**🧪 Kubernetes Pod Scheduling Lab Guide**

**🎯 Lab Objectives**

By the end of this lab, you will be able to:

* Understand why pods are not scheduled on master nodes by default.
* Work with **taints & tolerations**.
* Schedule pods manually using **nodeName** (hard assignment).
* Schedule pods using **nodeSelector** (flexible placement).
* Label nodes and target them with deployments.

**1. Environment Setup**

* A Kubernetes cluster bootstrapped with **kubeadm**.
* At least **1 control-plane node** (master) and **2 worker nodes** (say w1 and w2).
* kubectl configured (k alias assumed).

Verify nodes:

kubectl get nodes -o wide

Expected:

NAME STATUS ROLES AGE VERSION

master Ready control-plane ... v1.xx.x

w1 Ready <none> ...

w2 Ready <none> ...

**2. Why Pods Don’t Run on Master**

By default, kubeadm applies a **taint** on the master:

kubectl describe node master | grep Taints

You should see:

Taints: node-role.kubernetes.io/control-plane:NoSchedule

This means: **No pod will run here unless it tolerates this taint.**

👉 So if you try to deploy a pod without specifying anything, it will land on worker nodes.

**3. Create a Namespace**

We’ll isolate this lab in a namespace:

kubectl create namespace raman

**4. Deploy Using nodeSelector**

First, label your worker nodes so we can target them.

kubectl label node w1 env=prod

kubectl label node w2 env=dev

Verify labels:

kubectl get nodes --show-labels

**Deployment YAML (nodeSelector)**

deploy.yaml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-deployment

labels:

test: rk

namespace: raman

spec:

replicas: 4

selector:

matchLabels:

test: raman

template:

metadata:

labels:

test: raman

spec:

containers:

- name: raman-con

image: nginx

ports:

- containerPort: 3000

nodeSelector:

env: dev # Target only nodes labeled env=dev

Apply:

kubectl apply -f deploy.yaml

Check pods:

kubectl get pods -n raman -o wide

All replicas should be scheduled **only on node w2** (because it has env=dev).

**5. Deploy Using nodeName**

Now test **hard assignment**.  
Edit deploy.yaml:

spec:

...

template:

spec:

containers:

- name: raman-con

image: nginx

ports:

- containerPort: 3000

nodeName: w1 # Direct hard assignment

Apply:

kubectl apply -f deploy.yaml --force

Check:

kubectl get pods -n raman -o wide

👉 All replicas will be forced on **w1**, regardless of labels.

**6. Scheduling on Master (Bypassing Taint)**

If you want to test pods on master:

**Option 1: Remove taint (not recommended for production)**

kubectl taint nodes master node-role.kubernetes.io/control-plane:NoSchedule-

Now the master behaves like a normal schedulable node.

**Option 2: Add toleration (better practice)**

Edit deploy.yaml:

spec:

...

template:

spec:

tolerations:

- key: "node-role.kubernetes.io/control-plane"

operator: "Exists"

effect: "NoSchedule"

nodeName: master

Apply:

kubectl apply -f deploy.yaml

Pods will now land on the master despite its taint.

**7. Key Concepts Recap**

* **Taint** = "Keep away unless tolerated".
* **Toleration** = "I can survive on a tainted node".
* **nodeName** = Hard assignment, overrides taints (direct scheduling).
* **nodeSelector** = Flexible scheduling using labels.

**8. Clean Up**

When done, clean resources:

kubectl delete namespace raman

kubectl label node w1 env-

kubectl label node w2 env-

That’s the complete end-to-end lab. The fun twist here is that nodeName acts like a bossy parent (“you *will* go there and nowhere else”), while nodeSelector is more like a dating app filter (“I prefer nodes with env=dev”).

**🧪 Kubernetes Lab Guide: Taints, Tolerations, and Scheduling Policies**

**🎯 Lab Objectives**

* Understand different taint effects (NoSchedule, PreferNoSchedule, NoExecute).
* Practice adding and removing taints.
* Deploy workloads with/without tolerations.
* Observe how pods react to taints (scheduling and eviction).

