Introduction to **Kubernetes**

VCUBE
DEVOPS

Container orchestration

- Container orchestration automates the deployment, management, scaling, and networking of containers across the cluster. It is focused on managing the life cycle of containers.
- Enterprises that need to deploy and manage hundreds or thousands of Linux® containers and hosts can benefit from container orchestration.
- Container orchestration is used to automate the following tasks at scale:
  - ✓ Configuring and scheduling of containers
  - ✓ Provisioning and deployment of containers
  - ✓ Redundancy and availability of containers
  - ✓ Scaling up or removing containers to spread application load evenly across host infrastructure
  - ✓ Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
  - ✓ Allocation of resources between containers
  - ✓ External exposure of services running in a container with the outside world
  - ✓ Load balancing of service discovery between containers
  - ✓ Health monitoring of containers and hosts

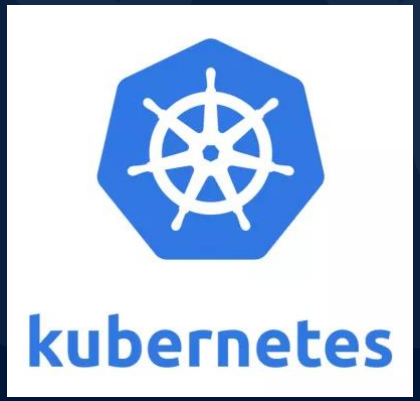
Swarm vs Kubernetes

Both Kubernetes and Docker Swarm are important tools that are used to deploy containers inside a cluster but there are subtle differences between the both

Features	Kubernetes	Docker Swarm
Installation & Cluster Configuration	Installation is complicated; but once setup, the cluster is very strong	Installation is very simple; but cluster is not very strong
GUI	GUI is the Kubernetes Dashboard	There is no GUI
Scalability	Highly scalable & scales fast	Highly scalable & scales 5x faster than Kubernetes
Auto-Scaling	Kubernetes can do auto-scaling	Docker Swarm cannot do auto-scaling
Rolling Updates & Rollbacks	Can deploy Rolling updates & does automatic Rollbacks	Can deploy Rolling updates, but not automatic Rollbacks
Data Volumes	Can share storage volumes only with other containers in same Pod	Can share storage volumes with any other container
Logging & Monitoring	In-built tools for logging & monitoring	3rd party tools like ELK should be used for logging & monitoring

Kubernetes

- Kubernetes also known as K8s, is an open-source Container Management tool
- It provides a container runtime, container orchestration, container-centric infrastructure orchestration, self-healing mechanisms, service discovery, load balancing and container (de)scaling.
- Initially developed by Google, for managing containerized applications in a clustered environment but later donated to CNCF
- Written in Golang
- It is a platform designed to completely manage the life cycle of containerized applications and services using methods that provide predictability, scalability, and high availability.



Kubernetes

Certified Kubernetes Distributions

- Cloud Managed: EKS by AWS, AKS by Microsoft and GKE by google
- Self Managed: OpenShift by Redhat and Docker Enterprise
- Local dev/test: Micro K8s by Canonical, Minikube
- Vanilla Kubernetes: The core Kubernetes project(baremetal), Kubeadm
- Special builds: K3s by Rancher, a light weight K8s distribution for Edge devices

Online Emulator: https://labs.play-with-k8s.com/

https://www.cncf.io/certification/software-conformance/

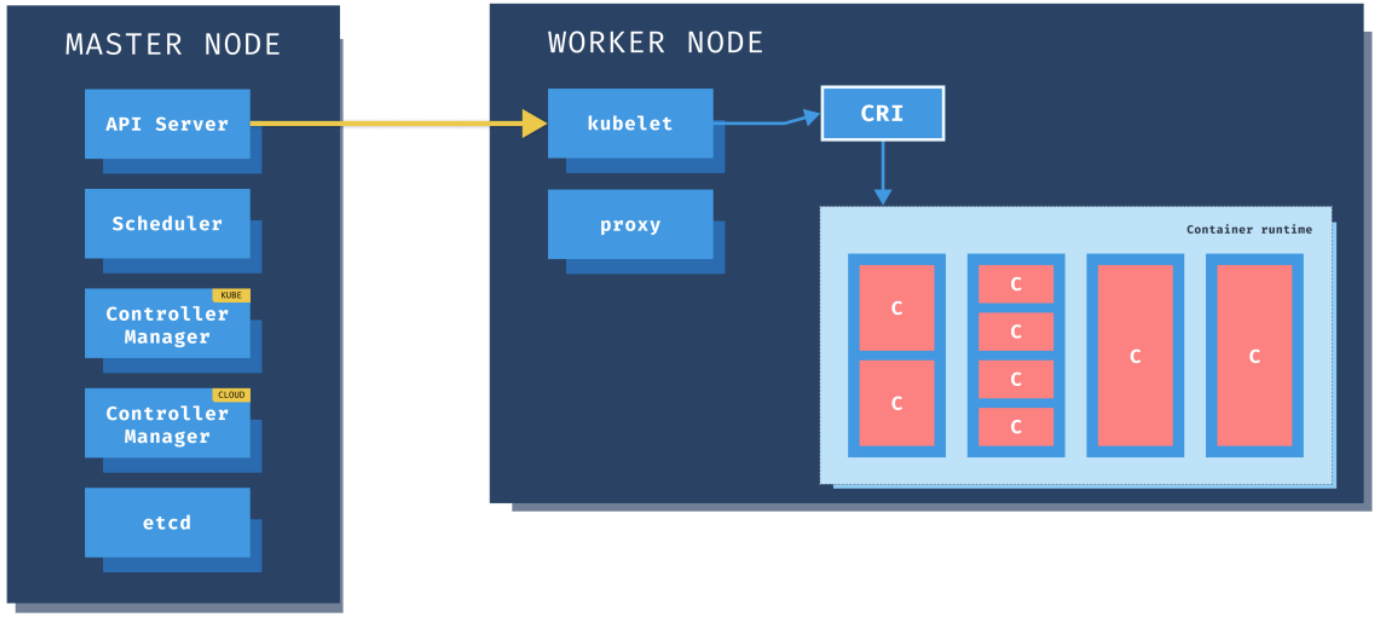
Kubernetes Cluster

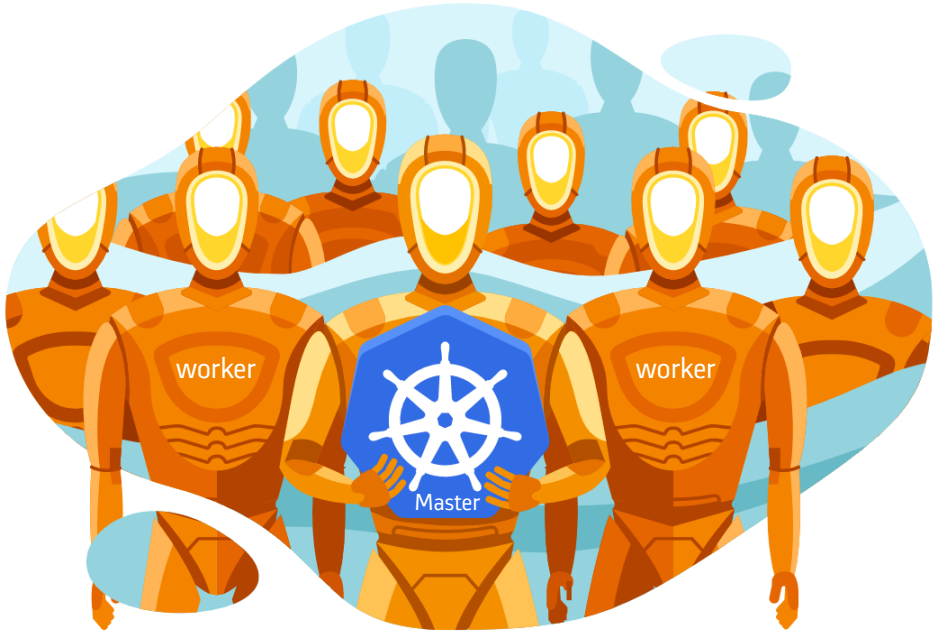
A Kubernetes cluster is a set of physical or virtual machines and other infrastructure resources that are needed to run your containerized applications. Each machine in a Kubernetes cluster is called a **node**.

There are two types of node in each Kubernetes cluster:

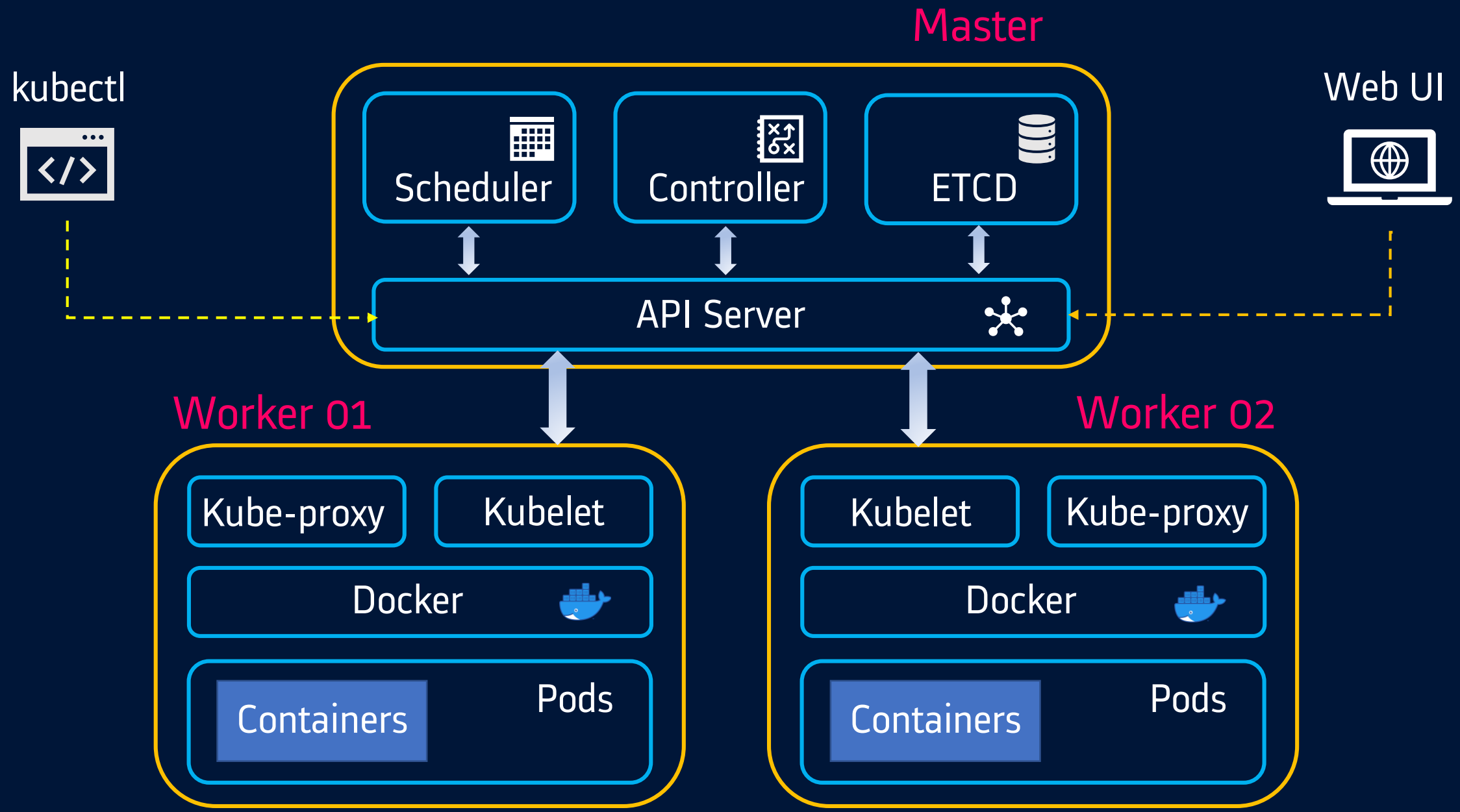
Master node(s): hosts the Kubernetes control plane components and manages the cluster

Worker node(s): runs your containerized applications

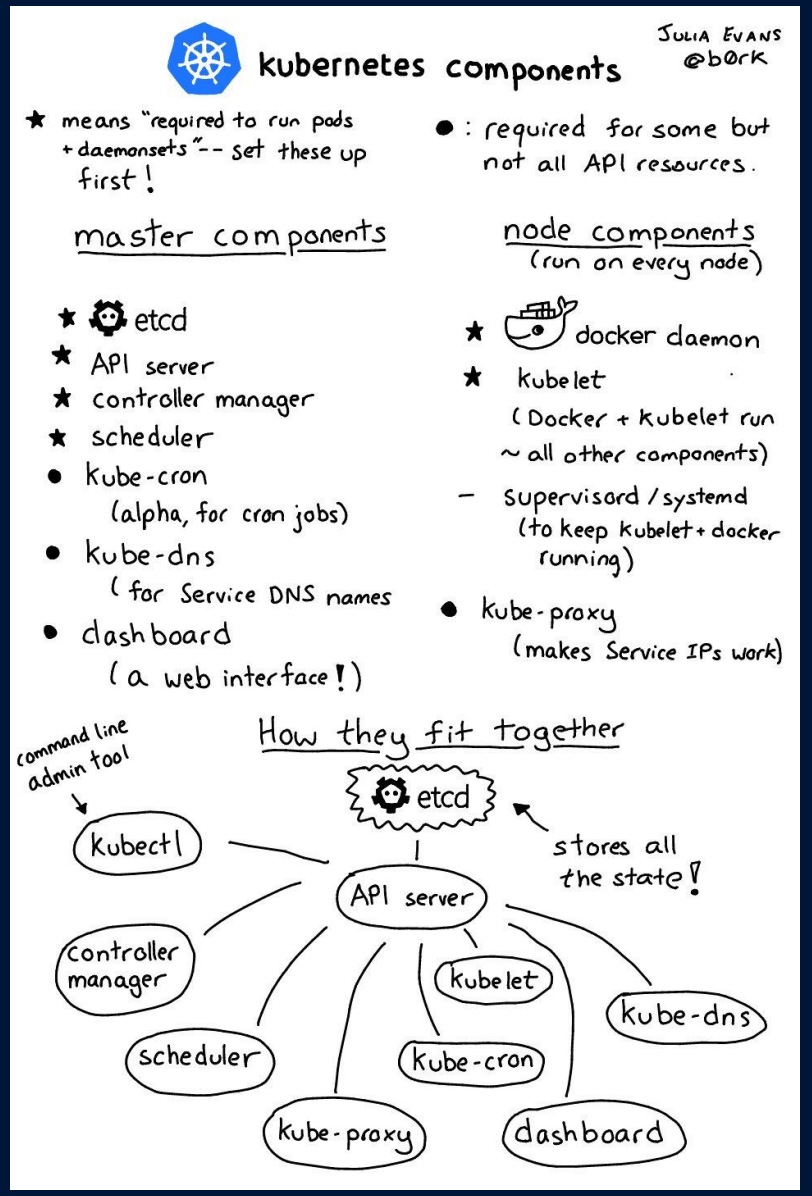




Kubernetes Architecture



Kubernetes Architecture

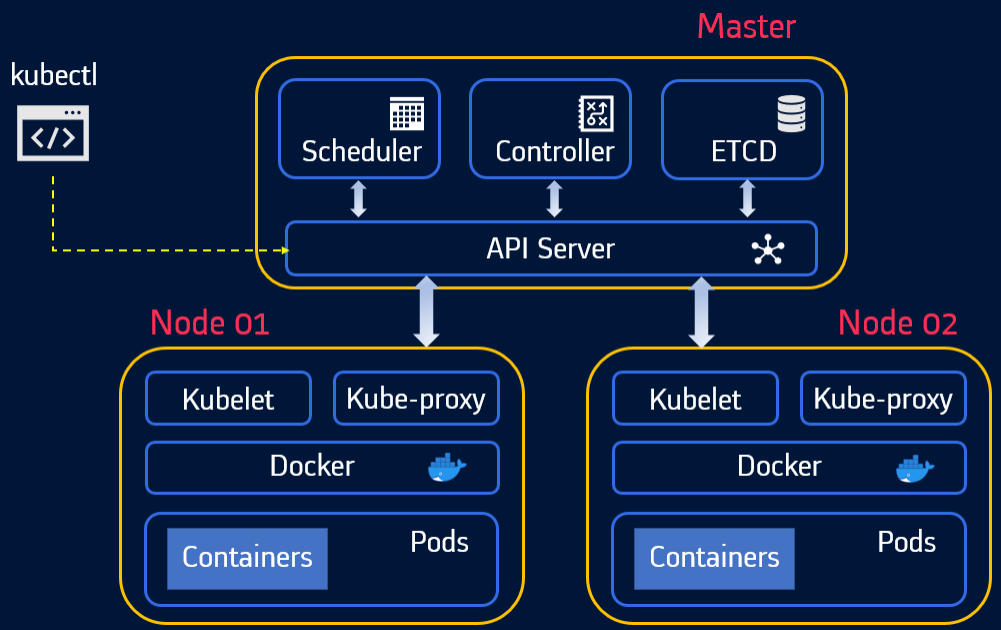


https://blog.alexellis.io/kubernetes-in-10-minutes/

Kubernetes Architecture

Kubernetes Master

- Master is responsible for managing the complete cluster.
- You can access master node via the CLI, GUI, or API
- The master watches over the nodes in the cluster and is responsible for the actual orchestration of containers on the worker nodes
- For achieving fault tolerance, there can be more than one master node in the cluster.
- It is the access point from which administrators and other users interact with the cluster to manage the scheduling and deployment of containers.
- It has four components: ETCD, Scheduler, Controller and API Server

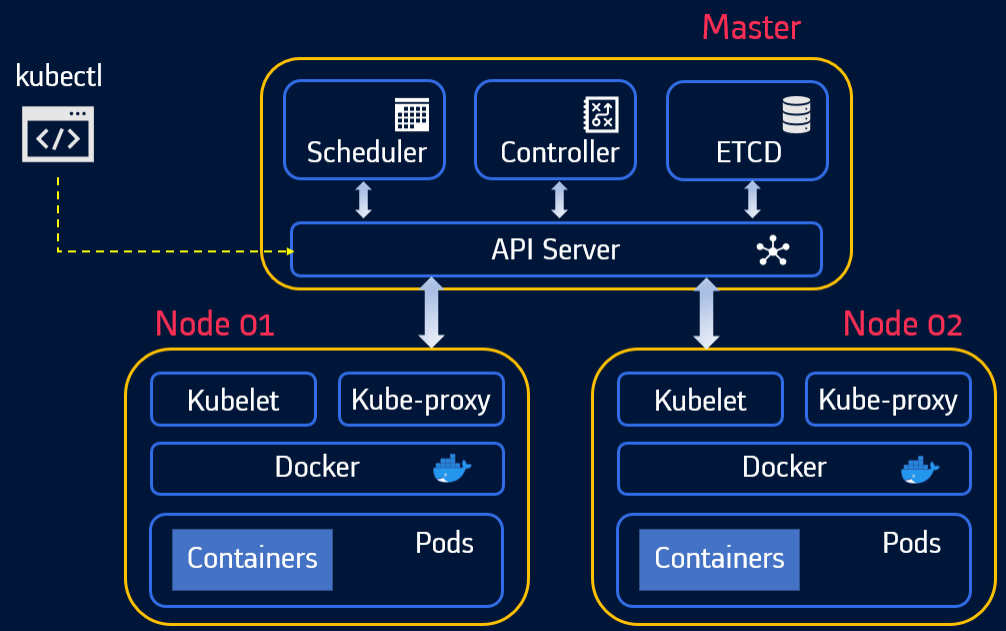


Kubernetes Architecture

Kubernetes Master

ETCD

- ETCD is a distributed reliable key-value store used by Kubernetes to store all data used to manage the cluster.
- When you have multiple nodes and multiple masters in your cluster, etcd stores all that information on all the nodes in the cluster in a distributed manner.
- ETCD is responsible for implementing locks within the cluster to ensure there are no conflicts between the Masters



Scheduler

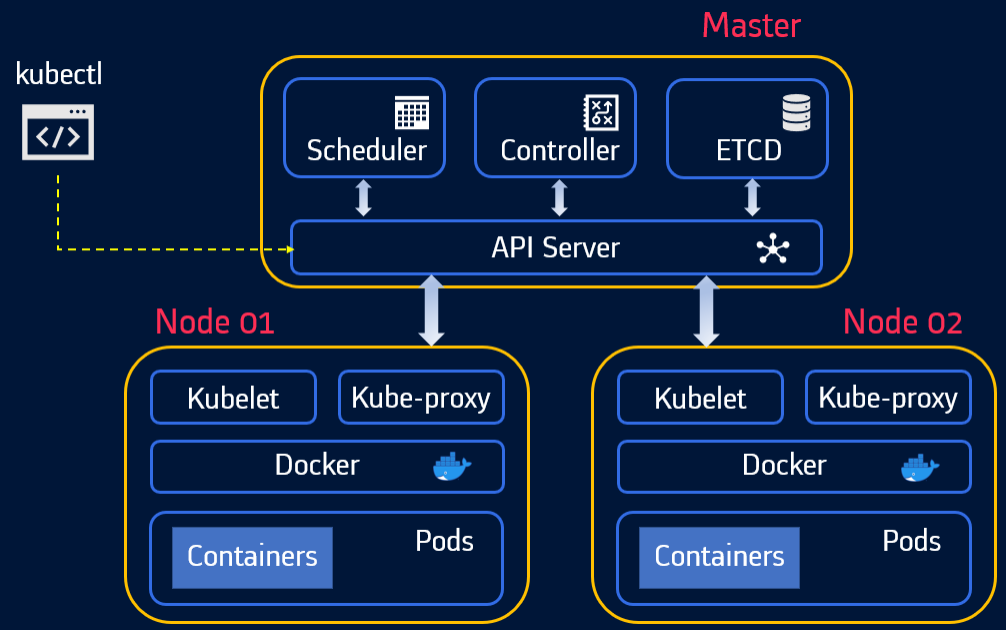
- The scheduler is responsible for distributing work or containers across multiple nodes.
- It looks for newly created containers and assigns them to Nodes.

Kubernetes Architecture

Kubernetes Master

API server manager

- Masters communicate with the rest of the cluster through the kube-apiserver, the main access point to the control plane.
- It validates and executes user's REST commands
- kube-apiserver also makes sure that configurations in etcd match with configurations of containers deployed in the cluster.



Controller manager

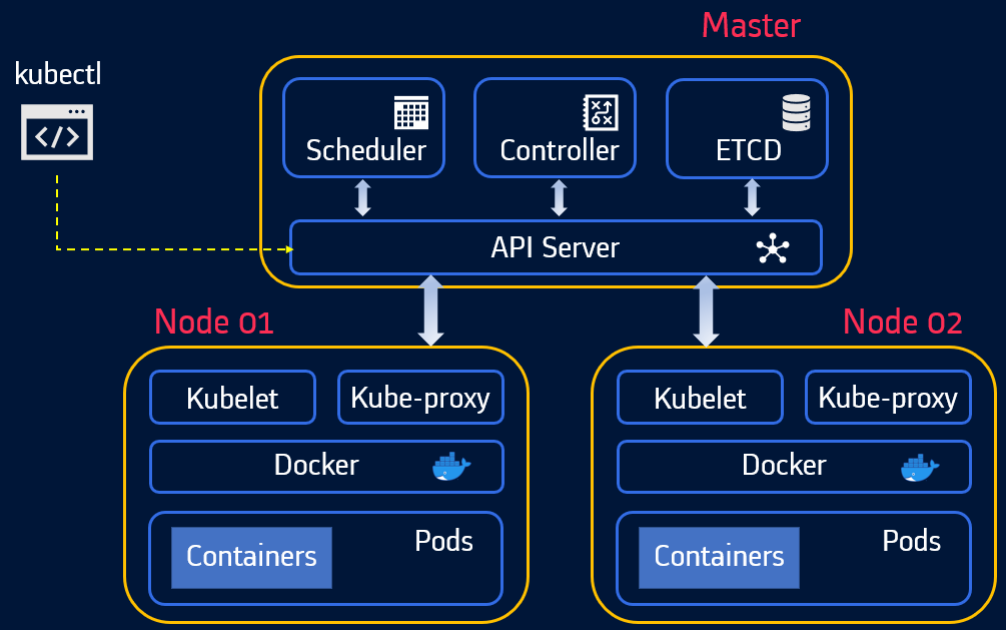
- The controllers are the brain behind orchestration.
- They are responsible for noticing and responding when nodes, containers or endpoints goes down. The controllers makes decisions to bring up new containers in such cases.
- The kube-controller-manager runs control loops that manage the state of the cluster by checking if the required deployments, replicas, and nodes are running in the cluster

Kubernetes Architecture

Kubernetes Master

Kubectl

- kubectl is the command line utility using which we can interact with k8s cluster
- Uses APIs provided by API server to interact.
- Also known as the kube command line tool or kubectl or kube control.
- Used to deploy and manage applications on a Kubernetes



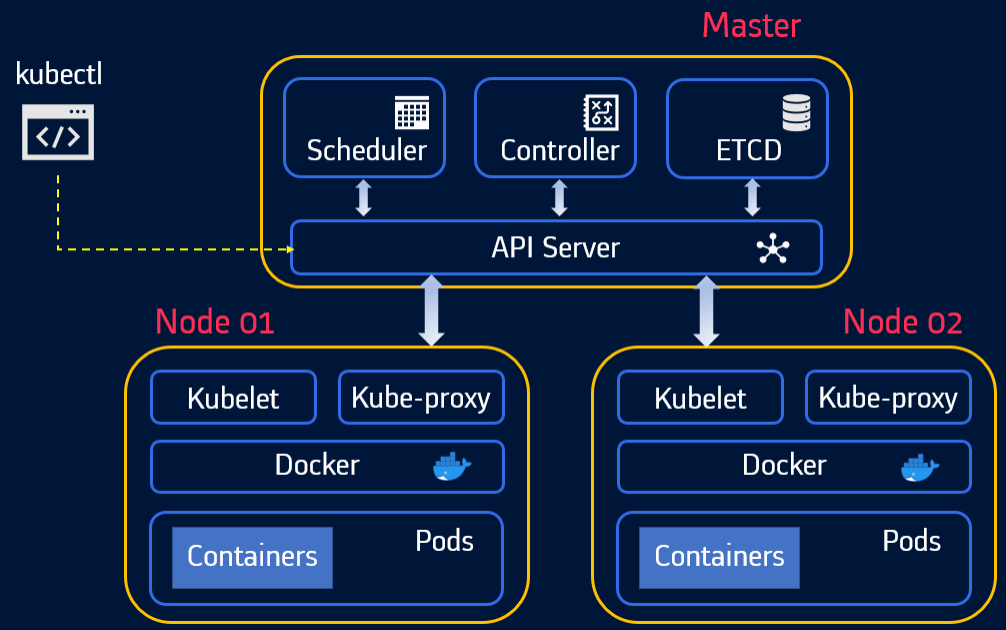
- kubectl run nginx used to deploy an application on the cluster.
- kubectl cluster-info used to view information about the cluster and the
- kubectl get nodes used to list all the nodes part of the cluster.

Kubernetes Architecture

Kubernetes Worker

Kubelet

- Worker nodes have the kubelet agent that is responsible for interacting with the master to provide health information of the worker node
- To carry out actions requested by the master on the worker nodes.



Kube proxy

- The kube-proxy is responsible for ensuring network traffic is routed properly to internal and external services as required and is based on the rules defined by network policies in kube-controller-manager and other custom controllers.

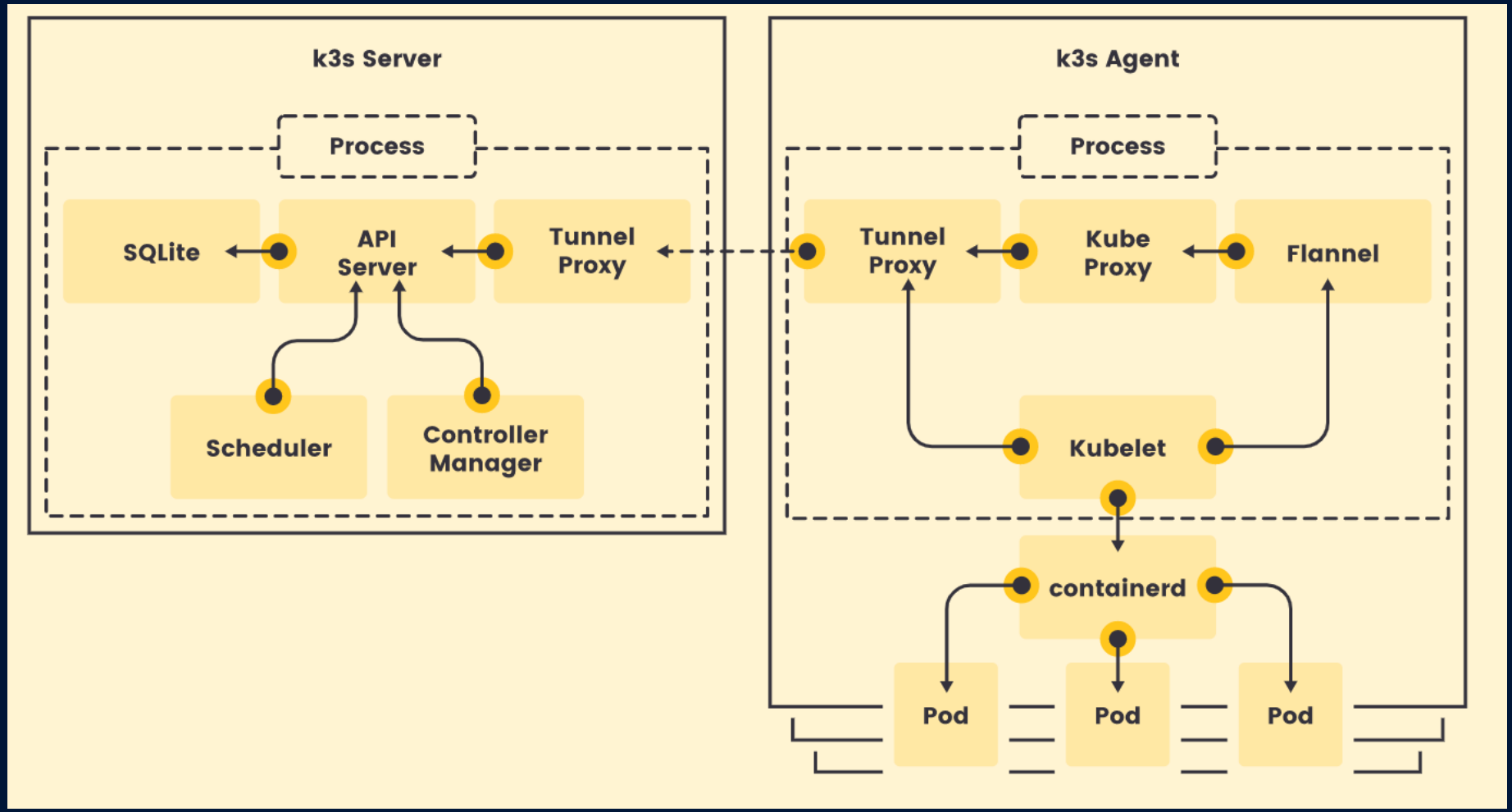
Kubernetes

What is K3s?

- K3s is a fully compliant Kubernetes distribution with the following enhancements:
  - ✓ Packaged as a single binary
  - ✓ <100MB memory footprint
  - ✓ Supports ARM and x86 architectures
  - ✓ Lightweight storage backend based on sqlite3 as the default storage mechanism to replace heavier ETCD server
  - ✓ Docker is replaced in favour of containerd runtime
  - ✓ Inbuilt Ingress controller (Traefik)

Kubernetes

K3s Architecture



Kubernetes

K3s Setup using VirtualBox

- Use 3VMs(1 master and 2 workers). All VMs should have bridge network adapter enabled
- Create a host only networking adapter(DHCP disabled) and connect all VMs to it. This is to have static IPs for all VMs in the cluster. Make sure static IPs are configured in each VM in the same subnet range of host only network

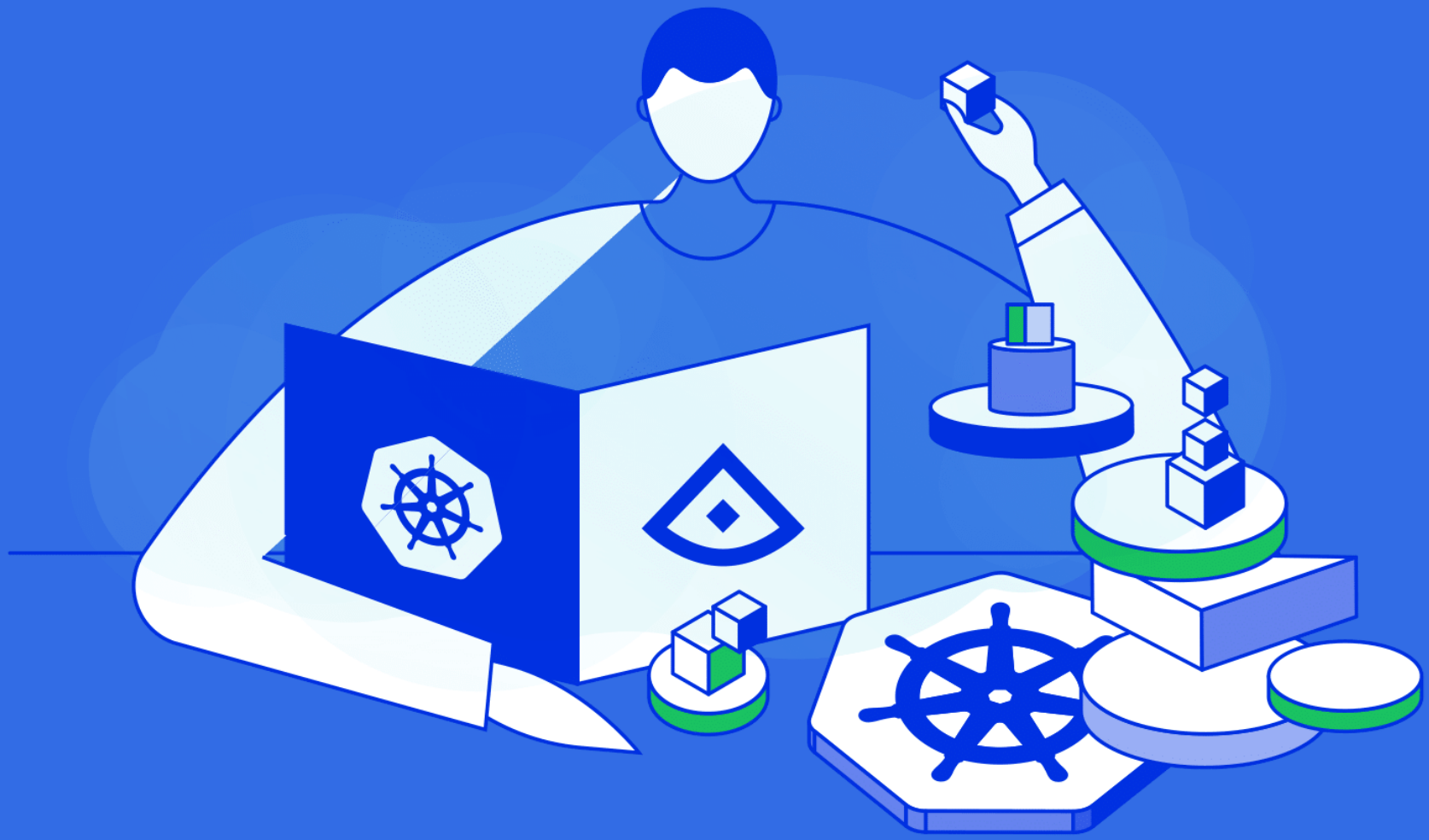
On Master

- bash -c "curl -sfL https://get.k3s.io | sh -“
- TOKEN=cat /var/lib/rancher/k3s/server/node-token
- IP = IP of master node where API server is running

On Worker nodes

- bash -c "curl -sfL https://get.k3s.io | K3S_URL=\"https://$IP:6443\" K3S_TOKEN=\"$TOKEN\" sh -"

https://medium.com/better-programming/local-k3s-cluster-made-easy-with-multipass-108bf6ce577c



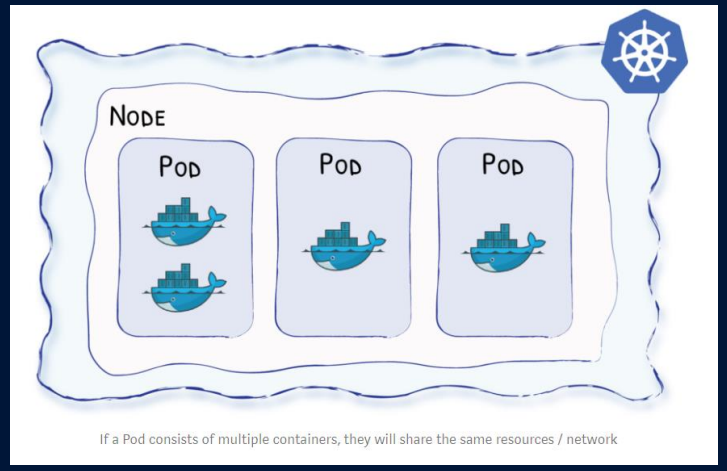
Kubernetes Pods

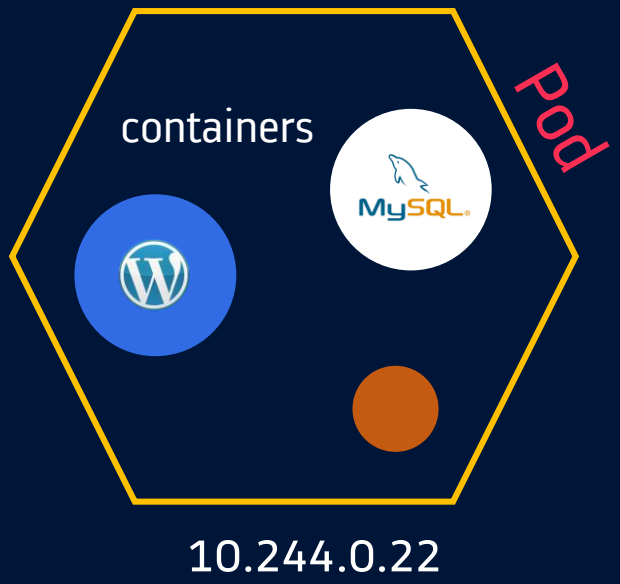
Kubernetes

Pods

- Basic scheduling unit in Kubernetes. Pods are often ephemeral
- Kubernetes doesn't run containers directly; instead it wraps one or more containers into a higher-level structure called a pod
- It is also the smallest deployable unit that can be created, schedule, and managed on a Kubernetes cluster. Each pod is assigned a unique IP address within the cluster.
- Pods can hold multiple containers as well, but you should limit yourself when possible. Because pods are scaled up and down as a unit, all containers in a pod must scale together, regardless of their individual needs. This leads to wasted resources.

Ex: nginx, mysql, wordpress..

