Kubernetes/AKS Documentation

Table of Contents

1. Introduction

This document provides a comprehensive guide to various Kubernetes and Azure Kubernetes Service (AKS) concepts and tasks. It is designed to offer detailed, step-by-step instructions for users operating on a WSL-based Windows system. The aim is to demystify complex Kubernetes and AKS functionalities, enabling users to effectively deploy, manage, and scale containerized applications.

2. System Requirements

To follow the instructions and perform the tasks outlined in this documentation, you will need the following:

- **Operating System:** Windows 10/11 with Windows Subsystem for Linux (WSL) installed and configured.
- **WSL Distribution:** A Linux distribution installed on WSL (e.g., Ubuntu).
- **kubectl:** The Kubernetes command-line tool, installed and configured to interact with your Kubernetes clusters.
- **Azure CLI:** The Azure command-line interface, installed and logged in to your Azure account (for AKS-related tasks).
- **Docker Desktop:** For local Kubernetes development and testing, Docker Desktop with Kubernetes enabled is recommended.
- **Internet Connection:** Required for downloading necessary tools, images, and accessing Azure services.
- **Azure Subscription:** An active Azure subscription with sufficient permissions to create and manage AKS clusters and related resources.

11. Conclusion

This documentation has provided an in-depth exploration of key Kubernetes and Azure Kubernetes Service (AKS) concepts, ranging from fundamental workload management with ReplicaSets and Deployments to advanced topics like autoscaling and persistent storage. By following the detailed instructions and understanding the underlying principles, users on WSL-based Windows systems can effectively deploy, manage, and scale their containerized applications in a Kubernetes environment. The learning outcomes include a solid grasp of core Kubernetes objects, service exposure mechanisms, persistent data management, and the operational aspects of managing AKS clusters, empowering users to build and maintain robust, scalable, and highly available applications.

12. References

- Kubernetes Replication Controller, Replica Set & Deployments
- Deployments, ReplicaSet, ReplicationController, Namespaces
- Replication Controller VS Deployment in Kubernetes - Stack Overflow
- Kubernetes ReplicaSet: Overview & How It Works
- ReplicaSet - Kubernetes
- Deployments | Kubernetes
- ReplicationController - Kubernetes
- Kubernetes Services: ClusterIP, Nodeport and LoadBalancer
- Service | Kubernetes
- The Difference Between ClusterIP, NodePort, And LoadBalancer
- ClusterIP, NodePort, and LoadBalancer: Kubernetes Service Types
- Persistent Volumes - Kubernetes
- Kubernetes Persistent Volumes - Tutorial and Examples
- Kubernetes Persistent Volume Claims Explained
- Azure Kubernetes Service (AKS)
- What is Azure Kubernetes Service (AKS)?
- Tutorial - Create an Azure Kubernetes Service (AKS) cluster
- Manually scale nodes in an Azure Kubernetes Service (AKS) cluster
- Upgrade options and recommendations for Azure Kubernetes Service (AKS)
- Configure Liveness, Readiness and Startup Probes - Kubernetes
- Add health probes to your AKS pods
- Taints and Tolerations | Kubernetes
- Use node taints in an Azure Kubernetes Service (AKS) cluster
- Configure a Pod to Use a PersistentVolume for Storage - Kubernetes
- Horizontal Pod Autoscaling - Kubernetes
- Use the cluster autoscaler in Azure Kubernetes Service (AKS)
- Scaling options for applications in Azure Kubernetes Service (AKS)

3. Deploy Replica Set, Replication Controller, and Deployment

This section details the deployment of Replica Sets, Replication Controllers, and Deployments in Kubernetes, highlighting their functionalities, advantages, and disadvantages. While Replication Controllers are an older construct, understanding them provides context for the evolution of Kubernetes workload management. ReplicaSets are a more flexible successor, and Deployments offer a higher-level abstraction for managing both.

3.1. Understanding the Concepts

Replication Controller (RC)

A Replication Controller ensures that a specified number of pod replicas are running at any given time. If too many pods are running, the Replication Controller terminates the excess. If too few are running, it starts more. RCs use equality-based selectors to manage pods, meaning they match pods based on exact label matches [1, 3].

Advantages: * **High Availability:** Ensures a desired number of pods are always available, providing basic self-healing capabilities. * **Load Distribution:** Distributes traffic across multiple pod replicas.

Disadvantages: * **Limited Selector Support:** Only supports equality-based selectors, which can be restrictive. * **Superseded:** Largely superseded by ReplicaSets and Deployments, offering fewer features and less flexibility [2, 6].

ReplicaSet (RS)

A ReplicaSet is the next-generation Replication Controller. Its primary purpose is to maintain a stable set of replica Pods running at any given time. The key difference from Replication Controllers is that ReplicaSets support set-based selectors, allowing for more complex matching conditions (e.g., matching pods with labels `app in (nginx, apache)`) [2, 4, 10].

Advantages: * **Enhanced Selector Support:** Supports set-based selectors, providing greater flexibility in defining pod sets. * **High Availability:** Similar to RCs, ensures the desired number of pod replicas are maintained. * **Foundation for Deployments:**

ReplicaSets are the building blocks for Deployments, which manage them automatically [5, 6].

Disadvantages: * **Lower-Level Abstraction:** While more powerful than RCs, ReplicaSets are still a lower-level construct. For advanced features like rolling updates and rollbacks, Deployments are preferred [8, 9]. * **Direct Interaction Rare:** You typically don't interact with ReplicaSets directly; Deployments manage them on your behalf [5].

Deployment

A Deployment provides declarative updates for Pods and ReplicaSets. It is a higher-level abstraction that allows you to describe the desired state of your application, and the Deployment Controller changes the actual state to the desired state at a controlled rate. Deployments are the recommended way to manage stateless applications in Kubernetes [6, 9].

Advantages: * **Declarative Updates:** Define the desired state, and Kubernetes handles the transition. * **Rolling Updates:** Allows for zero-downtime updates by gradually replacing old pods with new ones. * **Rollbacks:** Easily revert to a previous version of your application if an update causes issues. * **Pause/Resume:** Ability to pause and resume deployments for controlled updates. * **Manages ReplicaSets:** Automatically creates and manages ReplicaSets to achieve the desired number of replicas [5, 10]. * **Self-Healing:** Inherits self-healing capabilities from ReplicaSets.

Disadvantages: * **Not for Stateful Applications:** While powerful for stateless applications, Deployments are not ideal for stateful applications that require persistent storage and unique network identities (for which StatefulSets are used) [9].

3.2. Deployment Steps on WSL-based Windows System

Before proceeding, ensure you have `kubectl` configured to connect to your Kubernetes cluster (e.g., a local Docker Desktop Kubernetes instance or an AKS cluster). You will also need a text editor to create the YAML manifest files.

Step 1: Create a YAML file for a Replication Controller (Optional - for understanding only)

While not recommended for new deployments, understanding Replication Controllers is crucial for historical context. Create a file named `rc-example.yaml` with the

following content:

```
 apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx-rc
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Step 2: Apply the Replication Controller (Optional)

Open your WSL terminal and apply the YAML file:

```
kubectl apply -f rc-example.yaml
```

Step 3: Verify the Replication Controller and Pods (Optional)

Check the status of the Replication Controller and the created pods:

```
 kubectl get rc
kubectl get pods -l app=nginx
```

Step 4: Delete the Replication Controller (Optional)

To clean up, delete the Replication Controller:

```
kubectl delete -f rc-example.yaml
```

Step 5: Create a YAML file for a ReplicaSet

Create a file named `rs-example.yaml` with the following content. Notice the `apiVersion` and `selector` differences compared to the Replication Controller:

```yaml
 apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Step 6: Apply the ReplicaSet

Open your WSL terminal and apply the YAML file:

```
kubectl apply -f rs-example.yaml
```

Step 7: Verify the ReplicaSet and Pods

Check the status of the ReplicaSet and the created pods:

```
 kubectl get rs
kubectl get pods -l app=nginx
```

Step 8: Delete the ReplicaSet

To clean up, delete the ReplicaSet:

```
kubectl delete -f rs-example.yaml
```

Step 9: Create a YAML file for a Deployment (Recommended Approach)

Create a file named `deployment-example.yaml` with the following content. This is the most common and recommended way to deploy applications in Kubernetes:

```yaml
 apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Step 10: Apply the Deployment

Open your WSL terminal and apply the YAML file:

```
kubectl apply -f deployment-example.yaml
```

Step 11: Verify the Deployment, ReplicaSet, and Pods

Check the status of the Deployment, the ReplicaSet it created, and the pods:

