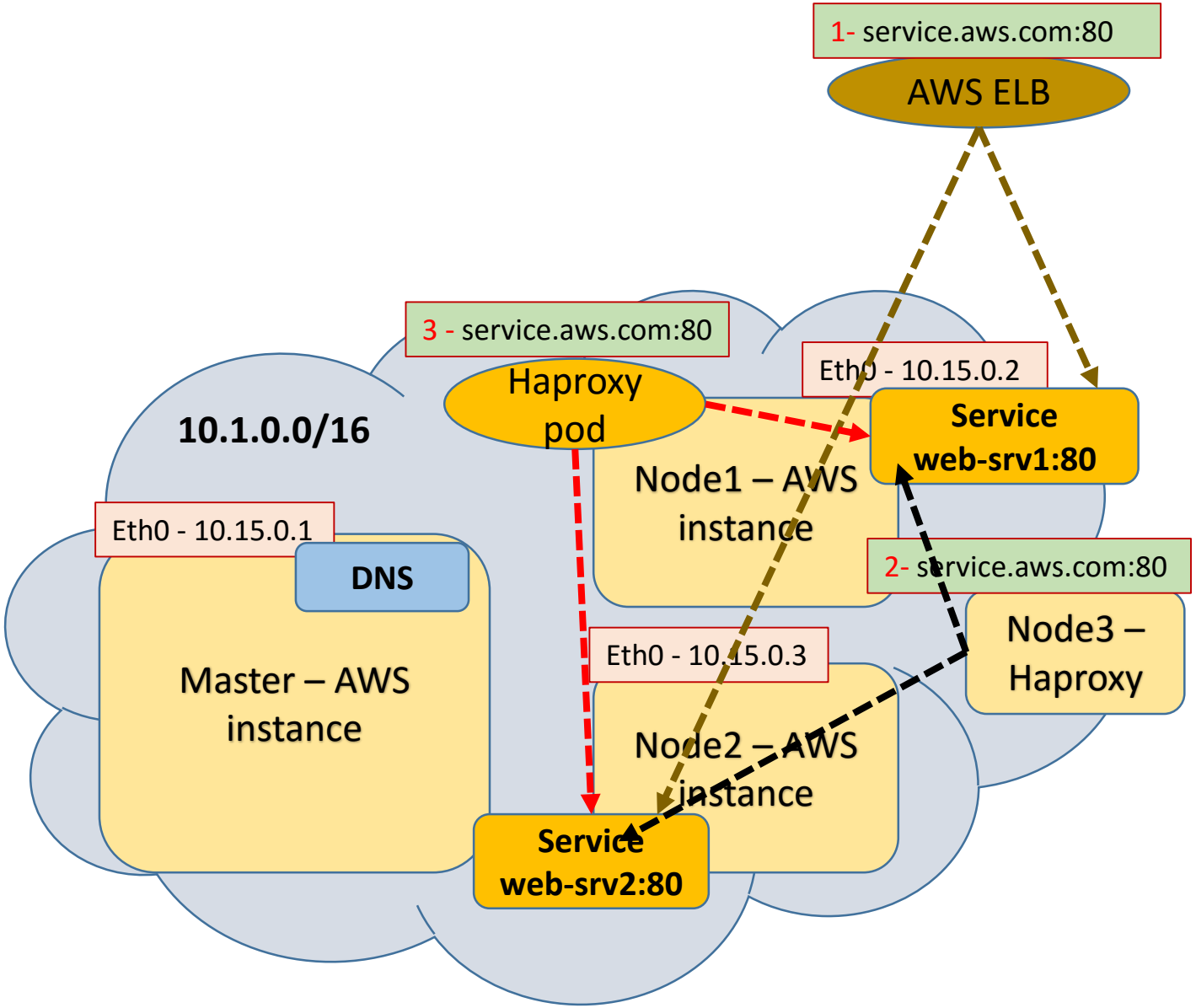
- For WEB apps – http/https
- Managed by users
- Routes created in project level and added to the router
- Unless shared, all routers see all routes
- For traffic to come in, admin needs to add a DNS record for the router or using wildcard
- Default - Haproxy container - listens on the host's IP and proxies traffic to the pods



Routing – alternative 2

Standalone Load Balancer –

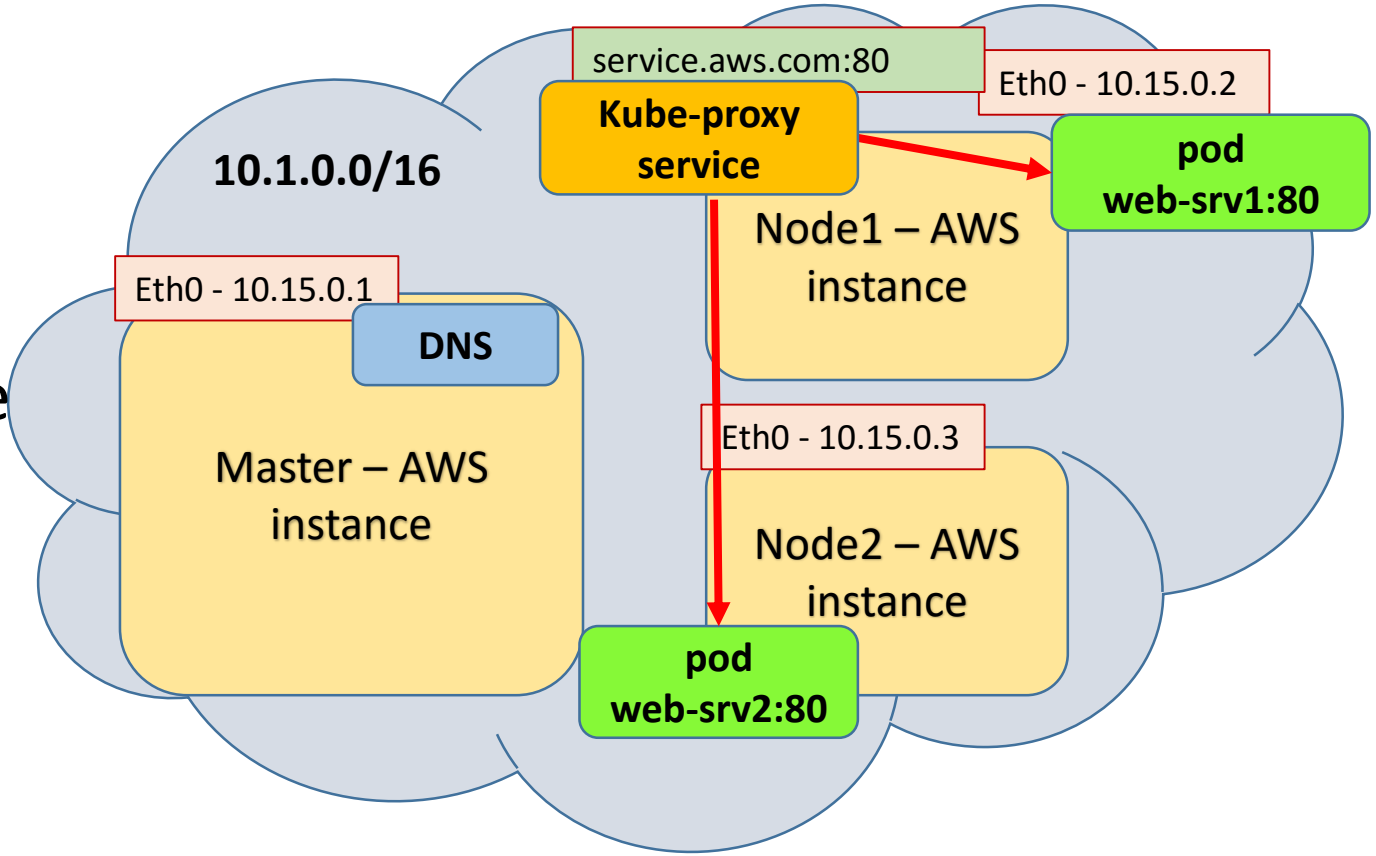
- All traffic types
- Alternatives are:
  1. AWS ELB
  2. Dedicated Cluster node
  3. Customized Haproxy pod
- IP Routing towards the internal cluster network – discussed later



Routing – alternative 3

Service external IP–

- Managed by Kubernetes Kube-Proxy service on each node
- The proxy assigns the IP/port and listens to incoming connections
- Redirects traffic to the pods
- All types of traffic
- Admin should take care of routing traffic towards the node
  - Iptables-based – all pods should be ready listen
  - User space - try all pods till it finds

