**Horizontal Pod Autoscaler (HPA) Lab Guide**

**Prerequisites:**

* You have access to a Kubernetes cluster.
* kubectl is configured to interact with your cluster.
* Metrics Server is deployed and running in the cluster (for resource monitoring).

**Step 1: Verify Metrics Server Installation**

Before deploying the HPA, ensure the **Metrics Server** is running correctly.

1. Clone the metrics server repo:

bash

Copy code

git clone https://github.com/ramannkhanna2/k8s\_metrics\_server.git

1. Apply the deployment files for the metrics server:

bash

Copy code

cd k8s\_metrics\_server

kubectl apply -f .

1. Verify that the metrics server is working:

bash

Copy code

kubectl top nodes

kubectl top pods

**Step 2: Create a Deployment**

You will now create a simple deployment using the centos-deployment YAML file.

1. **Create a deployment file** (deploy2.yml):

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: centos-deployment

spec:

replicas: 1

selector:

matchLabels:

app: centos

template:

metadata:

labels:

app: centos

spec:

containers:

- name: centos-container

image: ubuntu

command: ["sleep", "infinity"]

resources:

requests:

memory: "64Mi"

cpu: "250m"

limits:

memory: "128Mi"

cpu: "500m"

1. **Deploy the centos-deployment**:

bash

Copy code

kubectl apply -f deploy2.yml

1. **Verify the deployment**:

bash

Copy code

kubectl get pods

**Step 3: Autoscale the Deployment**

1. **Create HPA for centos-deployment**:

bash

Copy code

kubectl autoscale deployment centos-deployment --cpu-percent=70 --min=1 --max=5

1. **Check the HPA**:

bash

Copy code

kubectl get hpa

1. **Watch the HPA in real-time**:

bash

Copy code

kubectl get hpa --watch

**Step 4: Simulate Load to Trigger Autoscaling**

To trigger autoscaling, you'll increase CPU load on the pods.

1. **Get inside one of the running pods**:

bash

Copy code

kubectl exec -it <pod-name> -- /bin/bash

1. **Run some commands to increase CPU load**:

bash

Copy code

apt update -y

apt update -y

apt update -y

apt update -y

apt update -y

apt update -y

1. Exit the pod when done.
2. **Observe the HPA** adjusting the number of pods:

bash

Copy code

watch kubectl get hpa

kubectl get pods

As CPU usage rises above the target (70%), the HPA will automatically scale up the number of pods.

**Step 5: Delete the HPA**

After testing, you can delete the HPA configuration:

1. **Delete the HPA**:

bash

Copy code

kubectl delete hpa centos-deployment

**Step 6: Cleanup Resources**

1. **Delete the deployment**:

bash

Copy code

kubectl delete deployment centos-deployment

1. **Delete the metrics server** if no longer needed:

bash

Copy code

kubectl delete -f k8s\_metrics\_server/

**Commands Summary**

| **Command** | **Description** |
| --- | --- |
| git clone https://github.com/ramannkhanna2/k8s\_metrics\_server.git | Clone metrics server repo |
| kubectl apply -f . | Apply the metrics server YAML files |
| kubectl top nodes | Check the node resource usage |
| kubectl top pods | Check the pod resource usage |
| kubectl apply -f deploy2.yml | Apply the deployment file |
| kubectl autoscale deployment centos-deployment --cpu-percent=70 --min=1 --max=5 | Create HPA with target CPU utilization |
| kubectl exec -it <pod-name> -- /bin/bash | Enter a running pod to simulate CPU load |
| apt update -y | Increase CPU load inside the pod |
| kubectl get hpa --watch | Watch HPA scale the deployment based on CPU utilization |
| kubectl delete hpa centos-deployment | Delete the HPA configuration |

This guide walks through a complete scenario of creating a Kubernetes Horizontal Pod Autoscaler, simulating load, and watching it autoscale based on CPU utilization.

**Kubernetes Deployment and Rollout Management Lab Guide**

**Prerequisites:**

* Access to a Kubernetes cluster.
* kubectl is configured to communicate with the cluster.

**Step 1: Create a Deployment**

1. **Create a deployment file (deploy.yaml)**:

This deployment creates 50 replicas of an NGINX web server using the **Recreate** strategy.