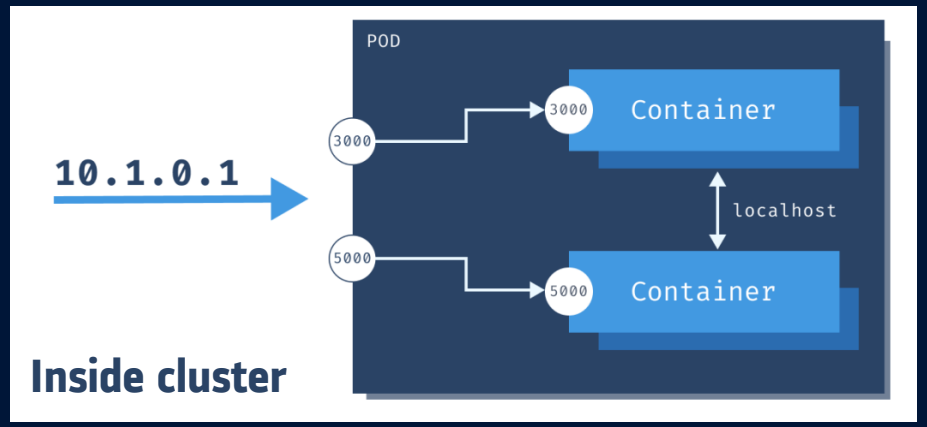




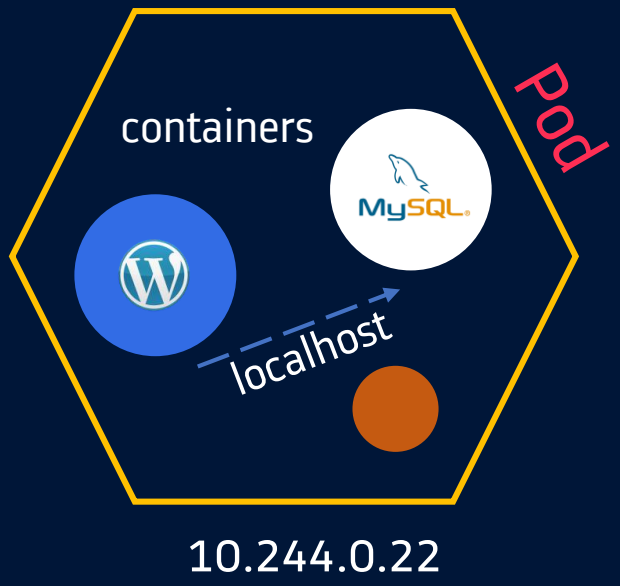
Kubernetes

Pods

- Any containers in the same pod will share the same storage volumes and network resources and communicate using localhost
- K8s uses YAML to describe the desired state of the containers in a pod. This is also called a Pod Spec. These objects are passed to the kubelet through the API server.
- Pods are used as the unit of replication in Kubernetes. If your application becomes too popular and a single pod instance can't carry the load, Kubernetes can be configured to deploy new replicas of your pod to the cluster as necessary.



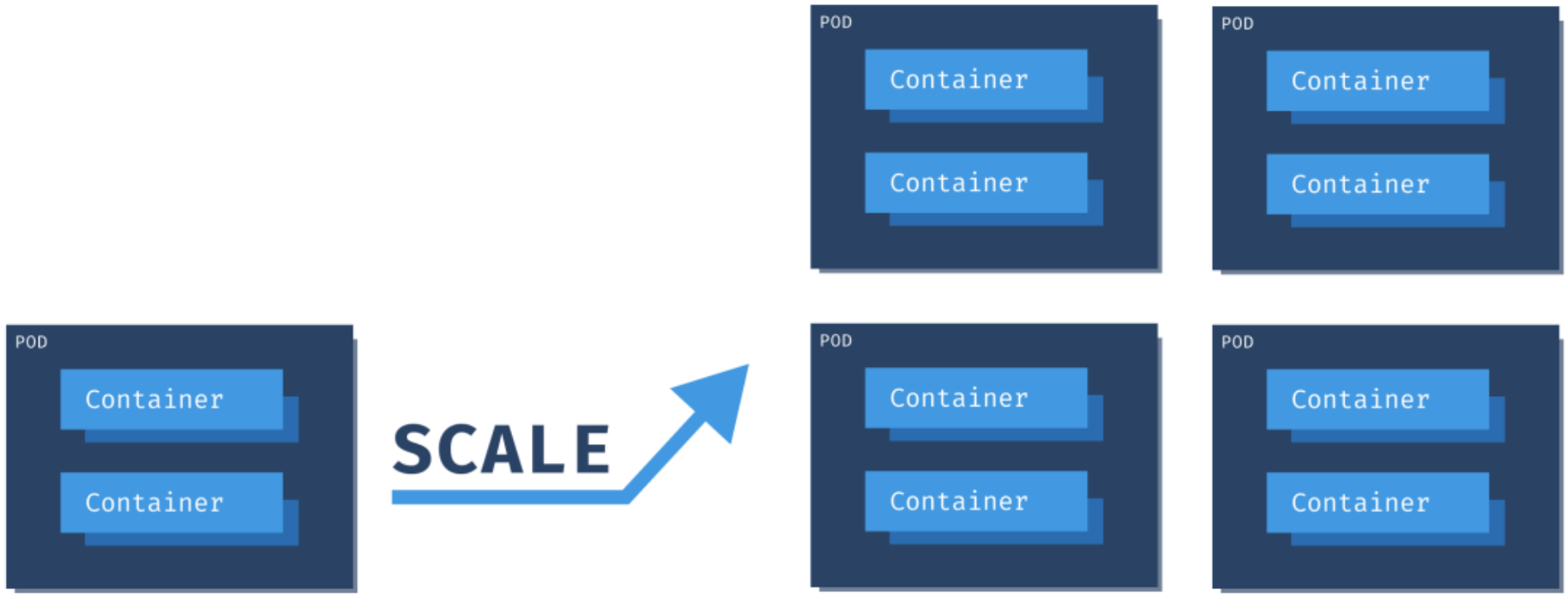
Using the example from the above figure, you could run curl 10.1.0.1:3000 to communicate to the one container and curl 10.1.0.1:5000 to communicate to the other container from other pods. However, if you wanted to talk between containers - for example, calling the top container from the bottom one, you could use http://localhost:3000.



Kubernetes

Scaling Pods

- All containers within the pod get scaled together.
- You cannot scale individual containers within the pods. The pod is the unit of scale in K8s.
- Recommended way is to have only one container per pod. Multi container pods are very rare.
- In K8s, **initcontainer** is sometimes used as a second container inside pod.



initcontainers are exactly like regular containers, except that they always run to completion. Each init container must complete successfully before the next one starts.
If a Pod's init container fails, Kubernetes repeatedly restarts the Pod until the init container succeeds

Kubernetes

Imperative vs Declarative commands

- Kubernetes API defines a lot of objects/resources, such as namespaces, pods, deployments, services, secrets, config maps etc.
- There are two basic ways to deploy objects in Kubernetes: Imperatively and Declaratively

Imperatively

- Involves using any of the verb-based commands like kubectl run, kubectl create, kubectl expose, kubectl delete, kubectl scale and kubectl edit
- Suitable for testing and interactive experimentation

Declaratively

- Objects are written in YAML files and deployed using kubectl create or kubectl apply
- Best suited for production environments

Kubernetes

Manifest /Spec file

- K8s object configuration files - Written in YAML or JSON
- They describe the desired state of your application in terms of Kubernetes API objects. A file can include one or more API object descriptions (manifests).

manifest file template

apiVersion - version of the Kubernetes API used to create the object

kind - kind of object being created

metadata - Data that helps uniquely identify the object, including a name and optional namespace

spec - configuration that defines the desired for the object

```
apiVersion: v1
kind: Pod
metadata:
  name: ...
spec:
  containers:
  - name: ...
---
apiVersion: v1
kind: Pod
metadata:
  name: ...
spec:
  containers:
  - name: ...
```

Multiple resource definitions

Kubernetes

Manifest files Man Pages

List all K8s API supported Objects and Versions

kubectl api-resources
kubectl api-versions

Man pages for objects

kubectl explain <object>.<option>
kubectl explain pod
kubectl explain pod.apiVersion
kubectl explain pod.spec

```
apiVersion: v1
kind: Pod
metadata:
  name: ...
spec:
  containers:
  - name: ...
---
apiVersion: v1
kind: Pod
metadata:
  name: ...
spec:
  containers:
  - name: ...
```

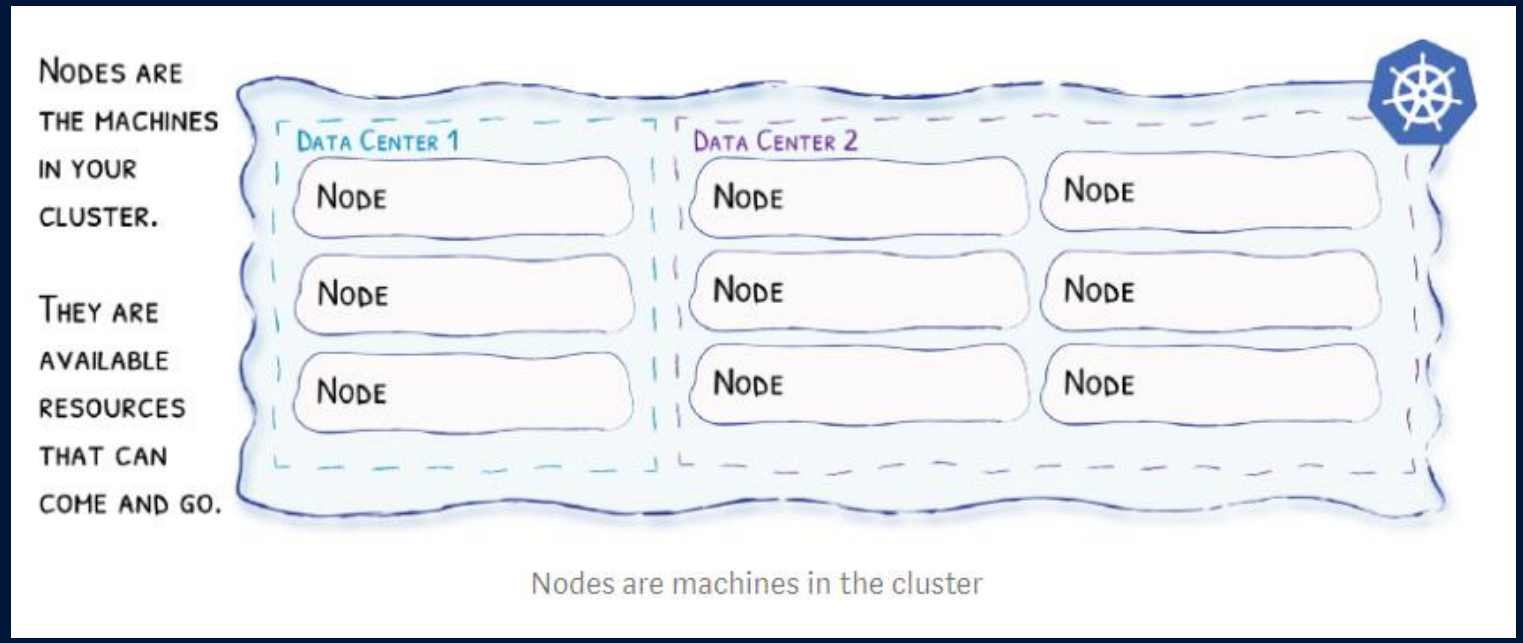
Multiple resource definitions

Kubernetes

Once the cluster is setup...

kubectl version

```
root@k3s-master:/home/osboxes# kubectl version -o yaml
clientVersion:
  buildDate: "2020-05-07T00:18:01Z"
  compiler: gc
  gitCommit: 698e444a03fe3a705849fc8d9ad9e35f1dffe286
  gitTreeState: clean
  gitVersion: v1.18.2+k3s1
  goVersion: go1.13.8
  major: "1"
  minor: "18"
  platform: linux/amd64
serverVersion:
  buildDate: "2020-05-07T00:18:01Z"
  compiler: gc
  gitCommit: 698e444a03fe3a705849fc8d9ad9e35f1dffe286
  gitTreeState: clean
  gitVersion: v1.18.2+k3s1
  goVersion: go1.13.8
  major: "1"
  minor: "18"
  platform: linux/amd64
```



Nodes are machines in the cluster

kubectl get nodes –o wide

```
root@k3s-master:/home/osboxes# kubectl get nodes -o wide
NAME          STATUS   ROLES    AGE    VERSION        INTERNAL-IP     EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION     CONTAINER-RUNTIME
k3s-slave01   Ready    <none>   7d4h   v1.18.2+k3s1   192.168.0.113   <none>        Ubuntu 18.04.3 LTS   5.0.0-23-generic   containerd://1.3.3-k3s2
k3s-master    Ready    master   7d4h   v1.18.2+k3s1   192.168.0.102   <none>        Ubuntu 18.04.3 LTS   5.0.0-23-generic   containerd://1.3.3-k3s2
k3s-slave02   Ready    <none>   7d4h   v1.18.2+k3s1   192.168.0.104   <none>        Ubuntu 18.04.3 LTS   5.0.0-23-generic   containerd://1.3.3-k3s2
root@k3s-master:/home/osboxes#
```

Kubernetes

Once the cluster is setup...

kubectl cluster-info

```
root@k8s-master:/home/osboxes# kubectl cluster-info
Kubernetes master is running at https://192.168.50.2:6443
KubeDNS is running at https://192.168.50.2:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

kubectl cluster-info dump --output-directory=/path/to/cluster-state # Dump current cluster state to /path/to/cluster-state

Kubernetes

Creating Pods

kubectl run <pod-name> --image <image-name>

kubectl run nginx --image nginx --dry-run=client

dry-run doesn't run the command but will show what the changes the command would do to the cluster

```
root@k-master:/home/osboxes# kubectl run nginx --image nginx --dry-run=client
pod/nginx created (dry run)
```

kubectl run nginx --image nginx --dry-run=client –o yaml

shows the command output in YAML. Shortcut to create a declarative yaml from imperative commands

```
root@k-master:/home/osboxes# kubectl run nginx --image nginx --dry-run=client -o yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

https://kubernetes.io/docs/reference/kubectl/cheatsheet/

Kubernetes

Creating Pods: Imperative way

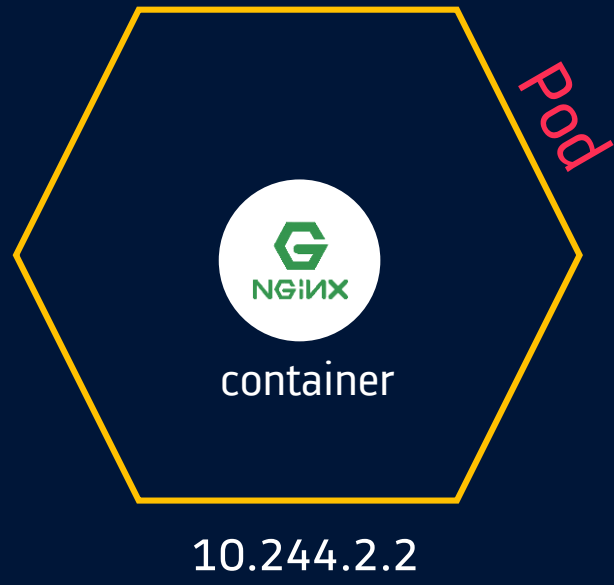
kubectl run test --image nginx --port 80 - Also exposes port 80 of container

kubectl get pods –o wide

```
root@k-master:/home/osboxes# kubectl get pods -o wide
NAME   READY   STATUS    RESTARTS   AGE     IP           NODE        NOMINATED NODE   READINESS GATES
test   1/1     Running   0          9m13s   10.244.2.2   k-slave02   <none>           <none>
root@k-master:/home/osboxes#
```

kubectl describe pod test – display extended information of pod

```
root@k-master:/home/osboxes# k describe pod test
Name:         test
Namespace:    default
Priority:     0
Node:         k-slave02/192.168.0.107
Start Time:   Mon, 18 May 2020 13:52:09 -0400
Labels:       run=test
Annotations:  <none>
Status:       Running
IP:           10.244.2.2
IPs:
  IP:  10.244.2.2
```

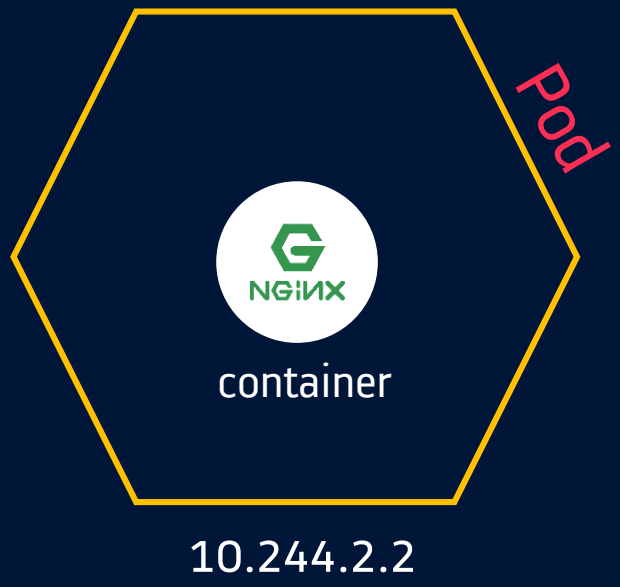


Kubernetes

Creating Pods

curl <ip-of-pod>

```
root@k-master:/home/osboxes# curl 10.244.2.2
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
```



Kubernetes

Creating Pods: Declarative way

- kubectl create –f pod-definition.yml
- kubectl apply –f pod-definition.yml – if manifest file is changed/updated after deployment and need to re-deploy the pod again
- kubectl delete pod <pod-name>

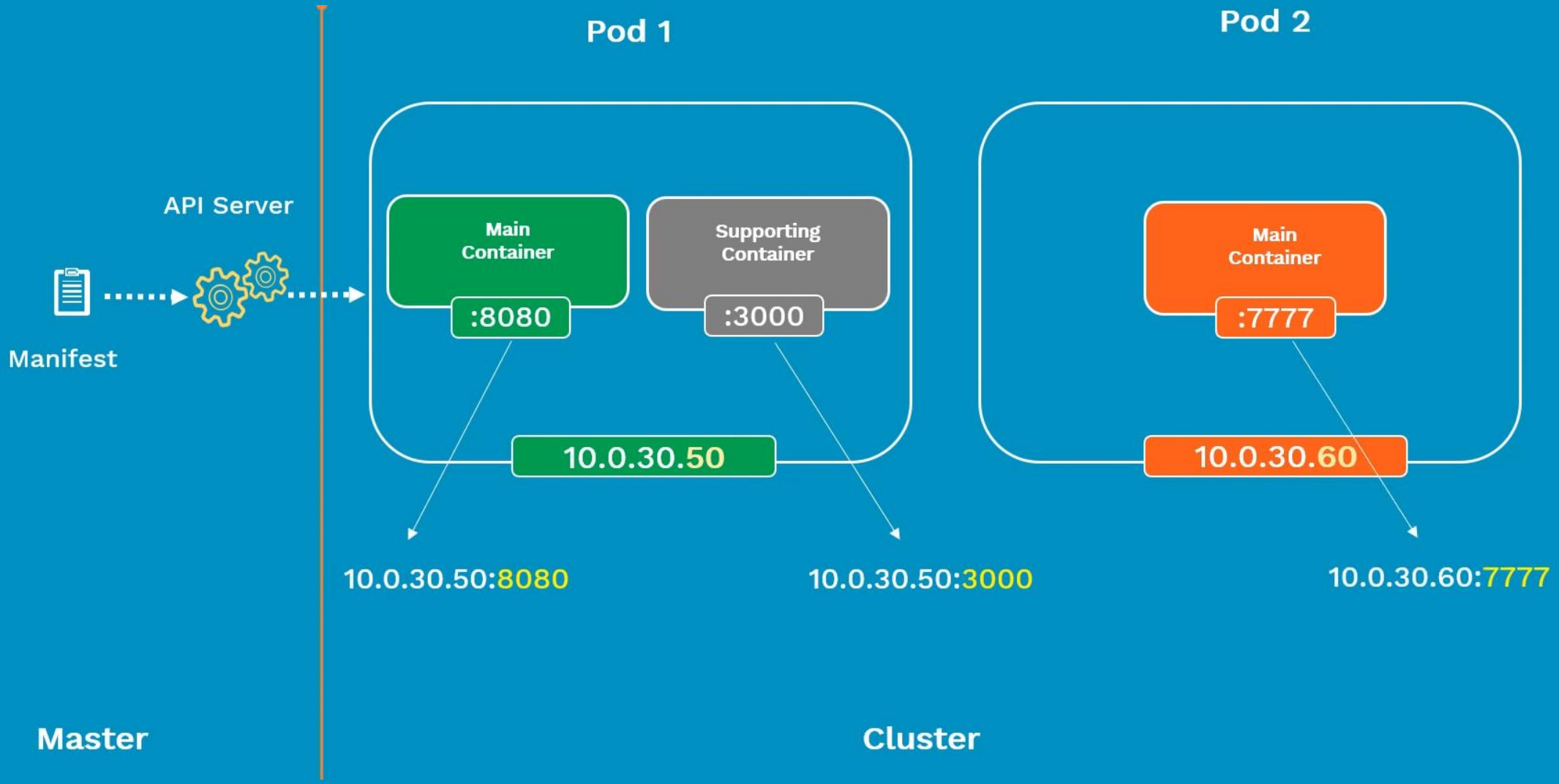
```yaml
# nginx-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
    tier: dev
spec:
  containers:
  - name: nginx-container
    image: nginx
```

Kind	apiVersion
Pod	v1
ReplicationController	V1
Service	v1
ReplicaSet	apps/v1
Deployment	apps/v1
DaemonSet	apps/v1
Job	batch/v1

```yaml
# pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: webapp
spec:
  containers:
  - name: nginx-container
    image: nginx
    ports:
      - containerPort: 80
```

Kubernetes

Pod Networking



Kubernetes

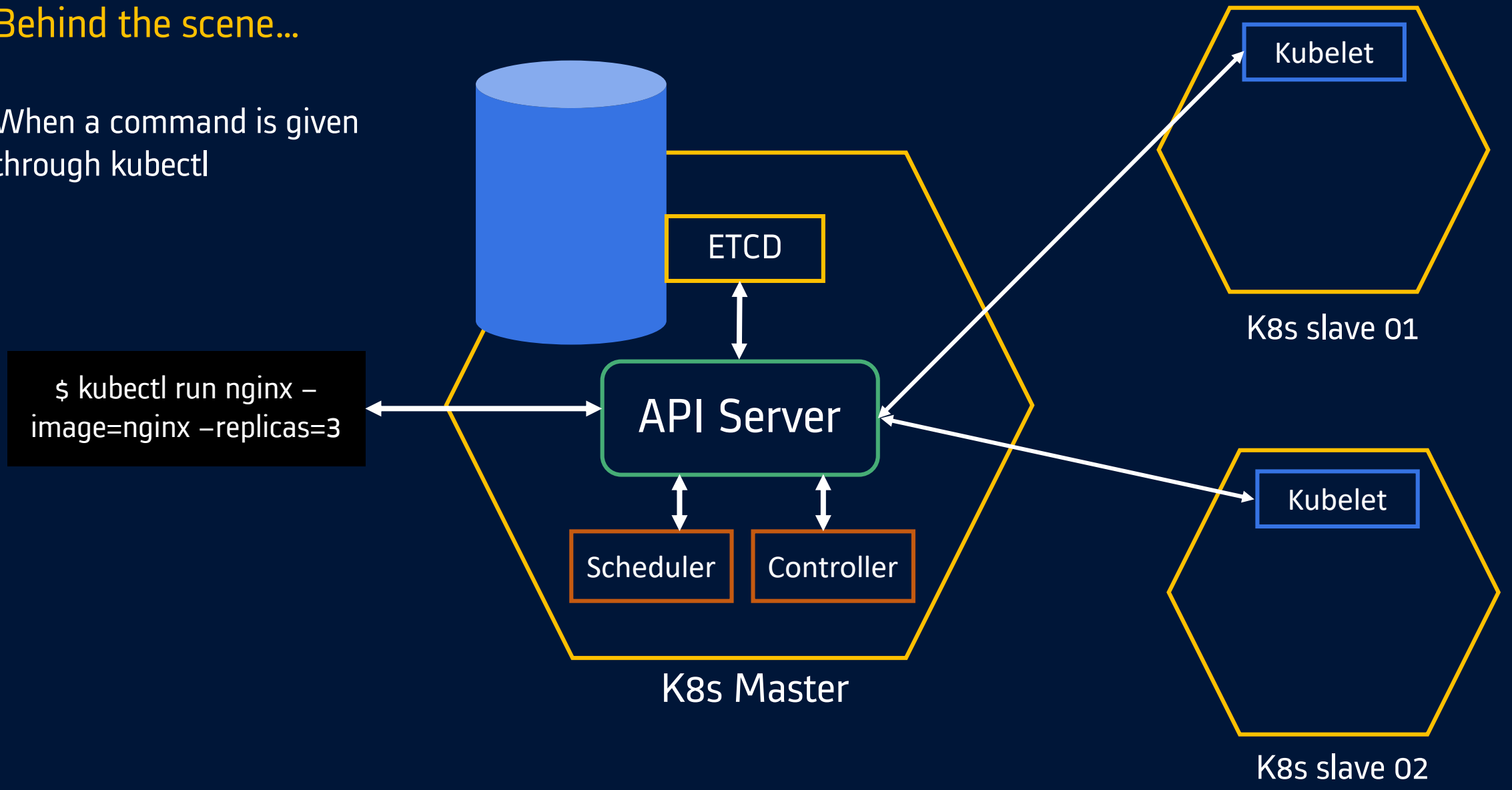
Replication Controller

- A single pod may not be sufficient to handle the user traffic. Also if this only pod goes down because of a failure, K8s will not bring this pod up again automatically
- In order to prevent this, we would like to have more than one instance or POD running at the same time inside the cluster
- Kubernetes supports different controllers(Replicacontroller & ReplicaSet) to handle multiple instances of a pod. Ex: 3 replicas of nginx webserver
- Replication Controller ensures high availability by replacing the unhealthy/dead pods with a new one to ensure required replicas are always running inside a cluster
- So, does that mean you can't use a replication controller if you plan to have a single POD? No! Even if you have a single POD, the replication controller can help by automatically bringing up a new POD when the existing one fails.
- Another reason we need replication controller is to create multiple PODs to share the load across them.
- Replica controller is deprecated and replaced by Replicaset

Kubernetes

Behind the scene...

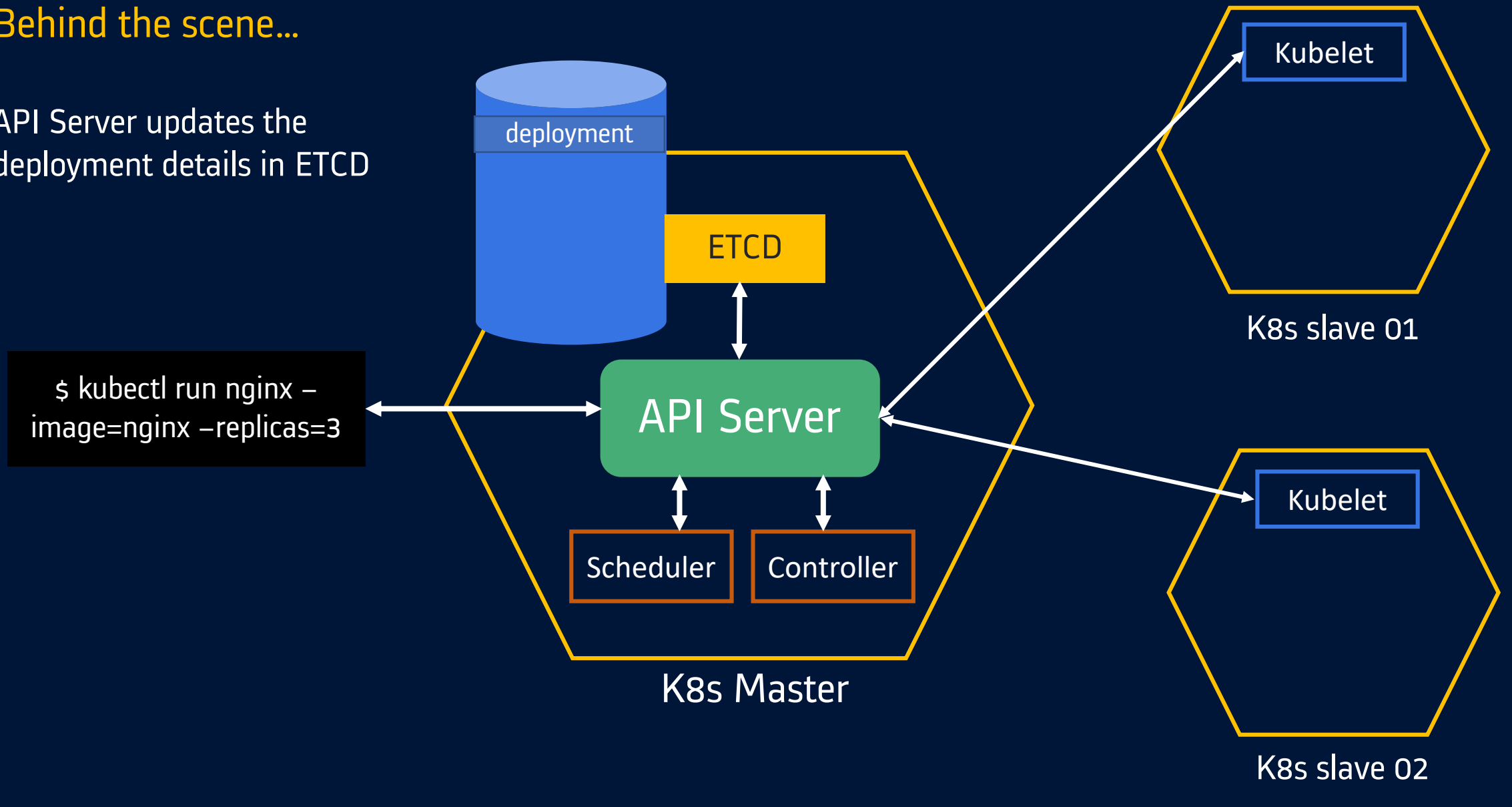
When a command is given through kubectl



Kubernetes

Behind the scene...

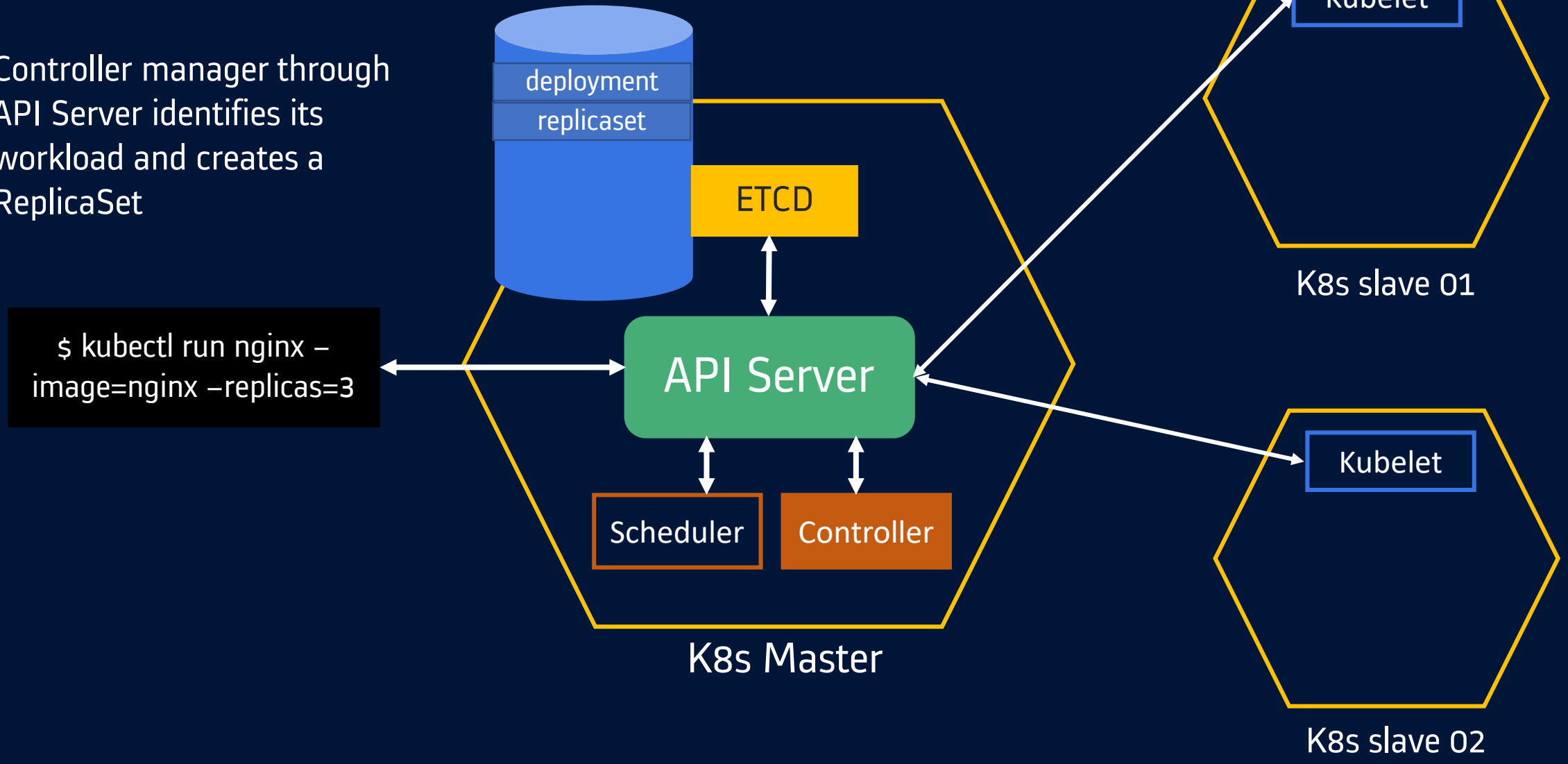
API Server updates the deployment details in ETCD



Kubernetes

Behind the scene...

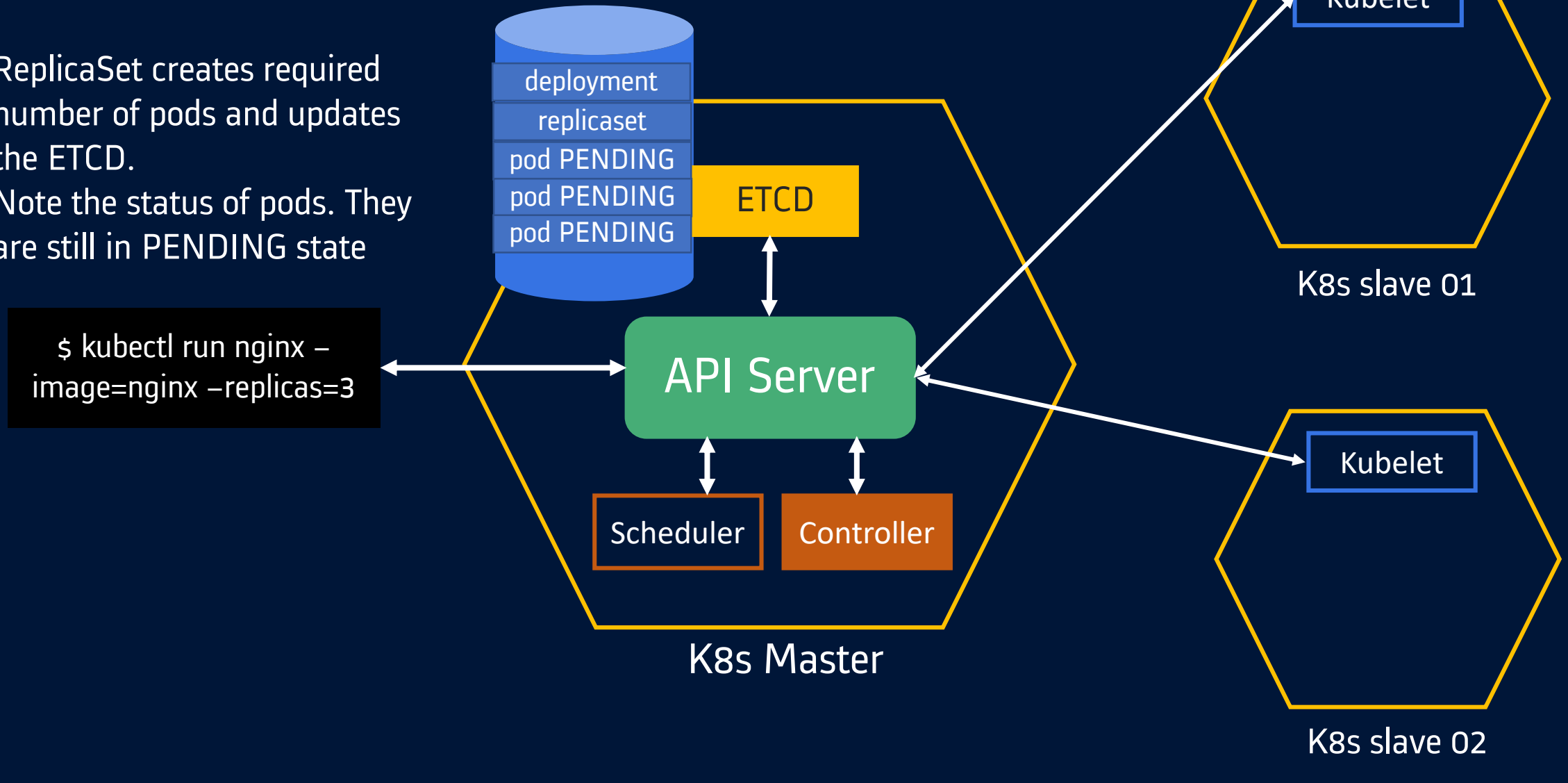
Controller manager through API Server identifies its workload and creates a ReplicaSet



Kubernetes

Behind the scene...

