**Kubernetes Hands-on Lab: Working with kubectl, Pods, and Namespaces**

**🧭 Lab Overview**

This lab introduces essential Kubernetes administration and troubleshooting operations using the kubectl CLI. You will:

* Explore cluster nodes and system pods
* Create and manage user-defined namespaces
* Deploy containerized workloads (Pods)
* Inspect Pod and Node details
* Understand the logical separation of workloads using namespaces

**⚙️ Pre-requisites**

* Kubernetes cluster is running (either local minikube/kind or cloud-based cluster like GKE/EKS/AKS).
* You have kubectl installed and configured to communicate with your cluster (kubectl config view should show context).
* Docker or container runtime (e.g., containerd) should be active on cluster nodes.

**🧠 Concepts Refresher**

Before you start executing, ensure understanding of the following:

| **Concept** | **Description** |
| --- | --- |
| **kubectl** | Command-line tool to interact with the Kubernetes API server. |
| **Node** | A worker machine in Kubernetes that runs Pods. |
| **Pod** | Smallest deployable unit in Kubernetes – abstraction over a container. |
| **Namespace** | Logical partition within a cluster that isolates workloads/resources. |
| **System Namespace** | Reserved Kubernetes namespaces such as kube-system for cluster components. |

**🧩 Step-by-Step Lab Guide**

**🧱 Step 1: Create Alias for kubectl**

alias k=kubectl

✅ **Explanation:**  
This creates a command alias k so that typing k executes kubectl. It improves command efficiency.

You can confirm alias creation:

alias

**🔍 Step 2: List Cluster Nodes**

k get nodes

✅ **Explanation:**  
Lists all worker and master nodes registered with the control plane.

🔹 **Expected Output Example:**

NAME STATUS ROLES AGE VERSION

minikube Ready control-plane 1h v1.29.0

**🧩 Step 3: Explore Pods Across Namespaces**

**Default namespace pods:**

k get pods

If no pods exist, output will be empty.

**All namespaces:**

k get pods -A

✅ **Explanation:**  
-A (or --all-namespaces) shows pods from all namespaces, including system pods.

🔹 **Sample Output:**

NAMESPACE NAME READY STATUS RESTARTS AGE

kube-system coredns-558bd4d5db-vl2p7 1/1 Running 0 10m

kube-system etcd-minikube 1/1 Running 0 10m

default nginx-deployment-6d4cf56db6-8tgz8 1/1 Running 0 5m

**⚙️ Step 4: Explore System Namespace**

kubectl get pods -n kube-system

✅ **Explanation:**  
-n specifies a namespace. kube-system contains internal components like:

* **kube-proxy**
* **CoreDNS**
* **etcd**
* **kube-scheduler**

🔹 **Expected Output:**

NAME READY STATUS RESTARTS AGE

coredns-558bd4d5db-vl2p7 1/1 Running 0 10m

kube-proxy-wxkq4 1/1 Running 0 10m

**🚀 Step 5: Run a Simple Pod in Default Namespace**

kubectl run ramanapp --image=nginx

✅ **Explanation:**  
Creates a **Pod named ramanapp** using the **nginx** container image (default namespace).

🔹 **What Happens Internally:**

* The Kubernetes API schedules the Pod to a node.
* The kubelet pulls the nginx image from Docker Hub.
* Container starts running inside the Pod.

🔹 **Verify Pod Status:**

kubectl get pods

🔹 **Expected Output:**

NAME READY STATUS RESTARTS AGE

ramanapp 1/1 Running 0 20s

**🧩 Step 6: View Pod Details (Wide Output)**

kubectl get pods -o wide

✅ **Explanation:**  
The -o wide flag adds extra information like **Node name**, **Pod IP**, and **Container Image**.

🔹 **Expected Output:**

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

ramanapp 1/1 Running 0 1m 10.244.0.5 minikube <none> <none>

**🧩 Step 7: Verify All Pods Across Namespaces Again**

kubectl get pods -A -o wide

✅ **Purpose:**  
Confirms your newly created Pod appears only in the **default namespace**, while system components remain under **kube-system**.

**⚙️ Step 8: Create a Custom Namespace**

kubectl create ns raman

✅ **Explanation:**  
Creates a user-defined namespace called raman for logical separation of workloads.

🔹 **Check All Namespaces:**

kubectl get ns

🔹 **Expected Output:**

NAME STATUS AGE

default Active 15m

kube-system Active 15m

raman Active 5s

**🚀 Step 9: Deploy Pod into Custom Namespace**

kubectl run ramanapp2 --image=nginx -n raman

✅ **Explanation:**  
Creates a new Pod (ramanapp2) inside the raman namespace.

🔹 **Verify in Namespace:**

kubectl get pods -n raman

🔹 **Expected Output:**

NAME READY STATUS RESTARTS AGE

ramanapp2 1/1 Running 0 10s

**🧭 Step 10: Explore All Namespaces & Pods**

k get pods -A

✅ **You’ll observe:**

* Default namespace → ramanapp
* raman namespace → ramanapp2
* kube-system namespace → system pods

🔹 **Output:**

NAMESPACE NAME READY STATUS RESTARTS AGE

default ramanapp 1/1 Running 0 10m

raman ramanapp2 1/1 Running 0 2m

kube-system coredns... 1/1 Running 0 15m

**🔍 Step 11: Describe and Inspect Pods in Detail**

k describe pod ramanapp2 -n raman

✅ **Explanation:**  
Displays deep details about Pod creation, scheduling, image pulling, networking, and event logs.