```
 kubectl get deployment
kubectl get rs -l app=nginx
kubectl get pods -l app=nginx
```

Step 12: Scale the Deployment

To scale the number of replicas, you can edit the deployment YAML or use the `kubectl scale` command:

```
kubectl scale deployment/nginx-deployment --replicas=5
```

Step 13: Perform a Rolling Update

To perform a rolling update, simply change the image version in your `deployment-example.yaml` file (e.g., from `nginx:latest` to `nginx:1.21.6`) and apply it again:

```
# ... (previous content)
      containers:
      - name: nginx
        image: nginx:1.21.6  # Changed image version
        ports:
        - containerPort: 80
```

Then apply the updated YAML:

```
kubectl apply -f deployment-example.yaml
```

Observe how Kubernetes gradually replaces the old pods with new ones, ensuring no downtime.

Step 14: Rollback a Deployment

If an update causes issues, you can easily roll back to a previous revision:

```
kubectl rollout history deployment/nginx-deployment
kubectl rollout undo deployment/nginx-deployment --to-revision=<revision-
number>
```

Step 15: Delete the Deployment

To clean up, delete the Deployment:

```
kubectl delete -f deployment-example.yaml
```

3.3. Illustrative Images

(Illustrative images for WSL terminal commands and Kubernetes resource states will be inserted here.)

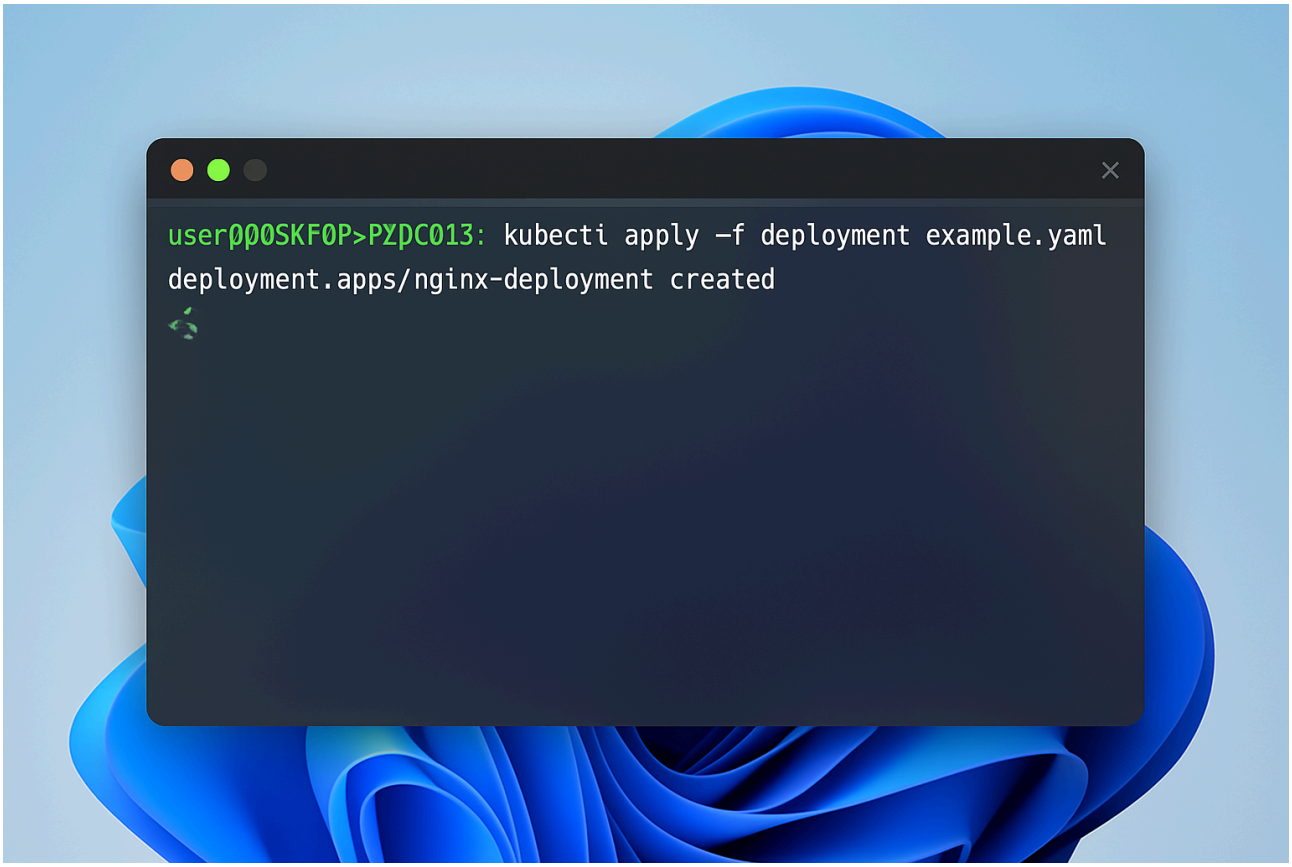

Figure 3.1: Applying a Kubernetes Deployment using kubectl in a WSL terminal.

4. Kubernetes Service Types (ClusterIP, NodePort, LoadBalancer)

Kubernetes Services are an abstract way to expose an application running on a set of Pods as a network service. They define a logical set of Pods and a policy by which to access them. Understanding the different service types is crucial for controlling how your applications are exposed both internally within the cluster and externally to the outside world [11, 12].

4.1. Understanding the Concepts

ClusterIP

This is the default Kubernetes Service type. A ClusterIP Service exposes the Service on an internal IP in the cluster. This means the Service is only reachable from within the Kubernetes cluster. It provides a stable virtual IP address that other Pods can use to communicate with the Service [13, 14].

Use Cases: * Internal communication between different microservices within your Kubernetes cluster. * Backend services that do not need to be exposed directly to external users.

NodePort

A NodePort Service exposes the Service on each Node’ s IP at a static port (the NodePort). A ClusterIP Service is automatically created, and the NodePort Service routes to it. This allows external traffic to reach your Service by accessing any Node's IP address on the specified NodePort [15, 16].

Use Cases: * Exposing services to external traffic for development or testing purposes. * When you need to expose a service on a specific port across all nodes and do not have a cloud load balancer.

LoadBalancer

A LoadBalancer Service exposes the Service externally using a cloud provider’ s load balancer. NodePort and ClusterIP Services are automatically created, and the external load balancer routes traffic to them. When you create a LoadBalancer Service in a cloud environment (like Azure, AWS, or GCP), the cloud provider provisions an external load balancer with a public IP address that directs traffic to your Service [17, 18].

Use Cases: * Exposing services to the internet for production applications. * When you require a dedicated, external IP address for your service and automatic load distribution across your Pods.

4.2. Comparison of Service Types

Feature	ClusterIP	NodePort	LoadBalancer
Visibility	Internal to the cluster	External, via Node IP and static port	External, via cloud provider's load balancer
Access	Only from within the cluster	From outside the cluster (NodeIP:NodePort)	From the internet (external IP)
Cost	No additional cost	No additional cost	May incur costs from cloud provider
Use Case	Internal microservices communication	Development, testing, on-premise exposure	Production deployments in cloud environments

4.3. Deployment Steps on WSL-based Windows System

This section will demonstrate how to create and expose services using each type. Ensure you have `kubectl` configured to interact with your Kubernetes cluster.

Step 1: Create a Deployment (if you don't have one)

First, let's ensure we have a running application to expose. We'll use the `nginx-deployment` created in the previous section. If you deleted it, recreate it using the `deployment-example.yaml` file:

```
 apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Apply the deployment:

```
kubectl apply -f deployment-example.yaml
```

Step 2: Create a ClusterIP Service

Create a file named `clusterip-service.yaml`:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-clusterip-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

Apply the service:

```
kubectl apply -f clusterip-service.yaml
```

Verify the ClusterIP Service:

```
kubectl get service nginx-clusterip-service
```

You will see an `CLUSTER-IP` assigned. This IP is only accessible from within the cluster.

