FINAL USE CASE

DEPLOYING A STATIC “HELLO WORLD” WEB APPLICATION USING AZURE FREE TIER

Introduction:

This project offers a comprehensive, hands-on exploration of the modern DevOps ecosystem, with a strong emphasis on cloud infrastructure, containerization, and automation techniques. It’s designed to guide learners through real-world practices, allowing them to gain a deeper understanding of how DevOps tools and workflows integrate to streamline application deployment and maintenance.

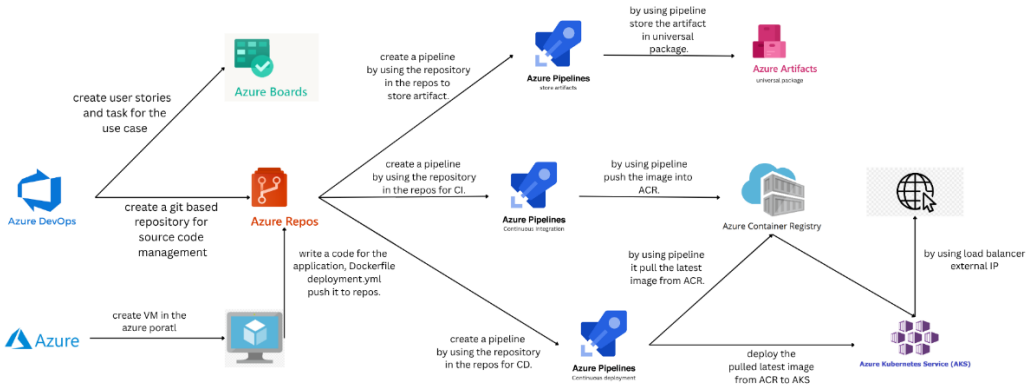
As part of this journey, you'll undertake a practical task: deploying a simple "Hello World" web application using basic HTML and CSS. This app will be hosted on Microsoft Azure's free tier services, giving you firsthand experience with cloud deployment while keeping costs at zero. The exercise not only builds foundational cloud skills but also lays the groundwork for more advanced DevOps applications.

Objective:

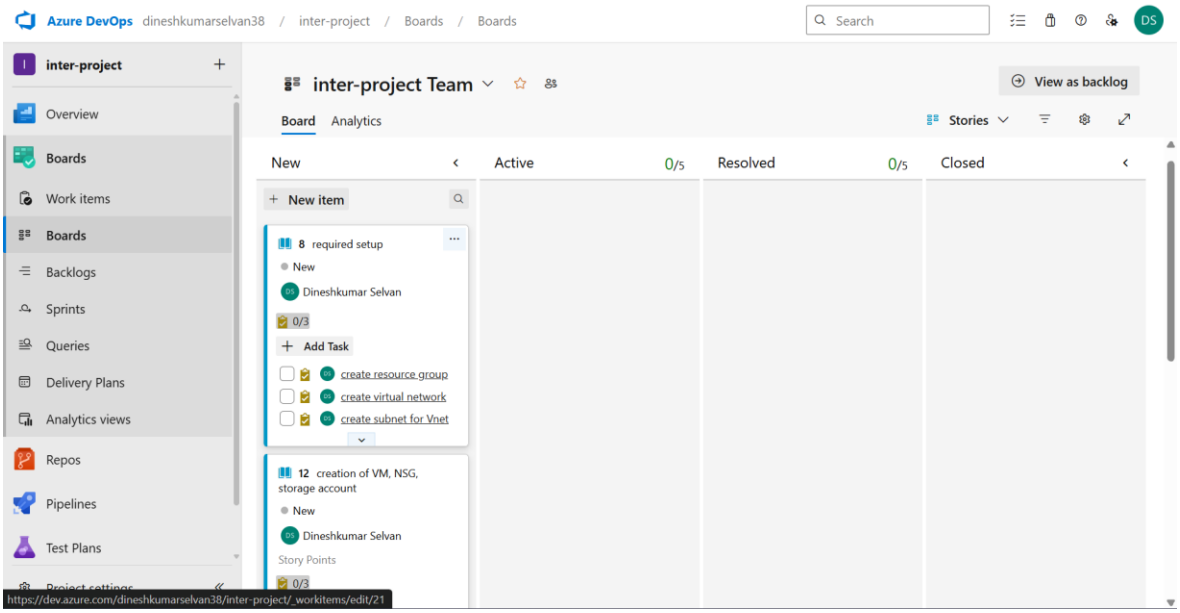
This project aims to equip you with *practical, real-world experience* in establishing a complete DevOps workflow from start to finish. Through each stage of this immersive exercise, you'll gain firsthand knowledge of deploying applications in a cloud environment using containers and orchestration tools—essential skills for today’s DevOps engineer.

By the project's conclusion, you'll grasp the core principles of continuous integration and continuous delivery (CI/CD), while learning how to manage cloud resources within Azure’s free-tier constraints to maintain a budget-friendly learning experience. This experiential journey is crafted to deepen your expertise in cloud-native development and contemporary deployment methodologies.

Architecture diagram:



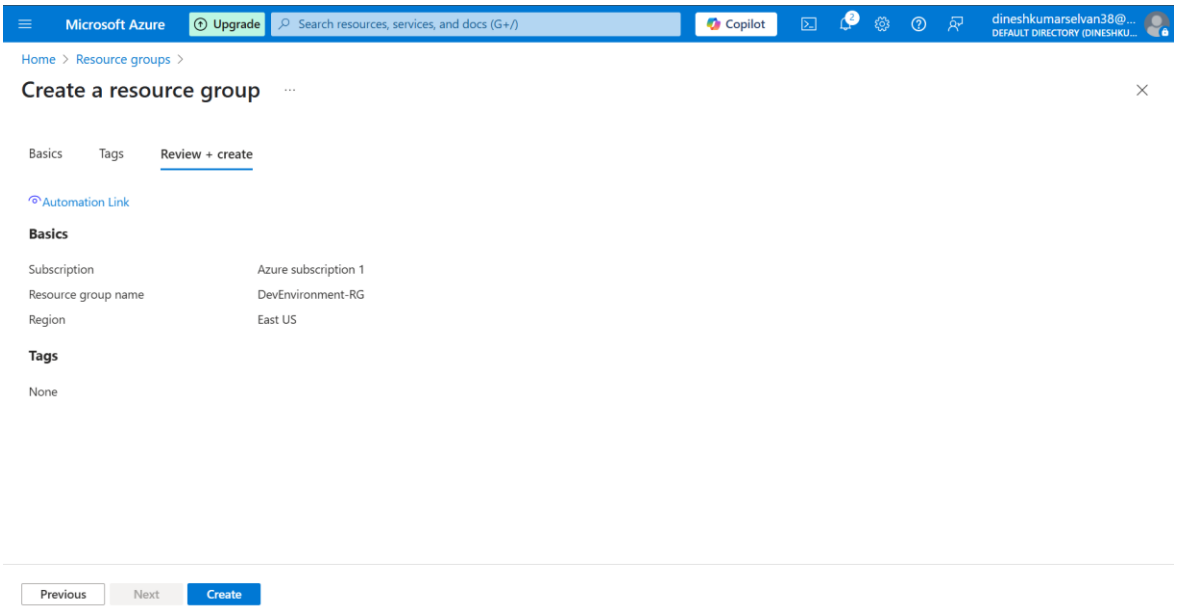
Create azure boards:



Using Azure Boards, create user stories to track the workflow of the use case.

Required Setup and Installations for CI:

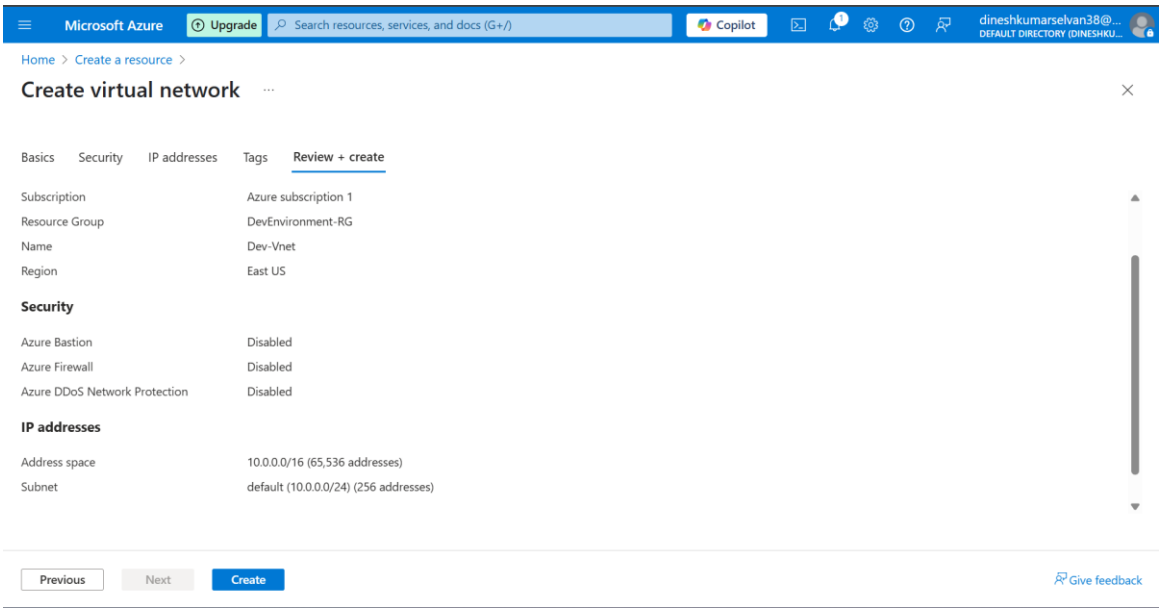
1. Azure Resource Group:



In the Azure Portal, search for **"Resource groups"** and click **+ Create**. Select your desired **subscription** and enter **DevEnvironment-RG** as the name. Choose **East US** as the region to host the group. Click **Review + Create**, then click **Create.**

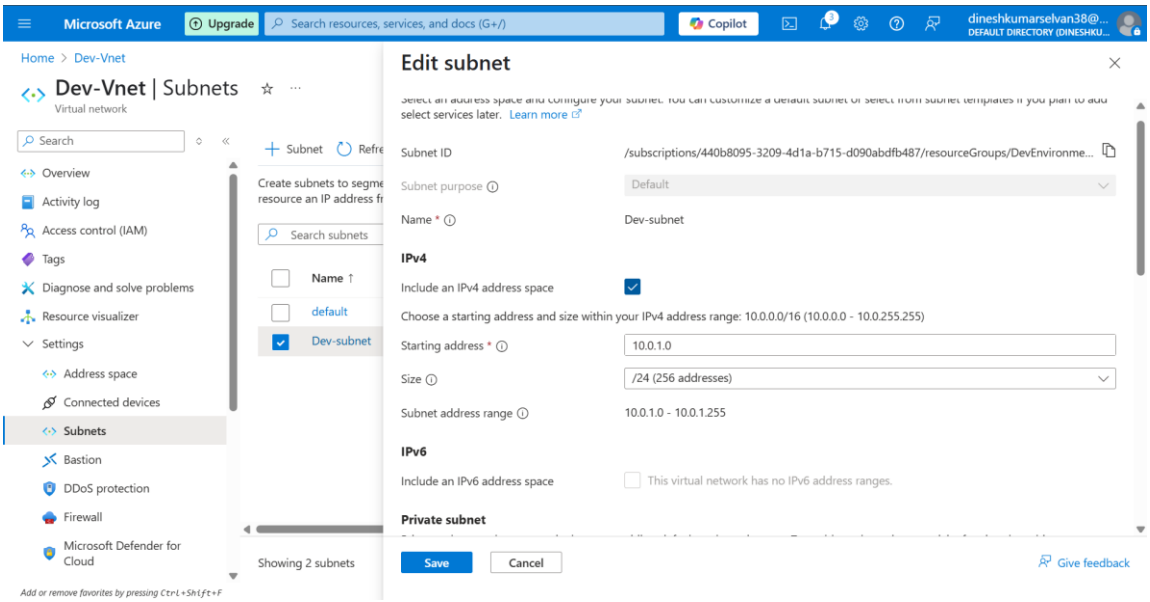
"All resources for this use case are created under the resource group 'DevEnvironment-RG' using the Azure UI."