🔹 **Focus On:**

* Events at the bottom (image pulled, started)
* Node scheduling info
* Container ports and image details

**🧩 Step 12: Clean Up (Optional)**

If you wish to clean up resources:

k delete pod ramanapp

k delete pod ramanapp2 -n raman

k delete ns raman

✅ **Explanation:**  
Removes the Pods and the namespace to revert cluster to its clean state.

**🧾 Lab Summary**

| **Concept** | **Command** | **Key Takeaway** |
| --- | --- | --- |
| Alias | alias k=kubectl | Shortcut for kubectl commands |
| Cluster Info | k get nodes | Verify node readiness |
| Pod Listing | k get pods -A | View all running pods |
| Namespace Mgmt | k create ns raman | Logical isolation |
| Pod Deployment | k run <pod> --image=<image> | Deploy container workloads |
| Inspection | k get pods -o wide / k describe | Check runtime & node mapping |

**🧠 Learning Validation (Discussion/Checkpoints)**

1. What is the difference between kubectl get pods and kubectl get pods -A?
2. Where do kube-proxy and CoreDNS run, and why?
3. How does a Pod get assigned to a node?
4. What happens internally when you execute kubectl run?
5. How are namespaces beneficial in multi-tenant clusters?

**🧰 Bonus Exercise**

1. Deploy another pod using a different image (e.g., busybox) inside the same namespace:
2. kubectl run testbox --image=busybox -n raman -- sleep 3600
3. Execute commands inside the container:
4. kubectl exec -it testbox -n raman -- sh
5. List all namespaces and delete the raman namespace once done.

**Kubernetes Lab: Declarative Pod Management, Logs, and Scaling Concepts**

**🧭 Lab Overview**

This lab focuses on **creating, inspecting, and managing Pods declaratively using YAML manifests**, exploring **API resources**, and understanding **pod replication and high availability concepts**.

You will:

1. Explore Kubernetes API resources.
2. Create a Pod using a YAML manifest.
3. View logs and describe Pods.
4. Delete and recreate Pods.
5. Understand how scaling and high availability are achieved with multiple Pods.
6. Learn about declarative vs. imperative approaches.

**⚙️ Pre-requisites**

* A working Kubernetes cluster.
* kubectl configured and tested (kubectl get nodes works).
* A custom namespace raman already exists.  
  If not, create one:
* kubectl create namespace raman

**🧠 Concepts Refresher**

| **Concept** | **Description** |
| --- | --- |
| **Declarative Configuration** | Define desired state in a YAML/JSON manifest; Kubernetes ensures actual state matches it. |
| **Imperative Command** | Directly instruct Kubernetes to perform an action (e.g., kubectl run). |
| **Pod Manifest** | YAML file describing a Pod’s metadata, containers, images, and networking details. |
| **Logs** | Output generated by containers, viewable via kubectl logs. |
| **Describe** | Provides deep-level debugging info, including events and conditions. |
| **Scaling** | Increasing or decreasing the number of Pod replicas to handle load or provide HA. |

**🧩 Step-by-Step Lab**

**🧱 Step 1: View All Available Kubernetes API Resources**

k api-resources

✅ **Explanation:**  
Lists all Kubernetes resource types (Pods, Services, Deployments, ReplicaSets, etc.) that the API server supports.

🔹 **Sample Output (Partial):**

NAME SHORTNAMES APIVERSION NAMESPACED KIND

pods po v1 true Pod

namespaces ns v1 false Namespace

deployments deploy apps/v1 true Deployment

replicasets rs apps/v1 true ReplicaSet

services svc v1 true Service

🔹 **Learning Point:**  
Every resource (Pod, Service, Deployment, etc.) has:

* **Kind** – what type of object it is.
* **API Version** – which API group/version defines it.
* **Namespaced?** – whether it belongs to a namespace or is cluster-wide.

**🧩 Step 2: Create a YAML Manifest File**

Create a file named pod.yaml:

apiVersion: v1

kind: Pod

metadata:

name: nginx-raman

namespace: raman

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

✅ **Explanation:**  
This manifest defines a single Pod named **nginx-raman** inside namespace **raman**.

| **Field** | **Description** |
| --- | --- |
| **apiVersion: v1** | Refers to the core Kubernetes API version. |
| **kind: Pod** | Resource type. |
| **metadata** | Holds identifiers like name and namespace. |
| **spec** | Desired state of the Pod. |
| **containers** | Defines one or more containers within this Pod. |
| **image** | Docker image to be pulled from registry. |
| **containerPort** | Exposed port inside container. |

**🚀 Step 3: Create Pod Declaratively**

k create -f pod.yaml

✅ **Explanation:**  
Creates resources from the file pod.yaml as defined in the manifest.

🔹 **Expected Output:**

pod/nginx-raman created

**🧩 Step 4: Verify Pod Creation**

k get pods -n raman

✅ **Explanation:**  
Lists Pods in the raman namespace to confirm creation.

