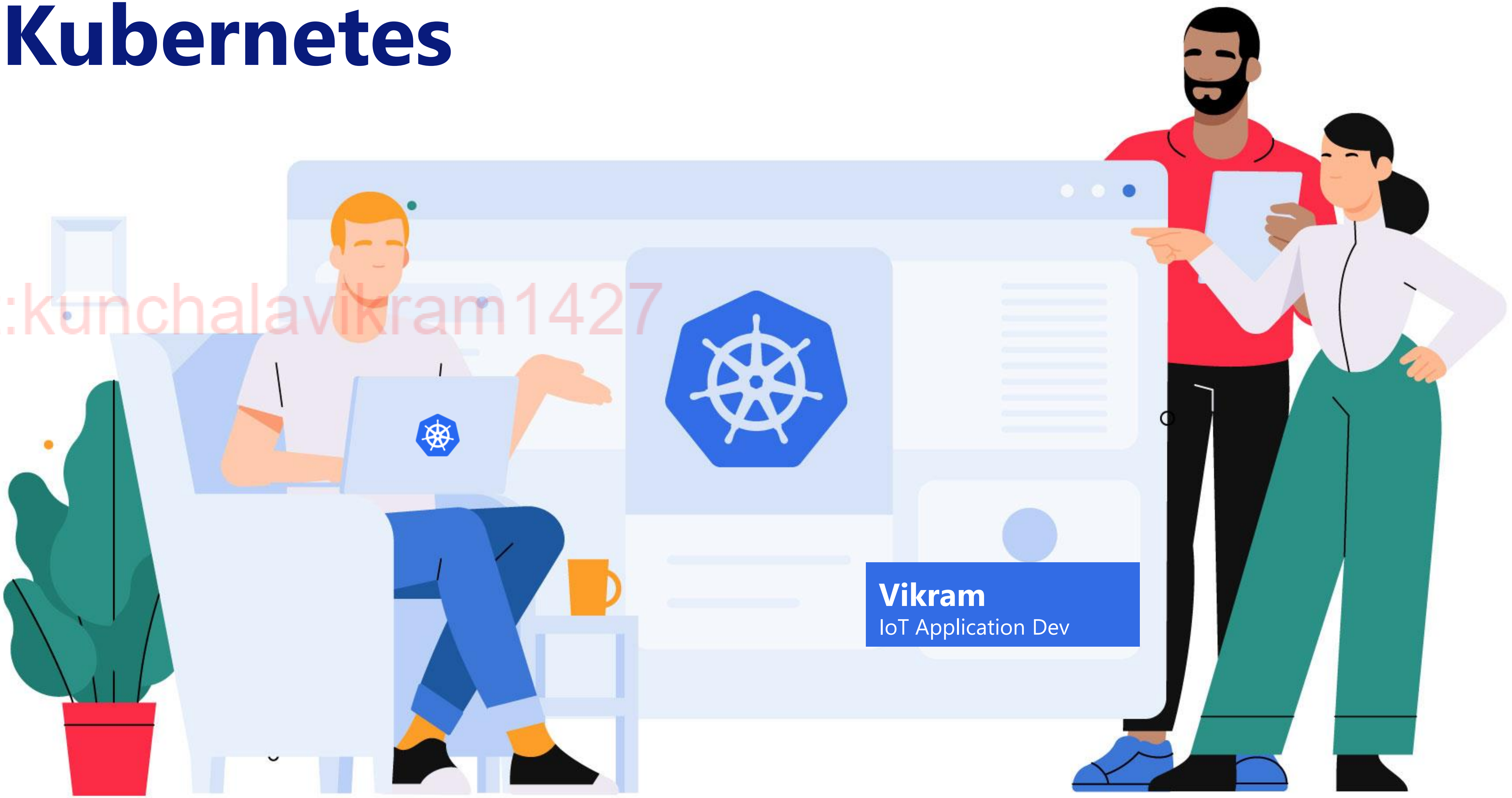
Introduction to **Kubernetes**



Vikram
IoT Application Dev

Contents

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

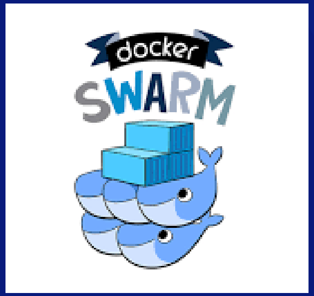
docker SWARM

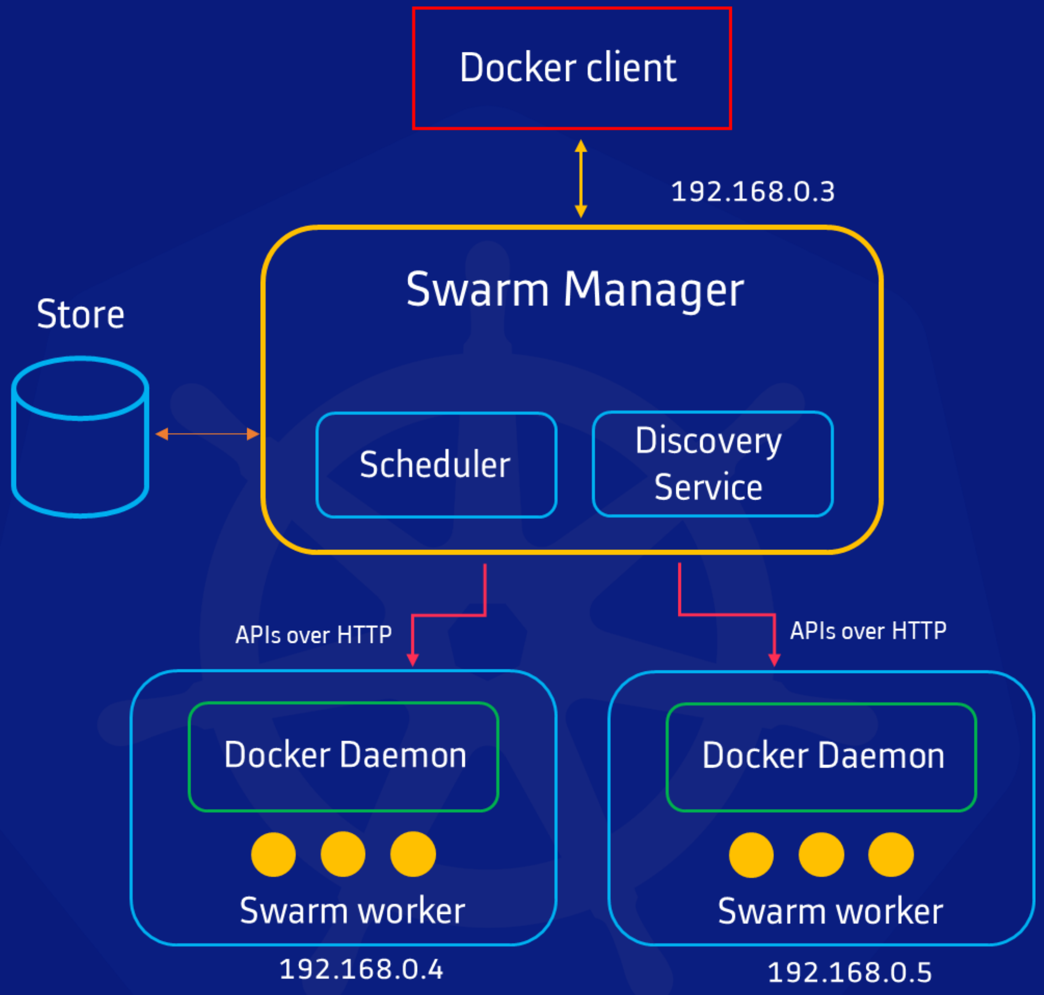
kubernetes

Container orchestration

Docker Swarm

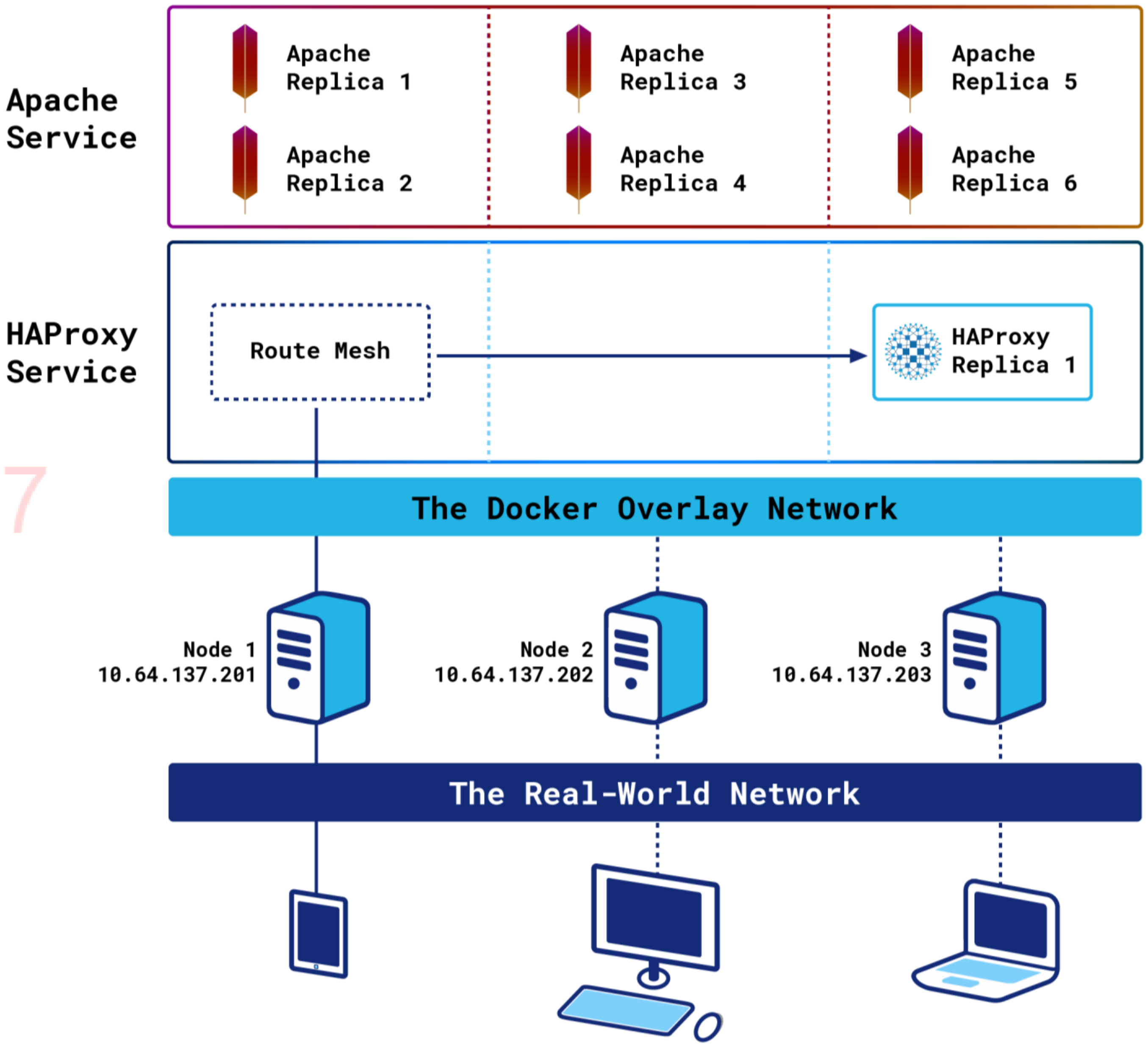
- Docker Swarm is an open-source container orchestration platform and is the native clustering engine for and by Docker
- Any software, services, or tools that run with Docker containers run equally well in Swarm
- It is an alternative to Kubernetes, manages containers and turns the desired state into reality
- It also fixes any future deviations from the desired state
- Docker CLI manages creation of a swarm, deploy application services to a swarm, and manage swarm behaviour





Docker Swarm

- Run containers in Docker Swarm
  https://github.com/kunchalavikram1427/Docker_public/blob/master/Docker_Made_Easy.pdf
- More on Load Balancing and DNS Service Discovery in Docker Swarm using HAProxy
  https://www.haproxy.com/blog/haproxy-on-docker-swarm-load-balancing-and-dns-service-discovery/



Docker Swarm vs Kubernetes

Both Kubernetes and Docker Swarm are two of the most used open-source orchestration platforms providing much of the same functionalities. However, there are several notable differences between them

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, only 3rd party tools
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 has many moving parts and there are countless ways to configure its pieces - from the various system components, network transport drivers, CLI utilities not to mention applications and workloads

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

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

Kubernetes Architecture

Cluster Setup

- There are different ways to bootstrap a cluster, like Minikube, Kind, MicroK8s, K3s, Kubeadm, Docker desktop, KOPS and many more
- We will see how to bootstrap a cluster using Kubeadm method