ReplicaSet creates required number of pods and updates the ETCD.
Note the status of pods. They are still in PENDING state

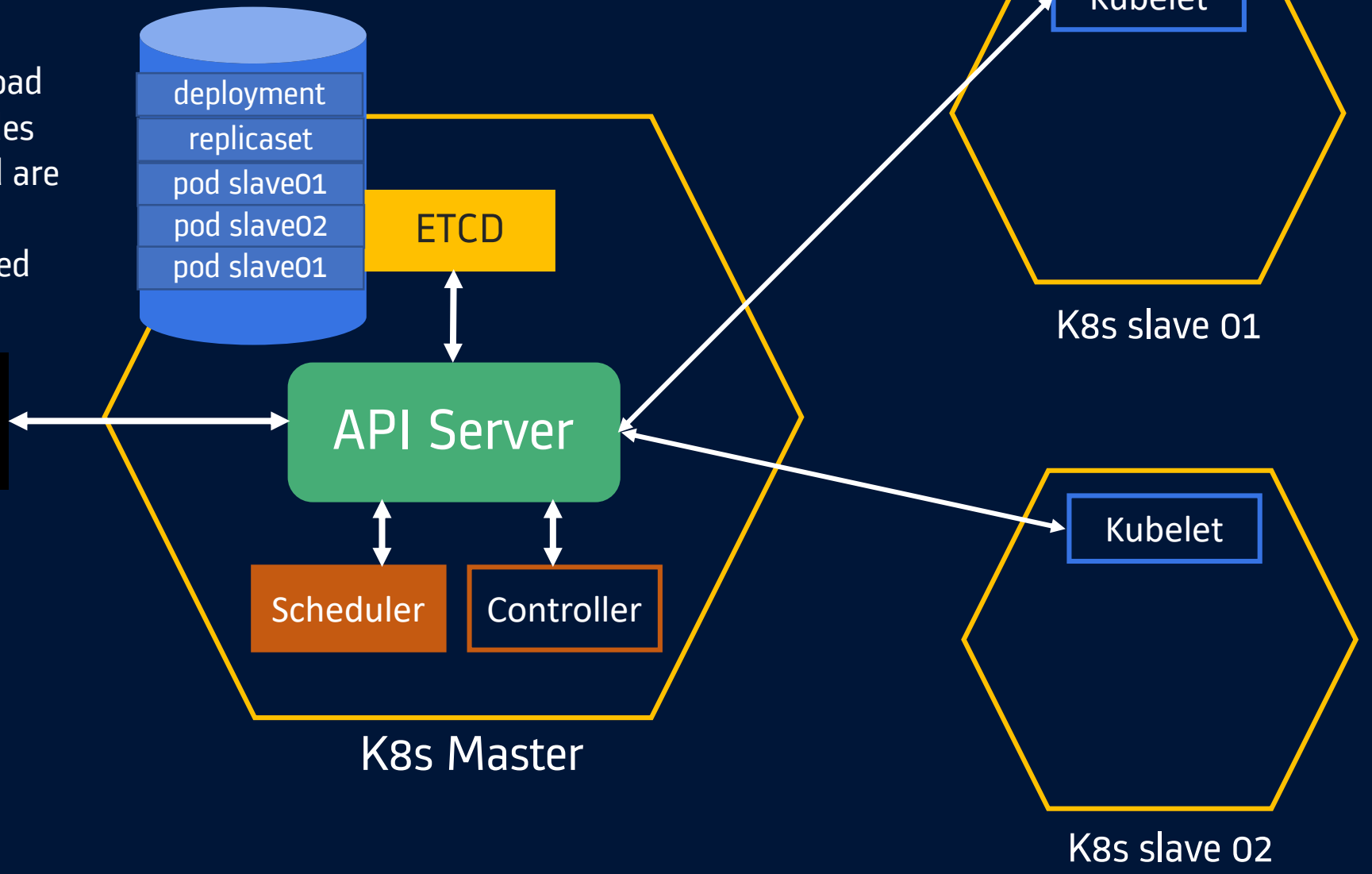


Kubernetes

Behind the scene...

Scheduler identifies its workload through API-Server and decides the nodes onto which the pod are to be scheduled.
At this stage, pods are assigned to a node

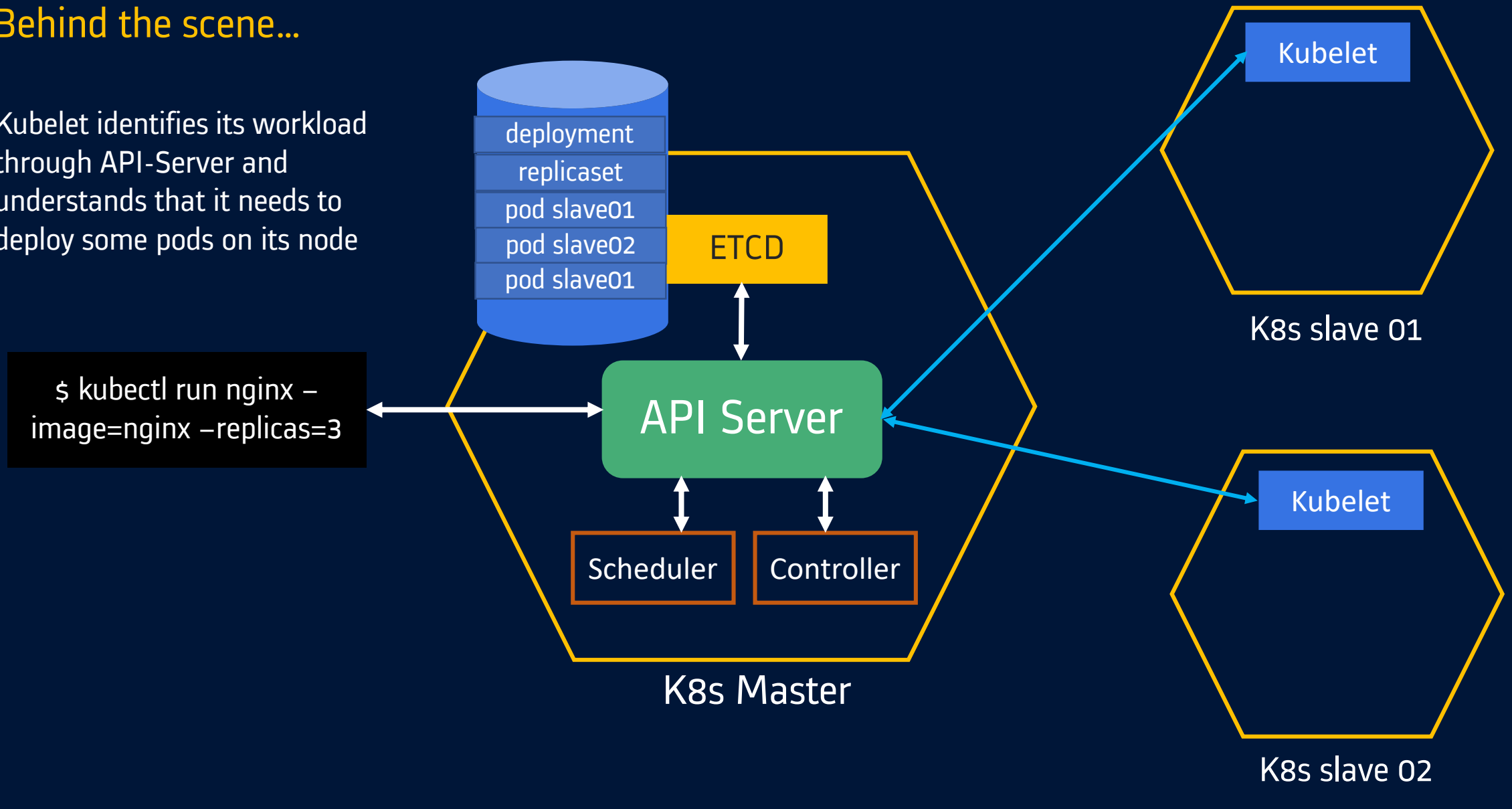
```
$ kubectl run nginx –
image=nginx –replicas=3
```



Kubernetes

Behind the scene...

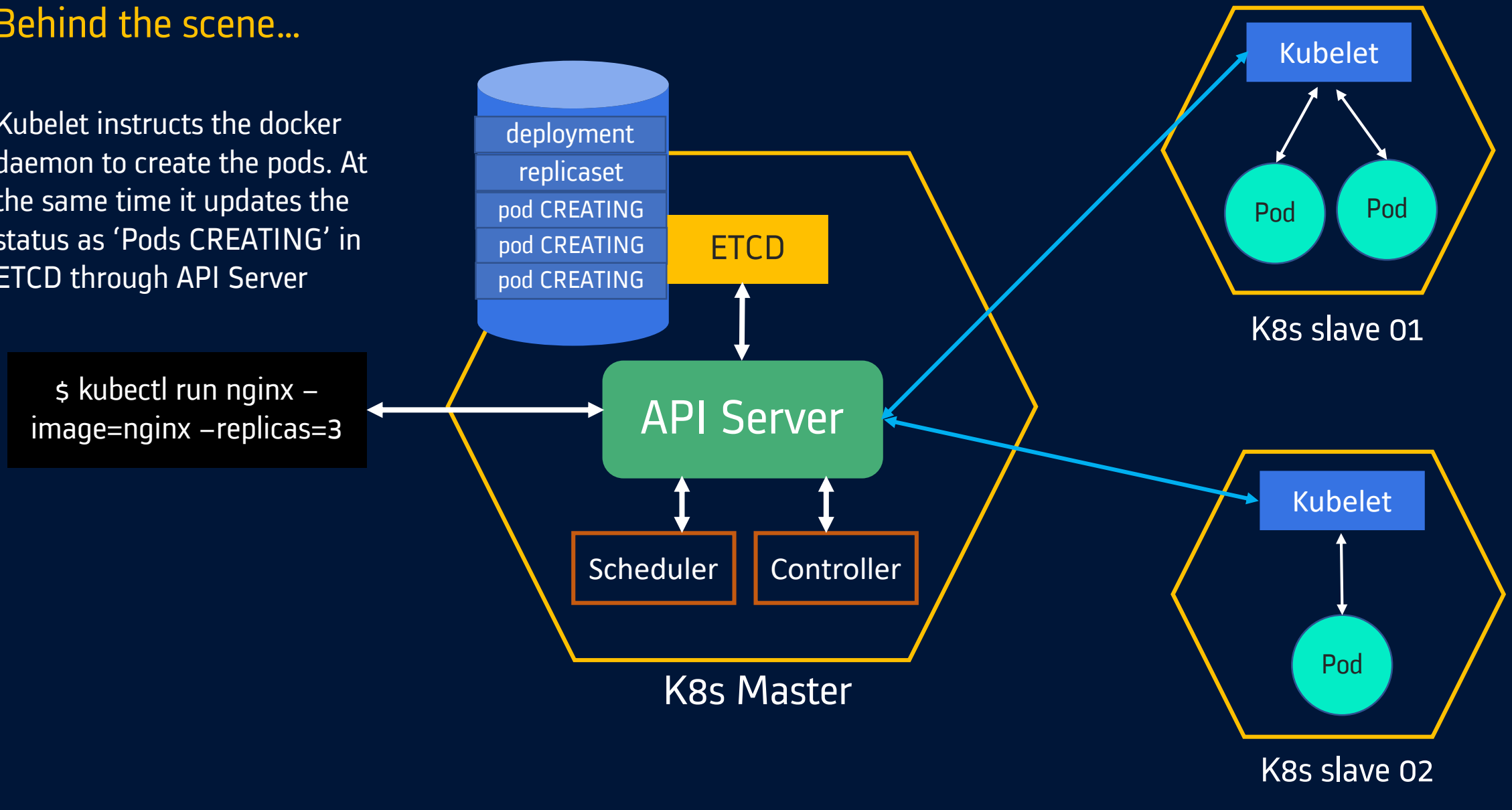
Kubelet identifies its workload through API-Server and understands that it needs to deploy some pods on its node



Kubernetes

Behind the scene...

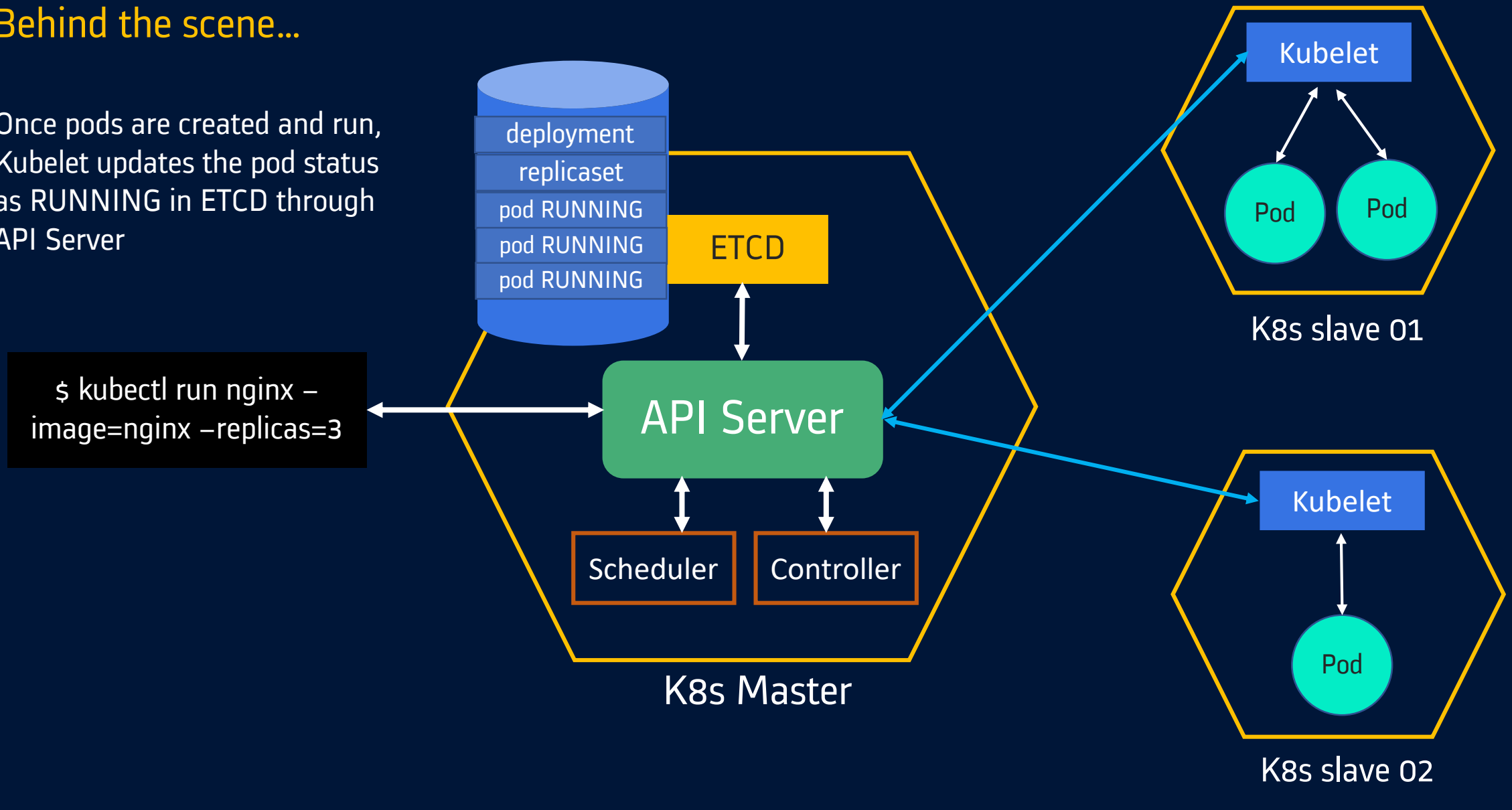
Kubelet instructs the docker daemon to create the pods. At the same time it updates the status as 'Pods CREATING' in ETCD through API Server



Kubernetes

Behind the scene...

Once pods are created and run, Kubelet updates the pod status as RUNNING in ETCD through API Server



Kubernetes

Labels and Selectors

Labels

- Labels are key/value pairs that are attached to objects, such as pods
- Labels allows to logically group certain objects by giving various names to them
- You can label pods, services, deployments and even nodes

```
"metadata": {
  "labels": {
    "key1" : "value1",
    "key2" : "value2"
  }
}
```

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

kubectl get pods -l environment=production
kubectl get pods -l environment=production, tier=frontend

https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

Kubernetes

Labels and Selectors

- If labels are not mentioned while deploying k8s objects using imperative commands, the label is auto set as app: <object-name>

kubectl run --image nginx nginx
kubectl get pods --show-labels

```
root@k8s-master:/home/osboxes# kubectl get pods --show-labels
NAME    READY   STATUS    RESTARTS   AGE   LABELS
nginx   1/1     Running   0          90s   run=nginx
```

Adding Labels

kubectl label pod nginx environment=dev

```
root@k8s-master:/home/osboxes# kubectl label pod nginx environment=dev
pod/nginx labeled
root@k8s-master:/home/osboxes# kubectl get pods --show-labels
NAME    READY   STATUS    RESTARTS   AGE     LABELS
nginx   1/1     Running   0          6m45s   environment=dev,run=nginx
root@k8s-master:/home/osboxes#
```

https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

Kubernetes

Labels and Selectors

Selectors

- Selectors allows to filter the objects based on labels
- The API currently supports two types of selectors: equality-based and set-based
- A label selector can be made of multiple requirements which are comma-separated

Equality-based Selector

- Equality- or inequality-based requirements allow filtering by label keys and values.
- Three kinds of operators are admitted =,==,!=

```
environment = production
tier != frontend
```

Used by Replication Controllers and Services

```
apiVersion: v1
kind: Pod
metadata:
  name: cuda-test
spec:
  containers:
    - name: cuda-test
      image: "k8s.gcr.io/cuda-vector-add:v0.1"
      resources:
        limits:
          nvidia.com/gpu: 1
  nodeSelector:
    accelerator: nvidia-tesla-p100
```

Kubernetes

Labels and Selectors

Selectors

- Selectors allows to filter the objects based on labels
- The API currently supports two types of selectors: equality-based and set-based
- A label selector can be made of multiple requirements which are comma-separated

Set-based Selector

- Set-based label requirements allow filtering keys according to a set of values.
- Three kinds of operators are supported: in,notin and exists (only the key identifier).

kubectl get pods -l 'environment in (production, qa)'

Used by ReplicaSets, Deployments, DaemonSets

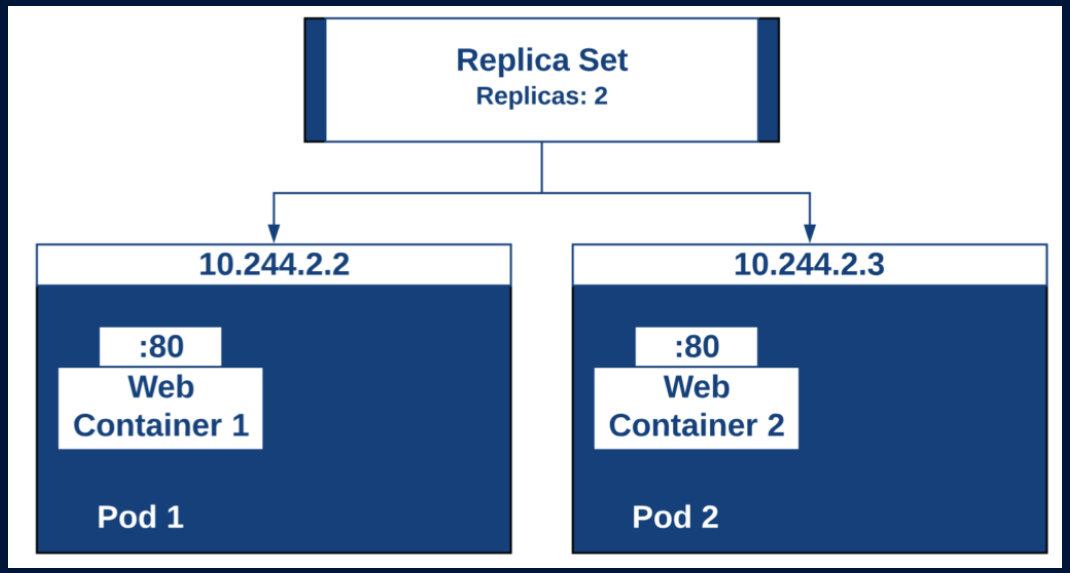
```
environment in (production, qa)
tier notin (frontend, backend)
partition
!partition
```

```
selector:
  matchLabels:
    component: redis
  matchExpressions:
    - {key: tier, operator: In, values: [cache]}
    - {key: environment, operator: NotIn, values: [dev]}
```

Kubernetes

ReplicaSet

- ReplicaSets are a higher-level API that gives the ability to easily run multiple instances of a given pod
- ReplicaSets ensures that the exact number of pods(replicas) are always running in the cluster by replacing any failed pods with new ones
- The replica count is controlled by the replicas field in the resource definition file
- Replicaset uses set-based selectors whereas replicacontroller uses equality based selectors

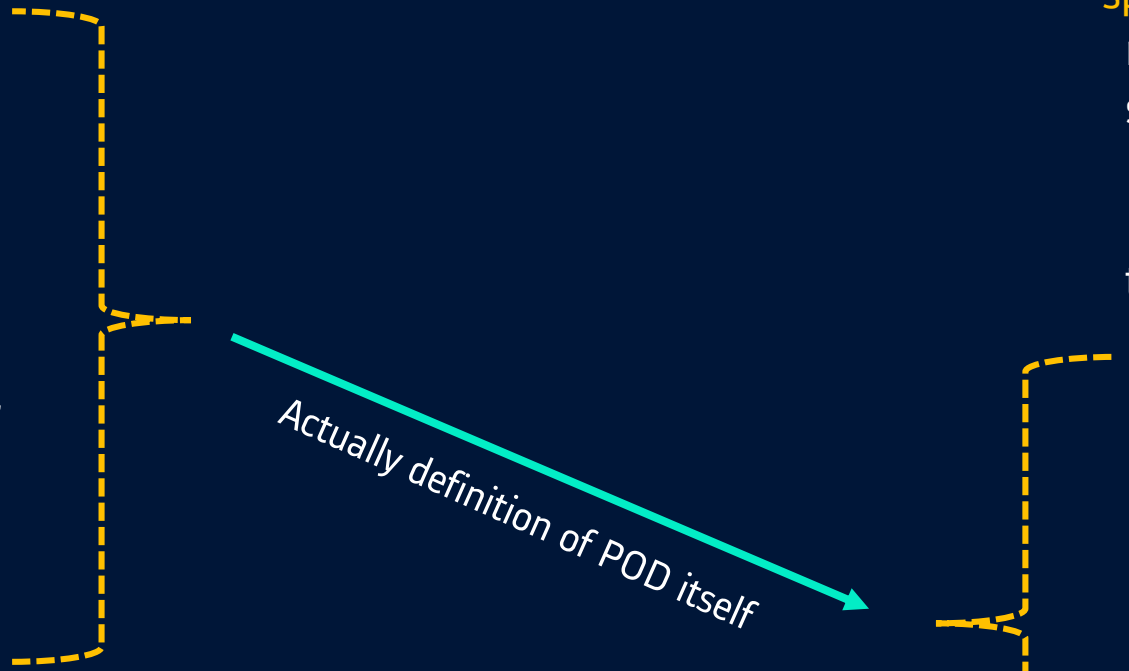


Kubernetes

Pod vs ReplicaSet

pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
    name: nginx-pod
    labels:
      app: webapp
  spec:
    containers:
      - name: nginx-container
        image: nginx
        ports:
        - containerPort: 80
```

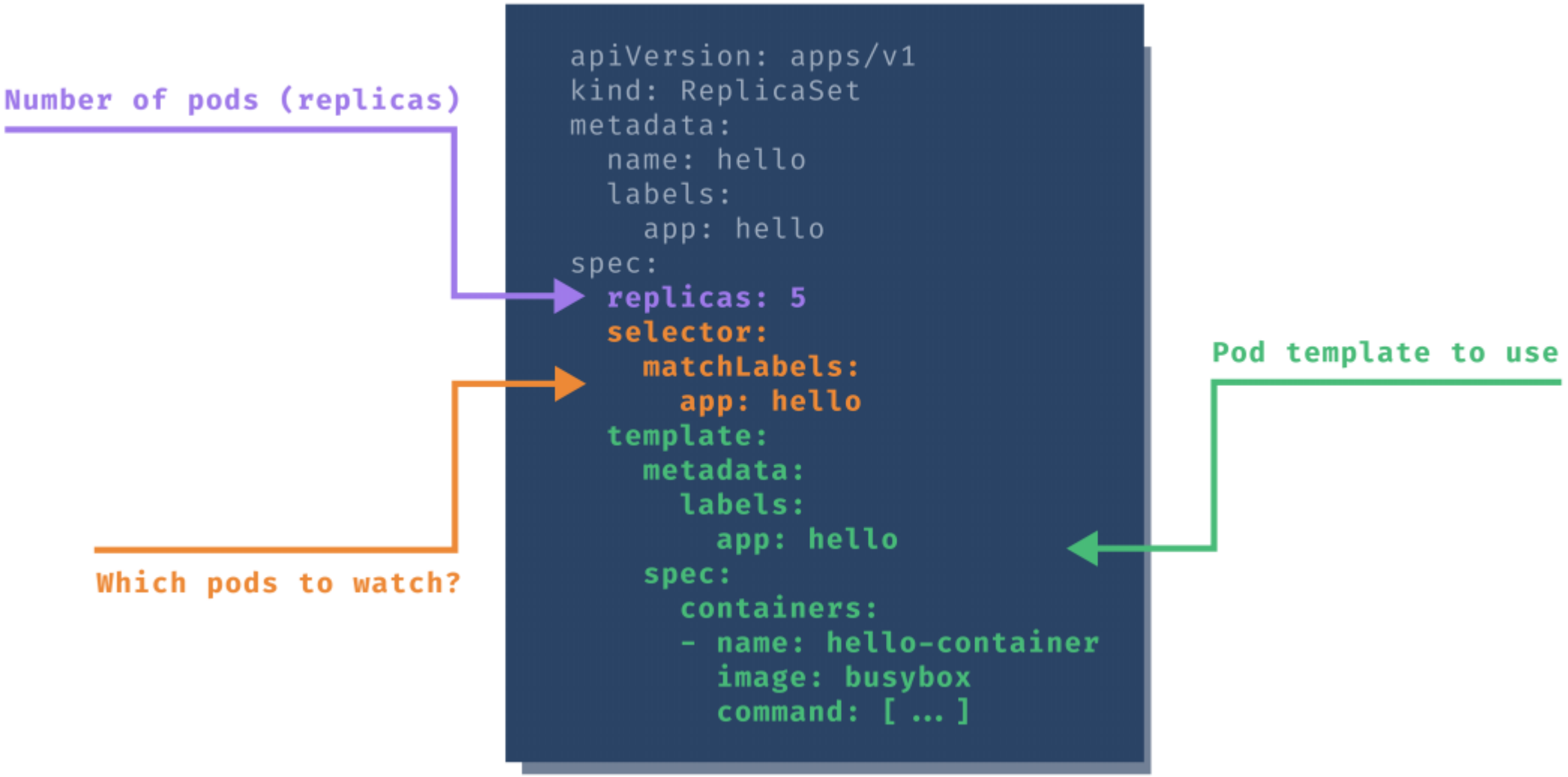


replicaset.yml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
  labels:
    app: webapp
    type: front-end
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      name: nginx-pod
      labels:
        app: webapp
    spec:
      containers:
        - name: nginx-container
          image: nginx
          ports:
          - containerPort: 80
```

Kubernetes

ReplicaSet Manifest file



Kubernetes

ReplicaSet

kubectl create –f replica-set.yml

```
root@k-master:/home/osboxes# kubectl create -f replica-set.yml
replicaset.apps/nginx-replicaset created
root@k-master:/home/osboxes#
```

kubectl get rs –o wide

```
root@k-master:/home/osboxes# kubectl get rs -o wide
NAME               DESIRED   CURRENT   READY   AGE   CONTAINERS        IMAGES   SELECTOR
nginx-replicaset   3         3         3       76s   nginx-container   nginx    app=webapp
root@k-master:/home/osboxes#
```

kubectl get pods –o wide

```
root@k-master:/home/osboxes# kubectl get pods -o wide
NAME                     READY   STATUS    RESTARTS   AGE   IP           NODE
nginx-replicaset-67nb9   1/1     Running   0          93s   10.244.1.9   k-slave01
nginx-replicaset-g85dz   1/1     Running   0          93s   10.244.2.4   k-slave02
nginx-replicaset-l9cpz   1/1     Running   0          93s   10.244.1.8   k-slave01
```

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
  labels:
    app: webapp
    type: front-end
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      name: nginx-pod
      labels:
        app: webapp
    spec:
      containers:
        - name: nginx-container
          image: nginx
          ports:
          - containerPort: 80
```

Kubernetes

ReplicaSet

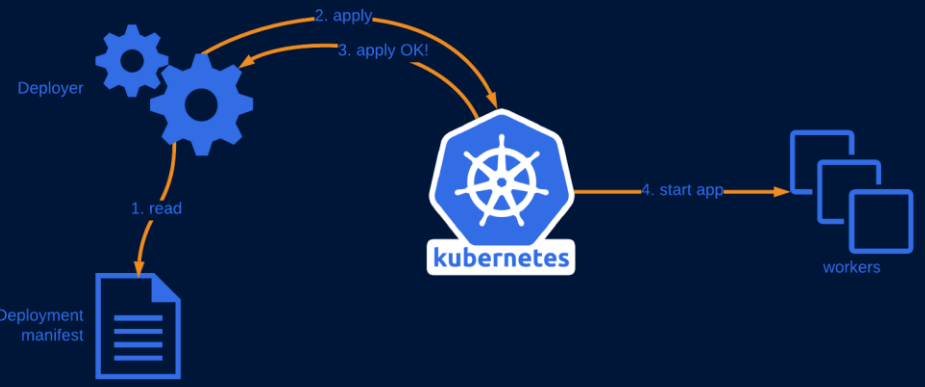
- kubectl edit replicaset <replicaset-name> - edit a replicaset; like image, replicas
- kubectl delete replicaset <replicaset-name> - delete a replicaset; like image, replicas
- kubectl delete -f replica-set.yml
- kubectl get all - get pods, replicasets, deployments, services all in one shot
- kubectl replace -f replicaset-definition.yml -replaces the pods with updated definition file
- kubectl scale --replicas=6 –f replicaset-definition.yml – scale using definition file
- kubectl scale --replicas=6 replicaset <replicaset-name> - using name of replicaset

Kubernetes Deployments

Kubernetes

Deployment

- A Deployment provides declarative updates for Pods and ReplicaSets.
- You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate.
- It seems similar to ReplicaSets but with advanced functions
- Deployment is the recommended way to deploy a pod or RS
- By default Kubernetes performs deployments in rolling update strategy.
- Below are some of the key features of deployment:
  - ✓ Easily deploy a RS
  - ✓ Rolling updates pods
  - ✓ Rollback to previous deployment versions
  - ✓ Scale deployment
  - ✓ Pause and resume deployment



Kubernetes

Deployment Strategy

- Whenever we create a new deployment, K8s triggers a Rollout.
- Rollout is the process of gradually deploying or upgrading your application containers.
- For every rollout/upgrade, a version history will be created, which helps in rolling back to working version in case of an update failure
- In Kubernetes there are a few different ways to release updates to an application
  - Recreate: terminate the old version and release the new one. Application experiences downtime.
  - RollingUpdate: release a new version on a rolling update fashion, one after the other. It's the default strategy in K8s. No application downtime is required.
  - Blue/green: release a new version alongside the old version then switch traffic

```
spec:
  replicas: 10
strategy:
    type: Recreate
```

```
spec:
  replicas: 10
strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 0
```

Kubernetes

Rolling Update Strategy

- By default, deployment ensures that only 25% of your pods are unavailable during an update and does not update more that 25% of the pods at a given time
- It does not kill old pods until/unless enough new pods come up
- It does not create new pods until a sufficient number of old pods are killed
- There are two settings you can tweak to control the process: maxUnavailable and maxSurge. Both have the default values set - 25%
- The maxUnavailable setting specifies the maximum number of pods that can be unavailable during the rollout process. You can set it to an actual number(integer) or a percentage of desired pods

    Let's say maxUnavailable is set to 40%. When the update starts, the old ReplicaSet is scaled down to 60%.
    As soon as new pods are started and ready, the old ReplicaSet is scaled down again and the new ReplicaSet is scaled up. This happens in such a way that the total number of available pods (old and new, since we are scaling up and down) is always at least 60%.

- The maxSurge setting specifies the maximum number of pods that can be created over the desired number of pods

    If we use the same percentage as before (40%), the new ReplicaSet is scaled up right away when the rollout starts. The new ReplicaSet will be scaled up in such a way that it does not exceed 140% of desired pods. As old pods get killed, the new ReplicaSet scales up again, making sure it never goes over the 140% of desired pods

Kubernetes

Deployments

- kubectl create deployment nginx --image nginx --dry-run -o yaml
- kubectl create -f deployment.yml --record (--record is optional, it just records the events in the deployment)

```
root@k-master:/home/osboxes# kubectl create -f deployment.yml
deployment.apps/nginx-deployment created
root@k-master:/home/osboxes#
```

- kubectl get deployments

```
root@k-master:/home/osboxes# kubectl get deployments -o wide
NAME               READY   UP-TO-DATE   AVAILABLE   AGE    CONTAINERS
nginx-deployment   5/5     5            5           117s   nginx-container
```

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

Kubernetes

Deployments

- kubectl describe deployment <deployment-name>

```
root@k-master:/home/osboxes# kubectl describe deployment nginx-deployment
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Tue, 19 May 2020 03:40:19 -0400
Labels:                 app=nginx
Annotations:            deployment.kubernetes.io/revision: 1
                        kubernetes.io/change-cause: kubectl create --filename=deployment.yml
Selector:               app=nginx
Replicas:               5 desired | 5 updated | 5 total | 5 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx-container:
    Image:        nginx
    Port:         80/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deployment-96577bc6d (5/5 replicas created)
```

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

Kubernetes

Deployments

- kubectl get pods –o wide

```
root@k-master:/home/osboxes# kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE     IP            NODE
TES
nginx-deployment-96577bc6d-2hkpk    1/1     Running   0          3m34s   10.244.2.20   k-slave02
nginx-deployment-96577bc6d-h5gdv    1/1     Running   0          3m34s   10.244.2.19   k-slave02
nginx-deployment-96577bc6d-nqtn6    1/1     Running   0          3m34s   10.244.2.22   k-slave02
nginx-deployment-96577bc6d-pd4cg    1/1     Running   0          3m34s   10.244.2.21   k-slave02
nginx-deployment-96577bc6d-tphhh    1/1     Running   0          3m34s   10.244.2.23   k-slave02
root@k-master:/home/osboxes#
```

- kubectl edit deployment <deployment -name> - perform live edit of deployment
- kubectl scale deployment <deployment -name> --replicas2
- kubectl apply –f deployment.yml – redeploy a modified yaml file; Ex: replicas changed to 5, image to nginx:1.18

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

Kubernetes

Deployments

- kubectl rollout status deployment <deployment -name>

```
root@k-master:/home/osboxes# k rollout status deployment.apps/nginx-deployment
Waiting for deployment "nginx-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 2 old replicas are pending termination...
Waiting for deployment "nginx-deployment" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "nginx-deployment" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "nginx-deployment" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "nginx-deployment" rollout to finish: 4 of 5 updated replicas are available...
deployment "nginx-deployment" successfully rolled out
```

- kubectl rollout history deployment <deployment -name>

```
root@k-master:/home/osboxes# k rollout history deployment.apps/nginx-deployment
deployment.apps/nginx-deployment
REVISION  CHANGE-CAUSE
1         kubectl create --filename=deployment.yml --record=true
2         kubectl create --filename=deployment.yml --record=true
```

Kubernetes

Deployments

- kubectl rollout undo deployment <deployment -name>

```
root@k-master:/home/osboxes# kubectl rollout undo deployment.apps/nginx-deployment
deployment.apps/nginx-deployment rolled back
root@k-master:/home/osboxes# k rollout history deployment.apps/nginx-deployment
deployment.apps/nginx-deployment
REVISION  CHANGE-CAUSE
2         kubectl create --filename=deployment.yml --record=true
3         kubectl create --filename=deployment.yml --record=true
```

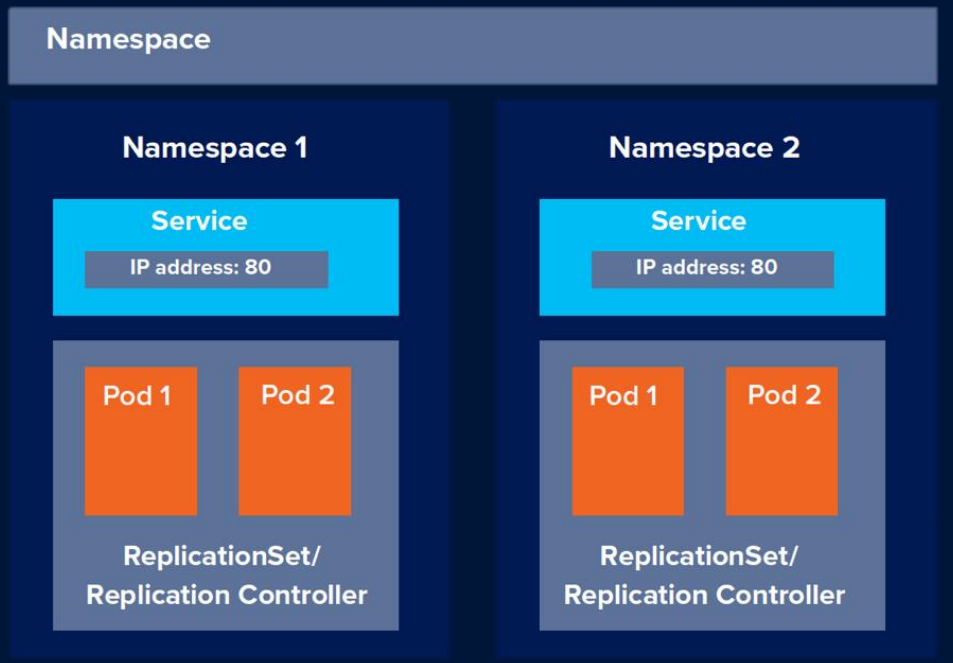
- kubectl rollout undo deployment <deployment -name> --to-revision=1
- kubectl rollout pause deployment <deployment -name>
- kubectl rollout resume deployment <deployment -name>
- kubectl delete -f <deployment-yaml-file> - deletes deployment and related dependencies
- kubectl delete all --all – deletes pods, replicasets, deployments and services in current namespace

Kubernetes

Namespaces

Namespaces are Kubernetes objects which partition a single Kubernetes cluster into multiple virtual clusters

- Kubernetes clusters can manage large numbers of unrelated workloads concurrently and organizations often choose to deploy projects created by separate teams to shared clusters.
- With multiple deployments in a single cluster, there are high chances of deleting deployments belong to deff prohjects.
- So namespaces allow you to group objects together so you can filter and control them as a unit/group.
- Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces.
- So each Kubernetes namespace provides the scope for Kubernetes Names it contains; which means that using the combination of an object name and a Namespace, each object gets a unique identity across the cluster



Kubernetes

Namespaces

By default, a Kubernetes cluster is created with the following three namespaces:

- default: It's a default namespace for users. By default, all the resource created in Kubernetes cluster are created in the default namespace
- Kube-system: It is the Namespace for objects created by Kubernetes systems/control plane. Any changes to objects in this namespace would cause irreparable damage to the cluster itself
- kube-public: Namespace for resources that are publicly readable by all users. This namespace is generally reserved for cluster usage like Configmaps and Secrets

Kubernetes

Namespaces

kubectl get namespaces

```
root@k8s-master:/home/osboxes# kubectl get ns
NAME              STATUS   AGE
default           Active   68m
kube-node-lease   Active   68m
kube-public       Active   68m
kube-system       Active   68m
```

kubectl get all -n kube-system (lists available objects under a specific namespace)

```
root@k8s-master:/home/osboxes# kubectl get all -n kube-system
NAME                                       READY   STATUS    RESTARTS   AGE
pod/coredns-66bff467f8-bm6kr               1/1     Running   0          68m
pod/coredns-66bff467f8-hj9ll               1/1     Running   0          68m
pod/etcd-k8s-master                        1/1     Running   0          68m
pod/kube-apiserver-k8s-master              1/1     Running   0          68m
pod/kube-controller-manager-k8s-master     1/1     Running   0          68m
pod/kube-flannel-ds-amd64-bc5vg            1/1     Running   0          67m
pod/kube-flannel-ds-amd64-cg48x            1/1     Running   0          68m
pod/kube-flannel-ds-amd64-xz8qn            1/1     Running   0          67m
pod/kube-proxy-rq77v                       1/1     Running   0          68m
pod/kube-proxy-tl99m                       1/1     Running   0          67m
pod/kube-proxy-w7bqz                       1/1     Running   0          67m
pod/kube-scheduler-k8s-master              1/1     Running   0          68m