Step 3: Create a NodePort Service

Create a file named `nodeport-service.yaml`:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-nodeport-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30000 # Optional: specify a port between 30000-32767
  type: NodePort
```

Apply the service:

```
kubectl apply -f nodeport-service.yaml
```

Verify the NodePort Service:

```
kubectl get service nginx-nodeport-service
```

You will see an `EXTERNAL-IP` (if available, otherwise `<pending>`) and a `PORT(S)` showing `80:30000/TCP` (or your chosen `nodePort`). You can access the service from outside the cluster using any Node's IP address and the `nodePort` (e.g., `http://<NodeIP>:30000`).

Step 4: Create a LoadBalancer Service (Requires a Cloud Provider like Azure AKS)

If you are running on a local Kubernetes (e.g., Docker Desktop), this step will likely show the `EXTERNAL-IP` as `<pending>` as there is no cloud load balancer provisioner. For a real LoadBalancer, you need an AKS cluster.

Create a file named `loadbalancer-service.yaml`:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-loadbalancer-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer
```

Apply the service:

```
kubectl apply -f loadbalancer-service.yaml
```

Verify the LoadBalancer Service:

```
kubectl get service nginx-loadbalancer-service
```

On an AKS cluster, after a short while, an `EXTERNAL-IP` will be assigned. You can then access your Nginx application using this public IP address.

Step 5: Clean Up Services

Delete the services:

```
kubectl delete -f clusterip-service.yaml
kubectl delete -f nodeport-service.yaml
kubectl delete -f loadbalancer-service.yaml
kubectl delete -f deployment-example.yaml # Delete the deployment as well
```

4.4. Illustrative Images

(Illustrative images for WSL terminal commands and Kubernetes service states will be inserted here.)

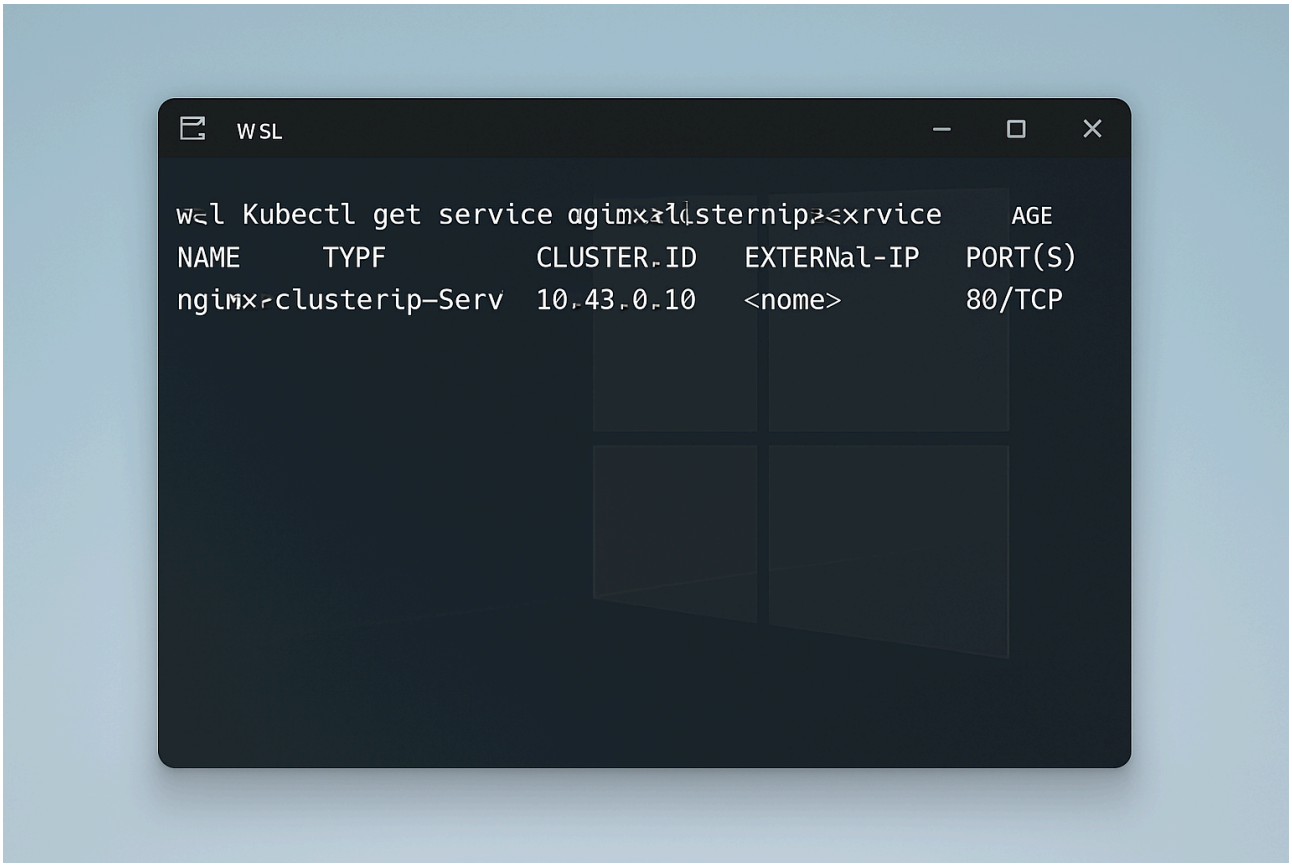

Figure 4.1: Verifying a ClusterIP Service in a WSL terminal.

5. PersistentVolume (PV) and PersistentVolumeClaim (PVC)

Kubernetes provides abstractions for managing storage, allowing users to consume storage resources without knowing the underlying infrastructure details. This is achieved through PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs). This separation allows developers to request storage without needing to know the specifics

of the underlying storage infrastructure, and administrators to manage storage resources independently of application deployments [19, 20, 21].

5.1. Understanding the Concepts

PersistentVolume (PV)

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a cluster-wide resource, independent of any specific Pod. PVs are defined with specific storage capacity, access modes (e.g., `ReadWriteOnce`, `ReadOnlyMany`, `ReadWriteMany`), and reclaim policies (e.g., `Retain`, `Recycle`, `Delete`). PVs have a lifecycle independent of Pods. They can be provisioned statically (by an administrator) or dynamically (by a StorageClass) [19, 20].

PersistentVolumeClaim (PVC)

A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod, which consumes Node resources. PVCs consume PV resources. Users describe the storage they need (e.g., amount of storage, access modes) in a PVC, and Kubernetes finds a suitable PV to bind to it. This abstracts the underlying storage details from the application. Kubernetes attempts to bind a PVC to an available PV that satisfies the PVC's requirements. Once bound, the PV is exclusively reserved for that PVC [19, 21].

5.2. Relationship between PV and PVC

PVs are the actual storage resources, while PVCs are requests for those resources. A PVC acts as a claim on a PV. Pods then use PVCs to access the persistent storage. This abstraction allows applications to be portable across different storage environments without needing to change their configuration, as long as the underlying Kubernetes cluster can provide the requested storage.

5.3. Deployment Steps on WSL-based Windows System

This section will guide you through creating a PersistentVolume (PV) and a PersistentVolumeClaim (PVC), and then attaching the PVC to a Pod. For demonstration purposes, we will use a `hostPath` PV, which mounts a directory from the host node. In a production environment, you would typically use cloud-provider specific storage (e.g., Azure Disk, AWS EBS) provisioned dynamically via a StorageClass.

Step 1: Create a directory on your WSL filesystem for the PV

Open your WSL terminal and create a directory that will serve as the persistent storage. For example:

```
mkdir -p /mnt/data/pv-data
```

Step 2: Define a PersistentVolume (PV)

Create a file named `pv-example.yaml` with the following content. This PV will use the `hostPath` type, pointing to the directory you just created.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: manual
  hostPath:
    path: "/mnt/data/pv-data"
```

Apply the PV:

```
kubectl apply -f pv-example.yaml
```

Step 3: Verify the PersistentVolume

Check the status of the PV:

```
kubectl get pv my-pv
```

It should show a status of `Available`.

Step 4: Define a PersistentVolumeClaim (PVC)

Create a file named `pvc-example.yaml`:

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: manual
```

Apply the PVC:

```
kubectl apply -f pvc-example.yaml
```

Step 5: Verify the PersistentVolumeClaim

Check the status of the PVC. It should show a status of `Bound` to `my-pv`.

```
kubectl get pvc my-pvc
```

Step 6: Attach the PVC to a Pod

Create a file named `pod-with-pvc.yaml` that references the `my-pvc`:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod-with-pvc
spec:
  containers:
    - name: my-container
      image: nginx
      ports:
        - containerPort: 80
      volumeMounts:
        - name: persistent-storage
          mountPath: /usr/share/nginx/html
  volumes:
    - name: persistent-storage
      persistentVolumeClaim:
        claimName: my-pvc
```

Apply the Pod:

```
kubectl apply -f pod-with-pvc.yaml
```

Step 7: Verify the Pod and write data to the volume

Check the status of the Pod:

```
kubectl get pod my-pod-with-pvc
```

Once the Pod is running, you can exec into it and write some data to the mounted volume:

```
kubectl exec -it my-pod-with-pvc -- bash
echo "Hello from persistent storage!" > /usr/share/nginx/html/index.html
exit
```

Now, if you access the Nginx service (if exposed), you should see the content. More importantly, if you delete and recreate the Pod, the data should persist.

Step 8: Clean Up

Delete the Pod, PVC, and PV:

```
kubectl delete -f pod-with-pvc.yaml
kubeclt delete -f pvc-example.yaml
kubeclt delete -f pv-example.yaml
```

Note: Due to the `Retain` reclaim policy on the PV, the underlying `hostPath` directory (`/mnt/data/pv-data`) will not be deleted automatically. You will need to manually remove it if desired.

5.4. Illustrative Images

(Illustrative images for WSL terminal commands and Kubernetes PV/PVC/Pod states will be inserted here.)

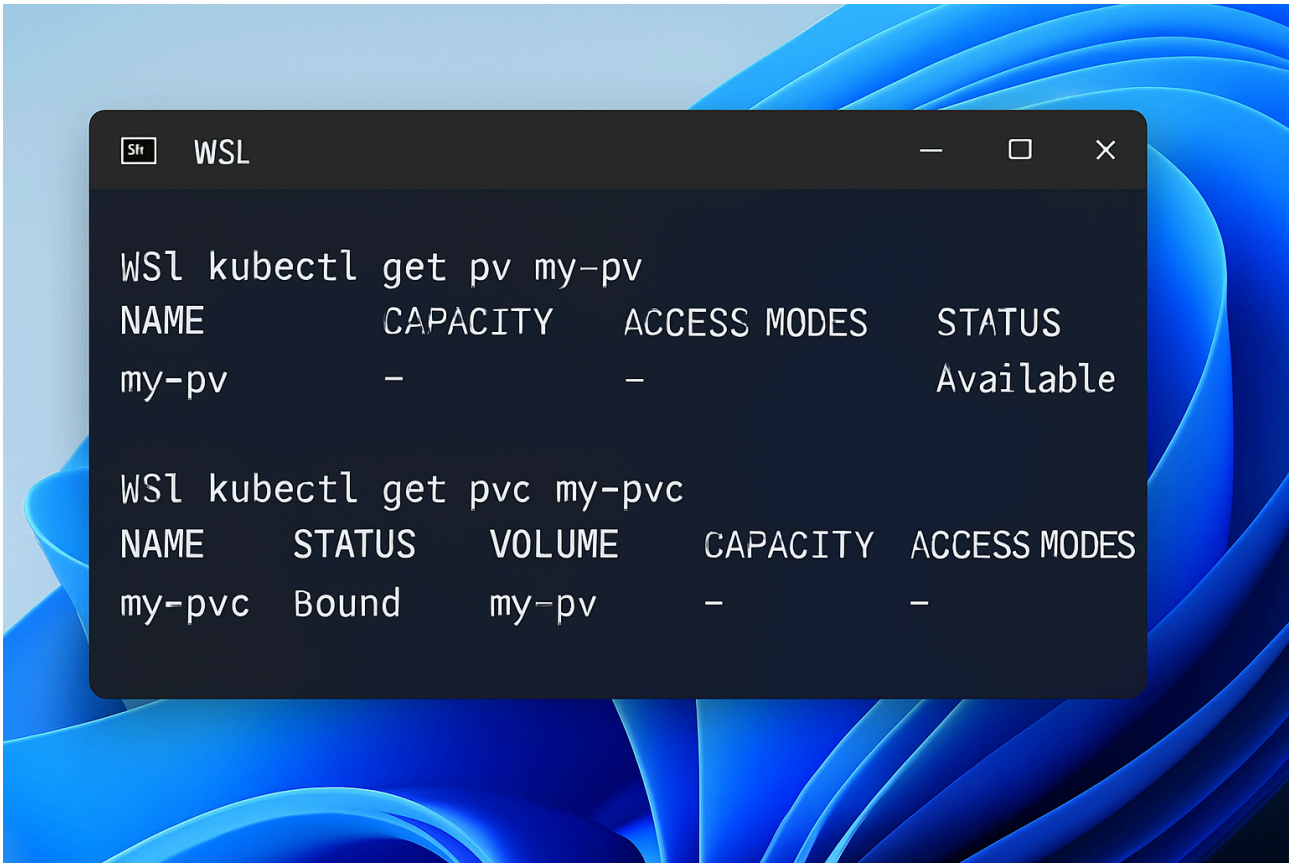

Figure 5.1: Verifying PersistentVolume and PersistentVolumeClaim status in a WSL terminal.

6. Managing Kubernetes with Azure Kubernetes Service (AKS)

Azure Kubernetes Service (AKS) is a fully managed Kubernetes service that simplifies the deployment, management, and operations of Kubernetes clusters in Azure. AKS handles critical tasks like health monitoring and maintenance, allowing users to focus on their applications rather than on infrastructure management. This section will cover the creation, management, scaling, and upgrading of AKS clusters [22, 23].

6.1. Creating and Managing AKS Clusters

Creating an AKS cluster involves defining various parameters such as the resource group, cluster name, Kubernetes version, and node pool configurations. Management includes monitoring, connecting to the cluster, and performing administrative tasks.

Step 1: Install Azure CLI and Log In

If you haven't already, install the Azure CLI on your WSL environment. Then, log in to your Azure account:

```
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
az login
```

Follow the on-screen instructions to complete the login process through your web browser.

Step 2: Create an Azure Resource Group

A resource group is a logical container for Azure resources. Create one for your AKS cluster:

```
az group create --name myAKSResourceGroup --location eastus
```

Step 3: Create an AKS Cluster

Now, create the AKS cluster within the resource group. This command creates a basic cluster with a single node pool and default settings. For production environments, you would customize these parameters further [24, 25].

```
az aks create --resource-group myAKSResourceGroup --name myAKSCluster --node-count 1 --enable-addons monitoring --generate-ssh-keys
```

This command might take several minutes to complete. The `--node-count 1` specifies one node in the default node pool, `--enable-addons monitoring` enables Azure Monitor for containers, and `--generate-ssh-keys` creates SSH keys for you if they don't exist.

Step 4: Get Cluster Credentials

Once the cluster is created, configure `kubectl` to connect to your new AKS cluster by downloading its credentials:

```
az aks get-credentials --resource-group myAKSResourceGroup --name myAKSCluster
```

This command merges the cluster's credentials into your `~/.kube/config` file, allowing `kubectl` to interact with your AKS cluster.

Step 5: Verify Cluster Connection

Check the nodes in your AKS cluster to confirm a successful connection:

```
kubectl get nodes
```

6.2. Scaling AKS Clusters

Scaling in AKS involves adjusting the number of nodes in your cluster or the number of pod replicas for your applications to meet changing demands. AKS supports both manual and automatic scaling [26, 28].

Step 1: Manually Scale the Node Count

You can manually increase or decrease the number of nodes in a specific node pool using the Azure CLI. For example, to scale the default node pool to 2 nodes:

```
az aks scale --resource-group myAKSResourceGroup --name myAKSCluster --node-
count 2
```

Step 2: Enable Cluster Autoscaler (Recommended for Automatic Node Scaling)

For automatic scaling of nodes based on resource demand, enable the Cluster Autoscaler. This will automatically adjust the number of nodes in your node pools within a specified range [28].

```
az aks update --resource-group myAKSResourceGroup --name myAKSCluster --
enable-cluster-autoscaler --min-count 1 --max-count 3
```

This command enables the cluster autoscaler on the default node pool, allowing it to scale between 1 and 3 nodes.

Step 3: Deploy an Application for Horizontal Pod Autoscaling (HPA)

To demonstrate HPA, deploy a sample application (e.g., an Nginx deployment) and define resource requests for its containers. Create `nginx-hpa-deployment.yaml`:

```
 apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-hpa-deployment
spec:
  selector:
    matchLabels:
      app: nginx-hpa
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx-hpa
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          requests:
            cpu: 100m
```

Apply the deployment:

```
kubectl apply -f nginx-hpa-deployment.yaml
```

Step 4: Create a Horizontal Pod Autoscaler (HPA)

Create an HPA that targets the `nginx-hpa-deployment` and scales based on CPU utilization. This HPA will try to maintain an average CPU utilization of 50% across the pods, scaling between 1 and 5 replicas [28].

