IBM Training
Red Hat OpenShift Application Development

—

Luca Floris
IBM Cloud Technical Consultant

OpenShift Application Developer Training - Agenda

Week 1 — 26th – 28th October 2020

Day 1
Cloud native application architecture
- Cloud–native overview
- Working with containers
- Lab 1: Working with Docker and Containers
- Introduction to Kubernetes and OpenShift
- OpenShift architecture overview
- Working with OpenShift

Day 2
Develop and deploy apps on OpenShift
- OpenShift Application Deployments
- Building and deploying application on OpenShift
- Labs 2-8: Deploying various types of Applications

Day 3
Develop and deploy apps on OpenShift continued…
- Package management with Helm
- Lab 9: Deploying Helm applications
- OpenShift Application Security
- Lab 10-11: OpenShift application security and quotas
- Image Registries
- Lab 12: Working with image registries

Week 2 — 2nd – 3rd November 2020

Day 4
DevOps: Continuous Delivery
- DevOps and DevSecOps
- WebSphere Liberty on OpenShift
- Lab 13: Set up a CI/CD pipeline on OpenShift using Jenkins to deploy a simple web application
- Transformation Advisor - Migrating WebSphere Applications to OpenShift

Microservices Architecture
- Microservices application architecture
- Developing microservices
- Twelve factor applications
- Refactoring monolith applications into microservices
- Lab 14 Build and deploy a polyglot microservices application on OpenShift

Day 5
Microservices architecture
- Service mesh
- Microservices security
- Lab 15: Istio lab

Agenda

Day 1

Cloud native application architecture

Cloud Native Overview
Introduction to Containers
Deploying containers
Introduction to Red Hat OpenShift
OpenShift 4 Architecture
OpenShift Container Networking
OpenShift Container Security
Introduction to OpenShift Storage

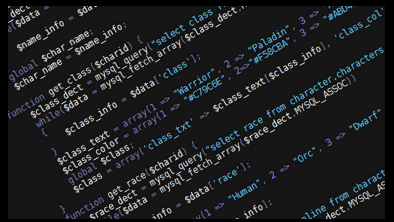
Cloud Native Application Development

Cloud Native Application Development

What is Cloud Native?

Less about where it runs

More about how it is built



Cloud Native Application Development

What is Cloud Native?

A cloud native application consists of discrete, reusable components known as microservices that are designed to integrate into any cloud environment.

Microservices work together as a whole to comprise an application, yet each can be independently scaled, continuously improved, and quickly iterated through automation and orchestration processes.

The flexibility of each microservice adds to the agility and continuous improvement of cloud-native applications.

Cloud Native Application Development

Advantages

Easier to manage

cloud native applications can be improved incrementally and automatically

Improvements can be made non-intrusively

Easy to scale up and down

Improved speed and innovation that’s demanded by today’s environment

Disadvantages

A lot more elements to manage

Additional skills/toolsets to manage the DevOps pipeline, replace traditional monitoring structures, and control microservices architecture.

Cloud native demands a business culture that can cope with the pace of that innovation.

Cloud Native Application Development

Development principles

Whether creating a new cloud native application or modernizing an existing application, developers adhere to a consistent set of principles:

Follow the microservices architectural approach: Break applications down to the single-function services known as microservices.

Rely on containers for maximum flexibility and scalability: Containers package software with all its code and dependencies in one place, allowing for flexibility, portability and scaling.

Adopt Agile methods: Agile methods speed the creation and improvement process.

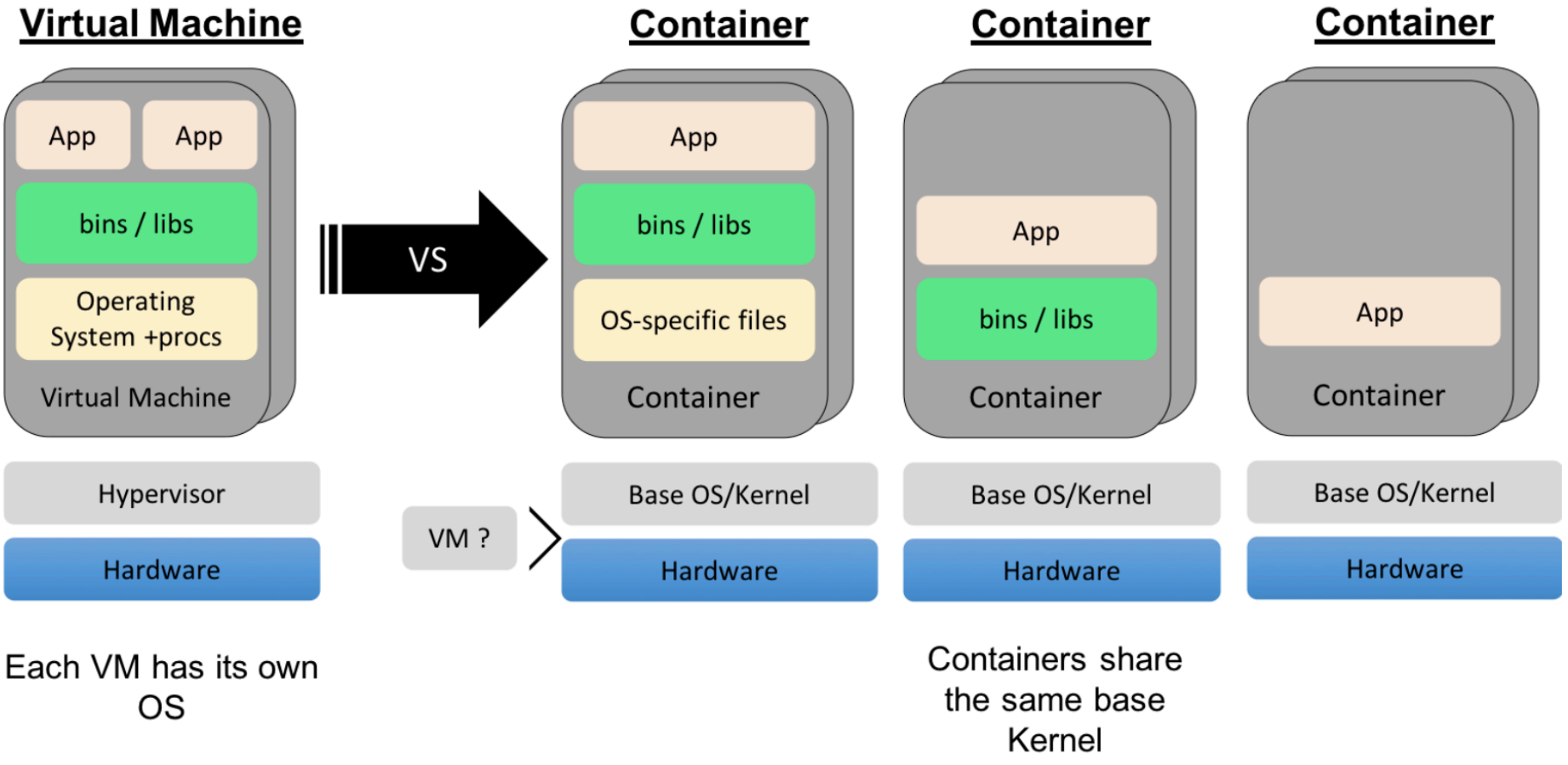
Introduction to Containers

What is a container?

- A container is a set of one or more processes that are isolated from the rest of the system.
- Containers provide many of the same benefits as virtual machines, such as security, storage, and network isolation.
- Low hardware resource requirements
- Quick to start and terminate.
- They also isolate the libraries and the runtime resources for an application

- ✓ Efficiency, elasticity, portability and reusability!
- ✓ Low hardware footprint
- ✓ Environment isolation
- ✓ Quick deployment
- ✓ Multiple environment deployment
- ✓ Reusability

Traditional Application vs Containers



Deploying Containers

Can be done on any machine that has a container runtime

Examples include Docker, Podman, cri-o, containerd etc

Commands are as simple as

```
$ docker run hello-world

Hello from Docker!
This message shows that your
installation appears to be working
correctly.
```

Container Persistence

A container can be stateless or stateful, but the 12 Factor Microservices methodology recommends apps to be as stateless as possible

Stateless requires no defined container storage, where as stateful requires a **volume**.

Volumes are mounted to the container by the container runtime and provide a means to write data to something accessible from outside the container

Where are my logs?

What if the container restarts?

I have a database, what now?

???

Container Images

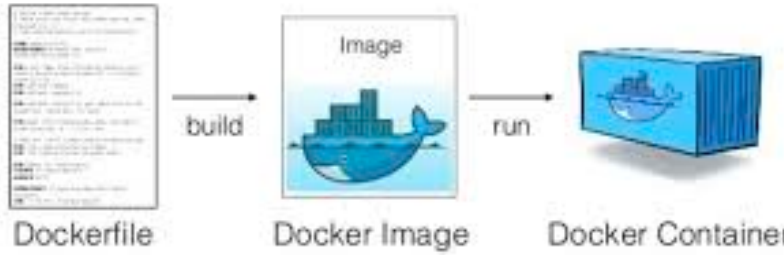
Contains all the dependencies required for the container to run

Foundation for running containers

Containers use an immutable view of the image, so multiple containers can use the same image simultaneously.

Images are created using a **Dockerfile**

Images are stored locally or in a accessible **image repository**



Running a Container Locally

Figure out WHAT and HOW you want the container to run

Does it need a name?

Does it need a volume?

Do you need to expose an application port?

With a name

```
docker run --name hello hello-world
```

Stateful

```
docker run --name hello -v $(pwd)/data:/data
hello-world
```

Port 3000 exposed

```
docker run --name hello –p 3000:3000 hello-
world
```

Additional environment variables

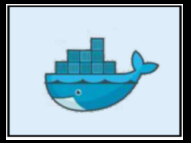
```
docker run --name hello –e MYVAR1=foo hello-
world
```

Building a Docker Image

Dockerfile

```
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install -r requirements.txt

# Make port 80 available to the world outside this contain
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

docker build -t myapp appdir/



Image

Application

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis

app = Flask(__name__)

@app.route("/")
def hello():

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname()
    return 'My hostname is %s'

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

Demo - Using the Lab Environment

Demo – Building a Container Application

Lab – Building a Container Application

Visit https://github.com/lfloris/openshift-dev-training/tree/main/Labs for lab materials

Go to Lab 1

Goals

Locally build and run a stateless container

Locally build and run a stateful container

Locally build and run a 2 tier Wordpress and MySQL container application

Locally build a custom Python Docker image

Locally deploy a 2 tier monitoring application

Kubernetes

What is Kubernetes?

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services

A framework to run distributed systems resiliently

Provides container orchestration that automates many of the manual processes involved in deploying, managing, and scaling containerized applications.

Benefits of Kubernetes Orchestration

Service discovery and load balancing

Automatic bin packing

Storage orchestration

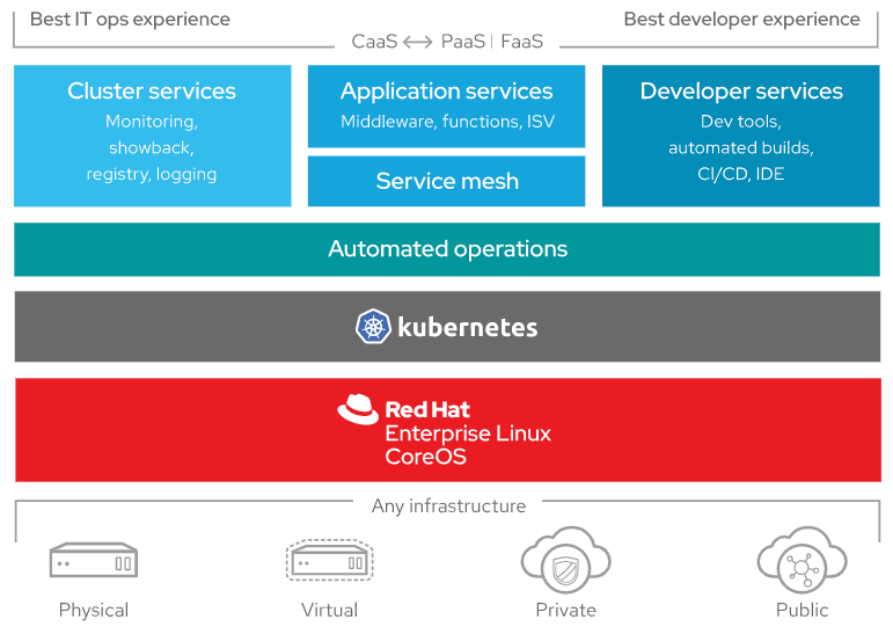
Self-healing

Automated rollouts and rollbacks

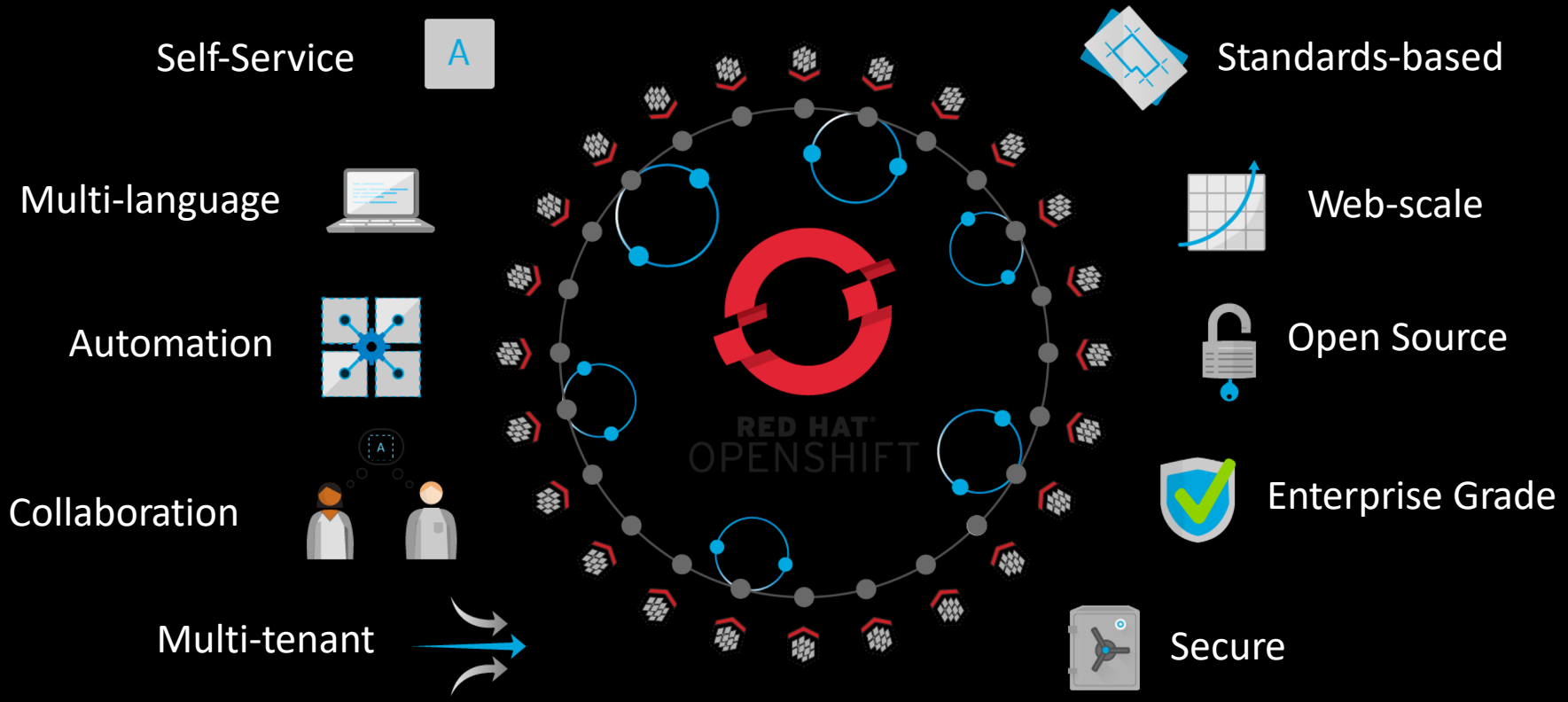
Secret and configuration management

Red Hat OpenShift

What is OpenShift?



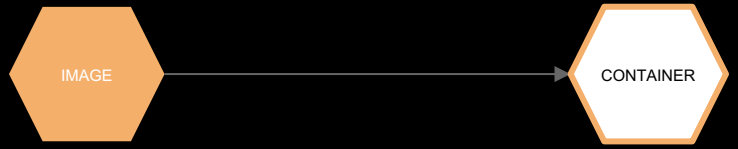
- Container host and runtime
- Enterprise Kubernetes
- Validated integrations
- Integrated container registry
- Developer workflows
- Easy access to services
- Improved installation based on immutable Red Hat® Enterprise Linux® CoreOS for consistency and upgradability.


Self-Service
Multi-language
Automation
Collaboration
Multi-tenant
RED HAT
OPENSHIFT
Standards-based
Web-scale
Open Source
Enterprise Grade
Secure

OpenShift Core Components - Pods

A Container is the smallest deployment unit

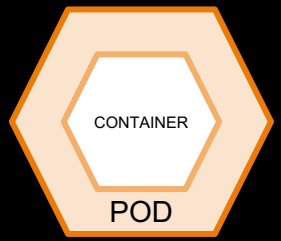
Containers are based on container images



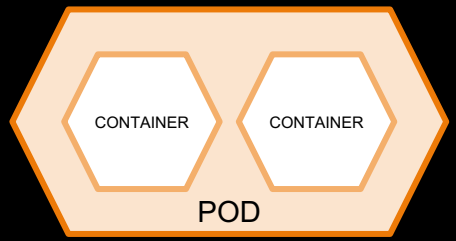
BINARY

RUNTIME

Containers are wrapped in pods that are the base units for deployment in OpenShift



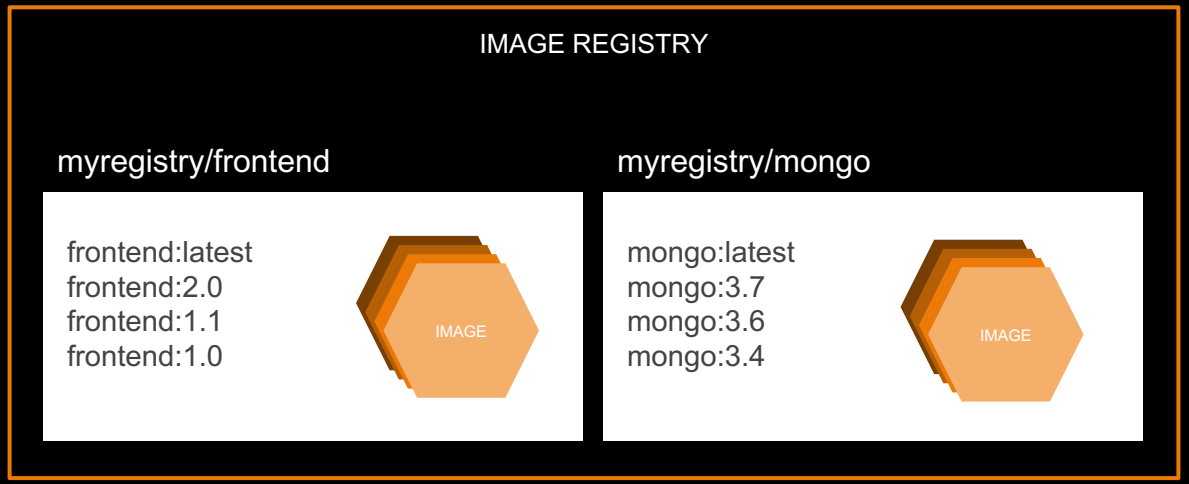
10.140.4.44



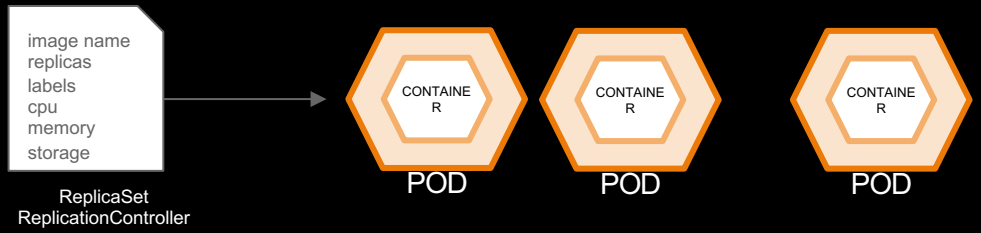
10.15.6.55

OpenShift Core Components - Images

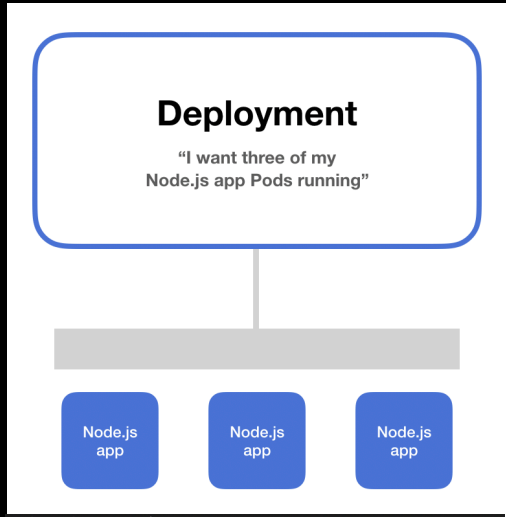
Images are stored in an Image Registry/Repository that keeps all versions of an image



OpenShift Core Components - ReplicaSets / Deployments

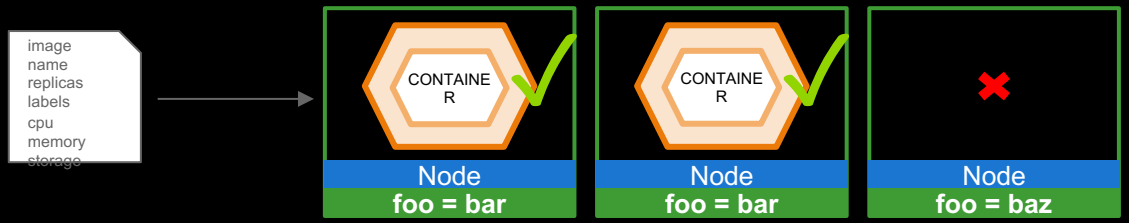


- Runs 1-N replicas of an application and automatically replaces any instances that fail or become unresponsive.
- Provide a way to define how an application runs in a Kubernetes cluster
- Control a vast array of deployment specifications