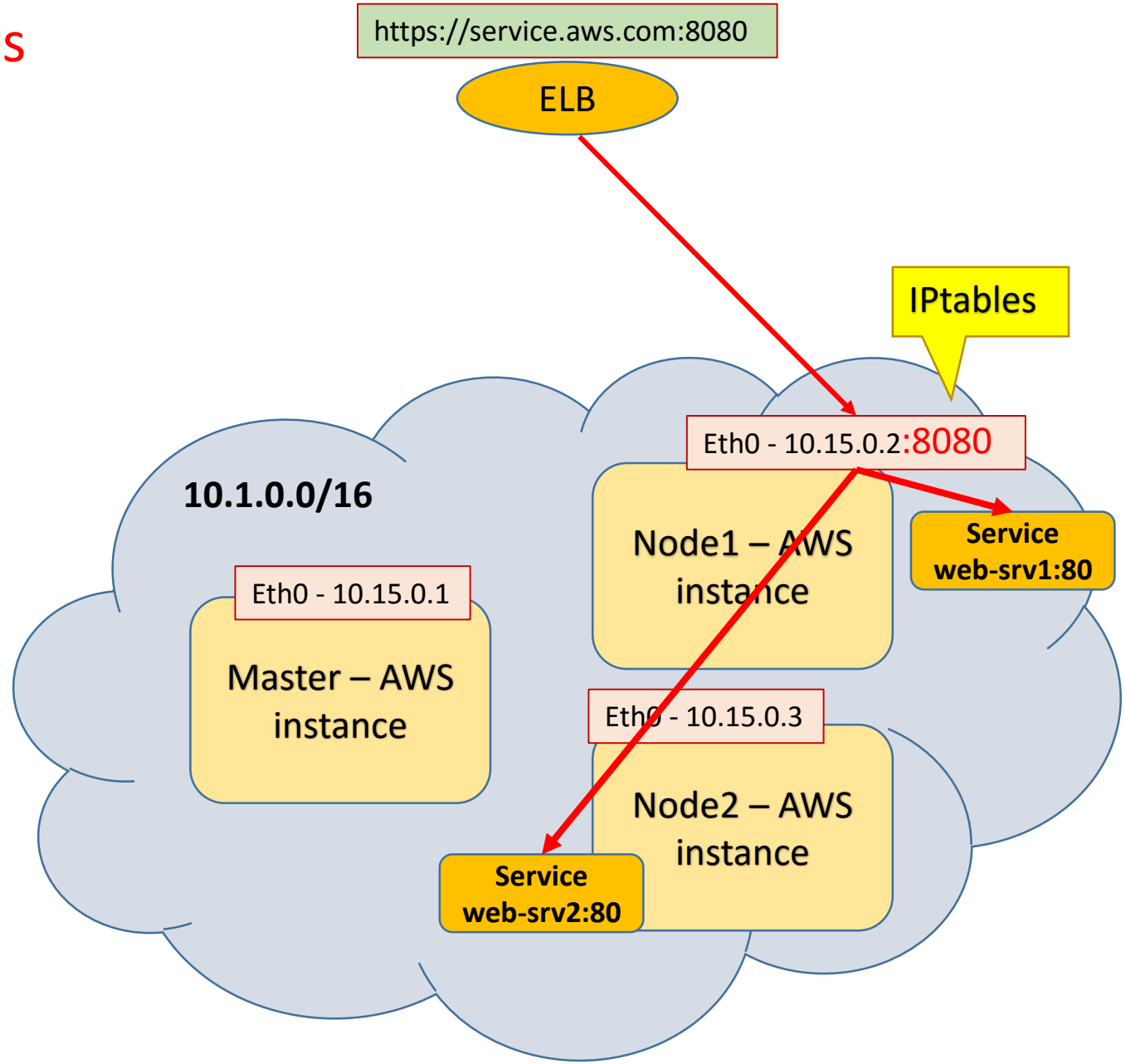


OpenShift@AWS – LB routing to cluster network

Concern – network routing towards the cluster network

Option 1 – AWS ELB

1. Forwards to the OpenShift node's IP using port mapping
2. Need application ports coordination - Manageability issues
3. Excessive IPtables for port mapping manipulation – prone to errors
4. Dependency on AWS services

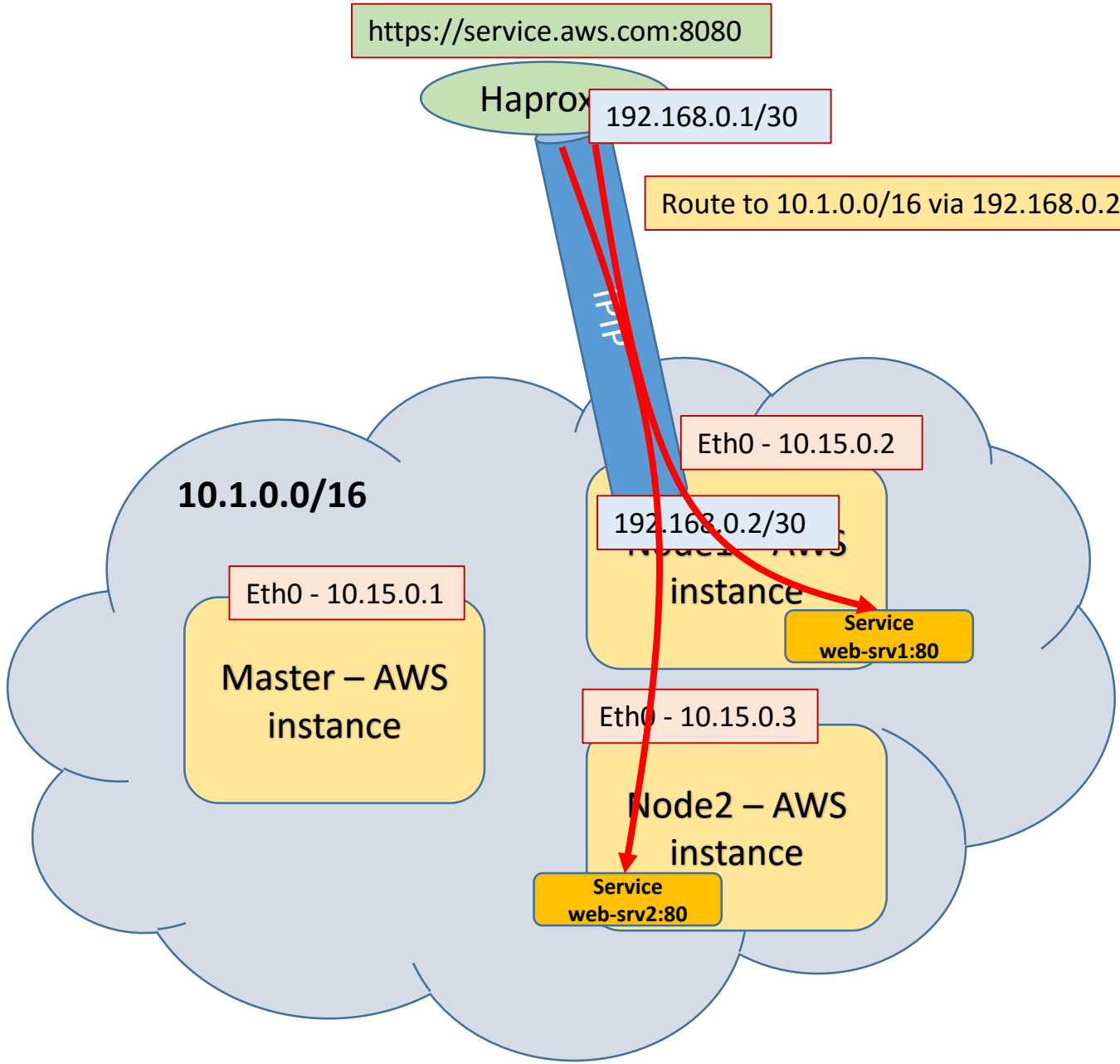


OpenShift@AWS – LB routing to the cluster network

Concern – network routing towards the cluster network

Option 2 – Tunneling

1. Tunnel the external Haproxy node to the cluster via a ramp node
2. Required extra configuration – complexity
3. Extra tunneling – performance issues
4. You need this instance to be continuously up - costly
5. AWS independency