```
 kubectl autoscale deployment nginx-hpa-deployment --cpu-percent=50 --min=1 --
max=5
```

Step 5: Generate Load to Test HPA

To see the HPA in action, you can generate some load on the Nginx application. First, get the name of one of the Nginx pods:

```
kubectl get pods -l app=nginx-hpa
```

Then, use `kubectl run` to create a temporary pod that sends continuous requests to the Nginx service. Replace `<nginx-pod-name>` with the actual pod name you retrieved:

```
kubectl run -it --rm load-generator --image=busybox -- /bin/sh -c "while true; do wget -q -O- http://nginx-hpa-deployment; done"
```

After a few minutes, observe the HPA. You can check its status:

```
kubectl get hpa
```

You should see the `REPLICAS` count increase as the CPU utilization rises. Once you stop the `load-generator` pod (by pressing `Ctrl+C` in its terminal), the HPA will eventually scale down the Nginx pods.

6.3. Upgrading AKS Clusters

Regularly upgrading your AKS clusters is crucial for security, new features, and bug fixes. AKS provides a managed upgrade experience for both the Kubernetes control plane and the node images [27].

Step 1: Check Available Upgrade Versions

Before upgrading, check which Kubernetes versions are available for your cluster:

```
az aks get-upgrades --resource-group myAKSResourceGroup --name myAKSCluster --output table
```

Step 2: Upgrade the AKS Cluster

To upgrade your AKS cluster to a specific version (e.g., `1.28.5`), use the `az aks upgrade` command:

```
az aks upgrade --resource-group myAKSResourceGroup --name myAKSCluster --kubernetes-version 1.28.5
```

This command upgrades both the control plane and the node images in your cluster. The process involves cordoning and draining nodes, upgrading them, and then uncordoning them, aiming for minimal downtime.

Step 3: Verify the Upgrade

After the upgrade command completes, verify the cluster's Kubernetes version:

```
az aks show --resource-group myAKSResourceGroup --name myAKSCluster --query
kubernetesVersion --output tsv
```

And check the node versions:

```
kubectl get nodes -o wide
```

6.4. Clean Up AKS Resources

To avoid incurring unnecessary costs, delete the resource group containing your AKS cluster and all associated resources when you are finished:

```
az group delete --name myAKSResourceGroup --yes --no-wait
```

6.5. Illustrative Images

(Illustrative images for Azure CLI commands and AKS cluster states will be inserted here.)

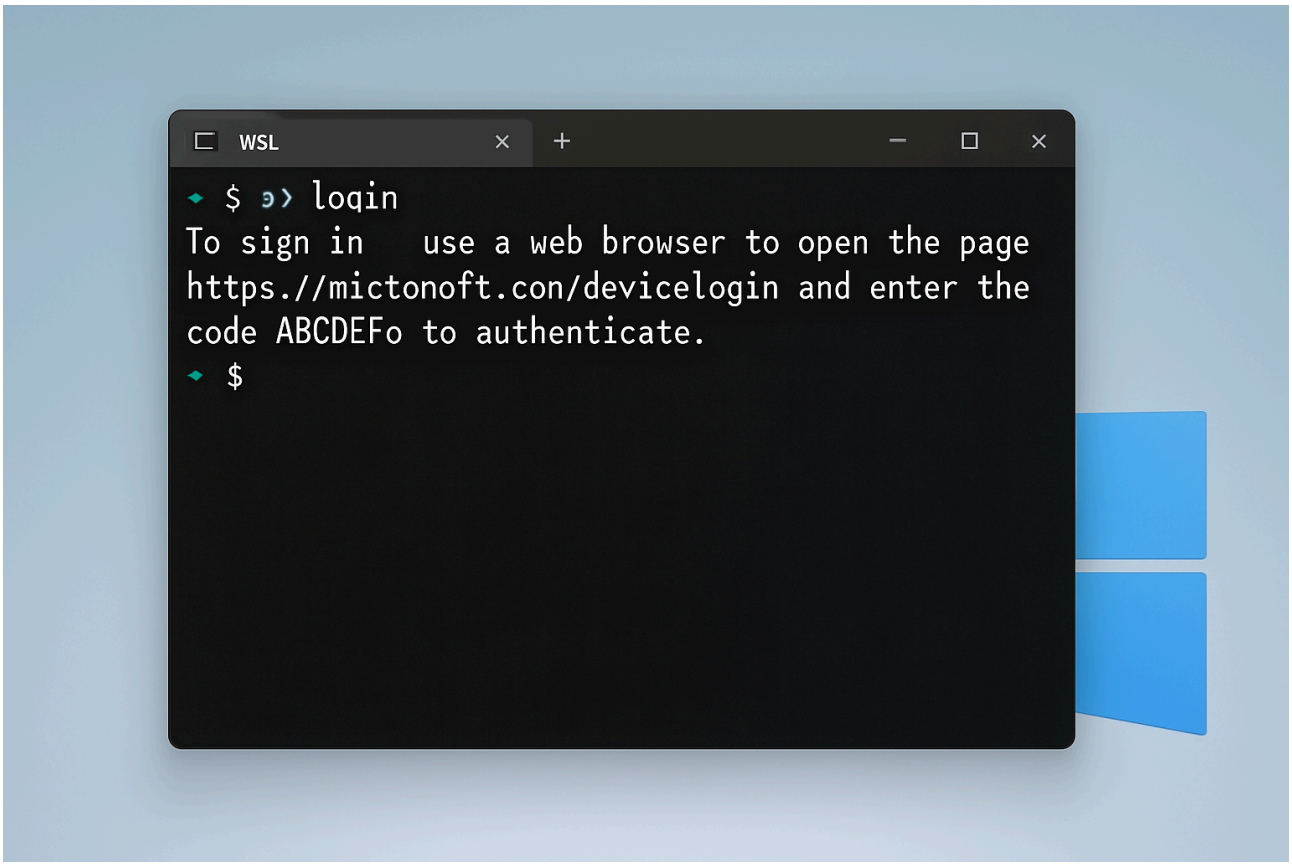

Figure 6.1: Logging into Azure CLI from a WSL terminal.

7. Configure Liveness and Readiness Probes for Pods in AKS Cluster

Kubernetes uses liveness and readiness probes to manage the health and availability of containers within Pods. These probes allow Kubernetes to react to the state of your applications, ensuring that only healthy Pods receive traffic and that unhealthy Pods are restarted. Implementing these probes is a critical best practice for building robust and resilient applications in Kubernetes and AKS [29, 30].

7.1. Understanding the Concepts

Liveness Probe

A liveness probe tells Kubernetes when to restart a container. If the liveness probe fails, Kubernetes restarts the container. This is particularly useful for catching deadlocks, where an application might be running but is unable to make progress (e.g., due to an internal error or resource exhaustion). Without a liveness probe, such a container would remain in a running state, consuming resources but providing no value [29].

Types of Liveness Probes: * **HTTP GET:** Kubernetes sends an HTTP GET request to a specified path on the container’s IP address and port. A successful response (status code 200-399) indicates health. This is suitable for web applications. * **TCP Socket:** Kubernetes attempts to open a TCP socket on a specified port. If the connection is established, the probe is considered successful. This is useful for applications that expose a TCP port but don't have an HTTP endpoint. * **Exec:** Kubernetes executes a command inside the container. The probe is successful if the command exits with a status code of 0. This is highly flexible and can be used for custom health checks.

Configuration Parameters: * `initialDelaySeconds`: The number of seconds after the container has started before liveness probes are initiated. This gives the application time to start up. * `periodSeconds`: How often (in seconds) to perform the probe. Defaults to 10 seconds. * `timeoutSeconds`: The number of seconds after which the probe times out. If the probe takes longer than this, it is considered a failure. Defaults to 1 second. * `successThreshold`: The minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. * `failureThreshold`: The minimum consecutive failures for the probe to be considered failed before Kubernetes restarts the container. Defaults to 3.

Readiness Probe

A readiness probe tells Kubernetes when a container is ready to start accepting traffic. If the readiness probe fails, Kubernetes removes the Pod’ s IP address from the Endpoints of all Services, preventing traffic from being sent to it. Once the probe succeeds, the Pod is added back to the Service Endpoints. This ensures that traffic is only routed to fully functional instances, preventing requests from being sent to applications that are still initializing or are temporarily unhealthy [29].

Use Case: * Applications that need time to warm up, load data, or connect to external services before they can serve requests. * Ensuring that during rolling updates, new Pods only receive traffic once they are fully ready.

Types of Readiness Probes: Same as liveness probes (HTTP GET, TCP Socket, Exec).

Configuration Parameters: Similar to liveness probes (`initialDelaySeconds`, `periodSeconds`, `timeoutSeconds`, `successThreshold`, `failureThreshold`).

7.2. Deployment Steps on WSL-based Windows System

This section will demonstrate how to configure liveness and readiness probes for a simple Nginx deployment. Ensure you have `kubectl` configured to interact with your Kubernetes cluster (e.g., an AKS cluster or Docker Desktop Kubernetes).

Step 1: Create a Deployment with Liveness and Readiness Probes

Create a file named `nginx-probes-deployment.yaml` with the following content. This example uses HTTP GET probes, which are common for web servers.

