**Kubernetes Hands-on Lab Guide**

**Lab Objectives**

* Get familiar with kubectl basic commands.
* Explore nodes, pods, and system components in a Kubernetes cluster.
* Work with namespaces.
* Compare Kubernetes pods with Docker containers.
* Understand how to list API resources.

**1. Environment Setup**

Before starting:

* A Kubernetes cluster should be running (can be Minikube, kind, kubeadm, or managed cluster like AKS/EKS/GKE).
* kubectl installed and configured (kubectl get nodes should return results).
* Alias set for convenience:

alias k=kubectl

**2. Checking Cluster Nodes**

To verify that the cluster is up:

k get nodes

This lists all worker and master nodes with status, roles, and Kubernetes version.

To get detailed info about a specific node (e.g., w1):

k describe node w1

This shows capacity (CPU, memory), allocated resources, labels, taints, and running pods.

**3. Understanding Pods**

Pods are the smallest deployable unit in Kubernetes.

* List pods in the **default namespace**:

k get pods

* List pods across **all namespaces**:

k get pods -A

* Get pods with more details (node name, IP, etc.):

k get pods -A -o wide

* Focus on pods running on master/control-plane nodes:

k get pods -A -o wide | grep -i master

**4. Kubernetes API Resources**

Kubernetes has many resource types (pods, services, deployments, configmaps, etc.).

Check which resource types the cluster supports:

k api-resources

This helps you understand the scope of Kubernetes objects available.

**5. Comparing with Docker**

Since Kubernetes runs containers under the hood, you can check which containers are running on the host:

docker ps

This lists active containers on the node. You’ll notice a mapping between Kubernetes pods and Docker containers.

**6. System Pods**

The **kube-system namespace** contains Kubernetes control-plane and infrastructure components.

List them:

k get pods -n kube-system

Get detailed view (node placement, IPs, etc.):

k get pods -n kube-system -o wide

You’ll see critical components like:

* etcd (key-value store)
* kube-apiserver
* kube-controller-manager
* kube-scheduler
* coredns
* kube-proxy

**7. Working with Namespaces**

Namespaces allow logical isolation inside a cluster.

* Create a new namespace:

k create ns raman

* Verify namespaces:

k get ns

Now you can deploy workloads inside raman namespace without interfering with other workloads.

**8. Cleanup (Optional)**

If you no longer need the namespace:

k delete ns raman

**Recap & Learning Points**

1. Nodes = Machines in the cluster.
2. Pods = Smallest deployable unit (one or more containers).
3. Namespaces = Logical partitions for organizing resources.
4. k api-resources helps you discover what’s supported in your cluster.
5. kube-system namespace = critical system components.
6. Docker shows you the actual containers behind Kubernetes pods.

**Kubernetes Lab Guide – Running and Exploring Pods with Custom Images**

**Lab Objectives**

* Understand how application code becomes a running container in Kubernetes.
* Differentiate between Docker’s role (building images) and Kubernetes’ role (running containers).
* Create and manage pods in custom namespaces.
* Execute commands inside pods for hands-on exploration.

**1. Docker Images vs Kubernetes Pods**

**Important concept first**:

* **Docker** is for **building images** (Dockerfile → docker build → docker push to registry).
* **Kubernetes** does **not build images**. It pulls an already-built image from a registry (DockerHub, ECR, ACR, GCR, etc.) and runs it inside a **pod**.
* A **pod** is a wrapper around one or more containers, providing networking, storage, and lifecycle management.
* Kubernetes provides **auto-healing**: if the container inside a pod crashes, Kubernetes will restart it automatically.

**2. Exploring Resources Again**

k api-resources

Lists all resource types supported in the cluster (pods, deployments, configmaps, etc.).

Check all running pods across namespaces:

k get pods -A

**3. Running Your First Pod**

Run a pod with an Apache HTTP server image (httpd) in the **default namespace**:

k run raman-pod --image httpd

Verify:

k get pods

See across namespaces:

k get pods -A

Delete it (demonstrating cleanup):

k delete pod raman-pod

**4. Running a Pod in a Custom Namespace**

Let’s isolate workloads in the raman namespace.

Create a pod inside namespace raman:

k run raman-pod --image httpd -n raman

Verify in default namespace:

k get pods

Check in raman namespace:

k get pods -n raman

Or across all namespaces:

k get pods -A

With more details:

k get pods -A -o wide

**5. Inspecting the Pod**

Get detailed info:

k describe pod raman-pod -n raman

This shows:

* Container image being used.
* Events (pulling image, container start).
* Pod IP and assigned node.
* Restart count (helps observe auto-healing).