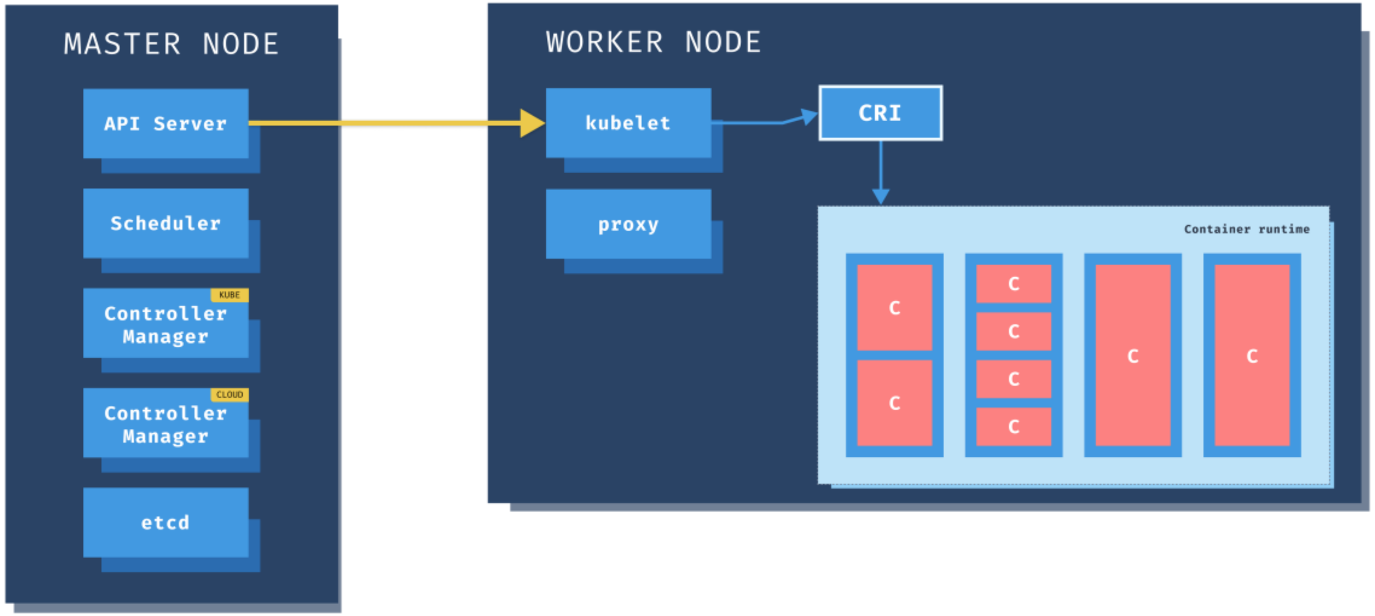
Kubeadm

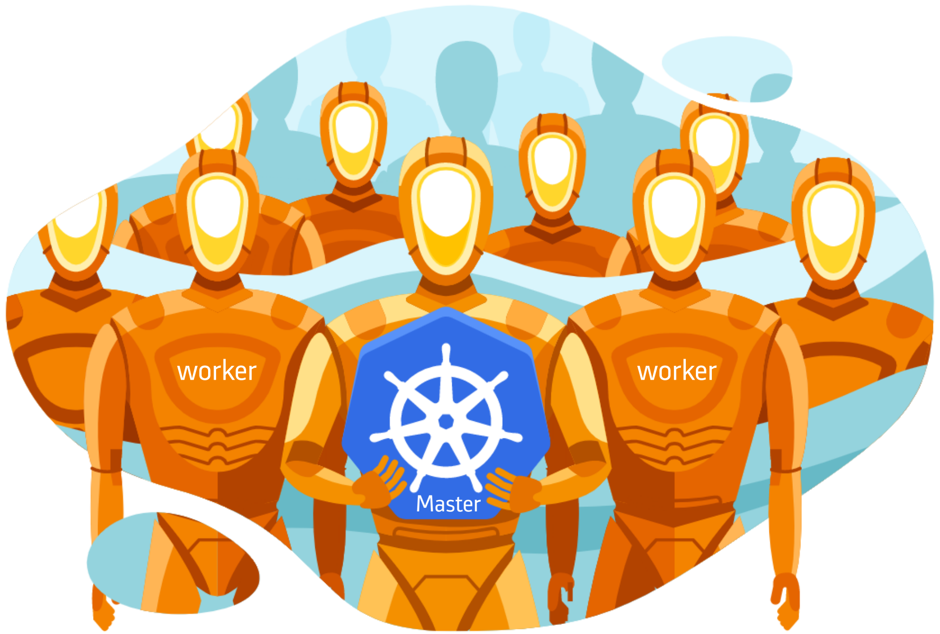
ONLINE METHOD
https://github.com/kunchalavikram1427/Kubernetes_public/blob/master/Bootstrap_K8s_Cluster_Kubeadm.pdf

OFFLINE METHOD
https://github.com/kunchalavikram1427/Kubernetes_public/blob/master/Kubernetes_offline_installation_RHEL7.pdf

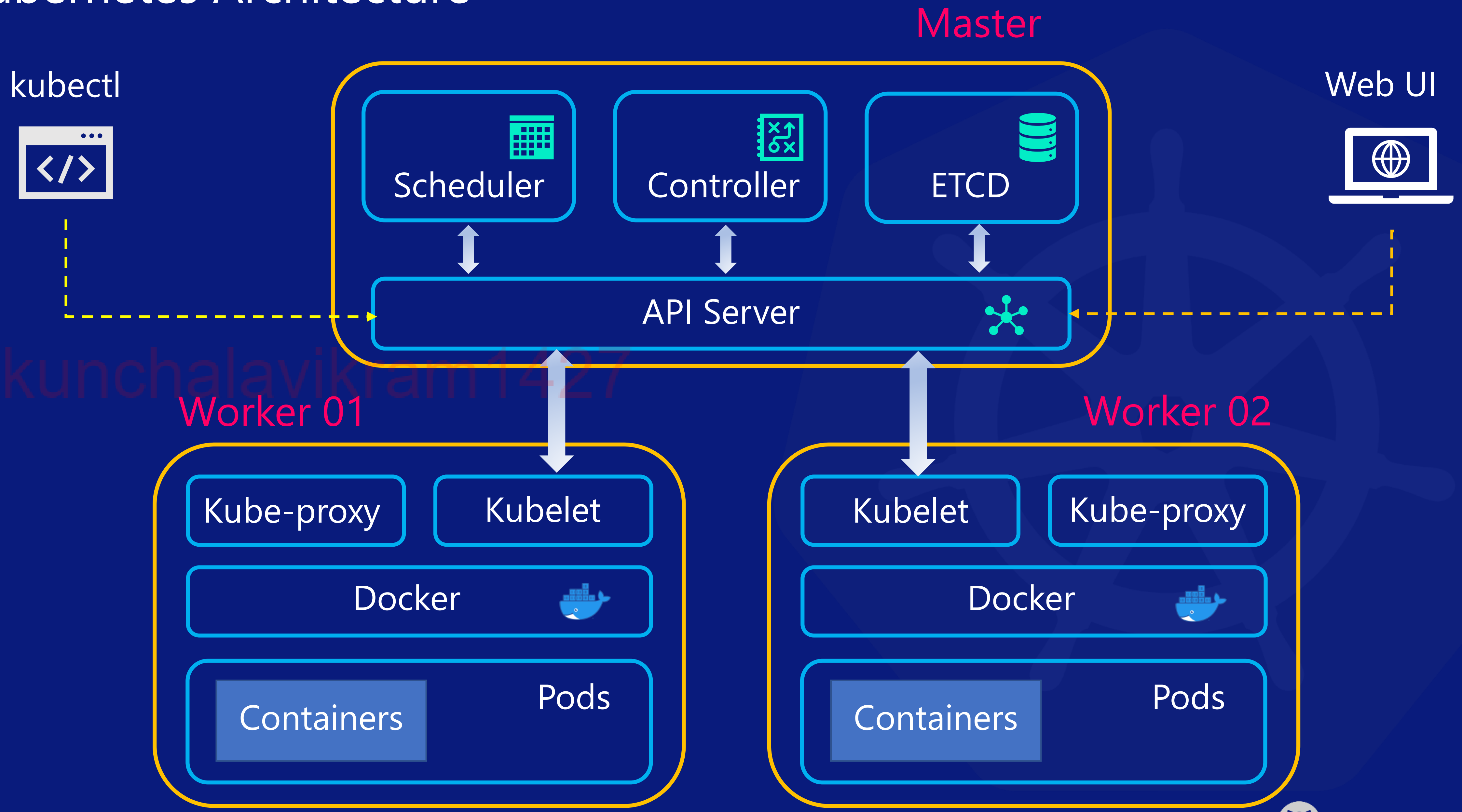
Kubernetes Cluster

- 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
- A node is the smallest unit of computing hardware in Kubernetes, likely be either a physical machine in a datacenter, or virtual machine hosted on a cloud provider like AWS, Azure, GCP or even small computing devices like RaspberryPi
- 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

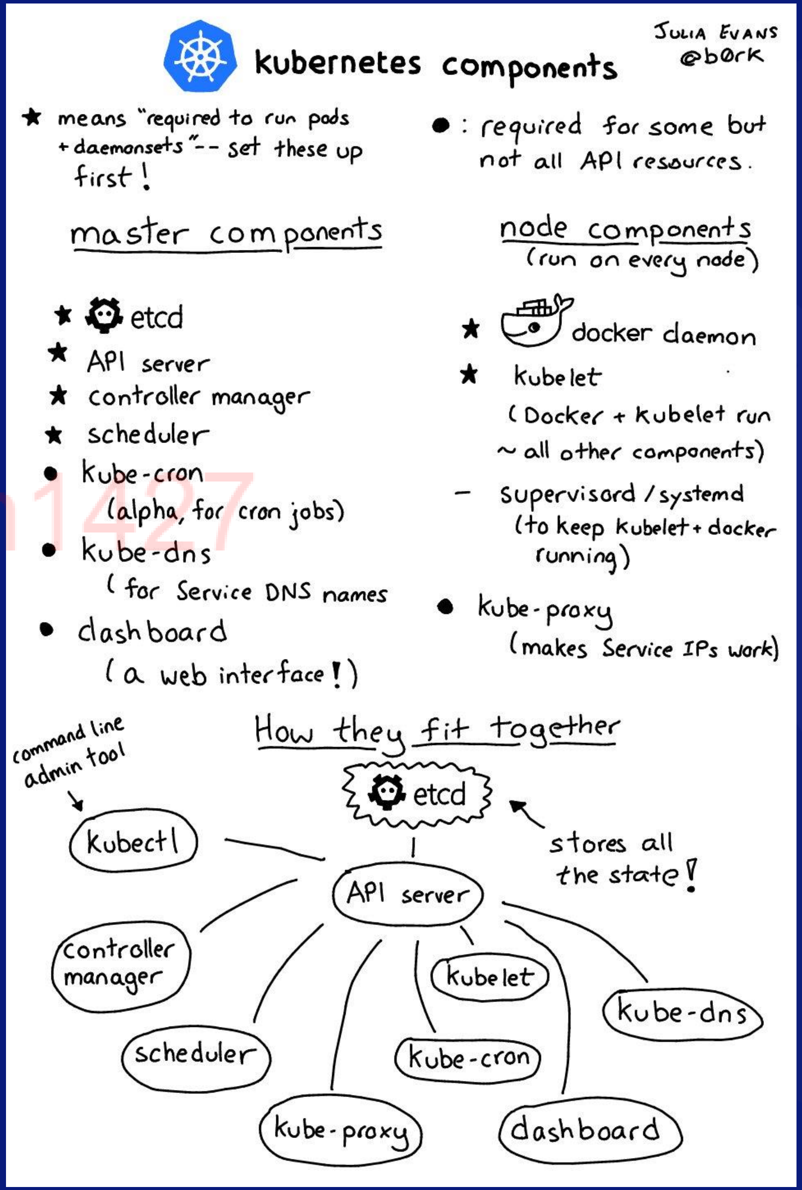




Kubernetes Architecture



Kubernetes Architecture

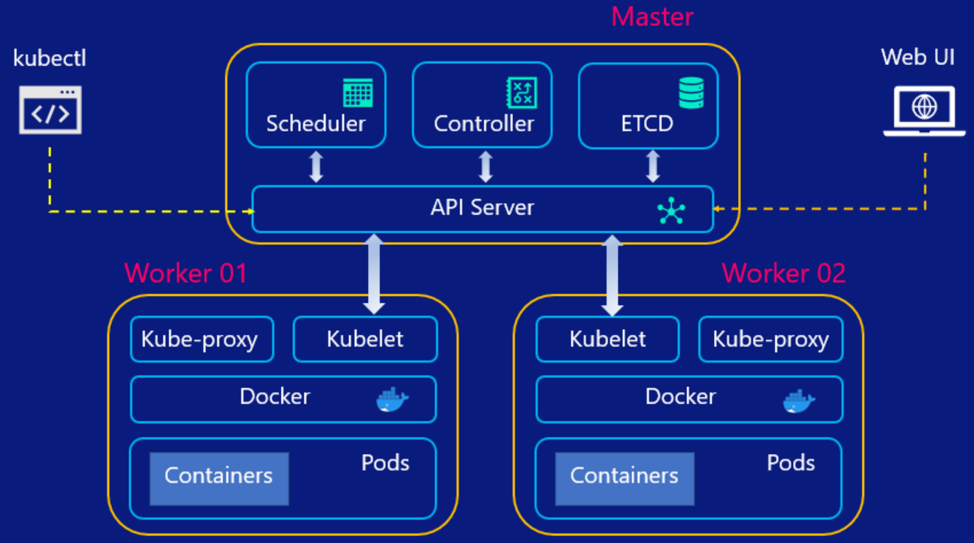


https://blog.alexellis.io/kubernetes-in-10-minutes/

Kubernetes Architecture

Master Node

- 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 – High Availability cluster setup
- 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 together known as Control Plane



