

## Lab Exercise: Introduction to Kubernetes & Container Orchestration

**Lab Title:** Introduction to Kubernetes & Container Orchestration: Deploying, Scaling, and Industry Context

**Duration:** 2 Hours

**Target Audience:** Student Group preparing lab materials for the class.

**Overview:** This lab introduces **Kubernetes (K8s)**, the industry standard for container orchestration, by using **Minikube** for local, hands-on experience. Students will deploy and scale a microservice, understand the declarative architecture, and then dive into how Kubernetes functions in the real world compared to proprietary cloud services like AWS EC2 Auto Scaling and Azure App Service.

### Part 1: Conceptual Foundation (20 Minutes)

Students will read and discuss the following high-level concepts before beginning the hands-on steps.

#### 1. Kubernetes Architecture (High-Level)

Kubernetes operates on a **desired state** model. You *declare* what you want (e.g., "I want 3 replicas of my web app running"), and Kubernetes continuously works to make the *actual state* match the *desired state*.

- **Control Plane (Master Node):** The brain of the cluster. It manages the desired state, schedules applications, and performs auto-healing.
  - **API Server:** The only component you interact with (via kubectl). It is the central management hub.
  - **etcd:** The cluster's single source of truth; it stores the entire desired state.
- **Worker Nodes:** The machines that run your applications.
  - **Kubelet:** An agent on the Node that communicates with the API Server, ensuring the Pods defined in the desired state are running and healthy.
  - **Container Runtime:** The software (like Docker or containerd) that actually pulls and runs the container images.

- **Pods:** The smallest deployable unit in Kubernetes, typically containing one application container (your Spring Boot image).

## 2. How Autoscaling Works (HPA)

The **Horizontal Pod Autoscaler (HPA)** is a **Controller** that constantly monitors a metric (like CPU usage) of a **Deployment**.

1. **Monitoring:** The HPA checks the average CPU utilization of all running Pods against a target you define (e.g., 50%).
2. **Calculation:** If the average utilization is above the target, the HPA calculates the ideal number of replicas needed.
3. **Action:** The HPA updates the replicas count in the **Deployment** object.
4. **Self-Healing:** The **Deployment Controller** sees the updated replicas count and automatically creates the new Pods on the Worker Nodes to match the new desired state.

## 3. K8s in the Real World: Alignment vs. Difference from Industry Tools

Feature	AWS EC2 Auto Scaling (IaaS)	Azure App Service (PaaS)	Kubernetes (EKS, AKS, GKE) (Container Orchestration)
<b>Abstraction Level</b>	<b>Infrastructure (VMs).</b> Scales the underlying virtual machines.	<b>Platform (Web Apps).</b> Fully managed web apps/APIs.	<b>Container (Pods).</b> Scales the containerized workload (Pods) <i>within</i> the VMs.
<b>Scaling Unit</b>	<b>EC2 Virtual Machines (VMs).</b> Scaling takes minutes.	<b>App Service Plan Instances.</b> Scaling is relatively fast.	<b>Pods.</b> Scaling takes seconds. Extremely fast scaling of microservices.
<b>Vendor Lock-in</b>	<b>High.</b> Tied deeply to AWS APIs.	<b>High.</b> Tied deeply to Azure APIs.	<b>Low (Portable).</b> YAML files can be used almost identically across AWS (EKS), Azure (AKS), GCP (GKE), and on-premise clusters.
<b>Control/Flexibility</b>	High control over the OS/VM layer.	Low control; highly opinionated platform.	<b>High control</b> over the deployment, networking, and storage <i>abstraction</i> .
<b>Primary Use</b>	Scaling monolithic applications or simple services	Hosting simple web applications and APIs quickly with zero	Deploying complex <b>Microservices</b> architectures that require cross-cloud portability, service mesh, and

	where the VM is the boundary.	infrastructure overhead.	advanced deployment strategies (e.g., Canary).
--	-------------------------------	--------------------------	--

**Conclusion:** Kubernetes is used in the real world when a company needs **flexibility, portability, and fine-grained control** over a complex microservices application. While cloud services like App Service and EC2 Auto Scaling offer faster, simpler deployment (PaaS/IaaS), Kubernetes provides the **open standard** that avoids vendor lock-in and allows the same deployment pipeline to target any cloud or on-premise environment.