NAME               TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
service/kube-dns   ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   68m
```

kubectl get all --all-namespaces (lists available objects under all available namespaces)

Kubernetes

Namespaces

Create a namespace

```
kubectl create ns dev # Namespace for Developer team
kubectl create ns qa # Namespace for QA team
kubectl create ns production # Namespace for Production team
```

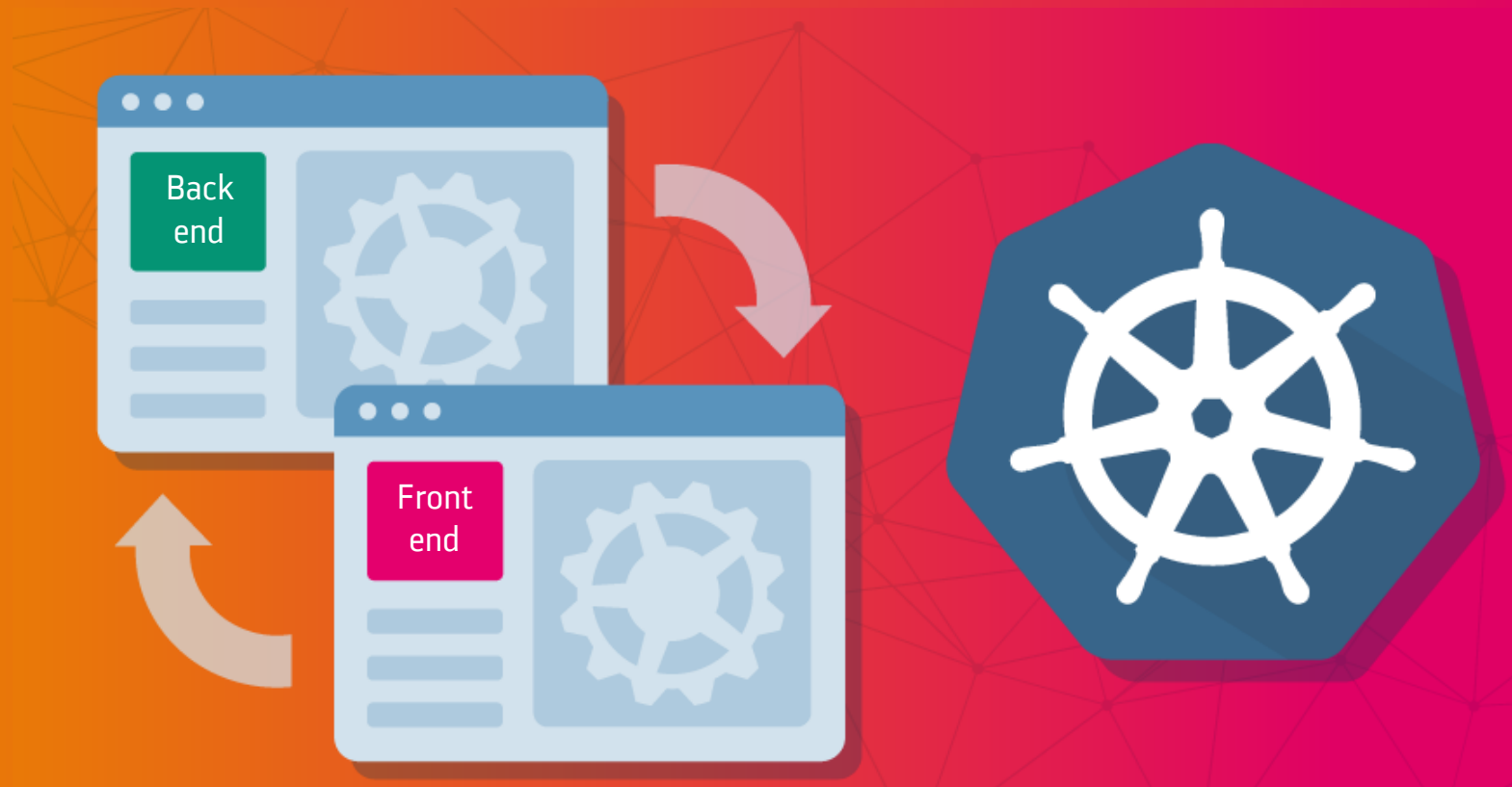
Deploy objects in a namespace

```
kubectl run nginx --image=nginx -n dev
kubectl get pod/nginx –n dev
kubectl apply --namespace=qa -f pod.yaml
```

```
root@k8s-master:/home/osboxes# kubectl run --image=nginx nginx -n dev
pod/nginx created
root@k8s-master:/home/osboxes# k get pods -n dev
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          7m37s
root@k8s-master:/home/osboxes#
```

Delete a namespace

```
kubectl delete ns production
```



Kubernetes Services

Kubernetes

Services

- Services logically connect pods across the cluster to enable networking between them
- The lifetime of an individual pod cannot be relied upon; everything from their IP addresses to their very existence are prone to change.
- Kubernetes doesn't treat its pods as unique, long-running instances; if a pod encounters an issue and dies, it's Kubernetes' job to replace it so that the application doesn't experience any downtime
- Services makes sure that even after a pod(back-end) dies because of a failure, the newly created pods will be reached by its dependency pods(front-end) via services. In this case, front-end applications always find the backend applications via a simple service(using service name or IP address) irrespective of their location in the cluster
- Services point to pods directly using labels. Services do not point to deployments or ReplicaSets. So, all pods with the same label gets attached to same service
- 3 types: ClusterIP, NodePort and LoadBalancer

Kubernetes

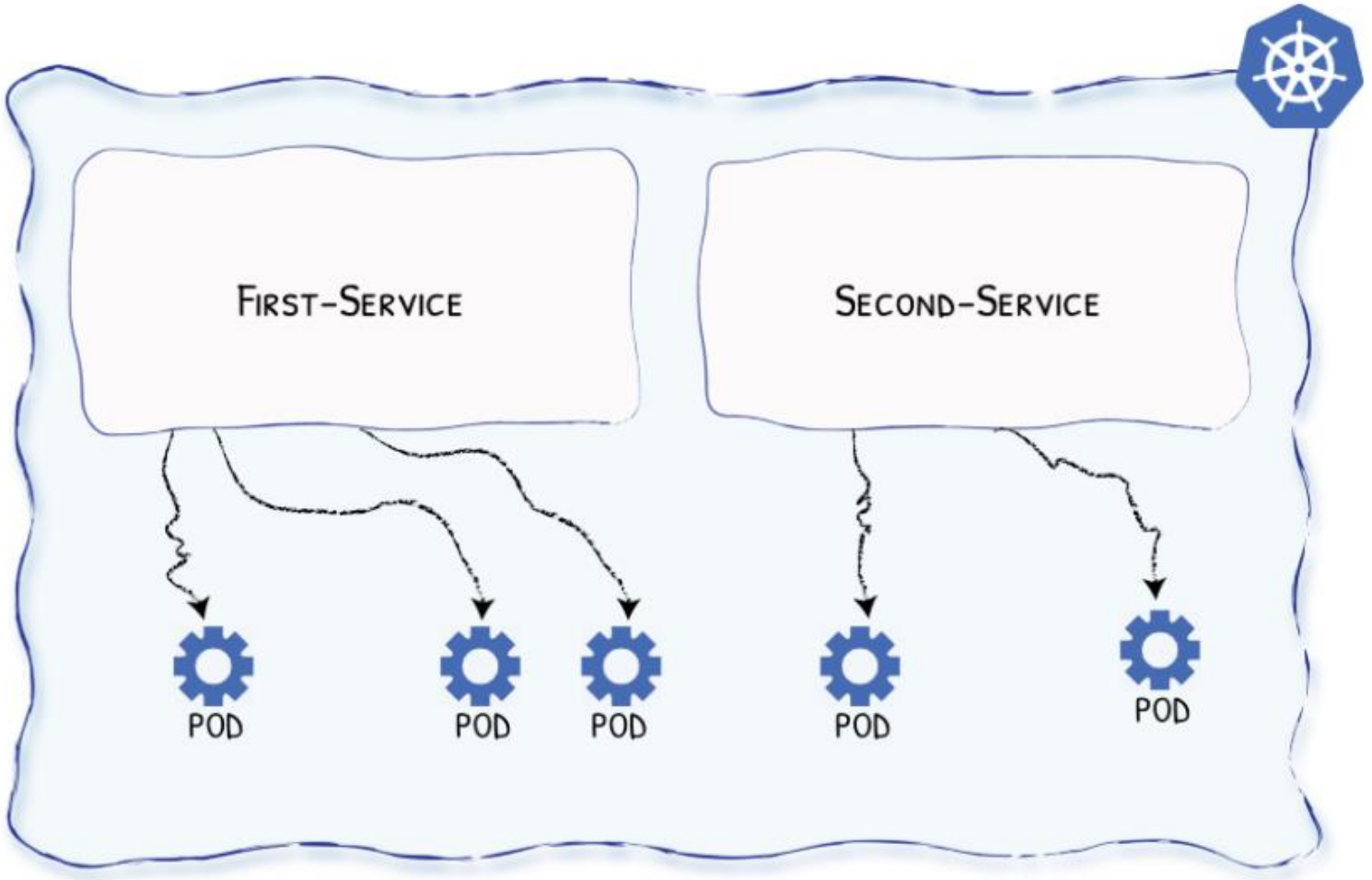
Services

Pods' lifecycle are erratic; they come and go by Kubernetes' will.

Not healthy? Killed.
Not in the right place? Cloned, and killed.

So how can you send a request to your application if you can't know for sure where it lives?
The answer lies in **services**.

Services are tied to the pods using pod labels and provides a stable end point for the users to reach the application.



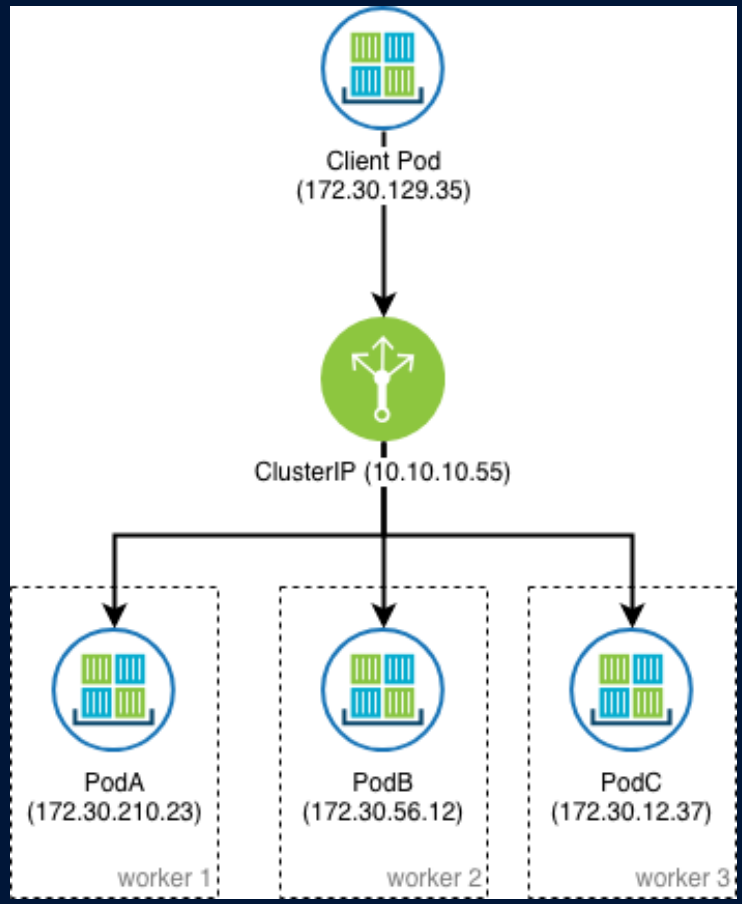
When requesting your application, you don't care about its location or about which pod answers the request.

Kubernetes

Services

ClusterIP

- ClusterIP service is the default Kubernetes service.
- It gives you a service inside your cluster that other apps inside your cluster can access
- It restricts access to the application within the cluster itself and no external access
- Useful when a front-end app wants to communicate with back-end
- Each ClusterIP service gets a unique IP address inside the cluster
- Similar to --links in Docker



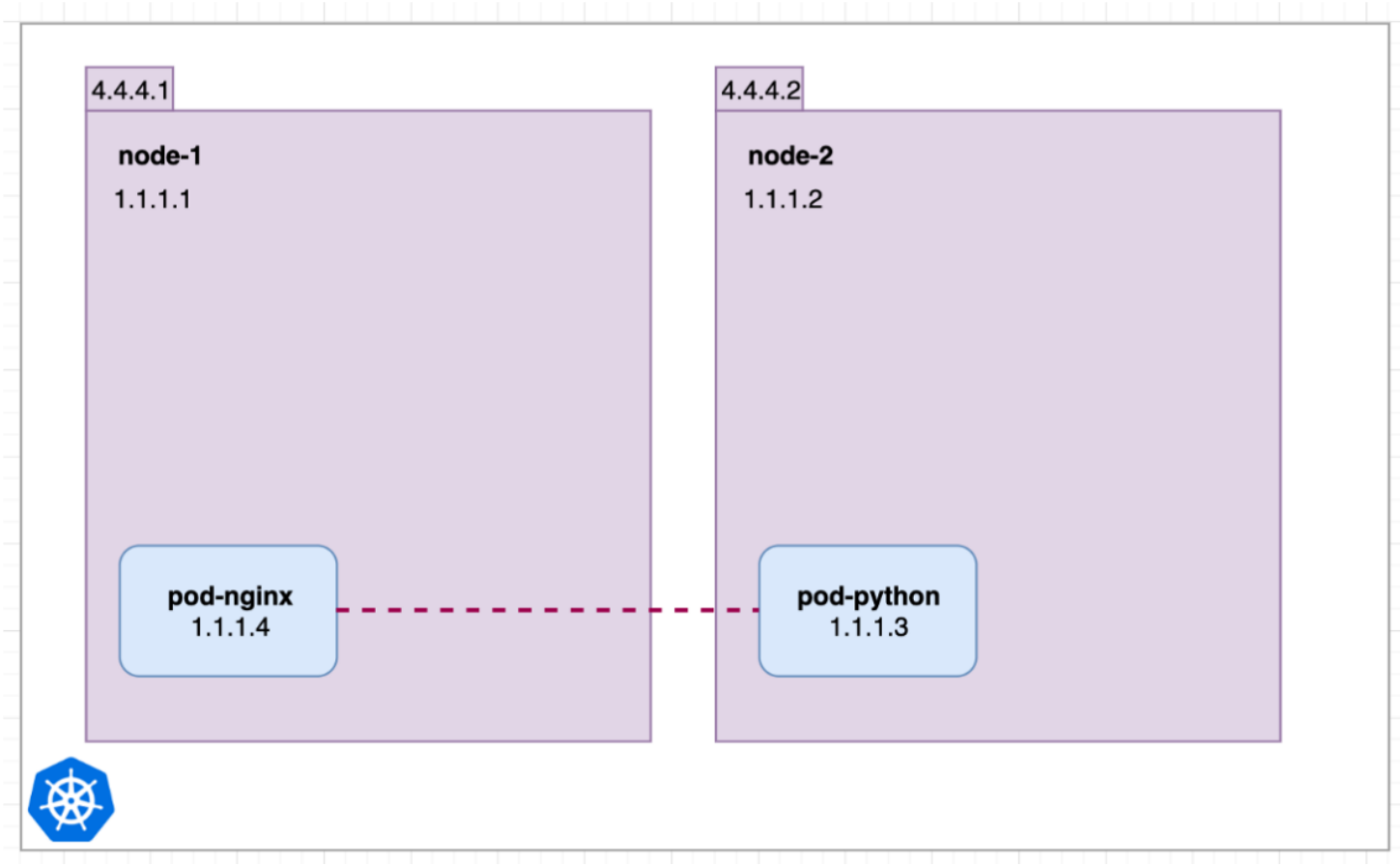
Services point to pods directly using labels!!!

https://kubernetes.io/docs/concepts/services-networking/service/

Kubernetes

When services are not available

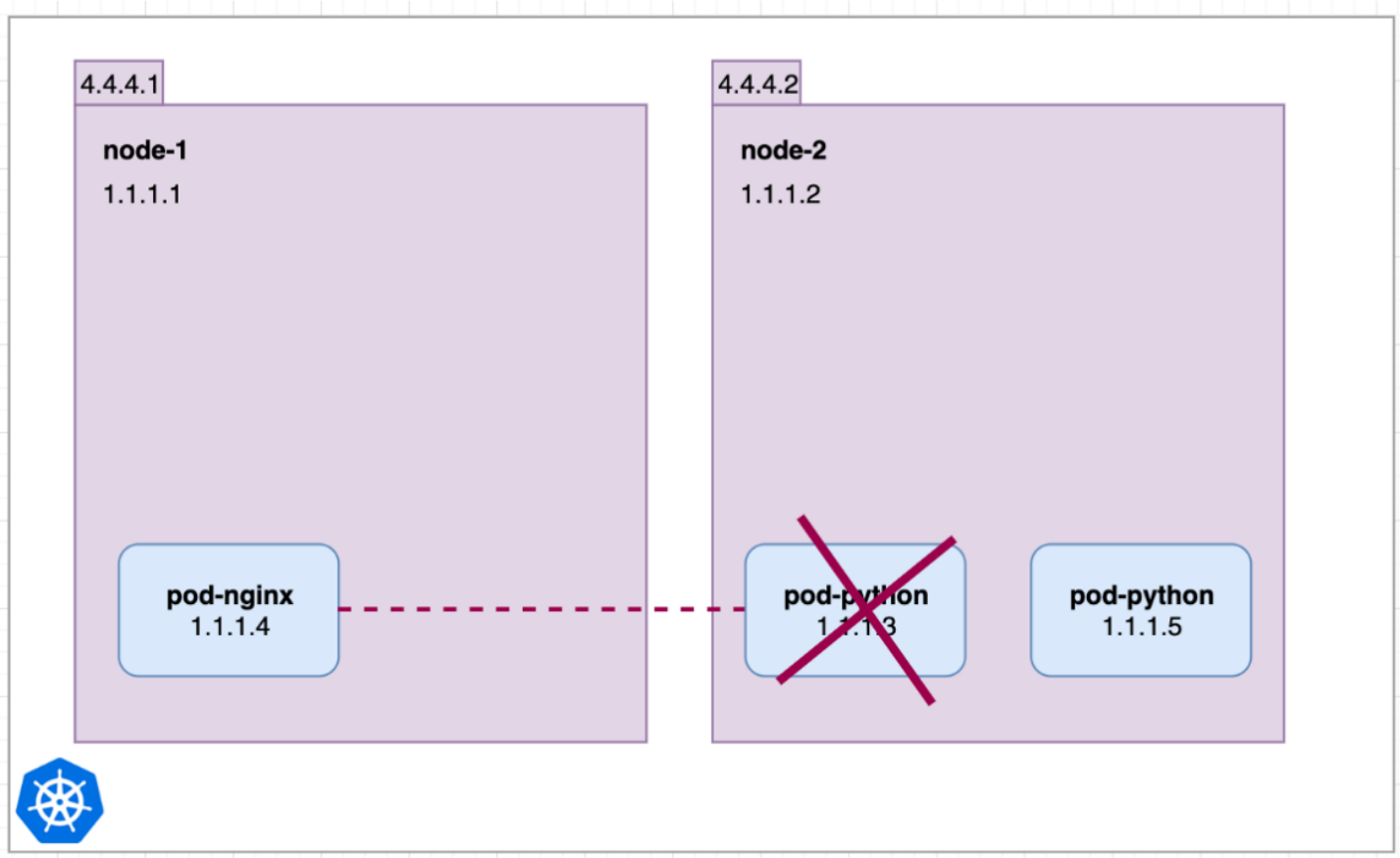
- Imagine 2 pods on 2 separate nodes node-1 & node-2 with their local IP address
- pod-nginx can ping and connect to pod-python using its internal IP 1.1.1.3.



Kubernetes

When services are not available

- Now let's imagine the pod-python dies and a new one is created.
- Now pod-nginx cannot reach pod-python on 1.1.1.3 because its IP is changed to 1.1.1.5.

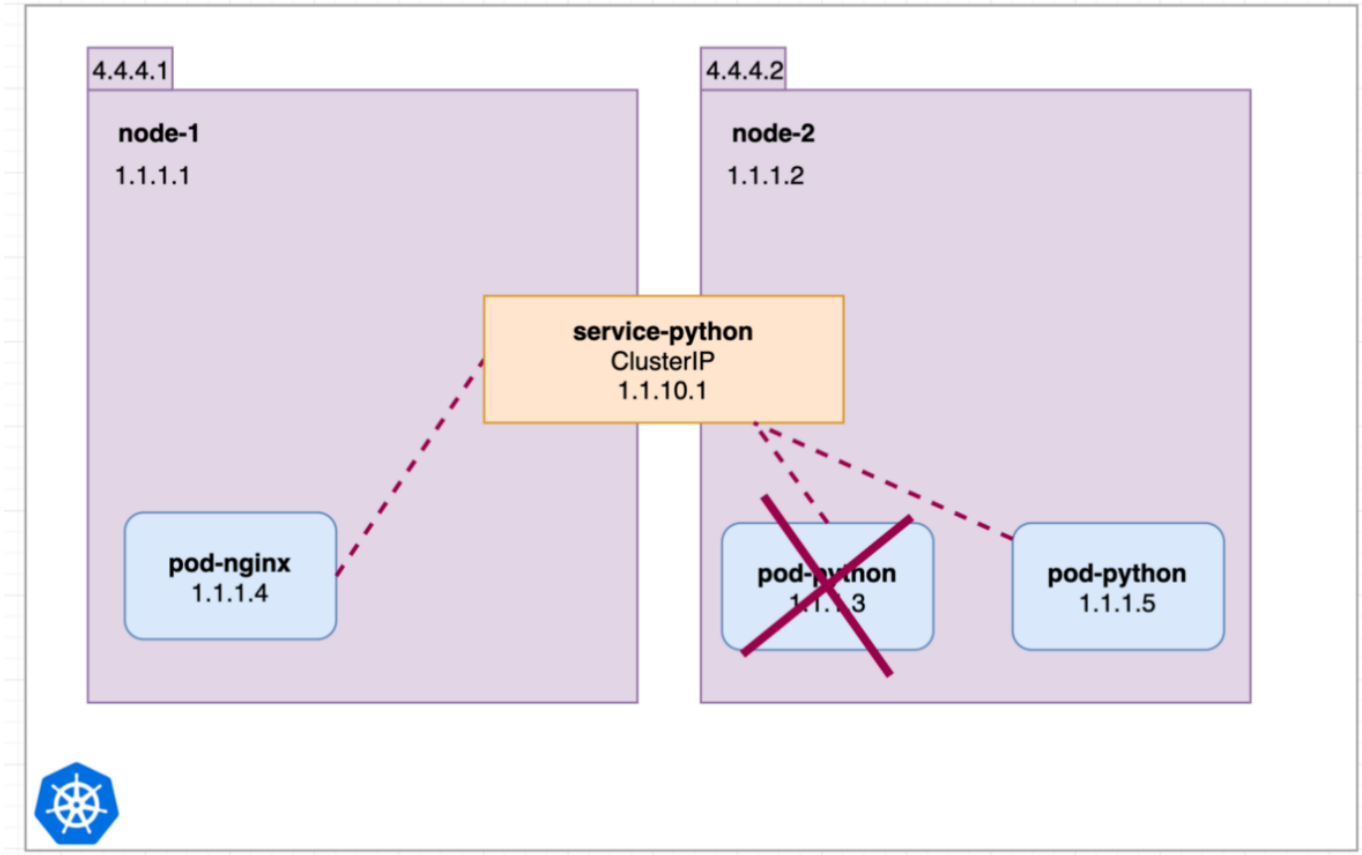


How do we remove this dependency?

Kubernetes

Enter services...

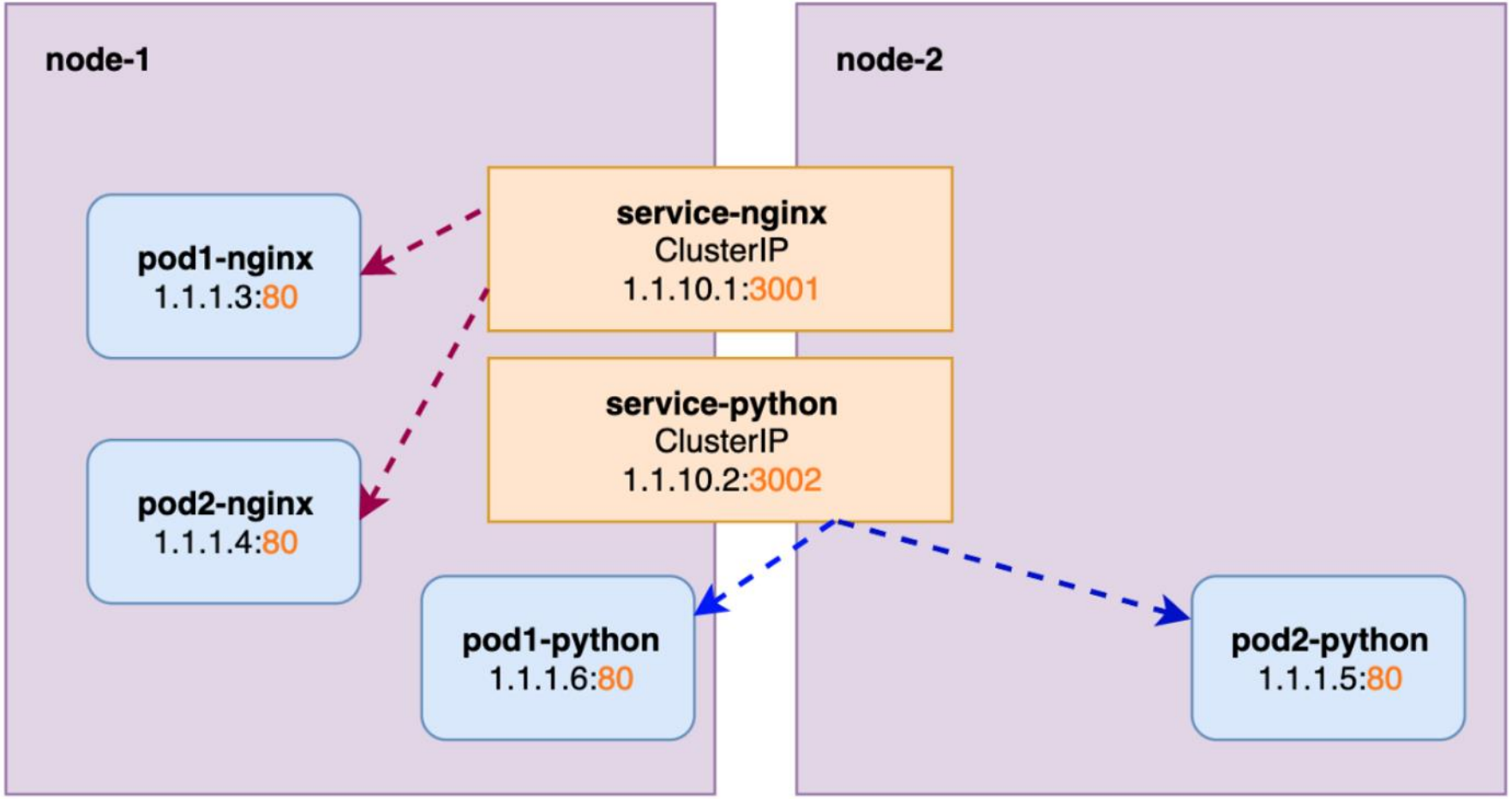
- Services logically connects pods together
- Unlike pods, a service is not scheduled on a specific node. It spans across the cluster
- Pod-nginx can always safely connect to pod-python using service IP 1.1.10.1 or the DNS name of service (service-python)
- Even if the python pod gets deleted and recreated again, nginx pod can still reach python pod using the service but not with IP of python pod directly



Kubernetes

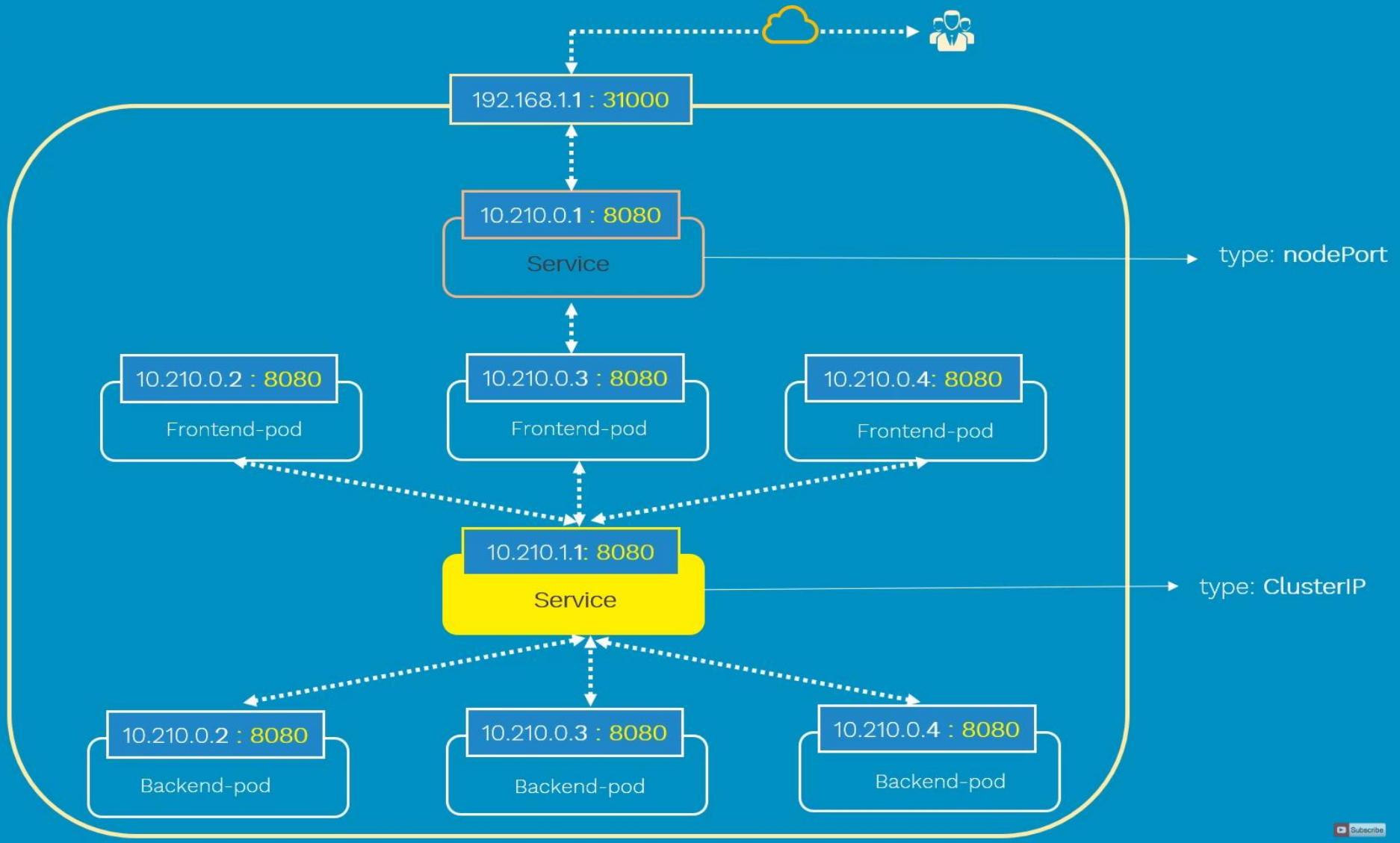
Enter services...

- Multiple ClusterIP services



Kubernetes

ClusterIP



Kubernetes

Services

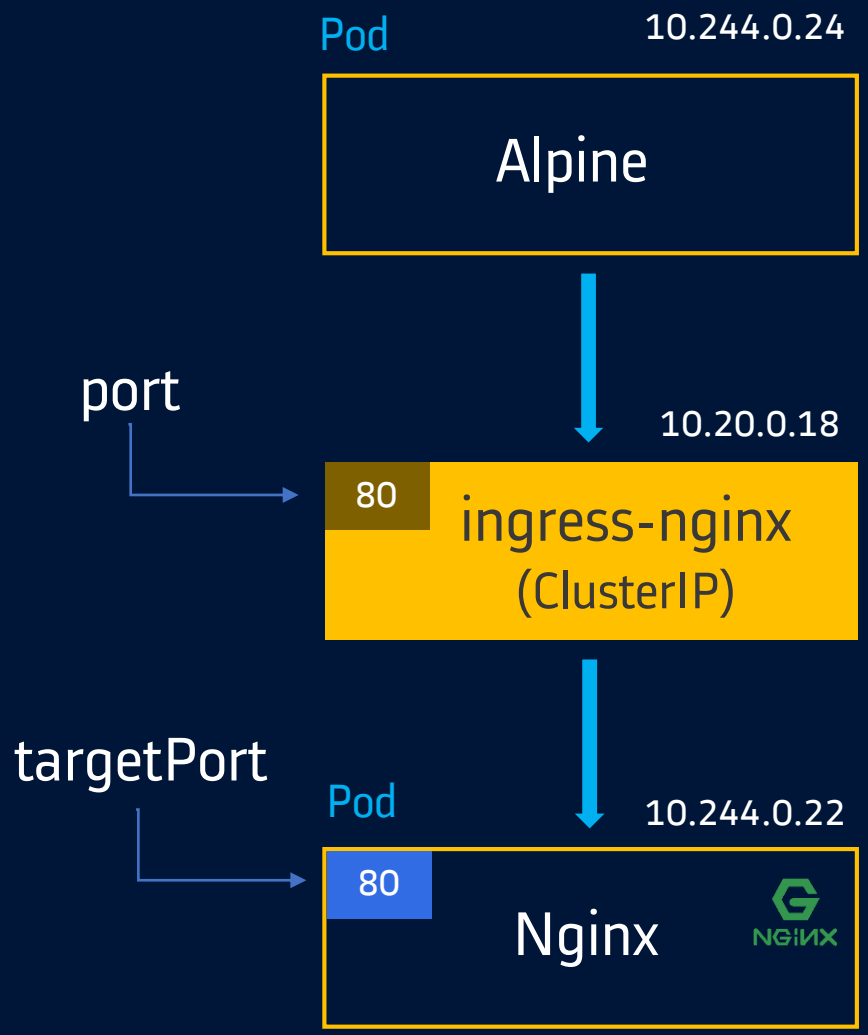
ClusterIP

clusterservice.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: nginx-backend
```

pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: backend-pod
  labels:
    app: nginx-backend
spec:
  containers:
  - name: nginx-container
    image: nginx
    ports:
      - containerPort: 80
```



Kubernetes

Services

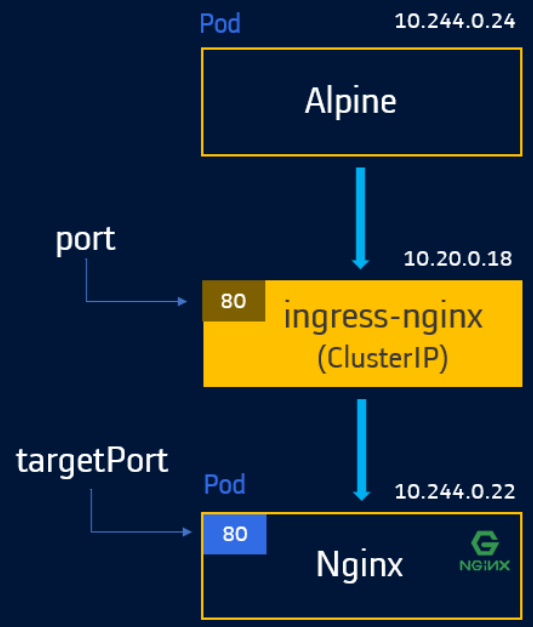
ClusterIP

clusterip-service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: nginx-backend
```