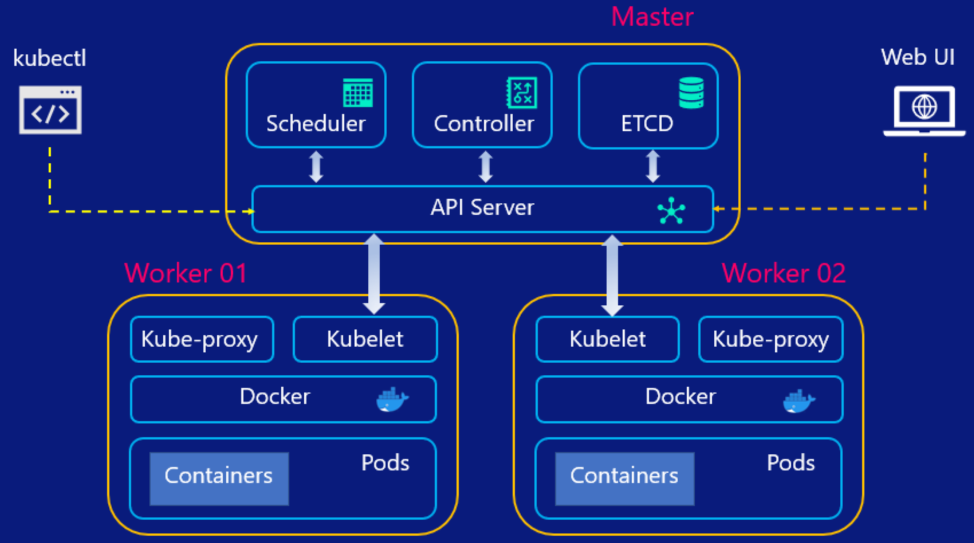
https://kubernetes.io/docs/concepts/overview/components/

Kubernetes Architecture

Master Node

API server

- Exposes k8s APIs and serves like front end for the control plane
- Masters communicate with the rest of the cluster through the kube-apiserver
- 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

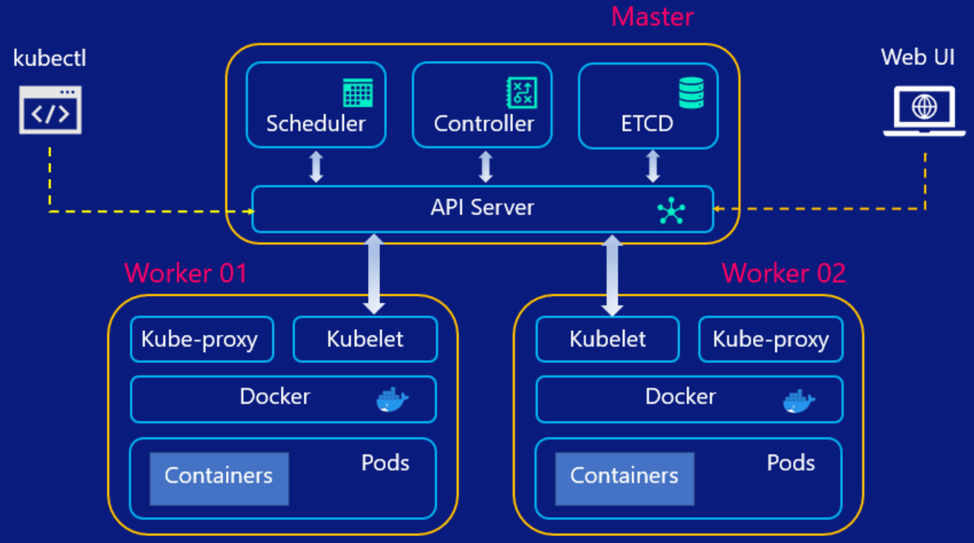
- The controllers are the brain behind orchestration
- They are responsible for noticing and responding when nodes, containers or endpoints goes down. The controllers makes decisions to bring up new containers in such cases
- The kube-controller-manager runs control loops that manage the state of the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

Kubernetes Architecture

Master Node

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

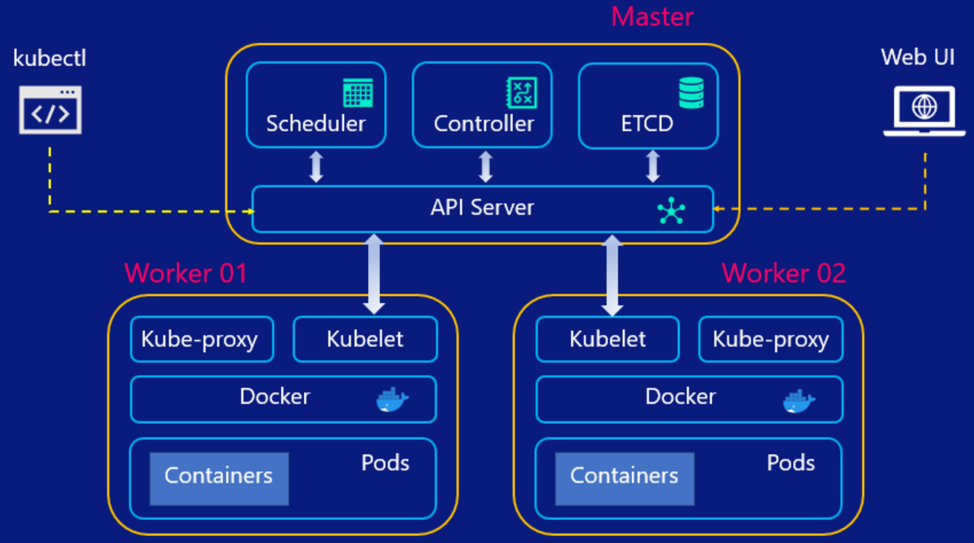
- The scheduler is responsible for distributing work or containers across multiple nodes
- It looks for newly created containers and schedules them onto the Nodes
- Scheduling decisions include factors such as : pod resource requirements, hardware/software/policy constraints, affinity and anti-affinity rules, taints and tolerations etc

Kubernetes Architecture

Worker Nodes

Kubelet

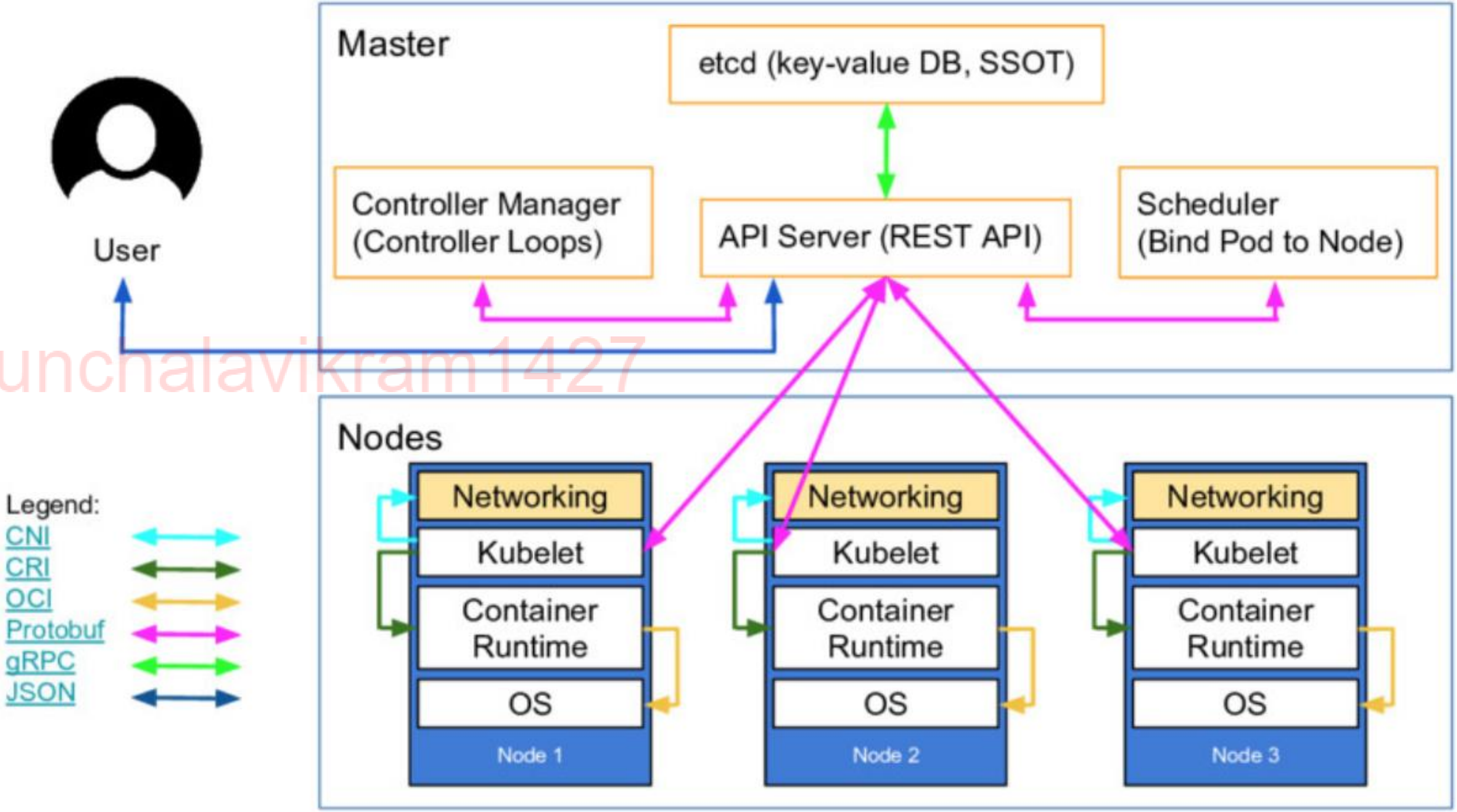
- Worker nodes have the kubelet agent that is responsible for interacting with the master to provide pod & health information of the worker node
- Carry out actions requested by the master on the worker nodes
- The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy
- The kubelet doesn't manage containers which were not created by Kubernetes. Ex: Containers created using docker commands



Kube proxy

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

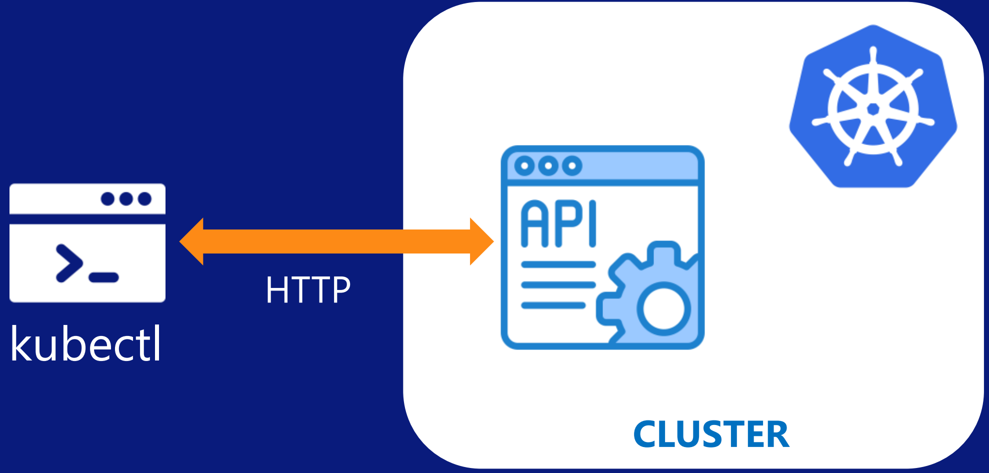
Kubernetes Architecture



Kubernetes Architecture

Kubectl

- kubectl is the command line utility using which we can interact with k8s cluster, like a client
- Kubernetes is fully controlled through its REST API
- Every Kubernetes operation is exposed as an API endpoint and can be executed by an HTTP request to this endpoint
- Kubectl uses these APIs to interact with the cluster
- Can deploy and manage applications on a Kubernetes