```yaml
 apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-probes-deployment
  labels:
    app: nginx-probes
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx-probes
  template:
    metadata:
      labels:
        app: nginx-probes
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
        livenessProbe:
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
          failureThreshold: 3
        readinessProbe:
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 10
          periodSeconds: 5
          failureThreshold: 3
```

In this configuration: * The `livenessProbe` checks the `/` path on port 80 every 5 seconds, starting 5 seconds after the container starts. If it fails 3 consecutive times, the container will be restarted. * The `readinessProbe` also checks the `/` path on port 80 every 5 seconds, but it waits 10 seconds before starting. If it fails 3 consecutive times, the Pod will be removed from the Service Endpoints.

Apply the deployment:

```
kubectl apply -f nginx-probes-deployment.yaml
```

Step 2: Verify Pod Status and Events

Monitor the status of your pods. Initially, they might be in a `ContainerCreating` state, then `Running`. The readiness probe will determine when they are `Ready`.

```
kubectl get pods -l app=nginx-probes
```

To see the probe events and understand their behavior, you can describe a pod:

```
kubectl describe pod <pod-name>
```

Look for the `Events` section, where you will see entries related to `Liveness` and `Readiness` probes succeeding or failing.

Step 3: Simulate a Liveness Probe Failure (Optional)

To see a liveness probe in action, you can simulate a failure. For example, if your application writes a health file that the probe checks, you could delete that file. For Nginx, it's harder to simulate a liveness failure without stopping the process. However, if you had an `exec` probe checking for a specific file, you could delete it:

```
# Example of an exec liveness probe checking for a file
livenessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

If you were using such a probe, you could `kubectl exec` into the pod and delete `/tmp/healthy`. After `failureThreshold` attempts, Kubernetes would restart the container.

Step 4: Simulate a Readiness Probe Failure (Optional)

Similarly, you can simulate a readiness probe failure. If your application has a `/ready` endpoint that returns a 500 error when not ready, you could configure the probe to hit that endpoint. For Nginx, you could temporarily block port 80 within the container (though this is more complex to demonstrate).

Step 5: Clean Up

Delete the deployment:

```
kubectl delete -f nginx-probes-deployment.yaml
```

7.3. Illustrative Images

(Illustrative images for WSL terminal commands and Kubernetes probe events will be inserted here.)

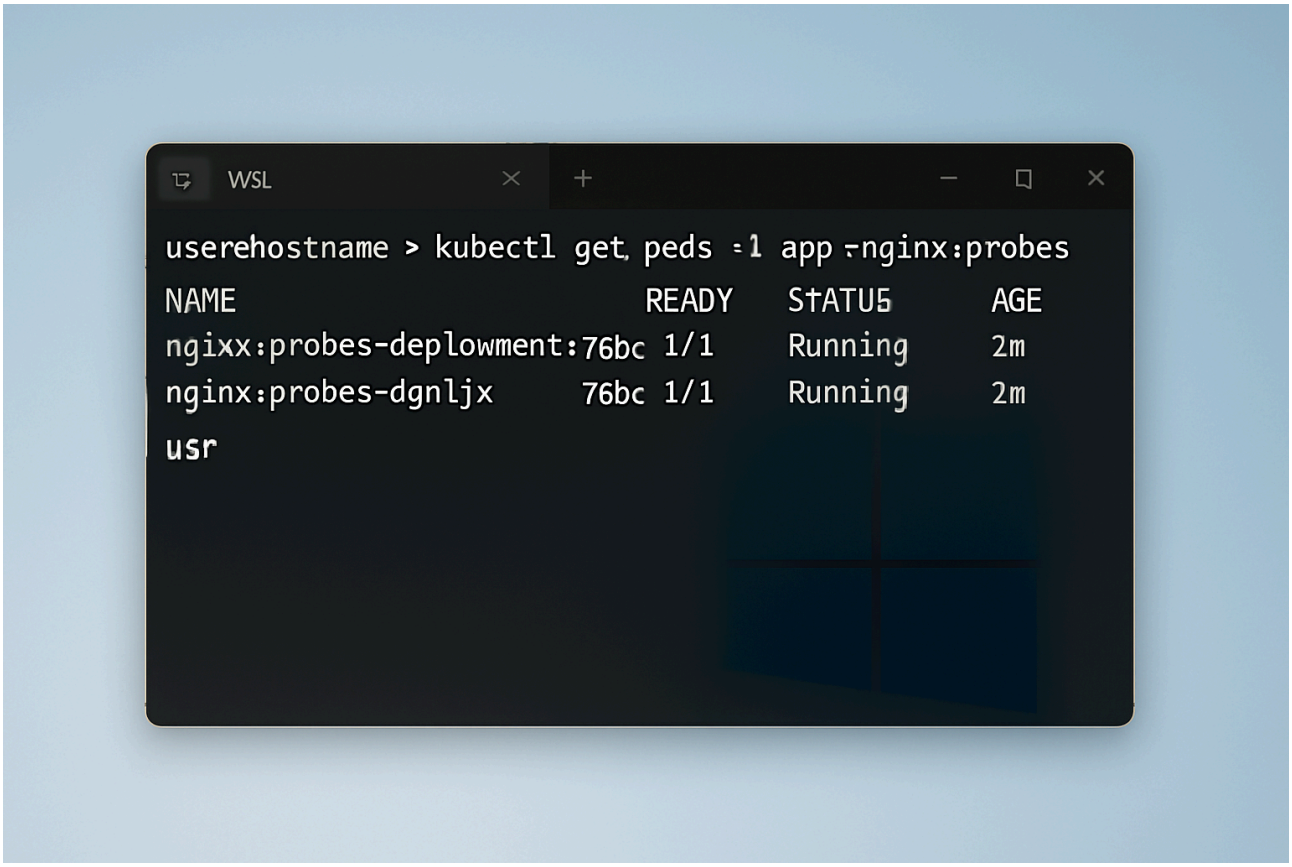

Figure 7.1: Verifying Pod status with liveness and readiness probes in a WSL terminal.

8. Configure Taints and Tolerations

Kubernetes Taints and Tolerations are powerful mechanisms that work together to ensure that Pods are scheduled onto appropriate Nodes. They allow you to control which Pods can be scheduled on which Nodes, which is particularly useful for dedicating Nodes to specific workloads or isolating problematic Nodes [31, 32].

8.1. Understanding the Concepts

Taints

Taints are applied to Nodes. They "repel" Pods, meaning that a Node with a taint will not schedule any Pods unless those Pods have a matching toleration. Taints are a key-

value pair associated with an effect. The effect specifies what happens to Pods that do not tolerate the taint [31, 34]:

- `NoSchedule`: Pods that do not tolerate this taint will not be scheduled on the Node. Existing Pods on the Node are not affected.
- `PreferNoSchedule`: The scheduler will try to avoid placing Pods that do not tolerate this taint on the Node, but it is not a strict requirement.
- `NoExecute`: Pods that do not tolerate this taint will not be scheduled on the Node, and existing Pods on the Node that do not tolerate the taint will be evicted.

Use Cases for Taints: * **Dedicated Nodes:** Dedicate a set of Nodes for specific users or workloads (e.g., GPU-enabled Nodes for machine learning tasks). * **Nodes with Special Hardware:** Mark Nodes with specialized hardware (e.g., GPUs, SSDs) so that only Pods requiring that hardware are scheduled on them. * **Eviction based on Node Conditions:** Automatically evict Pods from Nodes that are experiencing issues (e.g., network unreachable, disk pressure).

Tolerations

Tolerations are applied to Pods. A toleration allows a Pod to be scheduled on a Node that has a matching taint. It does not guarantee that the Pod will be scheduled on that Node, only that it will not be repelled by the taint. A toleration matches a taint by specifying the same key, value (optional), and effect. You can also specify an `operator` (either `Exists` or `Equal`) [31, 33].