2. Virtual Network (VNet):



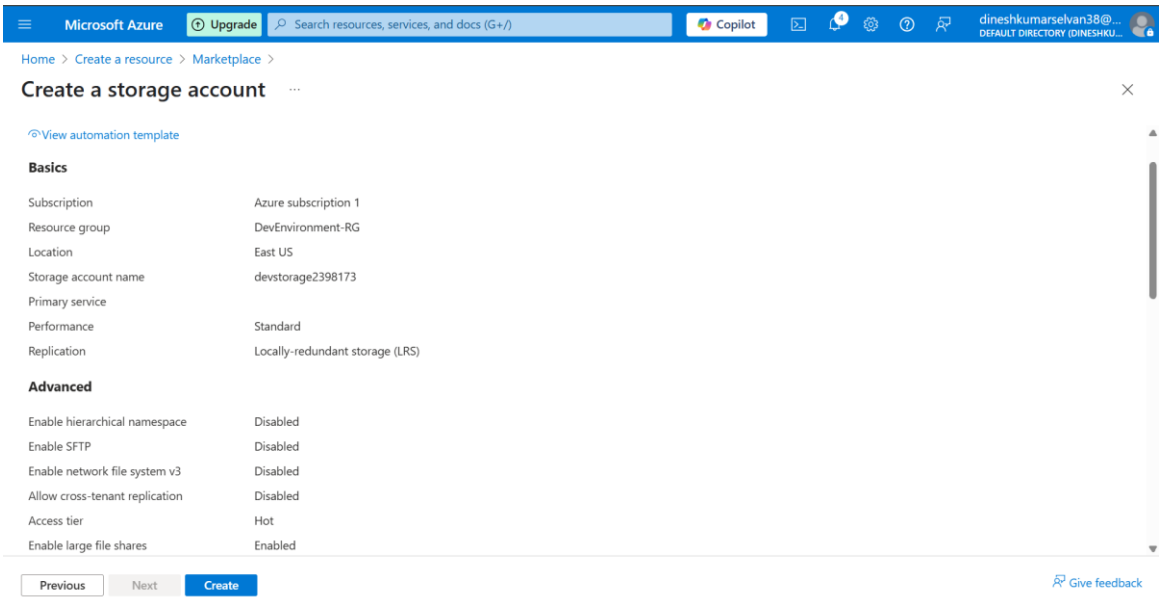
To create a virtual network named **Dev-Vnet**, log in to the Azure portal. Navigate to "Virtual Networks" and click **Create**. Under the "Basics" tab, set the name to *Dev-Vnet*, choose a resource group and region. In the "IP Addresses" tab, enter 10.0.0.0/16 as the address space. Review your settings, then click **Create** to deploy the network.

3. Subnet:



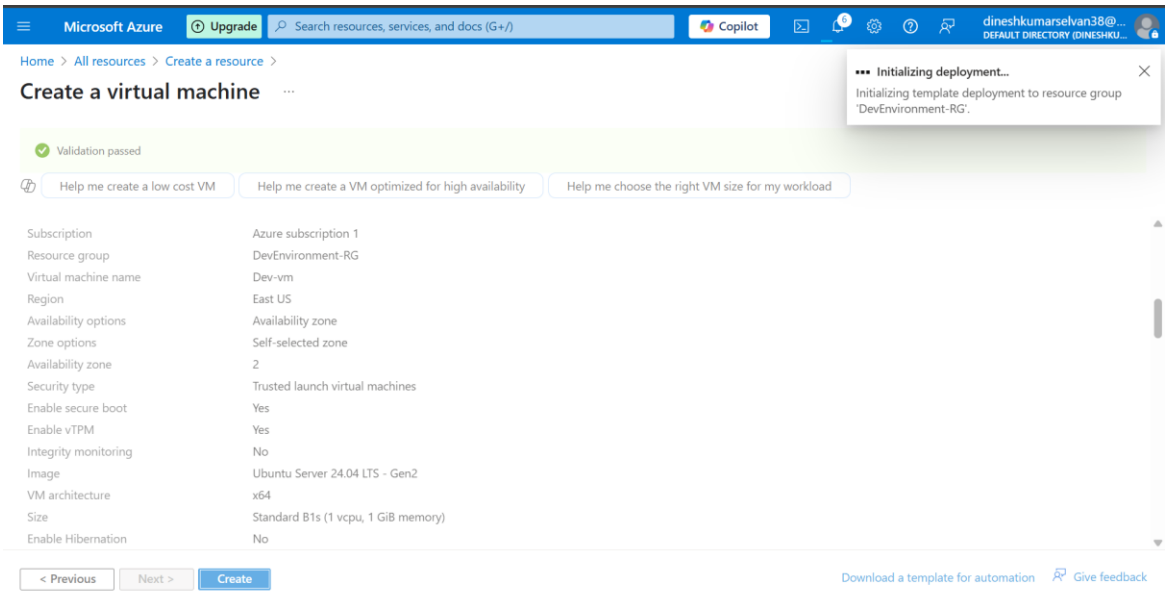
In the Azure Portal, go to **Virtual Networks** and select **Dev-Vnet**. Click on the **Subnets** tab in the left menu, then choose **+ Subnet**. Enter **Dev-subnet** as the name and 10.0.1.0/24 as the address range. Click **Save** to add the subnet to your virtual network.

4. Storage Account:



In the Azure Portal, search for **"Storage accounts"** and click **+ Create**. Choose your subscription and resource group, then enter **devstorage2398173** as the name. Select **Standard** for performance and **Locally-redundant storage (LRS)** for redundancy. Review your settings and click **Create** to deploy the storage account.

5. Virtual Machine:



Creating a virtual machine named **“Dev-vm”** using the Azure portal,

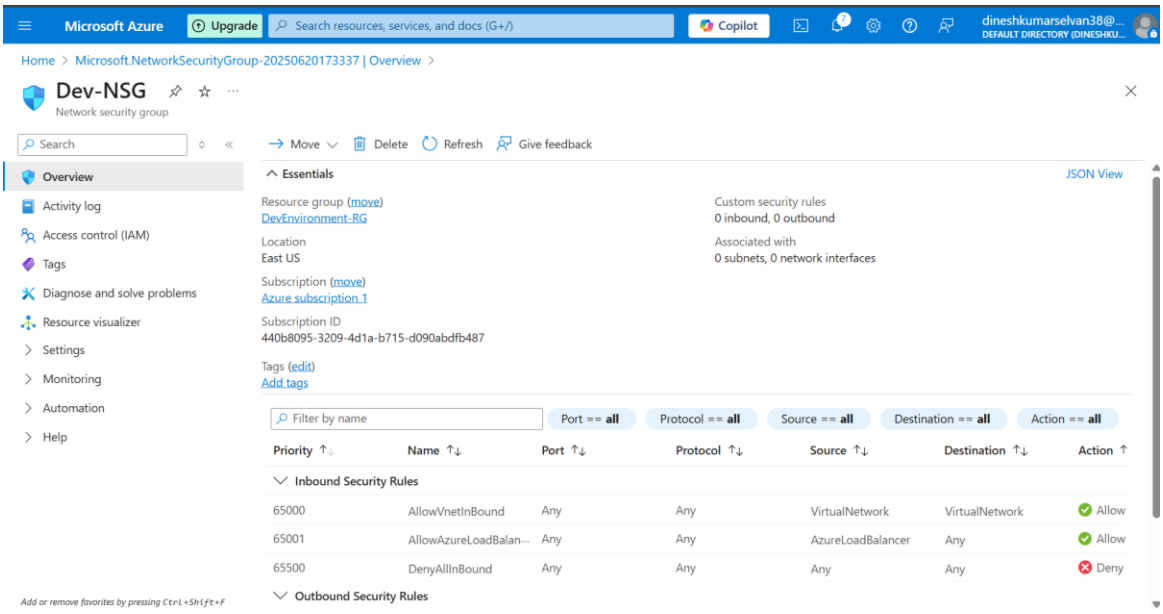
In **availability zone: 2**,

With the storage size: **Standard B1s (1 vcpu, 1 GiB memory),**

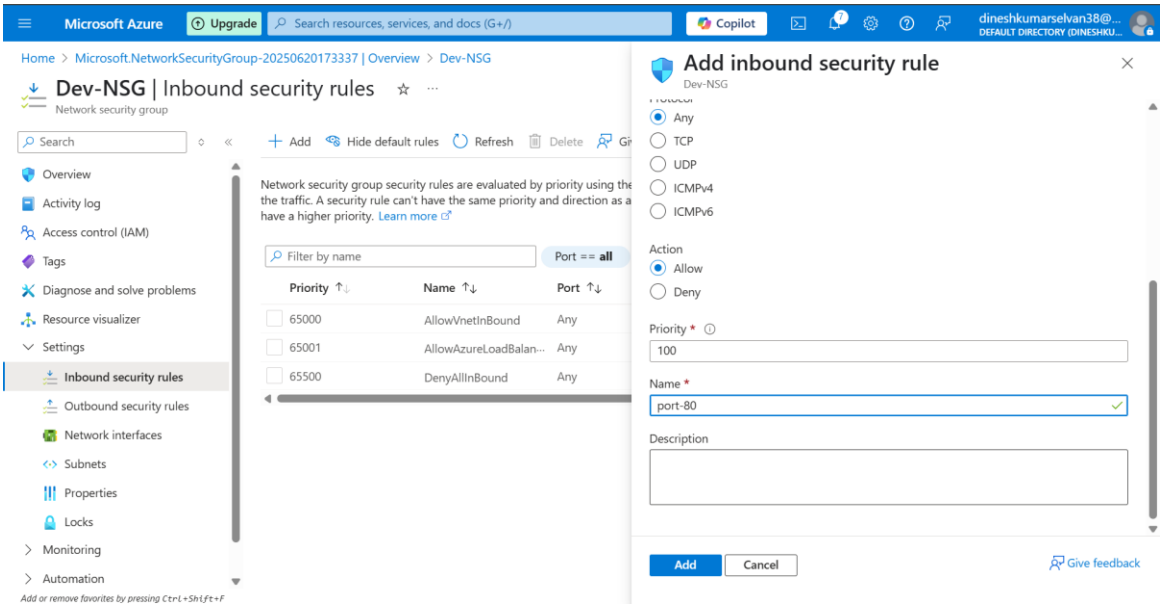
Image: **Ubuntu 24.04LTS – Gen2**,

located in East US, connected to the **Dev-Vnet** virtual network and the **Dev-subnet** subnet, and adding **tags**: Project=DevEnvironment, Owner=azure-dev-user. For development environment.

6. Network Security Group (NSG):

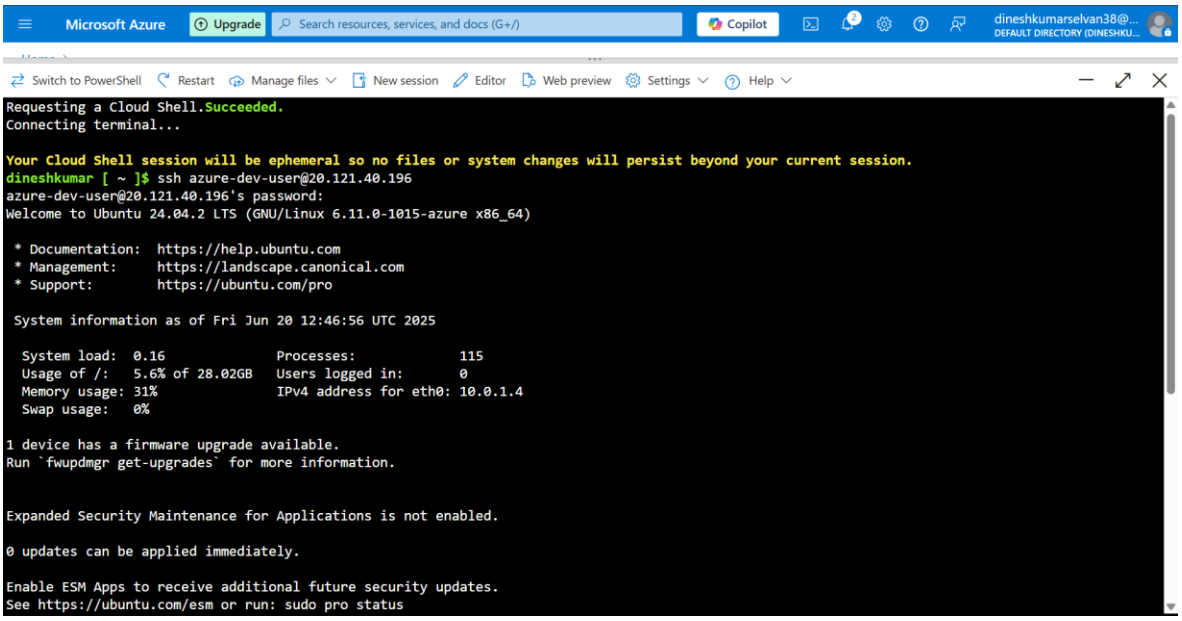


In the Azure Portal, search for **"Network Security Groups"** and click **+ Create**. Choose your subscription and resource group, then name it **Dev-NSG**. Select a region that matches your other resources (e.g., East US). Click **Review + Create**, then hit **Create** to deploy the NSG.