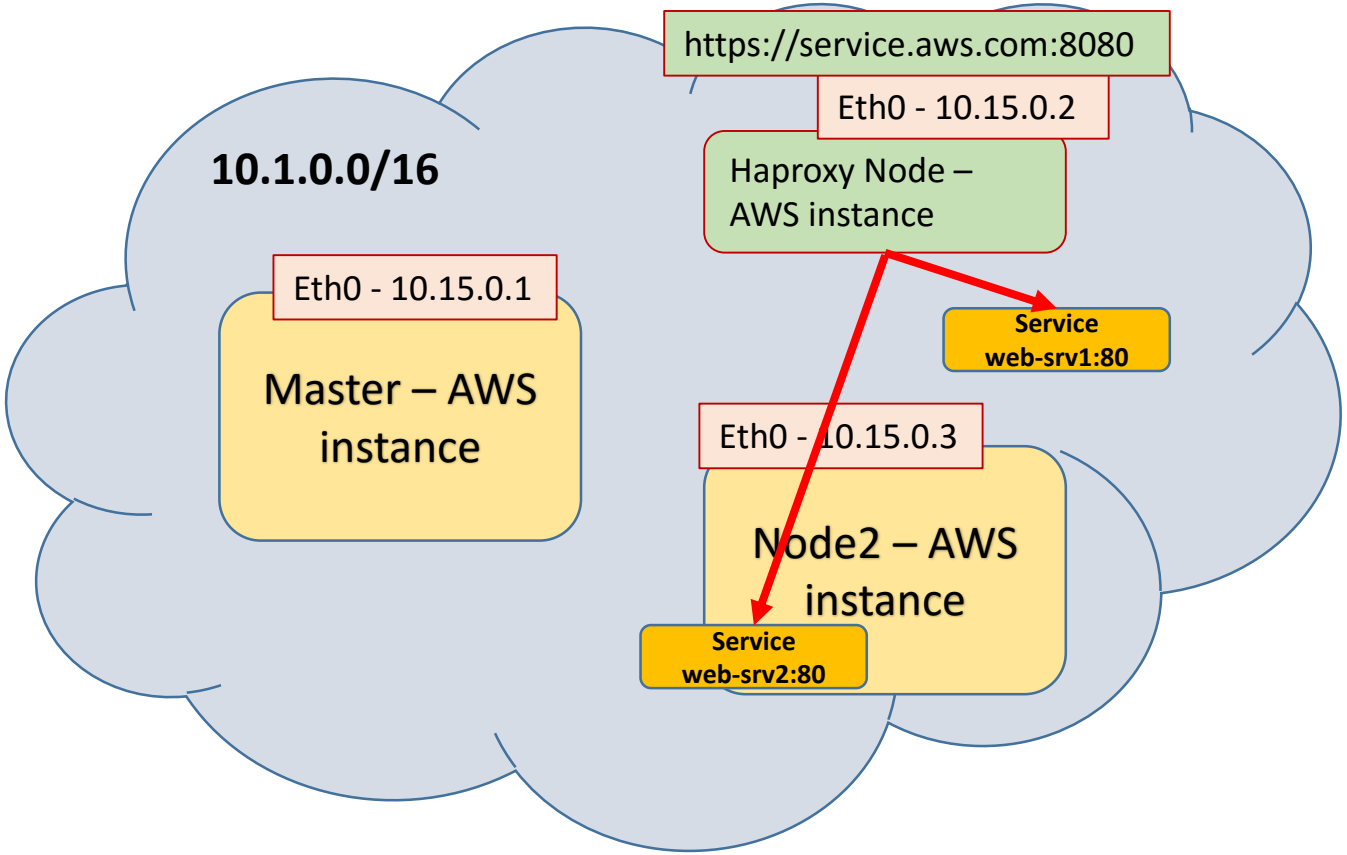


OpenShift@AWS – LB routing to the cluster network

Concern – network routing towards the cluster network

Option 3 – Haproxy move to cluster

1. Put the LB in a LB-only cluster node - disable scheduling
2. Service URL resolved to the node's IP
3. Full routing knowledge of the cluster
4. Simple and fast – native routing
5. AWS independency
6. You need this instance to be continuously up – costly

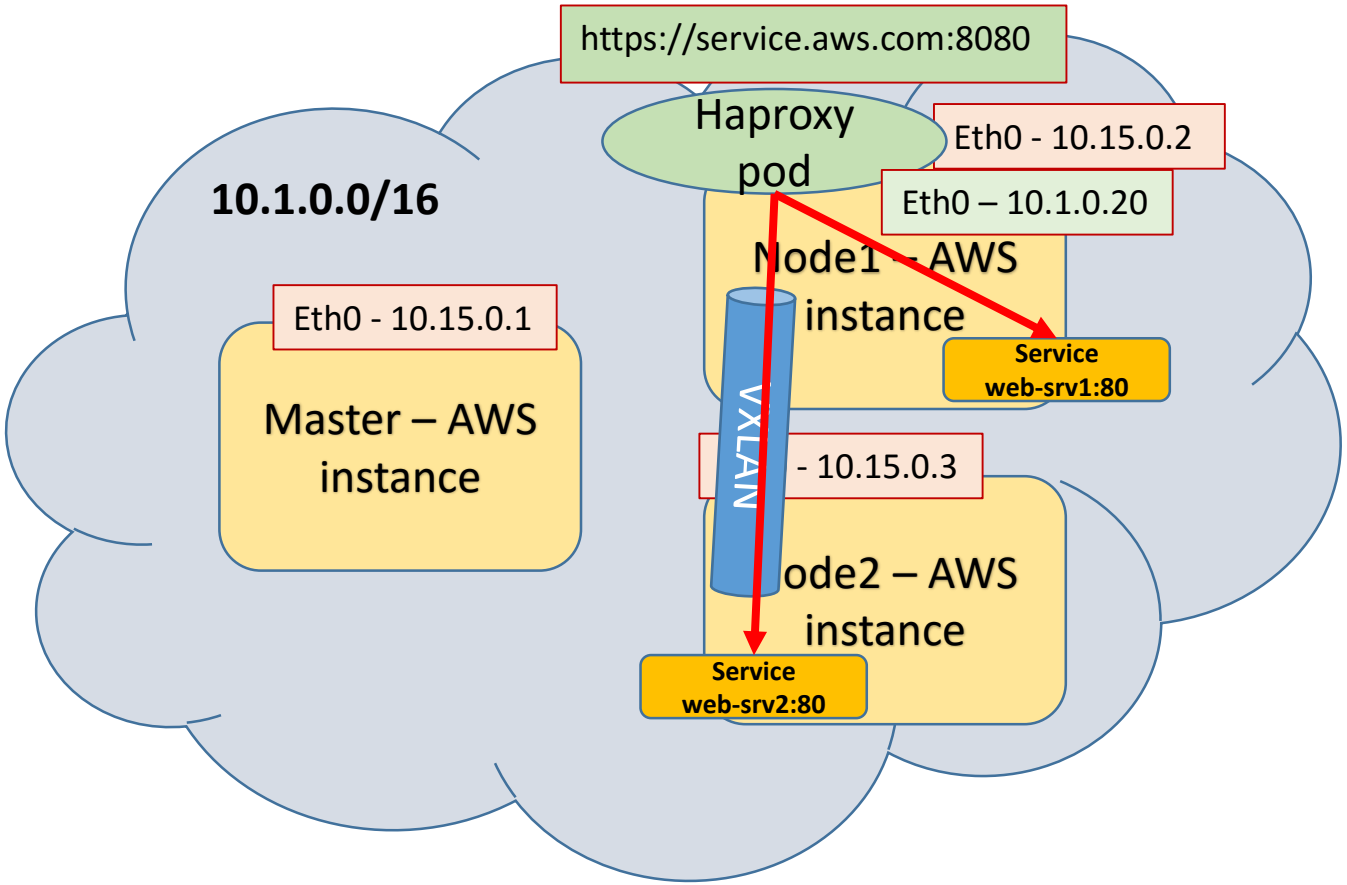


OpenShift@AWS – LB routing to the cluster network

Concern – network routing towards the cluster network

Option 4 – Haproxy container

1. Create Haproxy container
2. Service URL resolved to the container's IP
3. Full routing knowledge of the cluster
4. AWS independency
5. Use cluster overlay network – native
6. Overlay network - being used anyway



Network Security

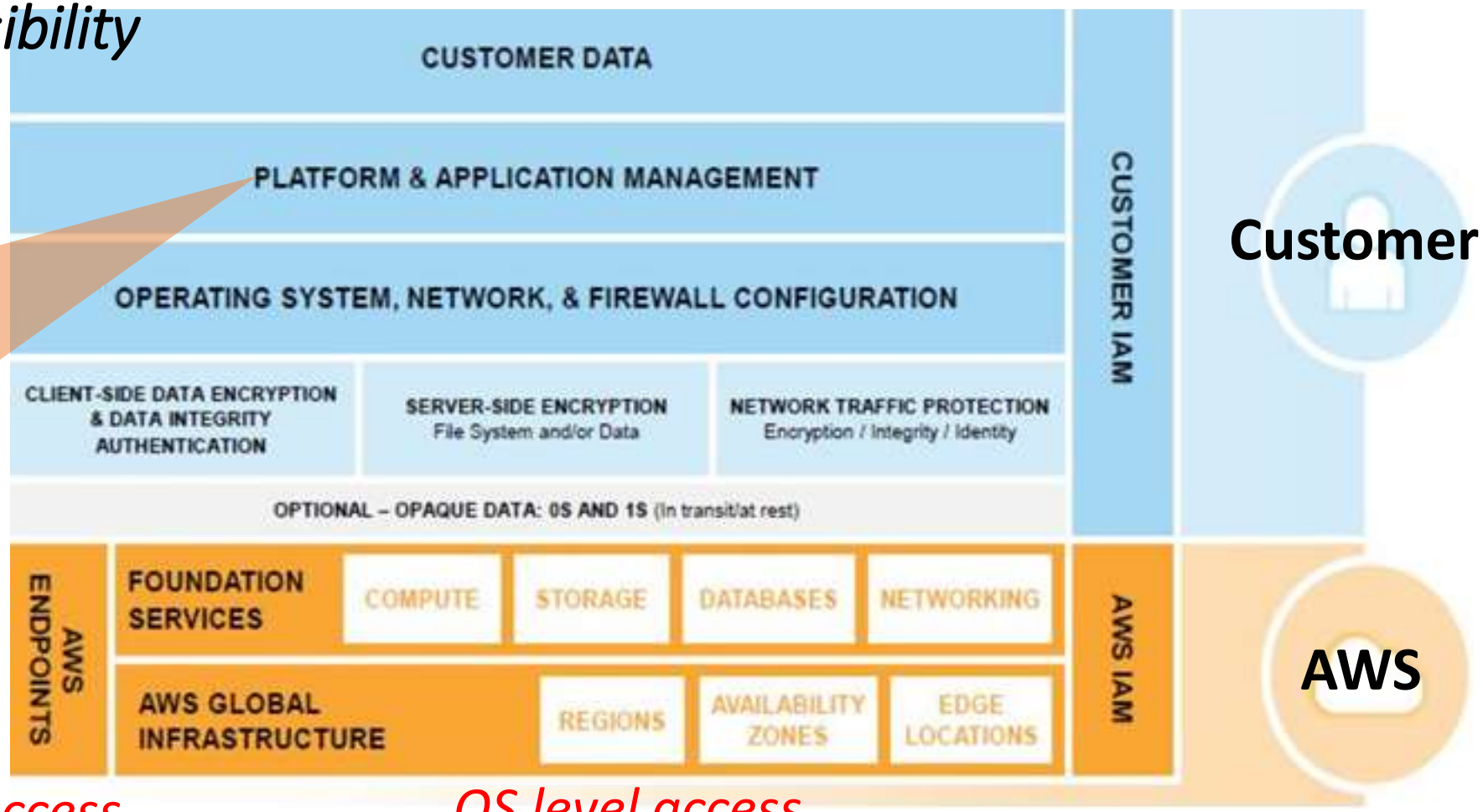
Requirements

Alternatives

Networking solutions capabilities

Shared Security responsibility Model

ELB, Lambda and application related services are optional. Not considered part of the shared trust model