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: gagan-app

labels:

app: web2-prod-app

spec:

replicas: 50

strategy:

type: Recreate

selector:

matchLabels:

app: web2-prod-app

template:

metadata:

labels:

app: web2-prod-app

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

1. **Apply the deployment**:

bash

Copy code

kubectl apply -f deploy.yaml --record=true

The --record=true flag records the current command as part of the rollout history.

1. **Verify the deployment and check the status**:

bash

Copy code

kubectl rollout status deployment gagan-app

1. **Check the running pods**:

bash

Copy code

kubectl get pods

1. **View deployment details**:

bash

Copy code

kubectl get deploy

1. **Describe the deployment to check the image used**:

bash

Copy code

kubectl describe deploy gagan-app | grep -i image

1. **Check the current ReplicaSet created by the deployment**:

bash

Copy code

kubectl get rs

**Step 2: Rolling Update to Change Image**

You will update the deployment to use a different version of the NGINX image.

1. **Edit the deployment YAML (deploy.yaml)** to update the image from nginx:latest to nginx:1.14:

yaml

Copy code

containers:

- name: nginx

image: nginx:1.14

ports:

- containerPort: 80

1. **Apply the updated deployment**:

bash

Copy code

kubectl apply -f deploy.yaml --record=true

1. **Check the rollout status**:

bash

Copy code

kubectl rollout status deployment gagan-app

1. **View rollout history**:

bash

Copy code

kubectl rollout history deployment gagan-app

1. **Check running ReplicaSets and ensure a new one is created**:

bash

Copy code

kubectl get rs

**Step 3: Rollout Another Update (nginx:1.14.1)**

1. **Edit the deployment file** to update the image to nginx:1.14.1:

yaml

Copy code

containers:

- name: nginx

image: nginx:1.14.1

ports:

- containerPort: 80

1. **Apply the updated deployment**:

bash

Copy code

kubectl apply -f deploy.yaml --record=true

1. **Check the rollout status**:

bash

Copy code

kubectl rollout status deployment gagan-app

1. **View the rollout history again**:

bash

Copy code

kubectl rollout history deployment gagan-app

1. **Confirm the new ReplicaSet has been created**:

bash

Copy code

kubectl get rs

**Step 4: Rollback to a Previous Revision**

If you need to rollback to a previous deployment version, Kubernetes supports that easily.

1. **Rollback to a specific revision (e.g., revision 1)**:

bash

Copy code

kubectl rollout undo deployment gagan-app --to-revision=1

1. **Check the rollout status after the rollback**:

bash

Copy code

kubectl rollout status deployment gagan-app

1. **Verify the image and other details of the current deployment**:

bash

Copy code

kubectl describe deploy gagan-app | grep -i image

1. **Check the ReplicaSet again to ensure the rollback occurred**:

bash

Copy code

kubectl get rs

**Step 5: Additional Rollout and Revision Management**

1. **View detailed rollout history for gagan-app**:

bash

Copy code

kubectl rollout history deployment gagan-app

1. **Describe a specific revision in the history**:

bash

Copy code

kubectl rollout history deployment gagan-app --revision=<revision-number>

1. **Get detailed information on ReplicaSets**:

bash

Copy code

kubectl get rs

**Step 6: Cleanup Resources**

1. **Delete the deployment**:

bash

Copy code

kubectl delete deployment gagan-app

1. **Verify that the deployment and associated ReplicaSets are removed**:

bash

Copy code

kubectl get deploy

kubectl get rs

**Commands Summary**

| **Command** | **Description** |
| --- | --- |
| kubectl apply -f deploy.yaml --record=true | Apply a deployment and record the command for rollout history |
| kubectl rollout status deployment gagan-app | Check the status of the rollout |
| kubectl rollout history deployment gagan-app | View the rollout history of the deployment |
| kubectl get rs | List all ReplicaSets |
| `kubectl describe deploy gagan-app | grep -i image` |
| kubectl rollout undo deployment gagan-app --to-revision=1 | Rollback to a specific revision |
| kubectl rollout history deployment gagan-app --revision=<revision-number> | View details of a specific revision |
| kubectl delete deployment gagan-app | Delete the deployment |

This lab guide covers the lifecycle of a Kubernetes deployment, including how to update images, monitor rollouts, manage rollback operations, and work with ReplicaSets. The example focuses on practical usage and best practices.