**6. Getting Into the Pod**

Pods aren’t just abstract objects — they’re running Linux containers you can shell into.

* Open an interactive bash session:

k exec -it raman-pod -n raman -- /bin/bash

* Run a simple command (without entering full shell):

k exec -it raman-pod -n raman -- ls

* Create a directory inside the container:

k exec -it raman-pod -n raman -- mkdir raman

* Verify it was created:

k exec -it raman-pod -n raman -- ls -ltr

This shows how Kubernetes doesn’t just schedule pods; it also lets you interact directly with the container runtime.

**7. Key Learning Points**

1. **Images are prebuilt**: Kubernetes pulls them; it does not build them.
2. **Pod = wrapper for containers**: provides lifecycle and networking.
3. **Auto-healing**: if a container dies, Kubernetes restarts it automatically.
4. **Namespaces**: isolate workloads for organization and multi-tenancy.
5. **kubectl exec**: interact with containers inside pods, just like SSH into VMs, but at the container level.

**Kubernetes Lab Guide – Running Custom Images in Pods**

**1. The Big Picture: Docker vs Kubernetes**

Before touching commands, internalize this workflow:

Application Code >>> Dockerfile >>> docker build . >>> Custom Image >>> Push to Registry

↓

Kubernetes pulls image → Pod → Running Container

* **Docker**: builds images (docker build) and stores/pushes them (to DockerHub, ACR, ECR, etc.).
* **Kubernetes**: does not build images; it consumes them. It schedules containers inside **pods**.
* **Auto-healing**: If a container inside a pod crashes, Kubernetes restarts it automatically.

**2. Exploring Kubernetes Resources**

Check what kinds of resources Kubernetes can manage:

k api-resources

**3. Checking Running Pods (All Namespaces)**

k get pods -A

This lists pods across all namespaces (default, kube-system, custom namespaces, etc.).

**4. Running a Pod in Default Namespace**

Let’s run an Apache HTTP Server (httpd) pod:

k run raman-pod --image httpd

Verify:

k get pods

Look cluster-wide:

k get pods -A

**5. Deleting the Pod**

Remove the pod to practice cleanup:

k delete pod raman-pod

Check again:

k get pods -A

**6. Running a Pod in a Custom Namespace**

Use the previously created namespace raman to keep things organized:

k run raman-pod --image httpd -n raman

Verify pod placement:

k get pods

k get pods -n raman

k get pods -A

Get details (IP, node placement, restarts, etc.):

k get pods -A -o wide

**7. Inspecting the Pod**

Dive into metadata and lifecycle events:

k describe pod raman-pod -n raman

You’ll see:

* Image pulled (httpd)
* Container created
* Events (scheduled, started, etc.)
* Pod IP, node name

**8. Interacting With the Pod**

**Entering a Shell**

k exec -it raman-pod -n raman -- /bin/bash

**Running Commands Without Full Shell**

* List contents:

k exec -it raman-pod -n raman -- ls

* Create a directory inside container:

k exec -it raman-pod -n raman -- mkdir raman

* Confirm directory creation:

k exec -it raman-pod -n raman -- ls

k exec -it raman-pod -n raman -- ls -ltr

This shows the pod is a live Linux container, fully interactive.

**9. Key Learning Points**

1. **Kubernetes runs images, not build them** – you must bring a custom image already built and pushed.
2. **Pod is a wrapper** around containers, with IP, lifecycle, and auto-healing built in.
3. **Namespaces** isolate workloads logically.
4. **kubectl exec** lets you work *inside* running containers just like SSH.
5. **Auto-healing** is visible if you manually kill the main process (kill -9 1 inside pod) and watch Kubernetes restart it.

**Kubernetes Lab Guide – Defining and Running Pods with YAML**

**1. The Application Flow**

Think of the lifecycle:

Code (App + DB interaction)

↓ Dockerfile (copy, build context)

↓ docker build → Custom Image

↓ Push to Docker Registry (DockerHub, ECR, ACR…)

↓ Kubernetes pulls image → Pod → Running Container(s)

* Kubernetes does **not** build images; it only runs them.
* You must have an image available in a registry before referencing it in YAML.

**2. Checking Pods Across Namespaces**

Before creating anything, inspect cluster state:

k get pods -A

**3. Writing Pod Definition YAML**

Create a file pod.yml:

apiVersion: v1

kind: Pod

metadata:

name: pavani-nginx

namespace: pavani

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

Key parts:

* **apiVersion**: The API group/version (v1 for core objects like Pod).
* **kind**: Resource type (Pod).
* **metadata**: Name + namespace (namespace must exist beforehand).
* **spec.containers**: List of containers in the pod, with image and ports.