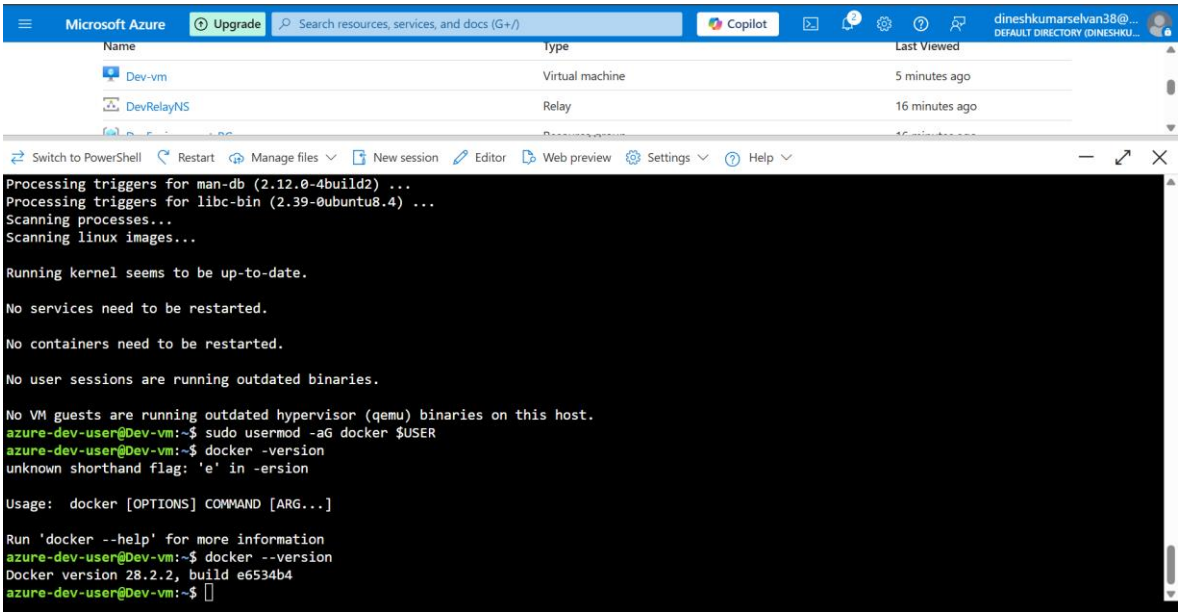


In the Azure Portal, go to **Network Security Groups** and select **Dev-NSG**. Click on **Inbound security rules**, then select **+ Add** to create a rule. Add a rule for **port 80** with priority **100**, and another for **port 20** with priority **110**. Set both to **Allow** and click **Add** to apply the rules.

7. Docker Installation on VM:

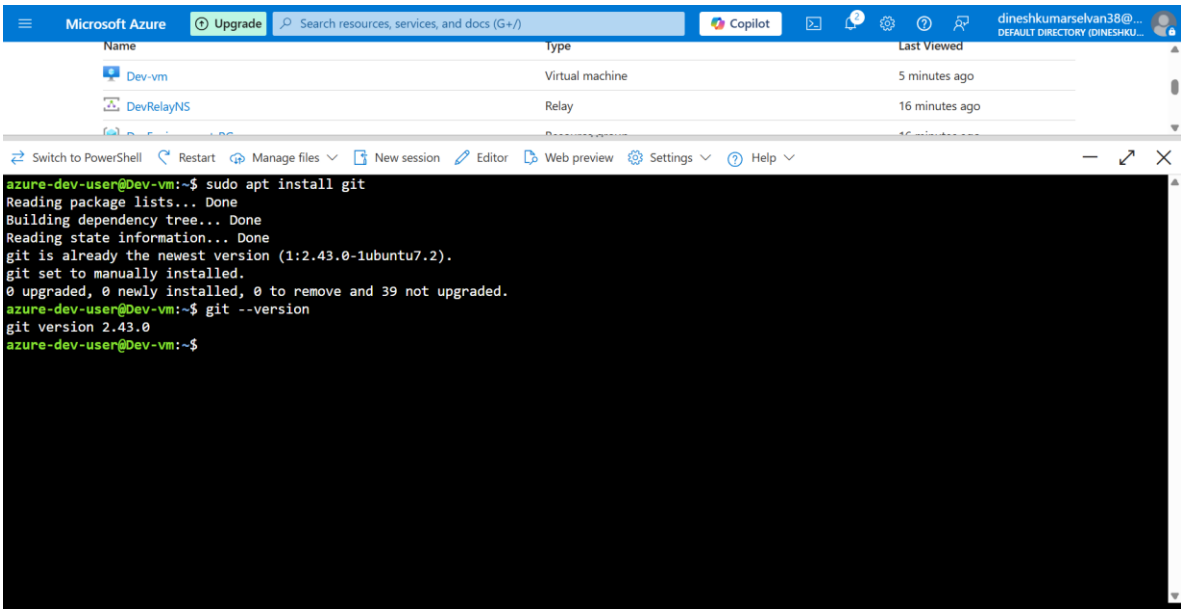


In the Cloud Shell, ensure you have SSH access enabled on the VM. Use the command: ssh azure-dev-user@20.121.40.196 to start the connection. If prompted, type "yes" to accept the fingerprint and enter the password if required. Once authenticated, you'll be logged into the **Dev-vm** via SSH.



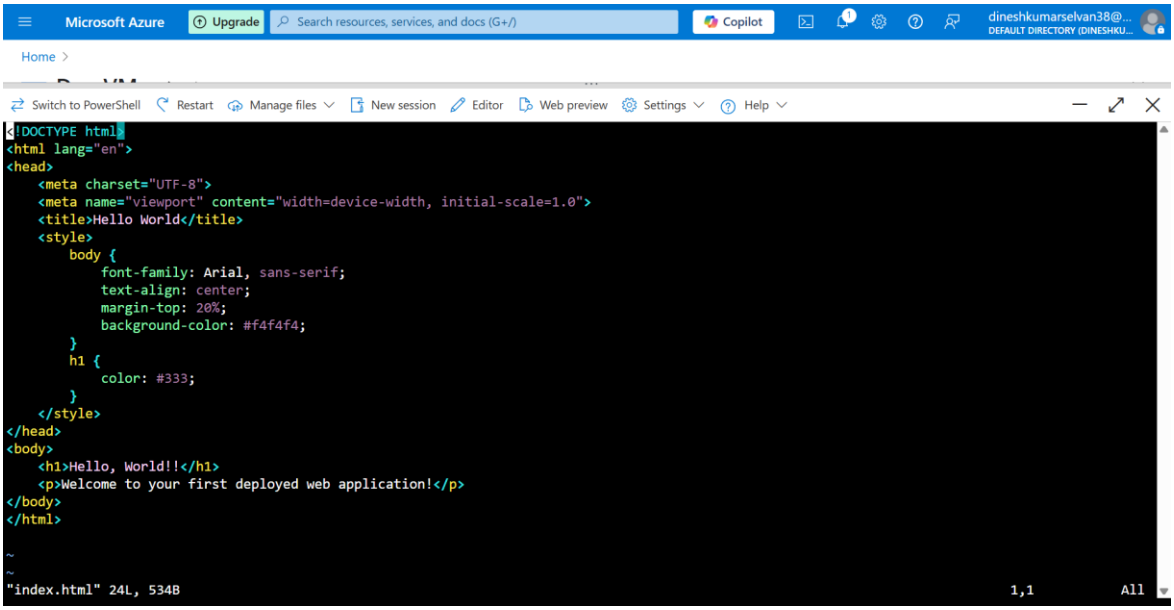
In the Dev-vm terminal, start by updating package info: sudo apt update. Install required dependencies: sudo apt install ca-certificates curl gnupg. Use the official script or repository to install Docker version **28.2.2** specifically. Verify installation with docker --version to confirm it's set up correctly.

8. Git Installation on VM:



In the Dev-vm terminal, first update the package list using sudo apt update. Download Git version **2.43.0** source from the official repository or GitHub. Compile and install it with make and sudo make install after extraction. Verify the installation using git --version to confirm it's 2.43.0.

9. HTML and CSS Web Application:



I created a 'Hello World' web application using HTML and inline CSS in the name index.html inside the hello-world-app folder. I then used Git for source code management and pushed the code to Azure Repos.

HTML and CSS:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Hello World</title>

<style>

body {

font-family: Arial, sans-serif;

text-align: center;

margin-top: 20%;

background-color: #f4f4f4;

}

h1 {

color: #333;

}

</style>

</head>

<body>

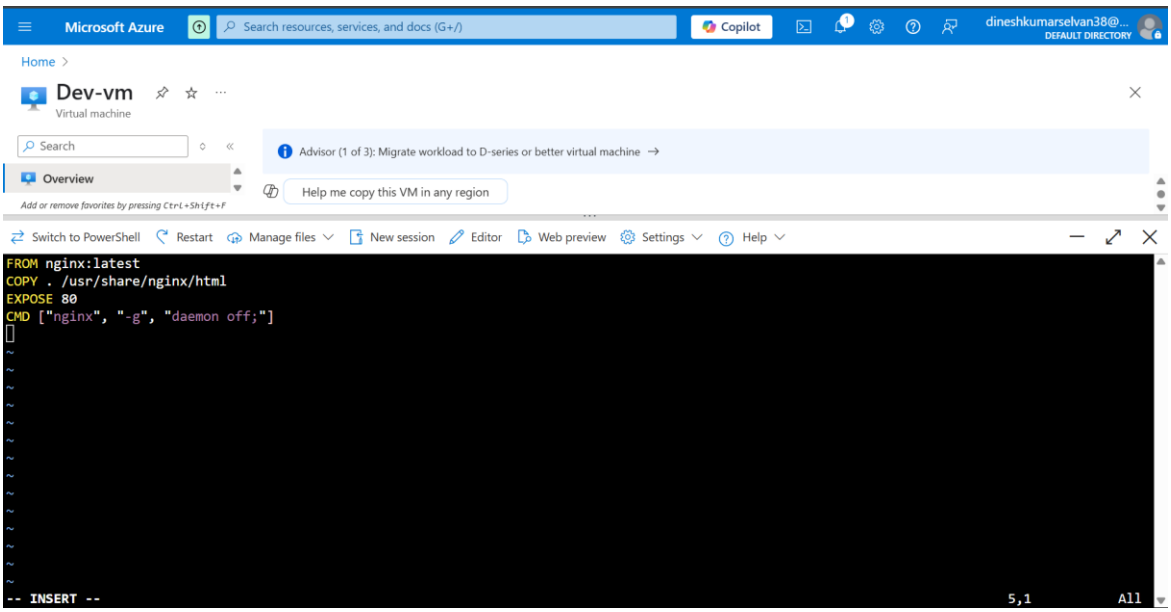
<h1>Hello, World!!</h1>

** Welcome to your first deployed web application! **

</body>

</html>

10. Dockerfile:



Create Dockerfile:

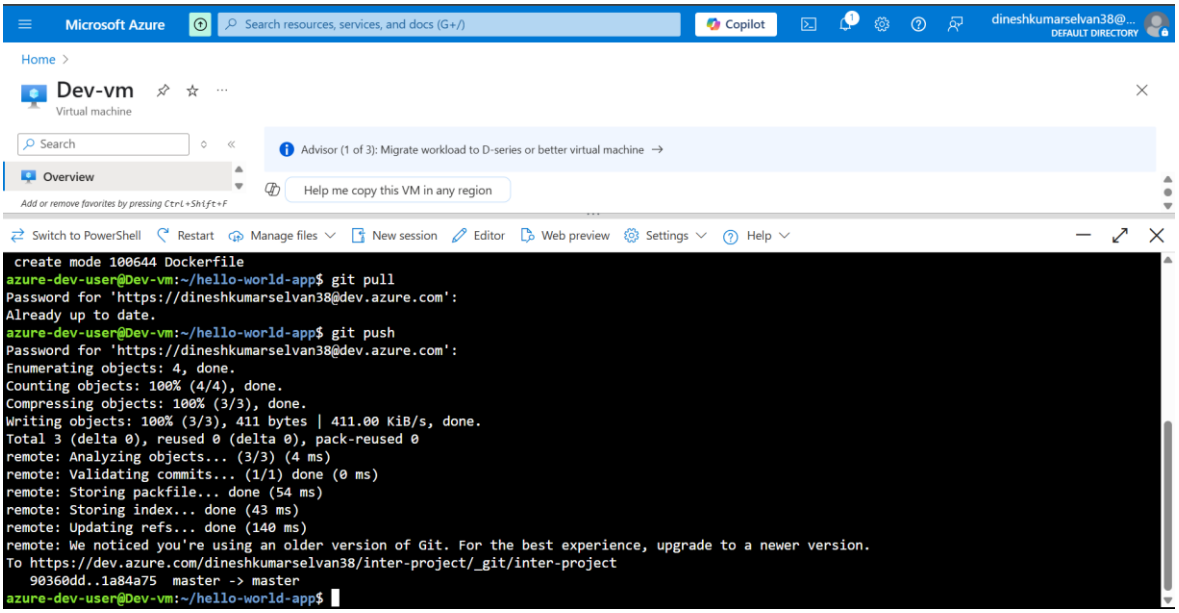
Dockerfile:

FROM nginx:latest

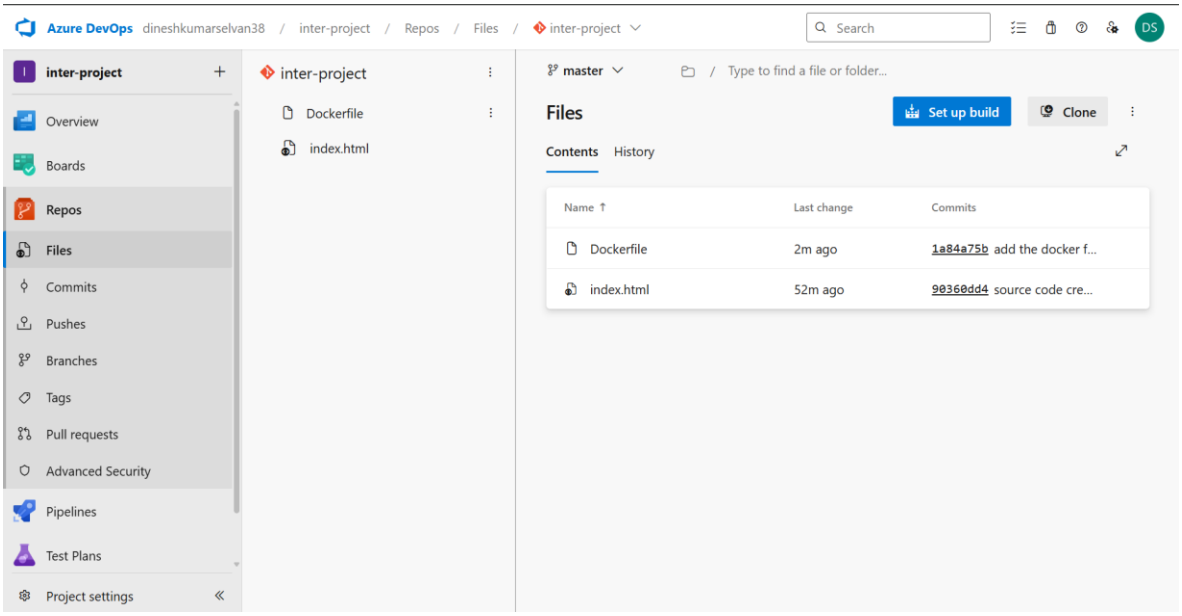
COPY . /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

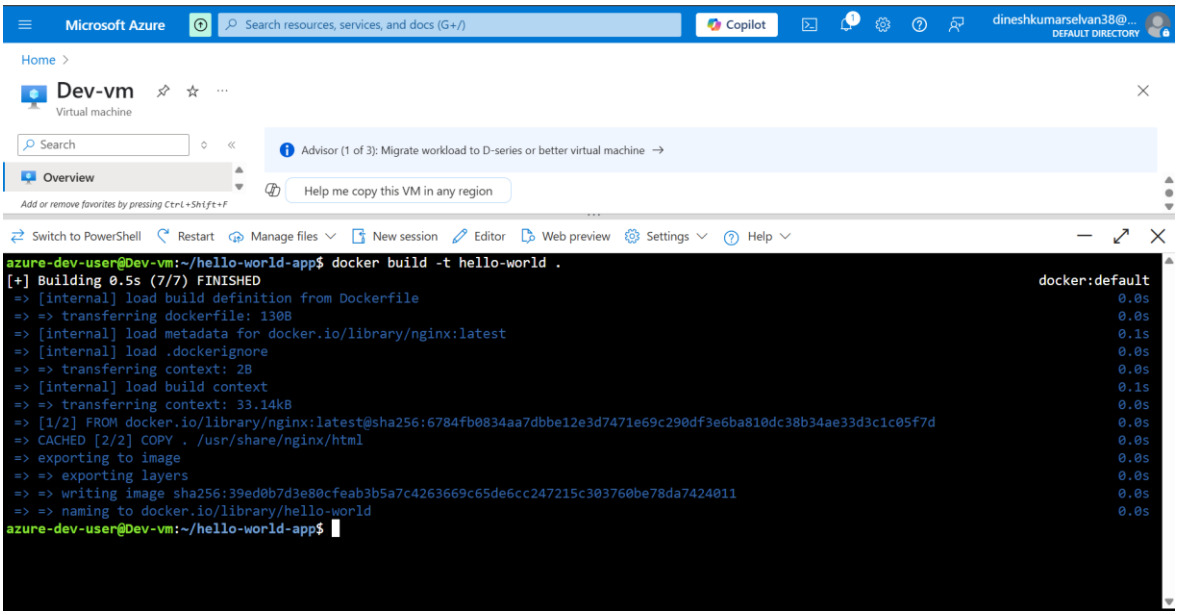


Push the dockerfile in azure repos.

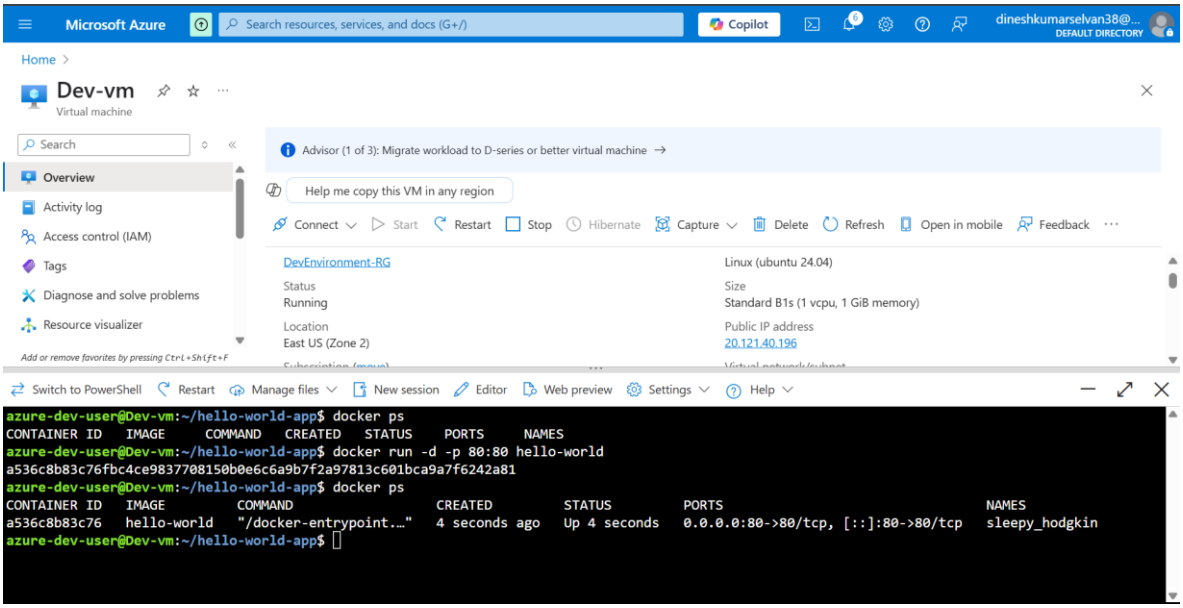


After the push operation the files appeared in the azure repos.

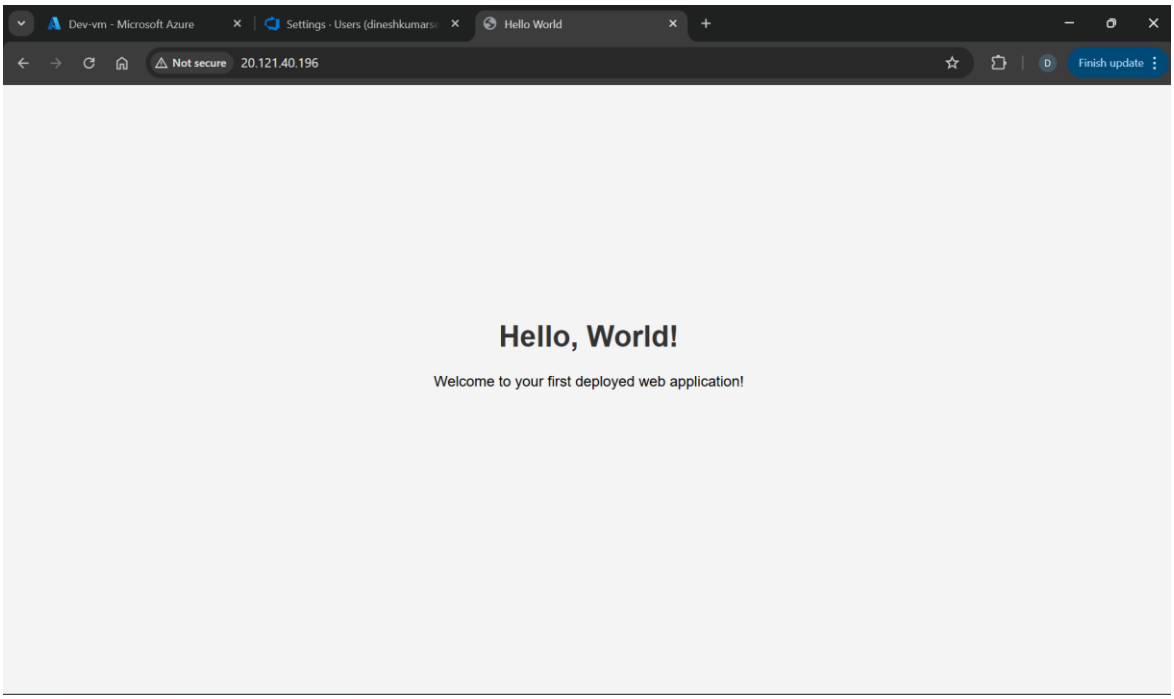
11. Docker Commands:



In the Dev-vm terminal, navigate to the directory containing your Dockerfile. Run docker build -t hello-world . to build the image and tag it as **hello-world**. This command reads the Dockerfile and packages the application into a container image. Once completed, check the image with docker images to confirm it's listed.

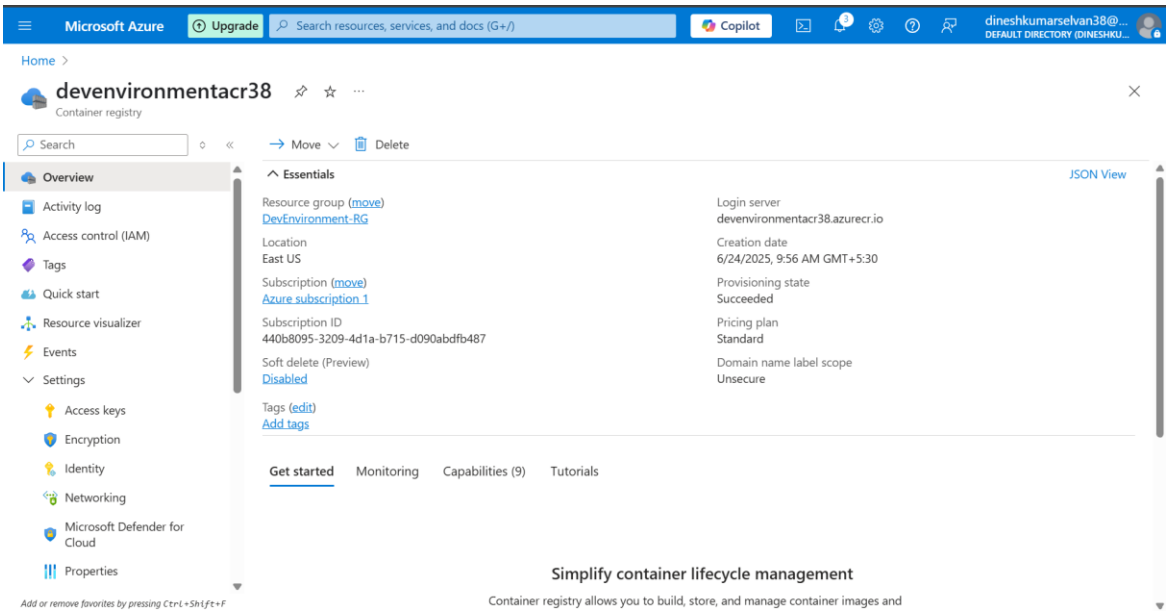


In the Dev-vm terminal, use the docker run command to start the container. Use the -d flag to run it in detached (background) mode. Use the -p flag to map a host port to a container port, like -p 8080:80. The full command looks like: docker run -d -p 8080:80 hello-world.



Open your web browser and enter the public IP of the VM followed by port 80. For example: http://20.121.40.196 (since port 80 is the default for HTTP). If everything is configured correctly, the web page or app will load. This confirms your Docker container is serving traffic successfully.