🔹 **Expected Output:**

NAME READY STATUS RESTARTS AGE

nginx-raman 1/1 Running 0 25s

**🔍 Step 5: Describe the Pod**

k describe pod nginx-raman -n raman

✅ **Explanation:**  
Shows full details including:

* Container status
* Node assigned
* Image used
* Events (Scheduled, Pulled, Created, Started)

🔹 **Focus Points:**

* **Events section** confirms lifecycle:
* Normal Scheduled Successfully assigned raman/nginx-raman to node-name
* Normal Pulled Successfully pulled image "nginx:1.14.2"
* Normal Created Created container nginx
* Normal Started Started container nginx

**📜 Step 6: View Container Logs**

k logs nginx-raman -n raman

✅ **Explanation:**  
Displays output generated by the containerized process inside the Pod (for Nginx, typically startup logs).

🔹 **Expected Output:**

/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty

Starting nginx: nginx started successfully

**🧰 Step 7: View All Pods (Cluster-Wide)**

k get pods -A -o wide

✅ **Explanation:**  
-A = All namespaces  
-o wide = Shows extended info (Node, IP, Image, etc.)

🔹 **Sample Output:**

NAMESPACE NAME READY STATUS RESTARTS AGE IP NODE

kube-system coredns... 1/1 Running 0 25m 10.244.0.3 node1

raman nginx-raman 1/1 Running 0 3m 10.244.0.10 node1

**🧹 Step 8: Delete Pods**

**Option 1 – Delete all Pods in the namespace:**

k delete pods --all -n raman

**Option 2 – Delete specific Pod:**

k delete pod nginx-raman -n raman

✅ **Explanation:**  
Deletes the Pod(s) gracefully by signaling the container to terminate.

🔹 **Expected Output:**

pod "nginx-raman" deleted

**🧩 Step 9: Recreate Pod (Reapply Manifest)**

k create -f pod.yaml

✅ **Purpose:**  
Demonstrates **declarative idempotency** — applying the same file again ensures cluster state matches the desired configuration.

🔹 **Expected Output:**

pod/nginx-raman created

**🔄 Scaling and High Availability Concepts**

Now, let’s discuss the conceptual part you mentioned:

“10 pods of nginx … scale up, scale down, sync with each other, have pod-level high availability.”

These concepts move us from **Pod** ➜ **Deployment/ReplicaSet** level.

**🧩 Step 10: Moving to Multi-Pod (Scaling)**

Single Pod manifests are **not scalable** — once deleted, they don’t auto-recover.  
To achieve **scalability and high availability**, we use **Deployments**.

**🧱 Step 11: Create a Scalable Deployment**

Create a new file nginx-deploy.yaml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-ha

namespace: raman

spec:

replicas: 10

selector:

matchLabels:

app: nginx-ha

template:

metadata:

labels:

app: nginx-ha

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

✅ **Explanation:**  
This defines a **Deployment** managing 10 Pods of nginx.

| **Field** | **Description** |
| --- | --- |
| **replicas** | Number of Pod copies to run (scaling). |
| **selector** | How the Deployment identifies its Pods. |
| **template** | Pod blueprint — same as your earlier Pod spec. |

**🚀 Step 12: Apply the Deployment**

k apply -f nginx-deploy.yaml

🔹 **Expected Output:**

deployment.apps/nginx-ha created

**🔍 Step 13: Verify Pods**

k get pods -n raman -o wide

✅ **Output Example:**

NAME READY STATUS RESTARTS AGE IP NODE

nginx-ha-6d8b7b7b44-abc12 1/1 Running 0 1m 10.244.0.11 node1

nginx-ha-6d8b7b7b44-xyz90 1/1 Running 0 1m 10.244.0.12 node2

...

**🔁 Step 14: Scaling Up and Down**

**Scale Up to 15 Pods:**

k scale deployment nginx-ha -n raman --replicas=15

**Scale Down to 5 Pods:**

k scale deployment nginx-ha -n raman --replicas=5

✅ **Explanation:**  
The Deployment’s controller automatically creates or terminates Pods to maintain the specified replica count.

🔹 **Check Status:**

k get deployment nginx-ha -n raman

**💡 Step 15: High Availability Insight**

* Kubernetes scheduler spreads Pods across available nodes for **fault tolerance**.
* If one node fails, Pods reschedule on healthy nodes automatically.
* Scaling ensures workload load-balancing across nodes.
* Deployments maintain Pod **replica synchronization** through **ReplicaSets**.

**🧾 Lab Summary**

| **Step** | **Command** | **Purpose** |
| --- | --- | --- |
| k api-resources | List all supported resource types |  |
| k create -f pod.yaml | Create Pod declaratively |  |
| k logs | View container output |  |
| k describe | Inspect Pod lifecycle and events |  |
| k delete pods | Clean up |  |
| k apply -f nginx-deploy.yaml | Create scalable Deployment |  |
| k scale | Scale Pods up/down dynamically |  |

**🧠 Discussion Points**

1. Difference between **Pod** and **Deployment**?
2. What ensures high availability when a Pod crashes?
3. Why is declarative YAML preferred in production?
4. How does Kubernetes maintain desired replica count?
5. Can scaling affect existing traffic? (Hint: rolling updates)

**🧰 Optional Challenge**

1. Expose the nginx-ha deployment via a Service:
2. kubectl expose deployment nginx-ha --type=NodePort --port=80 -n raman
3. Access the Nginx service from your browser using:
4. http://<NodeIP>:<NodePort>
5. Observe load balancing among the 10 Pods.