## Part 2: Hands-On Deployment (100 Minutes)

### Step 1: Local Kubernetes Environment Setup (15 mins)

1. **Tool Installation:** Install **Minikube** and **kubect**l.
2. **Cluster Start:** Start the local cluster: `minikube start`
3. **Local Docker Access:** Connect your shell to Minikube's Docker daemon (so K8s can see your local image): `eval $(minikube docker-env)`
4. **Container Image:** Build the existing Spring Boot application (from previous labs) into a Docker image: `docker build -t microservice:v1 .`
5. **Verification:** Check the node status: `kubect`l get nodes

### Step 2: Deployment and Service (25 mins)

1. **Deployment YAML:** Create a file (e.g., `deployment.yaml`) that defines the Pods and the desired replica count.<sup>15</sup>
  - a. *Self-Correction:* Use the local image name (`microservice:v1`) and set `imagePullPolicy: Never` so it doesn't try to pull from Docker Hub.
  - b. Set the initial number of replicas to replicas: 1.
2. **Service YAML:** Create a file (e.g., `service.yaml`) to expose the Deployment.
  - a. Use type: `NodePort` to expose the service outside the cluster.<sup>16</sup>
  - b. Map the container's port (e.g., 8080) to the service port.
3. **Apply Configuration:** Apply both YAML files to the cluster:
  - a. `kubect`l apply -f `deployment.yaml`
  - b. `kubect`l apply -f `service.yaml`
4. **Access Test:** Get the external URL and test the endpoint:
  - a. `minikube service <service-name> --url`

### Step 3: Declarative Management and Self-Healing (20 mins)

1. **Manual Scaling (Scale Up):** Increase the replica count from 1 to 3:
  - a. `kubectl scale deployment <deployment-name> --replicas=3`
  - b. Verify the new Pods are running: `kubectl get pods` (Observe 3 Pods)
2. **Self-Healing:** Manually kill one of the Pods:
  - a. `kubectl delete pod <one-of-the-pod-names>`
  - b. Verify Kubernetes automatically starts a new Pod to replace the deleted one and return the count to 3.

### Step 4: Horizontal Pod Autoscaler (HPA) (40 mins)

1. **Enable Metrics:** Ensure the necessary K8s component is running:
  - a. `minikube addons enable metrics-server`
2. **Create HPA:** Instruct Kubernetes to automatically scale the deployment based on CPU load.
  - a. Create the HPA object, setting the target to **50% CPU utilization**, min replicas to **1**, and max replicas to **5**.
  - b. Example command using `kubectl autoscale`:

```
kubectl autoscale deployment <deployment-name> --cpu-percent=50 --min=1 --max=5
```

3. **Monitor HPA:** Check the initial status: `kubectl get hpa` (It should show low utilization).
4. **Load Simulation:** Run a simple load test (e.g., a simple while loop with curl in a separate terminal) against the Service URL to drive up the CPU usage of the running Pod(s).
5. **Observation:** Repeatedly run `kubectl get hpa` and observe the **REPLICAS** count automatically increase from the initial number (e.g., 3) up to the maximum (5) as the CPU utilization rises above 50%.<sup>18</sup>
6. **Scale Down:** Stop the load simulation and observe the **REPLICAS** count automatically drop back down to the minimum of 1 (or the current desired state) after a short stabilization period.

### Lab Demonstration and Evaluation Guidelines

Students will demonstrate the core Kubernetes capabilities and provide the required conceptual explanation.

Component Demonstrated	Evidence Required for Lab Support	Focus
K8s Core Objects	Show the running Pods and the successful access of the service URL.	Deployment & Networking
Self-Healing	Manually delete a Pod and show that the <b>Deployment Controller</b> instantly creates a new one.	Declarative State
Horizontal Autoscaling	Show the <b>HPA creation command</b> and the <b>kubectl get hpa</b> output, demonstrating the replica count increasing automatically under simulated load.	Orchestration Power
Conceptual Understanding	Verbally explain: <b>How the sub claim is used in the security lab</b> , and how <b>Kubernetes HPA differs from AWS EC2 Auto Scaling</b> (Container vs. VM scaling).	Real-World Context

## Expected Outcomes

By the end of this lab, students should be able to:

- **Set up and interact** with a local Kubernetes cluster using Minikube and kubectl.
- **Deploy an application** using Kubernetes **Deployment** and expose it using a **Service**.
- **Demonstrate the declarative, self-healing, and scaling** principles of K8s.
- **Articulate the high-level K8s architecture** (Control Plane vs. Worker Node).
- **Compare and contrast Kubernetes** with proprietary cloud scaling solutions, focusing on **portability and the unit of scaling (Pod vs. VM)**.