**4. Creating the Pod**

Apply YAML manifest:

k create -f pod.yml

Check pod status:

k get pods -n pavani

With detailed placement info:

k get pods -n pavani -o wide

**5. Testing Pod Functionality**

Find pod IP (from -o wide or describe). Then curl the service:

curl <Pod-IP>:80

Since no Service is created yet, this IP is cluster-internal. Curl from inside cluster nodes (like master or worker node terminal).

**6. Deleting and Reapplying Pods**

Practice cleanup:

k delete -f pod.yml

Recreate with modified YAML:

k create -f pod.yml

Check again:

k get pods -n pavani

**7. Inspecting Pod Details**

Describe pod for events and state:

k describe pod pavani-nginx -n pavani

This shows scheduling events, image pull logs, container restarts, etc.

**8. Multi-Container Pod Exploration**

If your pod YAML defines multiple containers (example: nginx + redis in same pod), you can target specific containers:

k exec -it raman-nginx -c nginx -n raman -- /bin/bash

k exec -it raman-nginx -c redis -n raman -- /bin/bash

Explanation:

* -c specifies which container to exec into.
* Useful when a pod has sidecars (e.g., logging agent, database proxy).

**9. Pod Lifecycle Practice**

* **List history** of created pods:
* k get pods -n pavani -o wide
* **Delete all pods in namespace**:
* k delete pods -n pavani --all

**10. Key Learning Points**

1. **Declarative YAML > Imperative run**: YAML ensures repeatability and version control.
2. **Pod IPs are ephemeral**: restart the pod, and it gets a new IP. That’s why Services exist (stable networking).
3. **Multi-container pods**: useful when containers must share lifecycle/storage/network.
4. **Namespaces**: always match your YAML namespace with what exists in the cluster.
5. **kubectl describe and kubectl exec**: vital for debugging and exploration.

**Kubernetes Lab Guide – Deployments, ReplicaSets, Labels & Selectors**

**1. Why Deployment instead of Pods?**

* Running individual pods (kubectl run ramanapp1, ramanapp2 … ramanapp50) quickly becomes unmanageable.
* You need something that:
  + **Scales**: create tens/hundreds of pods together.
  + **Heals**: if one pod crashes, another is created.
  + **Manages lifecycle**: upgrades, rollbacks, sync with desired state.
* That’s where **ReplicaSets** and **Deployments** enter.

**2. Key Concepts**

* **Label**: Key-value pair attached to objects (app=nginx, env=prod).
* **Selector**: Query that matches labels (Deployment uses selectors to find/manage pods).
* **ReplicaSet**: Ensures a specified number of pod replicas are always running.
* **Deployment**: Manages ReplicaSets, provides rolling updates, rollbacks, scaling.

Think of it like this:

Deployment → manages ReplicaSet(s) → manages Pods

**3. Defining a Deployment Manifest**

Here’s your deploy.yaml (cleaned and annotated):

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-nginx-deployment

namespace: raman

labels:

purpose: demo

spec:

replicas: 3 # Desired number of pods

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx # Must match selector

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

Notes:

* replicas: 3 means Kubernetes will *always* maintain 3 nginx pods.
* Labels (app: nginx) let the Deployment know which pods it controls.
* Template defines the pod blueprint (containers, image, ports).

**4. Creating the Deployment**

k create -f deploy.yaml

Verify deployment object:

k get deploy -n raman

List pods it spawned:

k get pods -n raman

Show details with nodes/IPs:

k get pods -n raman -o wide

**5. ReplicaSet in Action**

Every Deployment creates a ReplicaSet behind the scenes:

k get rs -n raman

Delete one of the pods:

k delete pod <pod-name> -n raman

Check again:

k get pods -n raman -o wide

Kubernetes automatically creates a new pod to maintain the replica count.  
That’s **auto-healing** at the ReplicaSet level.

**6. Scaling the Deployment**

Update replica count in deploy.yaml or use command:

k scale deploy raman-nginx-deployment -n raman --replicas=10

Check pods:

k get pods -n raman

**7. Labels and Selectors in Practice**

Use labels to query pods:

k get pods --selector "app=nginx" -A

This shows only the pods with app=nginx.  
Labels make grouping and selecting pods easy (per app, team, or environment).