**Kubernetes Secrets and Pods Lab Guide**

**Objective:**

* Encrypt sensitive data (username and password).
* Create a Kubernetes secret from the encrypted data.
* Use the secret in two pods:
  + Mount the secret as a volume in one pod.
  + Use the secret as environment variables in another pod.

**Step 1: Encrypt Sensitive Data**

Kubernetes secrets store sensitive information (like passwords and API keys) in base64-encoded format. You will first encode your username and password.

1. **Base64 encode the password**:

bash

Copy code

echo 'ramankhanna123' | base64

Output:

bash

Copy code

cmFtYW5raGFubmExMjMK

1. **Base64 encode the username**:

bash

Copy code

echo 'ramankhanna' | base64

Output:

bash

Copy code

cmFtYW5raGFubmEK

**Step 2: Create a Kubernetes Secret YAML**

1. **Create a secret.yaml file** with the encoded values:

yaml

Copy code

apiVersion: v1

kind: Secret

metadata:

name: my-secrets

type: Opaque

data:

username: cmFtYW5raGFubmEK

password: cmFtYW5raGFubmExMjMK

1. **Apply the secret YAML** to create the secret in Kubernetes:

bash

Copy code

kubectl apply -f secret.yaml

1. **Verify the secret has been created**:

bash

Copy code

kubectl get secrets

1. **Describe the secret** to view detailed information:

bash

Copy code

kubectl describe secret my-secrets

1. **(Optional) View the secret in YAML format**:

bash

Copy code

kubectl get secret my-secrets -o yaml

Note: Kubernetes encodes the secret values using base64, and it's stored in the cluster.

**Step 3: Use Secret in Pods**

You will now create two pods: one that mounts the secret as a volume and one that uses the secret as environment variables.

**Pod 1: Mount Secret as a Volume**

1. **Create a secret-pod.yaml file** for the first pod (myapp-pod1), where the secret will be mounted as a volume:

yaml

Copy code

apiVersion: v1

kind: Pod

metadata:

name: myapp-pod1

labels:

app: myapp

spec:

containers:

- name: httpd-container

image: httpd

volumeMounts:

- name: credentials

mountPath: /tmp/creds

readOnly: true

volumes:

- name: credentials

secret:

secretName: my-secrets

1. **Apply the pod YAML to create the pod**:

bash

Copy code

kubectl apply -f secret-pod.yaml

1. **Verify that the pod is running**:

bash

Copy code

kubectl get pods

1. **Enter the pod and check the mounted secret**:

bash

Copy code

kubectl exec -it myapp-pod1 -- /bin/bash

1. **Navigate to the /tmp/creds directory** where the secret is mounted:

bash

Copy code

cd /tmp/creds

ls

You should see files named username and password.

1. **View the content of the username and password files**:

bash

Copy code

cat username

cat password

**Pod 2: Use Secret as Environment Variables**

1. **Extend the secret-pod.yaml file** to create the second pod (myapp-pod2), which uses the secret values as environment variables:

yaml

Copy code

---

apiVersion: v1

kind: Pod

metadata:

name: myapp-pod2

labels:

app: myapp

type: front-end

spec:

containers:

- name: httpd-container

image: httpd

env:

- name: SECRET\_USERNAME

valueFrom:

secretKeyRef:

name: my-secrets

key: username

- name: SECRET\_PASSWD

valueFrom:

secretKeyRef:

name: my-secrets

key: password

1. **Apply the pod YAML to create the second pod**:

bash

Copy code

kubectl apply -f secret-pod.yaml

1. **Verify that the pod is running**:

bash

Copy code

kubectl get pods

1. **Enter the pod and check the environment variables**:

bash

Copy code

kubectl exec -it myapp-pod2 -- /bin/bash

1. **Print the secret environment variables** inside the pod:

bash

Copy code

echo $SECRET\_USERNAME

echo $SECRET\_PASSWD

You should see the decoded values of ramankhanna and ramankhanna123.