## Lab: Hands-On Deployment (100 Minutes)

Goal: Deploy and scale a microservice on a local Kubernetes cluster while learning how Kubernetes maintains the desired state, self-heals and autoscale workloads.

### Part 1: Local Kubernetes Environment Setup (15 mins)

#### Prerequisites:

- Windows 10/11
- PowerShell (Run as Administrator where noted)
- WSL 2 (can be installed [here](#))
- Docker Desktop (can be installed [here](#))

Kubernetes is a container orchestrator. It manages how and where containers run to match the “desired state” you define. If you declare “3 Pods should be running” Kubernetes keeps that true at all times by restarting or recreating Pods when needed.

You will start by creating that environment locally using Minikube (which simulates a Kubernetes cluster on your computer)

## 1. Tool Installation: Install Minikube and kubectl.

- **Install kubectl:**

**Kubernetes** is managed through the API Server which you interact with using the kubectl command-line tool.

kubectl communicates with the control plane which decides where Pods run.

*# Using winget (recommended)*

```
winget install -e --id Kubernetes.kubectl
```

```
PS C:\Users\shane\OneDrive\Documents\Year4Semester1\AppliedSystemDesign\Labs\Week08\Lab1> winget install -e --id Kubernetes.kubectl
The 'msstore' source requires that you view the following agreements before using.
Terms of Transaction: https://aka.ms/microsoft-store-terms-of-transaction
The source requires the current machine's 2-letter geographic region to be sent to the backend service to function properly (ex. "US").

Do you agree to all the source agreements terms?
[Y] Yes [N] No: y
Found Kubernetes CLI [Kubernetes.kubectl] Version 1.34.1
This application is licensed to you by its owner.
Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.
Downloading https://dl.k8s.io/release/v1.34.1/bin/windows/amd64/kubectl.exe
59.2 MB / 59.2 MB
Successfully verified installer hash
Starting package install...
Path environment variable modified; restart your shell to use the new value.
Command line alias added: "kubectl"
Successfully installed
```

Exit out of current terminal and restart command prompt

*# Verify installation*

```
kubectl version --client
```

```
PS C:\Users\shane\OneDrive\Documents\Year4Semester1\AppliedSystemDesign\Labs\Week08\Lab1> kubectl version --client
Client Version: v1.34.1
Kustomize Version: v5.7.1
```

- **Install Minikube:**

Install Minikube to create a single-node cluster that acts as both the control plane and worker node.

*# Using winget (recommended)*

```
winget install -e --id Kubernetes.minikube
```

```
PS C:\Users\shane\OneDrive\Documents\Year4Semester1\AppliedSystemDesign\Labs\Week08\Lab1> winget install -e --id Kubernetes.minikube
Found Kubernetes - Minikube - A Local Kubernetes Development Environment [Kubernetes.minikube] Version 1.37.0
This application is licensed to you by its owner.
Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.
Successfully verified installer hash
Starting package install...
Successfully installed
```

Exit out of current terminal and restart command prompt

*# Verify installation*

```
minikube version
```

```
PS C:\Users\shane\OneDrive\Documents\Year4Semester1\AppliedSystemDesign\Labs\Week08\Lab1> minikube version
minikube version: v1.37.0
commit: 65318f4cfff9c12cc87ec9eb8f4cdd57b25047f3
```

- Minikube and kubectl can alternatively be installed from the following links:

- <https://github.com/kubernetes/minikube/releases> (github repo)
- <https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/>

## 2. Cluster Start: Start the local cluster: minikube start

- The control plane and the worker node will be launched using Docker (ensure Docker Desktop is running):