**1. Environment Setup**

Cluster:

* **master** node (control-plane, tainted by default).
* **w1** node labeled as env=prod.
* **w2** node labeled as env=dev.

Verify labels:

kubectl label node w1 env=prod --overwrite

kubectl label node w2 env=dev --overwrite

kubectl get nodes --show-labels

Check taints:

kubectl describe nodes | grep -i Taints

Expected baseline:

master node-role.kubernetes.io/control-plane:NoSchedule

w1 <none>

w2 <none>

**2. Taint Effects Explained**

* **NoSchedule** → Pods without tolerations will **not schedule** on this node.
* **PreferNoSchedule** → Scheduler **tries to avoid** the node, but may schedule if no alternatives.
* **NoExecute** → Not only prevents scheduling, but also **evicts already running pods** if they don’t tolerate it.

**3. Testing NoSchedule**

Apply taint:

kubectl taint nodes w1 env=prod:NoSchedule

Confirm:

kubectl describe node w1 | grep -i Taints

Expected:

Taints: env=prod:NoSchedule

**Deployment Without Toleration**

deploy.yaml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-deployment

namespace: raman

spec:

replicas: 3

selector:

matchLabels:

app: rk

template:

metadata:

labels:

app: rk

spec:

containers:

- name: nginx

image: nginx

Apply:

kubectl create ns raman

kubectl apply -f deploy.yaml

kubectl get pods -n raman -o wide

👉 Result: Pods will **not land on w1**, only on w2 (since w1 is tainted).

**Add Toleration**

Update deploy.yaml:

spec:

template:

spec:

tolerations:

- key: "env"

operator: "Equal"

value: "prod"

effect: "NoSchedule"

containers:

- name: nginx

image: nginx

Reapply:

kubectl apply -f deploy.yaml

kubectl get pods -n raman -o wide

👉 Now pods can land on **w1**, since toleration matches.

**4. Testing PreferNoSchedule**

Apply:

kubectl taint nodes w1 env=prod:PreferNoSchedule

👉 Pods **may still land** on w1 if scheduler has no other choice (for example, if w2 has no resources).

Test by deleting the existing deployment and redeploying:

kubectl delete deploy --all -n raman

kubectl apply -f deploy.yaml

kubectl get pods -n raman -o wide

**5. Testing NoExecute**

Apply:

kubectl taint nodes w1 env=prod:NoExecute

👉 This is different:

* New pods without toleration = **won’t schedule**.
* Existing pods without toleration = **evicted immediately**.

Check live:

kubectl get pods -n raman -o wide

kubectl describe pod <pod-name> -n raman

You’ll see pods either moving to w2 or stuck in Pending if no other node is available.

**Add Toleration for NoExecute**

tolerations:

- key: "env"

operator: "Equal"

value: "prod"

effect: "NoExecute"

Apply again, pods will survive on w1.

**6. Untainting Nodes**

Remove taint:

kubectl taint nodes w1 env=prod:NoSchedule-

kubectl taint nodes w1 env=prod:PreferNoSchedule-

kubectl taint nodes w1 env=prod:NoExecute-

Confirm:

kubectl describe node w1 | grep -i Taints

👉 Should now show Taints: <none>.

**7. Default Master Node Taint**

Recall:

kubectl taint nodes master node-role.kubernetes.io/control-plane=:NoSchedule

👉 That’s why workloads don’t land on master by default.  
To bypass:

* Remove taint:
* kubectl taint nodes master node-role.kubernetes.io/control-plane:NoSchedule-
* Or tolerate in pod spec:
* tolerations:
* - key: "node-role.kubernetes.io/control-plane"
* operator: "Exists"
* effect: "NoSchedule"

**8. Cleanup**

kubectl delete ns raman

kubectl taint nodes w1 env-

kubectl taint nodes master node-role.kubernetes.io/control-plane:NoSchedule

**🔑 Key Takeaways**

* NoSchedule: prevents new pods unless tolerated.
* PreferNoSchedule: scheduler avoids node but not guaranteed.
* NoExecute: prevents scheduling **and evicts running pods**.
* Tolerations let pods survive or schedule onto tainted nodes.
* Master taint exists by default in kubeadm clusters to protect control-plane.