```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: python-app
  labels:
    app: python
spec:
  replicas: 3
  selector:
    matchLabels:
      app: python
  template:
    metadata:
      labels:
        app: python
    spec:
      containers:
      - name: python
        image: mycluster.icp:8500/default/python-app:v1
        imagePullPolicy: IfNotPresent
        resources:
          requests:
            memory: 128Mi
            cpu: 100m
          limits:
            memory: 512Mi
            cpu: 200m
        ports:
        - containerPort: 80
```

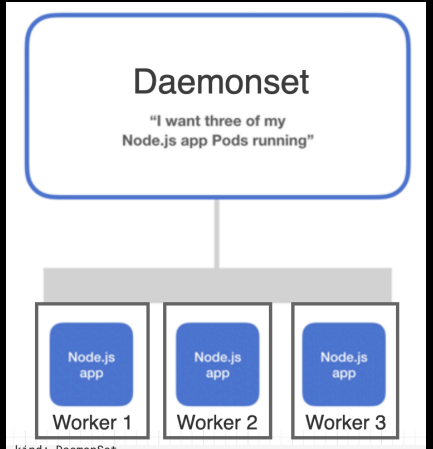

OpenShift Core Components - Daemonsets



A *DaemonSet* ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them.

As nodes are removed from the cluster, those Pods are garbage collected.

Deleting a DaemonSet will clean up the Pods it created.




```
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

OpenShift Core Components - StatefulSets

Manages the deployment and scaling of a set of Pods, and *provides guarantees about the ordering and uniqueness* of these Pods.

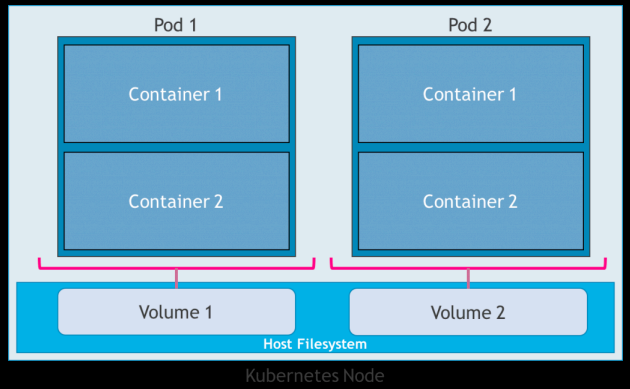
Like a Deployment, a StatefulSet manages Pods that are based on an *identical* container spec.

Unlike a Deployment, a StatefulSet *maintains a sticky identity for each of their Pods*. These pods are created from the same spec but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