**Note:** Ensure the *docker* command is available in your PowerShell PATH. Use command:

`minikube start --driver=docker`

```
PS C:\Users\shane\OneDrive\Documents\Year4Semester1\AppliedSystemDesign\Labs\Week08\Lab1> docker --version
Docker version 28.4.0, build d8eb465
PS C:\Users\shane\OneDrive\Documents\Year4Semester1\AppliedSystemDesign\Labs\Week08\Lab1> minikube start --driver=docker
🐳 minikube v1.37.0 on Microsoft Windows 11 Home 10.0.26200.6901 Build 26200.6901
🔧 Using the docker driver based on user configuration
🔧 Using Docker Desktop driver with root privileges
👉 Starting "minikube" primary control-plane node in "minikube" cluster
📥 Pulling base image v0.0.48 ...
📦 Downloading Kubernetes v1.34.0 preload ...
> preloaded-images-k8s-v18-v1...: 337.07 MiB / 337.07 MiB 100.00% 4.80 Mi
> gcr.io/k8s-minikube/kicbase...: 488.52 MiB / 488.52 MiB 100.00% 5.09 Mi
🔥 Creating docker container (CPUs=2, Memory=3900MB) ...
❗ Failing to connect to https://registry.k8s.io/ from inside the minikube container
To pull new external images, you may need to configure a proxy: https://minikube.sigs.k8s.io/docs/reference/networking/proxy/
📦 Preparing Kubernetes v1.34.0 on Docker 28.4.0 ...
🔗 Configuring bridge CNI (Container Networking Interface) ...
🔧 Verifying Kubernetes components...
  ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟 Enabled addons: storage-provisioner, default-storageclass
🏁 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

`kubectl get nodes`

If the node shows a Ready status your cluster is running.

That single node will schedule and run your application Pods.

**Concept:** A Pod is the smallest deployable unit in Kubernetes. It usually runs one container but can run several that share the same storage and network resources. Kubernetes never runs containers directly. It always manages them through Pods.

- ## 3. Local Docker Access: Connect your shell to Minikube's Docker daemon (so K8s can see your local image): `eval $(minikube docker-env)` \*note : eval is not a recognised term in windows

If using the Docker driver and want to build images available inside Minikube:



minikube -p minikube docker-env | [Invoke-Expression](#)

*Note: This sets DOCKER\_ environment variables in the current PowerShell session\**

#### 4. Add Controller and Build Existing Spring Boot Application (from previous labs)

Before building the Docker image, make sure your Spring Boot application includes a controller to handle HTTP requests.

*# Navigate to your application directory*

```
cd C:\path\to\your\spring\boot\app\src\main\java\com\example\demo
```

*# Create a new folder called controller and navigate into it*

```
mkdir controller
```

```
cd controller
```

*# Create a new java file called ComputeController*

```
echo ComputeController
```

```
import org.springframework.web.bind.annotation.*;
import java.util.*;

@RestController
public class ComputeController {

    @GetMapping("/")
    public Map<String, Object> health() {
        Map<String, Object> res = new HashMap<>();
        res.put("status", "ok");
        res.put("message", "Microservice is running!");
        res.put("endpoints", new String[]{"/", "/compute"});
        return res;
    }

    @GetMapping("/compute")
    public Map<String, Object> compute() {
        long result = 0;
        for (int i = 0; i < 100_000_000; i++) {
            result += i % 7;
        }
        Map<String, Object> res = new HashMap<>();
        res.put("status", "ok");
        res.put("message", "Computation complete");
    }
}
```

```
    res.put("result", result);  
    return res;  
}  
}
```

// Ensure to keep the package name at the top of file this already be automatically generated e.g. package com.example.demo.controller;

Before building your project check and update dependencies in pom.xml

*# Navigate back to your application directory*

`cd C:\path\to\your\spring\boot\app`

*# Open pom.xml and check the dependencies*

If spring-boot-starter is present, remove it

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter</artifactId>  
</dependency>
```

Then add spring-boot-starter-web (if not already present)

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-web</artifactId>  
</dependency>
```

The following is what the dependencies in the pom.xml should look like:

```

</properties>
▼<dependencies>
  ▼<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
  </dependency>
  ▼<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-docker-compose</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
  </dependency>
  ▼<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
  ▼<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-testcontainers</artifactId>
    <scope>test</scope>
  </dependency>
  ▼<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  ▼<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
▼<build>

```

Now you are ready to build your project

*# Navigate back to your application directory*

`mvn clean package`

1. **Container Image:** Build the existing Spring Boot application into a Docker image:

*# Navigate to your application directory*

`cd C:\path\to\your\spring\boot\app`

*# Build the Docker image*

```
docker build -t microservice:v1 .
```

*# If using Minikube's Docker daemon, this image will be directly available to the cluster*

### Sample Dockerfile (if you need one):

```
FROM eclipse-temurin:17-jre
```

```
WORKDIR /app
```

```
COPY target/microservice-0.0.1-SNAPSHOT.jar app.jar
```

```
EXPOSE 8080
```

```
ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

## 2. **Verification:** Check the node status: `kubectl get nodes`

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

```
PS C:\Users\shane\OneDrive\Documents\Year4Semester1\AppliedSystemDesign\Labs\Week08\Lab1> kubectl get nodes -o wide
NAME        STATUS    ROLES    AGE   VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE                                     KERNEL-VERSION   CONTAINER-RUNTIME
minikube    Ready     control-plane  14m   v1.34.0   192.168.49.2   <none>        Ubuntu 22.04.5 LTS   6.6.87.2-microsoft-standard-WSL2   docker://28.4.0
```

### Verification Checks:

- ☐ What driver did you use to start Minikube?
- ☐ Does `kubectl get nodes` show Ready?
- ☐ If using Docker driver, does `docker images` include `microservice:v1`?