pod.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: backend-pod
  labels:
    app: nginx-backend
spec:
  containers:
  - name: nginx-container
    image: nginx
    ports:
      - containerPort: 80
```



kubectl create –f clusterservice.yml
kubectl create –f pod.yml

root@alpine: # curl ingress-nginx

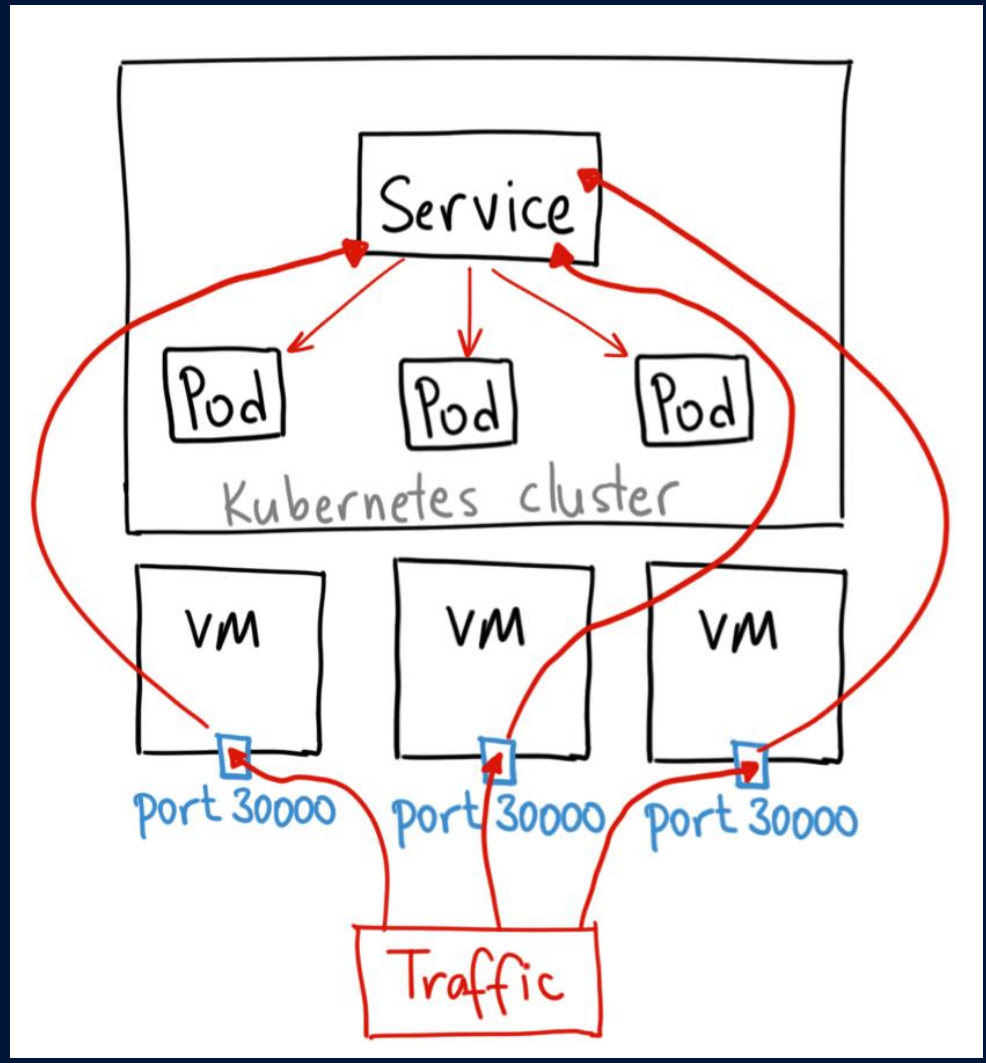
check the endpoints: kubectl describe svc/<svc-name>

Kubernetes

Services

NodePort

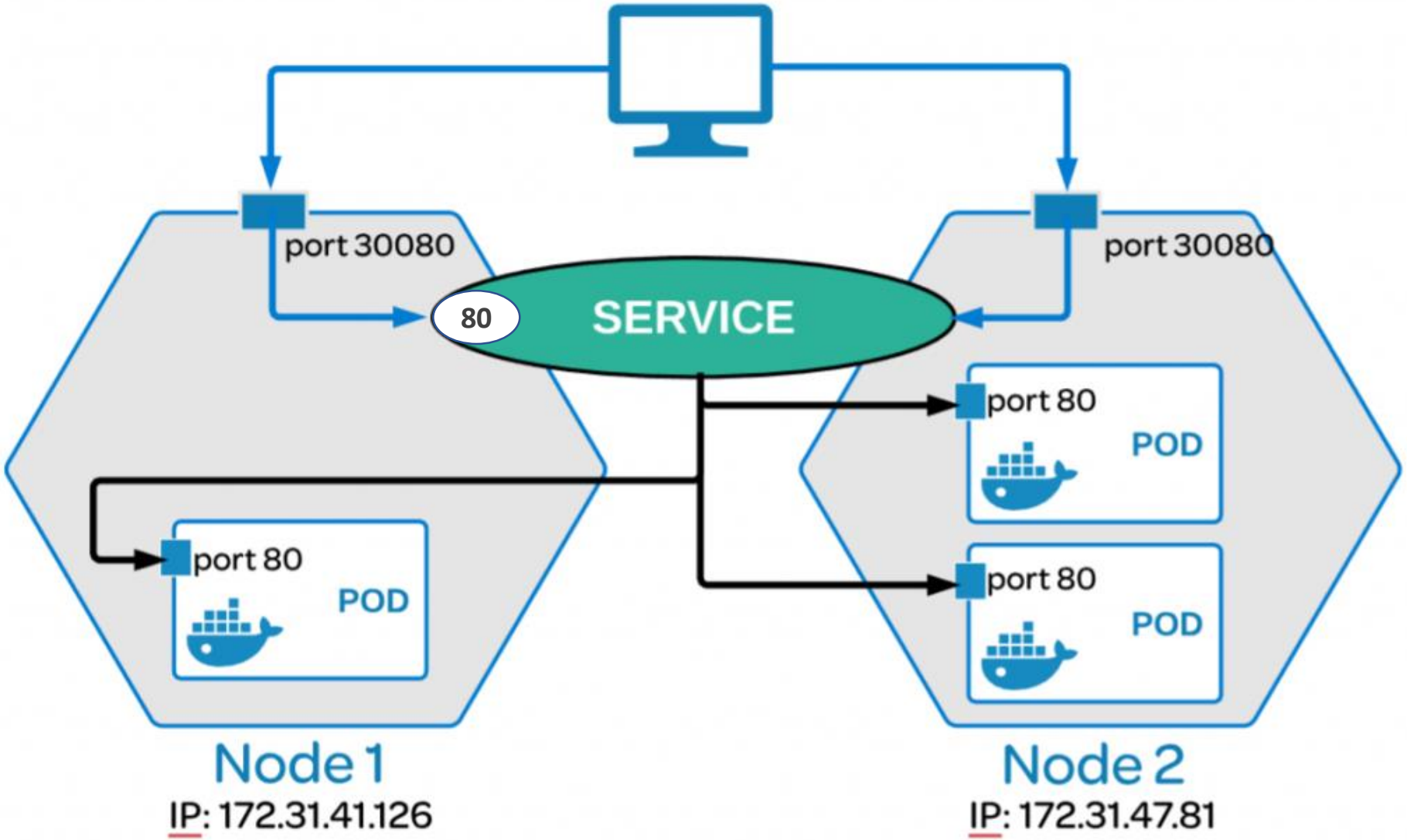
- NodePort opens a specific port on all the Nodes in the cluster and forwards any traffic that is received on this port to internal services
- Useful when front end pods are to be exposed outside the cluster for users to access it
- NodePort is build on top of ClusterIP service by exposing the ClusterIP service outside of the cluster
- NodePort must be within the port range 30000-32767
- If you don't specify this port, a random port will be assigned. It is recommended to let k8s auto assign this port



Kubernetes

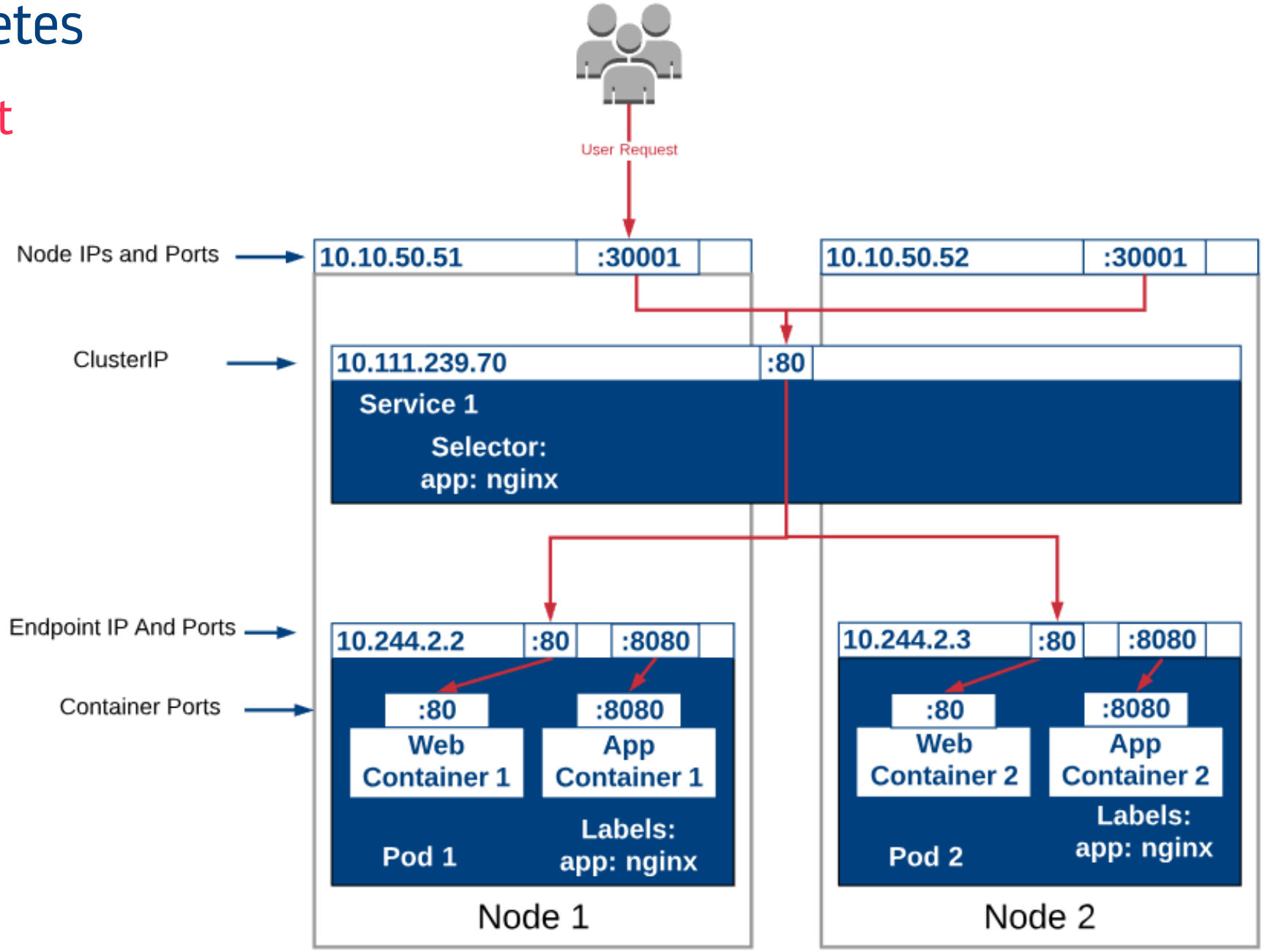
NodePort

```
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30080
```



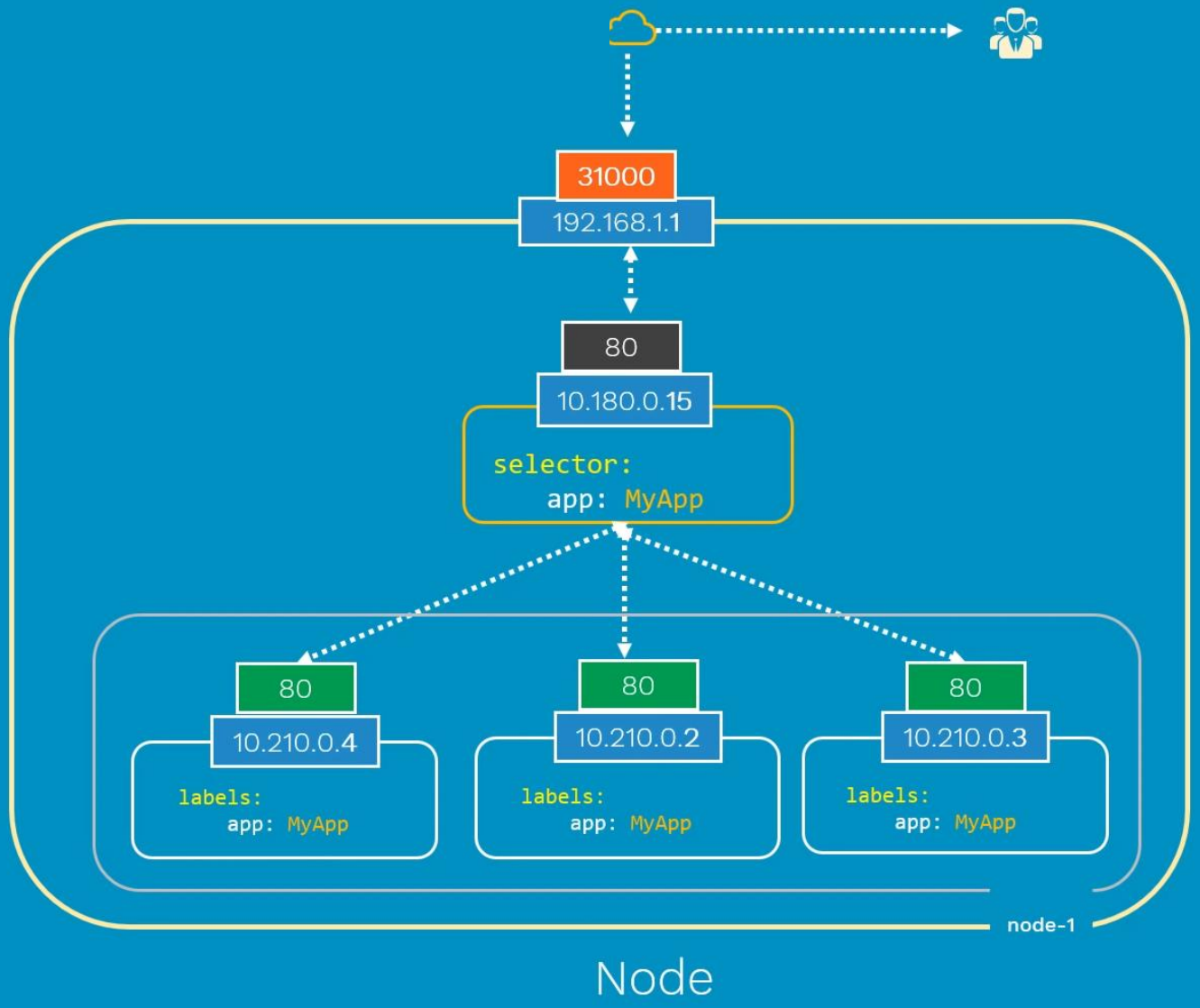
Kubernetes

NodePort



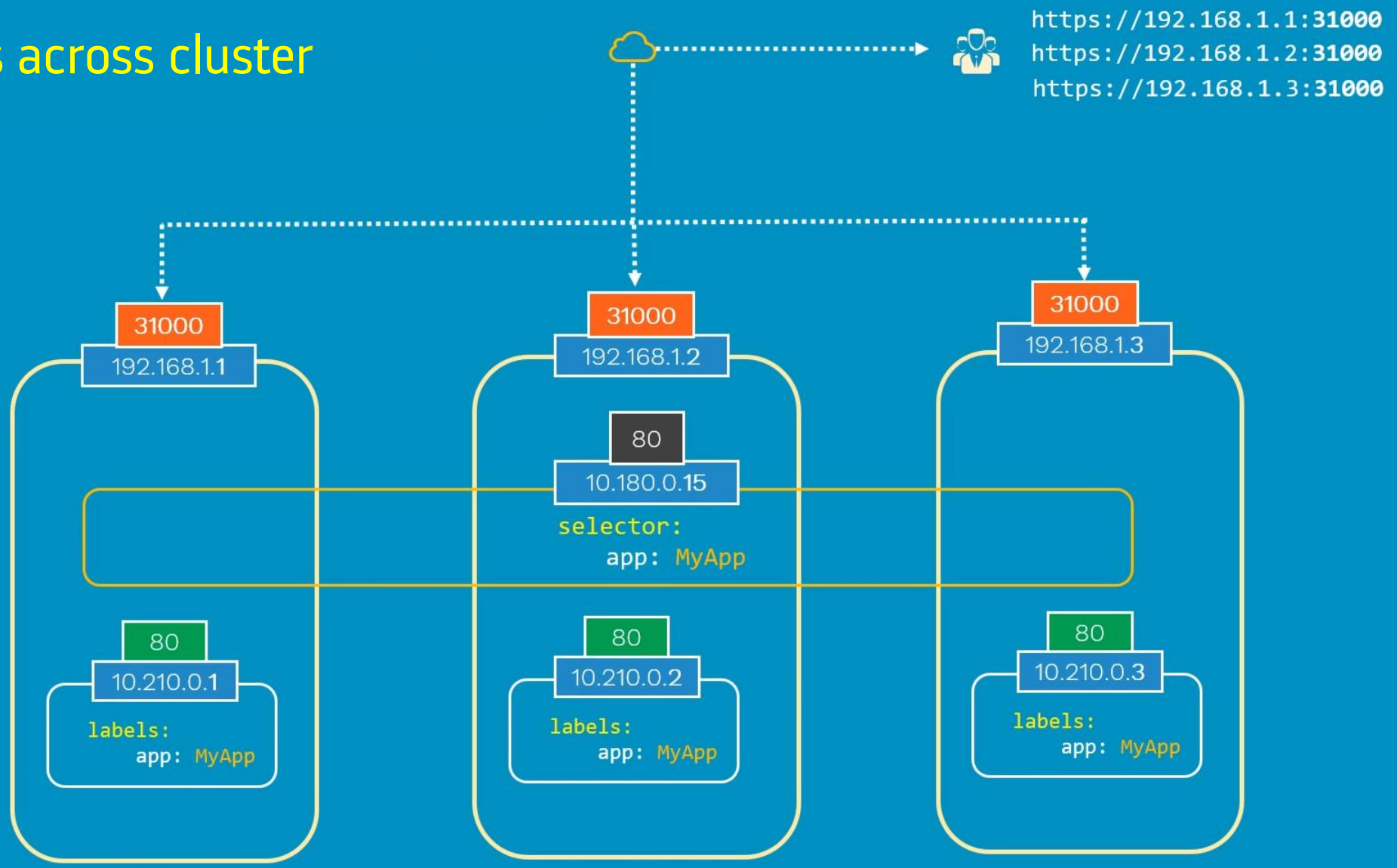
Kubernetes

Multi Instances in same node



Kubernetes

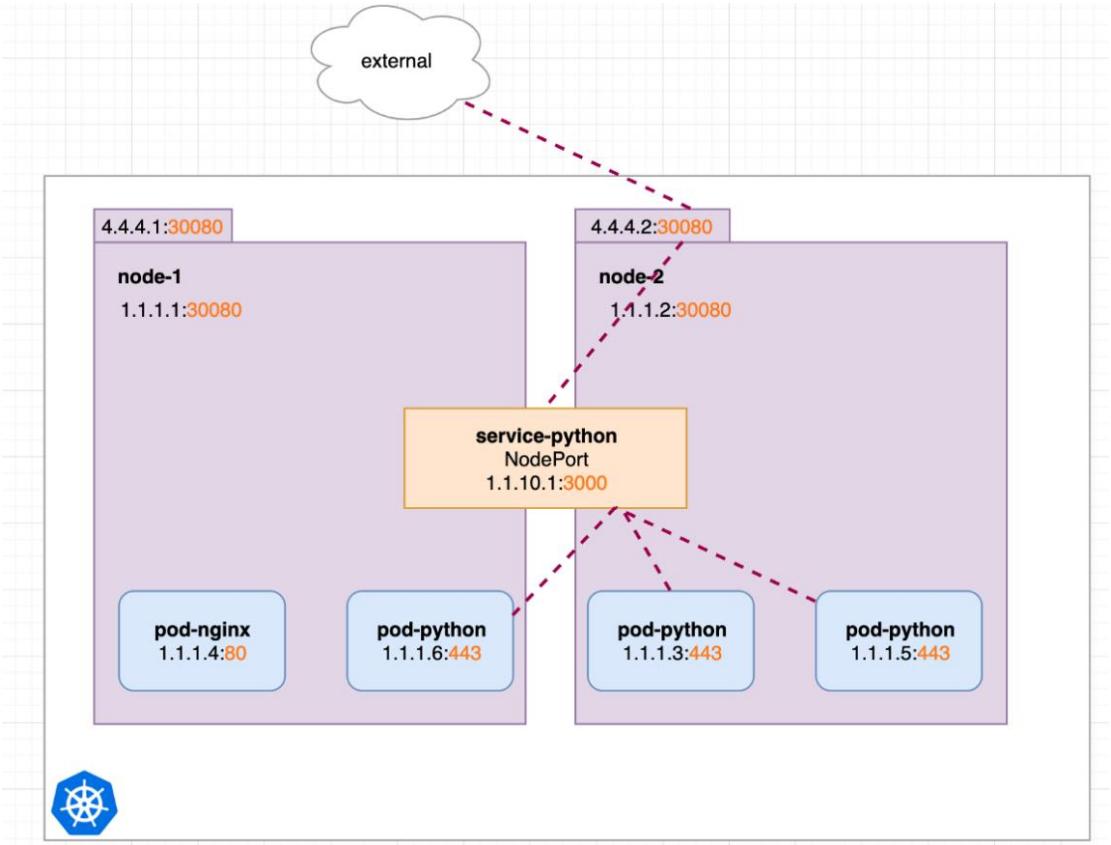
Multi Instances across cluster

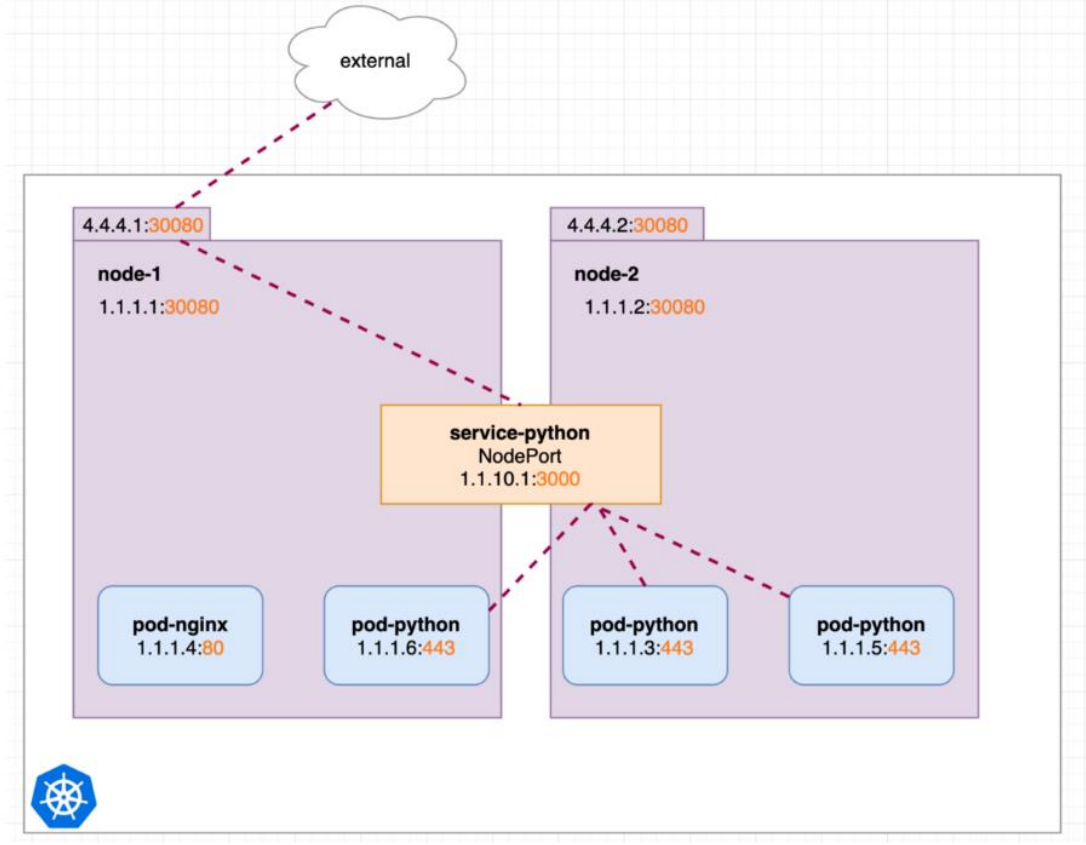


Kubernetes

NodePort

- Application can be reached from any of the available nodes in the cluster using **<node-ip>:<node-port>**





Kubernetes

NodePort

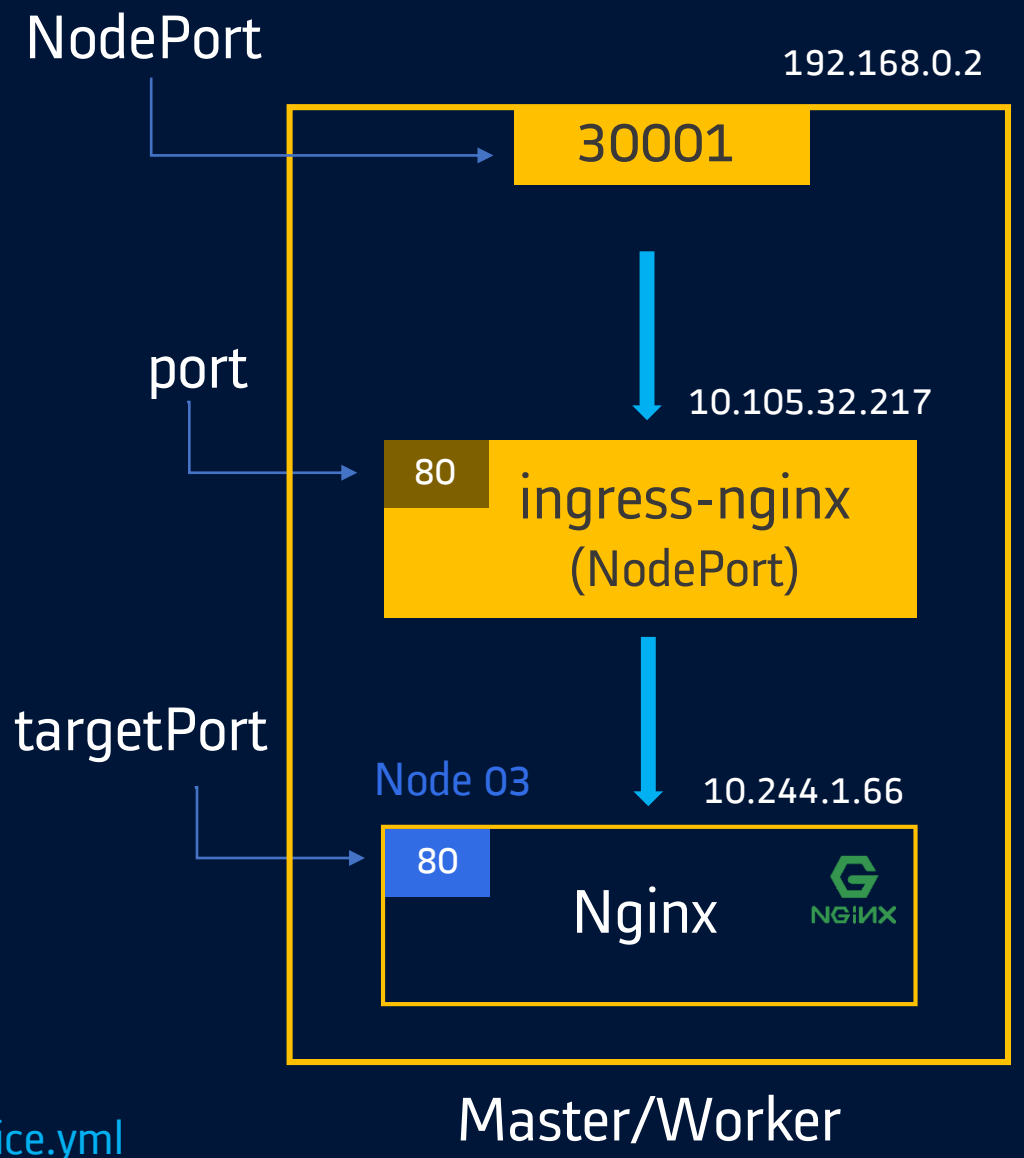
nodeport-service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport-service
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30001
    protocol: TCP
  selector:
    app: nginx-frontend
```

pod.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-frontend
  labels:
    app: nginx-frontend
spec:
  containers:
  - name: nginx-container
    image: nginx
    ports:
      - containerPort: 80
```

```
kubectl create –f nodeportservice.yml
kubectl create –f pod.yml
```

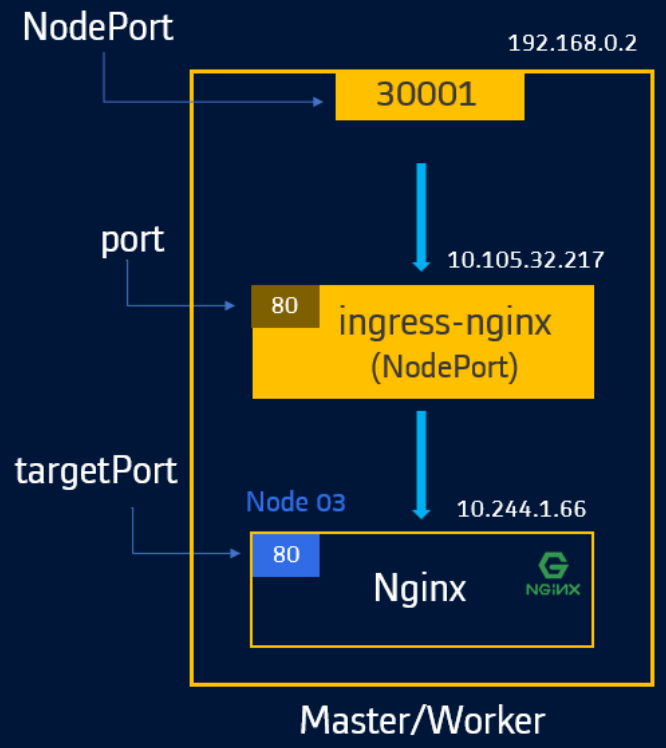


Kubernetes

Demo: NodePort

kubectl create –f nodeport-service.yml
kubectl create –f pod.yml

```
root@k-master:/home/osboxes# kubectl create -f node.yml
service/nodeport-nginx-service created
pod/nginx-frontend created
root@k-master:/home/osboxes#
```



kubectl get services

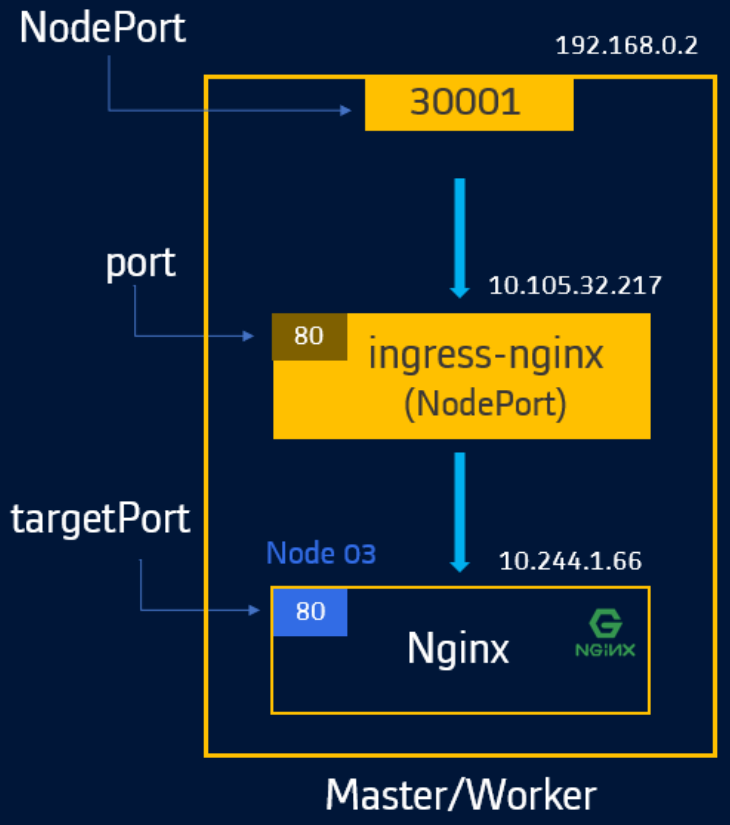
```
root@k-master:/home/osboxes# kubectl get services
NAME                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes              ClusterIP   10.96.0.1       <none>        443/TCP        13m
nodeport-nginx-service  NodePort    10.105.32.217   <none>        80:30001/TCP   2m3s
root@k-master:/home/osboxes# kubectl get pods
NAME             READY   STATUS    RESTARTS   AGE
nginx-frontend   1/1     Running   0          2m10s
root@k-master:/home/osboxes#
```

Kubernetes

Demo: NodePort

kubectl describe service <service-name>

```
root@k-master:/home/osboxes# kubectl describe service nodeport-nginx-service
Name:                     nodeport-nginx-service
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=nginx-frontend
Type:                     NodePort
IP:                       10.105.32.217
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  30001/TCP
Endpoints:                10.244.1.66:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
root@k-master:/home/osboxes#
```



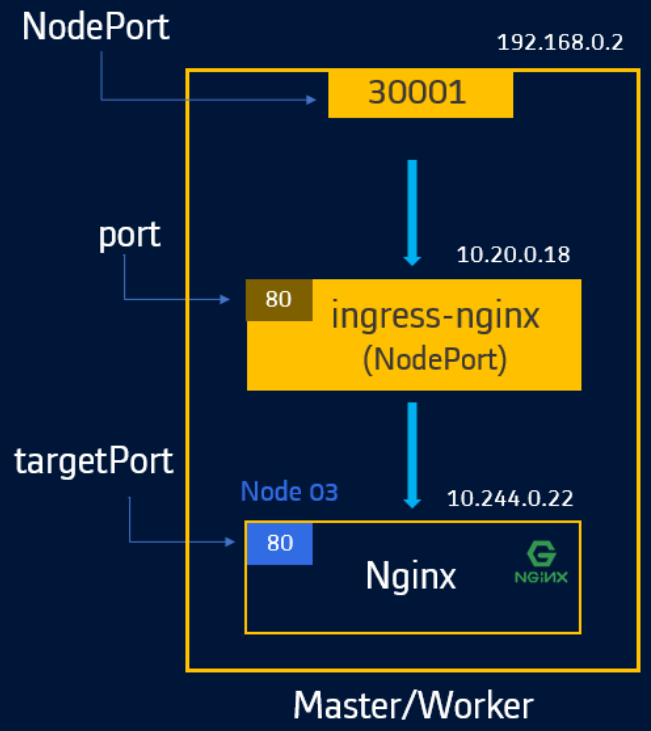
Kubernetes

Demo: NodePort

kubectl get nodes –o wide

```
root@k-master:/home/osboxes# kubectl get nodes -o wide
NAME        STATUS     ROLES    AGE   VERSION   INTERNAL-IP      EXTERNAL-IP
k-master    Ready      master   25h   v1.18.2   192.168.0.107    <none>
k-slave01   Ready      <none>   25h   v1.18.2   192.168.0.104    <none>
k-slave02   NotReady   <none>   25h   v1.18.2   192.168.0.108    <none>
root@k-master:/home/osboxes#
```

192.168.0.107:30001

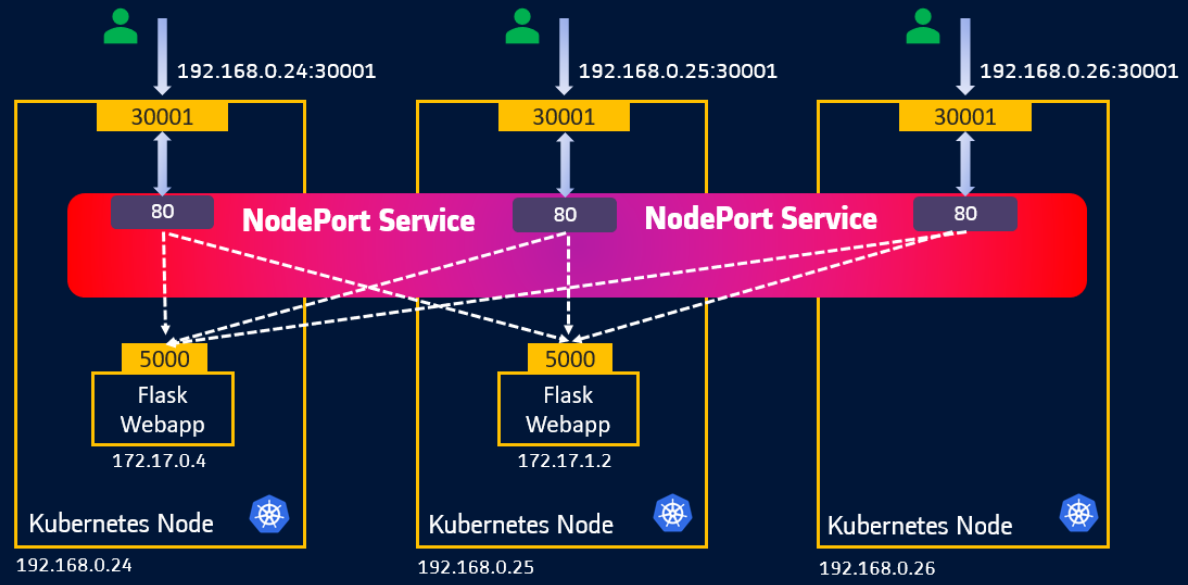


Kubernetes

NodePort Limitations

- In NodePort service, users can access application using the URL http://<node-ip>:<node-port>
- In Production environment, we do not want the users to have to type in the IP address every time to access the application
- So we configure a DNS server to point to the IP of the nodes. Users can now access the application using the URL http://xyz.com:30001
- Now, we don't want the user to have to remember port number either.
- However, NodePort service can only allocate high numbered ports which are greater than 30,000.
- So we deploy a proxy server between the DNS server and the cluster that proxies requests on port 80 to port 30001 on the nodes.
- We then point the DNS to proxy server's IP, and users can now access the application by simply visiting http://xyz.com

```
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 5000
    nodePort: 30001
    protocol: TCP
```



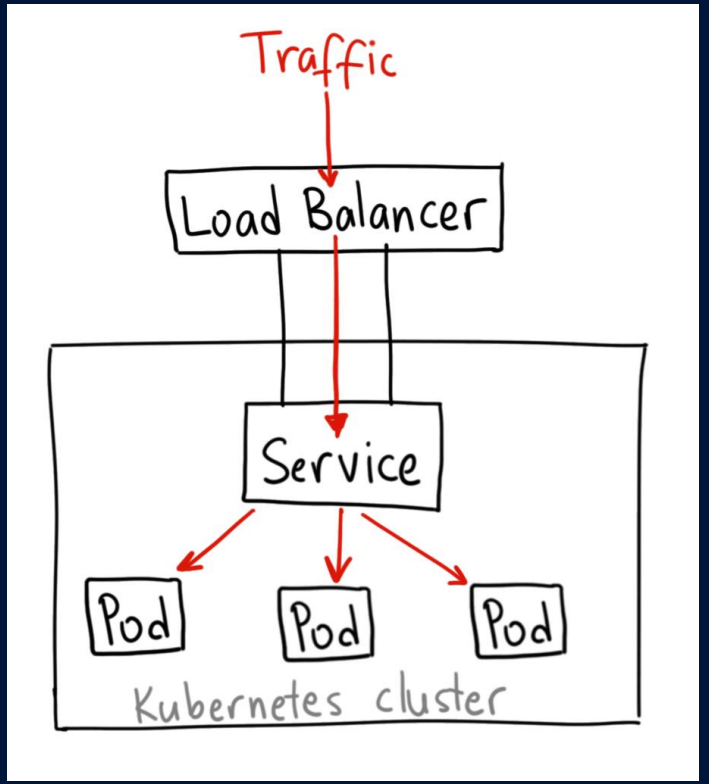
⚠ NodePort 30001 is being used only for demo. You can configure this port number in service manifest file or let K8s auto assign for you.

Kubernetes

Services

Load Balancer

- A LoadBalancer service is the standard way to expose a Kubernetes service to the internet
- On GKE(Google Kubernetes Engine), this will spin up a Network Load Balancer that will give you a single IP address that will forward all external traffic to your service
- All traffic on the port you specify will be forwarded to the service
- There is no filtering, no routing, etc. This means you can send almost any kind of traffic to it, like HTTP, TCP, UDP or WebSocket's
- Few limitations with LoadBalancer:
  - Every service exposed will gets it's own IP address
  - It gets very expensive to have external IP for each of the service(application)



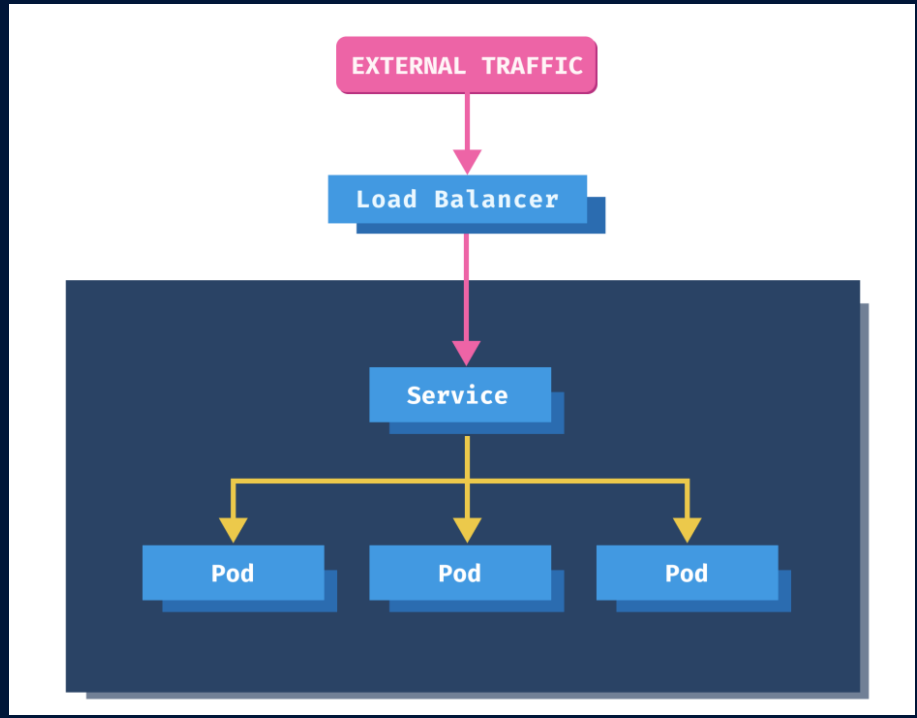
https://rancher.com/blog/2018/2018-06-08-load-balancing-user-apps-with-rancher/

Kubernetes

Services

Load Balancer

- On Google Cloud, AWS, or Azure, a service type of LoadBalancer in the service manifest file will immediately run an Elastic / Cloud Load Balancer that assigns externally IP (public IP) to your application
- But for on-prem or bare-metal k8s clusters, this functionality is not available
- Using service type as LoadBalancer on bare-metal will not assign any external IP and service resource will remain in Pending state forever