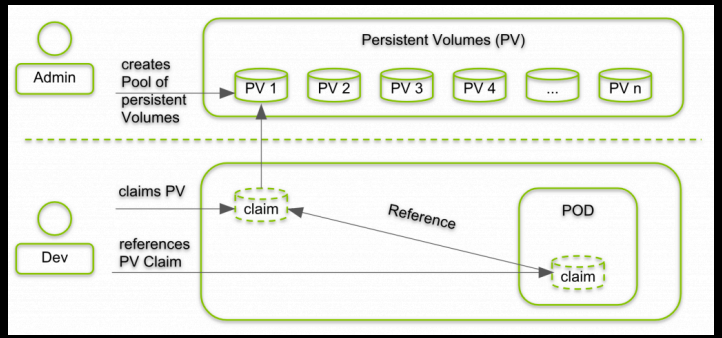

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
name: web
spec:
selector:
matchLabels:
  app: nginx # has to match .spec.template.metadata.labels
serviceName: "nginx"
replicas: 3 # by default is 1
template:
metadata:
  labels:
    app: nginx # has to match .spec.selector.matchLabels
spec:
  terminationGracePeriodSeconds: 10
  containers:
  - name: nginx
    image: k8s.gcr.io/nginx-slim:0.8
    ports:
    - containerPort: 80
      name: web
    volumeMounts:
    - name: www
      mountPath: /usr/share/nginx/html
volumeClaimTemplates:
- metadata:
  name: www
spec:
  accessModes: [ "ReadWriteOnce" ]
  storageClassName: "my-storage-class"
  resources:
    requests:
      storage: 1Gi
```

OpenShift Core Components - Persistence

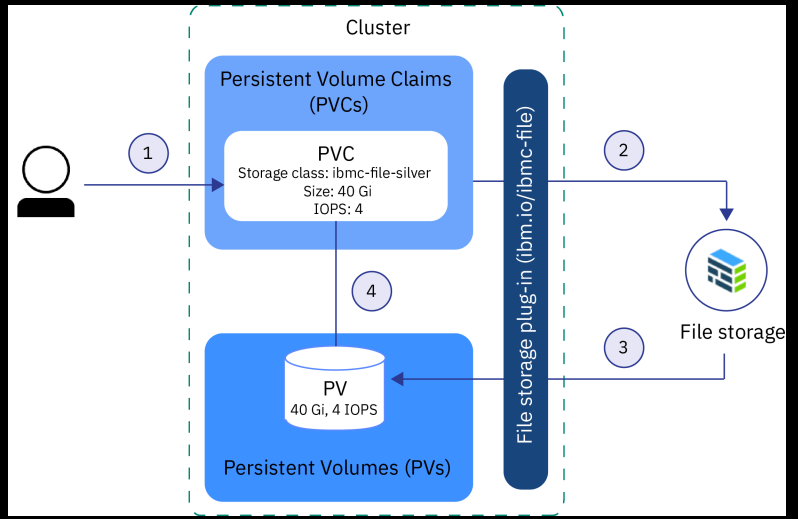
PersistentVolumes



PersistentVolumeClaims

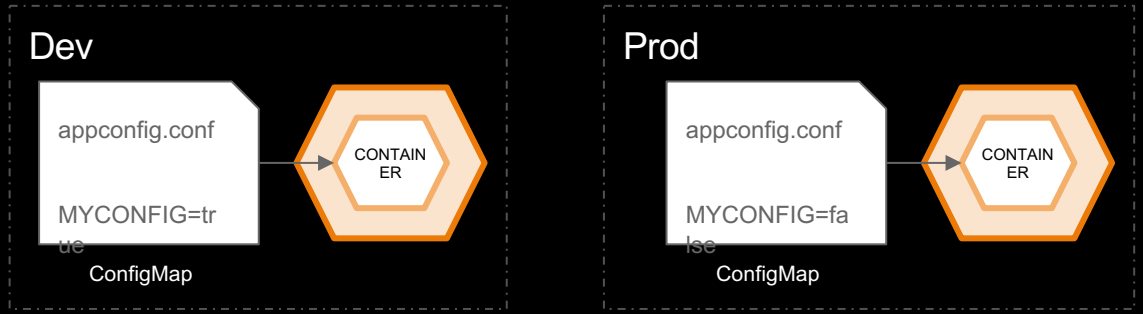


Storage Class



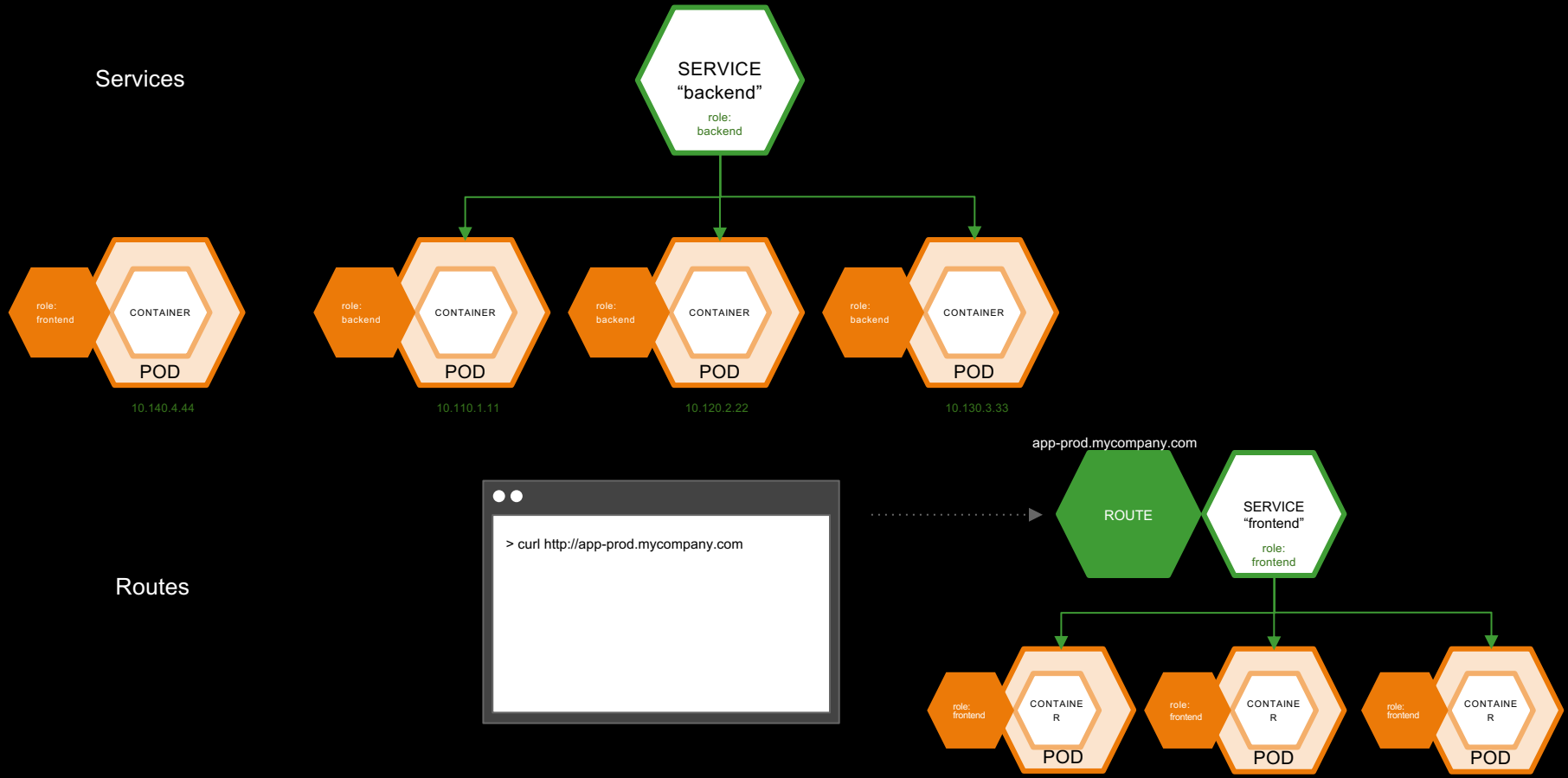
OpenShift Core Components - ConfigMaps and Secrets

ConfigMaps



Secrets

OpenShift Core Components - Services and Routes

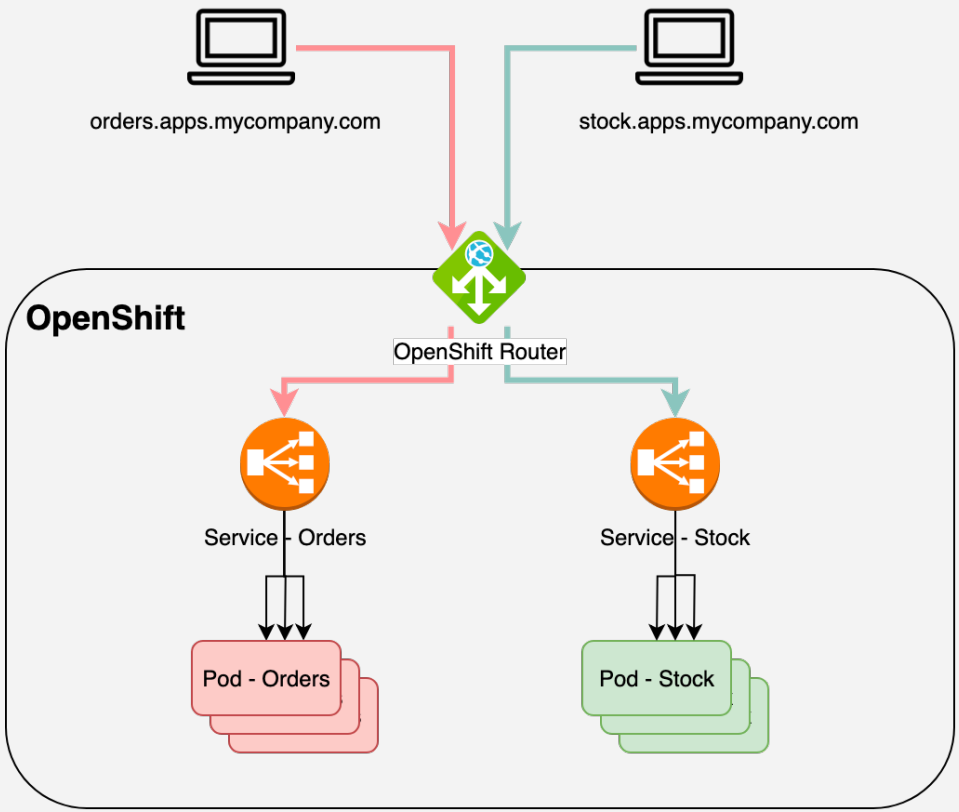


OpenShift Core Components - Routes

Routes are a specific OpenShift technology

Provides an easy way to expose a service

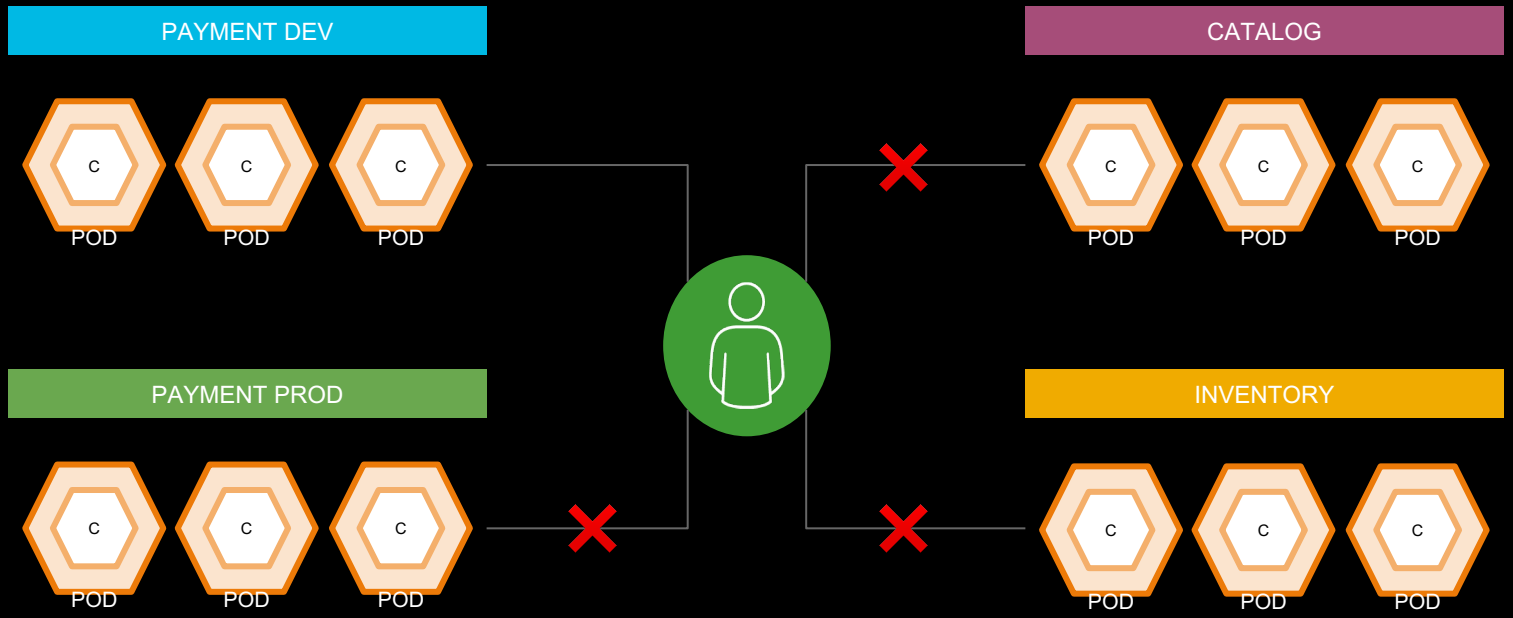
All exposed routes by default use the OpenShift Router at <name>.apps.mydomain.com



OpenShift Core Components - Projects

Projects == Kubernetes Namespaces

Projects isolate applications and resources within the cluster



OpenShift Core Components – Custom Resource Definitions

Custom Resource Definitions are extensions of the API Server

It defines your own object kinds and lets the API Server handle the entire lifecycle.

Typically paired with a custom controller

Specify a desired state of a collection of objects, controlled and monitored by the controller

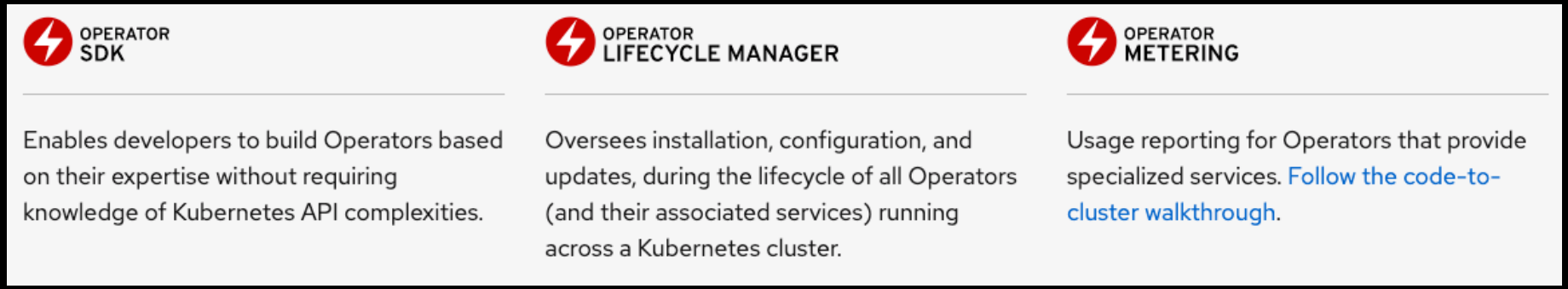
CRD’s form the basis of the operator life cycle for container native applications