12. Azure Container Registry (ACR):

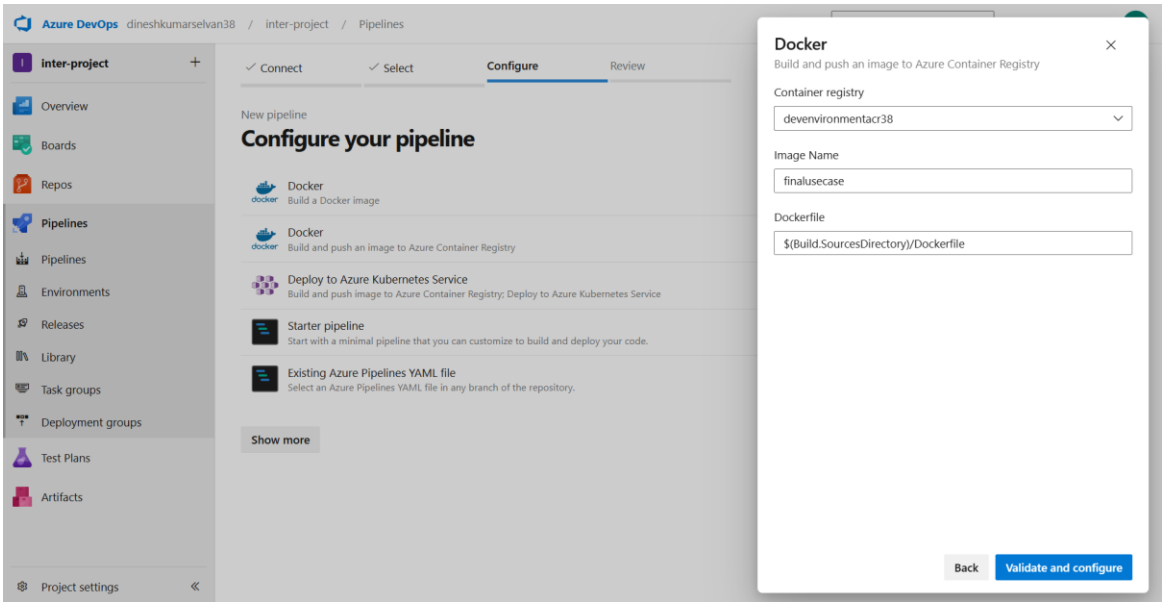


In the Azure Portal, search for **"Container Registries"** and click **+ Create**. Select your subscription and resource group, then name it **devenvironmentacr38**. Choose a region (same as your other resources), and keep the SKU as **Basic** for simplicity. Click **Review + Create**, then hit **Create** to deploy your ACR.

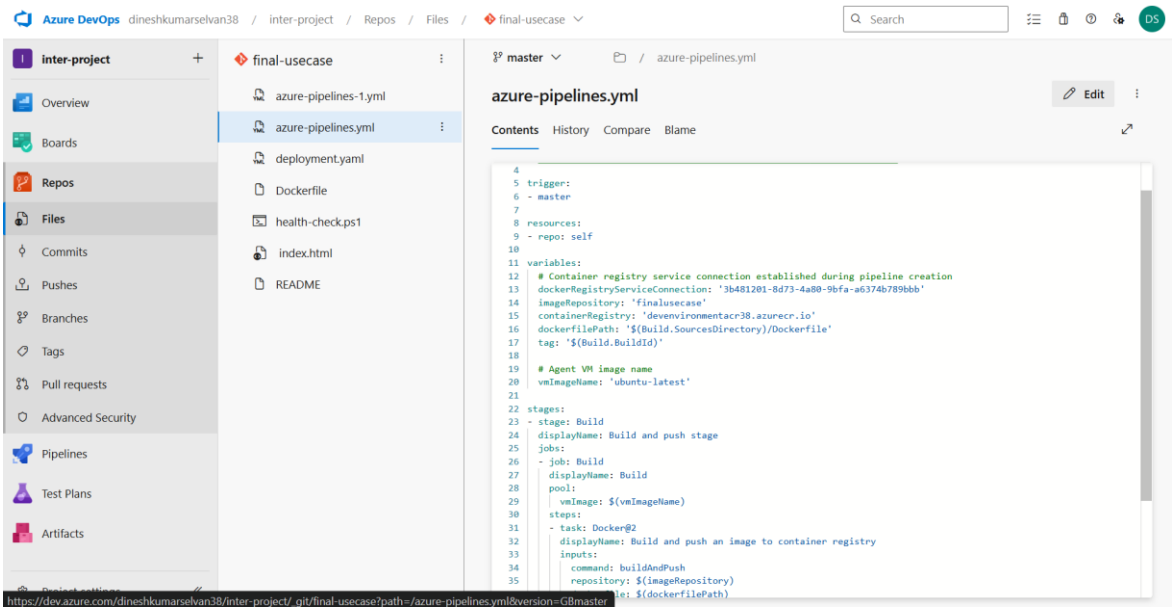
Step-by-Step CI Tasks:

1. Continuous Integration (CI): Build Pipeline:

Create a New Build Pipeline in Azure DevOps:



In your Azure DevOps repo, create a YAML file under .azure-pipelines/. Define steps to docker build your app and docker push to ACR using service connection. Include loginServer from your ACR and use variables for image name and tag. Run the pipeline to build the image and push it securely to **devenvironmentacr38**.



After the creation of the YAML pipeline it add azure-pipeline.yml file in the azure repos.

azure-pipeline.yml:

```
# Docker
# Build and push an image to Azure Container Registry
# https://docs.microsoft.com/azure/devops/pipelines/languages/docker
trigger:
- master
resources:
- repo: self
variables:
 # Container registry service connection established during pipeline creation
 dockerRegistryServiceConnection: '3b481201-8d73-4a80-9bfa-a6374b789bbb'
 imageRepository: 'finalusecase'
 containerRegistry: 'devenvironmentacr38.azurecr.io'
 dockerfilePath: '$(Build.SourcesDirectory)/Dockerfile'
 tag: '$(Build.BuildId)'
 # Agent VM image name
 vmImageName: 'ubuntu-latest'
stages:
- stage: Build
 displayName: Build and push stage
 jobs:
 - job: Build
 displayName: Build
```

pool:

vmImage: $(vmImageName)

steps:

- task: Docker@2

displayName: Build and push an image to container registry

inputs:

command: buildAndPush

repository: $(imageRepository)

dockerfile: $(dockerfilePath)

containerRegistry: $(dockerRegistryServiceConnection)

tags: |

$(Build.BuildId)

- task: PublishBuildArtifacts@1

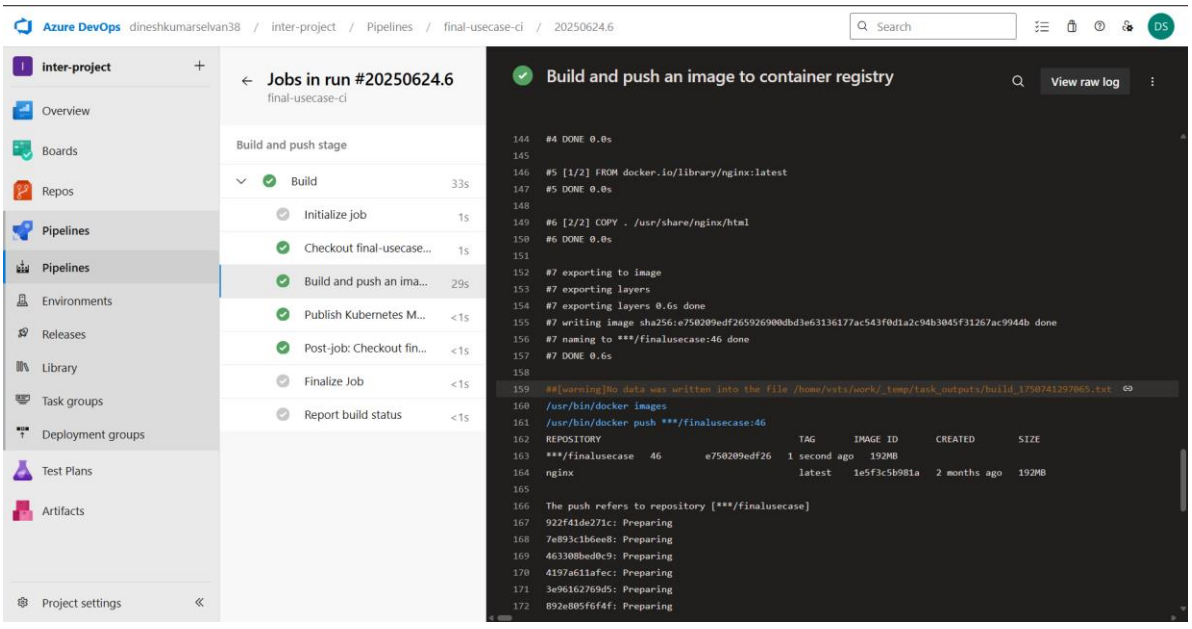
displayName: 'Publish Kubernetes Manifest'

inputs:

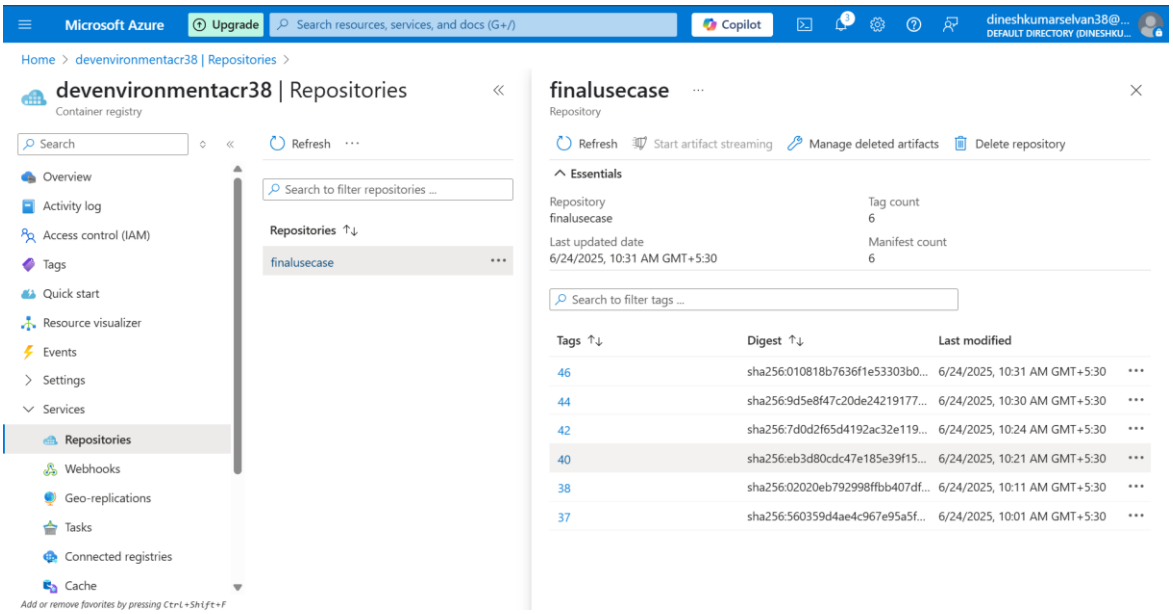
PathtoPublish: '$(Build.SourcesDirectory)/deployment.yaml'

ArtifactName: 'drop'

publishLocation: 'Container'

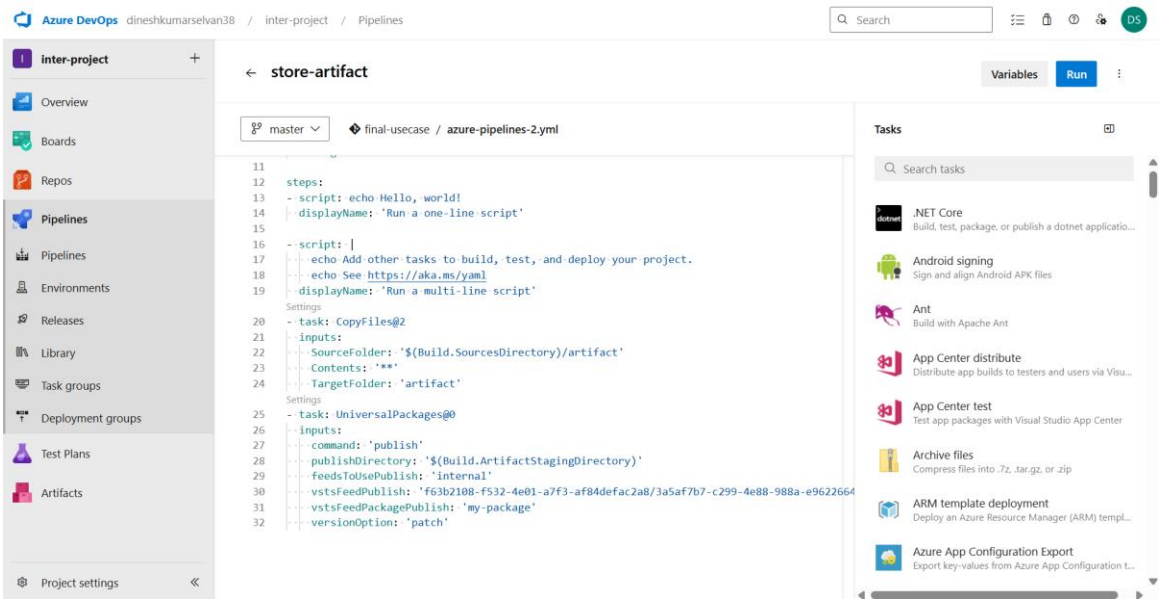


The build was success.



After the successful build, the Docker image was tagged as **finalusecase**. It was then pushed securely to the Azure Container Registry (ACR). Navigate to your ACR and open the **Repositories** section. You'll find **finalusecase** listed, confirming the image is stored and ready.