AWS level resource access

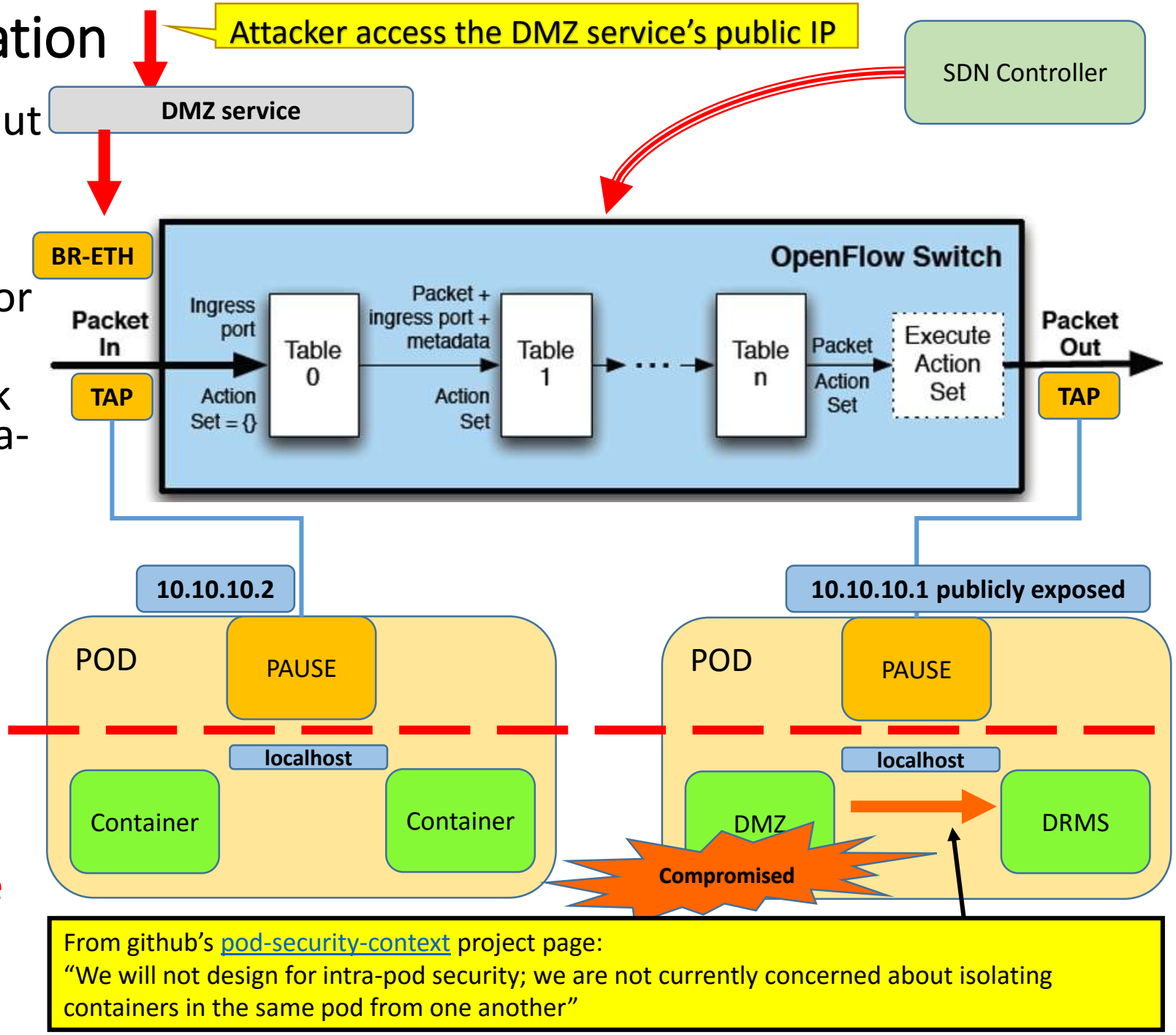
- Creating VPC, instance, network services, storage services…..
- Requires AWS AAA
- Managed only by AWS

OS level access

- SSH or RDP to the instance's OS…..
- Requires OS level AAA or certificates
- Managed by the customer

Intra-pod micro-segmentation

- For some reason, someone put containers with different sensitivity level within the same pod
- OVS uses IP/MAC/OVS port for policy (pod only attributes)
- No security policy or network segmentation applied to intra-pod containers
- Limited connections or TCP ports blocks - tweaks that won't help to deal with the newly discovered exploit
- "Security Contexts" feature doesn't apply to intra-pod security but to the pod level
- It should be presumed that containers in a pod share the same security level!



OpenShift SDN – network segmentation

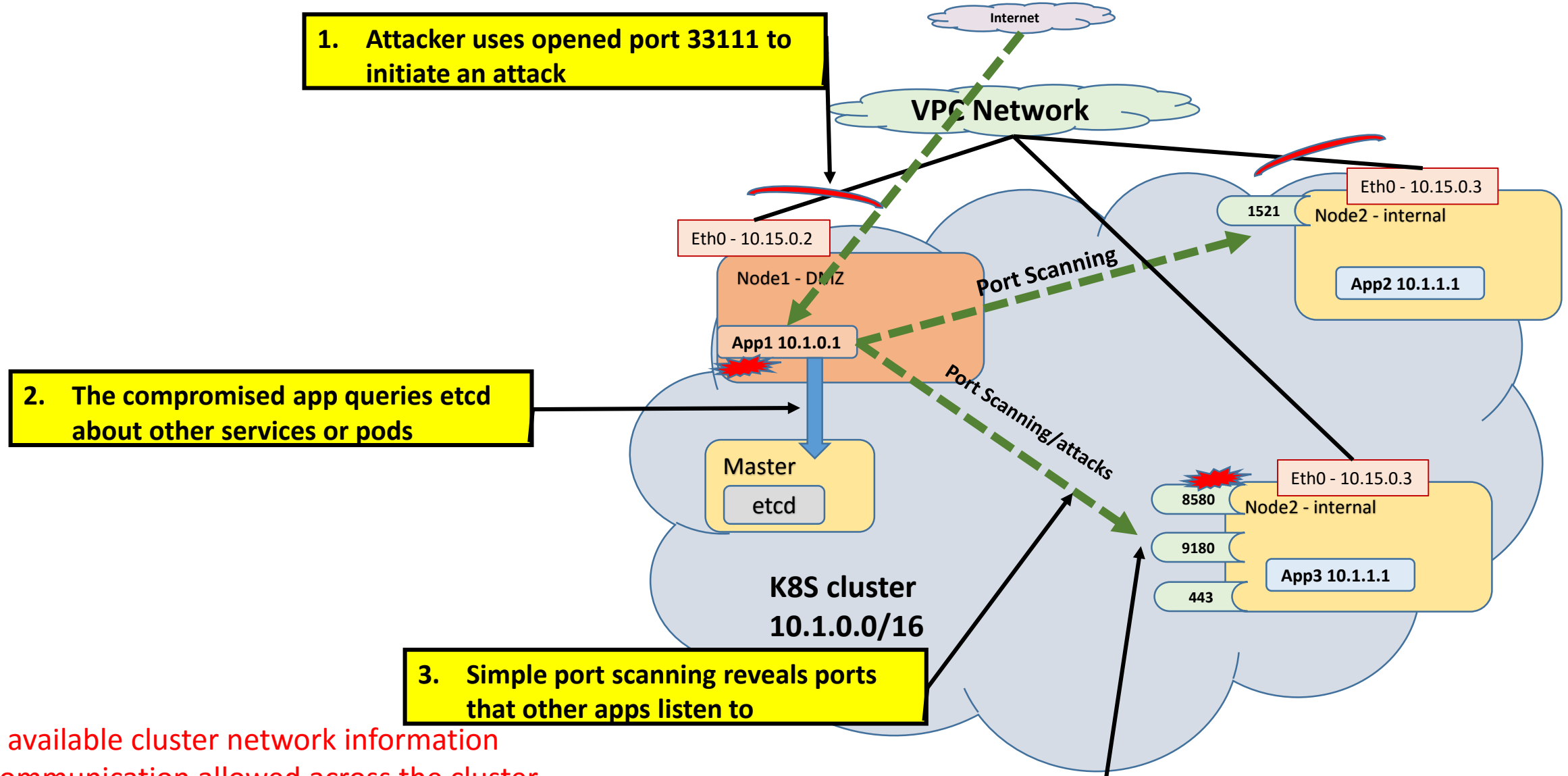
Three options:

1. *Separated nodes same cluster* – DMZ nodes and SECURE nodes – applying access control using security groups – communication freely allowed across the cluster – doesn't give real segmentation with ovs-subnets

2. *Separated clusters – DMZ and SECUR*E – different networks – implicit network segmentation – expensive but simple for short term

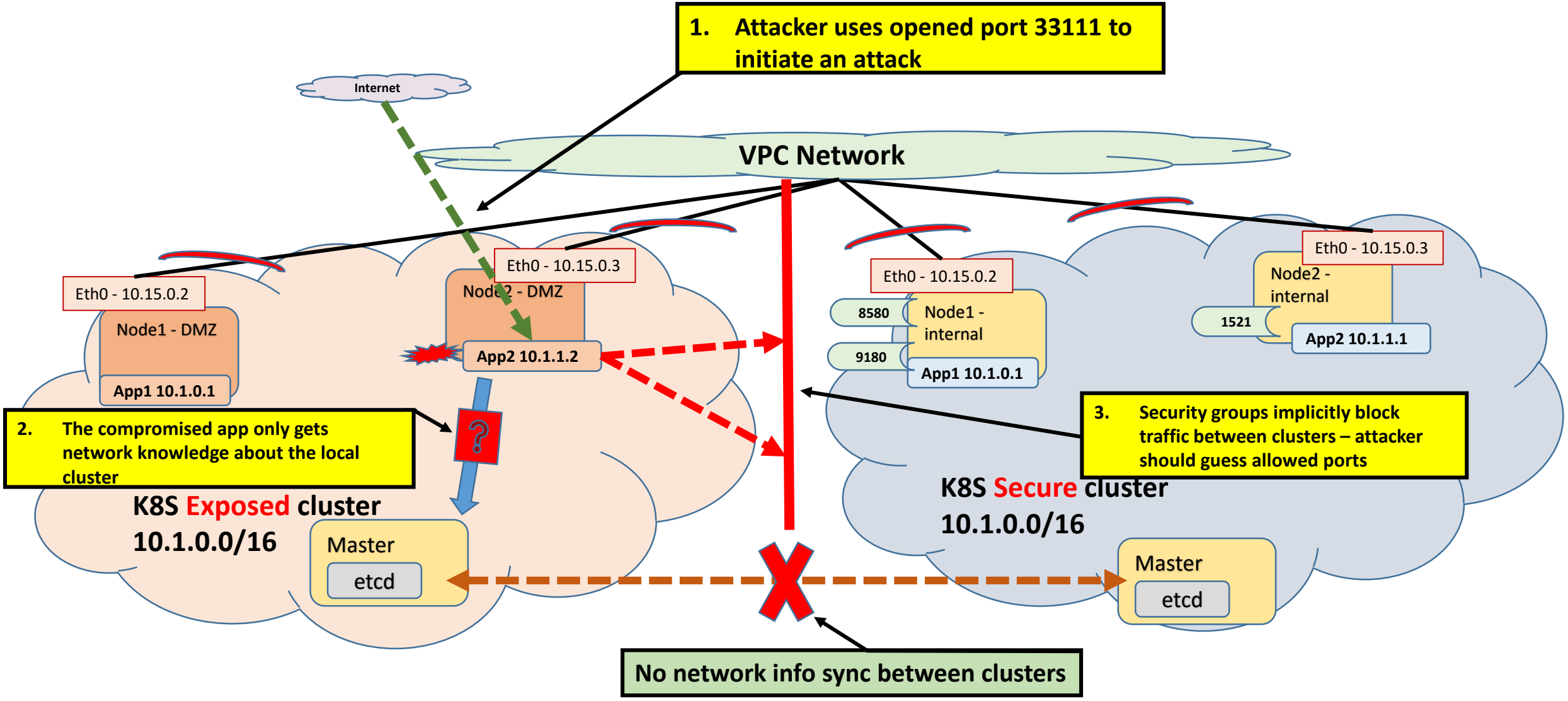
3. Using OpenShift's ovs-multitenant driver – gives micro-segmentation using projects' VNIDs

Option 1 - Node segregated – same cluster



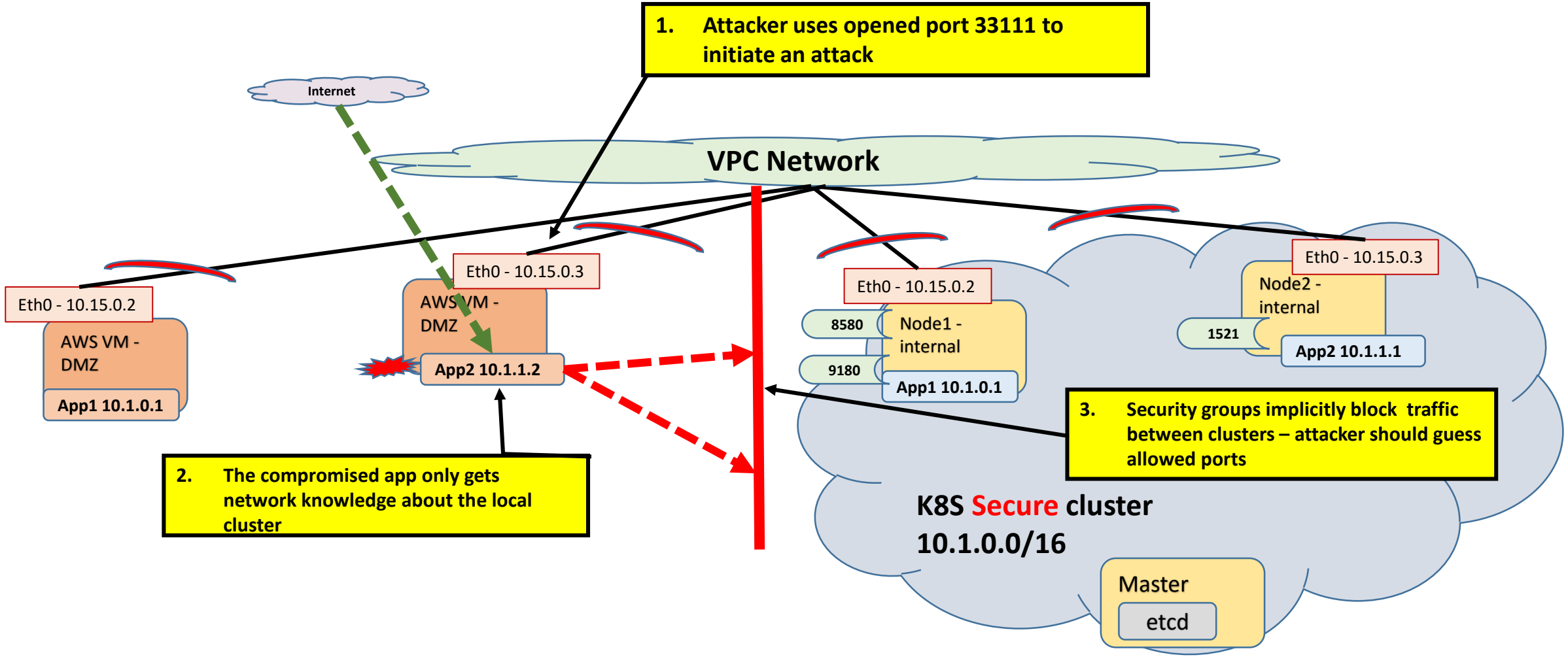
- Freely available cluster network information
- Free communication allowed across the cluster
- Other opened ports on the target can also be leveraged for attacks

Option 2 - Cluster level segmentation



- No cross cluster network visibility
- Only specific ports are allowed across clusters
- Ports on the target, other than the allowed, cannot be leveraged for attacks

Option 3 - Cluster level segmentation with native VMs



- No kubernetes cluster network visibility
- Only specific ports are allowed towards the cluster
- Ports on the target, other than the allowed, cannot be leveraged for attacks