### Troubleshooting (Windows):

- **Symptom:** `minikube start --driver=docker` hangs.  
**Fix:** Ensure Docker Desktop is running, and the WSL 2 based engine is enabled.
- **Symptom:** `kubectl` cannot connect (context not set).  
**Fix:** Run

```
kubectl config get-contexts
```

```
kubectl config use-context minikube
```

- **Symptom:** mvn clean package fails  
**Fix:** Verify Maven is installed and JAVA\_HOME points to correct JDK
- **Symptom:** docker build fails with permission errors  
**Fix:** Ensure PowerShell is running as Administrator and Docker Desktop is active
- **Symptom:** Image not visible in Minikube  
**Fix:** Run

```
minikube -p minikube docker-env | Invoke-Expression
and rebuild the image
```

## Step 2: Deployment and Service (25 mins)

In Kubernetes Deployments describe the desired state. They define how many replicas you want, which image to use and how much CPU or memory to allocate.

A Service provides stable network access to these Pods even when individual Pods are replaced or rescheduled.

1. *Deployment YAML: Create a file (deployment.yaml) that defines the Pods and the desired replica count*

*# Navigate to your application directory*

```
cd C:\path\to\your\spring\boot\app
```

*# Create a deployment.yaml file in current directory*

```
echo deployment.yaml
```

Paste the following into the deployment.yaml file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: microservice-deployment
  labels:
    app: microservice
spec:
  # Initial number of replicas
  replicas: 1
```

```

# Selector identifies Pods managed by this Deployment
selector:
  matchLabels:
    app: microservice

# Template for creating Pods
template:
  metadata:
    labels:
      app: microservice
  spec:
    containers:
      - name: microservice
        image: microservice:v1
        imagePullPolicy: Never # Don't pull from Docker Hub
        ports:
          - containerPort: 8080
        resources:
          requests:
            memory: "128Mi"
            cpu: "100m"
          limits:
            memory: "256Mi"
            cpu: "500m"

```

**Key Concept:** The Deployment Controller maintains the desired number of Pods. If a Pod crashes or is deleted the controller creates a new one. This behavior demonstrates the self-healing capability of Kubernetes.

## 2. Service YAML: Create a file (e.g., service.yaml) to expose the Deployment.

A Service exposes your Pods to external traffic by assigning a stable IP and port

# Navigate to your application directory

```
cd C:\path\to\your\spring\boot\app
```

# Create a service.yaml file in current directory

echo

```
echo service.yaml
```

Paste the following into the service.yaml file:

```

apiVersion: v1
kind: Service
metadata:
  name: microservice-service

```

```

labels:
  app: microservice
spec:
  type: NodePort

  # Selector matches Pods with this label
  selector:
    app: microservice

  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
    nodePort: 30080
    name: http

```

### 3. Apply Configuration: Apply both YAML files to the cluster:

- Apply deployment.yaml to cluster

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

Output: deployment.apps/microservice-deployment created

- Apply service.yaml to cluster

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

Output: service/microservice-service created

### 4. Access Test: Get the external URL and test the endpoint:

```
minikube service microservice-service --url
```

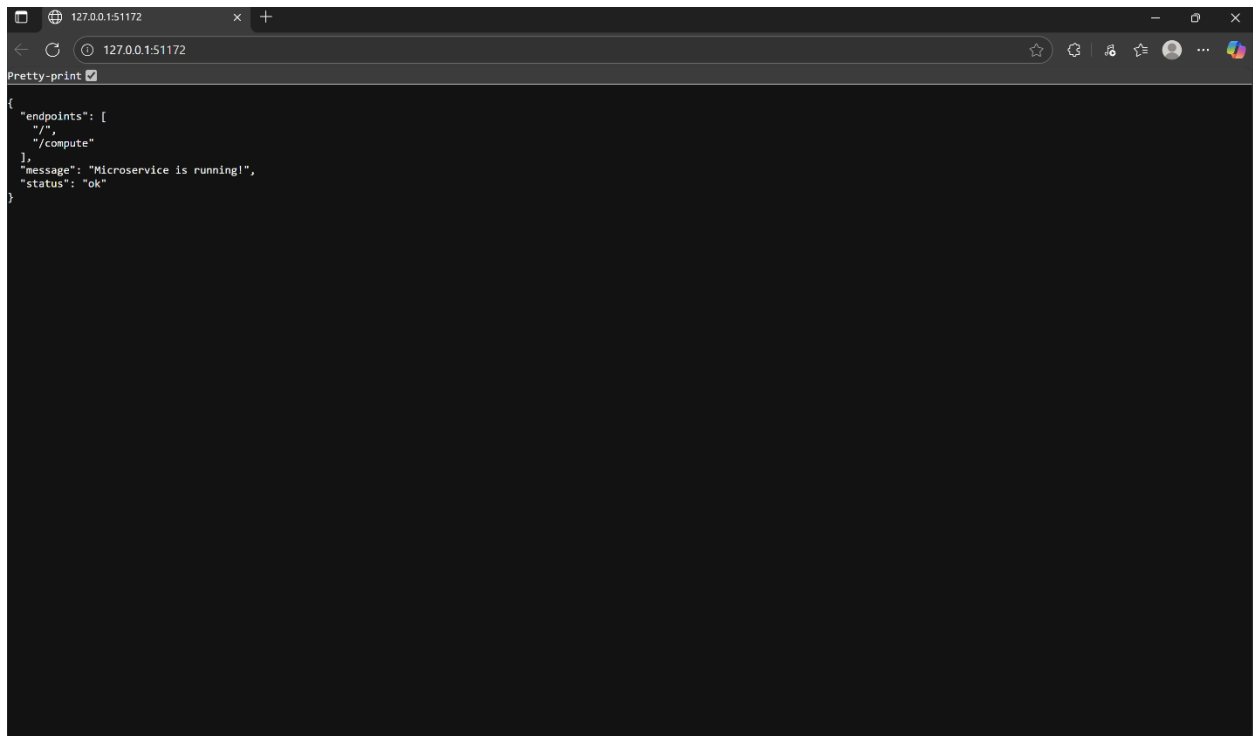
Output:

```

PS C:\Users\sohai\microservice> minikube service microservice-service --url
http://127.0.0.1:51172
! Because you are using a Docker driver on windows, the terminal needs to be open to run it.

```

When you open the URL in your browser should see a screen similar to the following:

A screenshot of a web browser window with the address bar showing '127.0.0.1:51172'. The page content displays a JSON object: 

```
{  "endpoints": [    "/",    "/compute"  ],  "message": "Microservice is running!",  "status": "ok"}
```

 The browser's developer tools are open, showing the 'Pretty-print' view of the JSON response.

**Key Concept:** The Service abstracts the network location of Pods. When Pods are recreated or scaled Kubernetes automatically updates the Service to point to the new ones.

#### Troubleshooting:

- **Symptom:** ImagePullBackOff  
**Fix:** Verify the image name and confirm imagePullPolicy: Never is set
- **Symptom:** Service URL not opening  
**Fix:** Verify Maven is installed and JAVA\_HOME points to correct JDK
- **Symptom:** mvn clean package fails  
**Fix:** Confirm port mapping in service.yaml and verify NodePort 30080 is not blocked by the firewall

## Step 3: Declarative Management and Self-Healing (20 mins)

### 1. Manual Scaling (Scale Up)

Scaling increases the number of Pods to handle higher load

# Increase the replica count from 1 to 3:



```
kubectl scale deployment microservice-deployment --replicas=3
```

*# Watch the Pods being created:*

```
kubectl get pods -w
```

*To stop watching hit ctrl + C*

**Expected output:**

NAME	READY	STATUS	RESTARTS	AGE
pod/microservice-deployment-7d4b8c9f6-xyz	1/1	Running	0	2m
pod/microservice-deployment-7d4b8c9f6-abc	1/1	Running	0	30s
pod/microservice-deployment-7d4b8c9f6-def	1/1	Running	0	15s

*# Verify the deployments:*

```
kubectl get deployment microservice-deployment
```

**Expected output:**

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
microservice-deployment	3/3	3	3	5m

## *2. Self-Healing*

**Delete one Pod manually:**

*# Get a Pod name*