- `kubectl run nginx` deploy an application to the cluster
- `kubectl cluster-info` view information about the cluster
- `kubectl get nodes` list all the nodes that are part of the cluster
- `kubectl get componentstatuses` get health status of control plane components

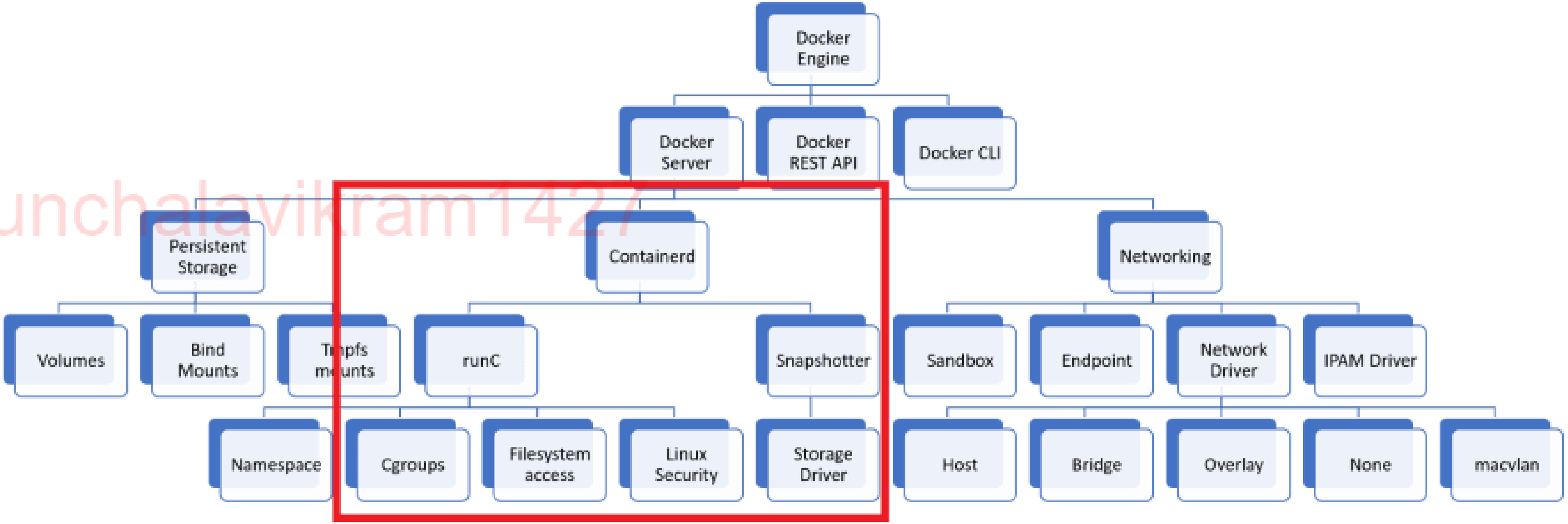
Kubernetes Architecture

Container Runtime Environment

- Containers are not first class objects in the Linux kernel
- Containers are fundamentally composed of several underlying kernel primitives: namespaces (who you are allowed to talk to), cgroups (the amount of resources you are allowed to use), and LSMs (Linux Security Modules—what you are allowed to do). Together, these kernel primitives allow us to set up secure, isolated, and metered execution environments for our processes
- Creating these environment manually each time we want to create a new isolated process would be tiresome and error prone
- To avoid this, all the components have been bundled together in a concept called a container
- The container runtime is the software that is responsible for running these containers
- The runtime executes the container, telling the kernel to assign resource limits, create isolation layers (for processes, networking, and filesystems), and so on, using a cocktail of mechanisms like control groups (cgroups), namespaces, capabilities, SELinux etc
- For Docker, `docker run` is what creates and runs the container, behind the scenes it is runc that is doing the process
- Kubernetes supports several container runtimes: Docker, containerd, CRI-O, rtk etc

Kubernetes Architecture

Container Runtime Environment



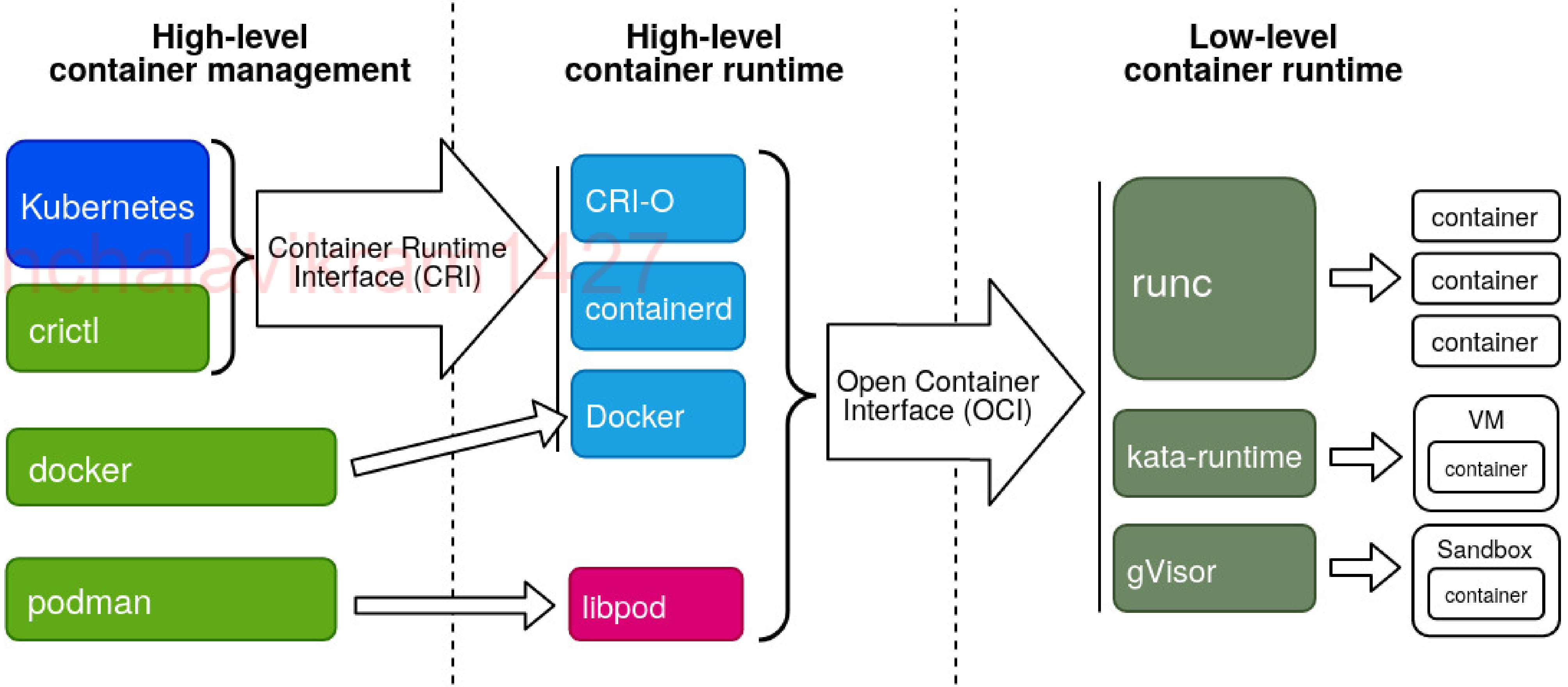
Kubernetes Architecture

CRI (Container Runtime Interface)

- CRI was introduced in Kubernetes 1.5 and acts as a bridge between the kubelet and the container runtime
- High-level container runtimes that want to integrate with Kubernetes are expected to implement CRI. The runtime is expected to handle the management of containers, pods, images etc
- When kubelet wants to run the workload, it uses CRI to communicate with the container runtime running on that same node
- In this way, CRI is simply an abstraction layer or API that allows you to switch out container runtime implementations instead of having them baked into the kubelet
- K8s after trying to support multiple versions of kubelet for different container runtime environments, and trying to keep up with the Docker interface changes, it decided to set a standard interface(CRI) to be implemented by all container runtimes
- This is to avoid large codebase for kubelet for supporting different Container Runtimes
- To implement a CRI, a container runtime environment must be compliant with the Open Container Initiative (OCI)
- OCI includes a set of specifications that container runtime engines must implement and a seed container runtime engine called runc, s a CLI tool for spawning and running containers according to the OCI specification

Kubernetes Architecture

CRI (Container Runtime Interface)



Kubernetes Architecture

Kubernetes deprecated Docker

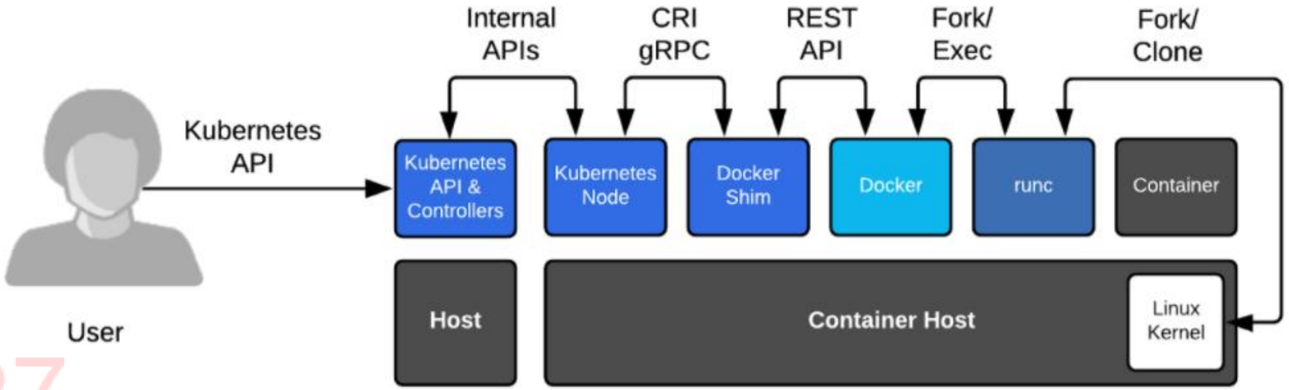
- Kubernetes is deprecating Docker as a container runtime after version 1.20, in favor of runtimes like *containerd* that use the Container Runtime Interface(CRI) created for Kubernetes
- Kubernetes is actually deprecating dockershim, which is a component in Kubernetes' kubelet implementation, communicating with Docker Engine
- Docker does not support Kubernetes Runtime API called CRI(Container Runtime Interface) and Kubernetes have been using a bridge service called dockershim. It converts Docker API and CRI, but it will no longer be provided from Kubernetes side within a few minor releases
- Kubernetes actually needs only container runtime. It doesn't need extra features provided by Docker like Docker Networks and Volumes which are never used by K8s and having them could pose a security risk. The less features you have, the smaller the attack surface becomes

> Docker support in the kubelet is now deprecated and will be removed in a future release. The kubelet uses a module called **dockershim** which implements CRI support for Docker, and it has seen maintenance issues in the Kubernetes community. It is advised to evaluate moving to a container runtime that is a full-fledged implementation of CRI (v1alpha1 or v1 compliant) as they become available.