Use Cases for Tolerations: * Allow specific Pods to run on tainted Nodes. * Enable Pods to remain on Nodes that have been tainted due to temporary issues.

8.2. How Taints and Tolerations Work Together

1. An administrator applies a taint to a Node.
2. The Kubernetes scheduler checks the taints on a Node before scheduling a Pod on it.
3. If a Pod has a toleration that matches a taint on a Node, the Pod can be scheduled on that Node.
4. If a Pod does not have a matching toleration for a taint with `NoSchedule` or `NoExecute` effect, it will not be scheduled (or will be evicted) from that Node.

8.3. Deployment Steps on WSL-based Windows System

This section will demonstrate how to apply taints to a Kubernetes node and then configure a Pod with a toleration to allow it to be scheduled on that tainted node. For AKS, you would typically apply taints to node pools.

Step 1: Get the Name of a Node

First, identify a node in your Kubernetes cluster. You can use `kubectl get nodes`:

```
kubectl get nodes
```

Choose one of the node names (e.g., `docker-desktop` if using Docker Desktop Kubernetes, or an AKS node name).

Step 2: Apply a Taint to a Node

Apply a taint to the chosen node. Replace `<node-name>` with the actual name of your node. This example uses a `NoSchedule` effect, meaning no new pods will be scheduled on this node unless they tolerate this taint.

```
kubectl taint node <node-name> dedicated=special:NoSchedule
```

Verify the taint has been applied:

```
kubectl describe node <node-name> | grep Taints
```

Step 3: Attempt to Deploy a Pod Without Toleration

Create a simple Nginx deployment without any tolerations. Save this as `nginx-no-toleration.yaml`:

```yaml
 apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-no-toleration
  labels:
    app: nginx-no-toleration
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-no-toleration
  template:
    metadata:
      labels:
        app: nginx-no-toleration
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Apply the deployment:

```
kubectl apply -f nginx-no-toleration.yaml
```

Observe the pod status:

```
kubectl get pods -l app=nginx-no-toleration
```

The pod will likely remain in a `Pending` state because it cannot be scheduled on the tainted node (assuming it's the only or primary node). You can describe the pod to see the scheduling failure:

```
kubectl describe pod <nginx-no-toleration-pod-name>
```

Look for events indicating `NoSchedule` or `Taints`.

Step 4: Deploy a Pod With Toleration

Now, create a deployment with a toleration that matches the taint applied to the node. Save this as `nginx-with-toleration.yaml`:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-with-toleration
  labels:
    app: nginx-with-toleration
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-with-toleration
  template:
    metadata:
      labels:
        app: nginx-with-toleration
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
      tolerations:
      - key: "dedicated"
        operator: "Equal"
        value: "special"
        effect: "NoSchedule"
```

Apply the deployment:

```
kubectl apply -f nginx-with-toleration.yaml
```

Verify the pod status:

```
kubectl get pods -l app=nginx-with-toleration
```

This pod should now be scheduled and run on the tainted node because it has the matching toleration.

Step 5: Clean Up

Remove the taint from the node:

```
kubectl taint node <node-name> dedicated=special:NoSchedule-
```

Delete the deployments:

```
kubectl delete -f nginx-no-toleration.yaml
kubectl delete -f nginx-with-toleration.yaml
```

8.4. Illustrative Images

(Illustrative images for WSL terminal commands and Kubernetes taint/toleration states will be inserted here.)

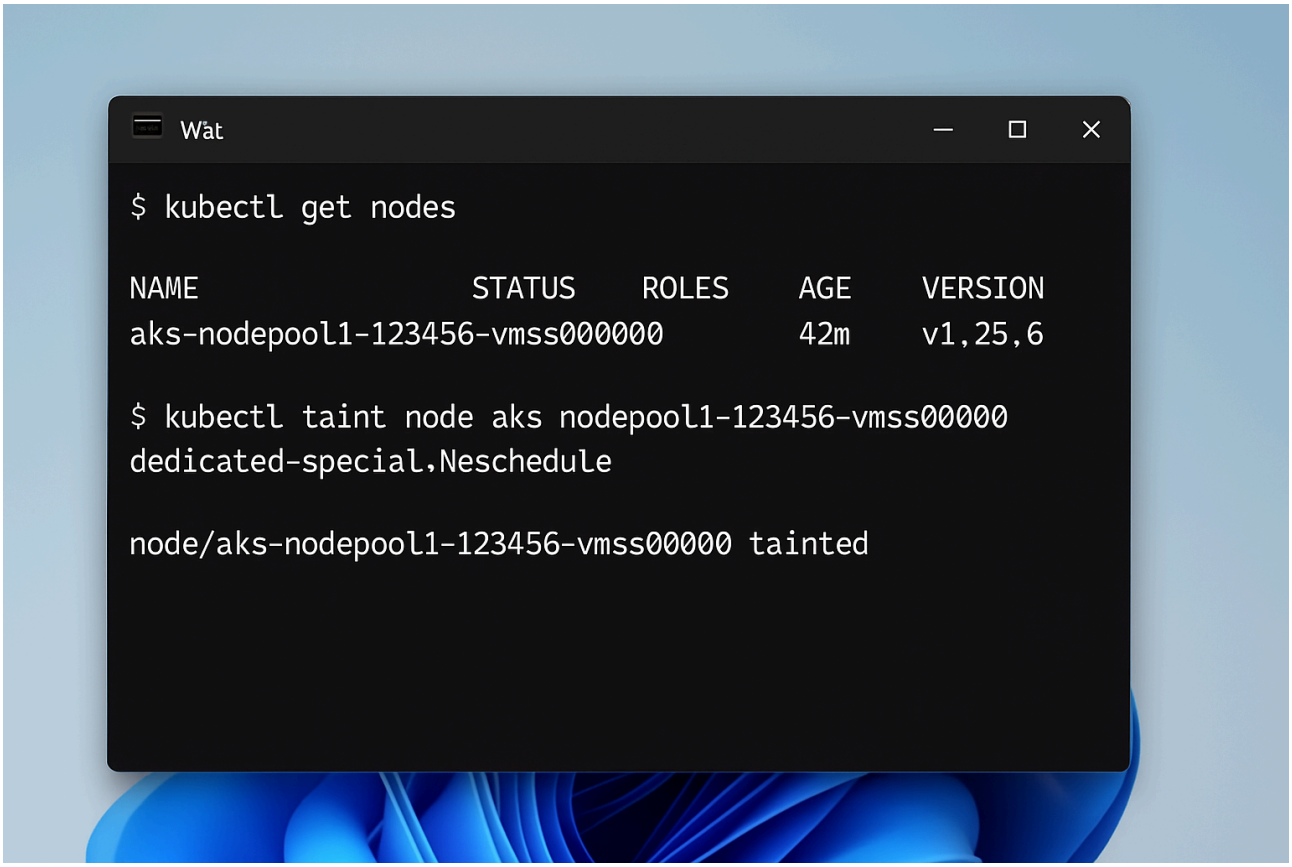

Figure 8.1: Tainting a Kubernetes node in a WSL terminal.

9. Create and Attach Persistent Volume Claims to Pods

This topic is covered in detail under section 5, "PersistentVolume (PV) and PersistentVolumeClaim (PVC)," specifically in subsection 5.3, "Deployment Steps on WSL-based Windows System." That section provides comprehensive instructions on defining PVs and PVCs, and then attaching PVCs to Pods for persistent storage.

10. Configure Autoscaling in Your Cluster (Horizontal Scaling)

Autoscaling in Kubernetes and AKS allows your applications and infrastructure to automatically adjust to changes in demand, ensuring optimal resource utilization and

performance. Horizontal scaling refers to increasing or decreasing the number of Pod replicas or Nodes. This section will detail the configuration of Horizontal Pod Autoscaler (HPA) and Cluster Autoscaler [37, 38].

10.1. Understanding the Concepts

Horizontal Pod Autoscaler (HPA)

HPA automatically scales the number of Pod replicas in a Deployment, ReplicaSet, or StatefulSet based on observed CPU utilization or other select metrics (e.g., memory usage, custom metrics). It ensures that your application has enough resources to handle the current load without over-provisioning [37, 39].

How it Works: 1. You define an HPA resource that specifies the target metric (e.g., 50% CPU utilization), the minimum number of replicas, and the maximum number of replicas. 2. The HPA controller periodically checks the metrics from the Metrics Server (or other metric sources). 3. If the observed metric value deviates from the target, the HPA calculates the desired number of replicas and updates the target workload resource (e.g., Deployment). 4. The Deployment controller then creates or deletes Pods to match the desired replica count.