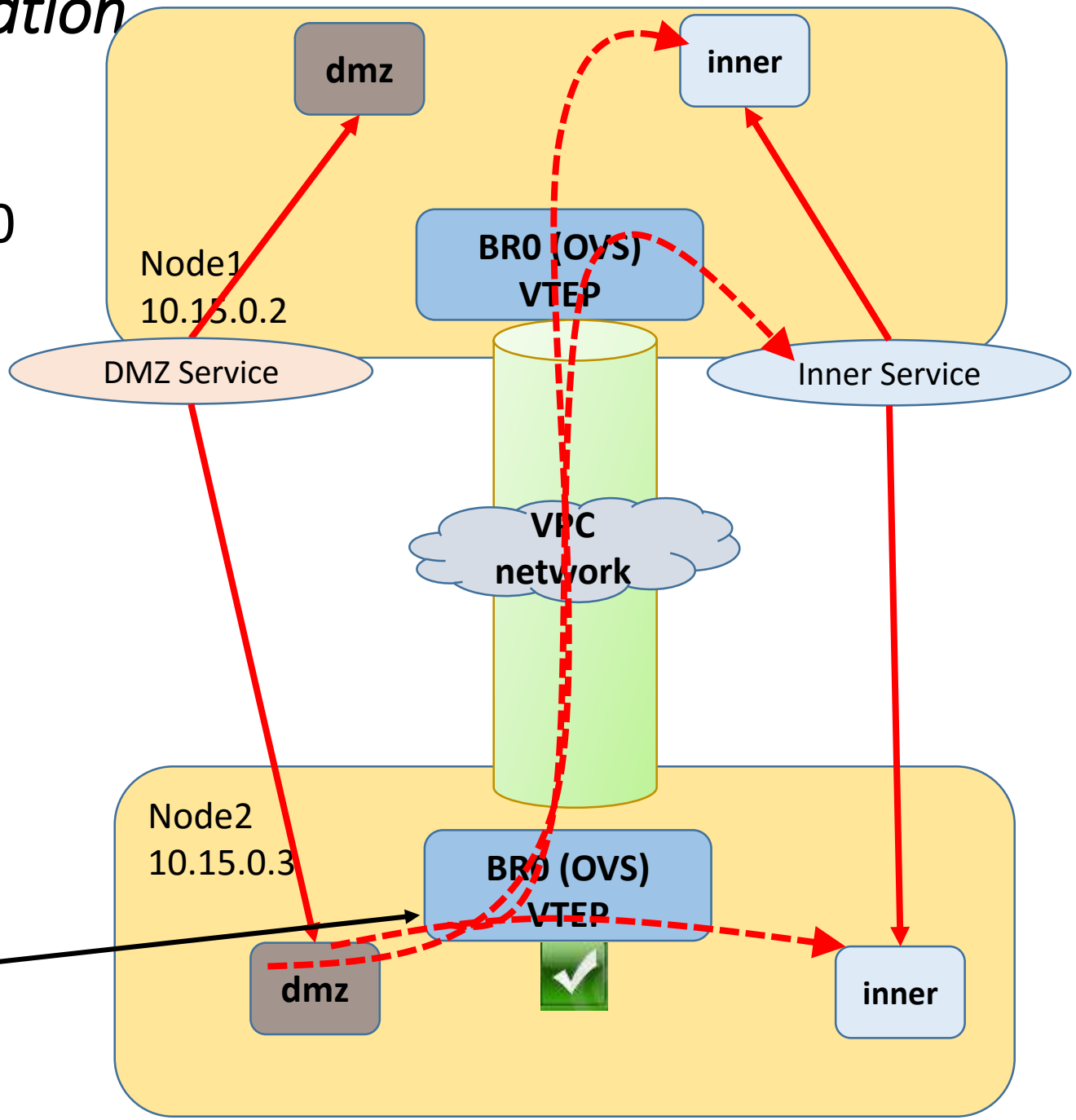
OpenShift SDN – network segmentation

OVS-Subnet plugin

- All projects are labeled with VNID 0
  So they allowed to communicate with all other pods and services

- No network segmentation

- Other filter mechanism required:
  OVS flows, Iptables, micro-segmentation solutions



OpenShift SDN – network segmentation

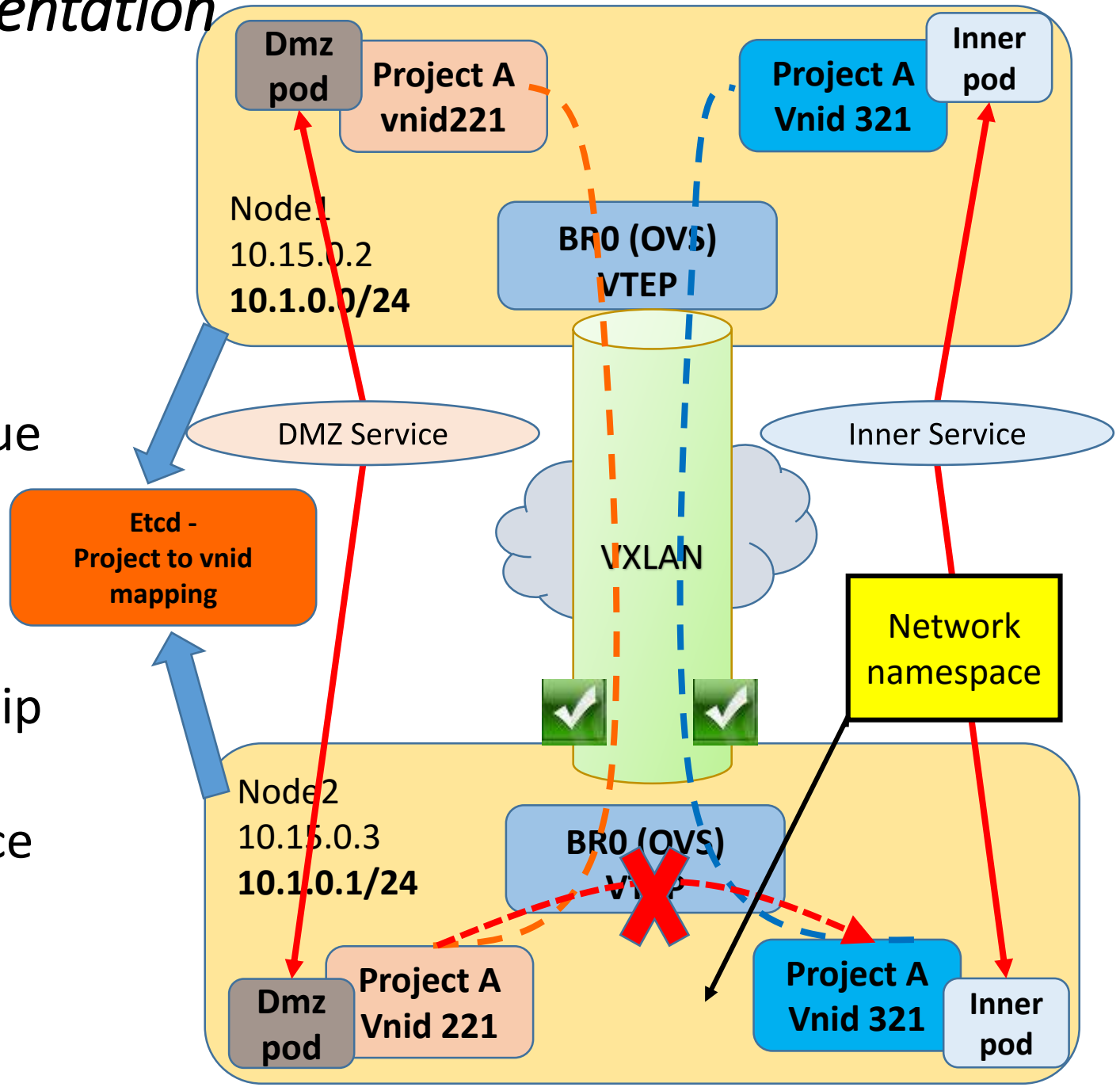
ovs-multitenant SDN plugin

OpenShift default project – VNID 0 – Allows access to/from all

All non-default projects – given unique VNID in case they are not joined together

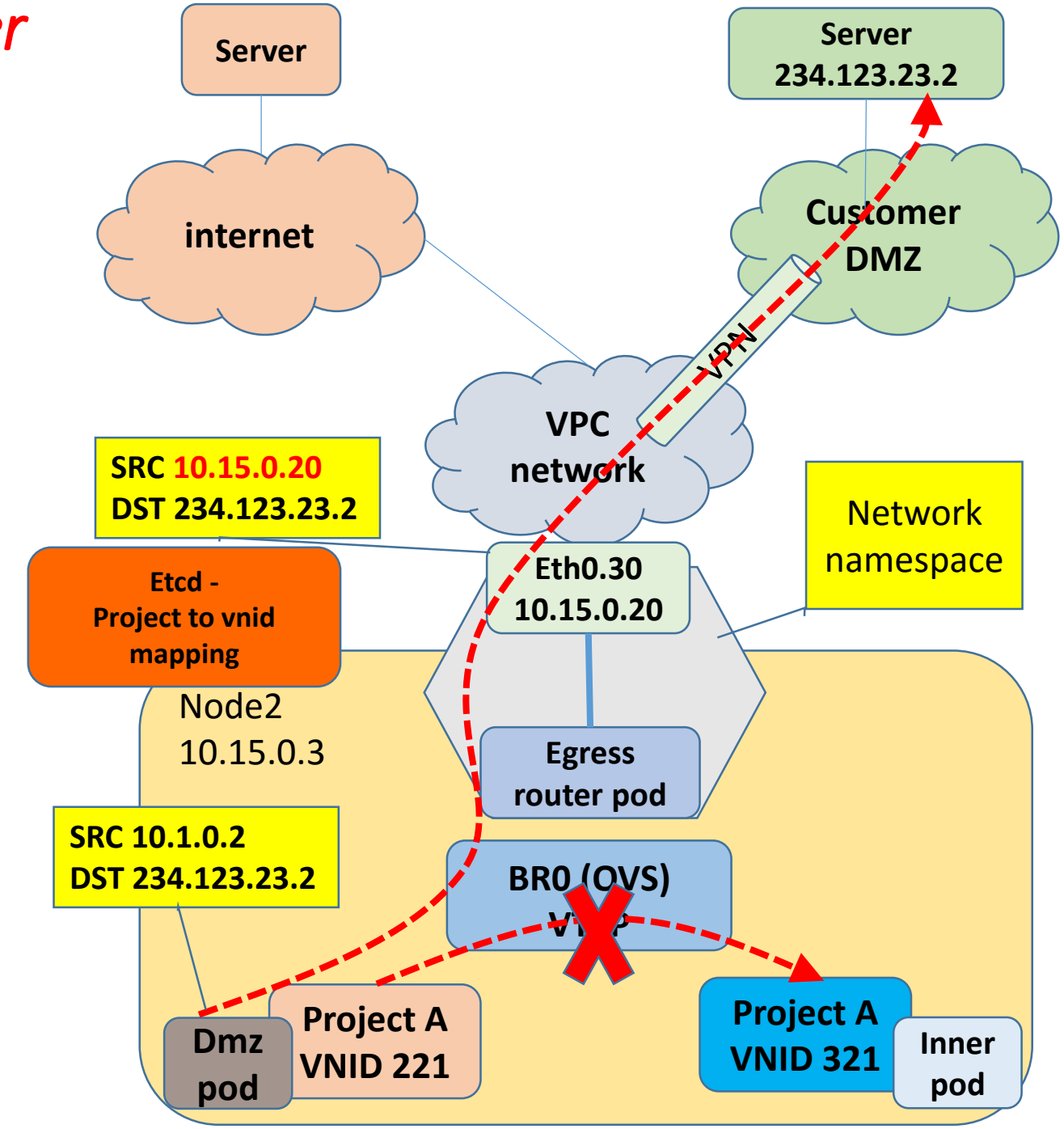
Pods – get their network association according to their project membership