```
kubectl get pods
```

*# Delete one Pod (replace with actual Pod name)*

```
kubectl delete pod microservice-deployment-7d4b8c9f6-xyz
```

*# Observe automatic Pod recreation:*

```
kubectl get pods -w
```

**Expected behavior:** - The deleted Pod will show Terminating status - A new Pod will be created automatically - The total Pod count will return to 3

**Key Concept:** *Kubernetes uses a declarative model. You declare what the cluster should look like and the system works to make it happen. You did not start containers manually. You only declared that three Pods should exist and Kubernetes made it true.*

*# Verify self-healing:*

```
kubectl get pods
```

#### Verification Checks:

- ☐ After scaling, does `kubectl get pods` show 3 Pods?
- ☐ After deleting a Pod, does it return to 3 Pods automatically?
- ☐ Can you explain why the Pod count returns to 3?

#### Troubleshooting:

- **Symptom:** New Pods remain Pending.

**Fix:** Check node capacity:

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

Look for events indicating resource constraints.

- **Symptom:** Frequent restarts (RESTARTS increasing).

**Fix:** Check logs and probe configurations:

```
kubectl logs <pod-name>
```

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

## Step 4: Horizontal Pod Autoscaler (HPA) (40 mins)

Manual scaling works but automatic scaling is more efficient.

The Horizontal Pod Autoscaler (HPA) adjusts the number of Pods in a Deployment based on observed CPU usage or other metrics.

**Key Concept:** The HPA is another controller in the Kubernetes control plane. It monitors metrics and changes the desired replica count for a Deployment to keep usage near a target level.

### 1. Enable Metrics

# *Enable the metrics-server addon:*

```
minikube get addons enable metrics-server
```

Output:

```
PS C:\Users\User\Desktop\CS4297> minikube addons enable metrics-server
* metrics-server is an addon maintained by Kubernetes. For any concerns contact minikube on GitHub.
You can view the list of minikube maintainers at: https://github.com/kubernetes/minikube/blob/master/OWNERS
- Using image registry.k8s.io/metrics-server/metrics-server:v0.8.0
* The 'metrics-server' addon is enabled
```

# *Verify metrics-server is running:*

```
kubectl rollout status deployment metrics-server -n kube-system
```

Output: deployment "metrics-server" successfully rolled out

*# Check metrics-server Pod:*

```
kubectl get pods -n kube-system | grep metrics-server
```

**Expected output:**

```
PS C:\Users\User\Desktop\CS4297> kubectl get pods -n kube-system | Select-String metrics-server
metrics-server-85b7d694d7-f9j5g    1/1    Running    0    16m
```

## 2. Create HPA

**Using kubectl autoscale command (recommended):**

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

**Method 2: Using HPA YAML (alternative):**

```
New-Item -Path "hpa.yaml" -ItemType File
```

**Content of hpa.yaml:**

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: microservice-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: microservice-deployment
  minReplicas: 1
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

**Apply HPA YAML:**

```
kubectl apply -f .\hpa.yaml
```

### Verify HPA creation:

```
kubectl get hpa
```

### Expected output:

NAME	REFERENCE	TARGETS	MINPODS	MAXPOD
microservice-hpa	Deployment/microservice-deployment	0%/50%	1	
5	30s			

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
microservice-deployment	Deployment/microservice-deployment	cpu: 7%/50%	1	5	3	113s

**Key Concept:** The HPA observes CPU usage, compares it to the target value and adjusts the Deployment's replica count to maintain performance while saving resources.

### 3. Monitor HPA

#### Watch HPA status:

```
kubectl get hpa -w
```

Output:

```
PS C:\Users\User\Desktop\CS4297> kubectl get hpa -w
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
microservice-deployment	Deployment/microservice-deployment	cpu: 6%/50%	1	5	3	5m28s

#### Get detailed HPA information:

```
kubectl describe hpa microservice-deployment
```

Output:

```
PS C:\Users\User\Desktop\CS4297> kubectl describe hpa microservice-deployment
```

Name: microservice-deployment  
Namespace: default  
Labels: <none>  
Annotations: <none>  
CreationTimestamp: Thu, 30 Oct 2025 01:00:33 +0000  
Reference: Deployment/microservice-deployment  
Metrics:  
  resource cpu on pods (as a percentage of request): 6% (6m) / 50%  
Min replicas: 1  
Max replicas: 5  
Deployment pods: 3 current / 3 desired  
Conditions:

Type	Status	Reason	Message
AbleToScale	True	ReadyForNewScale	recommended size matches current size
ScalingActive	True	ValidMetricFound	the HPA was able to successfully calculate a replica count from cpu resource utilization (percentage of request)
ScalingLimited	False	DesiredWithinRange	the desired count is within the acceptable range

Events: <none>

**Key Concept:** Kubernetes scaling is fast because it adjusts Pods inside existing nodes instead of starting new virtual machines as cloud autoscaling services do.