OpenShift Operators

An Operator is a method of packaging, deploying and managing a Kubernetes-native application.

The Operator is a piece of software running in a Pod on the cluster, interacting with the Kubernetes API server. It introduces new object types through Custom Resource Definitions, an extension mechanism in Kubernetes. These custom objects are the primary interface for a user; consistent with the resource-based interaction model on the Kubernetes cluster.

The Operator Framework is an open source project that provides developers and cluster administrators tooling to accelerate development and deployment of an Operator.



https://github.com/operator-framework/getting-started

Examples of Operators in OpenShift

- ✓ Authentication
- ✓ Cluster Autoscaler
- ✓ DNS
- ✓ Cluster Monitoring
- ✓ Network
- ✓ Operator-lifecycle-manager
- ✓ Storage
- ✓ Cluster Logging
- ✓ Console
- ✓ Image Registry

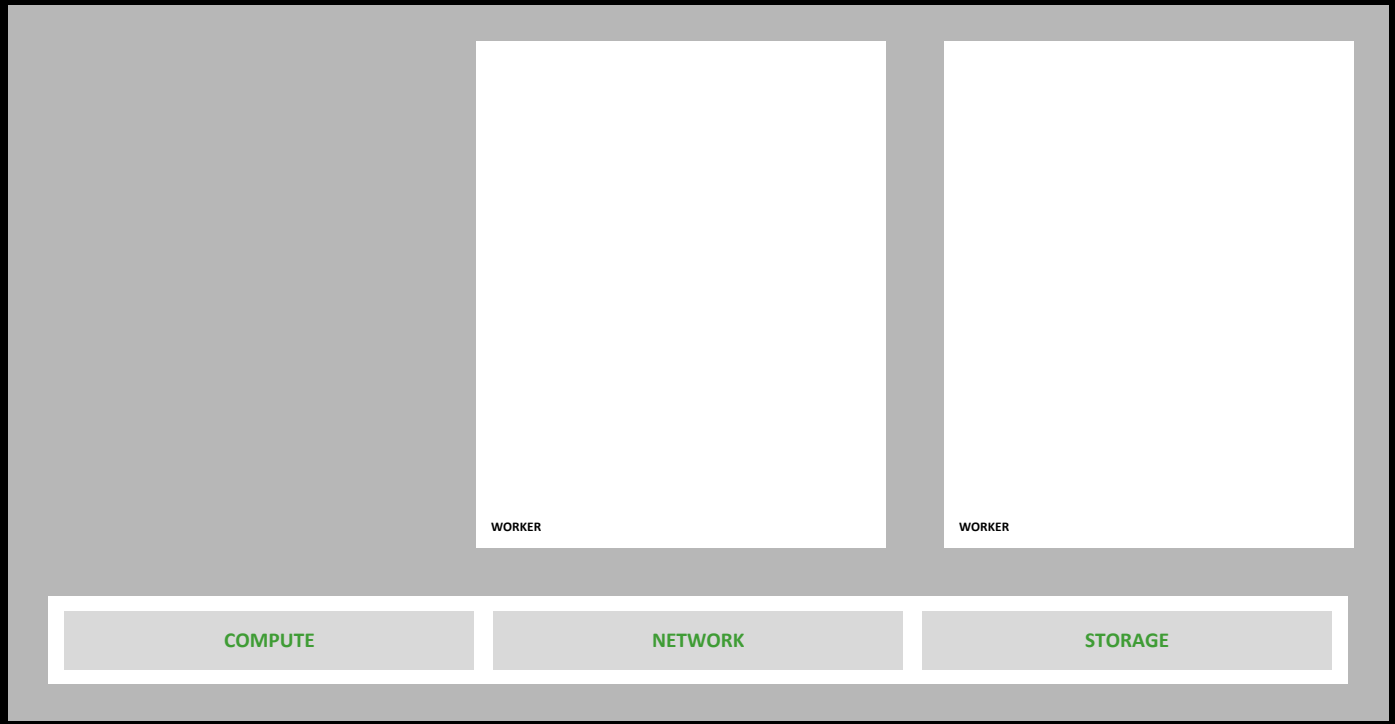
Demo – OpenShift User Interface and CLI

OpenShift 4 Architecture

You choose the infrastructure

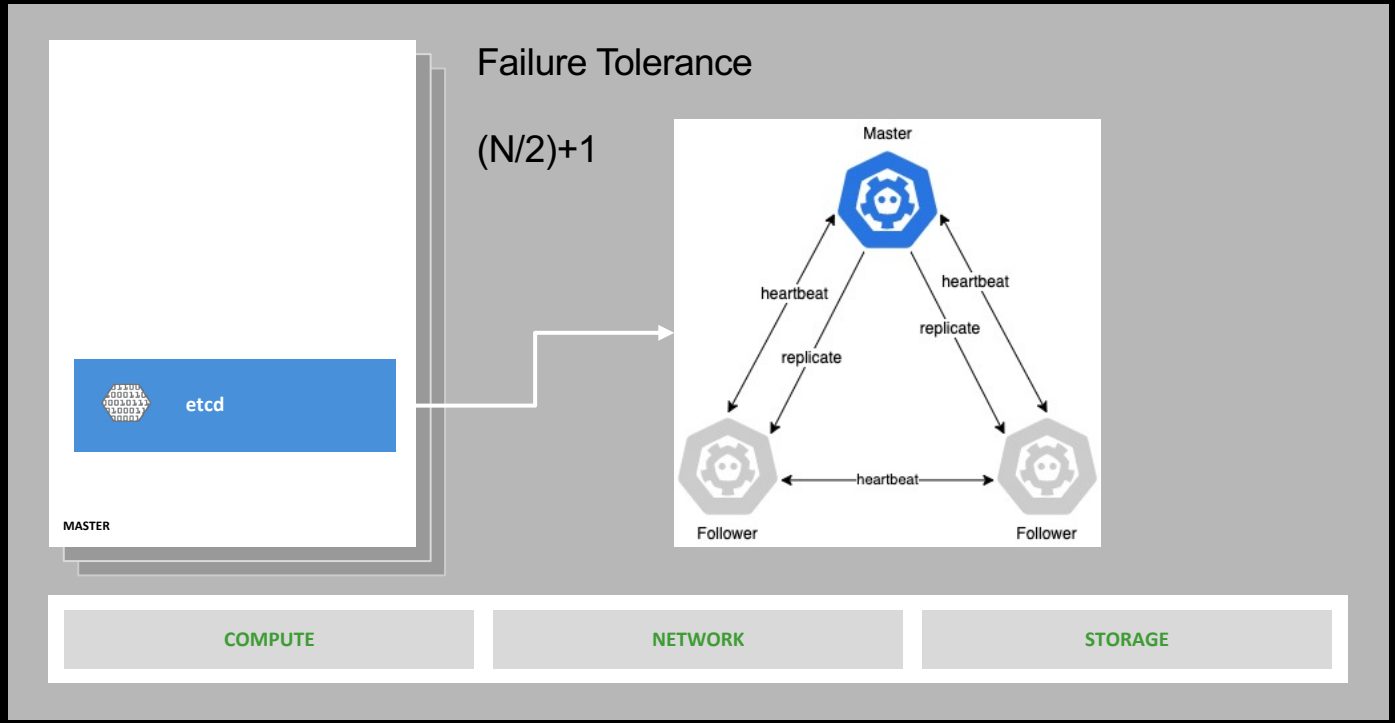
COMPUTE	NETWORK	STORAGE

Workers run workloads

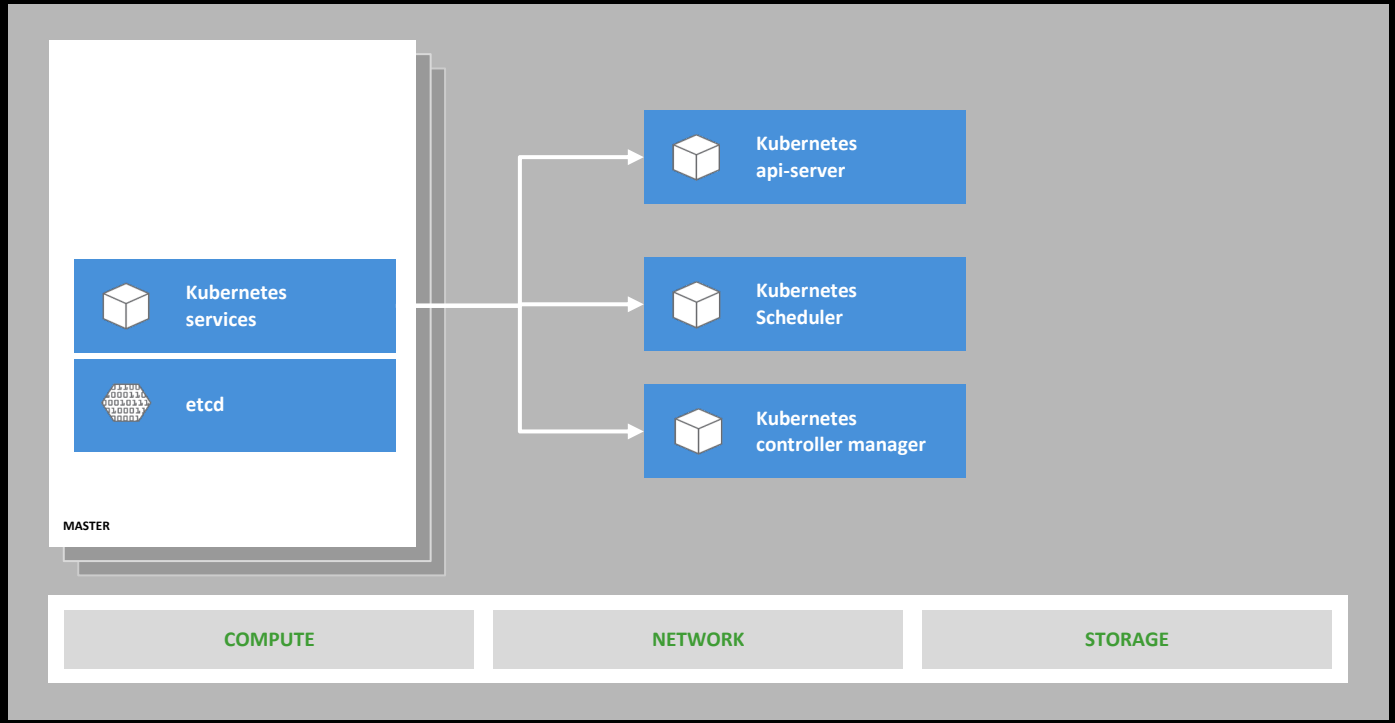


Masters are the control plane

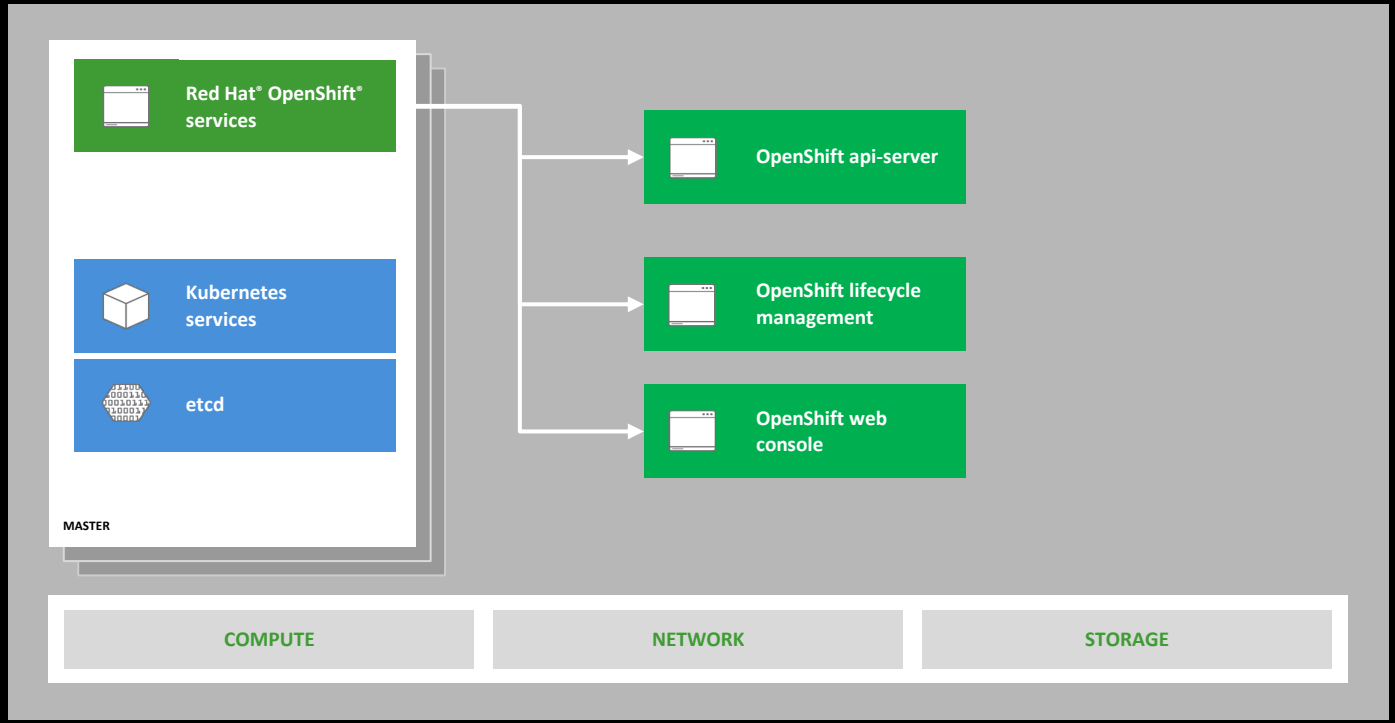
etcd stores the state of everything



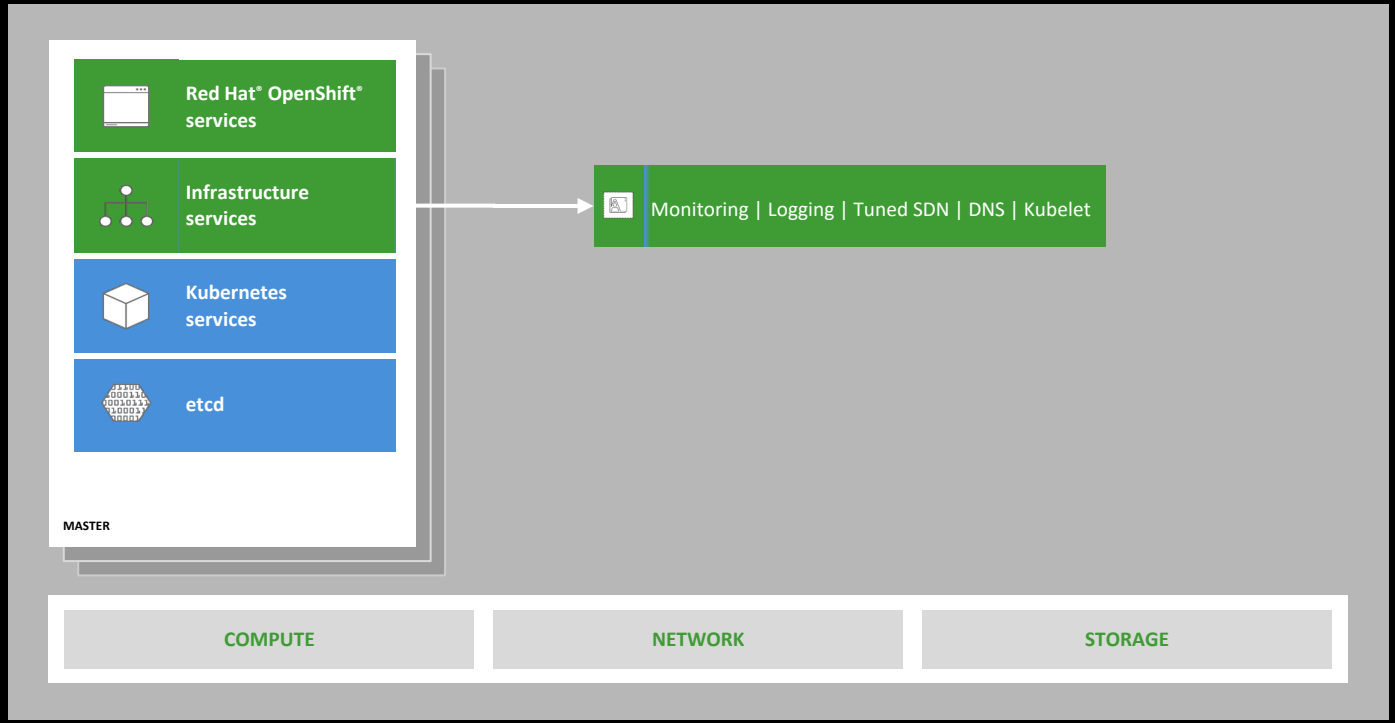
Core Kubernetes components



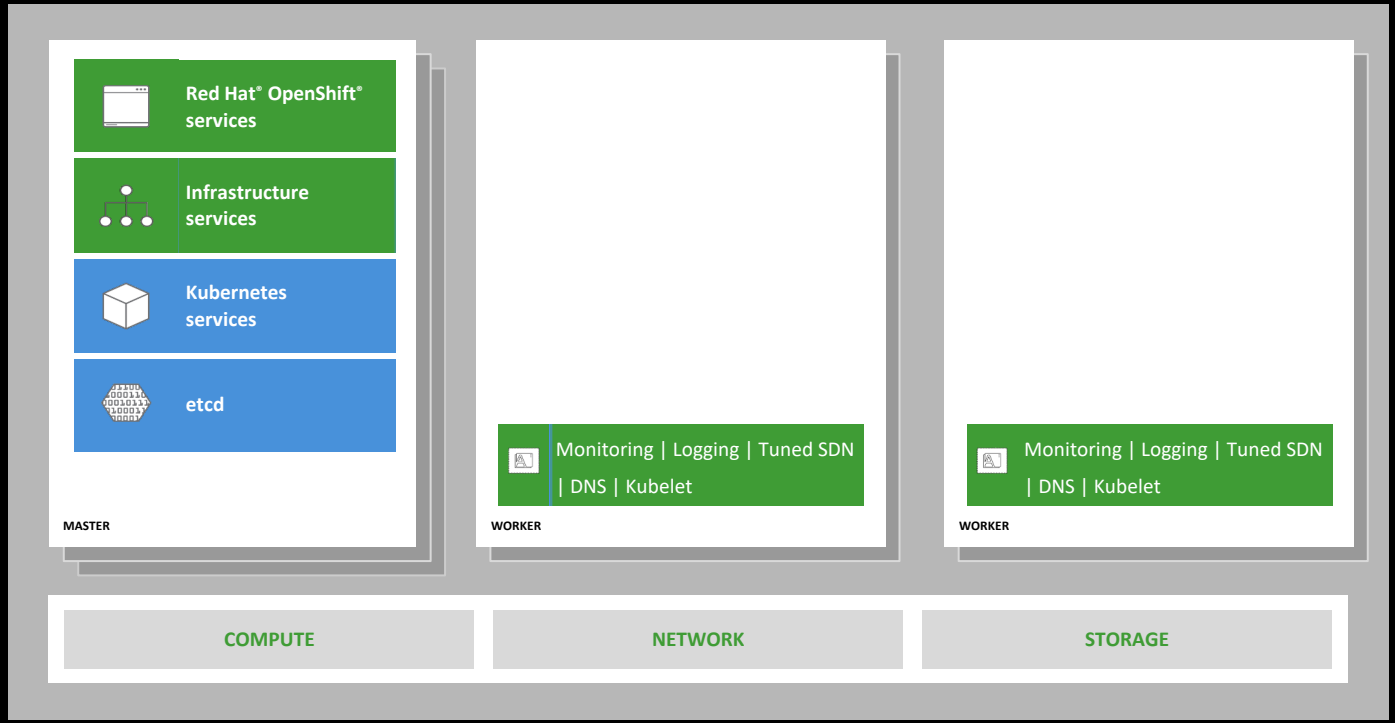
Core OpenShift components



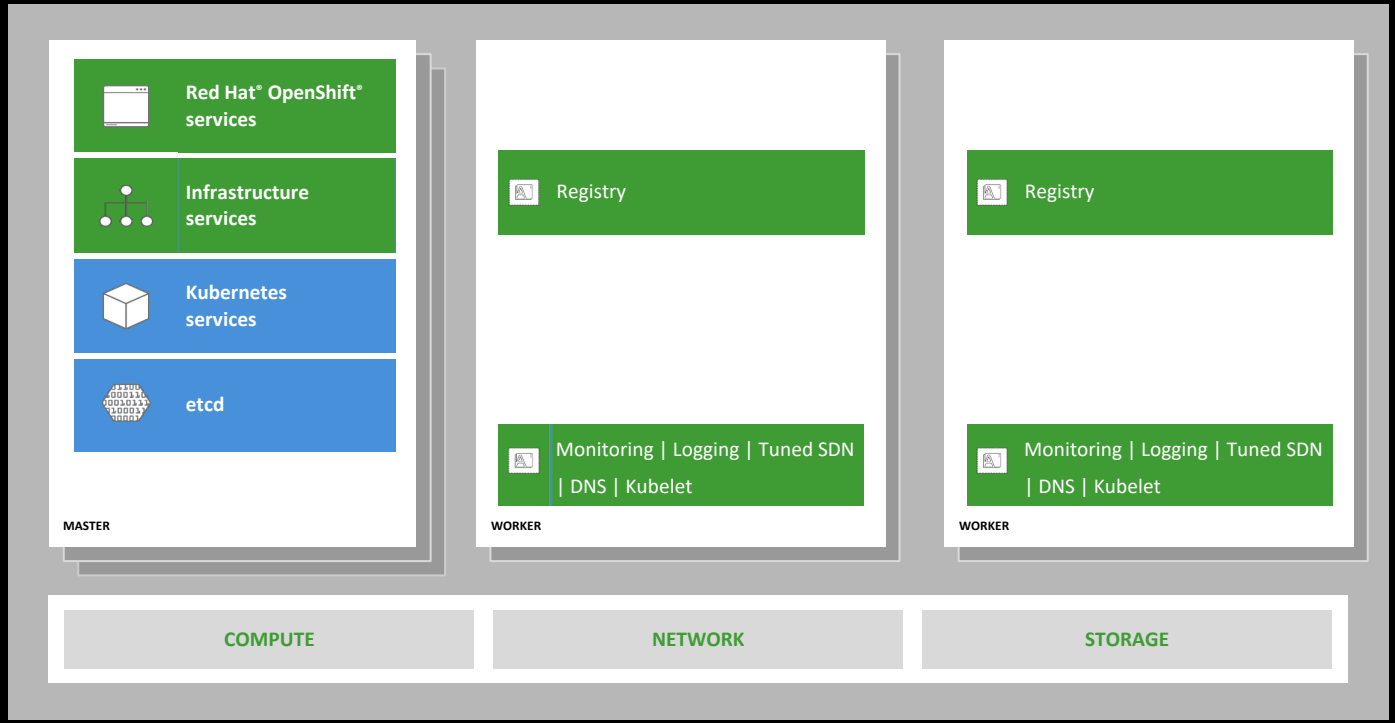
Internal and support infrastructure services



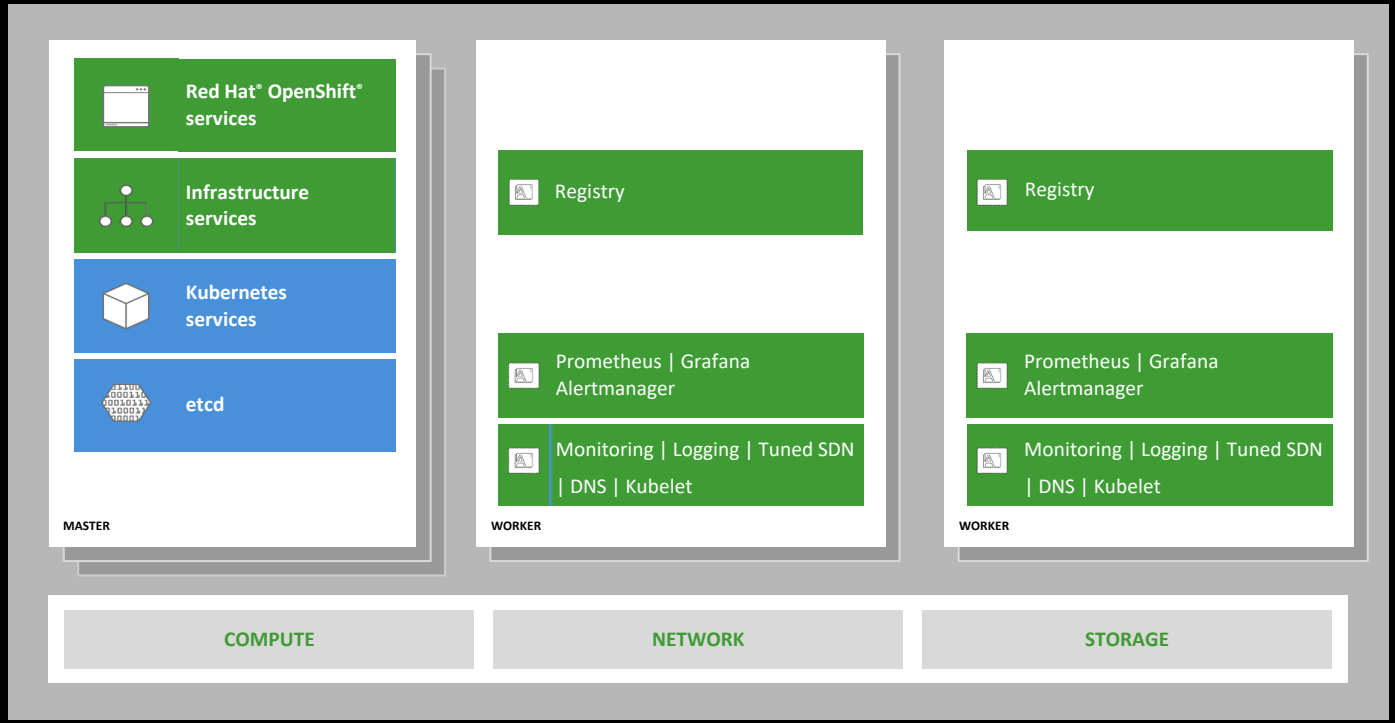
Runs on all hosts



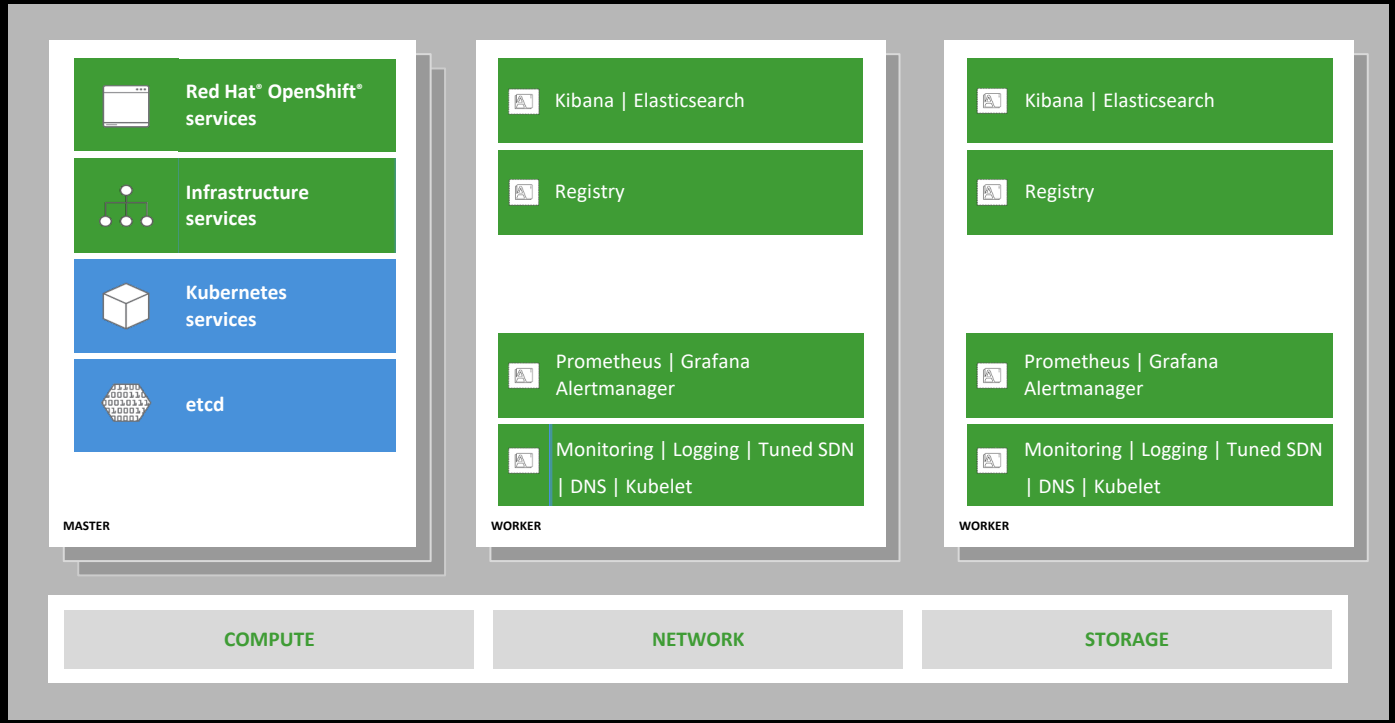
Integrated image registry



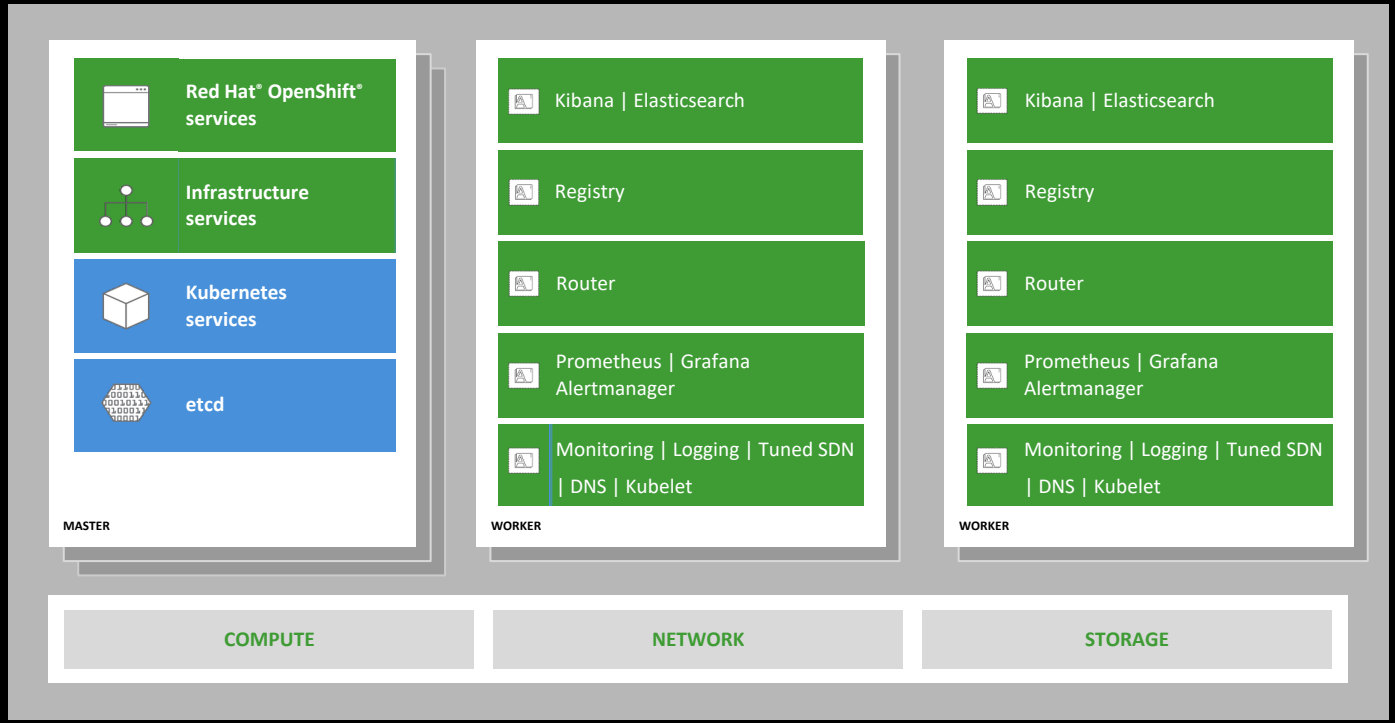
Cluster Monitoring



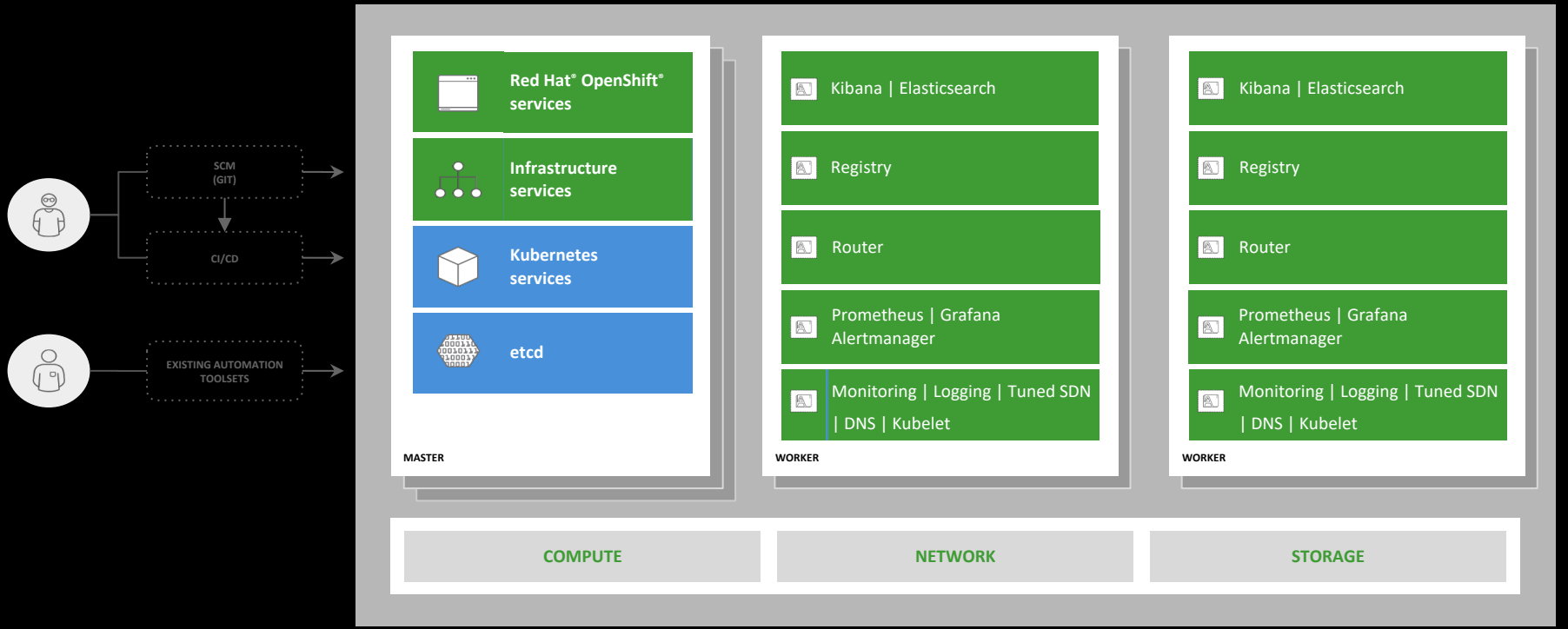
Log aggregation



Integrated routing



Dev and Ops via Web, CLI, API and IDE



Red Hat CoreOS

	RED HAT® ENTERPRISE LINUX®	RED HAT® ENTERPRISE LINUX CoreOS