Pod can access another pod or service only if they belong to the same VNID otherwise OVS blocks them



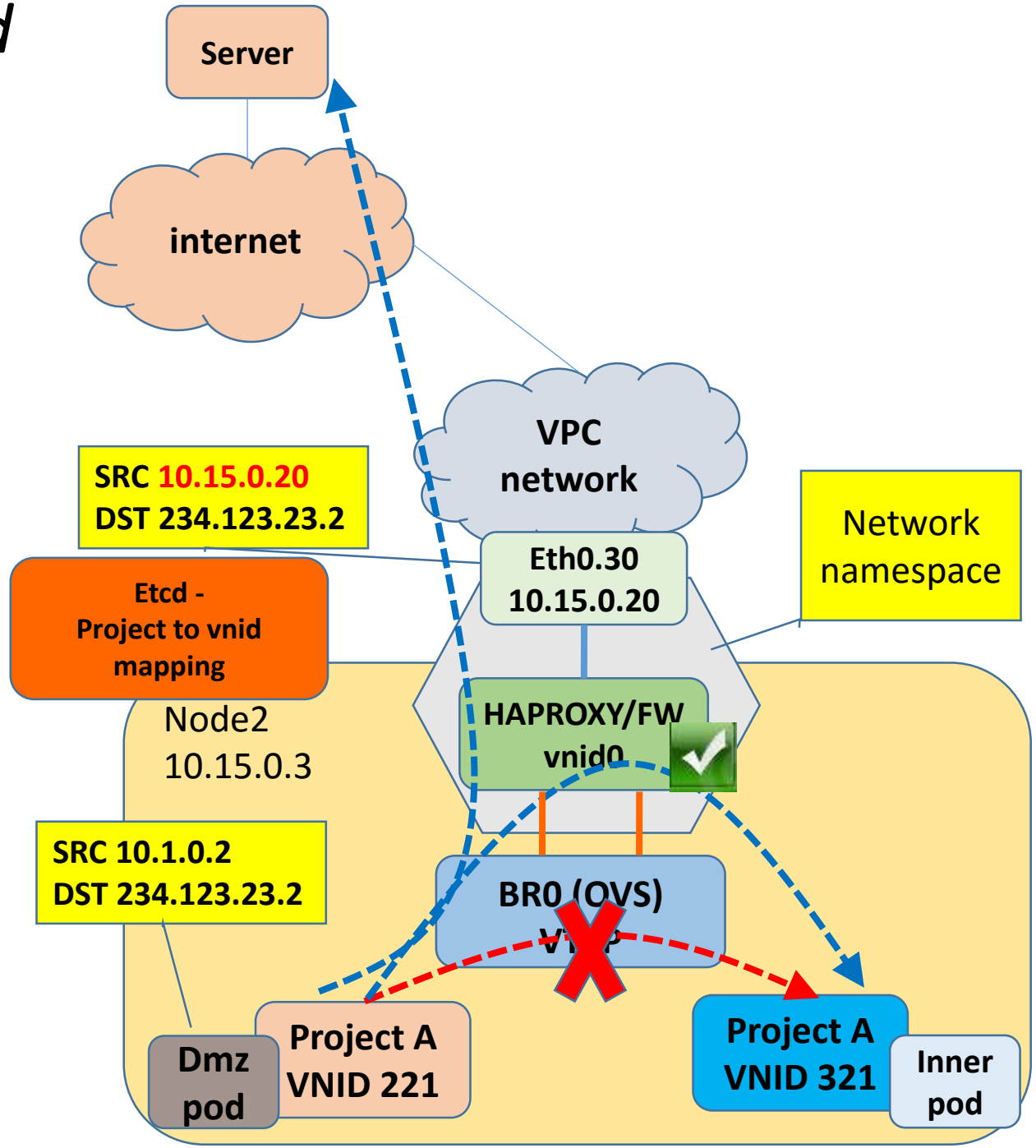
Controlled isolation - Egress Router

- A privileged pod
- Redirects pod's traffic to a specified external server when it allows connections from specific sources
- Can be called via a K8S service
- Forwards the traffic outside using its own private IP then it gets NATed by the node
- Steps –
  - Creates MACVLAN interface on the primary node interface
  - Moves this interface to the egress router pod's namespace



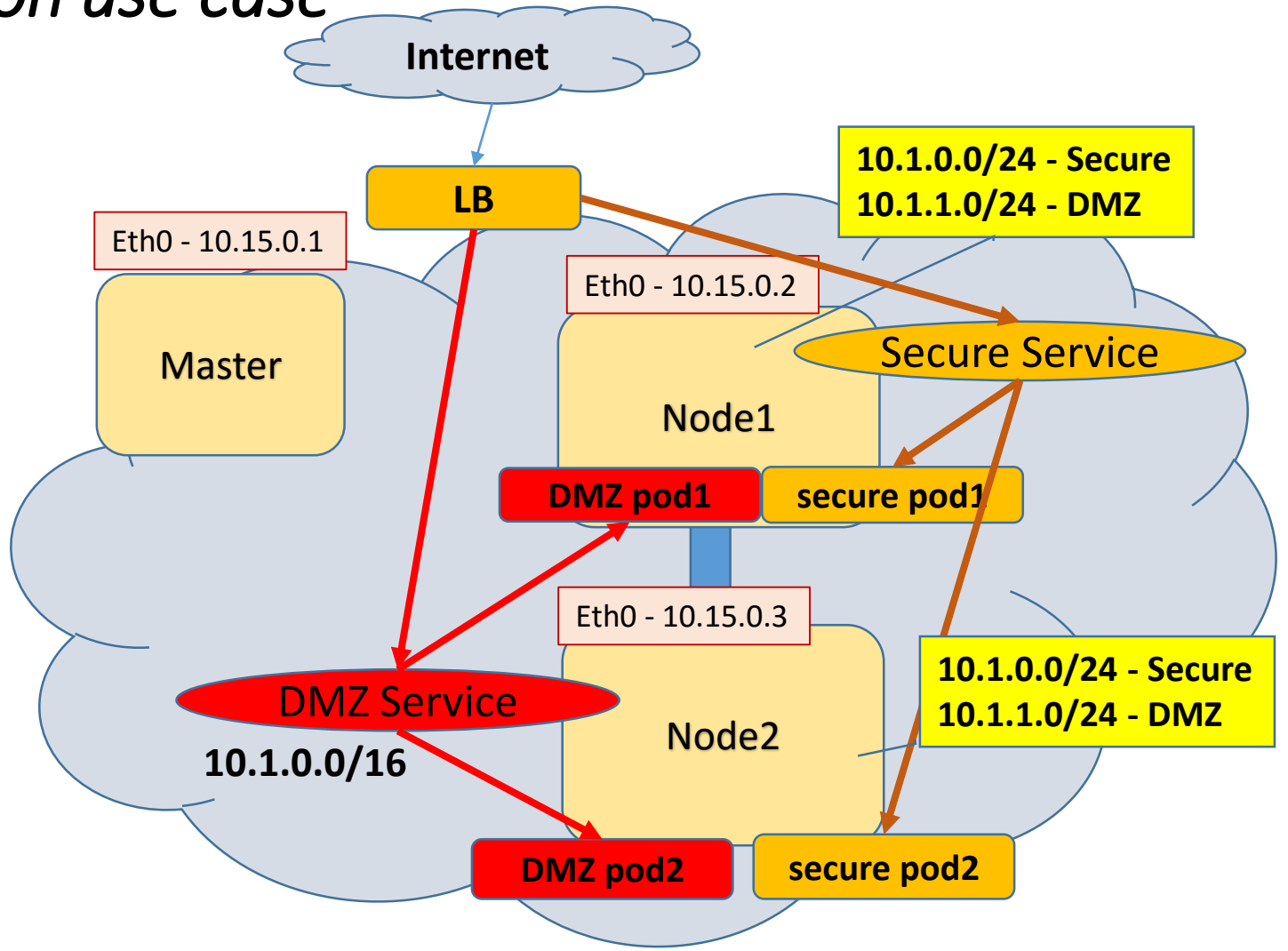
Controlled isolation – Gateway pod

- Created on the project level
- Can be applies only to isolated pods (not default or joined)
- Can be used to open specific rules to an isolated app
- If pod needs access to specific service belongs to different project, you may add EgressNetworkPolicy to the source pod's project



OpenShift SDN – L3 segmentation use case

1. Secure and DMZ subnets
2. Pods scheduled to multiple hosts and connected to subnets according to their sensitivity level
3. Another layer of segmentation
4. More “cloudy” method as all nodes can be scheduled equally with all types of PODs
5. Currently doesn’t seem to be natively supported
6. Nuage plugin supports this



AWS security groups inspection

Concern – Users may attach permissive security groups to instances

Q - Security group definition – manual or automatic?