```
spec:
  type: LoadBalancer
  selector:
    app: hello
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
```

```
kubectl --kubeconfig=[full path to cluster config file] get services
```

```
NAME                  TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)        AGE
kubernetes            ClusterIP      192.0.2.1      <none>          443/TCP        2h
sample-load-balancer  LoadBalancer   192.0.2.167    <pending>       80:32490/TCP   6s
```

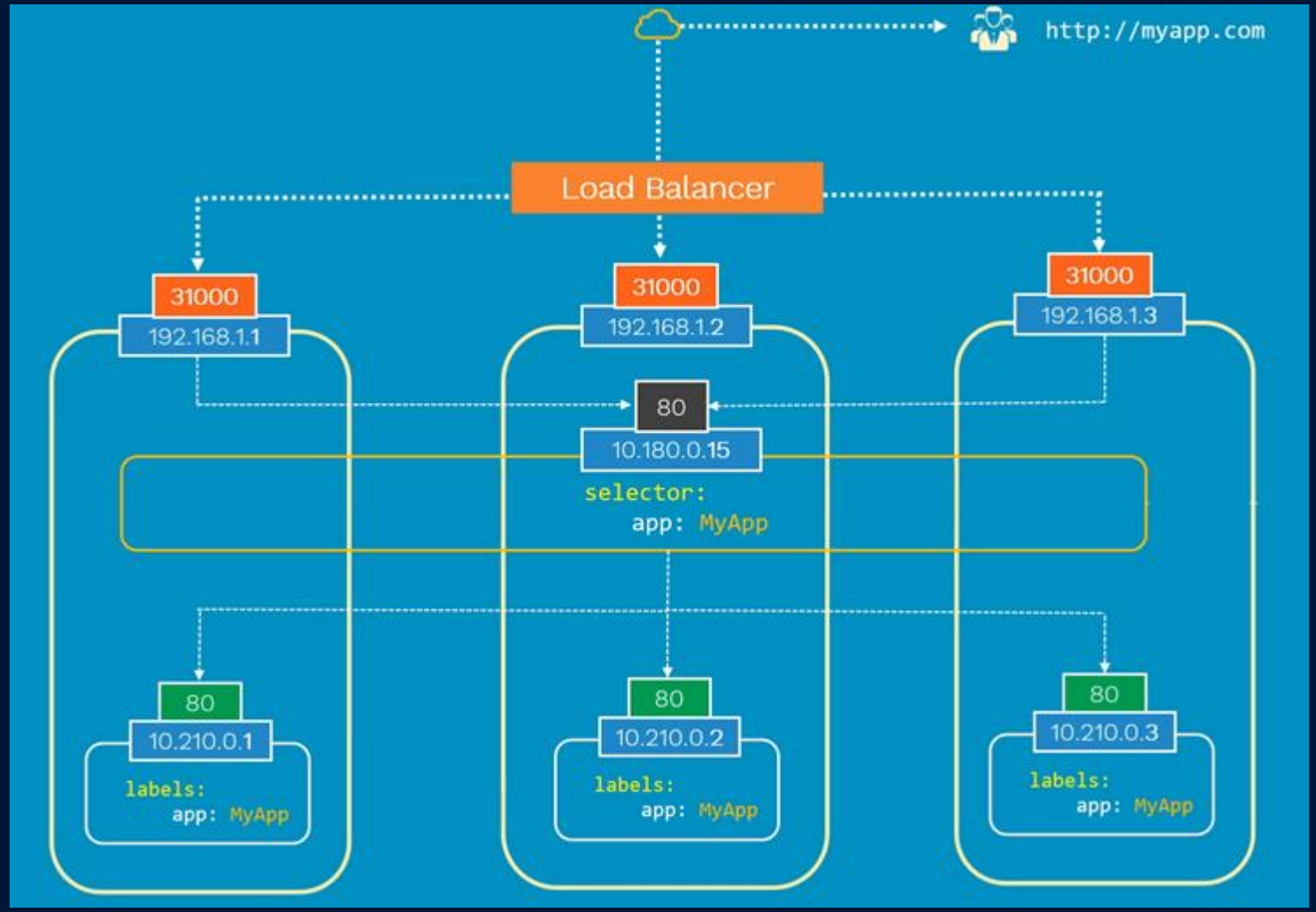
https://collabnix.com/3-node-kubernetes-cluster-on-bare-metal-system-in-5-minutes/

Kubernetes

Services

Load Balancer

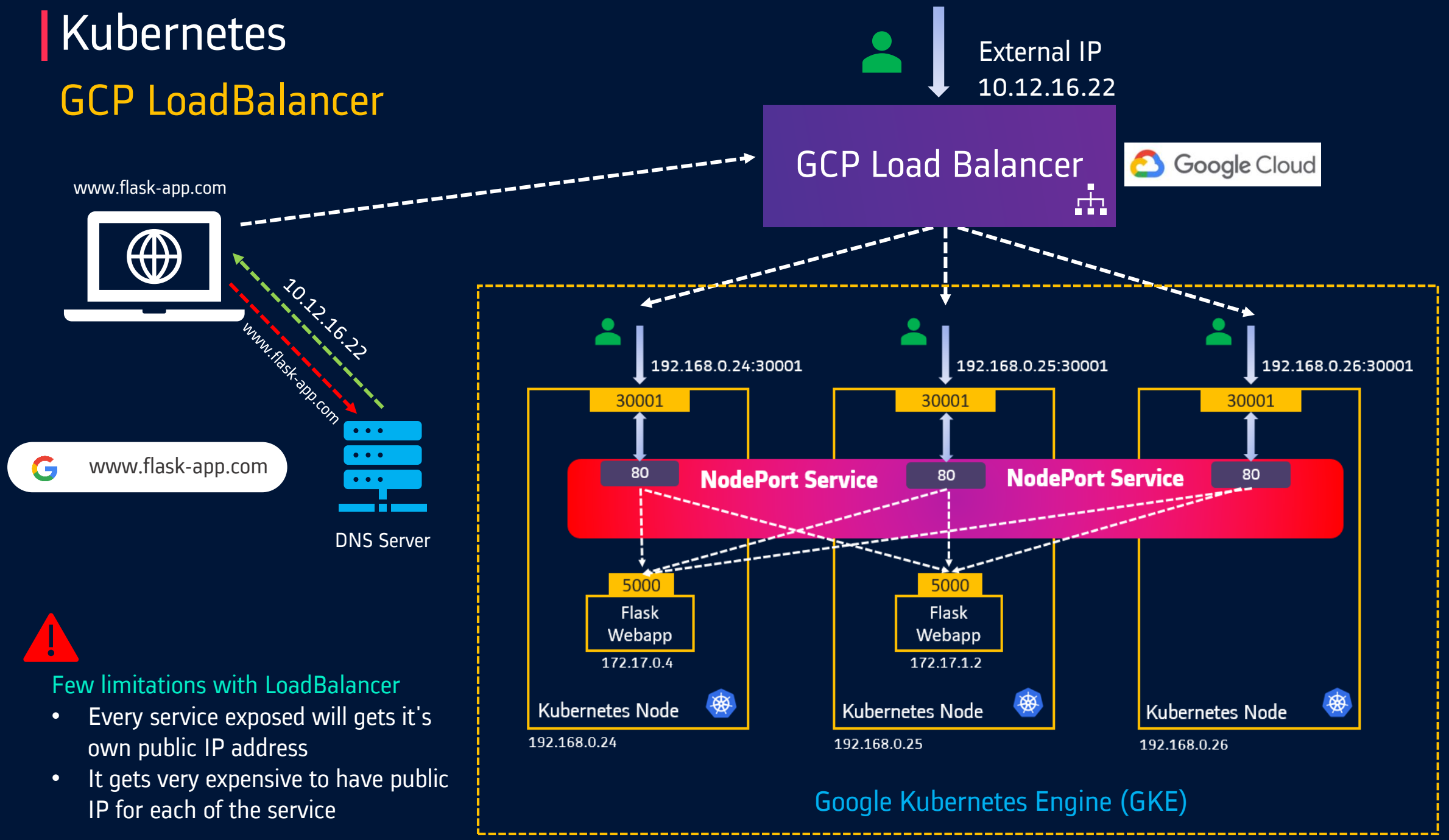
```
apiVersion: v1
kind: Service
metadata:
  name: lb-service
labels:
    app: hello
spec:
  type: LoadBalancer
  selector:
    app: hello
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
```



loadbalancer-service.yml

Kubernetes

GCP LoadBalancer



Few limitations with LoadBalancer

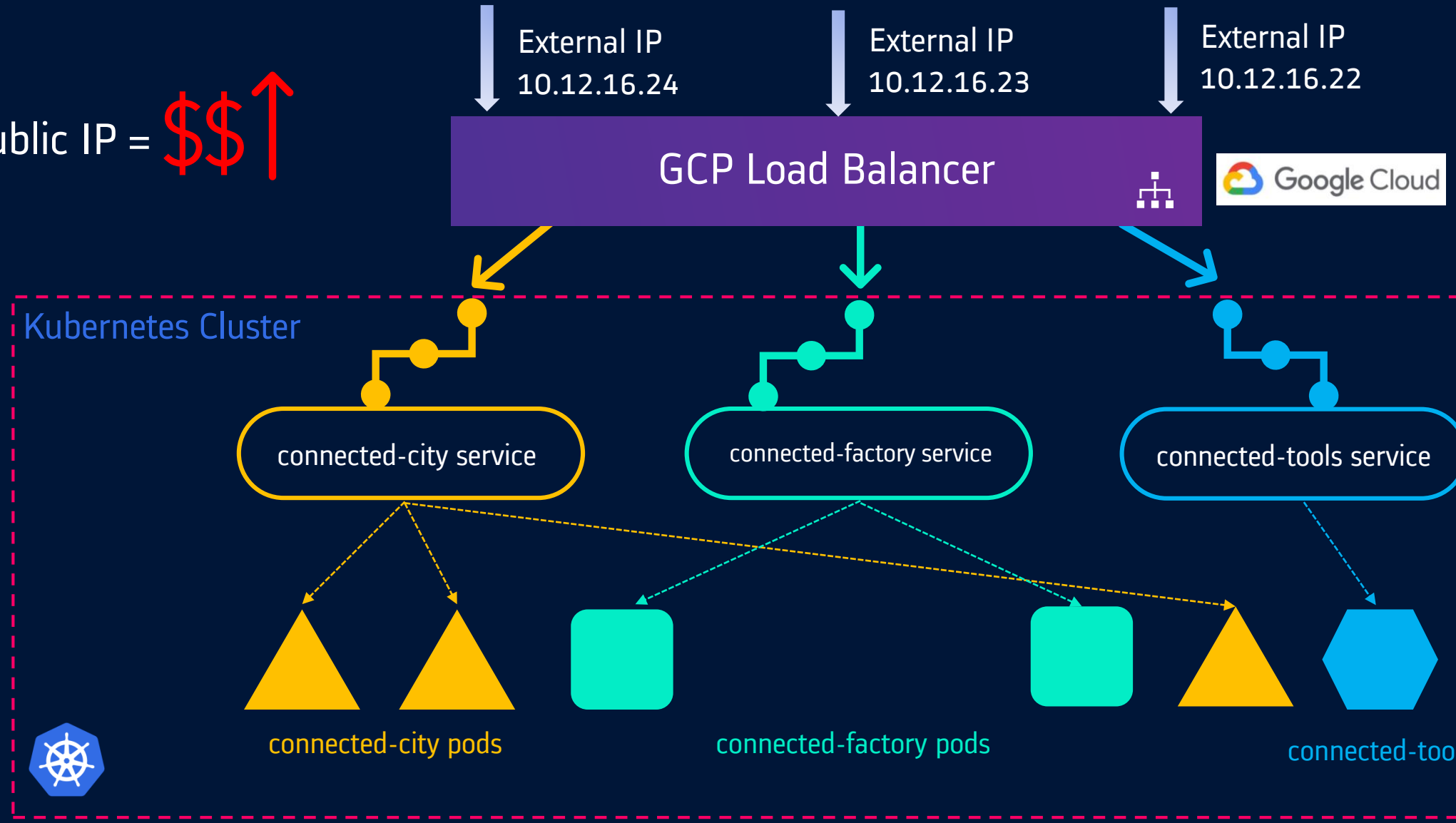
- Every service exposed will gets it's own public IP address
- It gets very expensive to have public IP for each of the service

Kubernetes

GCP LoadBalancer Cons

Public IP = $$↑

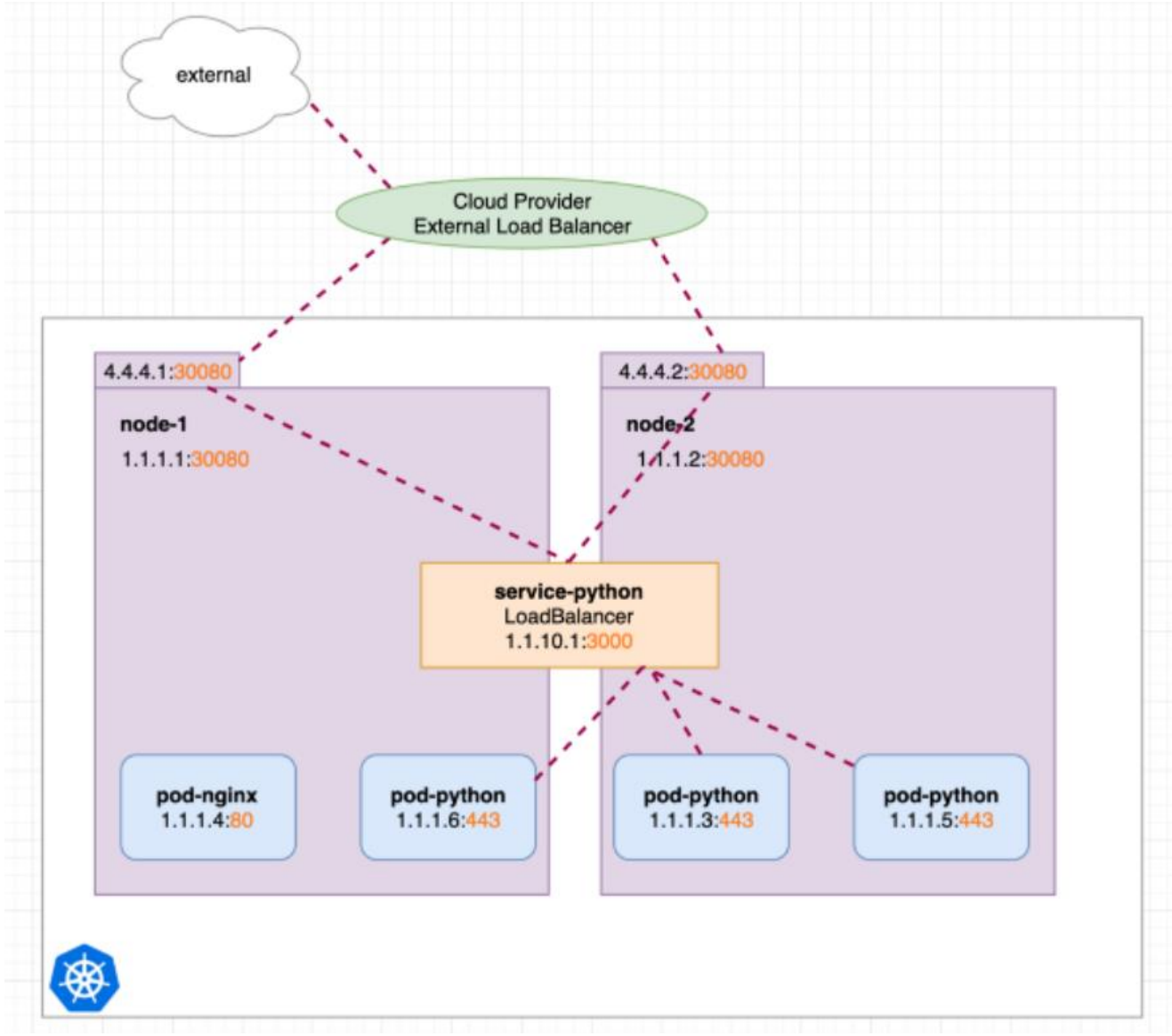
www.connected-city.com
www.connected-factory.com
www.connected-tools.com



Kubernetes

LoadBalancer

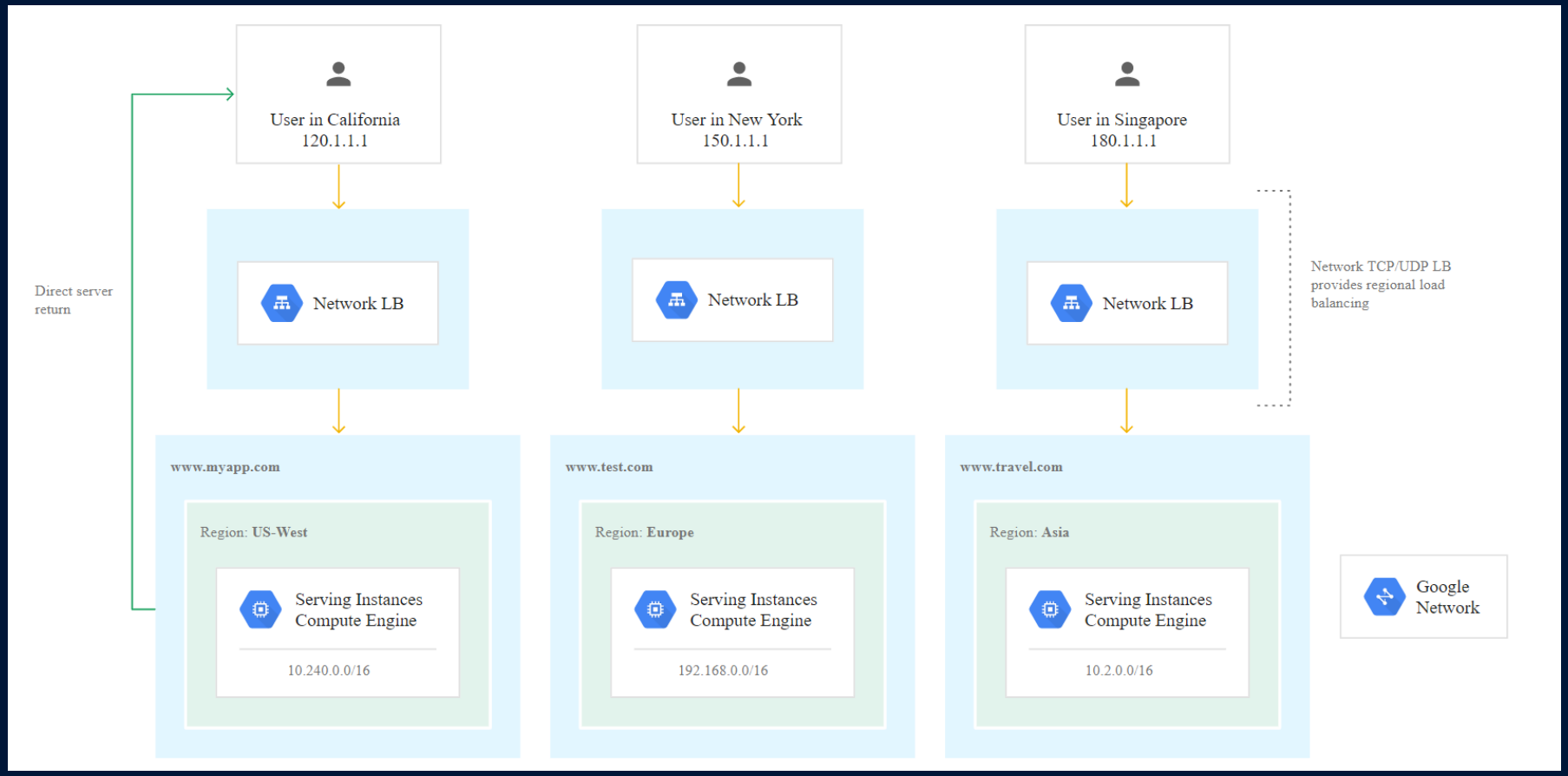
- Application can be reached using the external IP assigned by the LoadBalancer
- The LoadBalancer will forward the traffic to the available nodes in the cluster on the nodePort assigned to the service



Kubernetes

GCP LoadBalancer

Cons



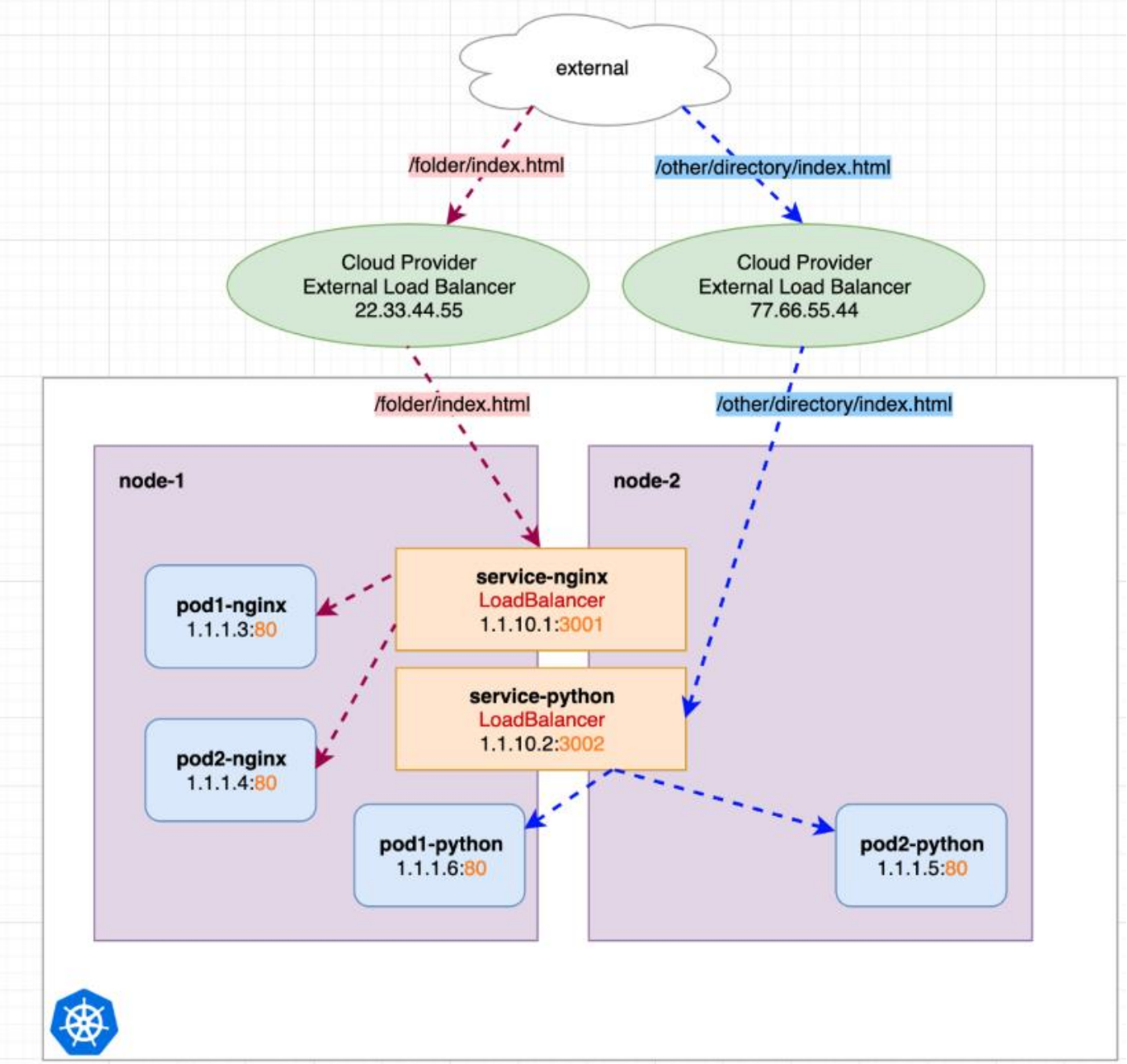
Every service exposed will gets it's own IP address

It gets very expensive to have external IP for each of the service(application)

Kubernetes

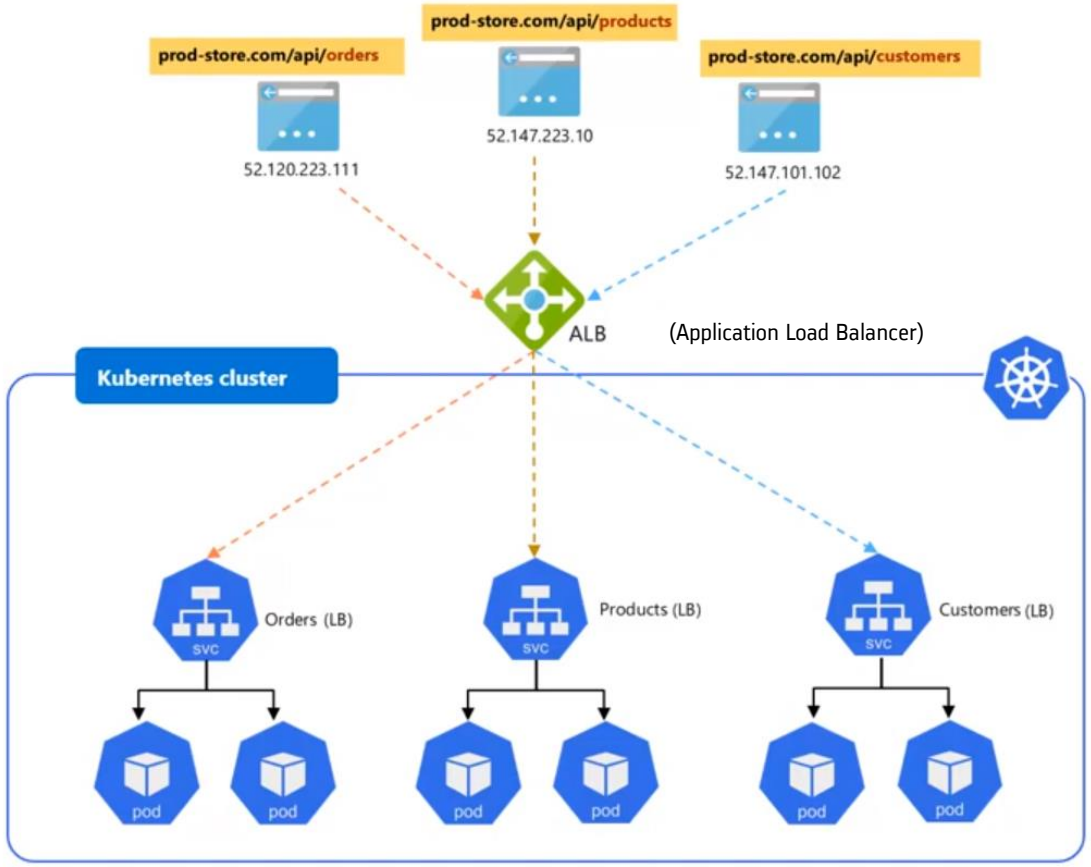
Cloud LoadBalancer: Cons

- Every service exposed will gets it's own IP address
- It gets very expensive to have external IP for each of the service(application)
- We see two LoadBalancers, each having its own IP. If we send a request to LoadBalancer 22.33.44.55 it gets redirected to our internal service-nginx. If we send the request to 77.66.55.44 it gets redirected to our internal service-python.
- This works great! But IP addresses are rare and LoadBalancer pricing depends on the cloud providers. Now imagine we don't have just two but many more internal services for which we would like to create LoadBalancers, costs would scale up.
- Might there be another solution which allows us to only use one LoadBalancer (with one IP) but still reach both of our internal services directly?.

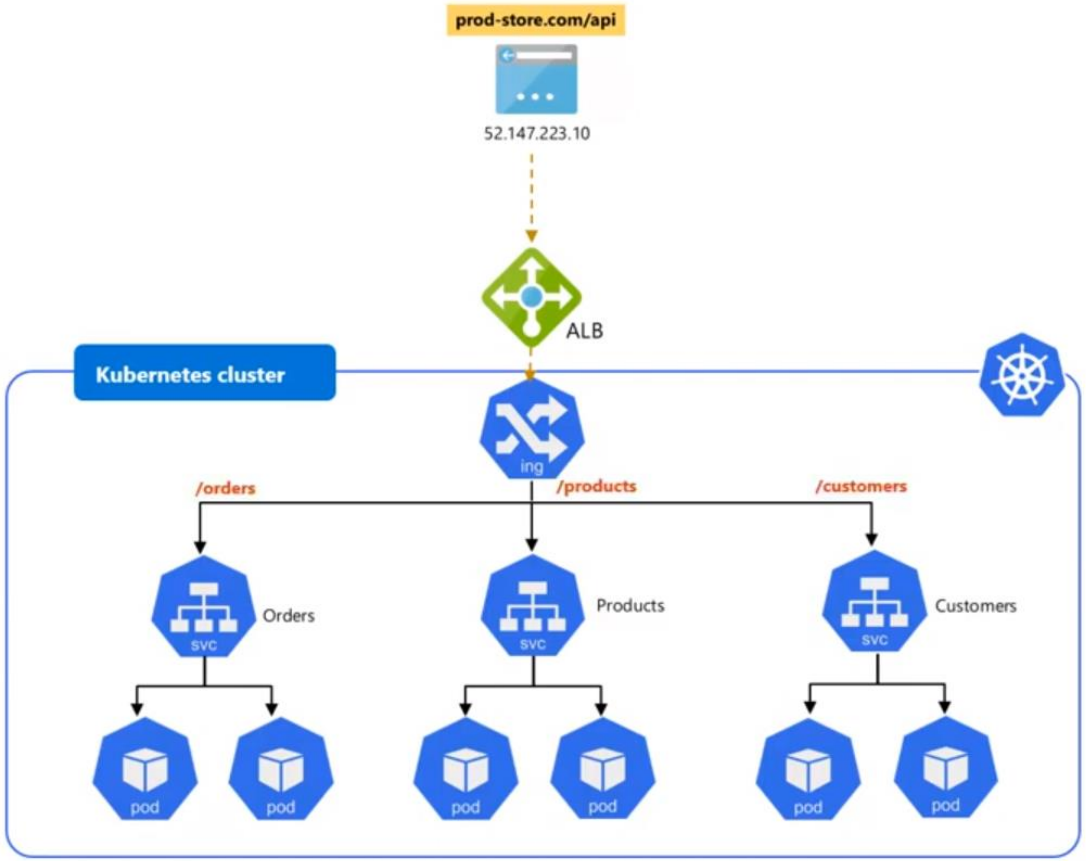


Kubernetes

LoadBalancer Vs Ingress



- Public IPs aren't cheap
- ALB can only handle limited IPs
- So SSL termination



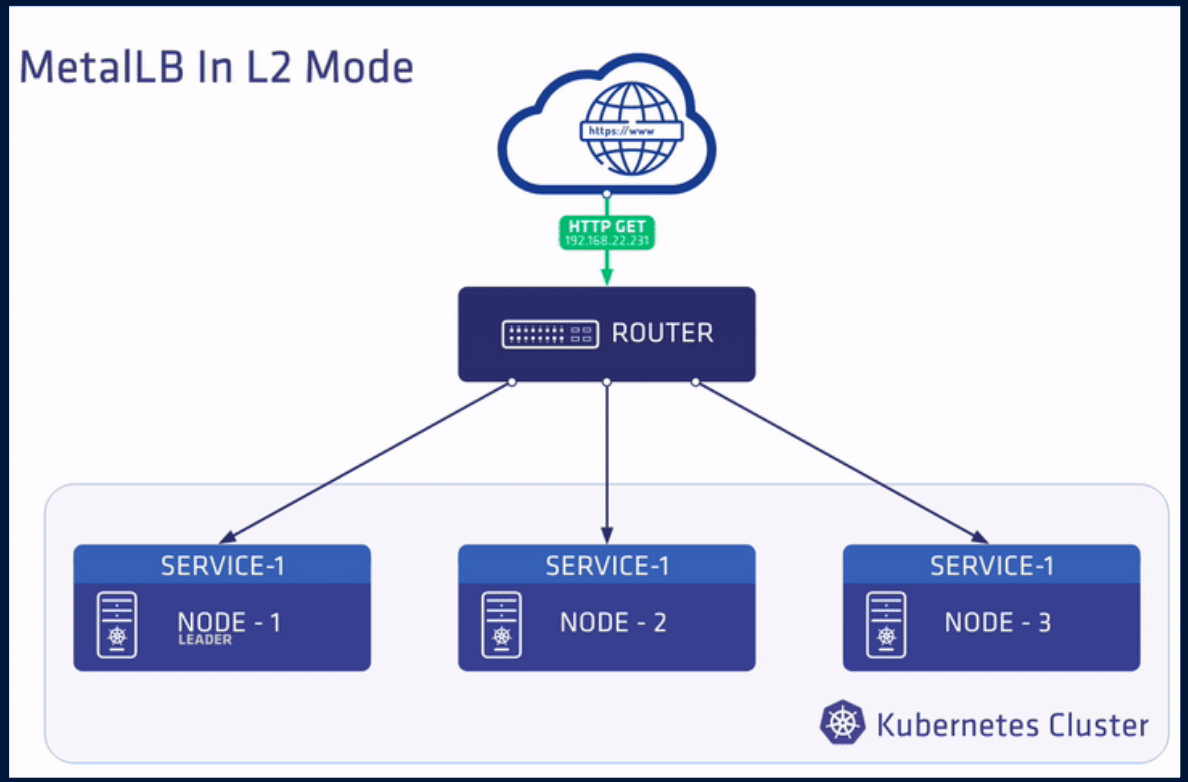
- Ingress acts as internal LoadBalancer
- Routes traffic based on URL path
- All applications will need only one public IP

Kubernetes

Services

MetalLB Load Balancer

- MetalLB is a load-balancer implementation for bare metal Kubernetes clusters.
- It allows you to create Kubernetes services of type “LoadBalancer” in bare-metal/on-prem clusters that don’t run on cloud providers like AWS, GCP, Azure and DigitalOcean.

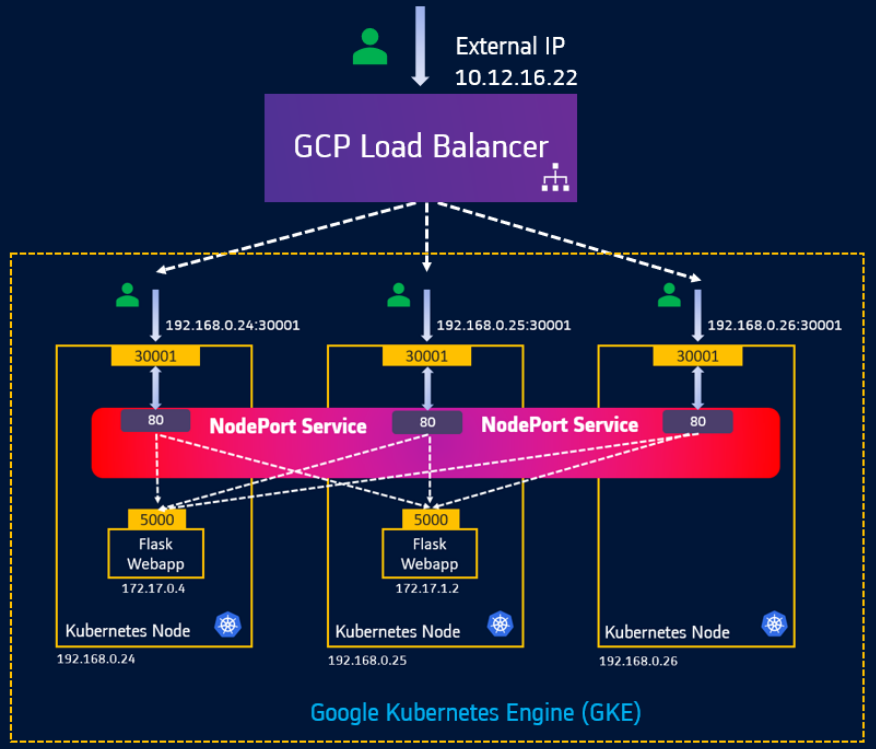


https://metallb.universe.tf/

Kubernetes

Ingress Resource(rules)

- With cloud LoadBalancers, we need to pay for each of the service that is exposed using LoadBalancer as the service type. As services grow in number, complexity to manage SSLs, Scaling, Auth etc., also increase
- Ingress allows us to manage all of the above within the Kubernetes cluster with a definition file, that lives along with the rest of your application deployment files
- Ingress controller can perform load balancing, Auth, SSL and URL/Path based routing configurations by being inside the cluster living as a Deployment or a DaemonSet
- Ingress helps users access the application using a single externally accessible URL, that you can configure to route to different services within your cluster based on the URL path, at the same time terminate SSL/TLS

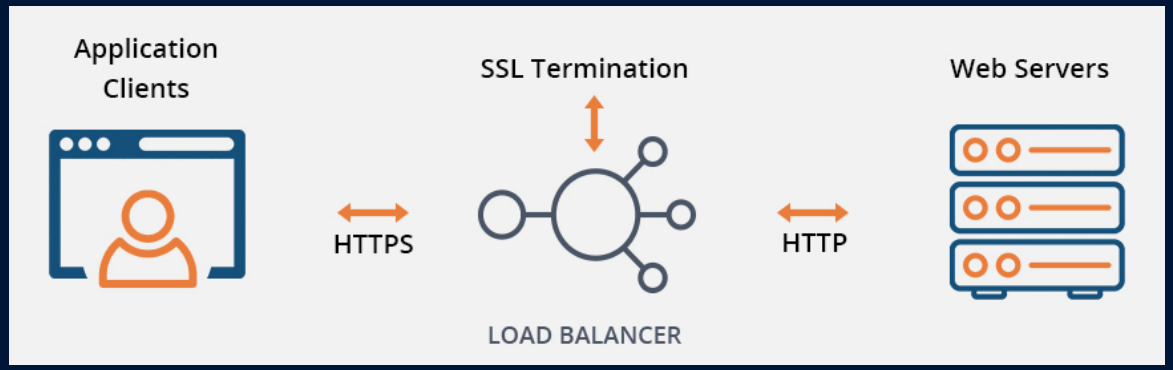


https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0
https://cloud.google.com/kubernetes-engine/docs/concepts/ingress

Kubernetes

Why SSL Termination at LoadBalancer?

- SSL termination/offloading represents the end or termination point of an SSL connection
- SSL termination at LoadBalancer decrypts and verifies data on the load balancer instead of the application server. Unencrypted traffic is sent between the load balancer and the backend servers
- It is desired because decryption is resource and CPU intensive
- Putting the decryption burden on the load balancer enables the server to spend processing power on application tasks, which helps improve performance
- It also simplifies the management of SSL certificates



Kubernetes

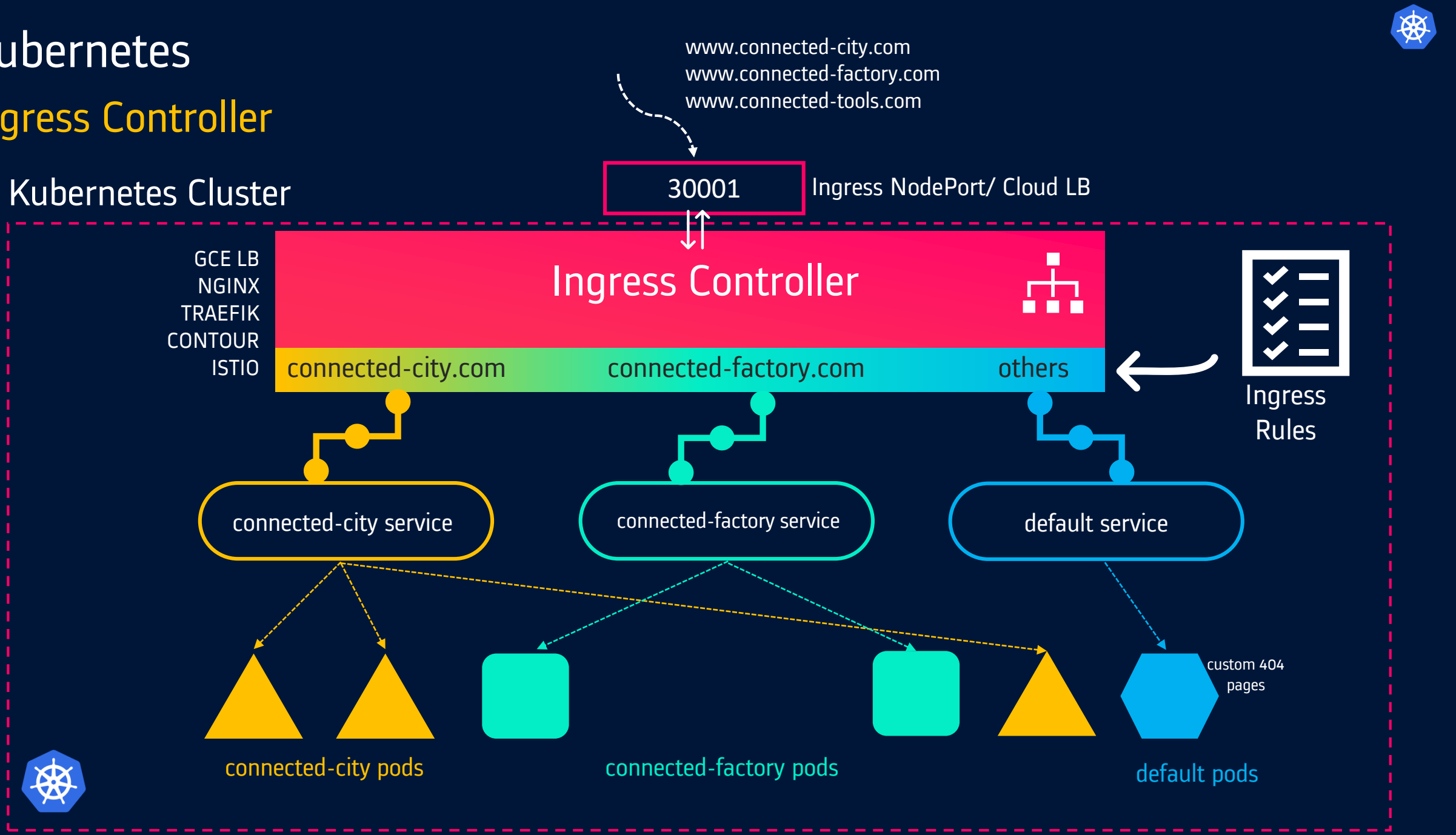
Ingress Controller

- Ingress resources cannot do anything on their own. We need to have an Ingress controller in order for the Ingress resources to work
- Ingress controller implements rules defined by ingress resources
- Ingress controllers doesn't come with standard Kubernetes binary, they have to be deployed separately
- Kubernetes currently supports and maintains GCE and nginx ingress controllers
- Other popular controllers include Traefik, HAProxy ingress, istio, Ambassador etc.,
- Ingress controllers are to be exposed outside the cluster using NodePort or with a Cloud Native LoadBalancer.
- Ingress is the most useful if you want to expose multiple services under the same IP address
- Ingress controller can perform load balancing, Auth, SSL and URL/Path based routing configurations by being inside the cluster living as a Deployment or a DaemonSet

https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

Kubernetes

Ingress Controller



Kubernetes

Nginx Ingress Controller

- Ingress-nginx is an Ingress controller for Kubernetes using NGINX as a reverse proxy and load balancer
- Officially maintained by Kubernetes community
- Routes requests to services based on the request host or path, centralizing a number of services into a single entrypoint.

  Ex: www.mysite.com or www.mysite.com/stats

Deploy Nginx Ingress Controller

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-0.32.0/deploy/static/provider/baremetal/deploy.yaml

https://github.com/kubernetes/ingress-nginx/blob/master/docs/deploy/index.md#bare-metal

Kubernetes

Ingress Rules

ingress-rules.yml

Path based routing

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-rules
spec:
  rules:
  - host:
    http:
      paths:
      - path: /nginx
        backend:
          serviceName: nginx-service
          servicePort: 80
      - path: /flask
        backend:
          serviceName: flask-service
          servicePort: 80
```