**🧪 Kubernetes Lab Guide: DaemonSets with Tolerations (Fluentd Logging Example)**

**🎯 Objectives**

By the end of this lab, you will:

* Understand what **DaemonSets** are and why they are used for cluster-wide agents.
* Deploy a **Fluentd DaemonSet** that tolerates control-plane taints.
* Verify pods are scheduled across *all* nodes (including master).
* Explore how tolerations ensure DaemonSets can run universally.

**1. Background: What is a DaemonSet?**

* A **DaemonSet** ensures **one pod runs on every node** (or on a subset, if restricted by selectors).
* Typical use: log shippers, monitoring agents, networking daemons.
* Without tolerations, DaemonSets **skip master/control-plane nodes** (since those are tainted).

**2. Setup**

Create a namespace:

kubectl create namespace raman

Check your nodes:

kubectl get nodes -o wide

kubectl describe nodes | grep -i Taints

Expected:

* master → node-role.kubernetes.io/control-plane:NoSchedule
* w1, w2 → normal worker nodes (plus any taints you added earlier).

**3. DaemonSet Manifest (with Tolerations)**

Save the following as daemon.yml:

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: raman-fluentd-elasticsearch

namespace: raman

labels:

k8s-app: fluentd-logging

spec:

selector:

matchLabels:

name: fluentd-elasticsearch

template:

metadata:

labels:

name: fluentd-elasticsearch

spec:

tolerations:

# Allow pods to run on master/control-plane nodes

- key: node-role.kubernetes.io/control-plane

operator: Exists

effect: NoSchedule

- key: node-role.kubernetes.io/master # for backward compatibility

operator: Exists

effect: NoSchedule

containers:

- name: fluentd-elasticsearch

image: quay.io/fluentd\_elasticsearch/fluentd:v5.0.1

resources:

limits:

memory: 100Mi

requests:

cpu: 100m

memory: 100Mi

volumeMounts:

- name: rklog

mountPath: /var/log # Mount host logs into the container

volumes:

- name: rklog

hostPath:

path: /var/log # HostPath volume from each node

**4. Deploy the DaemonSet**

kubectl apply -f daemon.yml

Check status:

kubectl get daemonset -n raman -o wide

You should see desired = current = number of nodes.

Check pods:

kubectl get pods -n raman -o wide

👉 There should be **one Fluentd pod per node**, including the master (thanks to tolerations).

**5. Verification**

**Describe one pod**

kubectl describe pod <pod-name> -n raman

Look for:

* Node it’s scheduled on.
* Mounted volume: /var/log.

**Inspect DaemonSet placement**

kubectl get pods -n raman -o wide | awk '{print $1,$7}'

This shows which pod is on which node. You should see one pod each on master, w1, and w2.

**6. Modify & Reapply**

If you remove tolerations:

tolerations: []

Reapply:

kubectl apply -f daemon.yml

kubectl get pods -n raman -o wide

👉 Pods will disappear from master, staying only on worker nodes.

This demonstrates the effect of tolerations in DaemonSets.

**7. Cleanup**

kubectl delete -f daemon.yml

kubectl delete namespace raman

**🔑 Key Takeaways**

* **DaemonSet** = ensures one pod per node, commonly used for system-level agents.
* **Taints on master** prevent normal pods from running there.
* **Tolerations** allow DaemonSets to ignore taints and run even on control-plane nodes.
* Mounting /var/log via hostPath lets Fluentd collect real node logs.

**🧪 Kubernetes Lab Guide: Working with Secrets (Environment Variables & Mounted Volumes)**

**🎯 Objectives**

By the end of this lab, you will:

* Understand how to encode sensitive information into Kubernetes Secrets.
* Create a Secret containing username and password.
* Use Secrets in a Deployment by **injecting them as environment variables**.
* Use Secrets in a Deployment by **mounting them as volumes**.
* Verify that applications can consume those values securely.