**8. Updating and Applying Changes**

If you edit the manifest (vi deploy.yaml):

k apply -f deploy.yaml

This updates the Deployment. Kubernetes does a **rolling update**: gradually replacing old pods with new ones, maintaining availability.

**9. Deployment Lifecycle Management**

* **Check Deployment**:
* k describe deploy raman-nginx-deployment -n raman
* **Check ReplicaSets**:
* k get rs -n raman
* **Rollback if needed**:
* k rollout undo deploy raman-nginx-deployment -n raman

**10. Key Learning Points**

1. **Deployment > ReplicaSet > Pod hierarchy**.
2. **Labels & Selectors** tie everything together.
3. **Scaling is declarative**: just update replicas, Kubernetes does the rest.
4. **Self-healing**: delete pods → they respawn automatically.
5. **Rolling updates**: Deployments let you upgrade images without downtime.

**Kubernetes Lab Guide – Exposing Deployments with Services**

**1. The Problem: Pod IPs are Ephemeral**

* Pods get a new IP every time they restart or are rescheduled.
* You cannot rely on pod IPs for client connections.
* Solution: **Kubernetes Service** = stable network endpoint + load balancer across pods.

**2. Types of Services**

* **ClusterIP** (default): Exposes the app inside the cluster only. Other pods/namespaces can talk to it, but not external users.
* **NodePort**: Exposes the app on a static port (between 30000–32767) on every node in the cluster. Accessible from outside via <NodeIP>:<NodePort>.
* **LoadBalancer**: For cloud providers (EKS, AKS, GKE), provisions an external LB.

In this lab you’ve defined a **NodePort** service.

**3. The Deployment Manifest**

Here’s your deploy.yaml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-deployment

namespace: raman

labels:

test: rk

spec:

replicas: 2 # two pods

selector:

matchLabels:

test: raman

template:

metadata:

labels:

test: raman

spec:

containers:

- name: raman-con

image: ramann123/hotstar:latest

ports:

- containerPort: 3000

**Key Points**

* Runs **2 replicas** of your custom image ramann123/hotstar:latest.
* Pods get the label test: raman.
* Container listens on port 3000.

**4. The Service Manifest**

Here’s your service.yaml:

apiVersion: v1

kind: Service

metadata:

name: raman-service

namespace: raman

spec:

type: NodePort

selector:

test: raman

ports:

- port: 3000 # stable Service port

targetPort: 3000 # container port inside pods

# nodePort: 30007 # optional, if omitted k8s auto-assigns

**Key Points**

* Type = NodePort → accessible externally.
* Selector test: raman matches the Deployment’s pod labels.
* Traffic flow:
* NodeIP:NodePort → Service → one of the raman pods (load-balanced)

**5. Creating Deployment and Service**

k create -f deploy.yaml

k create -f service.yaml

**6. Verifying**

Check pods:

k get pods -n raman -o wide

Check service:

k get svc -n raman

Describe service for NodePort details:

k describe svc raman-service -n raman

Output shows:

* ClusterIP (internal).
* NodePort (external, e.g., 30007).
* Endpoints (pod IPs being load-balanced).

**7. Testing Access**

Find your cluster node’s IP (kubectl get nodes -o wide or ifconfig/ip a on VM).  
Then test from outside (browser/curl):

curl http://<NodeIP>:<NodePort>

Example:

curl http://192.168.80.201:30007

**8. Auto-Healing in Action**

Delete one pod:

k delete pod <pod-name> -n raman

Watch pods again:

k get pods -n raman -o wide

Deployment + ReplicaSet will **immediately recreate** a pod, and Service keeps routing without downtime.

**9. ClusterIP Variant (Internal Only)**

If you change service type to ClusterIP:

spec:

type: ClusterIP

Then the app is reachable only *inside the cluster* (e.g., from another pod):

curl http://raman-service.raman.svc.cluster.local:3000

**10. Key Learning Points**

1. **Deployment = scale + self-heal pods**.
2. **Service = stable network endpoint** decoupled from ephemeral pod IPs.
3. **NodePort** makes apps accessible externally on each node.
4. **ClusterIP** is for internal-only communication between services.
5. Label–selector mechanism is the glue connecting Services → Pods → Deployments.

**11. Suggested Next Step**

* Try scaling your deployment:
* k scale deploy raman-deployment -n raman --replicas=5

Then re-check Service endpoints with:

k describe svc raman-service -n raman

* Explore **rolling updates**: update image: in deploy.yaml, run kubectl apply -f, and watch pods update one by one while service remains available.