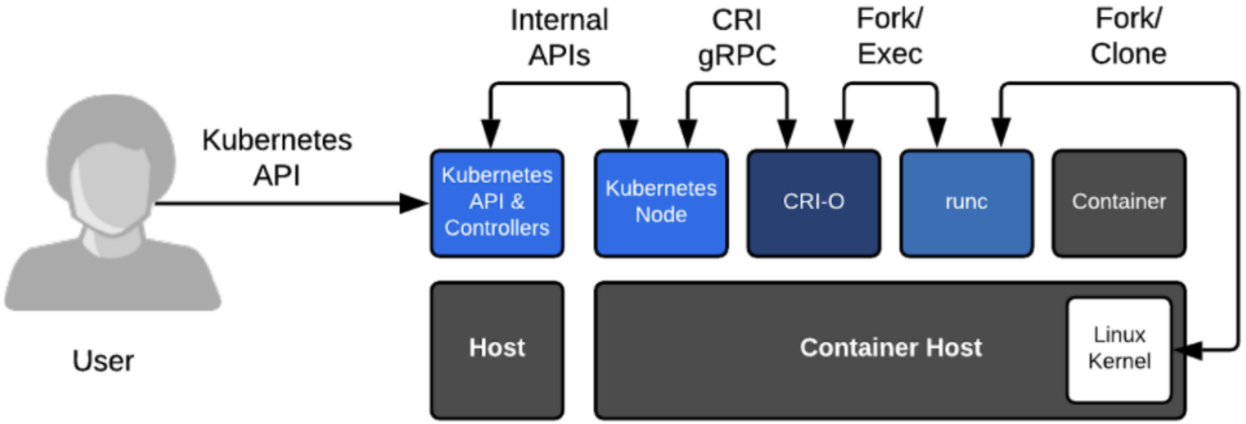
Kubernetes Architecture

Kubernetes deprecated Docker

Kubernetes using Docker



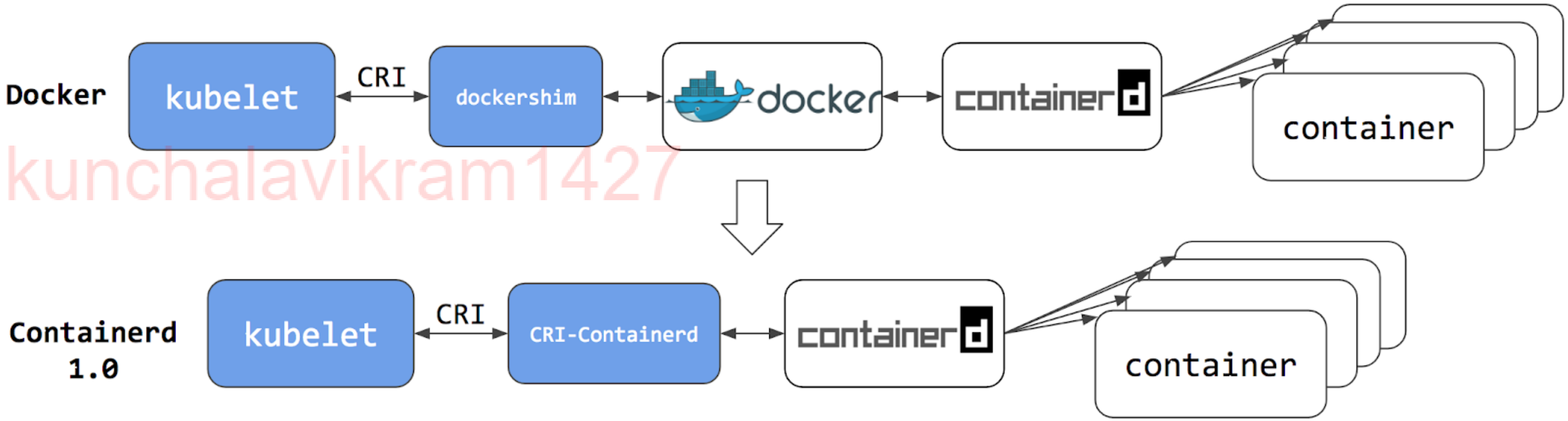
Kubernetes without Dockershim



With docker being deprecated, we should now use container runtimes like containerd, rkt, cri-o which supports container runtime interfaces developed for k8s

Kubernetes Architecture

Kubernetes deprecated Docker

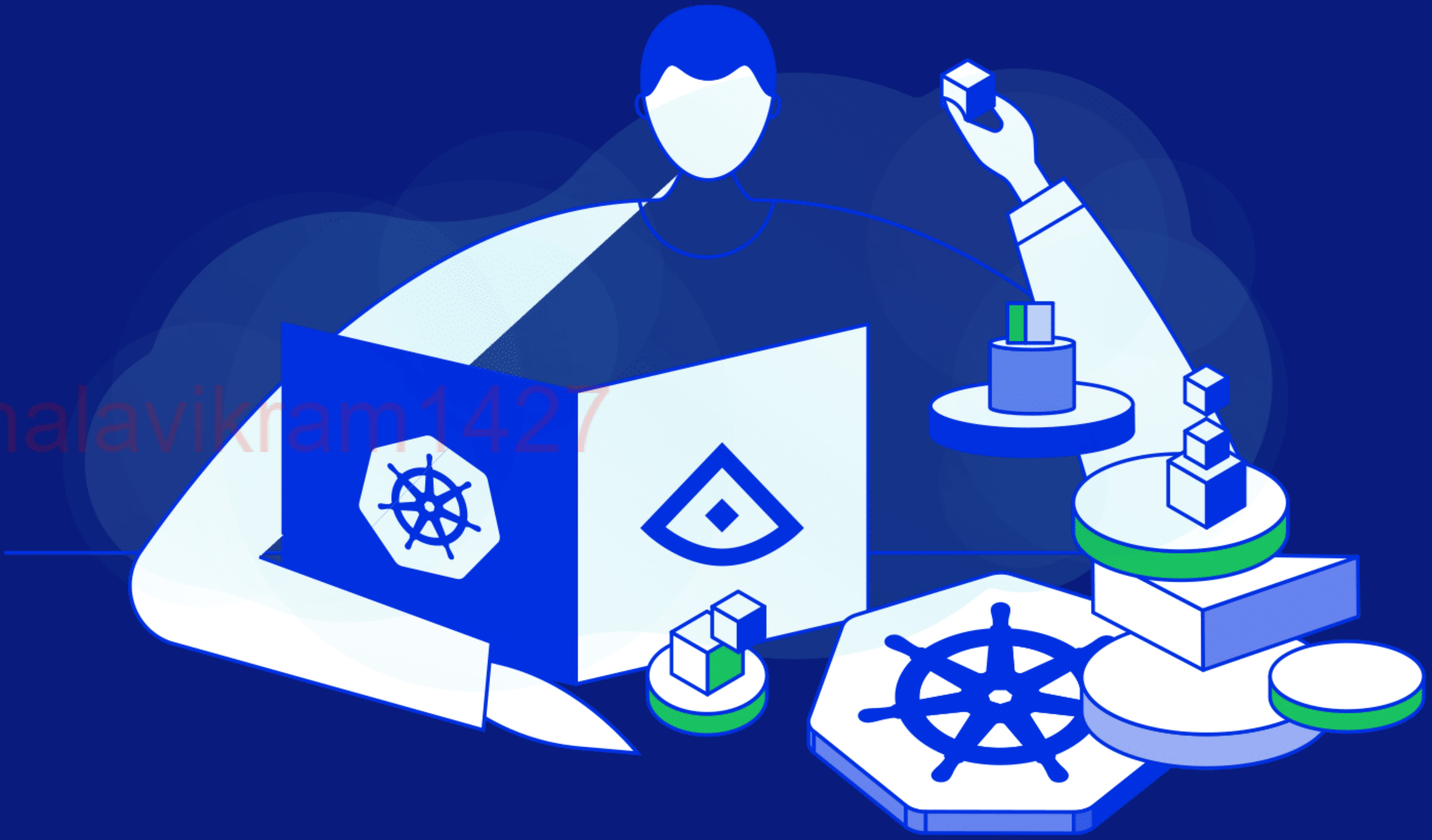


With docker being deprecated, we should now use container runtimes like containerd, rkt, cri-o which supports container runtime interfaces developed for k8s

Kubernetes Architecture

Kubernetes deprecated Docker

- Developers can still use the Docker platform to build, share, and run containers on Kubernetes! This change primarily impacts operators and administrators for Kubernetes and doesn't impact developer work flows. The images Docker builds are compliant with OCI (Open Container Initiative), are fully supported on containerd, and will continue to run great on Kubernetes
- The OCI runtime specification governs how containers are expressed at runtime, and almost every container engine on the planet uses runc which is not only the reference implementation, but also the gold standard for communicating with the Linux kernel to start containers
- If you're using Docker, you're already using containerd. Docker's runtime is build upon containerd while providing a great developer experience around it. Production environments may have no need for Docker's great developer experience, it's reasonable to directly use lightweight runtimes like containerd
- Docker set up in 2015 the Open Container Initiative (OCI) in order to support fully interoperable container standards, and created the containerd project, along with Google and IBM, in 2016, with the goal of this transition in mind
- Containerd was donated to the CNCF in 2017, and has grown to incorporate the containerd CRI project to interface with Kubernetes, as well as seeing a host of innovation and investment from across the industry, including from Amazon, Google, Microsoft and IBM



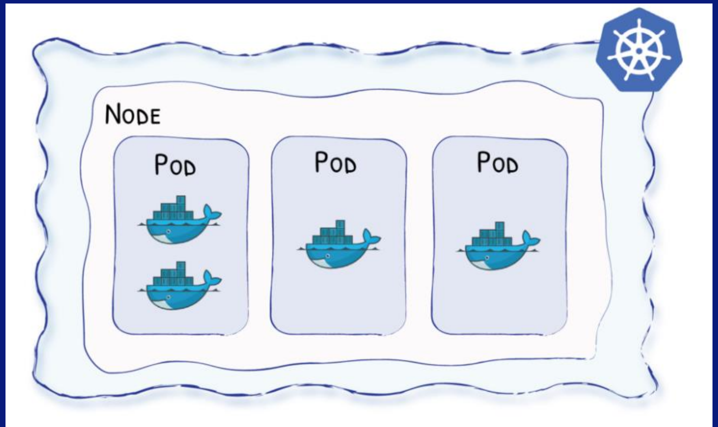
Pods

Kubernetes

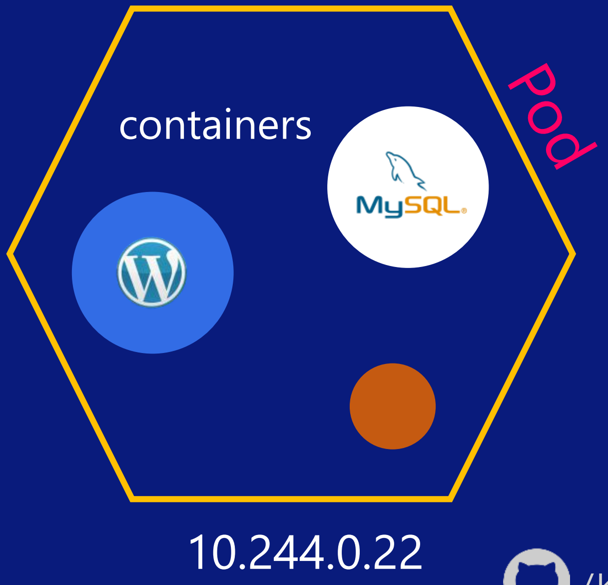
Pods

- 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, scheduled, 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..



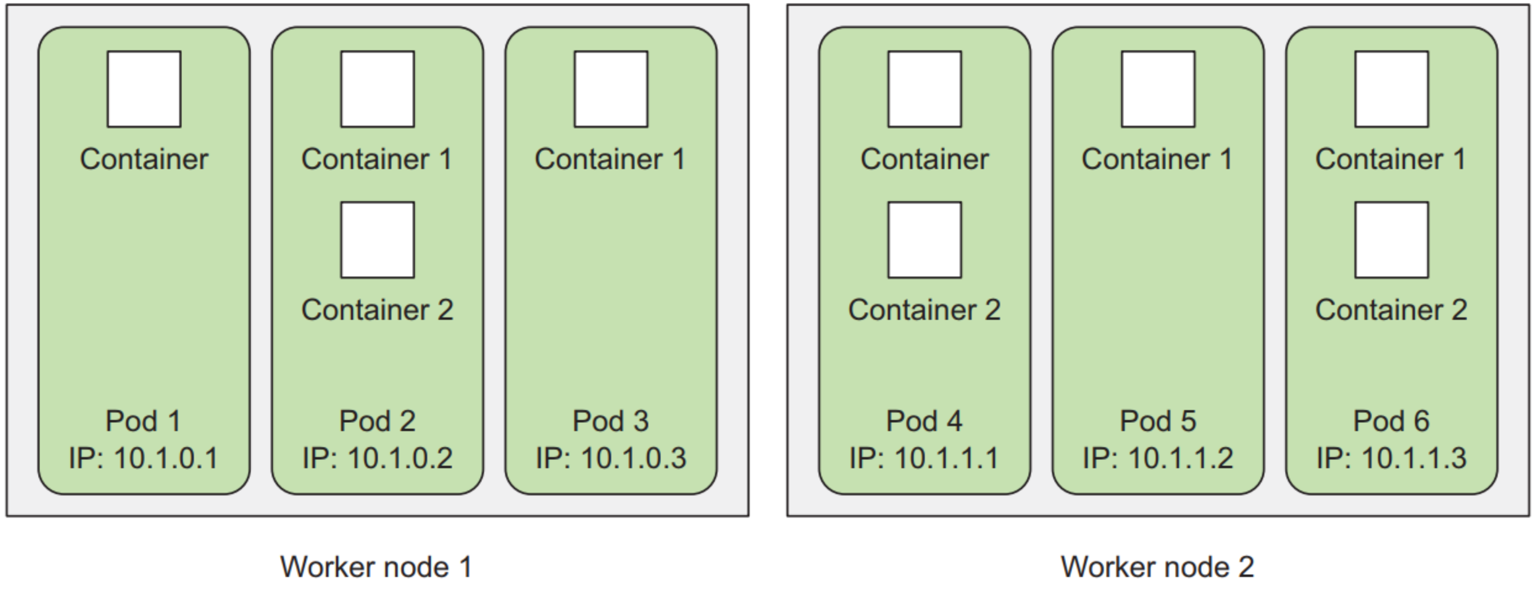
If a Pod consists of multiple containers, they will share the same resources / network