**Step 4: Cleanup Resources**

1. **Delete the pods**:

bash

Copy code

kubectl delete pod myapp-pod1 myapp-pod2

1. **Delete the secret**:

bash

Copy code

kubectl delete secret my-secrets

**Commands Summary**

| **Command** | **Description** |
| --- | --- |
| `echo 'ramankhanna123' | base64` |
| `echo 'ramankhanna' | base64` |
| kubectl apply -f secret.yaml | Apply the secret file |
| kubectl get secrets | List all secrets |
| kubectl describe secret my-secrets | Describe the secret for details |
| kubectl apply -f secret-pod.yaml | Create the pod that mounts the secret or uses it as environment variables |
| kubectl exec -it myapp-pod1 -- /bin/bash | Enter the pod and check the mounted secret files |
| kubectl exec -it myapp-pod2 -- /bin/bash | Enter the pod and check the secret environment variables |
| kubectl delete pod myapp-pod1 myapp-pod2 | Delete the pods |
| kubectl delete secret my-secrets | Delete the secret |

This guide walks through encrypting sensitive data, creating a Kubernetes secret, and utilizing it in two ways within pods. You can either mount the secret as a volume or use it as environment variables.

4o

**Kubernetes ConfigMaps and Pods Lab Guide**

This lab demonstrates how to create and use ConfigMaps in Kubernetes, specifically to mount configuration files into containers. The guide covers creating ConfigMaps from files, using those ConfigMaps in two pods (prod-nginx and dev-nginx), and mounting them as volumes.

**Objective:**

* Create ConfigMaps from HTML files.
* Deploy two NGINX pods: one for production and one for development.
* Mount the ConfigMaps as volumes inside the NGINX containers, serving HTML content.

**Step 1: Create HTML Files**

Before creating the ConfigMaps, create two simple HTML files representing production and development environments.

1. **Create prod.html** for production:

bash

Copy code

echo "Hello from production" > prod.html

1. **Create dev.html** for development:

bash

Copy code

echo "Hello from dev" > dev.html

**Step 2: Create ConfigMaps from Files**

Use the HTML files to create Kubernetes ConfigMaps.

1. **Check existing ConfigMaps (optional)**:

bash

Copy code

kubectl get configmaps

1. **Create a ConfigMap for the production environment**:

bash

Copy code

kubectl create configmap prod.cmap --from-file=prod.html

1. **Create a ConfigMap for the development environment**:

bash

Copy code

kubectl create configmap dev.cmap --from-file=dev.html

1. **Verify the created ConfigMaps**:

bash

Copy code

kubectl get configmaps

1. **Check the contents of the production ConfigMap**:

bash

Copy code

kubectl get configmap prod.cmap -o yaml

1. **Check the contents of the development ConfigMap**:

bash

Copy code

kubectl get configmap dev.cmap -o yaml

**Step 3: Create Pods and Mount ConfigMaps as Volumes**

Next, you'll create two NGINX pods (prod-nginx and dev-nginx) that serve the content from the corresponding ConfigMaps.

1. **Create a pods.yaml file** with the following content:

yaml

Copy code

apiVersion: v1

kind: Pod

metadata:

name: prod-nginx

labels:

app: prod-nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

volumeMounts:

- name: config-volume

mountPath: /usr/share/nginx/html

volumes:

- name: config-volume

configMap:

name: prod.cmap

items:

- key: prod.html

path: index.html

---

apiVersion: v1

kind: Pod

metadata:

name: dev-nginx

labels:

app: dev-nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

volumeMounts:

- name: config-volume

mountPath: /usr/share/nginx/html

volumes:

- name: config-volume

configMap:

name: dev.cmap

items:

- key: dev.html

path: index.html

In this YAML:

* + The prod-nginx pod mounts the prod.html file from the prod.cmap ConfigMap to /usr/share/nginx/html/index.html.
  + The dev-nginx pod mounts the dev.html file from the dev.cmap ConfigMap to /usr/share/nginx/html/index.html.