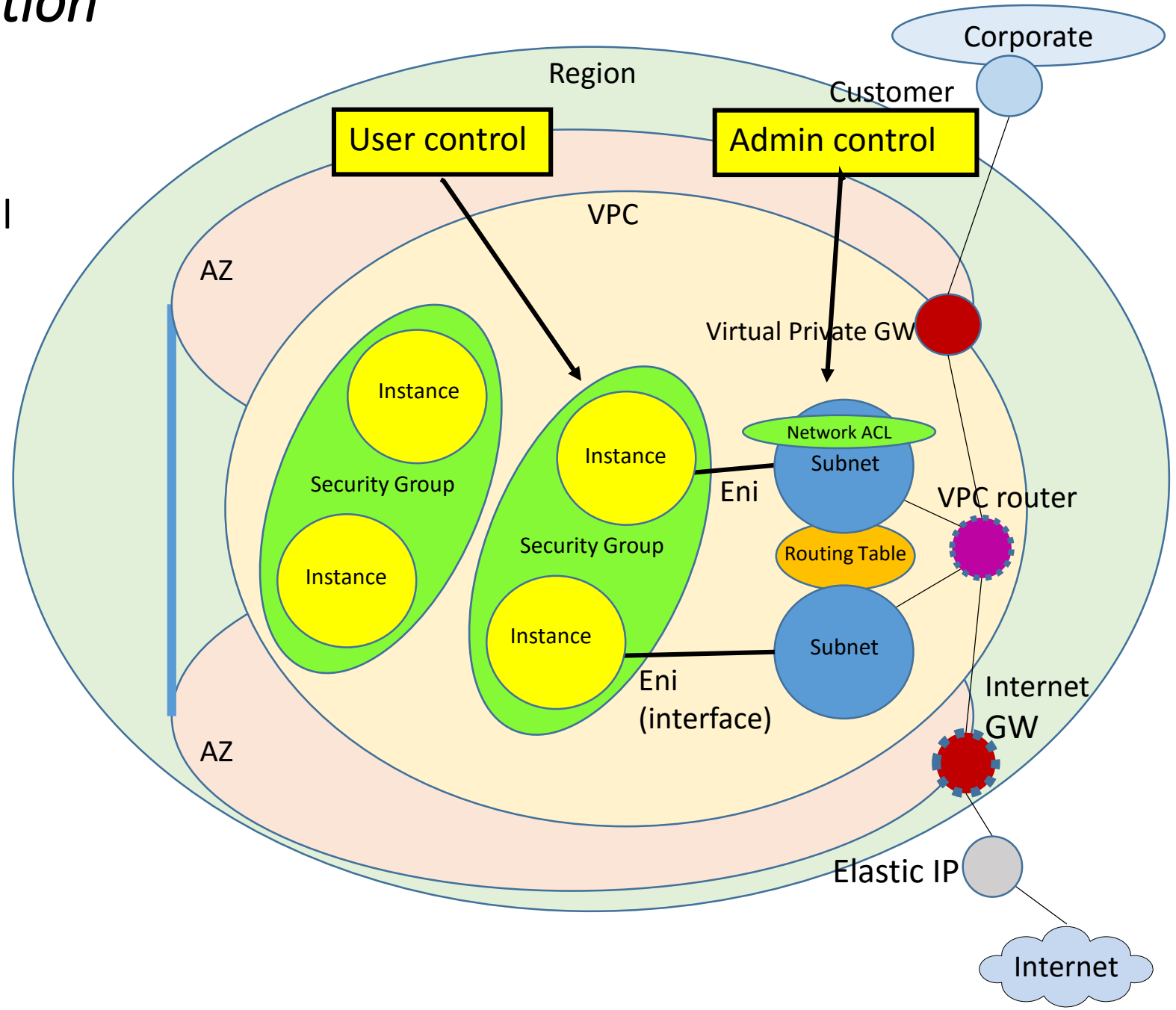
A - Proactive way –

- Wrapping security checks into continuous integration
- Using subnet-level Network ACL for more general deny rules – allowed only to sec admins
- Using third party tools: Dome9..

A - Reactive way –

Using tools such as aws_recipes and Scout2 (nccgroup) to inspect

Lots to be discussed

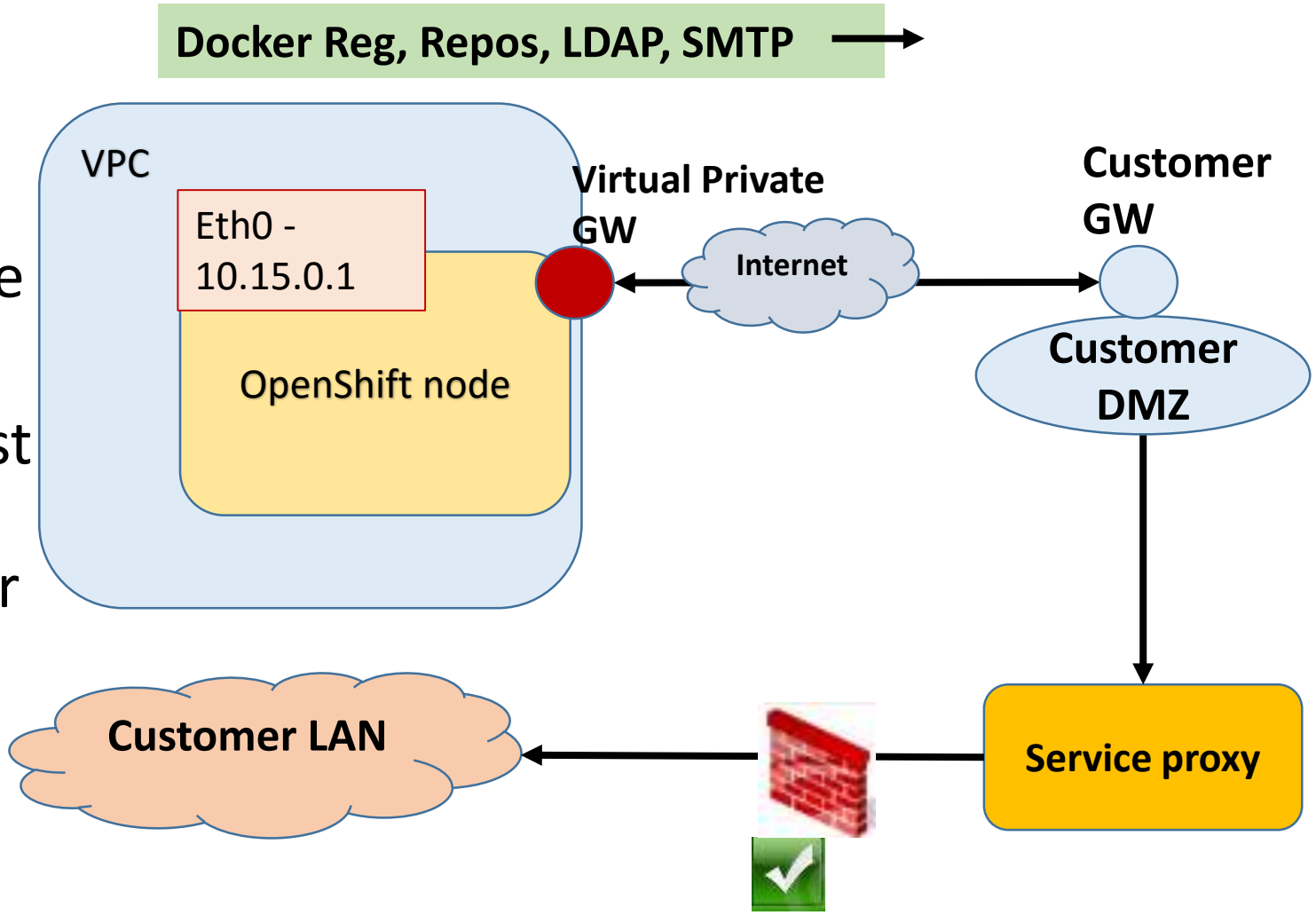


OpenShift – Internal customer services consumption

Yum repos, Docker registry, SMTP, LDAP

Leveraging the VPN tunnels from AWS to the customer DMZ

1. The node connects to the requested service proxy in the customer DMZ
2. The proxy initiates the request for the service sources by its own IP – allowed by customer firewall



Questions