Kubernetes

Pods

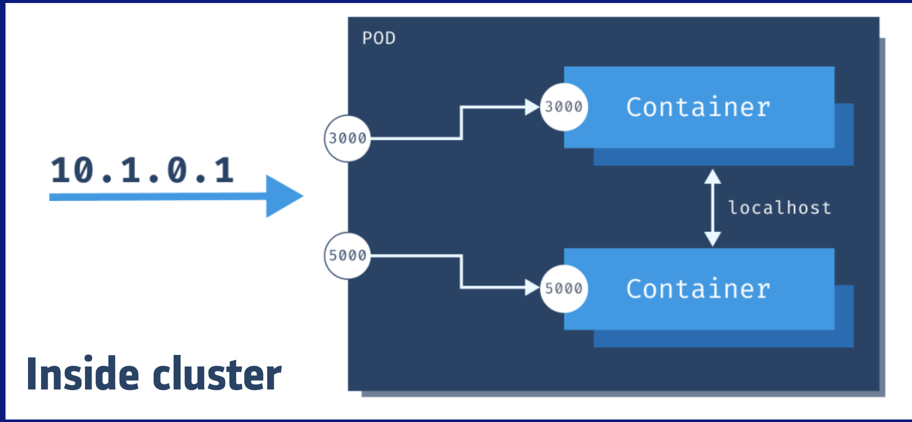
- A pod can have one or more tightly related containers that will always run together on the same worker node and in the same Linux namespace
- Each pod is like a separate logical machine with its own IP, hostname, processes, and so on, running a single application
- All the containers in a pod will appear to be running on the same logical machine, whereas containers in other pods, even if they're running on the same worker node, will appear to be running on a different one



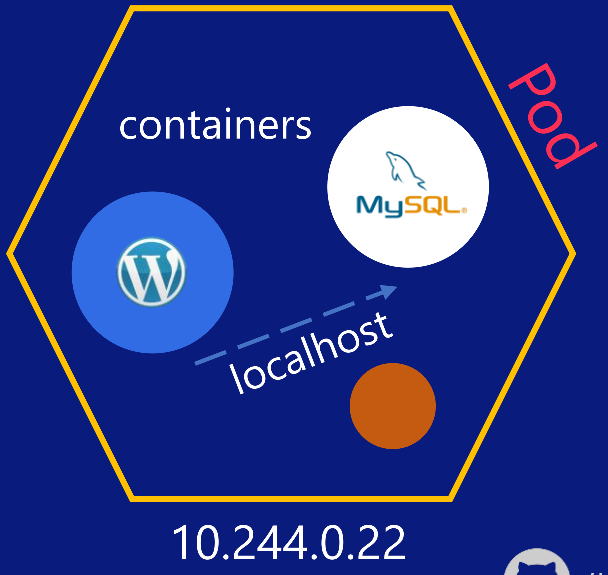
Kubernetes

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.



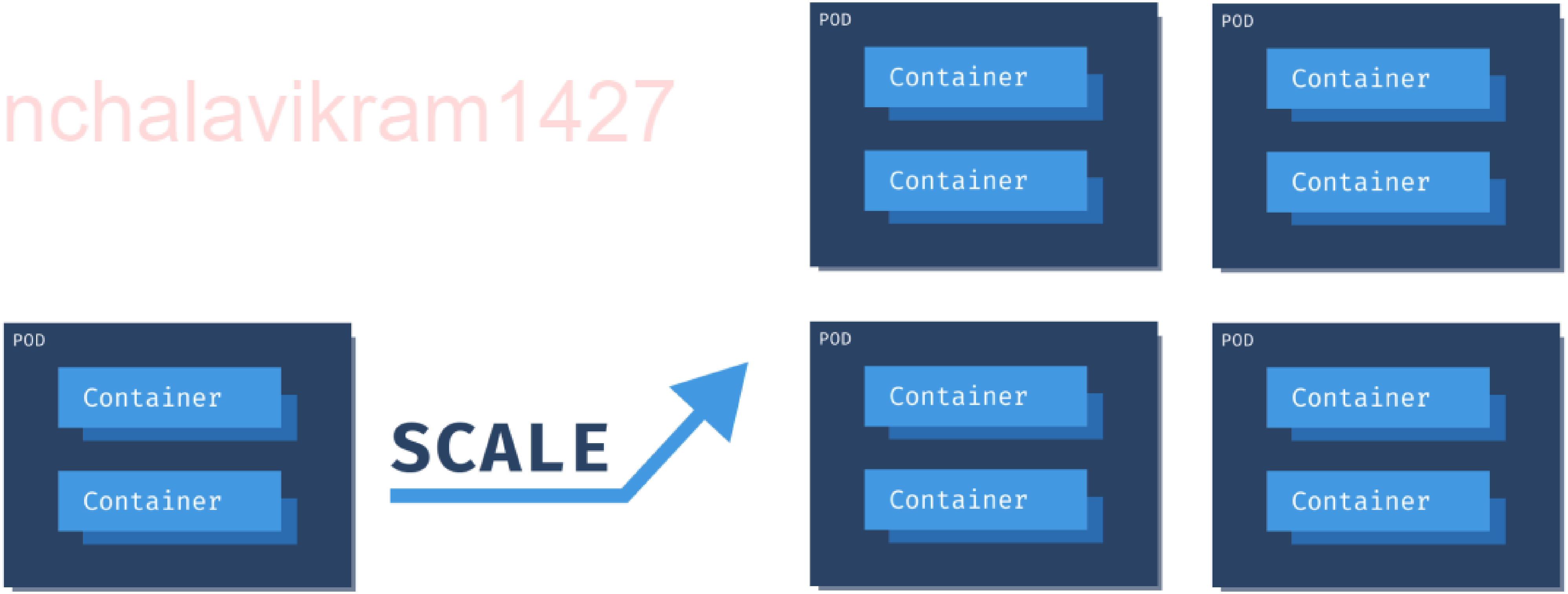
Using the example from the above figure, you could run curl 10.1.0.1:3000 to communicate to the one container and curl 10.1.0.1:5000 to communicate to the other container from other pods. However, if you wanted to talk between containers - for example, calling the top container from the bottom one, you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



Kubernetes

Init Containers

- Init containers are exactly like regular containers, except that they always run to completion
- Each init container must complete successfully before the next one starts
- They can contain utilities or custom code for setup that are not present in an app image Ex: sed, awk, python, or dig during setup, also functionalities like Clone a Git repository into a Volume
- If a Pod's init container fails, Kubernetes repeatedly restarts the Pod until the init container succeeds
- However, if the Pod has a restartPolicy set to Never, and an init container fails during startup of that Pod, Kubernetes treats the overall Pod as failed
- Init containers do not support lifecycle, livenessProbe, readinessProbe, or startupProbe because they must run to completion before the Pod can be ready
- If you specify multiple init containers for a Pod, kubelet runs each init container sequentially.
- Each init container must succeed before the next can run
- When all of the init containers have run to completion, kubelet initializes the application containers for the Pod and runs them as usual
- Because init containers run to completion before any app containers start, init containers offer a mechanism to block or delay app container startup until a set of preconditions are met. Once preconditions are met, all of the app containers in a Pod can start in parallel

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 these objects in Kubernetes: Imperatively and Declaratively

Imperatively

- Involves using any of the verb-based commands like kubectl run, 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

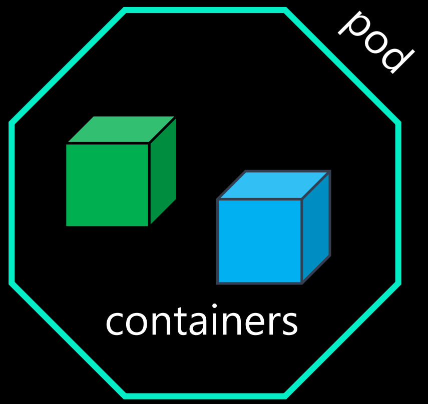
- K8s object configuration files - Written in YAML or JSON
- They describe the desired state of your application in terms of Kubernetes API objects
- A file can include one or more API object descriptions (manifests)
- Manifest file has 4 mandatory fields as shown below

✓ 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




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

Multiple resource definitions

Kubernetes

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
```

Terminal

```
$ kubectl api-resources
  NAME                 SHORTNAMES   APIGROUP   NAMESPACED   KIND
  bindings                                     true         Binding
  componentstatuses    cs                      false        ComponentStatus
  configmaps           cm                      true         ConfigMap
  endpoints            ep                      true         Endpoints
  events               ev                      true         Event
  limitranges          limits                  true         LimitRange
$ kubectl api-versions
  admissionregistration.k8s.io/v1
  admissionregistration.k8s.io/v1beta1
  apiextensions.k8s.io/v1
  apiextensions.k8s.io/v1beta1
  apiregistration.k8s.io/v1
  apiregistration.k8s.io/v1beta1
  apps/v1

$ kubectl explain pod or kubectl explain pod.apiVersion

  KIND:     Pod
  VERSION:  v1

  DESCRIPTION:
       Pod is a collection of containers that can run on a host. This resource is
       created by clients and scheduled onto hosts.

  FIELDS:
     apiVersion   <string>
       APIVersion defines the versioned schema of this representation of an
       object. Servers should convert recognized schemas to the latest internal
       value, and may reject unrecognized values. More info:
       https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources

     kind <string>
       Kind is a string value representing the REST resource this object
       represents. Servers may infer this from the endpoint the client submits
       requests to. Cannot be updated. In CamelCase. More info:
       https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds

     metadata     <Object>
       Standard object's metadata. More info:
       https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata

     spec <Object>
       Specification of the desired behavior of the pod. More info:
       https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status
```

Kubernetes

Man Pages

Few more...

```
kubectl api-resources --namespaced=true      # All namespaced resources
kubectl api-resources --namespaced=false     # All non-namespaced resources
kubectl api-resources -o name                # All resources with simple output
                                               (just the resource name)
kubectl api-resources -o wide                # All resources with expanded output
kubectl api-resources --verbs=list,get       # All resources that support the
                                               "list" and "get" request verbs
kubectl api-resources --api-group=extensions # All resources in the "extensions"
                                               API group
```

Kubernetes

In the cluster

kubectl get componentstatuses – status of k8s components
kubectl version -o yaml / kubectl version --short

```
$ k version -o yaml
clientVersion:
  buildDate: "2020-07-17T19:00:19Z"
  compiler: gc
  gitCommit: 4c6976793196d70bc5cd29d56ce5440c9473648e
  gitTreeState: clean
  gitVersion: v1.17.9-eks-4c6976
  goVersion: go1.13.9
  major: "1"
  minor: 17+
  platform: windows/amd64
serverVersion:
  buildDate: "2020-07-16T20:46:15Z"
  compiler: gc
  gitCommit: 6f56fa1d68a5a48b8b6fdefa8eb7ead2015a4b3a
  gitTreeState: clean
  gitVersion: v1.18.6+k3s1
  goVersion: go1.13.11
  major: "1"
  minor: "18"
  platform: linux/amd64
```

kubectl get nodes -o wide (or)
kubectl get nodes -o yaml/json – get nodes info in the output format of wide/json/yaml


```
root@k8s-master:/home/osboxes# kubectl get nodes -o wide
NAME          STATUS   ROLES                   AGE   VERSION   INTERNAL-IP       EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION     CONTAINER-RUNTIME
k8s-master    Ready    control-plane,master    17d   v1.20.2   192.168.107.127   <none>        Ubuntu 18.04.3 LTS   5.0.0-23-generic   docker://20.10.2
k8s-slave01   Ready    <none>                  17d   v1.20.2   192.168.107.90    <none>        Ubuntu 18.04.3 LTS   5.0.0-23-generic   docker://20.10.2
k8s-slave02   Ready    <none>                  17d   v1.20.2   192.168.107.76    <none>        Ubuntu 18.04.3 LTS   5.0.0-23-generic   docker://20.10.2
root@k8s-master:/home/osboxes#
```

kubectl cluster-info

```
root@k8s-master:/home/osboxes# kubectl cluster-info
Kubernetes control plane is running at https://192.168.56.2:6443
KubeDNS is running at https://192.168.56.2:6443/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`