**Kubernetes Lab: Working with Deployments and Replica Management**

**🧭 Lab Overview**

In this lab, you’ll:

1. Understand and create Kubernetes **Deployments** using YAML.
2. Observe how **ReplicaSets** maintain Pod counts automatically.
3. Perform **Pod deletion**, **recreation**, and **rolling updates** using kubectl.
4. Explore high availability and fault tolerance via Deployments.
5. Contrast **Pod** vs **Deployment** in terms of resilience and lifecycle.

**⚙️ Pre-requisites**

* A running Kubernetes cluster (minikube, kind, or cloud-managed).
* Namespace raman already exists. If not:
* kubectl create ns raman
* kubectl client properly configured.

**🧠 Concepts Refresher**

| **Concept** | **Description** |
| --- | --- |
| **Deployment** | Higher-level abstraction that manages a ReplicaSet to ensure desired Pods are always running. |
| **ReplicaSet** | Ensures a specific number of identical Pods are running at all times. |
| **Selector** | Matches Pods via labels so ReplicaSet knows which Pods to manage. |
| **Rolling Update** | Gradual replacement of Pods with new versions without downtime. |
| **Self-Healing** | If a Pod fails or is deleted, Deployment automatically recreates it. |

**🧩 Step-by-Step Lab**

**🧱 Step 1: Verify Namespace and Existing Pods**

k get pods -n raman

✅ **Purpose:**  
Ensure the raman namespace exists and check if previous Pods are running.  
If any existing pods interfere, you may delete them:

k delete pods --all -n raman

**🧩 Step 2: Create a Deployment Manifest (deploy.yaml)**

File: **deploy.yaml**

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment-raman

labels:

app: nginx

namespace: raman

spec:

replicas: 10

selector:

matchLabels:

app: rk

template:

metadata:

labels:

app: rk

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

**🔍 Step 3: Understand the Manifest (Deep Technical Breakdown)**

| **Section** | **Description** |
| --- | --- |
| **apiVersion: apps/v1** | Deployment belongs to the apps API group, version v1. |
| **kind: Deployment** | Declares the resource type. |
| **metadata** | Identifies resource name, namespace, and labels. |
| **spec.replicas: 10** | Instructs Kubernetes to maintain 10 Pods running this template. |
| **spec.selector.matchLabels** | Defines how the Deployment identifies which Pods belong to it. It must match template labels. |
| **spec.template.metadata.labels** | These labels are applied to Pods created by this Deployment. |
| **spec.template.spec.containers** | Defines container(s) that run in each Pod. |
| **containerPort: 80** | Opens port 80 inside the Pod for HTTP traffic. |

🔹 **Key Relationship:**  
The label in selector.matchLabels must **match exactly** with template.metadata.labels.  
Here both are app: rk — hence Pods are correctly associated with this Deployment.

**🚀 Step 4: Create the Deployment Declaratively**

k create -f deploy.yaml

✅ **Explanation:**  
Tells the Kubernetes API to create resources defined in deploy.yaml.

🔹 **Expected Output:**

deployment.apps/nginx-deployment-raman created

**🔍 Step 5: Verify Deployment Creation**

k get deployments -n raman

🔹 **Expected Output:**

NAME READY UP-TO-DATE AVAILABLE AGE

nginx-deployment-raman 10/10 10 10 1m

✅ **Explanation:**

* READY → number of Pods currently available.
* UP-TO-DATE → how many Pods are running the latest configuration.
* AVAILABLE → number of Pods ready to serve traffic.

**🧩 Step 6: View Created Pods**

k get pods -n raman -o wide

🔹 **Expected Output:**

NAME READY STATUS RESTARTS AGE IP NODE

nginx-deployment-raman-64b8785b9-6kg55 1/1 Running 0 2m 10.244.0.8 node1

nginx-deployment-raman-64b8785b9-lxk90 1/1 Running 0 2m 10.244.0.9 node2

...

✅ **Explanation:**  
Each Pod name follows this pattern:

<deployment-name>-<replicaset-hash>-<pod-id>

Kubernetes automatically manages naming and association through ReplicaSets.

**🧰 Step 7: Inspect Deployment Details**

k describe deployment nginx-deployment-raman -n raman

✅ **Explanation:**  
Displays internal structure, strategy, events, and ReplicaSet association.

🔹 **Key Fields to Observe:**

* **Selector** – app=rk
* **StrategyType** – RollingUpdate
* **Replicas** – desired vs current count
* **ReplicaSet Reference**

**🧱 Step 8: Inspect a Specific Pod**

k describe pod nginx-deployment-raman-64b8785b9-6kg55 -n raman

✅ **Purpose:**  
To understand:

* Which node the Pod runs on.
* Container image used.
* Lifecycle events (Pulled → Created → Started).

🔹 **Events Section Example:**

Normal Scheduled Successfully assigned raman/nginx-deployment-raman-64b8785b9-6kg55 to node1

Normal Pulled Container image "nginx:1.14.2" already present on machine

Normal Created Created container nginx

Normal Started Started container nginx

**🔁 Step 9: Observe Self-Healing Behavior**

Now, **delete one Pod manually** and see what happens.

k delete pod nginx-deployment-raman-64b8785b9-6kg55 -n raman

✅ **Expected Output:**

pod "nginx-deployment-raman-64b8785b9-6kg55" deleted

Immediately check again:

k get pods -n raman

🔹 **Observation:**  
You will again see **10 Pods running** — a new Pod replaces the deleted one automatically.