1. **Apply the pods.yaml file to create the pods**:

bash

Copy code

kubectl apply -f pods.yaml

1. **Verify that the pods are running**:

bash

Copy code

kubectl get pods

**Step 4: Test the NGINX Pods**

1. **Get the pod IP addresses**:

bash

Copy code

kubectl get pods -o wide

1. **Access the production NGINX pod** (prod-nginx):
   * Open a web browser or use curl to visit the production NGINX pod’s IP:

bash

Copy code

curl http://<prod-nginx-pod-ip>

You should see:

csharp

Copy code

Hello from production

1. **Access the development NGINX pod** (dev-nginx):
   * Open a web browser or use curl to visit the development NGINX pod’s IP:

bash

Copy code

curl http://<dev-nginx-pod-ip>

You should see:

csharp

Copy code

Hello from dev

**Step 5: Cleanup Resources**

1. **Delete the pods**:

bash

Copy code

kubectl delete pod prod-nginx dev-nginx

1. **Delete the ConfigMaps**:

bash

Copy code

kubectl delete configmap prod.cmap dev.cmap

**Commands Summary**

| **Command** | **Description** |
| --- | --- |
| echo "Hello from production" > prod.html | Create the production HTML file |
| echo "Hello from dev" > dev.html | Create the development HTML file |
| kubectl create configmap prod.cmap --from-file=prod.html | Create the production ConfigMap from prod.html |
| kubectl create configmap dev.cmap --from-file=dev.html | Create the development ConfigMap from dev.html |
| kubectl get configmaps | List all ConfigMaps |
| kubectl get configmap prod.cmap -o yaml | View the production ConfigMap in YAML format |
| kubectl apply -f pods.yaml | Create the pods using the ConfigMaps as volumes |
| kubectl get pods -o wide | Get the pod details along with IP addresses |
| kubectl delete pod prod-nginx dev-nginx | Delete the pods |
| kubectl delete configmap prod.cmap dev.cmap | Delete the ConfigMaps |

**Conclusion:**

This lab guide demonstrated how to create ConfigMaps from files, mount them as volumes in pods, and verify that NGINX serves the content from the mounted ConfigMap files.

**Kubernetes PersistentVolume (PV) and PersistentVolumeClaim (PVC) Lab Guide**

This lab demonstrates how to use Persistent Volumes (PV) and Persistent Volume Claims (PVC) in Kubernetes for managing storage in a cluster. We will also create a deployment that uses a shared NFS (Network File System) as a storage backend.

**Objective:**

* Learn how to create Persistent Volumes (PVs) and Persistent Volume Claims (PVCs).
* Deploy pods and applications that consume shared storage using NFS.
* Mount NFS to the worker node and verify persistent storage usage.

**Prerequisites:**

* A Kubernetes cluster with multiple worker nodes.
* NFS server running, accessible via its IP.
* Security groups configured to allow NFS traffic on port 2049 between nodes and NFS server.

**Step 1: Create an NFS-backed PersistentVolume (PV)**

1. **Create a PersistentVolume YAML (pv.yaml)**:

yaml

Copy code

apiVersion: v1

kind: PersistentVolume

metadata:

name: nfs-website

spec:

capacity:

storage: 11Mi

accessModes:

- ReadWriteMany

mountOptions:

- hard

- nfsvers=4.1

nfs:

path: /

server: 172.31.12.142 # Replace with your NFS server IP

This PersistentVolume allows storage of up to 11Mi and can be accessed in ReadWriteMany mode. It uses an NFS server located at 172.31.12.142 and mounts the root directory /.

1. **Apply the PersistentVolume YAML**:

bash

Copy code

kubectl apply -f pv.yaml

1. **Verify the PersistentVolume**:

bash

Copy code

kubectl get pv

**Step 2: Create a PersistentVolumeClaim (PVC)**

1. **Create a PersistentVolumeClaim YAML (pvc.yaml)**:

yaml

Copy code

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: nfs-demo

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 5Mi

volumeName: nfs-website

This PVC requests 5Mi of storage with ReadWriteMany access from the PersistentVolume named nfs-website.

1. **Apply the PVC YAML**:

bash

Copy code

kubectl apply -f pvc.yaml

1. **Verify the PVC**:

bash

Copy code

kubectl get pvc

1. **Describe the PVC** to check its status and ensure it is bound to the PV:

bash

Copy code

kubectl describe pvc nfs-demo

**Step 3: Create a Deployment Using NFS-backed PVC**

Now, we'll create a deployment with multiple replicas of NGINX pods, each consuming the same NFS-backed volume via the PVC.