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

dry-run doesn't run the command but will show what resources the command would deploy in the cluster

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


```
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: {}
```

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

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

Kubernetes

Creating Pods: Imperative Way

`kubectl run nginx-pod --image=nginx --port=80` – Exposes port 80 of the container
`kubectl get pods -o wide` – Get list of Pods

```
root@k8s-master:/home/osboxes# kubectl run nginx-pod --image=nginx
pod/nginx-pod created
root@k8s-master:/home/osboxes# kubectl get pods -o wide
NAME        READY   STATUS    RESTARTS   AGE   IP            NODE          NOMINATED NODE   READINESS GATES
nginx-pod   1/1     Running   0          22s   10.244.1.83   k8s-slave01   <none>           <none>
root@k8s-master:/home/osboxes#
```


`kubectl describe pod nginx-pod` – display extended information of pod like its IP, Port, Node on which it is scheduled, labels etc

```
root@k8s-master:/home/osboxes# kubectl describe pod nginx-pod
Name:         nginx-pod
Namespace:    default
Priority:     0
Node:         k8s-slave02/192.168.0.103
Start Time:   Fri, 05 Feb 2021 02:50:46 -0500
Labels:       run=nginx-pod
Annotations:  <none>
Status:       Running
IP:           10.244.2.75
IPs:
  IP:  10.244.2.75
```

10.244.2.75

Kubernetes

Creating Pods: Imperative Way

Checking pod logs

kubectl logs nginx-pod – check logs of the pod
kubectl logs -f nginx-pod – check logs of the pod in follow mode(real time)
kubectl logs -f --tail 10 nginx-pod – check last ‘n’ logs of the pod in follow mode

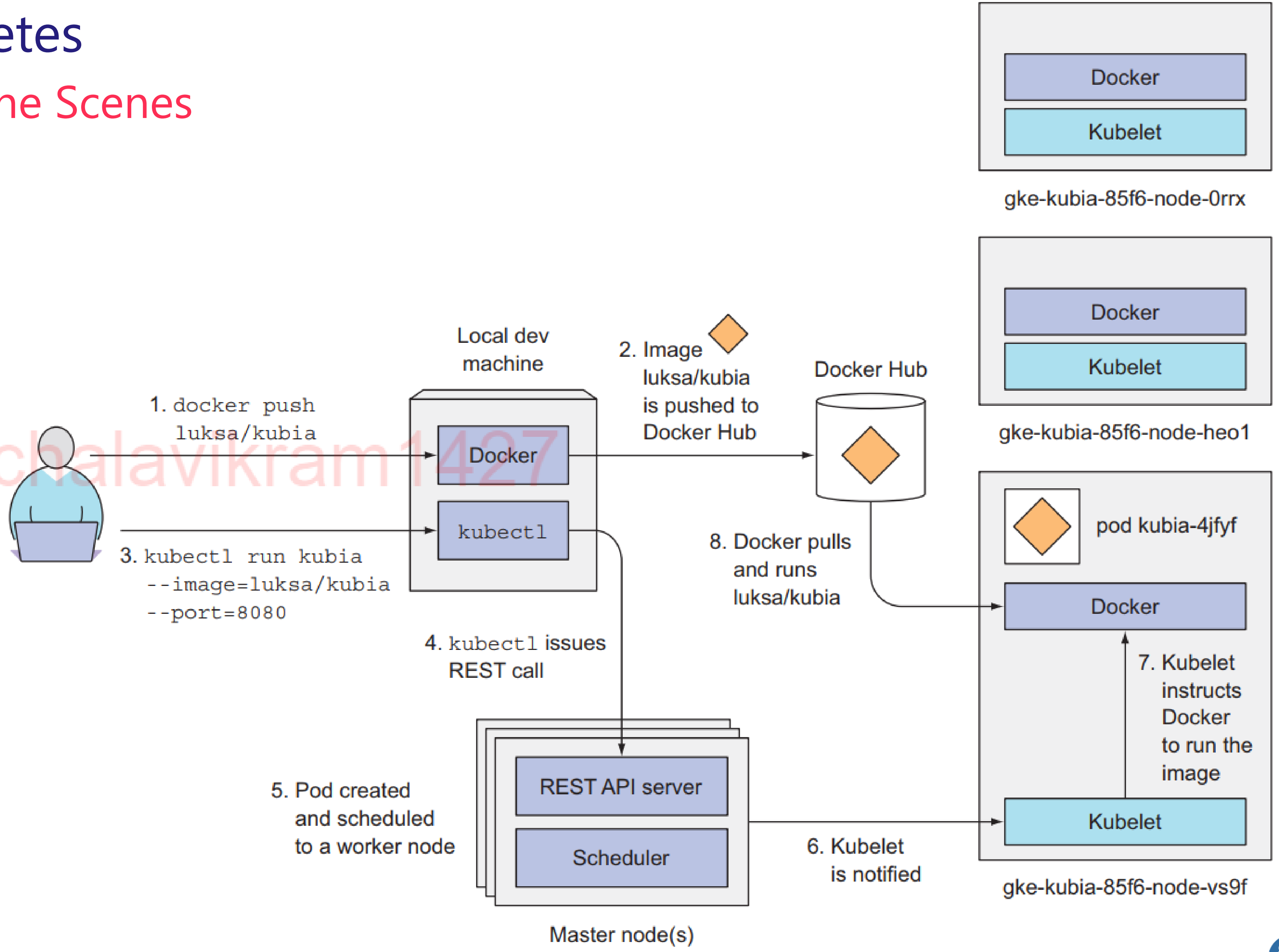
```
root@k8s-master:/home/osboxes/kubernetes# kubectl logs -f --tail 2 nginx-pod
/docker-entrypoint.sh: Configuration complete; ready for start up
10.244.0.0 - - [05/Feb/2021:11:36:02 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.58.0" "-"
```


Interactive shell access to a running pod

kubectl exec --stdin --tty nginx-pod -- /bin/sh
kubectl exec -it nginx-pod -- /bin/sh

```
root@k8s-master:/home/osboxes/kubernetes# kubectl exec --stdin --tty nginx-pod  -- /bin/sh
# hostname
nginx-pod
#
```

Kubernetes
Behind the Scenes



Kubernetes

Creating Pods: Imperative Way

- As mentioned earlier, each pod gets its own IP address, but this address is internal to the cluster and isn't accessible from outside of it.
- To make the pod accessible from the outside, you'll expose it through a Service object(to be discussed later)
- From within the cluster, we can curl the IP of the Pod

```
curl 10.244.2.75
```

```
root@k8s-master:/home/osboxes# curl 10.244.2.75
<!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>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Kubernetes

Creating Pods: Declarative Way

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


```
root@k8s-master:/home/osboxes/kubernetes# kubectl apply -f pod-definition.yml
pod/nginx-pod created
root@k8s-master:/home/osboxes/kubernetes# kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
nginx-pod   1/1     Running   0          6s
root@k8s-master:/home/osboxes/kubernetes# kubectl describe pod nginx-pod
Name:         nginx-pod
Namespace:    default
Priority:     0
Node:         k8s-slave02/192.168.0.103
Start Time:   Fri, 05 Feb 2021 06:52:33 -0500
Labels:       app=myapp
              env=dev
              project=iot
              region=asia
Annotations:  <none>
Status:       Running
IP:           10.244.2.77
IPs:
  IP:  10.244.2.77
```

Terminal

```
$ cat pod-definition.yml
  apiVersion: v1
  kind: Pod
  metadata:
    name: nginx-pod
    labels:
      app: myapp
      env: dev
      project: iot
      region: asia
  spec:
    containers:
    - name: nginx-container
      image: nginx
      ports:
      - containerPort: 80
```

```
# 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

Kubernetes

Creating Pods: Declarative Way

- We can directly generate the Manifest file from the imperative commands by using dry run and printing the output in yaml format
- Edit the YAML file as required and deploy it using `kubectl create` or `apply`

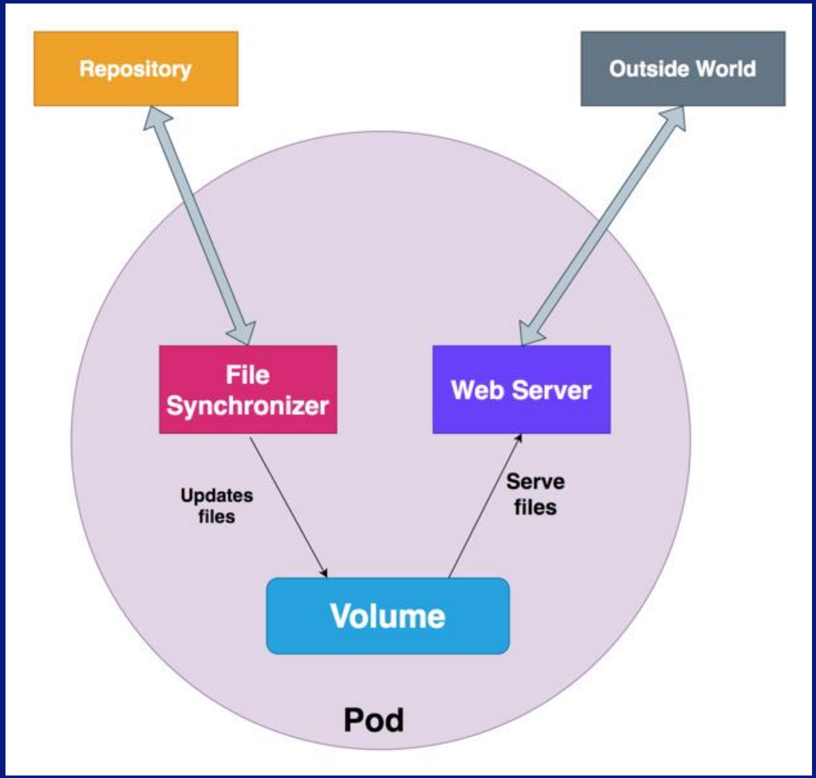
```
kubectl run nginx-pod --image=nginx --port=80 --dry-run=client -o yaml > pod-manifest.yml (or)
kubectl run nginx-pod --image=nginx --port=80 --dry-run=client -o yaml | tee pod-manifest.yml
```

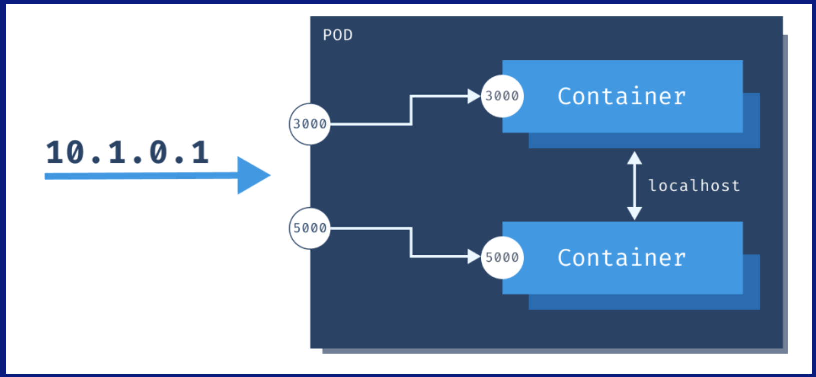
```
root@k8s-master:/home/osboxes/kubernetes# kubectl run nginx-pod --image=nginx --port=80 --dry-run=client -o yaml > pod-manifest.yml
root@k8s-master:/home/osboxes/kubernetes# cat pod-manifest.yml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx-pod
  name: nginx-pod
spec:
  containers:
  - image: nginx
    name: nginx-pod
    ports:
    - containerPort: 80
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
root@k8s-master:/home/osboxes/kubernetes#
```

Kubernetes

Multicontainer Pods

- The multi-container pods are the pods that contain two or more related containers that share resources like network namespace, shared volumes, Process namespace and work together as a single unit
- Since all the containers inside pod share the same network space, they can easily communicate on the localhost
- With shared process namespace, containers inside the pod can signal with each other. For this to be enabled, we need to have this setting shareProcessNamespace to true in the pod spec
- All the containers can have the same volume mounted so that they can communicate with each other by reading and modifying files in the storage volume. For example, there can be a helper container that pulls the files from the remote repo and updates the storage volume and the main container, which is a web server, serves those static files from the same storage to the outside world