✅ **Explanation:**  
The **ReplicaSet controller** notices one missing replica and spawns a new Pod to maintain the declared replica count (replicas: 10).

**🧩 Step 10: Modify the Deployment (Rolling Update Simulation)**

Open the file:

vi deploy.yaml

Change image version:

image: nginx:1.25.3

Then apply:

k apply -f deploy.yaml

✅ **Explanation:**  
Triggers a **rolling update** — gradually replaces old Pods running nginx:1.14.2 with new Pods running nginx:1.25.3.

🔹 **Check Status:**

k rollout status deployment nginx-deployment-raman -n raman

🔹 **Expected Output:**

deployment "nginx-deployment-raman" successfully rolled out

**🧠 Step 11: Verify ReplicaSet & Rollout History**

Check underlying ReplicaSets:

k get rs -n raman

🔹 **Output Example:**

NAME DESIRED CURRENT READY AGE

nginx-deployment-raman-64b8785b9 0 0 0 20m

nginx-deployment-raman-84a75dfb6 10 10 10 1m

✅ **Explanation:**

* Older ReplicaSet (64b8785b9) → old image version, scaled down.
* New ReplicaSet (84a75dfb6) → new version, scaled up.

You can check rollout history:

k rollout history deployment nginx-deployment-raman -n raman

**🧹 Step 12: Cleanup**

k delete deployment nginx-deployment-raman -n raman

✅ **Explanation:**  
Deleting a Deployment automatically removes associated ReplicaSets and Pods.

**🧾 Lab Summary**

| **Concept** | **Command** | **Purpose** |
| --- | --- | --- |
| **Deployment Creation** | k create -f deploy.yaml | Declarative workload creation |
| **Pod Inspection** | k get pods -n raman -o wide | Verify all replicas |
| **Self-Healing** | k delete pod ... | Observe automatic recreation |
| **Rolling Update** | k apply -f deploy.yaml | Update container version seamlessly |
| **ReplicaSet Mgmt** | k get rs -n raman | View underlying controllers |
| **Rollback** | k rollout undo deployment <name> | Revert to previous stable version |

**🧠 Deep Dive: Deployment Internals**

When you apply a Deployment manifest:

1. **Deployment Controller** creates a **ReplicaSet**.
2. The **ReplicaSet** ensures the defined number of **Pods** (replicas) exist.
3. If a Pod fails or is deleted:
   * ReplicaSet detects a mismatch between **desired** and **current** replicas.
   * It automatically creates new Pods to restore equilibrium.
4. During updates:
   * A new ReplicaSet is created.
   * Pods from the old ReplicaSet are gradually terminated while new ones spin up — this ensures **zero downtime**.

**⚙️ Advanced Exercises (Optional)**

1. **Scale Deployment**
2. k scale deployment nginx-deployment-raman -n raman --replicas=5

➤ Observe how Pods reduce automatically.

1. **Pause a Rollout**
2. k rollout pause deployment nginx-deployment-raman -n raman
3. **Rollback**
4. k rollout undo deployment nginx-deployment-raman -n raman
5. **Expose Deployment via Service**
6. k expose deployment nginx-deployment-raman --type=NodePort --port=80 -n raman

**🧩 Key Takeaways**

* **Pods** are ephemeral and non-resilient.
* **Deployments** provide:
  + Declarative management.
  + Scaling (replicas).
  + Self-healing.
  + Rolling updates & rollback.
* **ReplicaSets** are the engine maintaining Pod count.
* **Labels & Selectors** are the glue connecting resources.

**Kubernetes Lab: Exposing Applications Using Services (NodePort, ClusterIP, LoadBalancer)**

**🧭 Lab Overview**

In this lab, you will:

1. Understand the purpose and types of Kubernetes Services.
2. Create and apply a **NodePort Service** manifest (service.yaml).
3. Verify Pod–Service connectivity.
4. Access your deployed NGINX application externally.
5. Learn conceptual differences between **ClusterIP**, **NodePort**, and **LoadBalancer**.

**⚙️ Pre-requisites**

* You must have:
  + A running Kubernetes cluster.
  + Namespace raman already created (kubectl create ns raman).
  + A **Deployment** already running with label app: rk (from previous lab).
* kubectl CLI configured properly.

**🧠 Concepts Refresher**

**🔹 What is a Service?**

A **Service** in Kubernetes is a stable, virtual network endpoint that exposes a group of Pods (selected via labels) as a single access point.  
Because Pods are **ephemeral** (can restart, reschedule, or die), Services provide a **consistent network identity** to access them.

**🔹 Why Do We Need Services?**

Without a Service:

* Pods have **dynamic IPs** — they change each time Pods are recreated.
* You can’t reliably reach a Pod directly across nodes.

Services solve this by:

* Providing a **stable DNS name**.
* Acting as a **load balancer** across matching Pods.
* Enabling internal or external access depending on type.