**1. Background: Why Secrets?**

* Developers often need sensitive data (credentials, API keys, tokens) inside applications.
* Hardcoding values in YAML files is insecure.
* Kubernetes **Secret** object allows storage of sensitive data in a base64-encoded form.
* Secrets can be mounted as **environment variables** or **files**.

**2. Encode Data**

Before creating a Secret, encode values with base64 (this is not encryption, just encoding).

echo 'ramankhanna' | base64

# cmFtYW5raGFubmEK

echo 'ramankhanna123' | base64

# cmFtYW5raGFubmExMjMK

**3. Create a Secret**

Create secret.yml:

apiVersion: v1

kind: Secret

metadata:

name: raman-secret

namespace: raman

type: Opaque

data:

username: cmFtYW5raGFubmEK

password: cmFtYW5raGFubmExMjMK

Apply:

kubectl create -f secret.yml

Verify:

kubectl get secrets -n raman

kubectl describe secret raman-secret -n raman

👉 The Data section will show keys but not plaintext values.

**4. Method 1: Use Secrets as Environment Variables**

Create deploy2-env.yml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-deployment-env

namespace: raman

labels:

app: myapp

spec:

replicas: 2

selector:

matchLabels:

app: myapp

type: front-end

template:

metadata:

labels:

app: myapp

type: front-end

spec:

containers:

- name: httpd-container

image: httpd

env:

- name: SECRET\_USERNAME

valueFrom:

secretKeyRef:

name: raman-secret

key: username

- name: SECRET\_PASSWD

valueFrom:

secretKeyRef:

name: raman-secret

key: password

Apply:

kubectl apply -f deploy2-env.yml

Verify:

kubectl get pods -n raman

kubectl exec -it <pod-name> -n raman -- env | grep SECRET\_

👉 You should see:

SECRET\_USERNAME=ramankhanna

SECRET\_PASSWD=ramankhanna123

**5. Method 2: Use Secrets as Mounted Volumes**

Create deploy2-vol.yml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-deployment-vol

namespace: raman

labels:

app: myapp

spec:

replicas: 2

selector:

matchLabels:

app: myapp

type: front-end

template:

metadata:

labels:

app: myapp

type: front-end

spec:

containers:

- name: httpd-container

image: httpd

volumeMounts:

- name: creds-volume

readOnly: true

mountPath: "/opt/creds"

volumes:

- name: creds-volume

secret:

secretName: raman-secret

Apply:

kubectl apply -f deploy2-vol.yml

Verify:

kubectl get pods -n raman

kubectl exec -it <pod-name> -n raman -- ls /opt/creds

👉 You’ll see two files: username and password.

Check contents:

kubectl exec -it <pod-name> -n raman -- cat /opt/creds/username

kubectl exec -it <pod-name> -n raman -- cat /opt/creds/password

Expected:

ramankhanna

ramankhanna123

**6. Comparison: Env Vars vs Mounted Volumes**

* **Env vars**
  + Simple, good for apps that expect config via environment.
  + But secrets may appear in kubectl describe pod output → visible to anyone with RBAC read access.
* **Mounted volumes**
  + More secure, values stored as files.
  + Easier to rotate (update secret, pods automatically see updated files).
  + Ideal for apps that read configs from files.

**7. Cleanup**

kubectl delete deployment raman-deployment-env -n raman

kubectl delete deployment raman-deployment-vol -n raman

kubectl delete secret raman-secret -n raman

**🔑 Key Takeaways**

* Always encode sensitive info before putting in Secrets (base64).
* Secrets can be consumed in two ways:
  + **Environment variables** (easy, but slightly less secure).
  + **Mounted as files** (preferred in production).
* Kubernetes doesn’t encrypt secrets at rest by default—encryption at rest should be enabled at the cluster level for true security.