BENEFITS	• 10+ year enterprise life cycle • Industry standard security • High performance on any infrastructure • Customizable and compatible with wide ecosystem of partner solutions	• Self-managing, over-the-air updates • Immutable and tightly integrated with OpenShift • Host isolation is enforced via Containers • Optimized performance on popular infrastructure
WHEN TO USE	When customization and integration with additional solutions is required	When cloud-native, hands-free operations are a top priority

Immutable Operating System

Red Hat Enterprise Linux CoreOS is versioned with OpenShift

CoreOS is tested and shipped in conjunction with the platform. Red Hat runs thousands of tests against these configurations.

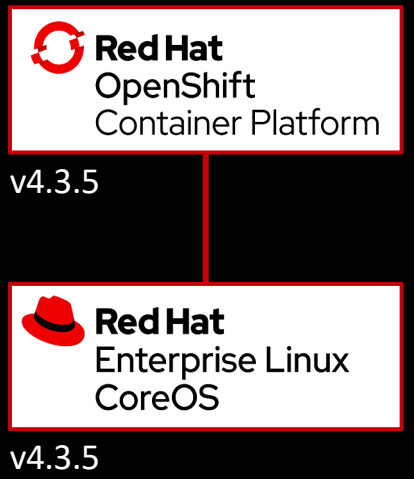
RHEL CoreOS admins are responsible for:

Nothing. 😀🙌

Red Hat Enterprise Linux CoreOS is managed by the cluster

The Operating system is operated as part of the cluster, with the config for components managed by Machine Config Operator:

- CRI-O config
- Kubelet config
- Authorized registries
- SSH config



Common Architectures

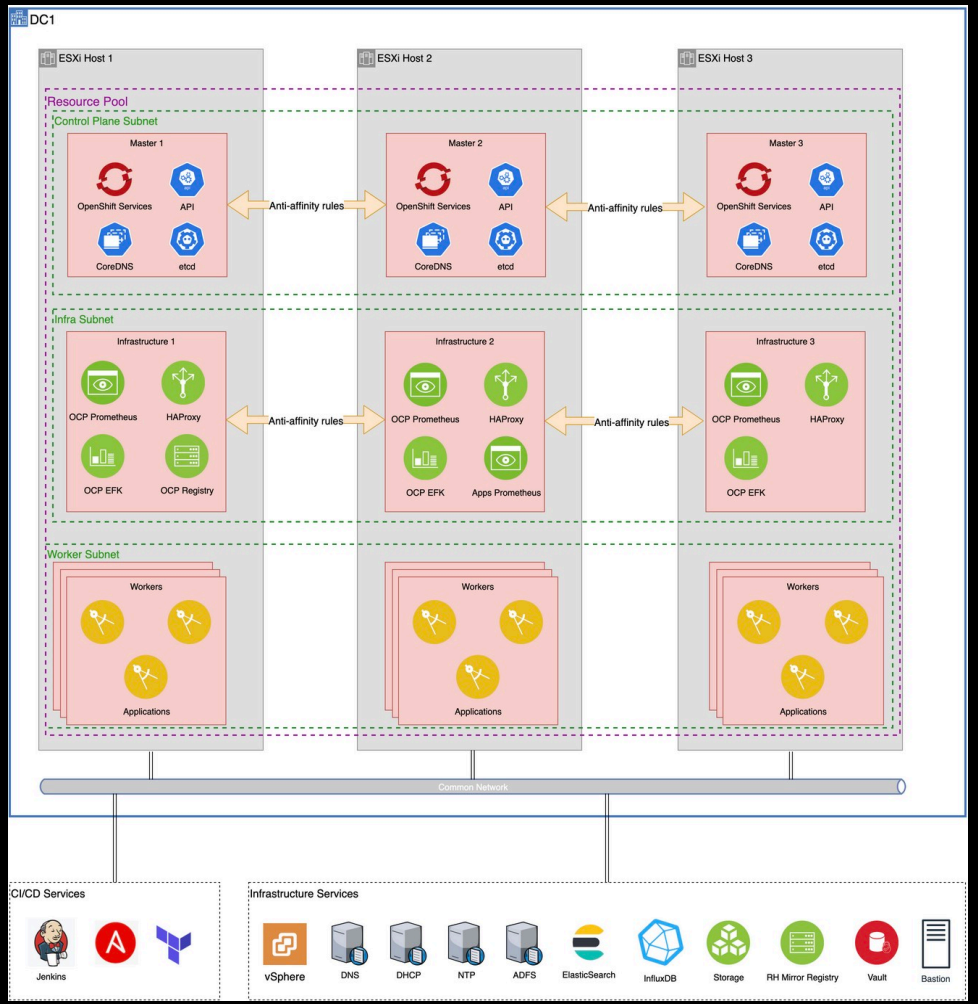
Common Architectures

Single cluster, single datacenter

A single cluster is distributed within one datacenter

Control plane is distributed across ESXi hosts/availability zones

Implements HA/DR measures to protect against host failure



Common Architectures

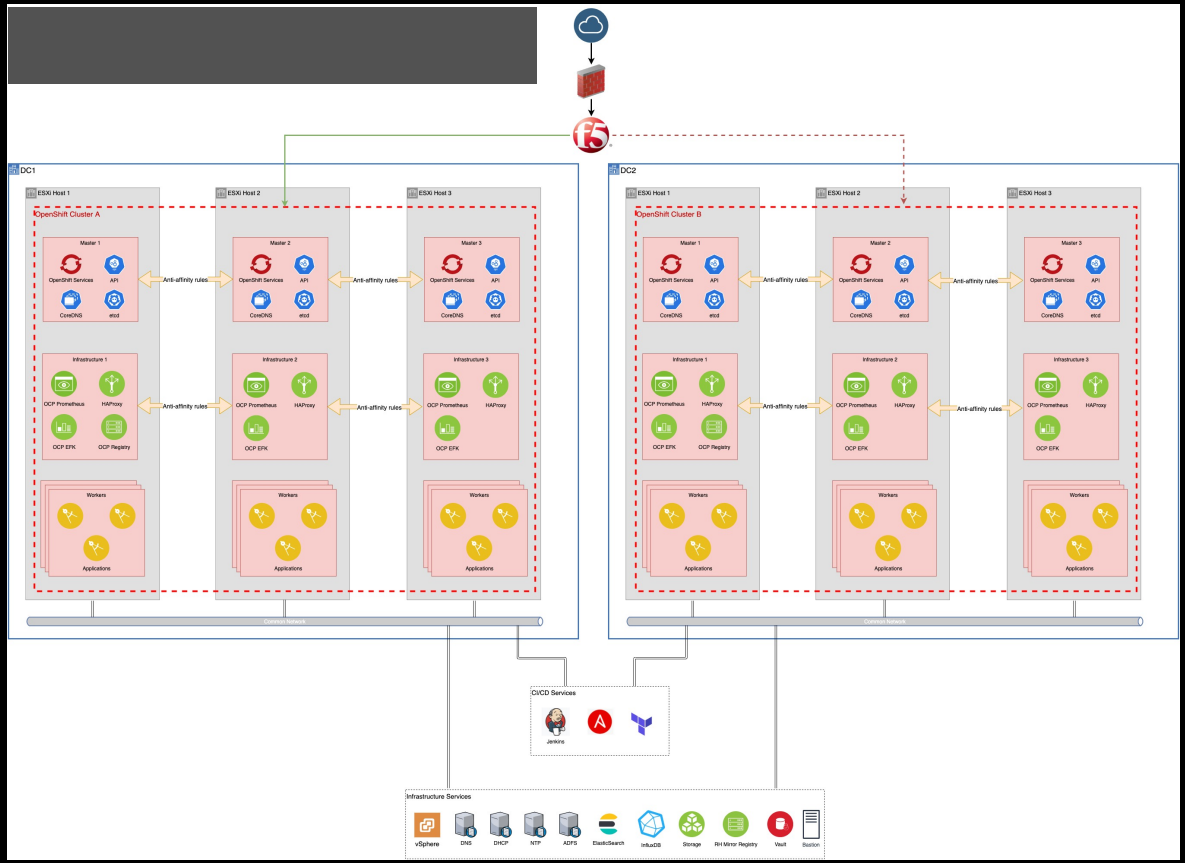
Two clusters, multiple datacentres

Multiple clusters are deployed to different datacenters in an active/active or active/passive approach

Traffic is routed to the active cluster via a global load balancer

Implements the same principles as the single cluster, single datacenter strategy

Implements HA/DR measures to protect against site failure



Common Architectures

Single cluster, multiple datacentres (stretched)

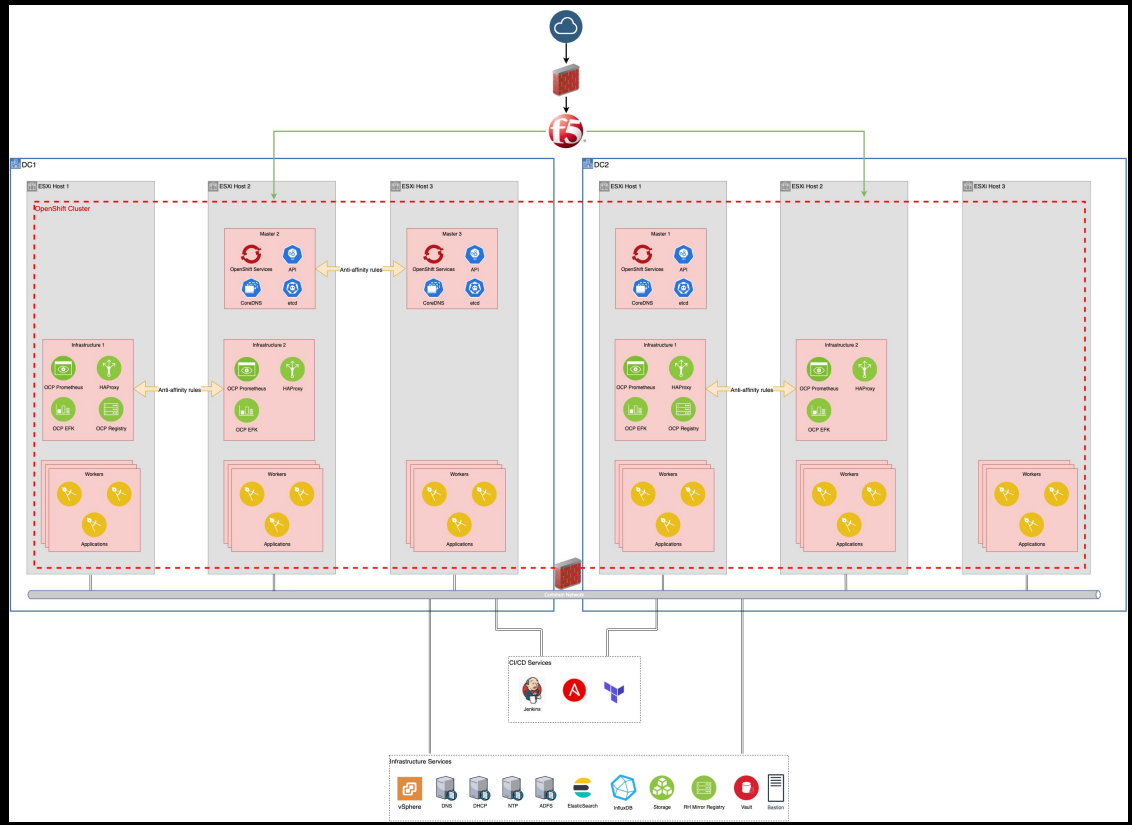
Single cluster deployed across two datacenters with stretched networking approach

Requires low latency