**🔹 Types of Services**

| **Service Type** | **Access Scope** | **Description** |
| --- | --- | --- |
| **ClusterIP** (default) | Internal-only | Exposes the app inside the cluster only. |
| **NodePort** | External access via Node IP + Port | Exposes the app on a port across all cluster nodes. |
| **LoadBalancer** | External with public IP (cloud providers) | Provisions an external load balancer (e.g., AWS ELB, GCP LB). |

**🧩 Step-by-Step Lab**

**🧱 Step 1: Verify Existing Deployment and Pods**

k get deploy -n raman

🔹 **Expected Output:**

NAME READY UP-TO-DATE AVAILABLE AGE

nginx-deployment-raman 10/10 10 10 10m

Check Pods:

k get pods -n raman -o wide

🔹 **Output Example:**

NAME READY STATUS RESTARTS AGE IP NODE

nginx-deployment-raman-64b8785b9-k8d52 1/1 Running 0 10m 10.244.0.10 node1

✅ **Explanation:**  
Pods are labeled with app: rk, which the Service will use in its selector.

**🧩 Step 2: Create the Service Manifest (service.yaml)**

File: **service.yaml**

apiVersion: v1

kind: Service

metadata:

name: raman-service

namespace: raman

spec:

type: NodePort

selector:

app: rk

ports:

- port: 80

targetPort: 80

nodePort: 30007

**🔍 Step 3: Deep Breakdown of the YAML**

| **Field** | **Meaning** |
| --- | --- |
| **apiVersion: v1** | Core API group for basic Kubernetes objects. |
| **kind: Service** | Declares a Service resource. |
| **metadata.name** | Name of the Service — becomes part of its DNS entry. |
| **namespace** | Associates the Service to the raman namespace. |
| **type: NodePort** | Exposes the app externally via a static port (30000–32767) on each node. |
| **selector** | Targets all Pods with label app: rk. |
| **port** | Port number Service listens on internally. |
| **targetPort** | Port inside the Pod to forward traffic to (usually container port). |
| **nodePort** | Fixed external port on each node to reach the Service. |

🔹 **Important:**  
Kubernetes automatically maps traffic like this:

Client Request --> NodeIP:nodePort --> ClusterIP (virtual endpoint) --> Target Pod(s)

**🚀 Step 4: Apply the Service Manifest**

k create -f service.yaml

🔹 **Expected Output:**

service/raman-service created

**🧩 Step 5: Verify the Service**

k get svc -n raman

🔹 **Expected Output:**

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

raman-service NodePort 10.108.174.24 <none> 80:30007/TCP 10s

✅ **Explanation:**

* **CLUSTER-IP** → internal virtual IP assigned by Kubernetes.
* **PORT(S)** → mapping of internal to external ports (80:30007).
* **TYPE** → NodePort confirms external accessibility.

**🔍 Step 6: Describe the Service (Deep View)**

k describe svc raman-service -n raman

🔹 **Output (Example):**

Name: raman-service

Namespace: raman

Selector: app=rk

Type: NodePort

IP: 10.108.174.24

Port: <unset> 80/TCP

TargetPort: 80/TCP

NodePort: <unset> 30007/TCP

Endpoints: 10.244.0.10:80,10.244.0.11:80,10.244.0.12:80

✅ **Explanation:**

* **Endpoints** show all Pod IPs behind the Service — these are automatically updated as Pods are created or destroyed.
* **NodePort 30007** means any node’s IP at port 30007 will route to the backend Pods.

**🧩 Step 7: Access the Application**

1. Get your Node IP(s):
2. k get nodes -o wide

Example output:

NAME STATUS ROLES AGE VERSION INTERNAL-IP EXTERNAL-IP

raman-node Ready worker 1h v1.29.0 192.168.49.2 <none>

1. Test from your local machine (or the node itself):
2. curl http://192.168.49.2:30007

or open in a browser:

http://<NodeIP>:30007

✅ **Expected Result:**  
Nginx default welcome page (HTML content: “Welcome to nginx!”)

**🧩 Step 8: Observe Service–Pod Relationship**

Run:

k get endpoints -n raman

🔹 **Expected Output:**

NAME ENDPOINTS AGE

raman-service 10.244.0.10:80,10.244.0.11:80,10.244.0.12:80 1m

✅ **Explanation:**  
Endpoints are dynamically managed — every Pod with label app=rk is automatically registered and load-balanced behind the Service.

**🧰 Step 9: Test Service Self-healing (Dynamic Endpoint Mapping)**

Try deleting one Pod:

k delete pod <any-pod-name> -n raman

Then check endpoints again:

k get endpoints -n raman

🔹 **Observation:**  
A new Pod replaces the deleted one, and the endpoint list updates automatically — ensuring continuous availability.

**🧹 Step 10: Cleanup (Optional)**

k delete svc raman-service -n raman

✅ **Explanation:**  
Removes the Service and its virtual IP. Pods remain unaffected.

**🧾 Lab Summary**

| **Resource** | **Purpose** | **Key Command** |
| --- | --- | --- |
| **Pod** | Smallest deployable object — runs container(s). | kubectl run / YAML |
| **Deployment** | Manages multiple Pod replicas. | kubectl create -f deploy.yaml |
| **Service** | Exposes Pods via stable endpoint. | kubectl create -f service.yaml |
| **NodePort Service** | External access on each Node IP. | Port range: 30000–32767 |
| **ClusterIP Service** | Internal-only service (default type). |  |
| **LoadBalancer Service** | External via cloud provider LB. |  |

**🌐 Visual Architecture**

+----------------------------+

| Client / Browser |

+-------------+--------------+

|

http://<NodeIP>:30007

|

+---------v---------+

| NodePort Service |

| (raman-service) |

+---------+---------+

|

+-------+--------+

| ClusterIP Proxy|

+-------+--------+

|

+------------------+------------------+

| | |

Pod 1 (10.244.0.10) Pod 2 (10.244.0.11) Pod 3 (10.244.0.12)

| Nginx 80/TCP | | Nginx 80/TCP | | Nginx 80/TCP |

✅ **Traffic Flow:**

* External user → Node IP:30007
* NodePort forwards to ClusterIP → Load-balanced across Pod IPs

**🧠 Discussion / Validation Questions**

1. What is the difference between port, targetPort, and nodePort?
2. How does Kubernetes route traffic between Service and Pods?
3. What happens to Service endpoints when a Pod crashes?
4. What are the default port ranges for NodePort services?
5. Why is ClusterIP the default service type?