2. store the artifact in azure artifact using pipeline:



Create a starter pipeline and add the file copy and universal package task using assistance using YAML.

azure-pipeline-2.yml:

 # Starter pipeline

 # Start with a minimal pipeline that you can customize to build and deploy your code.

Add steps that build, run tests, deploy, and more:

 # https://aka.ms/yaml

trigger:

- master

pool:

vmImage: ubuntu-latest

steps:

- script: echo Hello, world!

displayName: 'Run a one-line script'

- script: |

echo Add other tasks to build, test, and deploy your project.

echo See https://aka.ms/yaml

displayName: 'Run a multi-line script'

- task: CopyFiles@2

inputs:

SourceFolder: '$(Build.SourcesDirectory)/artifact'

Contents: ''**

TargetFolder: 'artifact'

- task: UniversalPackages@0

inputs:

command: 'publish'

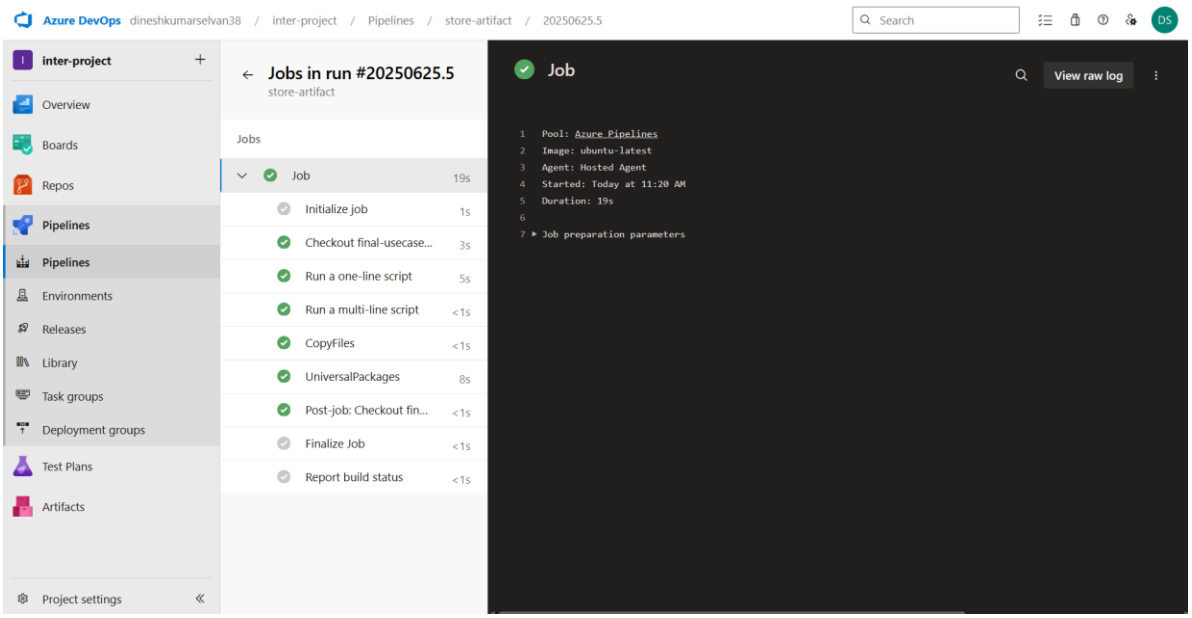
publishDirectory: '$(Build.ArtifactStagingDirectory)'

feedsToUsePublish: 'internal'

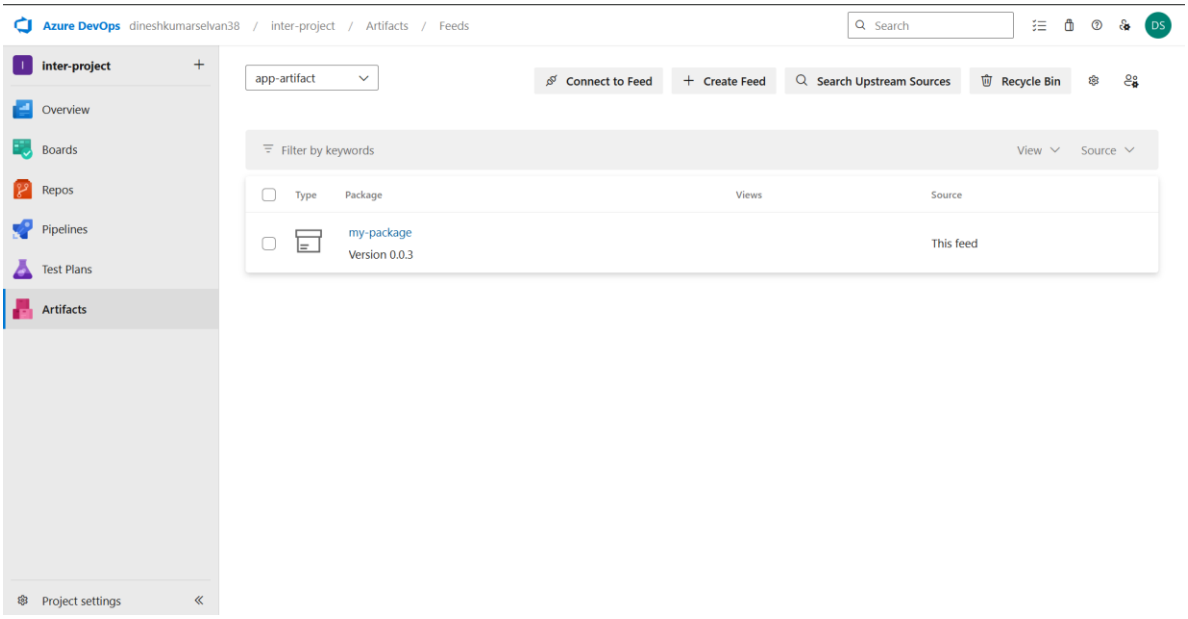
vstsFeedPublish: 'f63b2108-f532-4e01-a7f3-af84defac2a8/3a5af7b7-c299-4e88-988a-e9622664a6d8'

vstsFeedPackagePublish: 'my-package'

versionOption: 'patch'



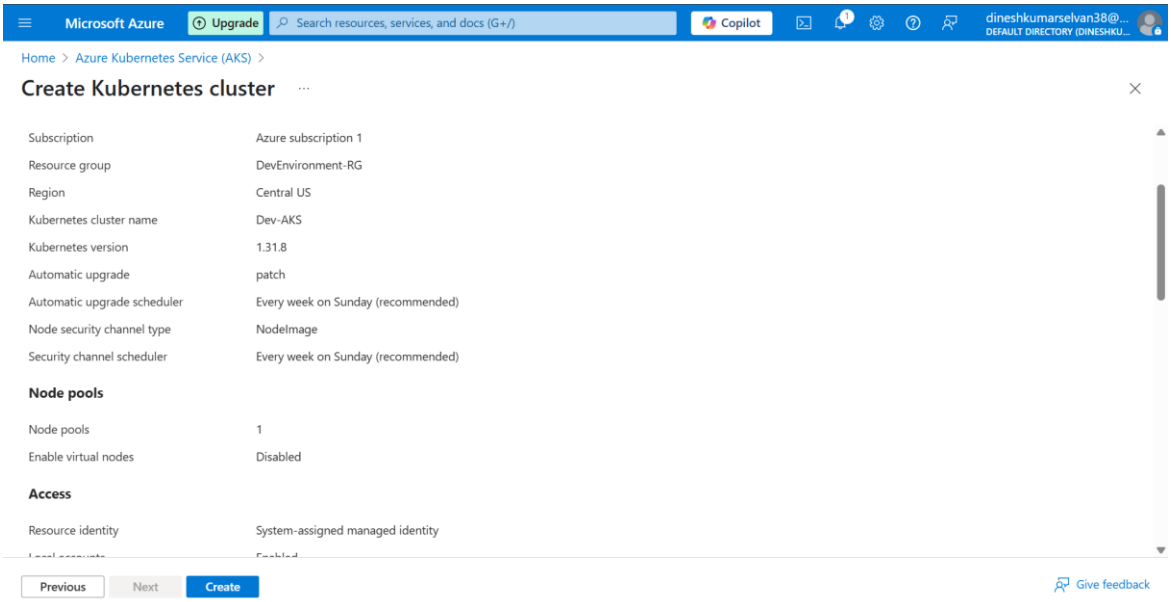
The build was success.



After a successful build, the pipeline publishes the artifact to **Azure Artifacts**. Navigate to **Azure DevOps > Artifacts**, and select your feed. You'll find the artifact listed with its version (e.g., 1.0.0 or custom tag). This confirms versioned artifacts are stored and available for use or deployment.

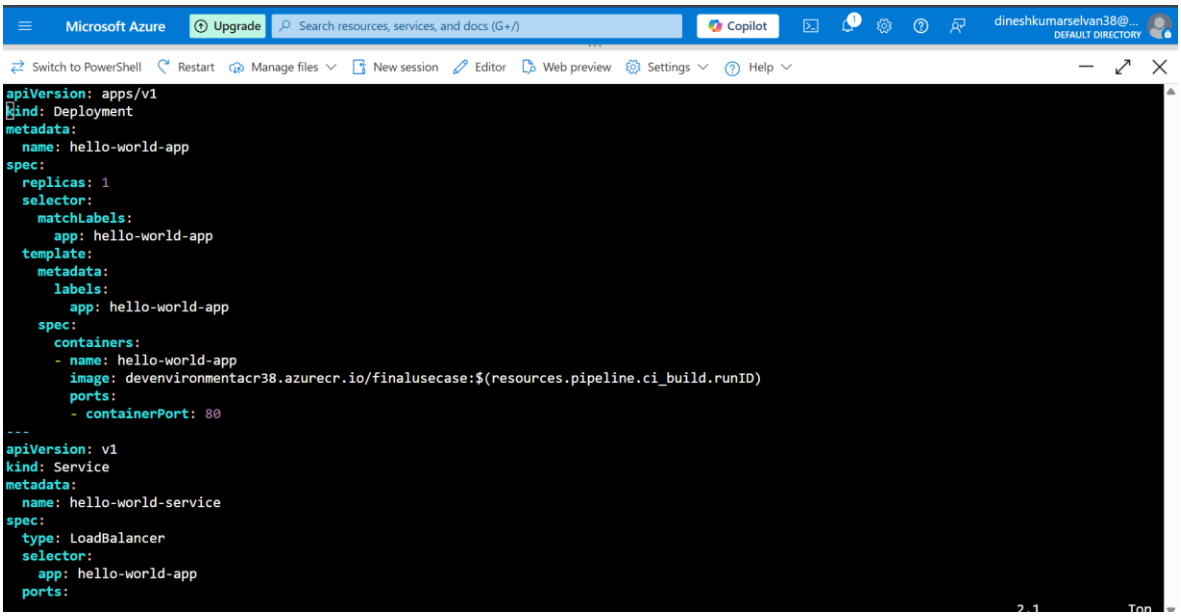
Required Setup and Installations for CD:

1. create AKS:



In the Azure Portal, search for **"Kubernetes services"** and click **+ Create**. Choose your subscription, resource group, and name the cluster **Dev-AKS**. Set the node pool to have **1 node** and choose the **Standard_D2ps_v6** VM size. Click **Review + Create**, then hit **Create to deploy the AKS cluster.**

2. Kubernetes Deployment Manifest:



Create the deployment.yml file with service for the load balancer type in it and push it to the azure repos.

deployment.yml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: hello-world-app

spec:

replicas: 1

selector:

matchLabels:

app: hello-world-app

template:

metadata:

labels:

app: hello-world-app

spec:

containers:

- name: hello-world-app

image: devenvironmentacr38.azurecr.io/finalusecase:$(resources.pipeline.ci_build.runID)

ports:

- containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

name: hello-world-service

spec:

type: LoadBalancer

selector:

app: hello-world-app

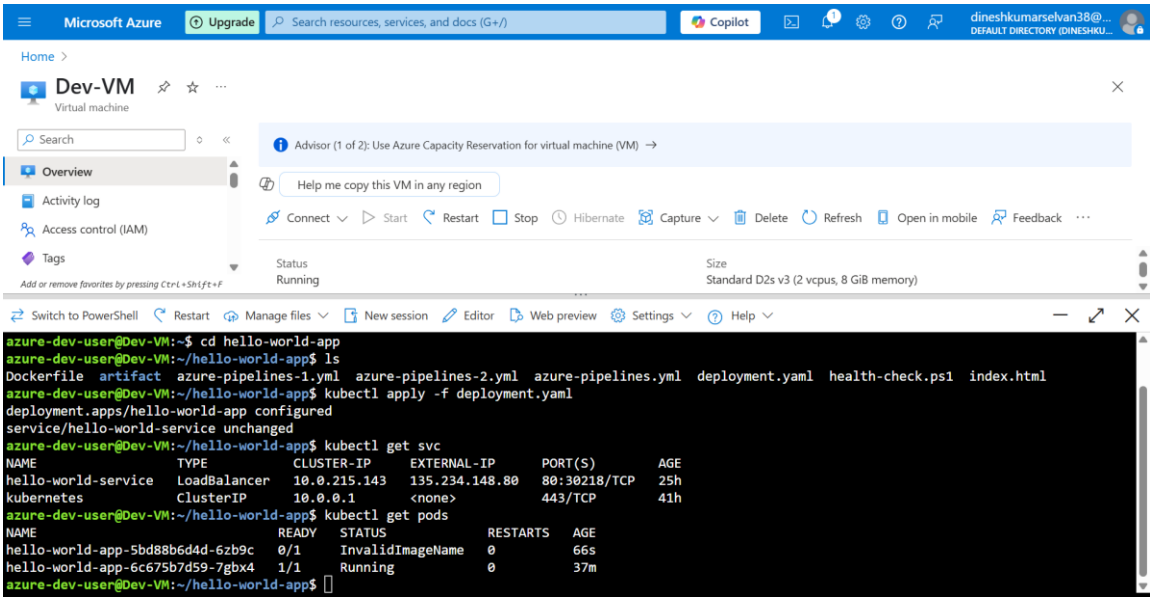
ports:

- protocol: TCP

port: 80

targetPort: 80

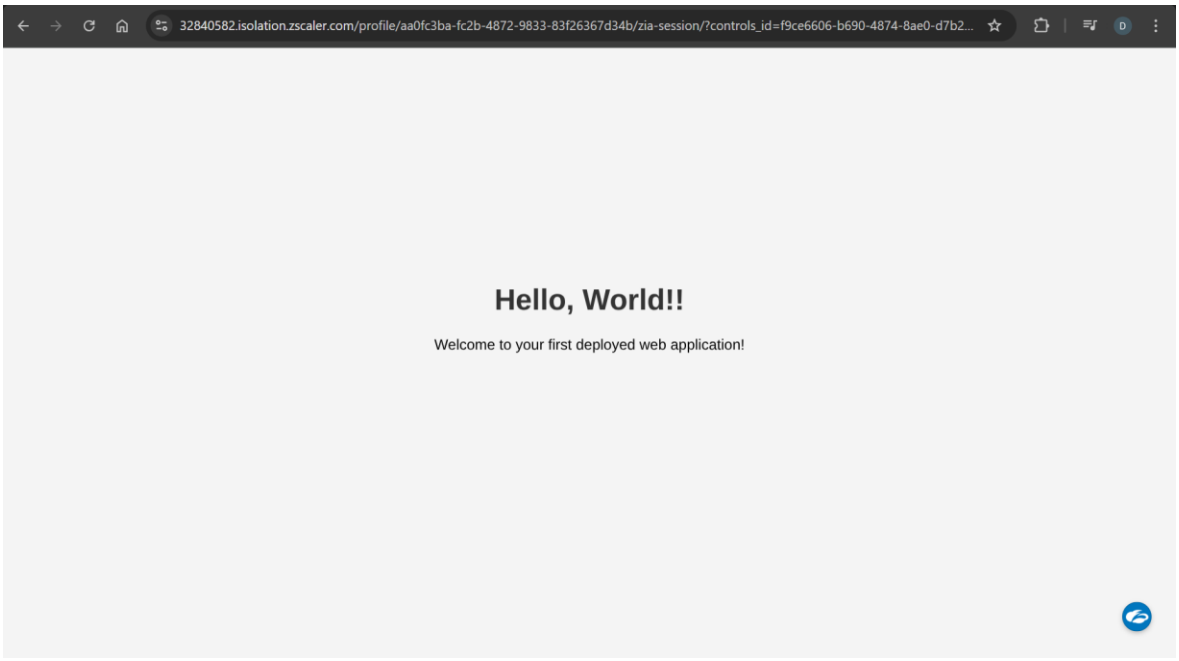
3. run the cluster in VM:



By using **kubctl apply** command we can able apply a configuration to a resource by file, **-f** by using this tag we can mention filename, **deployment.yaml** is the file we have the instruction in YAML format.

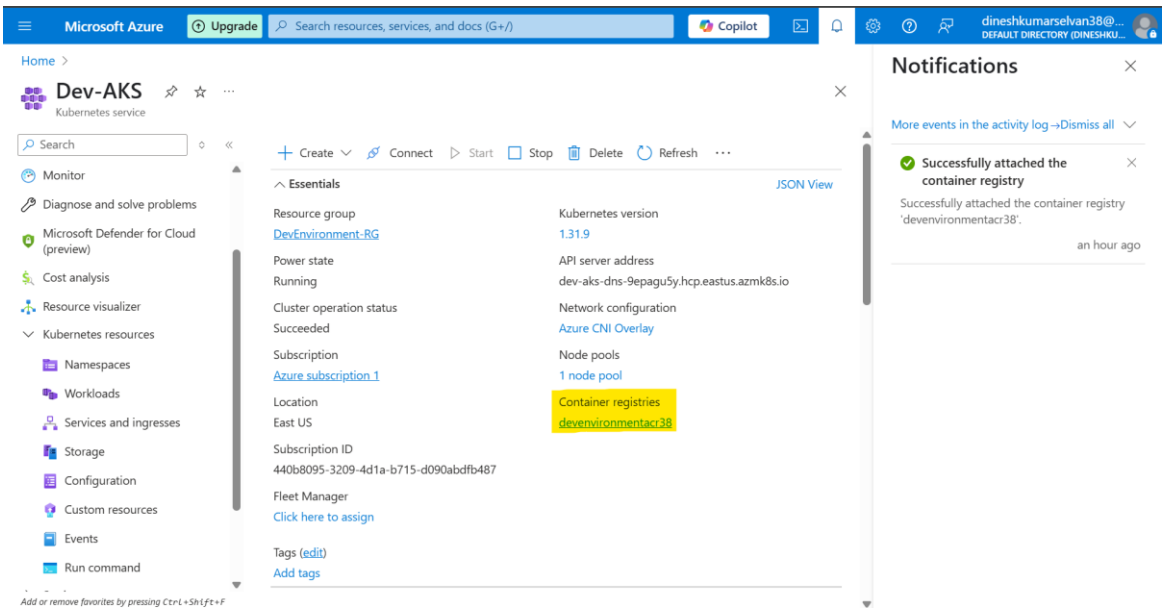
By using **kubectl get pods** to show the running pods.

By using **kubectl get svc** to show the running services.



In the Azure Portal, go to **Kubernetes services** and select your AKS cluster. Navigate to **Services and Ingress** under the **Workloads** menu. Locate the service exposing your app and note the **External IP** assigned. Open a web browser and enter that IP—your deployed web app should appear!

4. connection between Azure Kubernetes Service (AKS) and Azure Container Registry (ACR):

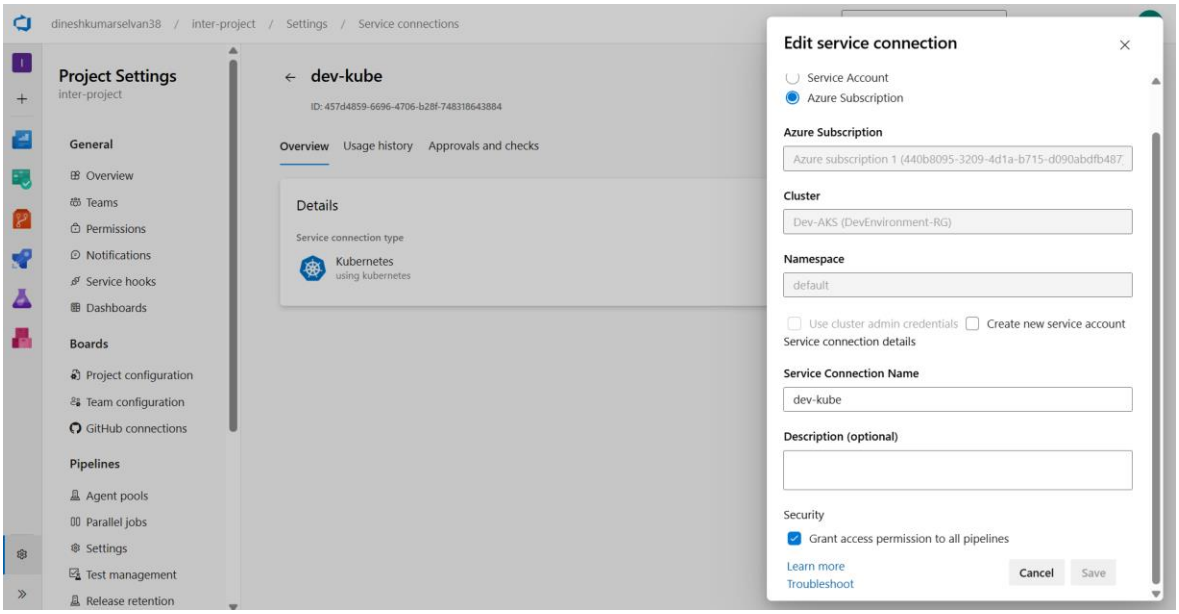


In the Azure Portal, go to your **Dev-AKS** Kubernetes cluster. Under **Settings**, select **Integrations**, then click on **Container registry**. Choose your registry, such as **devenvironmentacr38**, from the dropdown list. Click **Apply** to establish the connection between AKS and your ACR.

Step-by-Step CD Tasks:

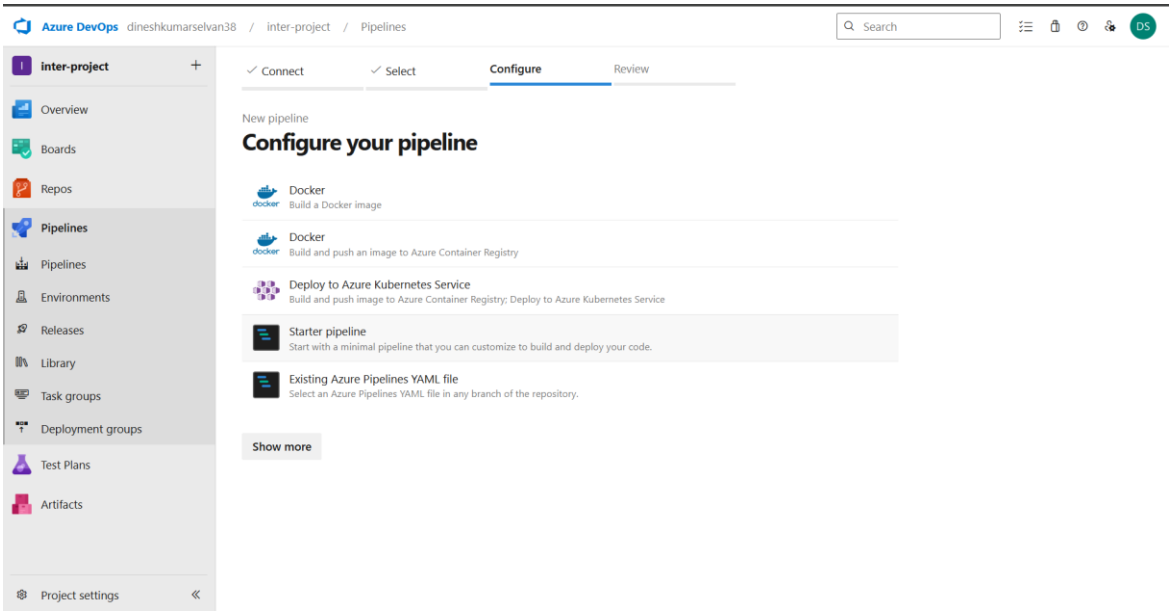
1. Continuous Deployment (CD): Build Pipeline:

1.1 Create a service connection for Kubernetes:

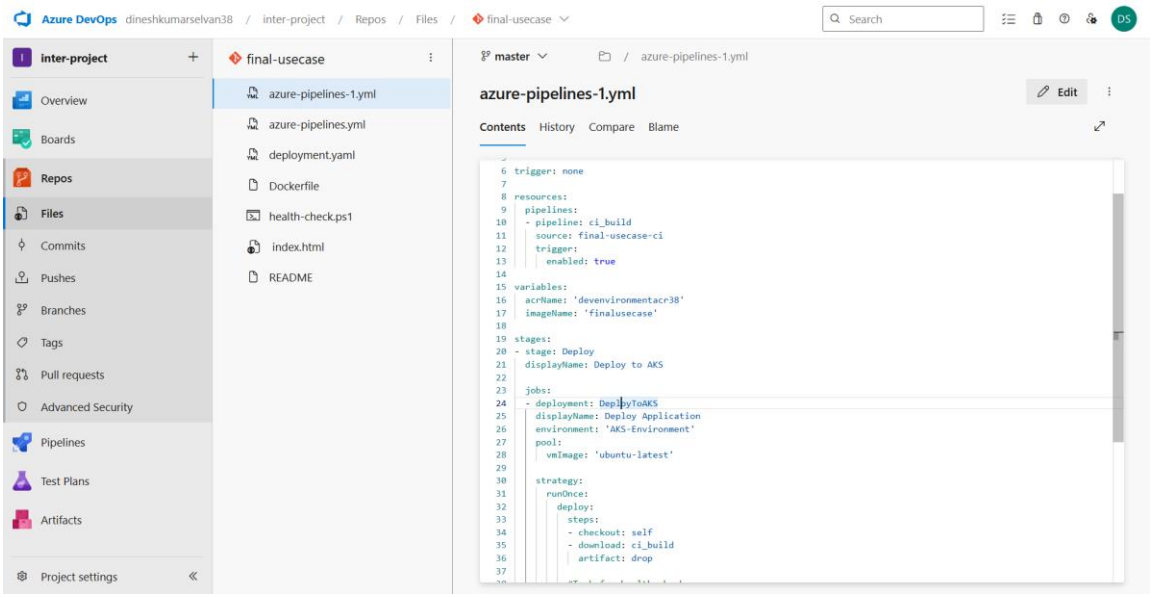


In the Azure Portal, navigate to **Kubernetes services** from the homepage or search bar. Choose the correct **subscription** from the dropdown at the top of the page. From the list of clusters, locate and click on the **Dev-AKS** cluster you created. This opens the overview and configuration settings for managing your AKS cluster.