Intra-datacenter implements the same principles as the single cluster, single datacenter strategy

Implements HA/DR measures to protect against site failure

Datacenter hosting both masters is a weak point



Some Considerations for Designing OpenShift Clusters

Workload and Application considerations

What availability is needed for the application?

RPO and RTO for the application?

What are your workload requirements?

Platform Considerations

Are we understanding the capacity needs? Do we need N+1 or N+2?

What tooling is foreseen to replicate the data between prod and DR?

What are the external integrations required? Logging, monitoring, storage?

Container considerations

How is the traffic sprayed over the HA components?

Do we allow DR for individual applications or only for the cluster?

Infrastructure considerations

How is the HA foreseen for the infrastructure?

Do we have a dark DR site or an active one?

Does it need to scale?

Single cluster, or multiple clusters?

Organization considerations

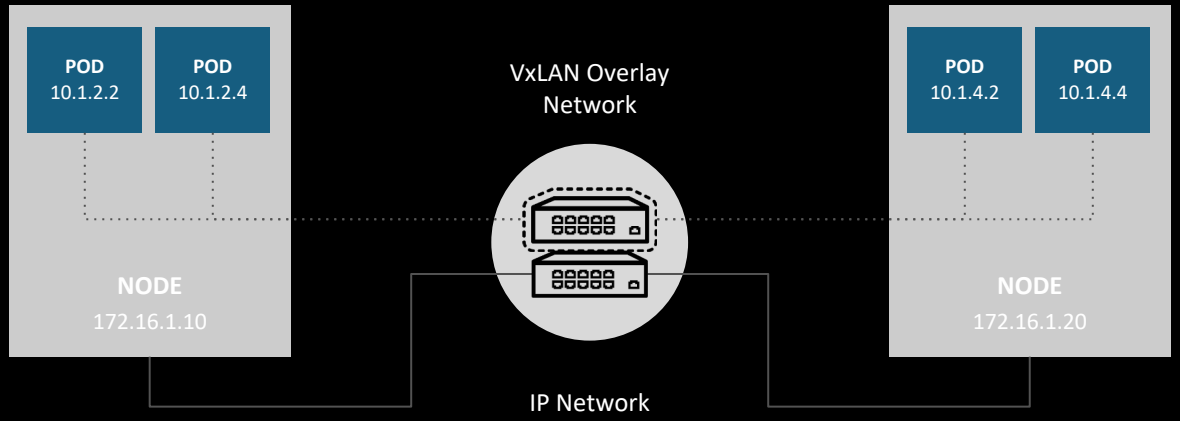
Who is managing and monitoring the platform for availability?

Do we apply regular switches between prod and DR?

What are the High Availability and Disaster Recovery strategies?

OpenShift Networking

OpenShift Networking



OpenShift Networking - OpenShift SDN

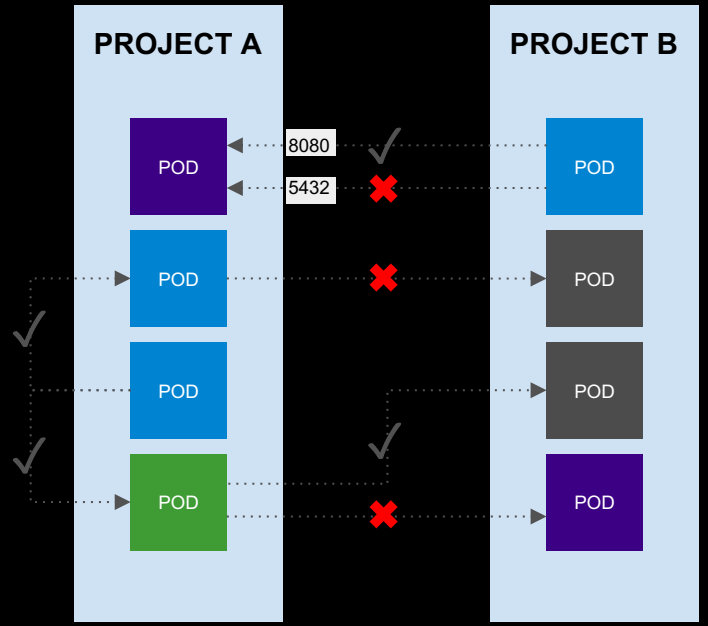
OpenShift uses software-defined-networking approach for a unified cluster network, which enables pod-to-pod communication.

OpenShift SDN is the default Container Network Interface (CNI)

Governed by the Cluster Network Operator (CNO)

Network Policy	Multitenant	Subnet
Allows project administrators to configure their own isolation policies using NetworkPolicy objects	Provides project-level isolation for Pods and Services	Provides a flat Pod network

OpenShift Networking
- Network Policy



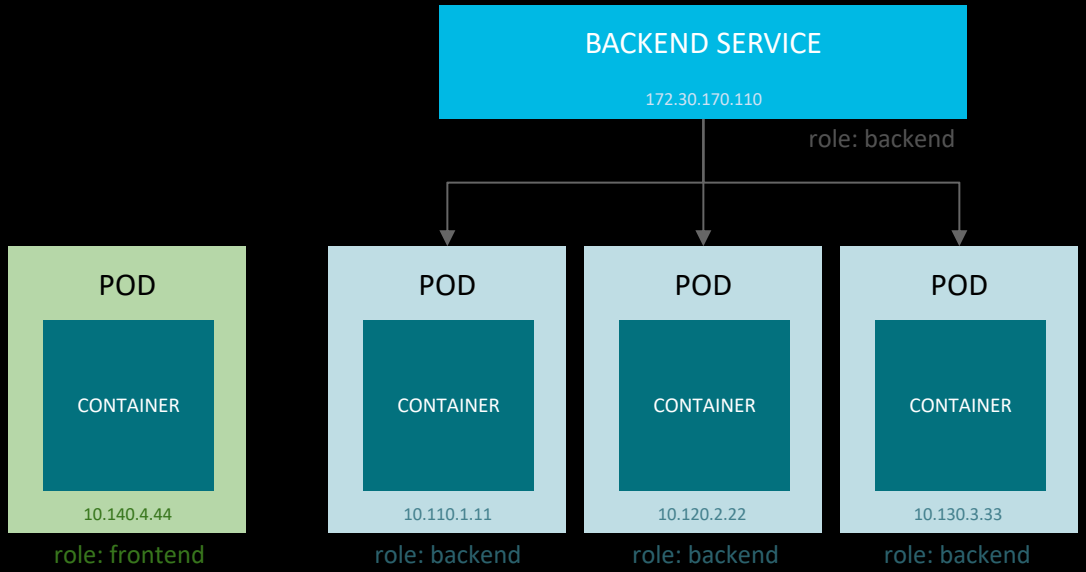
Example Policies

- Allow all traffic inside the project
- Allow traffic from green to grey
- Allow traffic to purple on 8080