#### 4. Load Simulation

##### Method 1: PowerShell loop (Windows-specific):

```
# Get the service URL
$svcUrl = minikube service microservice-svc --url

# Generate Load
for ($i=0; $i -lt 100000; $i++) {
    try {
        Invoke-WebRequest -UseBasicParsing -Uri $svcUrl | Out-Null
    } catch {}
    Start-Sleep -Milliseconds 50
}
```

##### Method 2: Using kubectl run (cross-platform):

```
kubectl run loader --image=busybox --restart=Never -- /bin/sh -c "while true;
do wget -q -O- http://microservice-svc:8080/ > /dev/null; done"
```

##### Output:

```
PS C:\Users\User\Desktop\CS4297> kubectl run loader --image=busybox --restart=Never -- /bin/sh -c "while true; do wget -q -O- http://microservice-svc:8080/ > /dev/null; done"
pod/loader created
```

##### Method 3: Using Apache Bench (if available):

```
# Install Apache Bench (if not available)
# Download from: https://httpd.apache.org/docs/2.4/programs/ab.html

# Run Load test
ab -n 10000 -c 10 http://$(minikube ip):30080/
```

#### 5. Observation

##### Monitor HPA scaling up:

```
kubectl get hpa -w
```

**Expected behavior:** - CPU utilization will increase above 50% - HPA will scale up replicas from 3 to 5 - You'll see the REPLICAS count increase.

##### Output:

```
PS C:\Users\User\Desktop\CS4297> kubectl get hpa -w
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
microservice-deployment	Deployment/microservice-deployment	cpu: 117%/50%	1	5	5	18m
microservice-deployment	Deployment/microservice-deployment	cpu: 56%/50%	1	5	5	18m
microservice-deployment	Deployment/microservice-deployment	cpu: 92%/50%	1	5	5	19m
microservice-deployment	Deployment/microservice-deployment	cpu: 86%/50%	1	5	5	20m

## Monitor Deployment scaling:

```
kubectl get deployment microservice-deployment -w
```

Output:

### Expected output during scaling:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
microservice-deployment	3/5	5	3	10m
microservice-deployment	4/5	5	4	10m
microservice-deployment	5/5	5	5	10m

## 6. Scale Down

**Stop the load simulation:** - Press Ctrl+C in the PowerShell loop terminal - Or delete the loader Pod:

```
kubectl delete pod loader
```

**Expected Output:** pod "loader" deleted from default namespace

### Observe scale down:

```
kubectl get hpa -w
```

**Expected behavior:** - CPU utilization will drop below 50% - After stabilization period (default 5 minutes), HPA will scale down - Replicas will return to minimum (1) or current desired state

### Expected Output:

```
PS C:\Users\User\Desktop\CS4297> kubectl get hpa -w
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
microservice-deployment	Deployment/microservice-deployment	cpu: 42%/50%	1	5	5	31m
microservice-deployment	Deployment/microservice-deployment	cpu: 17%/50%	1	5	5	31m
microservice-deployment	Deployment/microservice-deployment	cpu: 6%/50%	1	5	5	32m

### Verification Checks:

- ☐ Does `kubectl get hpa` show metrics and desired replicas changing?
- ☐ Do Deployment replicas increase up to max (5) when load is applied?
- ☐ Do replicas scale down when load stops?
- ☐ Can you explain the HPA feedback loop?

### Troubleshooting (Windows and HPA):

- **Symptom:** `kubectl get hpa` shows unknown or no metrics.

**Fix:** Ensure metrics-server is running and healthy:

```
kubectl get deployment metrics-server -n kube-system
kubectl logs -n kube-system deploy/metrics-server
```

- **Symptom:** No scaling despite load.

**Fix:** Confirm CPU requests are defined in Deployment:

```
resources:  
  requests:  
    cpu: "100m" # This is required for HPA
```

- **Symptom:** HPA v2 API issues.

**Fix:** Use autoscaling/v2 with correct fields; Minikube versions may require specific HPA API versions.

---