**🧰 Optional Bonus Exercises**

1. **Change Service Type to ClusterIP:**
2. k patch svc raman-service -n raman -p '{"spec":{"type":"ClusterIP"}}'
3. **Expose Deployment Using Shortcut Command:**
4. kubectl expose deployment nginx-deployment-raman --type=NodePort --port=80 -n raman
5. **Verify DNS Resolution:**
6. k run dns-test --image=busybox:1.28 -it --rm -- nslookup raman-service.raman.svc.cluster.local

**🧩 Key Takeaways**

* **Services decouple frontend access from Pod IPs.**
* **NodePort** provides external connectivity — ideal for local or test environments.
* **ClusterIP** is internal-only — ideal for service-to-service communication.
* **LoadBalancer** integrates with cloud providers for real-world external load balancing.
* Label–selector matching automatically syncs Pods and Services dynamically.
* Deleting Pods doesn’t break connectivity — endpoints are self-updating.

**Kubernetes Lab: Controlling Pod Placement with Node Selectors**

**🧭 Lab Overview**

In this lab, you will:

1. Understand how Kubernetes schedules pods on worker nodes.
2. Use the nodeSelector field in a Deployment to **force pods to run on specific nodes**.
3. Verify node labels and apply custom labels.
4. Observe pod scheduling and placement.
5. Explore troubleshooting if scheduling fails (e.g., label mismatch or taints).

**⚙️ Pre-requisites**

Before starting:

* A working **multi-node Kubernetes cluster** (1 master + 1 or more worker nodes).
* Namespace raman already created:
* kubectl create ns raman
* kubectl CLI configured to connect to your cluster.

**🧠 Concepts Overview**

**🔹 1. Kubernetes Scheduler Overview**

Kubernetes automatically decides **which node** each pod should run on based on:

* **Resource availability** (CPU, memory)
* **Constraints** like labels, taints, tolerations, affinity, etc.
* **Scheduling policies**

By default, the scheduler can place pods on *any available node*.

However, sometimes we want to control placement, for example:

* Run database pods on SSD-backed nodes.
* Run frontend pods on nodes with GPU.
* Pin training pods to a specific node for data locality.

That’s where **nodeSelector** and **node affinity** come in.

**🔹 2. nodeSelector — The Simplest Scheduling Constraint**

nodeSelector is a key–value pair that restricts which node(s) a pod can be scheduled on.

🧩 Syntax:

spec:

nodeSelector:

<label-key>: <label-value>

Example:

nodeSelector:

kubernetes.io/hostname: raman-worker

This ensures pods **only run on the node** where that label key–value pair matches.

**🔹 3. Nodes and Labels**

Each node in a Kubernetes cluster has metadata labels like:

* kubernetes.io/hostname
* beta.kubernetes.io/os
* kubernetes.io/arch

You can see these using:

kubectl get nodes --show-labels

You can also add your own custom labels:

kubectl label node <node-name> env=dev

**🧩 Step-by-Step Lab**

**🧱 Step 1: Inspect Your Cluster Nodes**

Run:

k get nodes -o wide

🔹 **Expected Output:**

NAME STATUS ROLES AGE VERSION INTERNAL-IP EXTERNAL-IP OS-IMAGE KERNEL-VERSION CONTAINER-RUNTIME

raman-master Ready control-plane 2d v1.29.0 192.168.49.2 <none> Ubuntu 22.04.3 LTS 5.15.0-91-generic containerd://1.7.3

raman-worker Ready <none> 2d v1.29.0 192.168.49.3 <none> Ubuntu 22.04.3 LTS 5.15.0-91-generic containerd://1.7.3

**🧩 Step 2: View Node Labels**

Run:

k describe node raman-master | grep -i "Labels"

🔹 **Sample Output:**

Labels: beta.kubernetes.io/arch=amd64

beta.kubernetes.io/os=linux

kubernetes.io/hostname=raman-master

node-role.kubernetes.io/control-plane=

✅ Note the label kubernetes.io/hostname=raman-master.  
This is what we’ll use in the Deployment manifest to target this node.

**🧩 Step 3: Deployment Manifest with NodeSelector**

File: **deploy.yaml**

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment-raman

labels:

app: nginx

namespace: raman

spec:

replicas: 5

selector:

matchLabels:

app: rk

template:

metadata:

labels:

app: rk

spec:

# 🔹 Node Selector ensures pods run only on the node named 'raman-master'

nodeSelector:

kubernetes.io/hostname: raman-master

containers:

- name: nginx

image: nginxdemos/hello

ports:

- containerPort: 80

**🧱 Step 4: Apply the Deployment**

k apply -f deploy.yaml

🔹 **Expected Output:**

deployment.apps/nginx-deployment-raman created

**🧩 Step 5: Verify Pod Placement**

k get pods -n raman -o wide

🔹 **Expected Output:**

NAME READY STATUS RESTARTS AGE IP NODE

nginx-deployment-raman-7688f997b9-2hrdl 1/1 Running 0 10s 10.244.0.15 raman-master

nginx-deployment-raman-7688f997b9-9sl8b 1/1 Running 0 10s 10.244.0.16 raman-master

...