```
apiVersion: extensions/v1beta1
kind: NetworkPolicy
metadata:
  name: allow-to-purple-on-8080
spec:
  podSelector:
    matchLabels:
      color: purple
  ingress:
  - ports:
    - protocol: tcp
      port: 8080
```

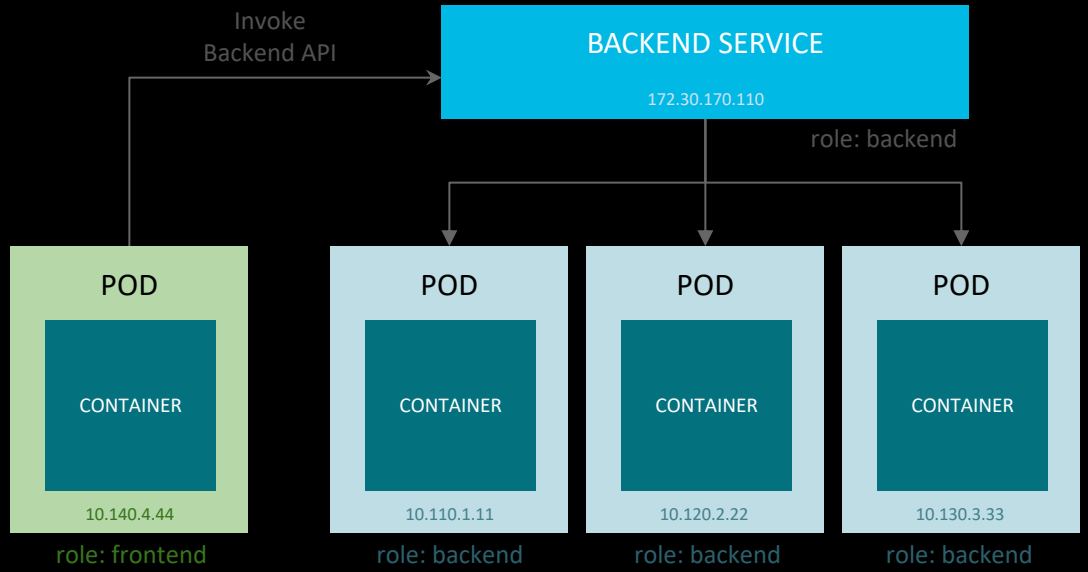
OpenShift Networking - Services

Services provide internal load-balancing and service discovery across pods



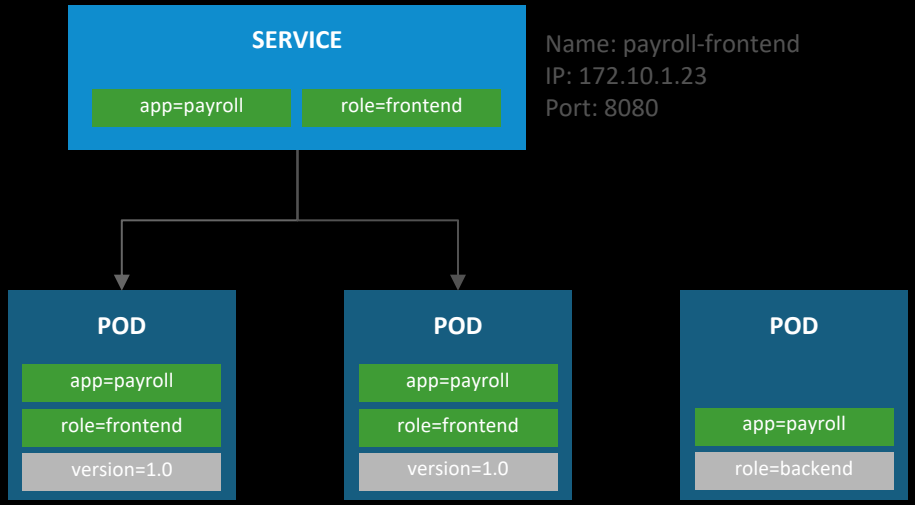
OpenShift Networking - Services

Apps can talk to each other via services



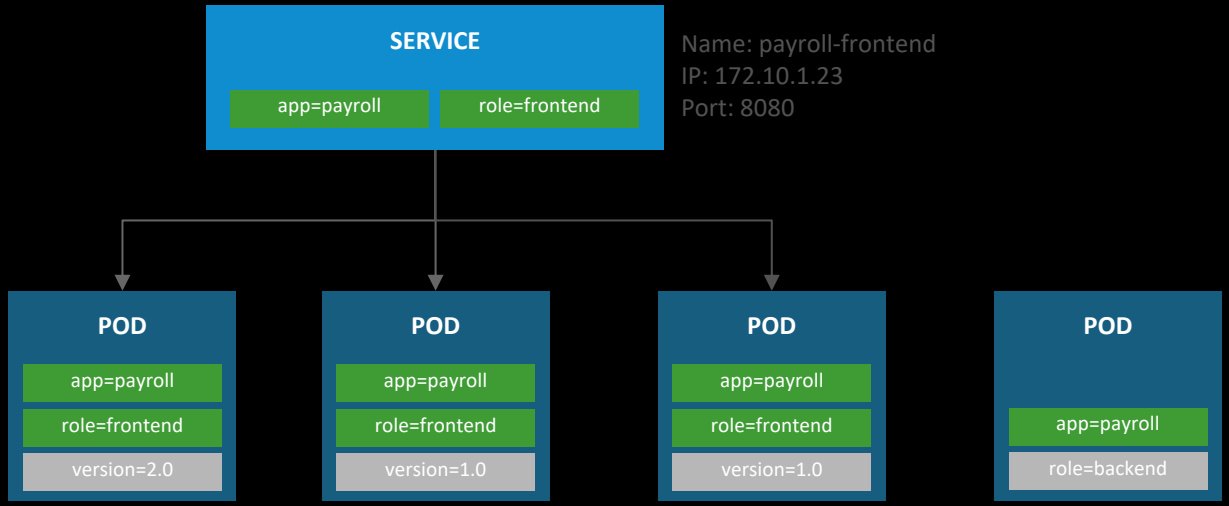
OpenShift Networking
- Service Discovery

Built-in service discovery and internal load-balancing



OpenShift Networking
- Service Discovery

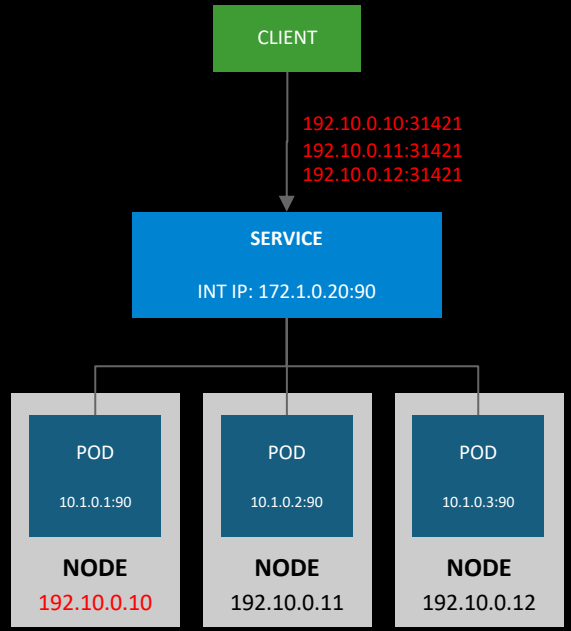
Built-in service discovery and internal load-balancing



OpenShift Networking - NodePort

External traffic to a service on a random port with NodePort

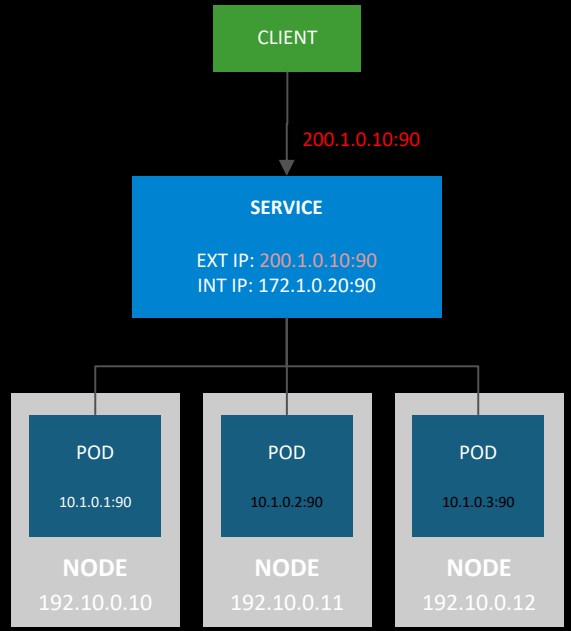
- NodePort binds a service to a unique port on all the nodes
- Traffic received on any node redirects to a node with the running service
- Ports in 30K-60K range which usually differs from the service
- Firewall rules must allow traffic to all nodes on the specific port



OpenShift Networking - External Traffic

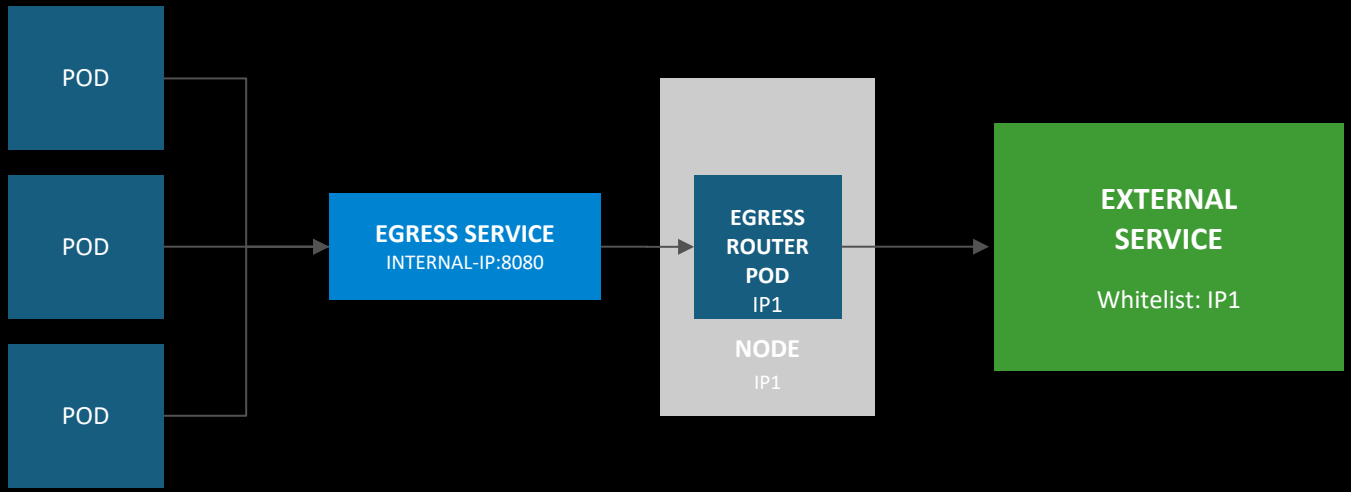
External traffic to a service with Ingress

- Access a service with an external IP on any TCP/UDP port, such as
  - Databases
  - Message Brokers
- Automatic IP allocation from a predefined pool using Ingress IP Self-Service
- IP failover pods provide high availability for the IP pool



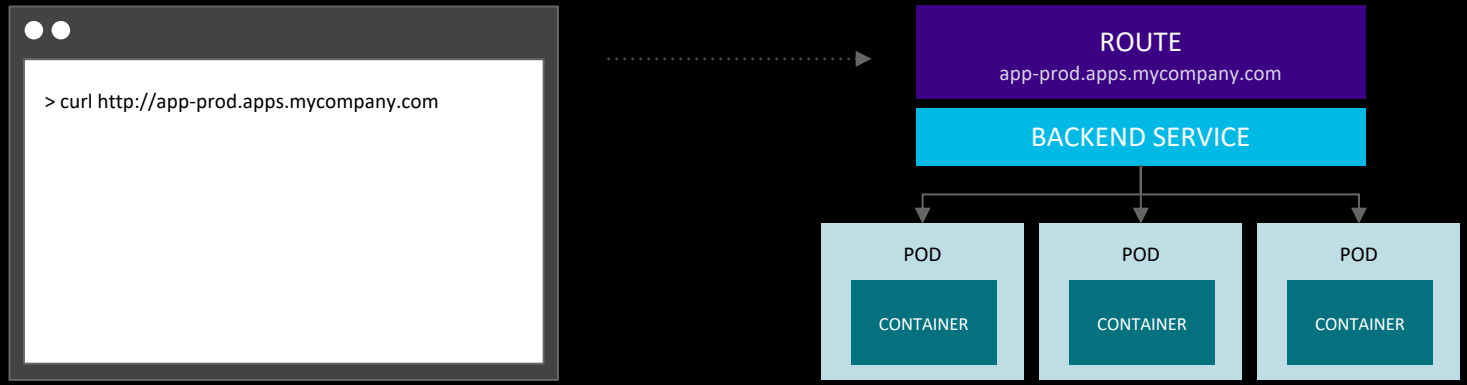
OpenShift Networking
- Egress Traffic

Control outgoing traffic source IP with Egress Router



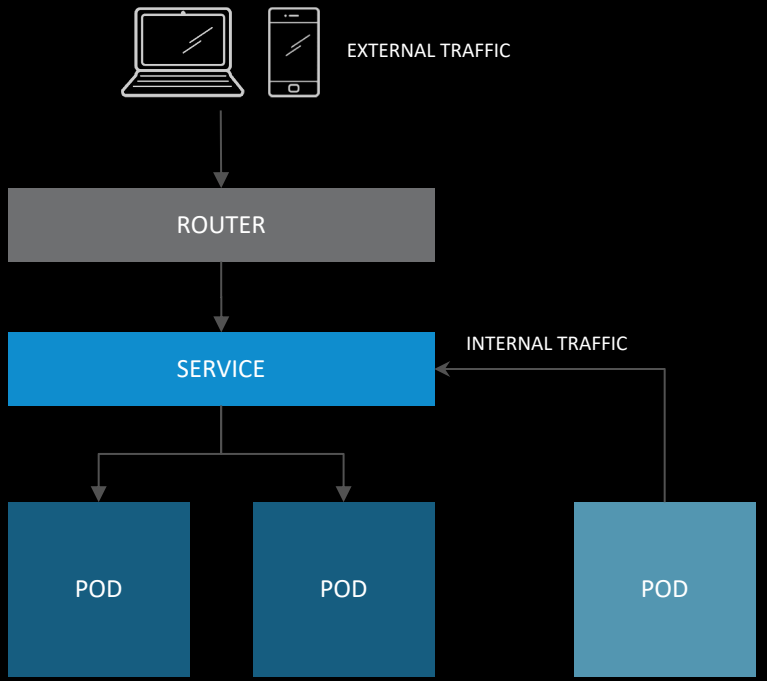
OpenShift Networking - Routes

Routes add services to the external load-balancer and provide readable URLs for the app



OpenShift Networking - Routes

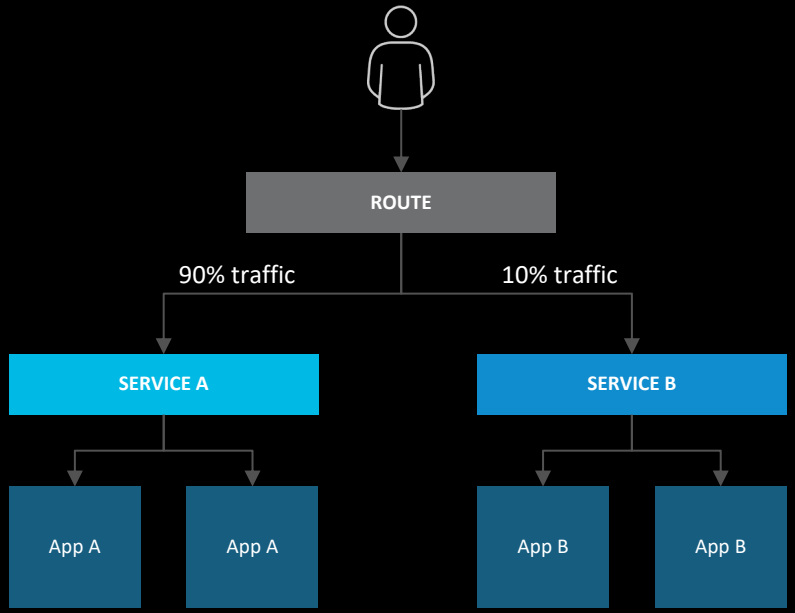
Route exposes service internally



OpenShift Networking - Route Traffic Splitting

Route Split Traffic

Split Traffic Between Multiple Services For A/B Testing, Blue/Green and Canary Deployments



OpenShift Networking
- Multiple Pod Networks

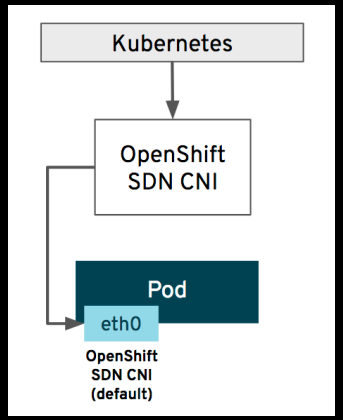
Can add multiple networks to a pod using Multus

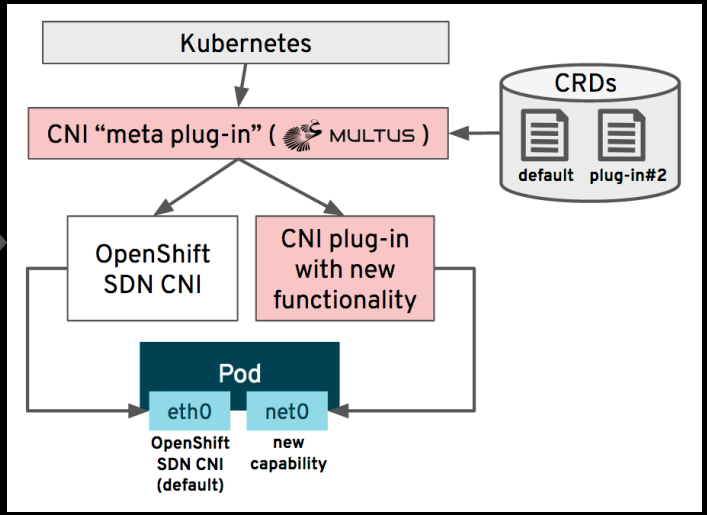
Use Cases

- Performance
- Security

Additional Networks

- bridge
- host-device
- macvlan
- ipvlan
- SR-IOV





OpenShift Security

OpenShift Security

CONTROL
Application Security

- Container Content
- CI/CD Pipeline
- Container Registry
- Deployment Policies

DEFEND
Infrastructure

- Container Platform
- Container Host Multi-tenancy
- Network Isolation
- Storage
- Audit & Logging
- API Management

EXTEND

- Security Ecosystem

OpenShift Security

To make the correct security design decisions we need to understand:

The type of workloads the client is going to run

The type of external interactions we need for the applications

The way of interacting with the platform and which interfaces they are going to use.

The different integrations for IAM and other security components

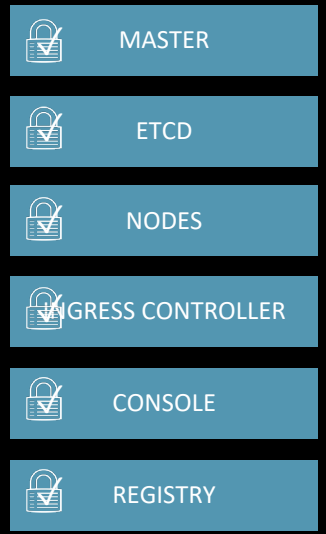
The isolation and segmentation is needed for the different groups/teams

The data protection needs for the different workloads

How containers need to be secured, monitored for mutations
Etc.

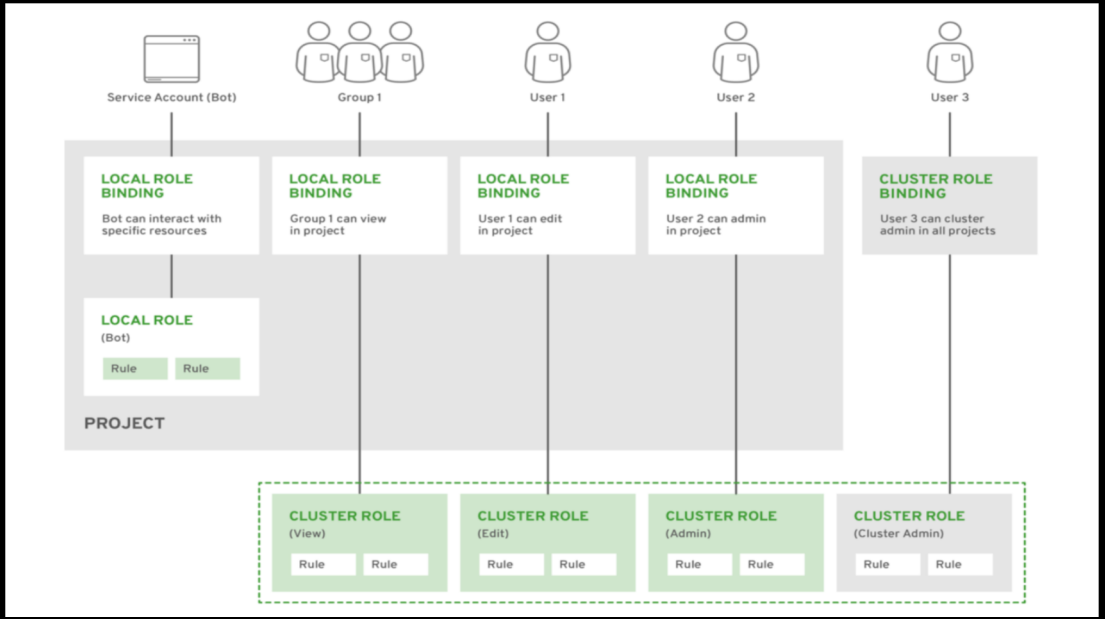
OpenShift Security
- Certificates and Certificate Management

- OpenShift provides its own internal CA
- Certificates are used to provide secure connections to
  - master (APIs) and nodes
  - Ingress controller and registry
  - etcd
- Certificate rotation is automated
- Optionally configure external endpoints to use custom certificates

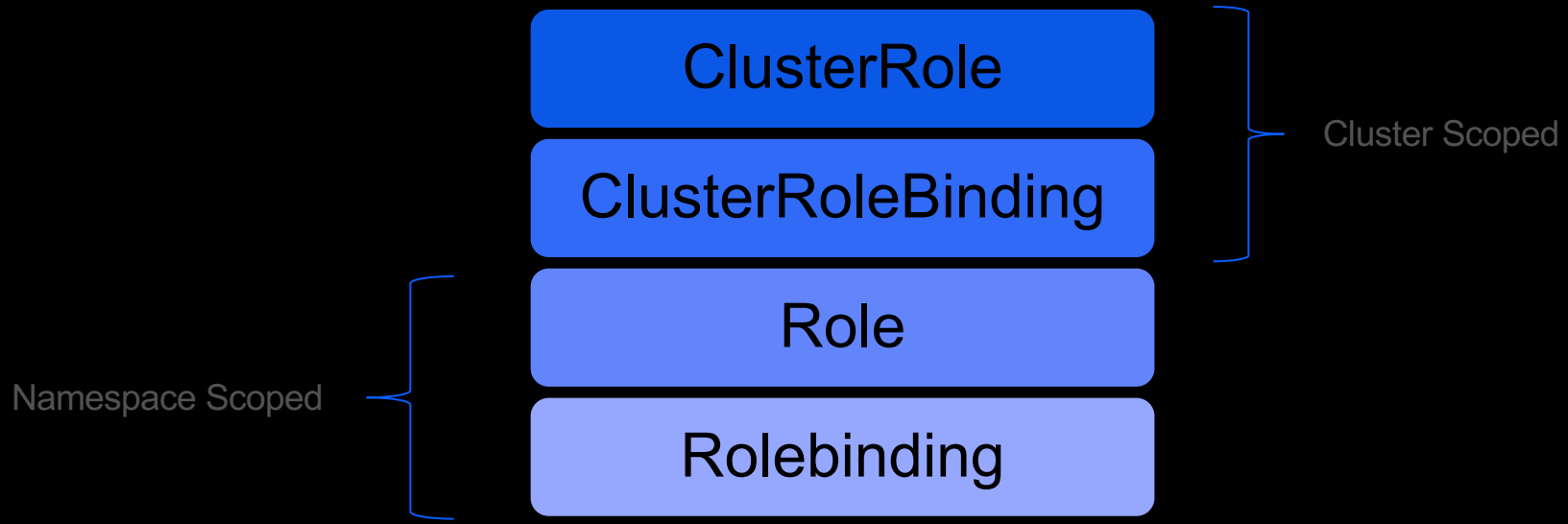


OpenShift Security
- Role Based Access Control (RBAC)

- Project scope & cluster scope available
- Matches request attributes (verb,object,etc)
- If no roles match, request is denied (deny by default)
- Operator- and user-level roles are defined by default
- Custom roles are supported



OpenShift Security
- Role Based Access Control (RBAC)



OpenShift Security
- Role Based Access Control (RBAC)

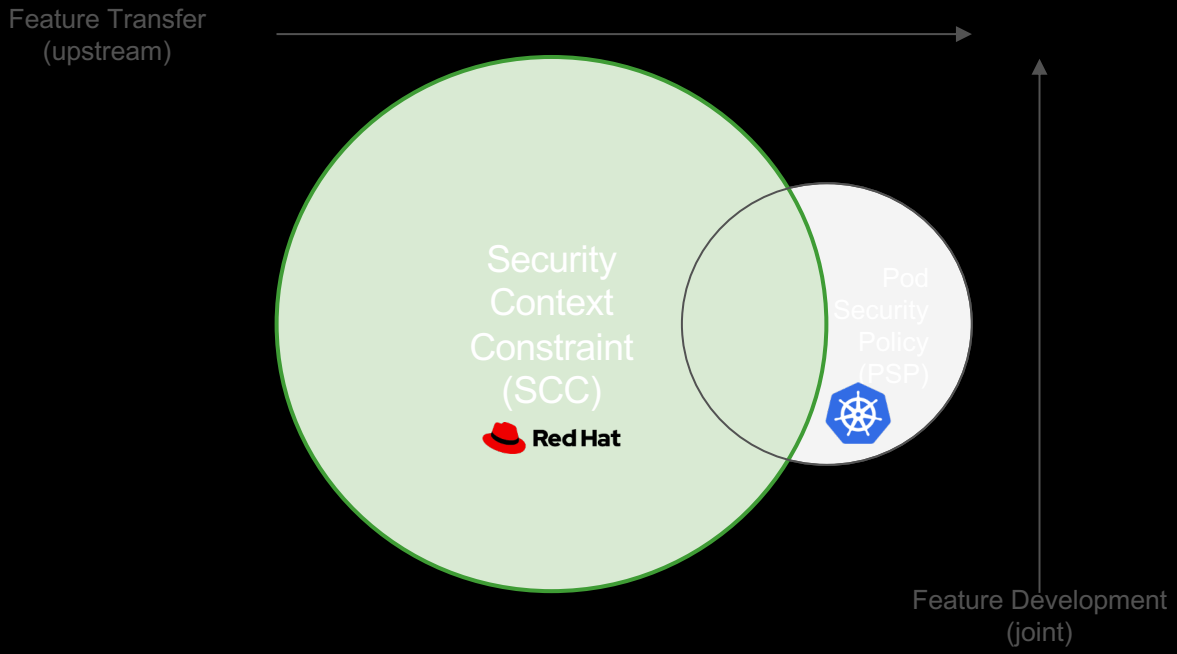
Default Cluster Role	Description
admin	A project manager. If used in a local binding, an admin has rights to view any resource in the project and modify any resource in the project except for quota.
basic-user	A user that can get basic information about projects and users.
cluster-admin	A super-user that can perform any action in any project. When bound to a user with a local binding, they have full control over quota and every action on every resource in the project.
cluster-status	A user that can get basic cluster status information.
edit	A user that can modify most objects in a project but does not have the power to view or modify roles or bindings.
self-provisioner	A user that can create their own projects.
view	A user who cannot make any modifications, but can see most objects in a project. They cannot view or modify roles or bindings.

OpenShift Security
- Security Context Constraints

SCCs allow an administrator to control:

- Whether a pod can run privileged containers.
- The capabilities that a container can request.
- The use of host directories as volumes.
- The SELinux context of the container.
- The container user ID.
- The use of host namespaces and networking.
- The allocation of an FSGroup that owns the pod's volumes.
- The configuration of allowable supplemental groups.
- Whether a container requires the use of a read only root file system.
- The usage of volume types.
- The configuration of allowable seccomp profiles.

OpenShift Security
- Security Context Constraints



OpenShift Security - etcd Encryption

By default, etcd data is not encrypted in OpenShift Container Platform.

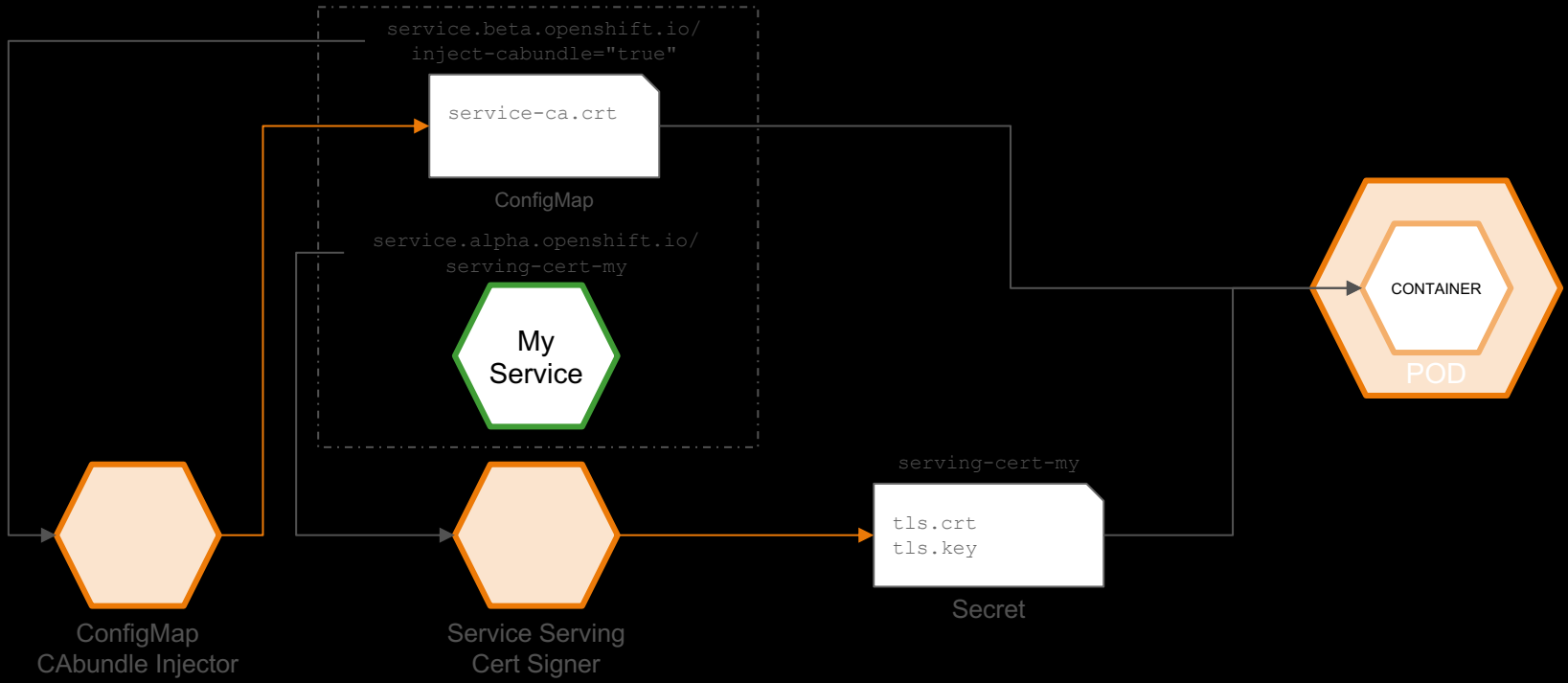
When you enable etcd encryption, the following OpenShift API server and Kubernetes API server resources are encrypted:

- Secrets
- ConfigMaps
- Routes
- OAuth access tokens
- OAuth authorize tokens

When you enable etcd encryption, encryption keys are created.

These keys are rotated on a weekly basis. You **must** have these keys in order to restore from an etcd backup.

OpenShift Security
- Service Certificates



OpenShift Security
- Network Security

A network policy is a specification of how groups of pods are allowed to communicate with each other and other network endpoints.

A default installation of OpenShift allows all pods to communicate with all other pods in the system - there is no pre-defined network segregation unless you select the multitenant mode

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: front-allow
  namespace: development
spec:
  podSelector:
    matchLabels:
      role: backend
      app: my-app
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: webfront
          app: my-app
    ports:
    - protocol: TCP
      port: 3306
```