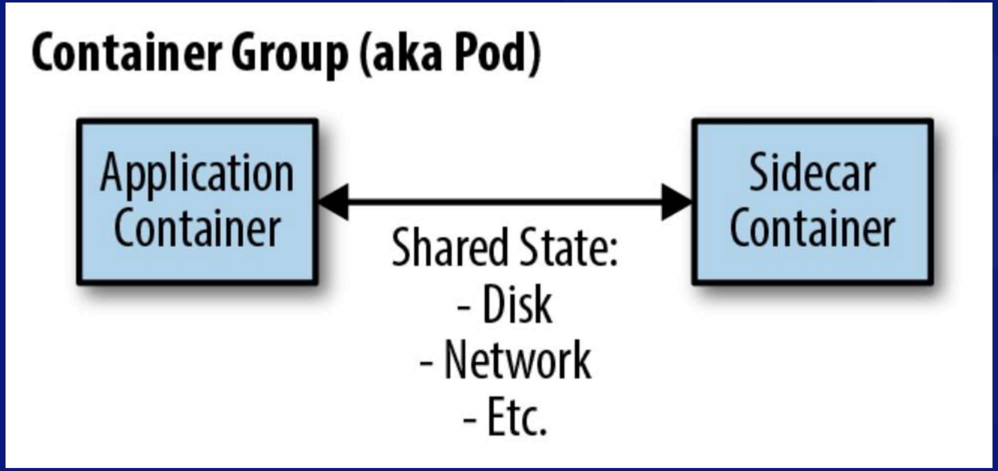
Kubernetes

Multicontainer Design Patterns

- There are several advantages with multiple containers inside a pod. The commonly used multi-container design patterns include Sidecar pattern and Ambassador/proxy pattern

Sidecar pattern

- The sidecar pattern is a single-pod pattern made up of two containers. The first is the application container. It contains the core logic for the application. Without this container, the application would not exist
- In addition to the application container, there is a sidecar container. The role of the sidecar is to augment and improve the application container, often without the application container's knowledge.
- In its simplest form, a sidecar container can be used to add functionality to the main container that might otherwise be difficult to improve. Ex: Monitoring, logging, configuration, etc



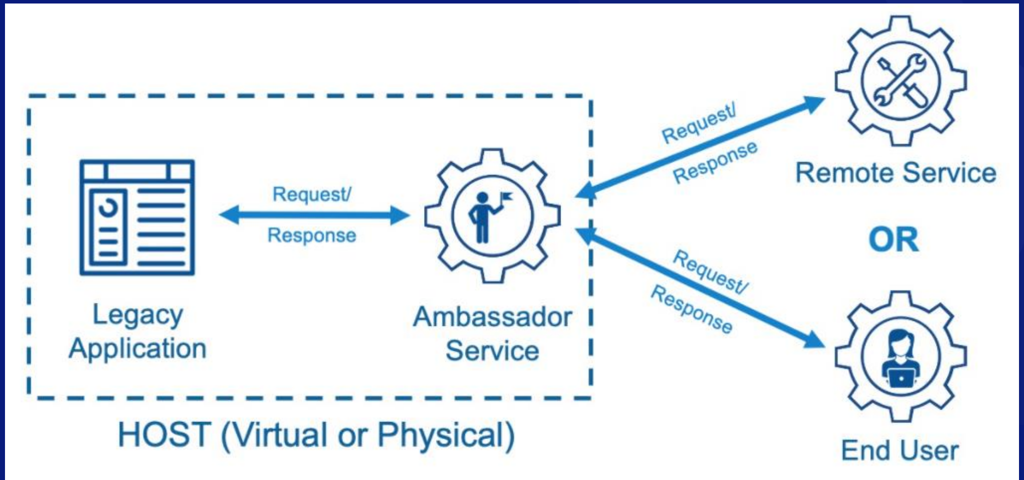
Kubernetes

Multicontainer Design Patterns

- There are many advantages with multiple containers inside a pod. The commonly used multi-container design patterns include Sidecar pattern Ambassador/proxy pattern

Ambassador pattern

- In this pattern, the helper service can send network requests on behalf of the main application. This is considered as a proxy server that can be co-located with the main application.
- This pattern can be useful for offloading common client connectivity tasks such as monitoring, logging, routing, security (such as TLS), and resiliency patterns in a language agnostic way. It is often used with legacy applications, or other applications that are difficult to modify, in order to extend their networking capabilities.
- It can also enable a specialized team to implement those features.



Kubernetes

Multicontainer Pods

- Pod with 2 containers can share same volume space(to be discussed in the volumes section)

```
k apply -f multicontainer-pod-01.yml
k logs -f pod/multi-container-pod -c ubuntu-container-01
k logs -f pod/multi-container-pod -c ubuntu-container-02
```



```
root@k8s-master:/home/osboxes/kubernetes# k apply -f multicontainer-pod-01.yml
pod/multi-container-pod created
root@k8s-master:/home/osboxes/kubernetes# k logs -f pod/multi-container-pod -c ubuntu-container-01
hello from container 1 of host multi-container-pod
hello from container 1 of host multi-container-pod
hello from container 1 of host multi-container-pod
hello from container 1 of host multi-container-pod
hello from container 1 of host multi-container-pod
hello from container 1 of host multi-container-pod
hello from container 1 of host multi-container-pod
hello from container 1 of host multi-container-pod
^C
root@k8s-master:/home/osboxes/kubernetes# k logs -f pod/multi-container-pod -c ubuntu-container-02
hello from container 2 of host multi-container-pod
hello from container 2 of host multi-container-pod
hello from container 2 of host multi-container-pod
hello from container 2 of host multi-container-pod
hello from container 2 of host multi-container-pod
hello from container 2 of host multi-container-pod
hello from container 2 of host multi-container-pod
hello from container 2 of host multi-container-pod
^Xhello from container 2 of host multi-container-pod
^C
root@k8s-master:/home/osboxes/kubernetes#
```

Terminal

```
$ cat multicontainer-pod-01.yml

apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod
spec:
  containers:
  - name: ubuntu-container-01
    image: ubuntu
    command: ["/bin/sh"]
    args: ["c", "while true; do echo hello
        from container 1 of host `hostname`
        ; sleep 5;done"]
  - name: ubuntu-container-02
    image: ubuntu
    command: ["/bin/sh"]
    args: ["c", "while true; do echo hello
        from container 2 of host `hostname`
        ; sleep 5;done"]
```

Kubernetes

Run a temp pod

- You can also fire up an interactive Pod within a Kubernetes cluster that is deleted once you exit the interactive session
- `--rm` ensures that the pod is deleted when you exit the interactive shell
- If you omit --rm, you should delete the pod manually with `kubectl delete pod/<pod-name>`
- If you want to detach from the shell and leave it running with the ability to re-attach, omit the --rm. You will then be able to reattach with: `kubectl attach $pod-name -c $pod-container -i -t` after you exit the shell
- -i/--tty: The combination of these two are what allows us to attach to an interactive session
- --: Delimits the end of the kubectl run options from the positional arg (bash)
- bash: Overrides the container's CMD. In this case, we want to launch bash as our container's command


```
kubectl run test-pod --image=alpine --rm -it
kubectl run my-shell --rm -it --image ubuntu -- bash
kubectl run tmp-shell --rm -i --tty --image centos -- /bin/bash
```

```
root@k8s-master:/home/osboxes# kubectl run test-pod --image=alpine --rm -it
If you don't see a command prompt, try pressing enter.
/ # wget -qO- www.nginx.com
<!DOCTYPE html>
<html lang="en-US">
<head>

  <script>
  //some global vars. DO NOT change these variables names. These variables are being used in GTM.
  var NX_GDPR_FUNCTIONAL_COOKIE_CONSENT = NX_GDPR_FUNCTIONAL_COOKIE_CONSENT || "no"; // possible values are 'yes' and 'no'
  var NX_GDPR_SOCIAL_COOKIE_CONSENT = NX_GDPR_SOCIAL_COOKIE_CONSENT || "no"; // possible values are 'yes' and 'no'
  //end global vars
  </script>
```

In the above example, wget will quietly (flag -q) download and output the content of URL to stdout (flag -O -)
We can also use options like timeout `wget --timeout=10 -q -O- www.google.com`

Kubernetes

Interacting with running Pods

• 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 run -i --tty busybox --image=busybox -- sh	# Run pod as interactive shell
• kubectl run nginx --image=nginx -n mynamespace	# Run pod nginx in a specific namespace
• kubectl attach my-pod -i	# Attach to Running Container
• kubectl exec my-pod -- ls /	# Run command in existing pod (1 container case)
• kubectl exec --stdin --tty my-pod -- /bin/sh	# Interactive shell access to a running pod (1 container case)
• kubectl exec my-pod -c my-container -- ls /	# Run command in existing pod (multi-container case)

Kubernetes

Init Containers

- Init container must complete successfully before the next one starts
- In this case, the init container counts from 10 to 0 before the main container executes its process

```
k apply -f initcontainers-03.yml
k logs -f pod/init-containers-pod -c busybox-int
k logs -f pod/init-containers-pod -c busybox-main
```



```
root@k8s-master:/home/osboxes/kubernetes# k apply -f ic.yml
pod/init-containers-pod created
root@k8s-master:/home/osboxes/kubernetes# k logs -f pod/init-containers-pod -c busybox-int
10
9
8
7
6
5
4
3
2
1
0
root@k8s-master:/home/osboxes/kubernetes# k logs -f pod/init-containers-pod -c busybox-main
hello from main container
^C
root@k8s-master:/home/osboxes/kubernetes# kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
init-containers-pod   1/1     Running   0          28s
root@k8s-master:/home/osboxes/kubernetes#
```

https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-initialization/#create-a-pod-that-has-an-init-container

Terminal

```
$ cat initcontainers-03.yml
  apiVersion: v1
  kind: Pod
  metadata:
    name: init-containers-pod
  spec:
    containers:
    - name: busybox-main
      image: busybox
      command: ["/bin/sh"]
      args: ["-c", "echo 'hello from main
           container' ; sleep 10m"]

    # This containers counts from 10 to 1
    initContainers:
    - name: busybox-int
      image: busybox
      command: ["/bin/sh"]
      args: ["-c", 'i=10; while [ $i -ge 0 ]; do
           echo $i; sleep 0.5; i=$(expr $i - 1);
           done']
```

Kubernetes

Kubectl shortcuts

- alias k=kubectl
  - k version --short
- export do="--dry-run=client -o yaml"
  - k run pod1 --image=nginx $do
- Watch the cluster in Realtime: It prints out the current context, namespace and then all possible Kubernetes objects
  - watch -n 0.5 "kubectl config current-context; echo ''; kubectl config view | grep namespace; echo ''; kubectl get namespace,node,ingress,pod,svc,job,cronjob,deployment,rs,pv,pvc,secret,ep -o wide"

UP NEXT

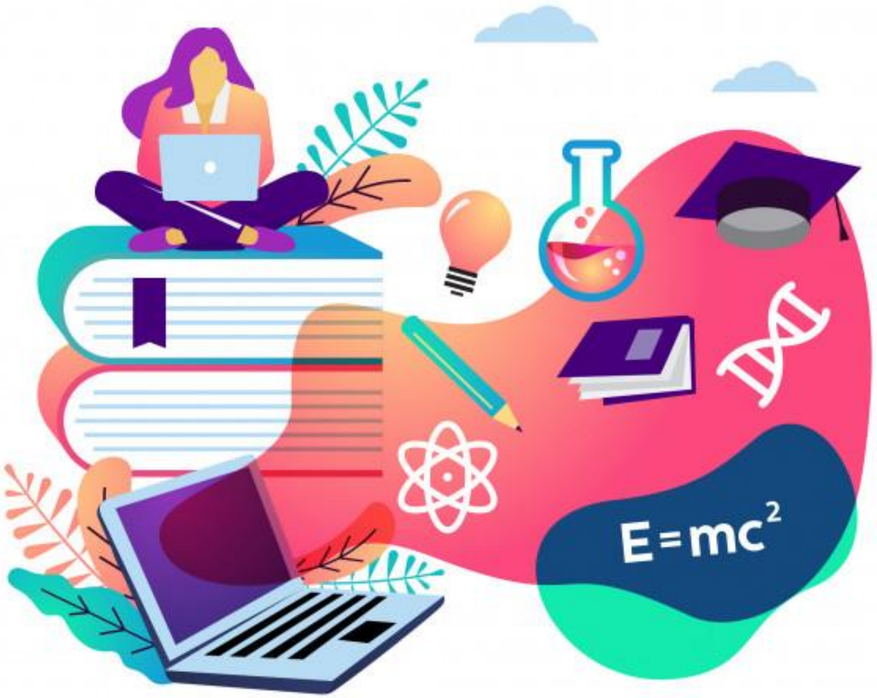
ReplicaSets, Deployments and Services

Subscribe to my Facebook page:
https://www.facebook.com/vikram.devops
and join my group:
https://www.facebook.com/groups/171043094400359
for all latest updates on DevOps, Python and IoT

https://www.youtube.com/channel/UCE1cGZfooxT7-VbqVbuKjMg

References

- 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/
  - https://dev.to/inductor/wait-docker-is-deprecated-in-kubernetes-now-what-do-i-do-e4m
- Kubeadm
  - https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/
  - https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/
- 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/



Q&A

git:kunchalavikram1427


Thank You