Configuration: * To use HPA, your containers must have CPU requests and limits defined. * You create an HPA object using `kubectl autoscale deployment <deployment-name> --cpu-percent=<target-percentage> --min=<min-replicas> --max=<max-replicas>` or by defining a YAML manifest.

Cluster Autoscaler

The Cluster Autoscaler automatically adjusts the number of Nodes in your Kubernetes cluster. It works in conjunction with HPA to provide comprehensive scaling capabilities. While HPA scales the number of Pods, the Cluster Autoscaler ensures there are enough underlying Nodes to run those Pods [38, 40].

How it Works: 1. The Cluster Autoscaler monitors for Pods that cannot be scheduled due to insufficient resources (e.g., CPU, memory) on existing Nodes. 2. If such Pods are detected, the Cluster Autoscaler adds more Nodes to the cluster to accommodate them. 3. It also monitors for underutilized Nodes. If a Node is consistently underutilized and its Pods can be safely moved to other Nodes, the Cluster Autoscaler removes that Node from the cluster.

Integration with AKS: * In AKS, the Cluster Autoscaler integrates with Azure Virtual Machine Scale Sets to manage the underlying virtual machines that form your Node Pools. * You enable and configure the Cluster Autoscaler for your AKS cluster using the Azure CLI (`az aks update --enable-cluster-autoscaler` and specifying `--min-count` and `--max-count`).

10.2. Relationship between HPA and Cluster Autoscaler

- HPA scales the number of Pods *within* the existing Nodes.
- Cluster Autoscaler scales the number of Nodes *in the cluster* to provide enough capacity for the Pods.
- They work together to provide comprehensive autoscaling: HPA handles application-level scaling, while Cluster Autoscaler handles infrastructure-level scaling. When HPA scales up Pods, if there aren't enough resources, the Cluster Autoscaler will add more nodes. When HPA scales down Pods, if nodes become underutilized, the Cluster Autoscaler will remove them.

10.3. Deployment Steps on WSL-based Windows System

This section will demonstrate how to configure HPA and Cluster Autoscaler. For the Cluster Autoscaler, you will need an AKS cluster.

Step 1: Deploy an Application for Horizontal Pod Autoscaling (HPA)

To demonstrate HPA, deploy a sample application (e.g., an Nginx deployment) and define resource requests for its containers. Create `nginx-hpa-deployment.yaml`:

```yaml
 apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-hpa-deployment
spec:
  selector:
    matchLabels:
      app: nginx-hpa
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx-hpa
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          requests:
            cpu: 100m
```

Apply the deployment:

```
kubectl apply -f nginx-hpa-deployment.yaml
```

Step 2: Create a Horizontal Pod Autoscaler (HPA)

Create an HPA that targets the `nginx-hpa-deployment` and scales based on CPU utilization. This HPA will try to maintain an average CPU utilization of 50% across the pods, scaling between 1 and 5 replicas.

```
kubectl autoscale deployment nginx-hpa-deployment --cpu-percent=50 --min=1 --
max=5
```

Step 3: Generate Load to Test HPA

To see the HPA in action, you can generate some load on the Nginx application. First, get the name of one of the Nginx pods:

```
kubectl get pods -l app=nginx-hpa
```

Then, use `kubectl run` to create a temporary pod that sends continuous requests to the Nginx service. Replace `<nginx-pod-name>` with the actual pod name you retrieved:

```
kubectl run -it --rm load-generator --image=busybox -- /bin/sh -c "while true; do wget -q -O- http://nginx-hpa-deployment; done"
```

After a few minutes, observe the HPA. You can check its status:

```
kubectl get hpa
```

You should see the `REPLICAS` count increase as the CPU utilization rises. Once you stop the `load-generator` pod (by pressing `Ctrl+C` in its terminal), the HPA will eventually scale down the Nginx pods.

Step 4: Enable Cluster Autoscaler (for AKS)

If you have an AKS cluster, you can enable the Cluster Autoscaler on your node pool. This will automatically adjust the number of nodes in your node pools within a specified range.

```
az aks update --resource-group myAKSResourceGroup --name myAKSCluster --enable-cluster-autoscaler --min-count 1 --max-count 3
```

This command enables the cluster autoscaler on the default node pool, allowing it to scale between 1 and 3 nodes. You can verify its status using `az aks show`.

Step 5: Clean Up

Delete the HPA and the deployment:

```
kubectl delete hpa nginx-hpa-deployment
kubectl delete deployment nginx-hpa-deployment
```

If you enabled the Cluster Autoscaler on AKS, you can disable it:

```
az aks update --resource-group myAKSResourceGroup --name myAKSCluster --disable-cluster-autoscaler
```

10.4. Illustrative Images

(Illustrative images for WSL terminal commands and Kubernetes autoscaling states will be inserted here.)

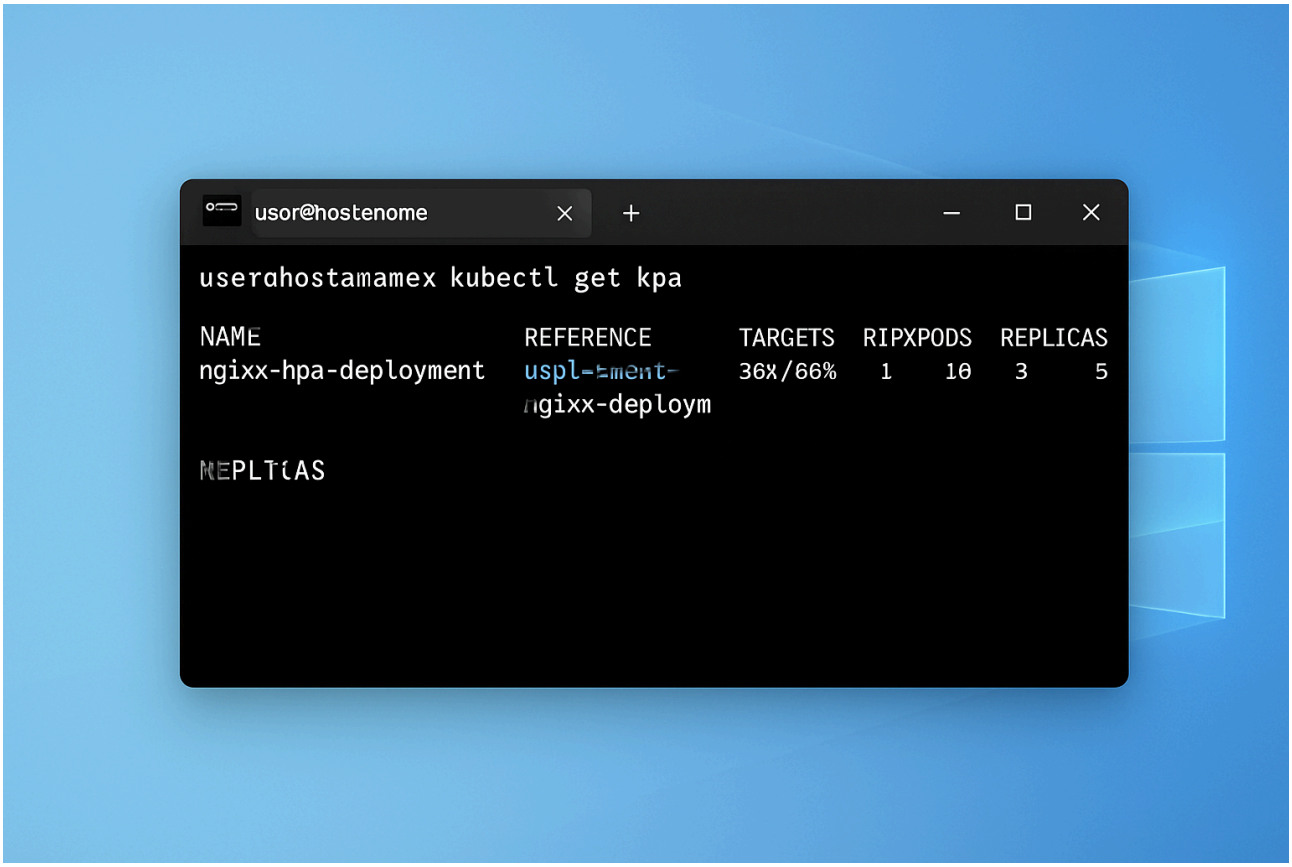

Figure 10.1: Verifying Horizontal Pod Autoscaler status in a WSL terminal.