Host based routing

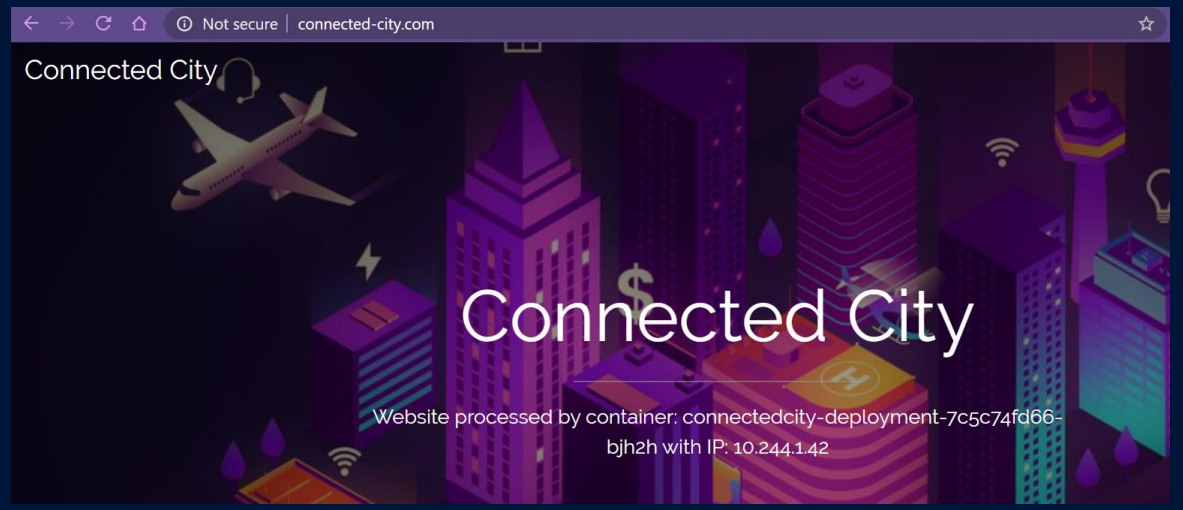
```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-rules
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: nginx-app.com
    http:
      paths:
      - backend:
          serviceName: nginx-service
          servicePort: 80
  - host: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

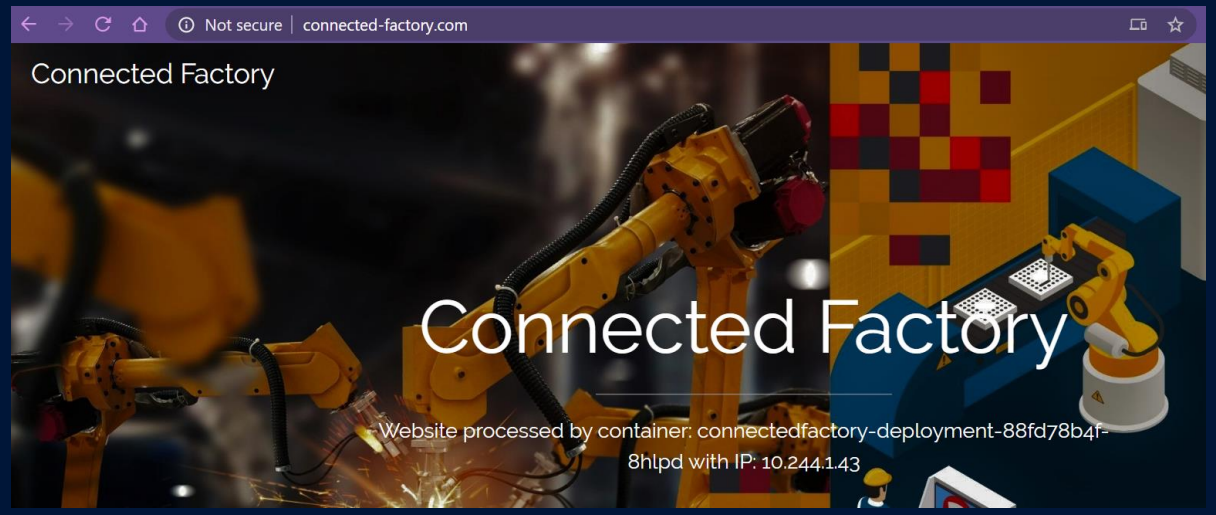
Ingress-controller executes these ingress-rules by comparing with the http requested URL in the http header

Kubernetes

Demo: Ingress

- 3VMs K8s Cluster + 1 VM for Reverse Proxy
- Deploy Ingress controller
- Deploy pods
- Deploy services
- Deploy Ingress rules
- Configure external reverse proxy
- Update DNS names
- Access applications using URLs
  - connected-city.com
  - connected-factory.com

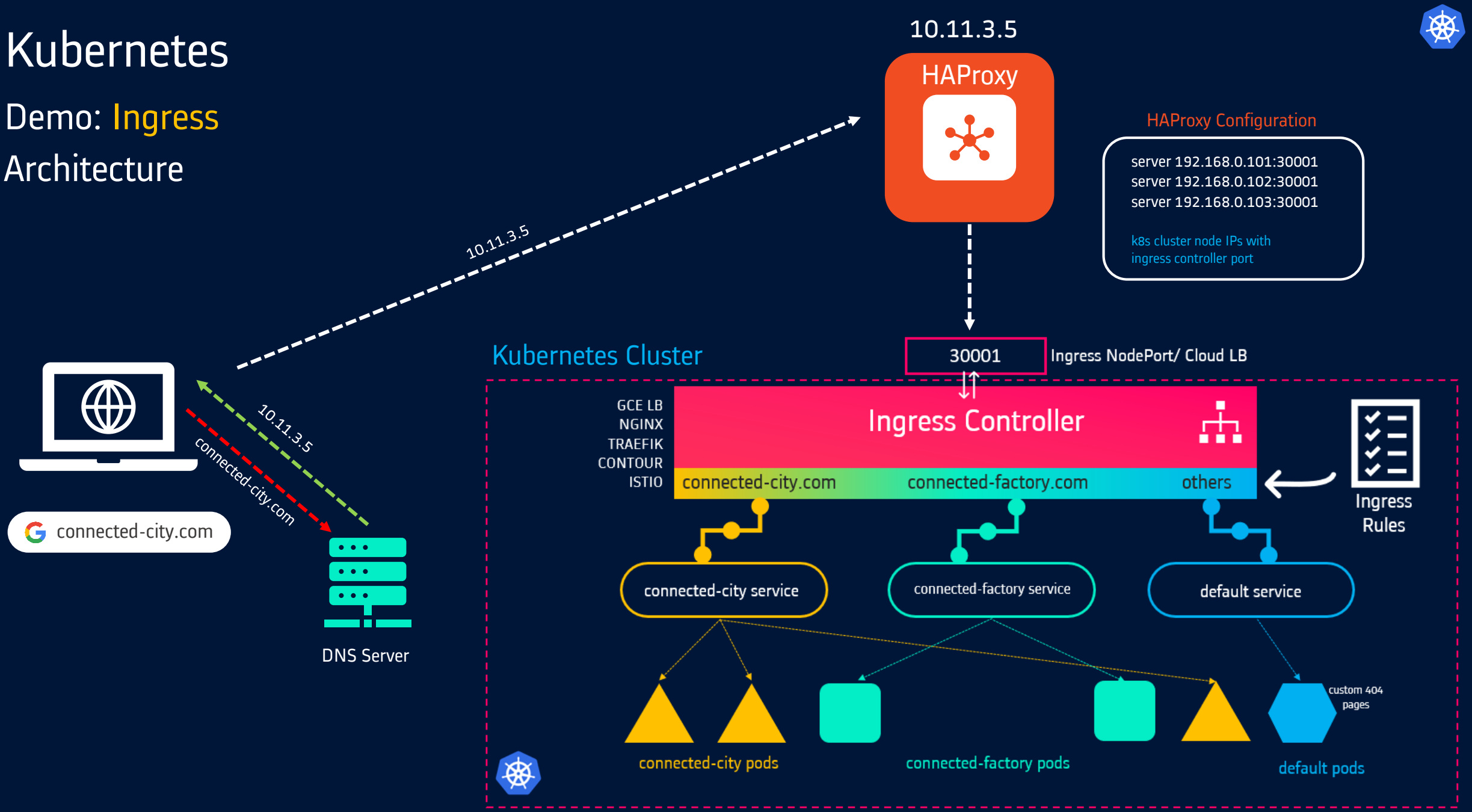




Kubernetes

Demo: Ingress

Architecture



Kubernetes

Demo: Ingress

1. Deploy Nginx Ingress Controller

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-0.32.0/deploy/static/provider/baremetal/deploy.yaml

2. Deploy pods and services

kubectl apply –f <object>.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: connectedcity-service
spec:
  ports:
  - port: 80
    targetPort: 5000
  selector:
    app: connectedcity
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: connectedcity-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: connectedcity
  template:
    metadata:
      labels:
        app: connectedcity
    spec:
      containers:
      - name: connectedcity
        image: kunchalavikram/connectedcity:v1
        ports:
        - containerPort: 5000
```

Application-1
Deployment + ClusterIP service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: connectedfactory-service
spec:
  ports:
  - port: 80
    targetPort: 5000
  selector:
    app: connectedfactory
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: connectedfactory-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: connectedfactory
  template:
    metadata:
      labels:
        app: connectedfactory
    spec:
      containers:
      - name: connectedfactory
        image: kunchalavikram/connectedfactory:v1
        ports:
        - containerPort: 5000
```

Application-2
Deployment + ClusterIP service

Kubernetes

Demo: Ingress

3. Deploy ingress rules manifest file

- Host based routing rules
- Connects to various services depending upon the host parameter

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-rules
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: connected-city.com
    http:
      paths:
      - backend:
          serviceName: connectedcity-service
          servicePort: 80
  - host: connected-factory.com
    http:
      paths:
      - backend:
          serviceName: connectedfactory-service
          servicePort: 80
```

kubectl apply –f <object>.yml

Kubernetes

Demo: Ingress

4. Deploy HA Proxy LoadBalancer

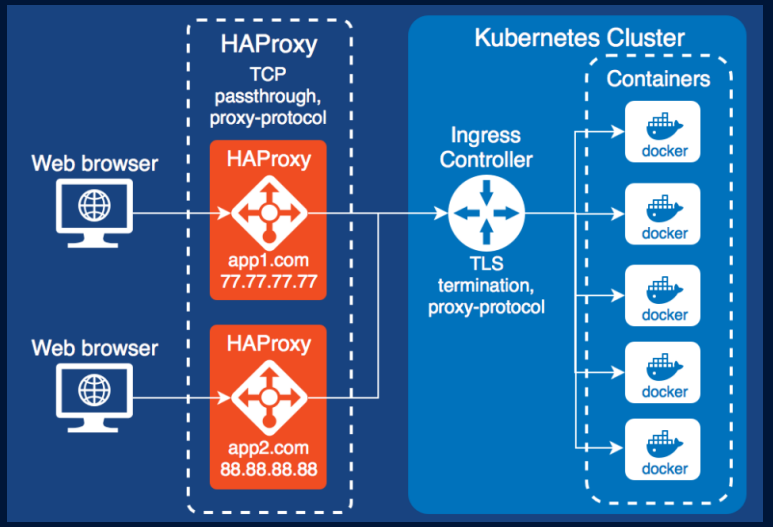
- Provision a VM
- Install HAProxy using package manager
  - apt install haproxy –y
- Restart HAProxy service after modifying the configuration
  - systemctl stop haproxy
  - add configuration to /etc/haproxy/haproxy.cfg
  - systemctl start haproxy && systemctl enable haproxy

10.11.3.5

/etc/haproxy/haproxy.cfg

```
# Configure HAProxy to listen on port 80
frontend http_front
   bind *:80
   default_backend http_back

# Configure HAProxy to route requests to swarm nodes on port 8080
backend http_back
   balance roundrobin
   mode http
   server srv1 192.168.0.101:32174
   server srv2 192.168.0.102:32174
   server srv3 192.168.0.103:32174
root@proxyserver:/home/osboxes#
```



Kubernetes

Demo: Ingress

5. Update dummy DNS entries

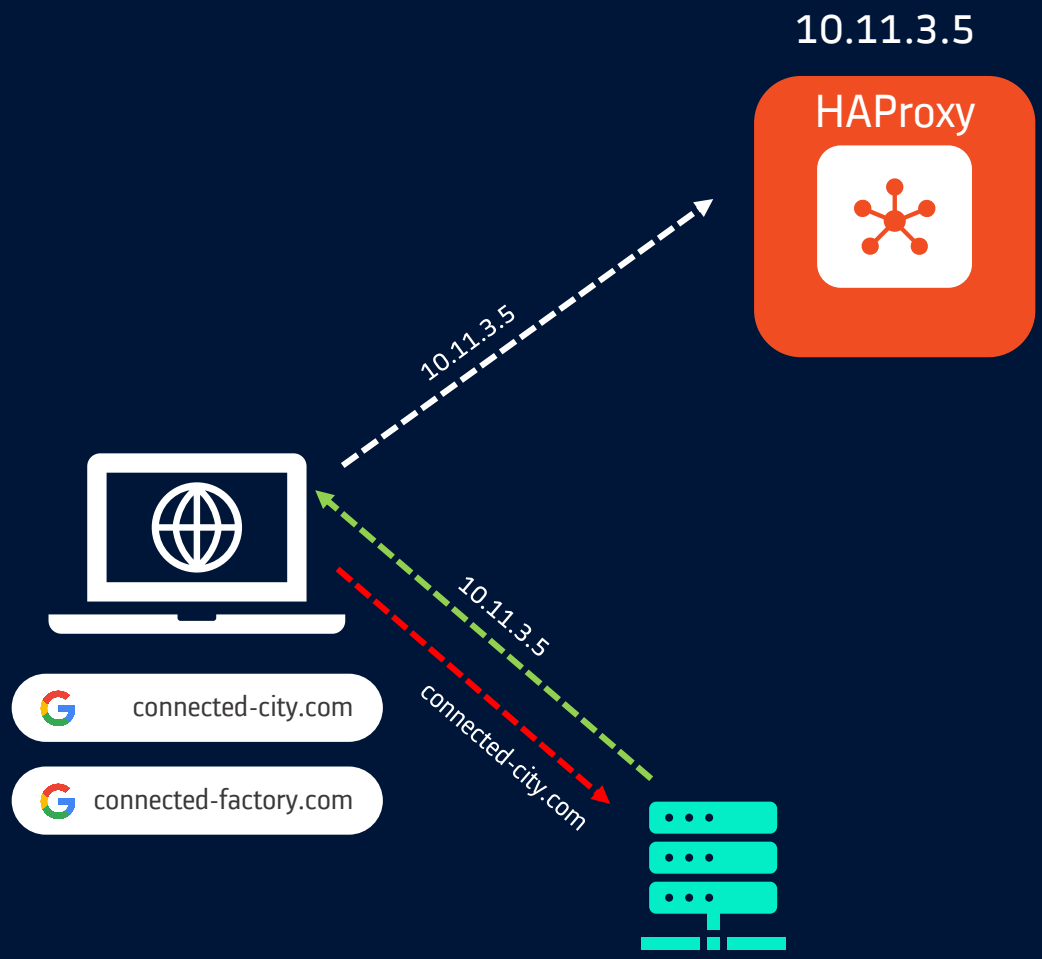
Both DNS names to point to IP of HAProxy server

windows

```
C:\Windows\System32\drivers\etc\hosts
192.168.0.105 connected-city.com
192.168.0.105 connected-factory.com
ipconfig /flushdns
```

linux

```
/etc/hosts
192.168.0.105 flask-app.com
```

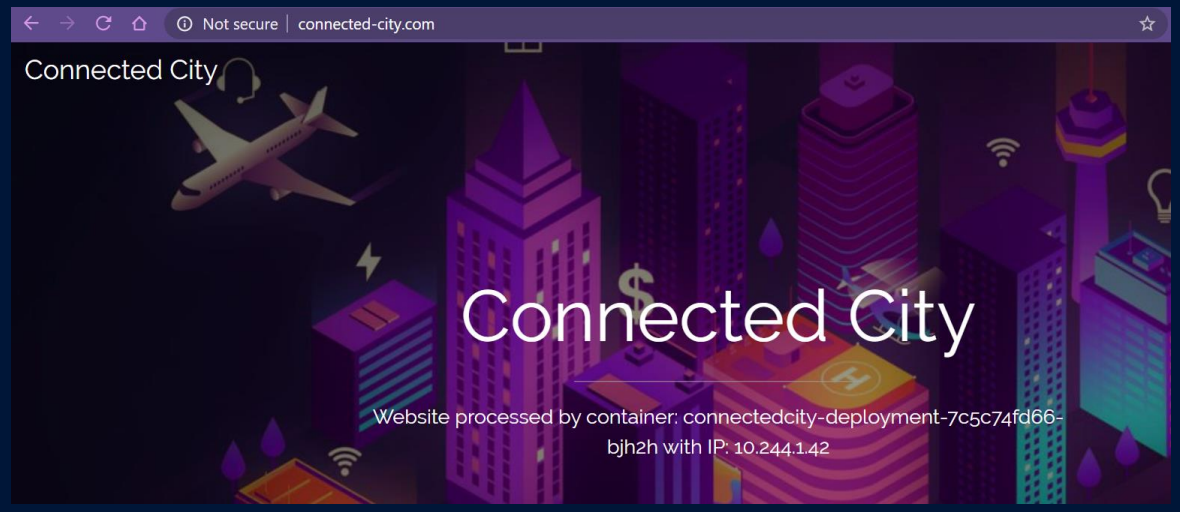


Kubernetes

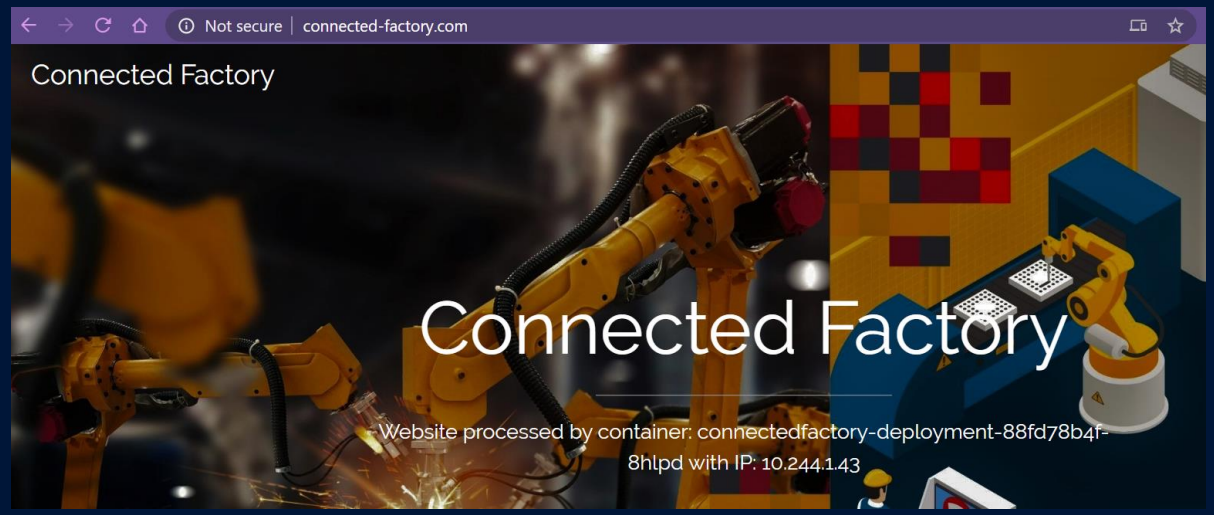
Demo: Ingress

6. Access Application through URLs

connected-city.com

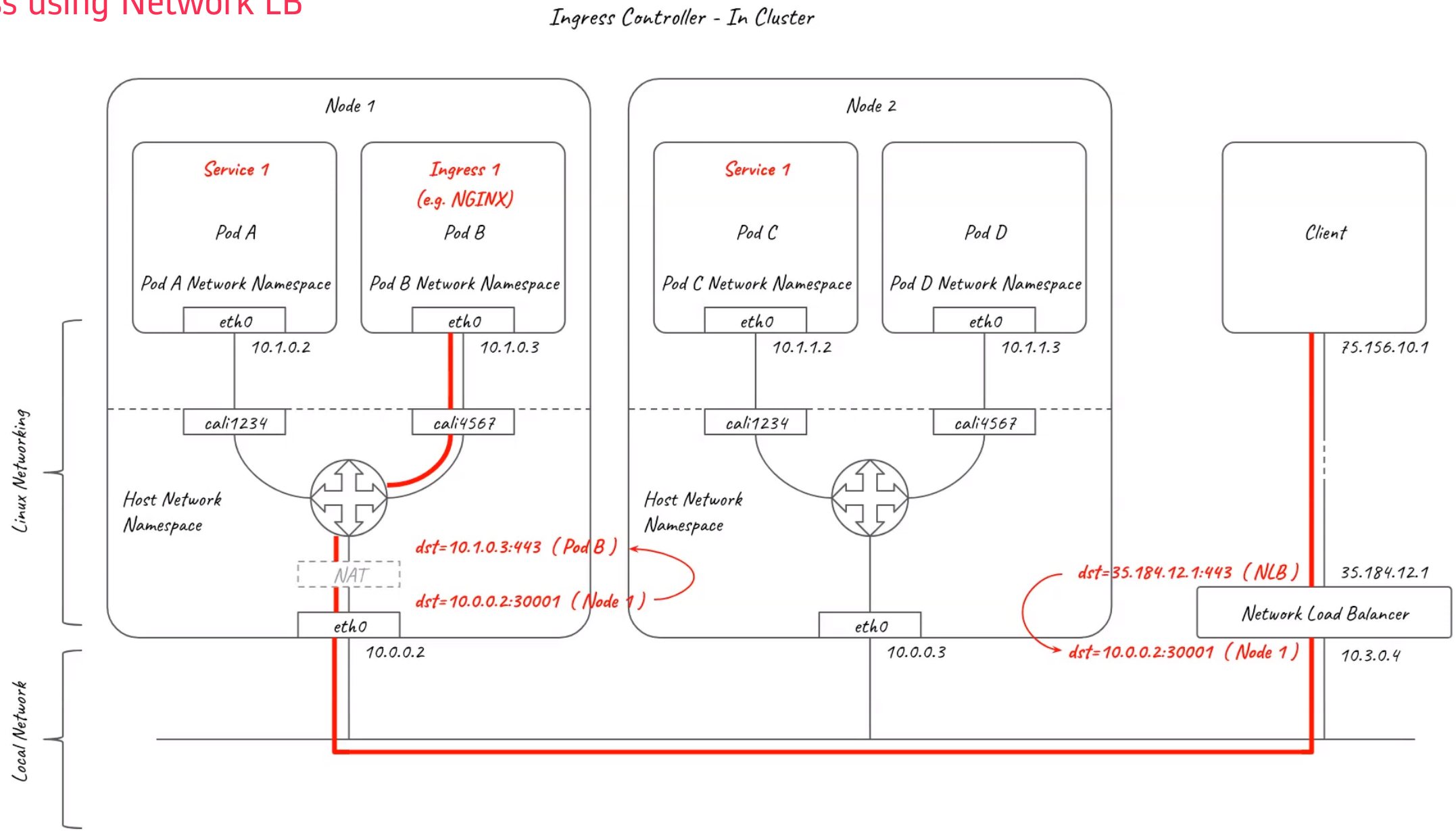


connected-factory.com



Kubernetes

Ingress using Network LB

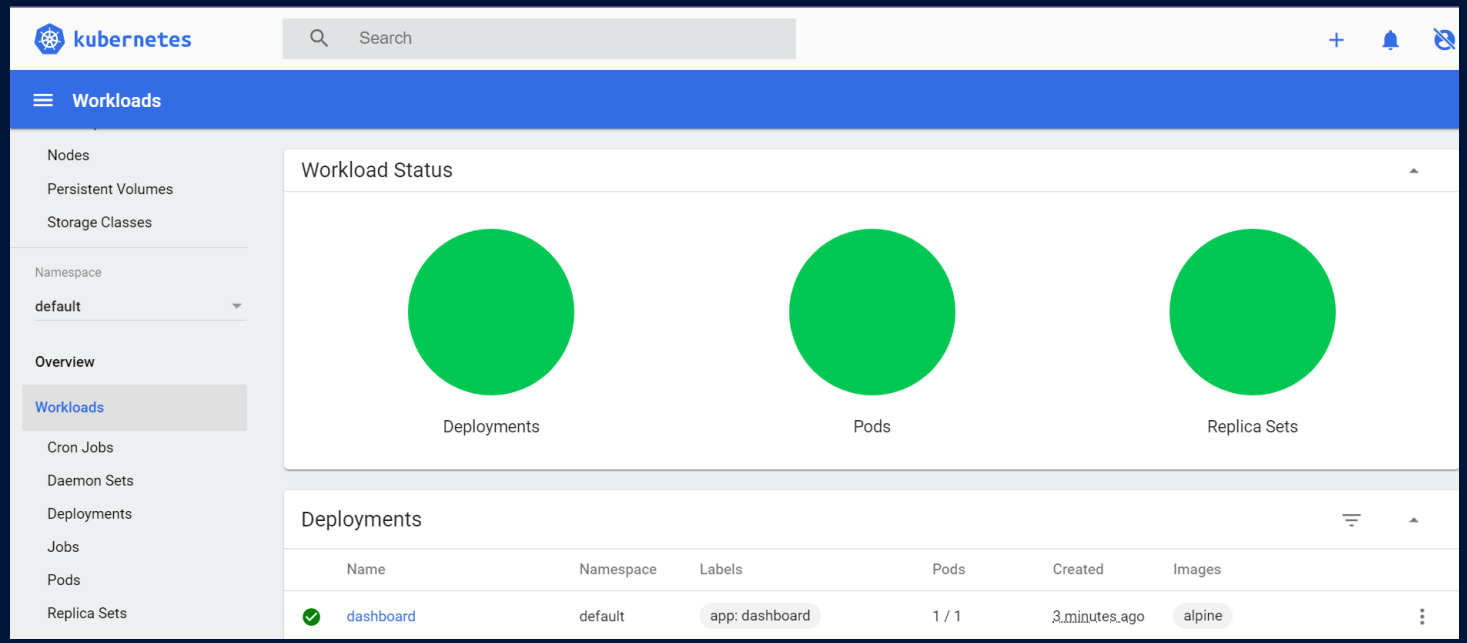


Kubernetes

Dashboard

- Default login access to dashboard is by using token or kubeconfig file. This can be bypassed for internal testing but not recommended in production
- Uses NodePort to expose the dashboard outside the Kubernetes cluster
- Change the service to ClusterIP and use it in conjunction with ingress resources to make it accessible through a DNS name(similar to previous demos)

https://github.com/kunchalavikram1427/kubernetes/blob/master/dashboard/insecure-dashboard-nodeport.yaml



https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/
https://devblogs.microsoft.com/premier-developer/bypassing-authentication-for-the-local-kubernetes-cluster-dashboard/

Kubernetes

Volumes

- By default, container data is stored inside own its file system
- Containers are ephemeral in nature. When they are destroyed, the data inside them gets deleted
- Also when running multiple containers in a Pod it is often necessary to share files between those Containers
- In order to persist data beyond the lifecycle of pod, Kubernetes provide volumes
- A volume can be thought of as a directory which is accessible to the containers in a pod
- The medium backing a volume and its contents are determined by the volume type

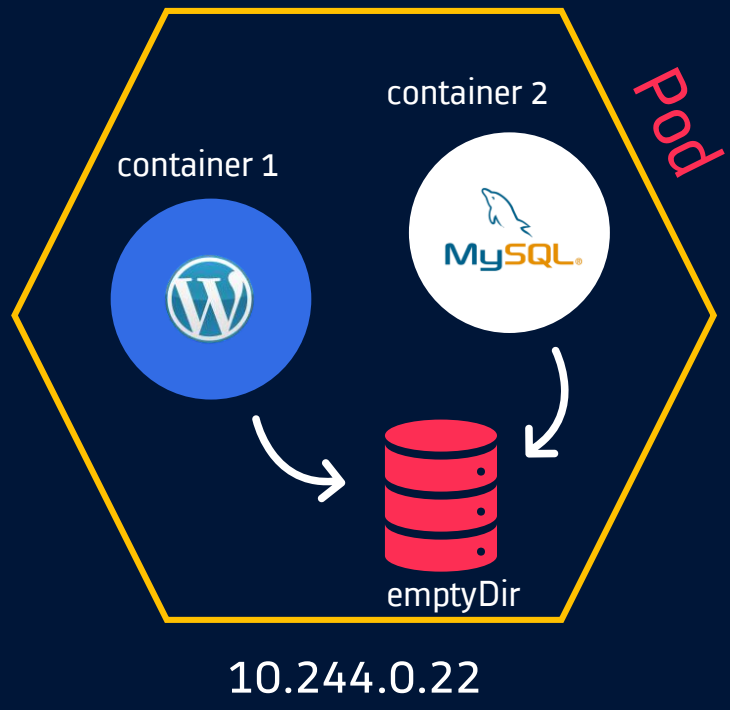
Types of Kubernetes Volumes

- There are different types of volumes you can use in a Kubernetes pod:
  - ❑ Node-local memory (emptyDir and hostPath)
  - ❑ Cloud volumes (e.g., awsElasticBlockStore, gcePersistentDisk, and azureDiskVolume)
  - ❑ File-sharing volumes, such as Network File System (NFS)
  - ❑ Distributed-file systems (e.g., CephFS and GlusterFS)
  - ❑ Special volume types such as PersistentVolumeClaim, secret, configmap and gitRepo

Kubernetes

emptyDir

- emptyDir volume is first created when a Pod is assigned to a Node
- It is initially empty and has same lifetime of a pod
- emptyDir volumes are stored on whatever medium is backing the node - that might be disk or SSD or network storage or RAM
- Containers in the Pod can all read and write the same files in the emptyDir volume
- This volume can be mounted at the same or different paths in each Container
- When a Pod is removed from a node for any reason, the data in the emptyDir is deleted forever
- Mainly used to store cache or temporary data to be processed



Kubernetes

emptyDir

kubectl apply -f emptyDir-demo.yml

kubectl exec -it pod/emptydir-pod -c container-2 -- cat /cache/date.txt
kubectl logs pod/emptydir-pod -c container-2

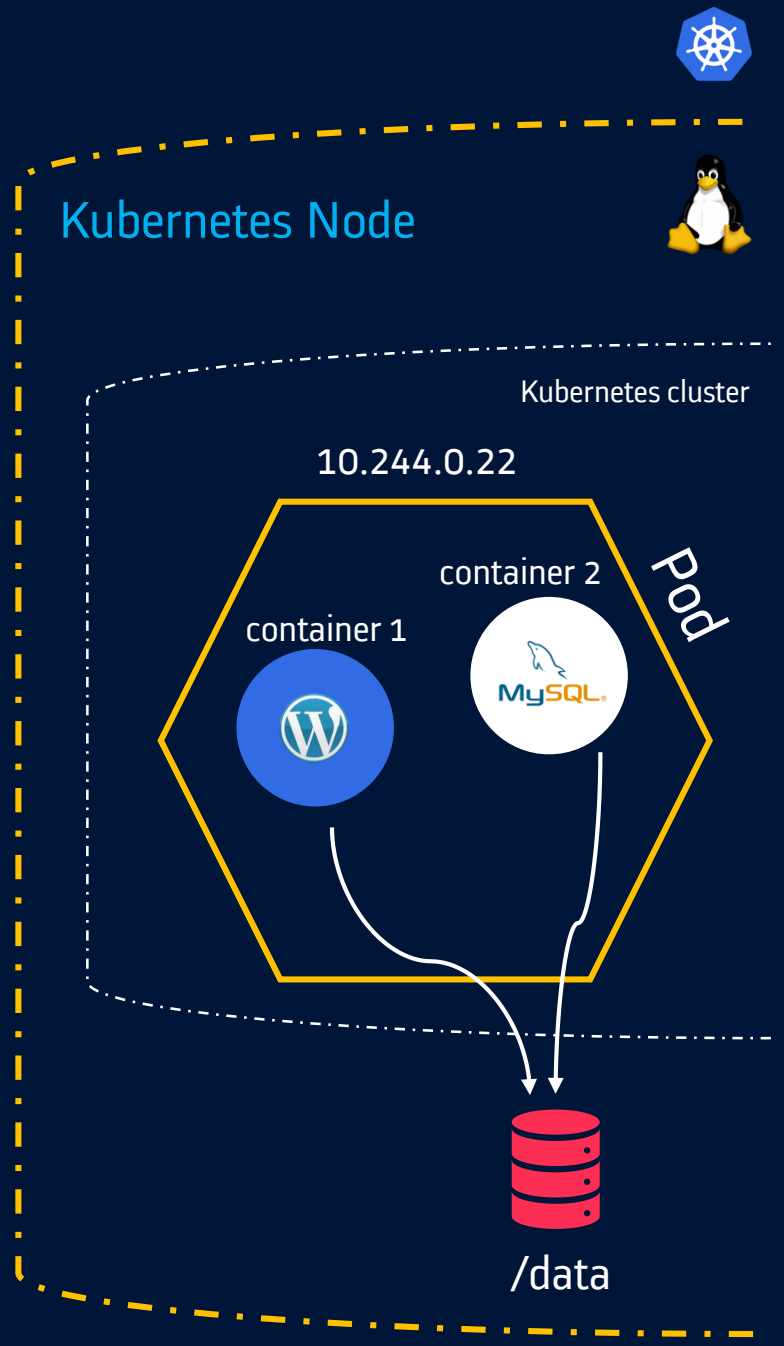
```
root@k8s-master:/home/osboxes/demo_kubernetes# k exec -it  pod/emptydir-pod -c container-2 -- cat /cache/date.txt
Tue Jun  2 08:10:17 UTC 2020
root@k8s-master:/home/osboxes/demo_kubernetes# k logs pod/emptydir-pod -c container-2
Tue Jun  2 08:10:17 UTC 2020
root@k8s-master:/home/osboxes/demo_kubernetes#
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-pod
  labels:
    app: busybox
    purpose: emptydir-demo
spec:
  volumes:
  - name: cache-volume
    emptyDir: {}
  containers:
  - name: container-1
    image: busybox
    command: ["/bin/sh","-c"]
    args: ["date >> /cache/date.txt; sleep 1000"]
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  - name: container-2
    image: busybox
    command: ["/bin/sh","-c"]
    args: ["cat /cache/date.txt; sleep 1000"]
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
```

Kubernetes

hostPath

- This type of volume mounts a file or directory from the host node's filesystem into your pod
- hostPath directory refers to directory created on Node where pod is running
- Use it with caution because when pods are scheduled on multiple nodes, each nodes get its own hostPath storage volume. These may not be in sync with each other and different pods might be using a different data
- Let's say the pod with hostPath configuration is deployed on Worker node 2. Then host refers to worker node 2. So any hostPath location mentioned in manifest file refers to worker node 2 only
- When node becomes unstable, the pods might fail to access the hostPath directory and eventually gets terminated



https://kubernetes.io/docs/concepts/storage/volumes/#hostpath

Kubernetes

hostPath

kubectl apply -f hostPath-demo.yml

kubectl logs pod/hostpath-pod -c container-1
kubectl exec -it pod/hostpath-pod -c container-1 -- ls /cache

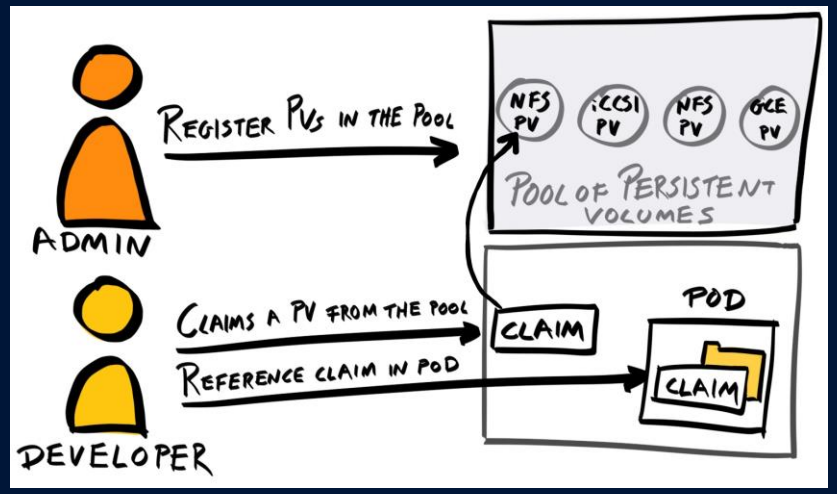
```
root@k8s-master:/home/osboxes/demo_kubernetes# kubectl logs pod/hostpath-pod -c container-1
1.txt
2.txt
3.txt
root@k8s-master:/home/osboxes/demo_kubernetes# kubectl exec -it  pod/hostpath-pod -c container-1 -- ls /cache
1.txt  2.txt  3.txt
root@k8s-master:/home/osboxes/demo_kubernetes#
```

```
apiVersion: v1
kind: Pod
metadata:
  name: hostpath-pod
spec:
  volumes:
  - name: hostpath-volume
    hostPath:
      path: /data
      type: DirectoryOrCreate
  containers:
  - name: container-1
    image: busybox
    command: ["/bin/sh","-c"]
    args: ["ls /cache ; sleep 1000"]
    volumeMounts:
    - mountPath: /cache
      name: hostpath-volume
```

Kubernetes

Persistent Volume and Persistent Volume Claim

- Managing storage is a distinct problem inside a cluster. You cannot rely on emptyDir or hostPath for persistent data.
- Also providing a cloud volume like EBS, AzureDisk often tends to be complex because of complex configuration options to be followed for each service provider
- To overcome this, PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. To do this, K8s offers two API resources: PersistentVolume and PersistentVolumeClaim.