**🧪 Kubernetes Lab Guide: Managing Configurations with ConfigMaps**

**🎯 Objectives**

By the end of this lab you will:

* Understand the difference between **Secrets** and **ConfigMaps**.
* Create ConfigMaps from files.
* Inject ConfigMap data into Pods as mounted files.
* Swap environments (dev vs prod) without changing the container image.
* Verify behavior by checking mounted content inside Pods and exposing service.

**1. Background**

* **Secrets** → store confidential key/value data (passwords, tokens, API keys).
* **ConfigMaps** → store **non-confidential** configuration (URLs, ports, logging levels, HTML templates).

**Example configuration keys:**

NODE\_ENV=production

APP\_ENV=dev

LOG\_LEVEL=info

PORT=8080

BACKGROUND\_COLOUR=blue

👉 ConfigMaps help **decouple application code (image)** from **environment-specific configuration**.

* One container image can run in dev, staging, or prod — only configuration changes.

**2. Prepare Config Files**

Create two files locally:

echo "hello from prod" > prod.html

echo "hello from dev" > dev.html

Check:

cat prod.html

# hello from prod

cat dev.html

# hello from dev

**3. Create ConfigMaps**

Create a **ConfigMap from file** for prod:

kubectl create cm prod.cmap --from-file=prod.html -n raman

Create one for dev:

kubectl create cm dev.cmap --from-file=dev.html -n raman

Verify:

kubectl get cm -n raman

kubectl describe cm prod.cmap -n raman

kubectl describe cm dev.cmap -n raman

kubectl get cm prod.cmap -n raman -o yaml

👉 Each ConfigMap stores the HTML file contents as a key (prod.html or dev.html).

**4. Create Deployment Using ConfigMap**

Create deploy3.yml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

namespace: raman

labels:

app: nginx

spec:

replicas: 2 # run 2 pods

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

volumeMounts:

- name: rk

mountPath: /usr/share/nginx/html

volumes:

- name: rk

configMap:

# Uncomment for prod

#name: prod.cmap

#items:

# - key: prod.html

# path: index.html

# For dev (currently active)

name: dev.cmap

items:

- key: dev.html

path: index.html

Explanation:

* volumeMounts: Mounts the ConfigMap into container at /usr/share/nginx/html.
* items: Maps specific key from ConfigMap (e.g., dev.html) to a file (index.html).
* Swapping between prod.cmap and dev.cmap switches environment without new image.

**5. Deploy and Verify**

Apply:

kubectl apply -f deploy3.yml

Check Pods:

kubectl get pods -n raman

Check content inside a Pod:

kubectl exec -it <nginx-pod-name> -n raman -- cat /usr/share/nginx/html/index.html

Expected output (since dev.cmap is active):

hello from dev

**6. Expose Deployment**

Expose the deployment as a service:

kubectl expose deploy nginx-deployment -n raman \

--type=NodePort --port=80 --target-port=80 --name=raman-cmsvc

Check service:

kubectl get svc -n raman

You’ll see something like:

raman-cmsvc NodePort 10.104.188.45 <none> 80:30987/TCP 5m

👉 Access service in browser or with curl using <NodeIP>:<NodePort>.

**7. Switch to Prod Config**

Edit deploy3.yml to point to prod.cmap:

volumes:

- name: rk

configMap:

name: prod.cmap

items:

- key: prod.html

path: index.html

Apply changes:

kubectl apply -f deploy3.yml

Check content again:

kubectl exec -it <nginx-pod-name> -n raman -- cat /usr/share/nginx/html/index.html

Expected:

hello from prod

**8. Cleanup**

kubectl delete deploy nginx-deployment -n raman

kubectl delete svc raman-cmsvc -n raman

kubectl delete cm prod.cmap dev.cmap -n raman

**🔑 Key Takeaways**

* **ConfigMaps** hold non-sensitive configurations; **Secrets** hold sensitive data.
* ConfigMaps can be consumed:
  + As **environment variables** (good for simple configs like LOG\_LEVEL).
  + As **files** (great for structured configs like XML, JSON, or HTML templates).
* This pattern enables **12-factor app best practices**: keep code and config separate.
* Switching environments is as simple as swapping ConfigMap references—no need to rebuild container images.