1. **Create a deployment YAML (deploy4.yaml)**:

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: nfs-raman

spec:

replicas: 10

selector:

matchLabels:

role: nfs-raman

template:

metadata:

labels:

role: nfs-raman

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

volumeMounts:

- name: nfs

mountPath: /usr/share/nginx/deploydata

volumes:

- name: nfs

persistentVolumeClaim:

claimName: nfs-demo

This deployment creates 10 replicas of NGINX containers. Each container will mount the NFS-backed PVC (nfs-demo) to /usr/share/nginx/deploydata.

1. **Apply the deployment YAML**:

bash

Copy code

kubectl apply -f deploy4.yaml

1. **Verify the deployment**:

bash

Copy code

kubectl get pods -o wide

**Step 4: Expose the Deployment**

1. **Expose the NGINX deployment as a NodePort service**:

bash

Copy code

kubectl expose deployment nfs-raman --name nfs-svc --type NodePort --target-port 80 --port 80

1. **Check the service**:

bash

Copy code

kubectl get svc

You will see the external IP and port assigned to the service. You can now access NGINX on the worker node's public IP and port.

1. **Test by browsing**:
   * Open a web browser and visit: http://<worker-node-public-ip>:<NodePort> to verify the service is running.

**Step 5: Mount NFS on Worker Node and Test Volumes**

1. **Install NFS client on the worker node**:

bash

Copy code

sudo apt-get install -y nfs-common

1. **Create a directory for mounting NFS**:

bash

Copy code

mkdir /efs

1. **Mount the NFS directory**:

bash

Copy code

sudo mount -t nfs4 -o nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=600,retrans=2,noresvport <NFS-server-IP>:/ /efs

Replace <NFS-server-IP> with your NFS server’s IP (e.g., 172.31.12.142).

1. **Verify the mount**:

bash

Copy code

df -h

The NFS directory should be listed as mounted.

1. **Check the contents of the NFS mount**:

bash

Copy code

cd /efs

ls -l

Any data written by the NGINX pods to /usr/share/nginx/deploydata will appear here.

**Step 6: Interact with the Pods**

1. **Exec into a pod** to check the volume:

bash

Copy code

kubectl exec -it <pod-name> -- /bin/bash

1. **List the contents of the mounted volume inside the pod**:

bash

Copy code

cd /usr/share/nginx/deploydata

ls

Any data shared across pods through the NFS volume should appear here.

**Step 7: Cleanup**

1. **Delete the deployment**:

bash

Copy code

kubectl delete deployment nfs-raman

1. **Delete the service**:

bash

Copy code

kubectl delete svc nfs-svc

1. **Delete the PVC**:

bash

Copy code

kubectl delete pvc nfs-demo

1. **Delete the PV**:

bash

Copy code

kubectl delete pv nfs-website

**Commands Summary**

| **Command** | **Description** |
| --- | --- |
| kubectl apply -f pv.yaml | Create the NFS-backed PersistentVolume |
| kubectl apply -f pvc.yaml | Create the PersistentVolumeClaim |
| kubectl get pv | List PersistentVolumes |
| kubectl get pvc | List PersistentVolumeClaims |
| kubectl describe pvc nfs-demo | Check details of the PVC |
| kubectl apply -f deploy4.yaml | Deploy NGINX with PVC |
| kubectl expose deployment nfs-raman | Expose the NGINX deployment as a service |
| kubectl get svc | Get service details |
| sudo mount -t nfs4 ... | Mount the NFS share on a worker node |
| kubectl exec -it <pod-name> -- /bin/bash | Exec into a running pod |
| kubectl delete deployment nfs-raman | Delete the deployment |
| kubectl delete pvc nfs-demo | Delete the PVC |
| kubectl delete pv nfs-website | Delete the PV |

**Conclusion:**

This guide demonstrated how to set up Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) using an NFS server. It also covered deploying applications that use shared storage, exposing services, and mounting NFS on worker nodes.