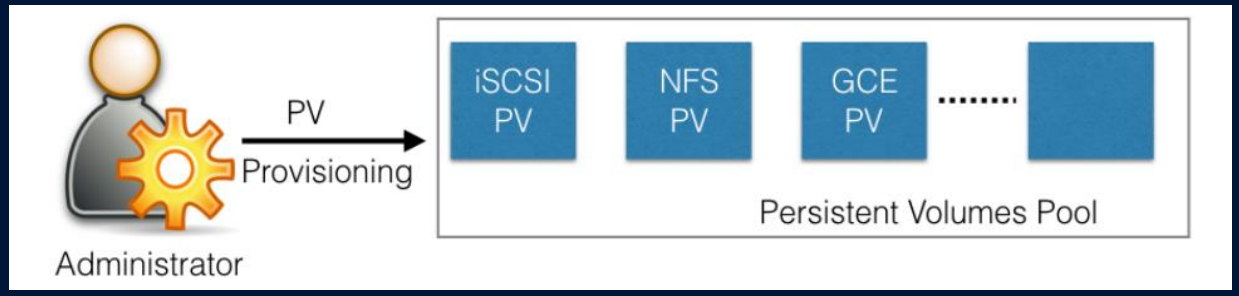


Kubernetes

Persistent Volume and Persistent Volume Claim

Persistent volume (PV)

- A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes(pre-defined provisioners and parameters to create a Persistent Volume)
- Admin creates a pool of PVs for the users to choose from
- It is a cluster-wide resource used to store/persist data beyond the lifetime of a pod
- PV is not backed by locally-attached storage on a worker node but by networked storage system such as Cloud providers storage or NFS or a distributed filesystem like Ceph or GlusterFS
- Persistent Volumes provide a file system that can be mounted to the cluster, without being associated with any particular node

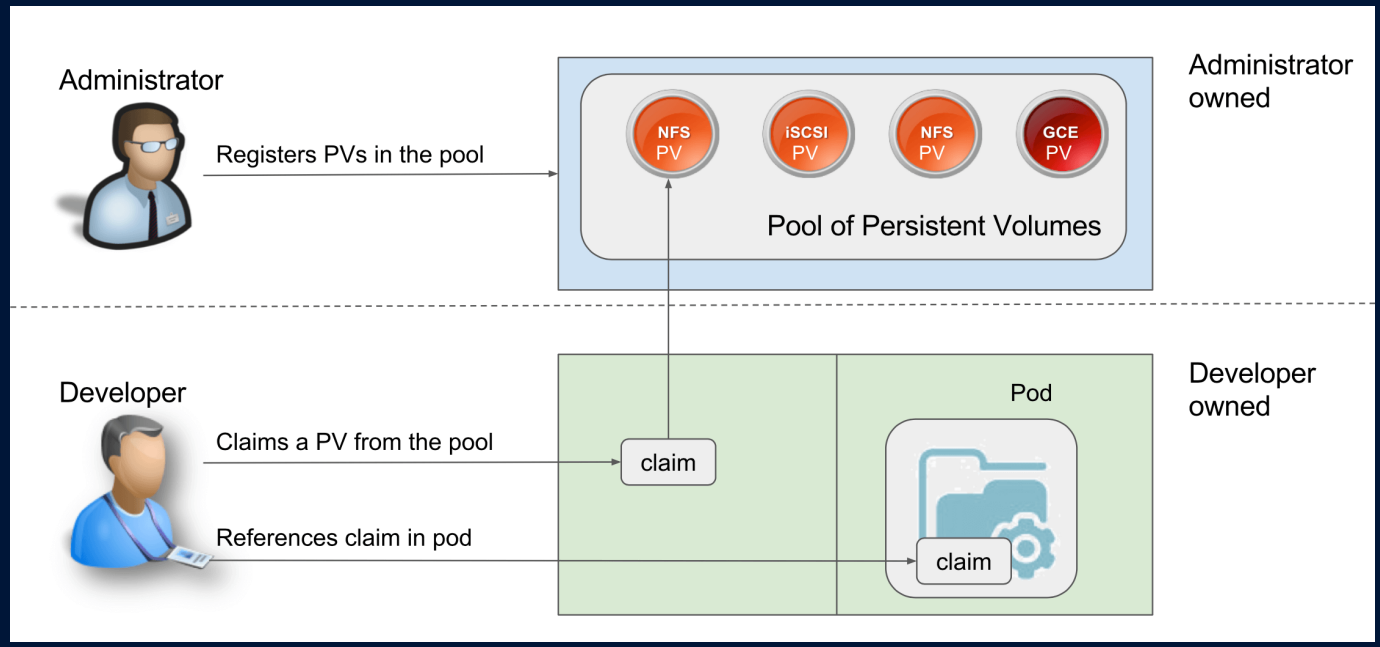


Kubernetes

Persistent Volume and Persistent Volume Claim

Persistent Volume Claim (PVC)

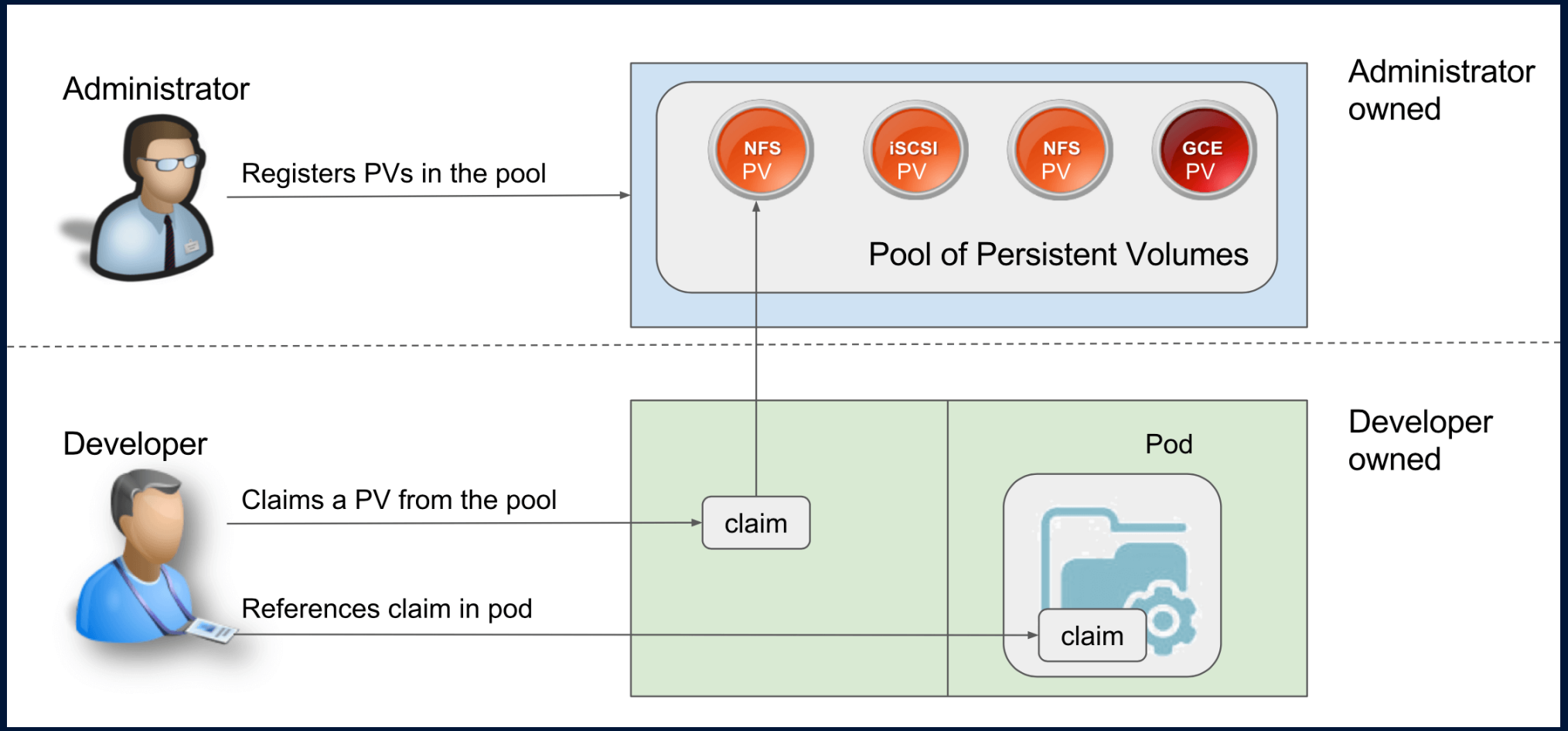
- In order to use a PV, user need to first claim it using a PVC
- PVC requests a PV with the desired specification (size, speed, etc.) from Kubernetes and then binds it to a resource(pod, deployment…) as a volume mount
- User doesn't need to know the underlying provisioning. The claims must be created in the same namespace where the pod is created.



https://www.learnitguide.net/2020/03/kubernetes-persistent-volumes-and-claims.html

Kubernetes

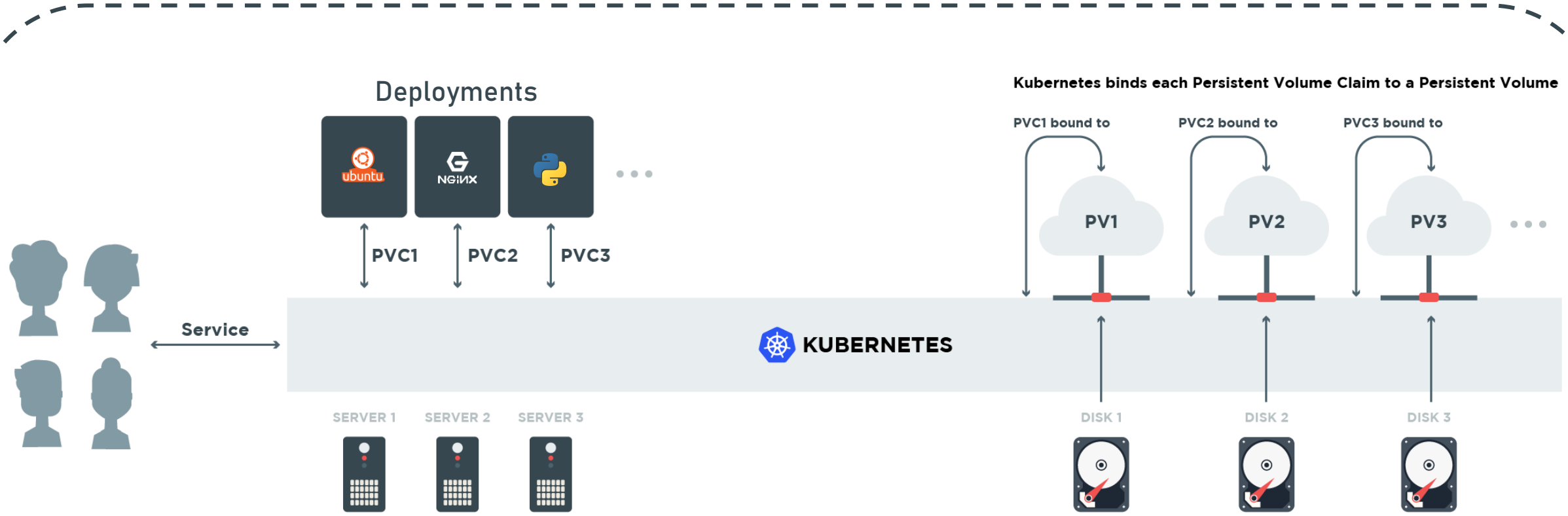
Persistent Volume and Persistent Volume Claim



https://www.learnitguide.net/2020/03/kubernetes-persistent-volumes-and-claims.html

Kubernetes

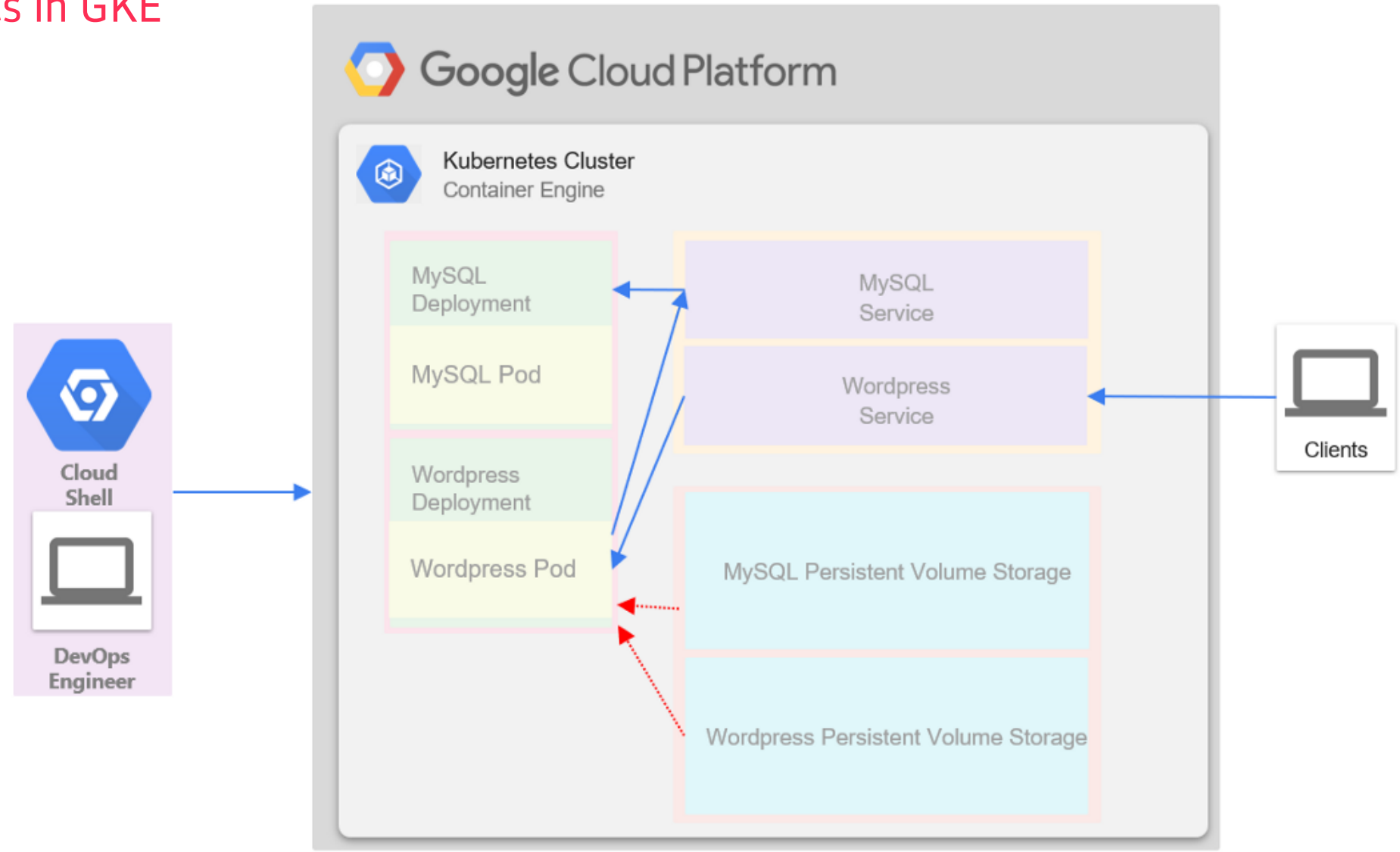
Using PVCs



Kubernetes Cluster

Kubernetes

Using PVCs in GKE



https://medium.com/google-cloud/introduction-to-docker-and-kubernetes-on-gcp-with-hands-on-configuration-part-3-kubernetes-with-eb41f5fc18ae

Kubernetes

Logs

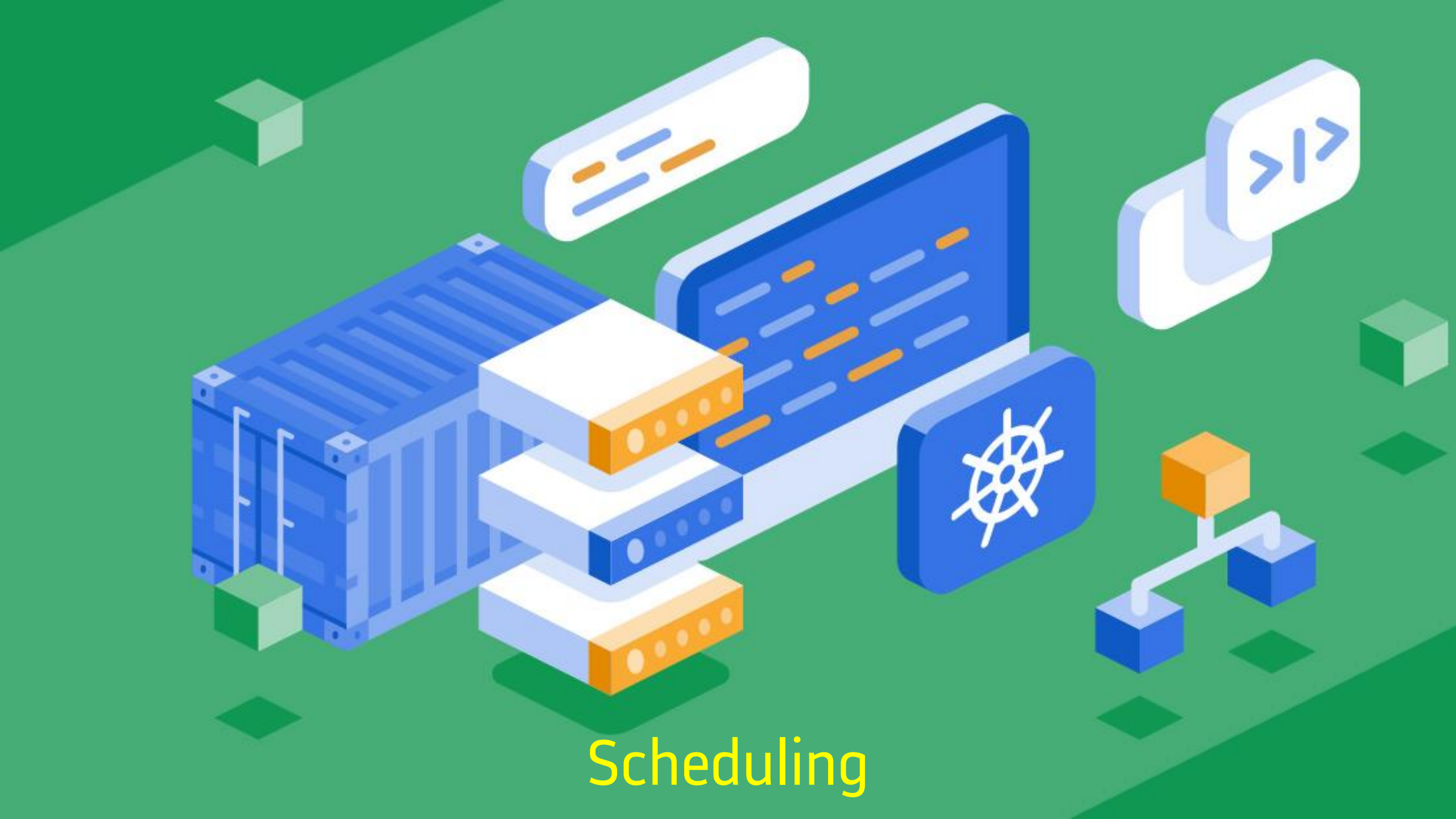
```
kubectl logs my-pod                                 # dump pod logs (stdout)
kubectl logs -l name=myLabel                        # dump pod logs, with label name=myLabel (stdout)
kubectl logs my-pod -c my-container                 # dump pod container logs (stdout, multi-container case)
kubectl logs -l name=myLabel -c my-container        # dump pod logs, with label name=myLabel (stdout)
kubectl logs -f my-pod                              # stream pod logs (stdout)
kubectl logs -f my-pod -c my-container              # stream pod container logs (stdout, multi-container case)
kubectl logs -f -l name=myLabel --all-containers    # stream all pods logs with label name=myLabel (stdout)
kubectl logs my-pod -f --tail=1                     # stream last line of pod logs
kubectl logs deploy/<deployment-name>               # dump deployment logs
```

Kubernetes

Interaction with pods

```
kubectl run -i --tty busybox --image=busybox -- sh                  # Run pod as interactive shell
kubectl run nginx –it --image=nginx  -- bash                        # Run pod nginx
kubectl run nginx --image=nginx  --dry-run -o yaml > pod.yaml       # Run pod nginx and write its spec into a file called
                                                                      pod.yaml
kubectl attach my-pod -i                                            # Attach to Running Container
kubectl exec my-pod -- ls /                                         # Run command in existing pod (1 container case)
kubectl exec my-pod -c my-container -- ls /                         # Run command in existing pod (multi-container case)
```

```
root@k8s-master:/home/osboxes# kubectl run nginx -it --image nginx -- bash
If you don't see a command prompt, try pressing enter.
root@nginx:/#
```



Scheduling

Kubernetes

Scheduling

- Kubernetes users normally don't need to choose a node to which their Pods should be scheduled
- Instead, the selection of the appropriate node(s) is automatically handled by the Kubernetes scheduler
- Automatic node selection prevents users from selecting unhealthy nodes or nodes with a shortage of resources
- However, sometimes manual scheduling is needed to ensure that certain pods only scheduled on nodes with specialized hardware like SSD storages, or to co-locate services that communicate frequently(availability zones), or to dedicate a set of nodes to a particular set of users
- Kubernetes offers several ways to manual schedule the pods. In all the cases, the recommended approach is to use label selectors to make the selection
- Manual scheduling options include:
  1. nodeName
  2. nodeSelector
  3. Node affinity
  4. Taints and Tolerations

https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

Kubernetes

Scheduling

nodeName

- nodeName is a field of PodSpec
- nodeName is the simplest form of node selection constraint, but due to its limitations it is typically not used
- When scheduler finds no nodeName property, it automatically adds this and assigns the pod to any available node
- Manually assign a pod to a node by writing the nodeName property with the desired node name.
- We can also schedule pods on Master by this method
- Some of the limitations of using nodeName to select nodes are:
  - If the named node does not exist, the pod will not be run, and in some cases may be automatically deleted
  - If the named node does not have the resources to accommodate the pod, the pod will fail and its reason will indicate why, for example OutOfmemory or OutOfcpu
  - Node names in cloud environments are not always predictable or stable

Kubernetes

Scheduling

nodeName

```
NAME                STATUS   ROLES    AGE   VERSION   INTERNAL-IP      EXTERNAL-IP
node/k8s-master     Ready    master   5d    v1.18.3   192.168.0.108    <none>
node/k8s-slave01    Ready    <none>   5d    v1.18.3   192.168.0.110    <none>
node/k8s-slave02    Ready    <none>   5d    v1.18.3   192.168.0.109    <none>
```

kubectl apply –f nodeName.yml

```
root@k8s-master:/home/osboxes/demo_kubernetes# kubectl apply -f nodeName.yml
pod/nginx created
root@k8s-master:/home/osboxes/demo_kubernetes# kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP            NODE         NOMINATED NODE   READINESS GATES
nginx   1/1     Running   0          44s   10.244.0.26   k8s-master   <none>           <none>
root@k8s-master:/home/osboxes/demo_kubernetes#
```

nodeName.yml

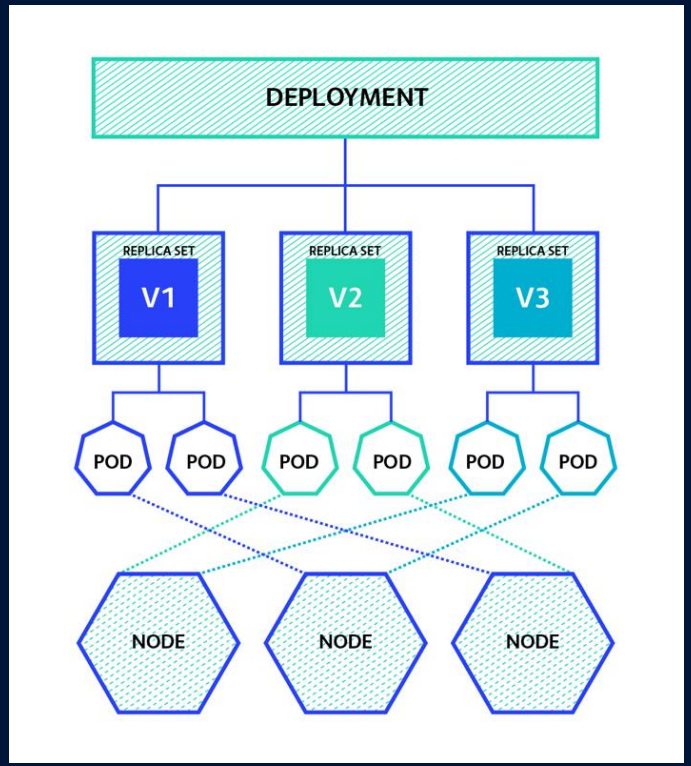
```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
  nodeName: k8s-master
```

Kubernetes

Scheduling

nodeSelector

- nodeSelector is a field of PodSpec
- It is the simplest recommended form of node selection constraint
- It uses labels(key-value pairs) to select matching nodes onto which pods can be scheduled
- Disadvantage with nodeSelector is it uses hard preferences i.e., if matching nodes are not available pods remain in pending state!



check default node labels

kubectl describe node <node-name>

```
root@k8s-master:/home/osboxes/demo_kubernetes# k describe node k8s-master | grep -i label
Labels:             beta.kubernetes.io/arch=amd64
root@k8s-master:/home/osboxes/demo_kubernetes#
```

Kubernetes

Scheduling

nodeSelector

Add labels to nodes

kubectl label nodes <node-name> <label-key>=<label-value>

kubectl label nodes k8s-slave01 environment=dev

```
root@k8s-master:/home/osboxes/demo_kubernetes# kubectl describe node k8s-slave01 | grep -i labels -A 5
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    environment=dev
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=k8s-slave01
                    kubernetes.io/os=linux
root@k8s-master:/home/osboxes/demo_kubernetes#
```

delete a label: kubectl label node <nodename> <labelname> -

Kubernetes

Scheduling

nodeSelector

kubectl apply –f nodeSelector.yml
kubectl get pods –o wide --show-labels
kubectl describe pod <pod-name>

```
root@k8s-master:/home/osboxes/demo_kubernetes# kubectl apply -f nodeSelector.yml
pod/nginx created
root@k8s-master:/home/osboxes/demo_kubernetes# kubectl get pods -o wide --show-labels
NAME    READY   STATUS    RESTARTS   AGE   IP             NODE         NOMINATED NODE   READINESS GATES   LABELS
nginx   1/1     Running   0          20s   10.244.1.113   k8s-slave01   <none>           <none>            env=test
root@k8s-master:/home/osboxes/demo_kubernetes# kubectl describe pod nginx | grep -i selector -A 5
Node-Selectors:  environment=dev
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age   From                 Message
  ----    ------     ----  ----                 -------
root@k8s-master:/home/osboxes/demo_kubernetes#
```

nodeSelector.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
    environment: dev
```

Kubernetes

Scheduling

nodeAffinity

- Node affinity is specified as field nodeAffinity in PodSpec
- Node affinity is conceptually similar to nodeSelector – it allows you to manually schedule pods based on labels on the node. But it has few key enhancements:
  - nodeAffinity implementation is more expressive. The language offers more matching rules besides exact matches created with a logical AND operation in nodeSelector
  - Rules are soft preferences rather than hard requirements, so if the scheduler can't find a node with matching labels, the pod will still be scheduled on other nodes

There are currently two types of node affinity rules:

1. requiredDuringSchedulingIgnoredDuringExecution: Hard requirement like nodeSelector. No matching node label, no pod scheduling!
2. preferredDuringSchedulingIgnoredDuringExecution: Soft requirement. No matching node label, pod gets scheduled on other nodes!

The IgnoredDuringExecution part indicates that if labels on a node change at runtime such that the affinity rules on a pod are no longer met, the pod will still continue to run on the node.

Kubernetes

Scheduling

nodeAffinity

kubectl apply –f nodeAffinity.yml

- Pod gets scheduled on the node has the label environment=production
- If none of the nodes has this label, pod remains in pending state.
- To avoid this, use affinity preferredDuringSchedulingIgnoredDuringExecution

nodeAffinity.yml

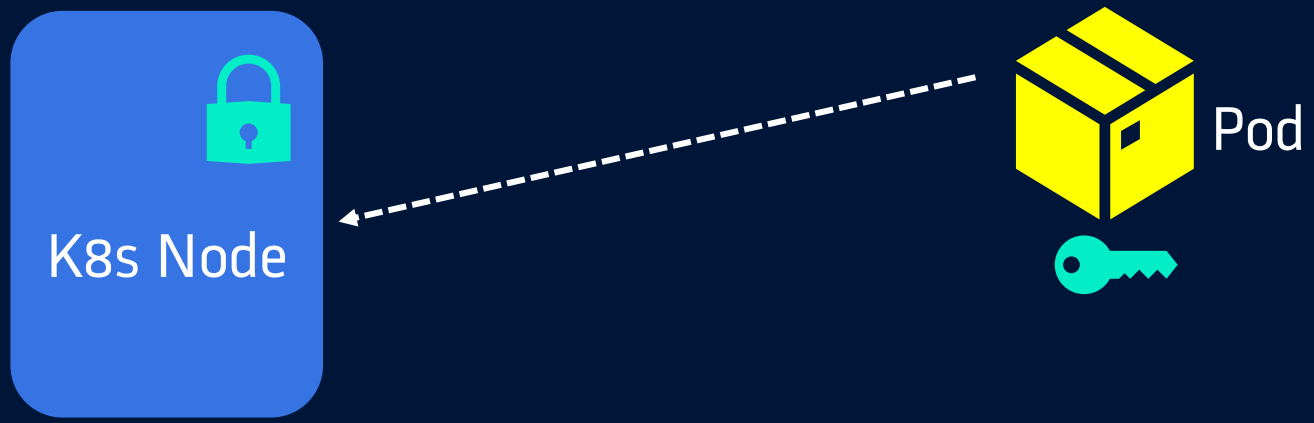
```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: environment
            operator: In
            values:
            - prod
  containers:
  - name: nginx-container
    image: nginx
```

Kubernetes

Scheduling

Taints and Tolerations

- Node affinity, is a property of Pods that attracts them to a set of nodes (either as a preference or a hard requirement)
- Taints are the opposite – they allow a node to repel a set of pods.
- Taints are applied to nodes(lock)
- Tolerations are applied to pods(keys)
- In short, pod should tolerate node's taint in order to run in it. It's like having a correct key with pod to unlock the node to enter it
- Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes



Kubernetes

Scheduling

Taints and Tolerations

- By default, Master node is tainted. So you cannot deploy any pods on Master
- To check taints applied on any node use kubectl describe node <node-name>

Apply taint to nodes

kubectl taint nodes <node-name> key=value:<taint-effect>

- taint's key and value can be any arbitrary string
- taint effect should be one of the supported taint effects such as
  1. NoSchedule: no pod will be able to schedule onto node unless it has a matching toleration.
  2. PreferNoSchedule: soft version of NoSchedule. The system will try to avoid placing a pod that does not tolerate the taint on the node, but it is not required
  3. NoExecute: node controller will immediately evict all Pods without the matching toleration from the node, and new pods will not be scheduled onto the node

Kubernetes

Scheduling

Taints and Tolerations

Apply taint to nodes

kubectl taint nodes k8s-slave01 env=stag:NoSchedule

```
root@k8s-master:/home/osboxes/demo_kubernetes# kubectl taint nodes k8s-slave01 env=stag:NoSchedule
node/k8s-slave01 tainted
root@k8s-master:/home/osboxes/demo_kubernetes#
```

- In the above case, node k8s-slave01 is tained with label env=stag and taint effect as NoSchedule. Only pods that matches this taint will be scheduled onto this node

Check taints on nodes

kubectl describe node k8s-slave01 | grep -i taint

```
root@k8s-master:/home/osboxes/demo_kubernetes# kubectl describe node k8s-slave01 | grep -i taint
Taints:             env=stag:NoSchedule
root@k8s-master:/home/osboxes/demo_kubernetes#
```

Kubernetes

Scheduling

Taints and Tolerations

Apply tolerations to pods

kubectl apply –f taint_toleration.yml
kubectl get pods –o wide

```
root@k8s-master:/home/osboxes/demo_kubernetes# kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP              NODE          NOMINATED NODE   READINESS GATES
nginx-deployment-547c5b4448-jg8v8   1/1     Running   0          32s   10.244.1.117    k8s-slave01   <none>           <none>
nginx-deployment-547c5b4448-k59sf   1/1     Running   0          32s   10.244.1.118    k8s-slave01   <none>           <none>
nginx-deployment-547c5b4448-ngl88   1/1     Running   0          32s   10.244.2.109    k8s-slave02   <none>           <none>
root@k8s-master:/home/osboxes/demo_kubernetes#
```

- Here pods are scheduled onto both the slave nodes.
- Only slave01 is tainted here and matching tolerations are added to pods. So pods are scheduled onto slave-01 as well.
- If we remove the tolerations from the pods and deploy them, they will get scheduled onto slave-02 only as slave01 is tainted and matching toleration is removed/not available with pods!

taint_toleration.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
    spec:
      containers:
      - name: nginx-container
        image: nginx
      tolerations:
      - key: "env"
        operator: "Equal"
        value: "stag"
        effect: "NoSchedule“
```

References

- Docker Installation on Ubuntu
  https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-18-04
- K3s Installation
  https://k33g.gitlab.io/articles/2020-02-21-K3S-01-CLUSTER.html
  https://medium.com/better-programming/local-k3s-cluster-made-easy-with-multipass-108bf6ce577c
- Kubernetes 101
  https://medium.com/google-cloud/kubernetes-101-pods-nodes-containers-and-clusters-c1509e409e16
  https://jamesdefabia.github.io/docs/user-guide/kubectl/kubectl_run/
- Kubeadm
  https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/
- k3s
  https://rancher.com/docs/
- Kubectl commands
  https://kubernetes.io/docs/reference/kubectl/cheatsheet/
  https://kubernetes.io/docs/reference/kubectl/overview/
- Deployments
  https://www.bmc.com/blogs/kubernetes-deployment/
  https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
- Services
  https://kubernetes.io/docs/concepts/services-networking/service/#headless-services
  https://www.edureka.co/community/19351/clusterip-nodeport-loadbalancer-different-from-each-other
  https://theithollow.com/2019/02/05/kubernetes-service-publishing/
  https://www.ovh.com/blog/getting-external-traffic-into-kubernetes-clusterip-nodeport-loadbalancer-and-ingress/
  https://medium.com/@JockDaRock/metalloadbalancer-kubernetes-on-prem-baremetal-loadbalancing-101455c3ed48
  https://medium.com/@cashisclay/kubernetes-ingress-82aa960f658e
- Ingress
  https://www.youtube.com/watch?v=QUfn0EDMmtY&list=PLVSHGLlFuAh89j0mcWZnVhfYgvMmGI0lF&index=18&t=0s
- K8s Dashboard
  https://github.com/kubernetes/dashboard
  https://github.com/indeedeng/k8dash
- YAML
  https://kubeyaml.com/


Thank
You