OpenShift Security
- Network Security

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

podSelector: Each NetworkPolicy includes a podSelector which selects the grouping of pods to which the policy applies.

policyTypes: Each NetworkPolicy includes a policyTypes list which may include either Ingress, Egress, or both. The policyTypes field indicates whether or not the given policy applies to ingress traffic to selected pod, egress traffic from selected pods, or both.

ingress: Each NetworkPolicy may include a list of whitelist ingress rules. Each rule allows traffic which matches both the from and ports sections.

egress: Each NetworkPolicy may include a list of whitelist egress rules. Each rule allows traffic which matches both the to and ports sections.

OpenShift DNS

Built-in internal DNS to reach services by name

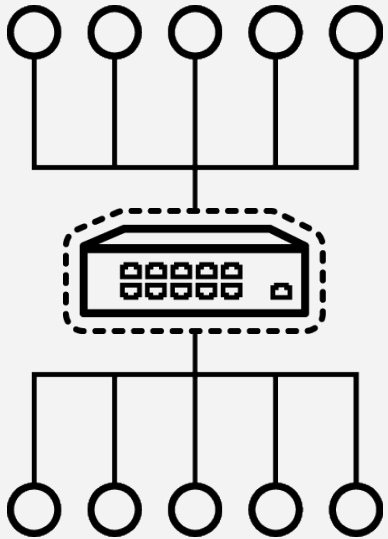
Split DNS is supported via DNSmasq

- Master answers DNS queries for internal services
- Other name servers serve the rest of the queries

You can use DNS forwarding to override the forwarding configuration identified in `etc/resolv.conf` on a per-zone basis by specifying which name server should be used for a given zone.

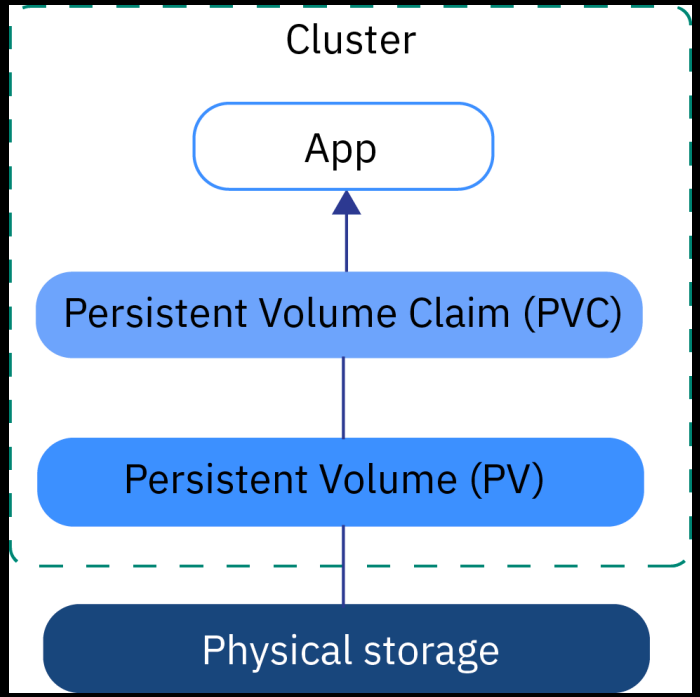
`my-svc.my-namespace.svc.cluster.local`

The DNS Operator deploys and manages CoreDNS to provide a name resolution service to pods, enabling DNS-based Kubernetes Service discovery in OpenShift.



OpenShift Storage

OpenShift Storage



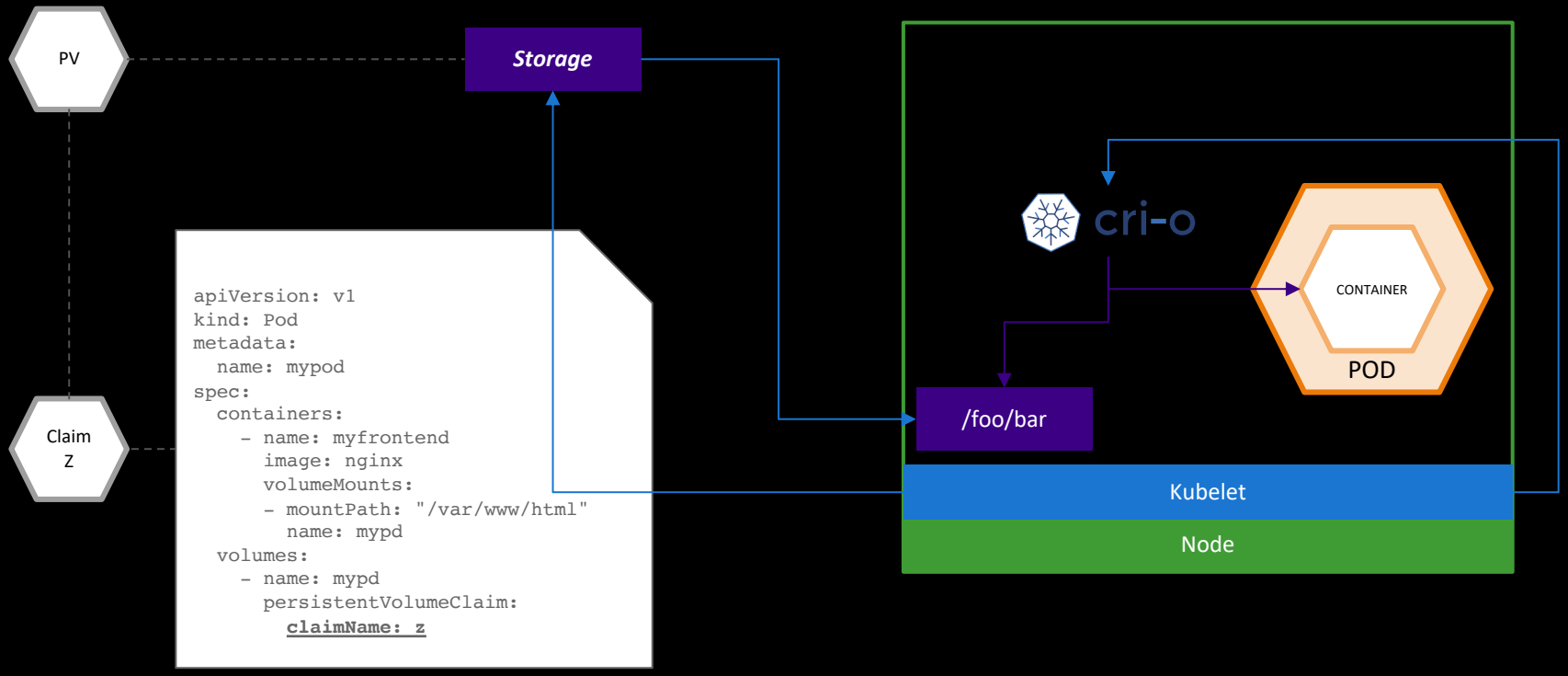
OpenShift leverages Kubernetes Persistent Volumes

Persistent Volume (PV) is a piece of storage, provisioned by an administrator or dynamically provisioned using Storage Classes

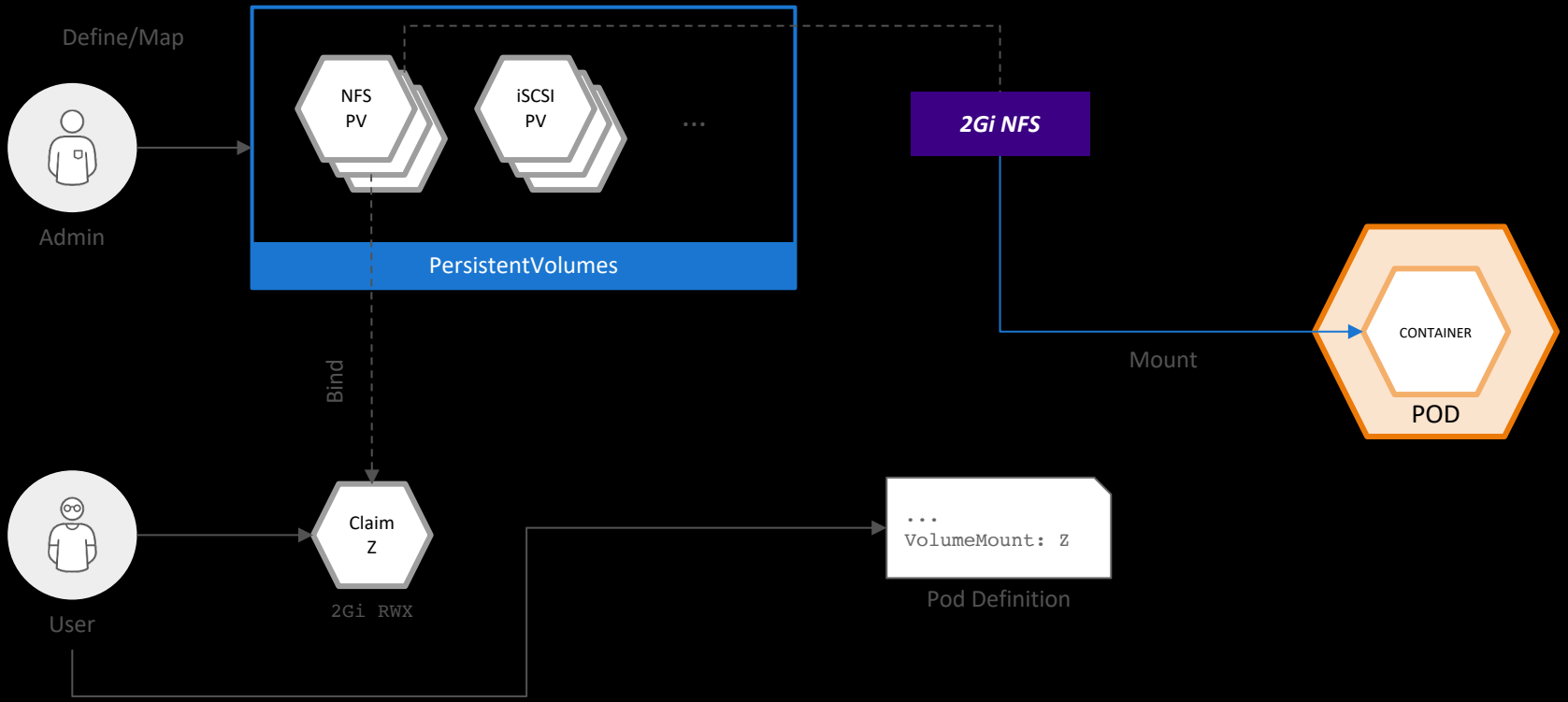
Persistent Volume Claim (PVC) is a claim for that storage by a user

Storage Classes (SC) allow allocating storage technologies and dynamic provisioning

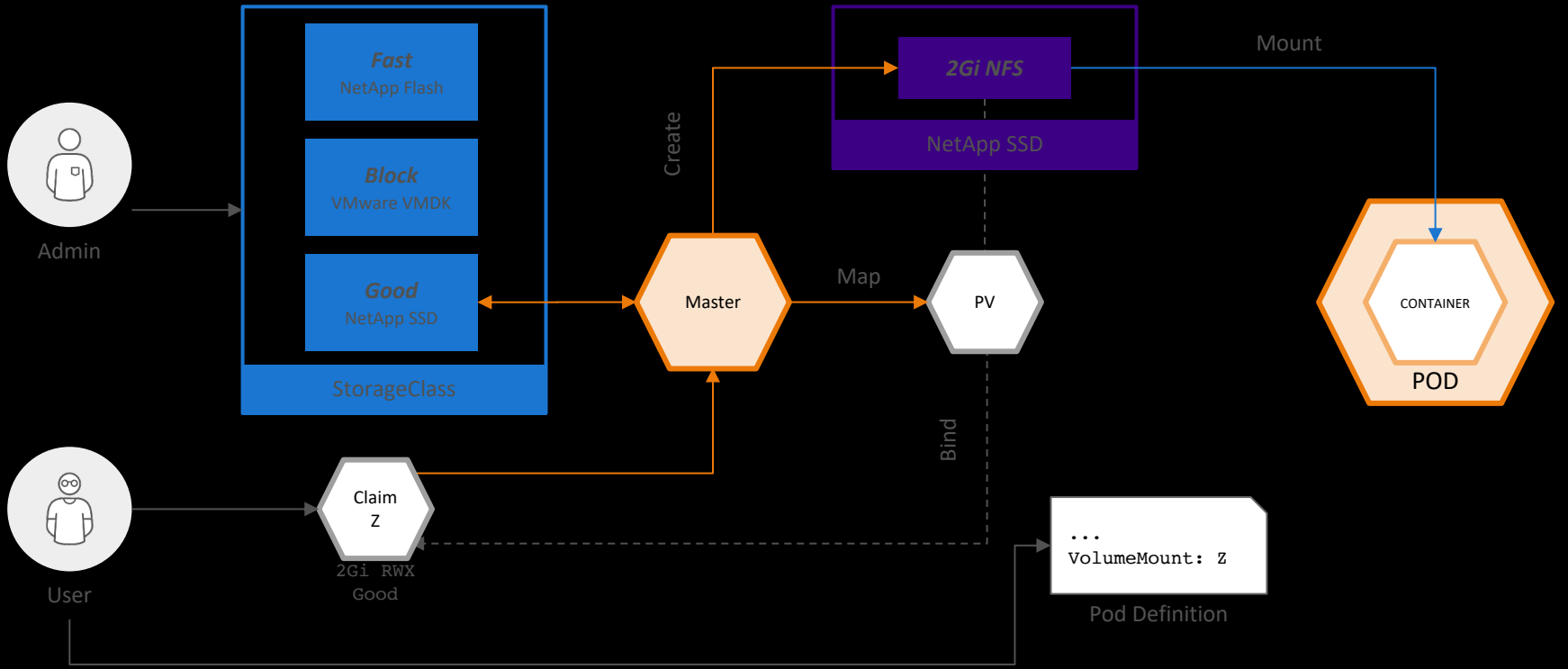
OpenShift Storage
- PV Consumption



OpenShift Storage
- Static Volume Provisioning



OpenShift Storage
- Dynamic Provisioning



OpenShift Storage
- Supported Providers

Volume Plug-in	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWS EBS	✅	-	-
Azure File	✅	✅	✅
Azure Disk	✅	-	-
Cinder	✅	-	-
Fibre Channel	✅	✅	-
GCE Persistent Disk	✅	-	-
HostPath	✅	-	-
iSCSI	✅	✅	-
Local volume	✅	-	-
NFS	✅	✅	✅
Red Hat OpenShift Container Storage	✅	-	✅
VMware vSphere	✅	-	-

Questions or Discussions?