✅ **Verification Point:**  
All pods are scheduled on the **raman-master** node — as defined by the nodeSelector.

**🧩 Step 6: Verify Scheduler Behavior**

Run:

k describe pod <pod-name> -n raman

🔹 Look for section:

Node-Selectors: kubernetes.io/hostname=raman-master

Node: raman-master/192.168.49.2

✅ **Explanation:**  
The scheduler evaluated your nodeSelector, found the node with matching label, and bound the pod there.

**🧩 Step 7: Testing Alternate Node (Optional)**

Now, let’s change the Deployment to point to another node.

Edit:

vi deploy.yaml

Change:

nodeSelector:

kubernetes.io/hostname: raman-worker

Apply:

k apply -f deploy.yaml

Then check:

k get pods -n raman -o wide

🔹 **Expected:** All pods are now on the **raman-worker** node.

**🧩 Step 8: Handling Scheduling Errors**

If you specify a nodeSelector that does **not match any node label**, pods will **not be scheduled**.

Try an invalid label:

nodeSelector:

kubernetes.io/hostname: non-existent-node

Apply and check:

k get pods -n raman

🔹 **Output:**

NAME READY STATUS RESTARTS AGE

nginx-deployment-raman-xxss5 0/1 Pending 0 30s

Then describe:

k describe pod <pod-name> -n raman

You’ll see:

Warning FailedScheduling scheduler 0/2 nodes are available: 2 node(s) didn't match node selector.

✅ **Concept:** Scheduler honors your nodeSelector constraint — if no node matches, pod stays Pending.

**🧩 Step 9: Label Nodes Manually (Custom Labels)**

You can label a node with any key–value pair:

k label node raman-worker env=prod

Verify:

k get nodes --show-labels | grep raman-worker

Update deployment to use:

nodeSelector:

env: prod

Apply again — pods will now be scheduled only to that labeled node.

**🧩 Step 10: Cleanup (Optional)**

k delete -f deploy.yaml

✅ **Result:**  
Deployment and pods deleted.

**📘 YAML Explanation Summary**

| **Field** | **Description** |
| --- | --- |
| **apiVersion: apps/v1** | Uses the Deployment API. |
| **kind: Deployment** | Manages multiple replicas of a pod. |
| **metadata.name** | Logical name of the Deployment. |
| **replicas: 5** | Number of pod instances to maintain. |
| **selector.matchLabels** | Matches pods using label app: rk. |
| **template.metadata.labels** | Labels applied to the pods. |
| **spec.nodeSelector** | Restricts pod scheduling to nodes with matching labels. |
| **containers.image** | NGINX demo image for testing HTTP response. |
| **ports.containerPort** | Exposes port 80 on each pod. |

**🌐 Architecture Diagram**

+--------------------------------------------------------------+

| Kubernetes Cluster |

| |

| +------------------+ +------------------+ |

| | raman-master | | raman-worker | |

| | Labels: | | Labels: | |

| | kubernetes.io/hostname=raman-master | |

| | (No match) |

| | +-------------------------------------+ |

| | | Pods (5 replicas) running on master | |

| | +-------------------------------------+ |

| +----------------------------------------+ |

| |

+--------------------------------------------------------------+

✅ **Key Insight:**  
Scheduler only deploys pods on nodes matching the selector labels.

**🧠 Discussion & Validation Questions**

1. What happens when the node label doesn’t match your nodeSelector?
2. How can you view existing labels of a node?
3. Can you have multiple nodeSelector key–value pairs? (→ Yes, but all must match.)
4. What’s the difference between nodeSelector and *node affinity*?
5. Why might you want to restrict certain workloads to specific nodes?

**🧰 Optional Advanced Exercise**

**🧩 Step 11: Use Node Affinity (Advanced Version of NodeSelector)**

Try replacing nodeSelector with affinity:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: kubernetes.io/hostname

operator: In

values:

- raman-master

Apply it:

k apply -f deploy.yaml

✅ This gives more flexibility (operators like In, NotIn, Exists) compared to basic nodeSelector.

**🧾 Key Takeaways**

| **Concept** | **Description** |
| --- | --- |
| **Scheduler** | Automatically decides where pods should run. |
| **Node Labels** | Metadata tags attached to each node. |
| **nodeSelector** | Simplest constraint mechanism for pinning pods. |
| **Affinity/Anti-Affinity** | Advanced scheduling rules for more complex requirements. |
| **Pending Pods** | Indicate no node matched your scheduling conditions. |

**🧩 Real-world Use Cases**

| **Use Case** | **Example** |
| --- | --- |
| **Workload segregation** | Run production workloads only on “prod” labeled nodes. |
| **Hardware specialization** | Run ML jobs on nodes labeled gpu=true. |
| **Security** | Isolate sensitive pods on dedicated nodes. |
| **Data locality** | Run database pods near storage nodes. |