1.2 Create a New Build Pipeline in Azure DevOps:



In Azure DevOps, go to Pipelines and click **"New Pipeline"**. Choose your repository and select **"Starter pipeline"** as the template. Azure generates a basic YAML file with predefined trigger and jobs. Customize it as needed and click **Save and Run** to execute your pipeline.



After creating the YAML pipeline in Azure DevOps, it gets saved automatically. The file is named **azure-pipeline-1.yml** and added to your Azure Repos repository. You can find it under the root directory or .azure-pipelines/ folder. It serves as the blueprint for your build and deployment pipeline.

azure-pipeline-1.yml:

 # Starter pipeline

 # Start with a minimal pipeline that you can customize to build and deploy your code.

 # Add steps that build, run tests, deploy, and more:

 # https://aka.ms/yaml

trigger: none

resources:

pipelines:

- pipeline: ci_build

source: final-usecase-ci

trigger:

enabled: true

variables:

acrName: 'devenvironmentacr38'

imageName: 'finalusecase'

stages:

- stage: Deploy

displayName: Deploy to AKS

jobs:

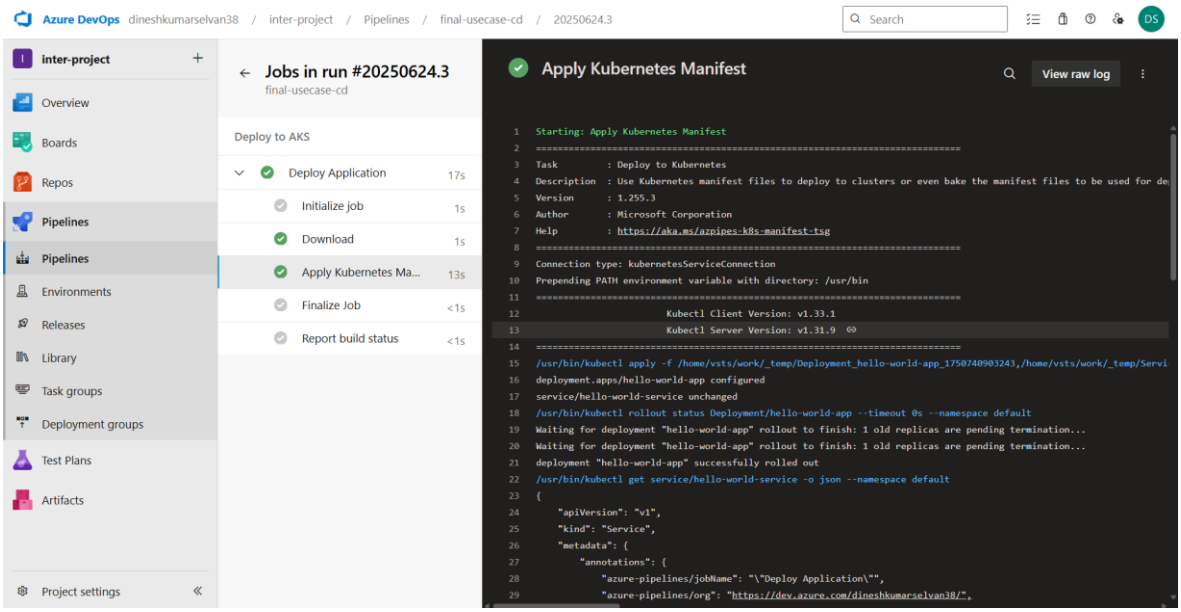
- deployment: DeployToAKS

displayName: Deploy Application

environment: 'AKS-Environment'

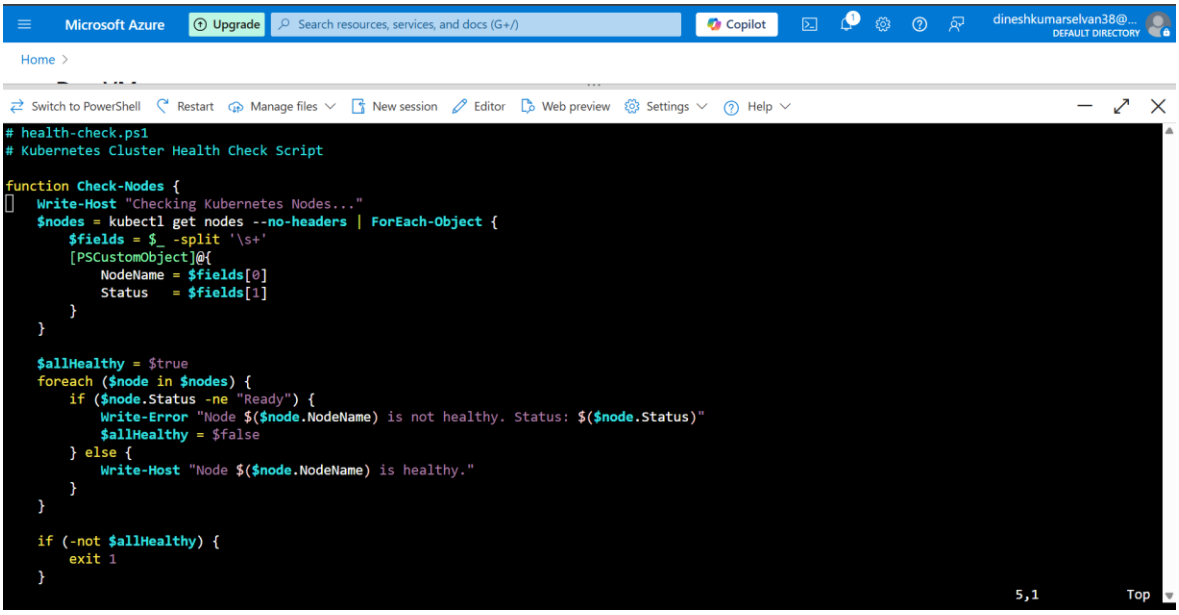
pool:

```
vmImage: 'ubuntu-latest'
strategy:
runOnce:
deploy:
steps:
- checkout: self
- download: ci_build
artifact: drop
#Task for health check
- task: PowerShell@2
displayName: 'Kubernetes Cluster Health Check'
inputs:
targetType: 'filePath'
filePath: 'health-check.ps1'
pwsh: true
# Task to apply the Kubernetes manifest to your AKS cluster
- task: KubernetesManifest@1
displayName: Apply Kubernetes Manifest
inputs:
kubernetesServiceConnection: 'AKS-Environment-Dev'
namespace: 'default'
manifests: |
$(Pipeline.Workspace)/ci_build/drop/deployment.yaml
containers: |
$(acrName).azurecr.io/$(imageName):$(resources.pipeline.ci_build.runID)
```



The build was success.

2. Kubernetes cluster health check:

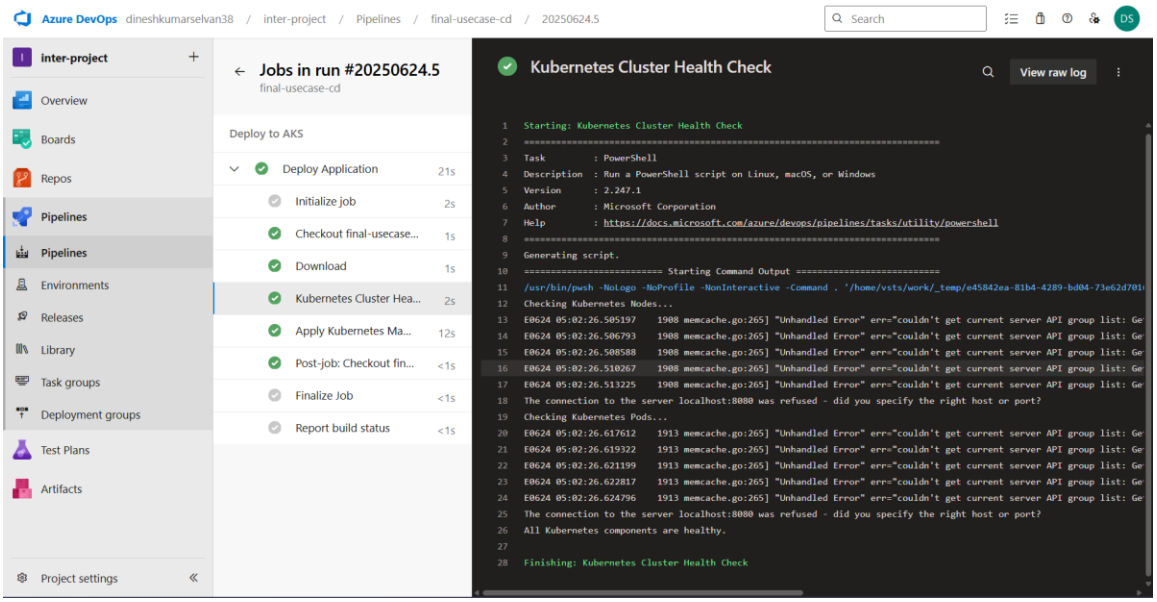


To check the health of the cluster by using powershell script called **health-check.ps**.

health-check.ps1:

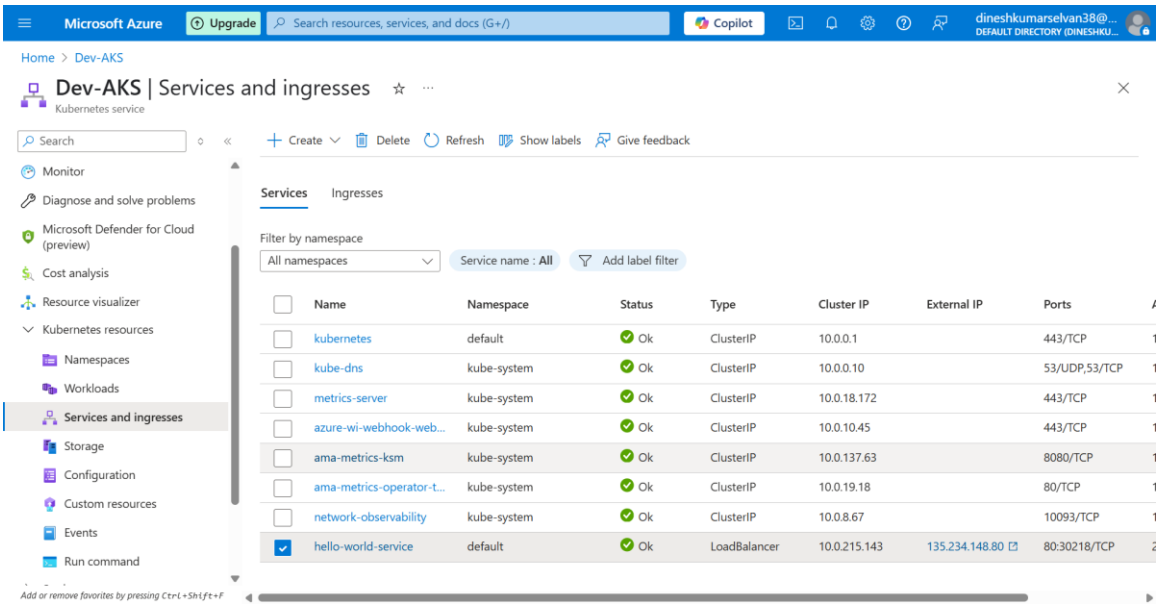
```
# health-check.ps1

# Kubernetes Cluster Health Check Script

function Check-Nodes {

Write-Host "Checking Kubernetes Nodes..."

$nodes = kubectl get nodes --no-headers | ForEach-Object {

$fields = $_ -split '\s+'

[PSCustomObject]@{

NodeName = $fields[0]

Status   = $fields[1]
```

```
}
}
$allHealthy = $true
foreach ($node in $nodes) {
if ($node.Status -ne "Ready") {
Write-Error "Node $($node.NodeName) is not healthy. Status: $($node.Status)"
$allHealthy = $false
} else {
Write-Host "Node $($node.NodeName) is healthy."
}
}
if (-not $allHealthy) {
exit 1
} }
```

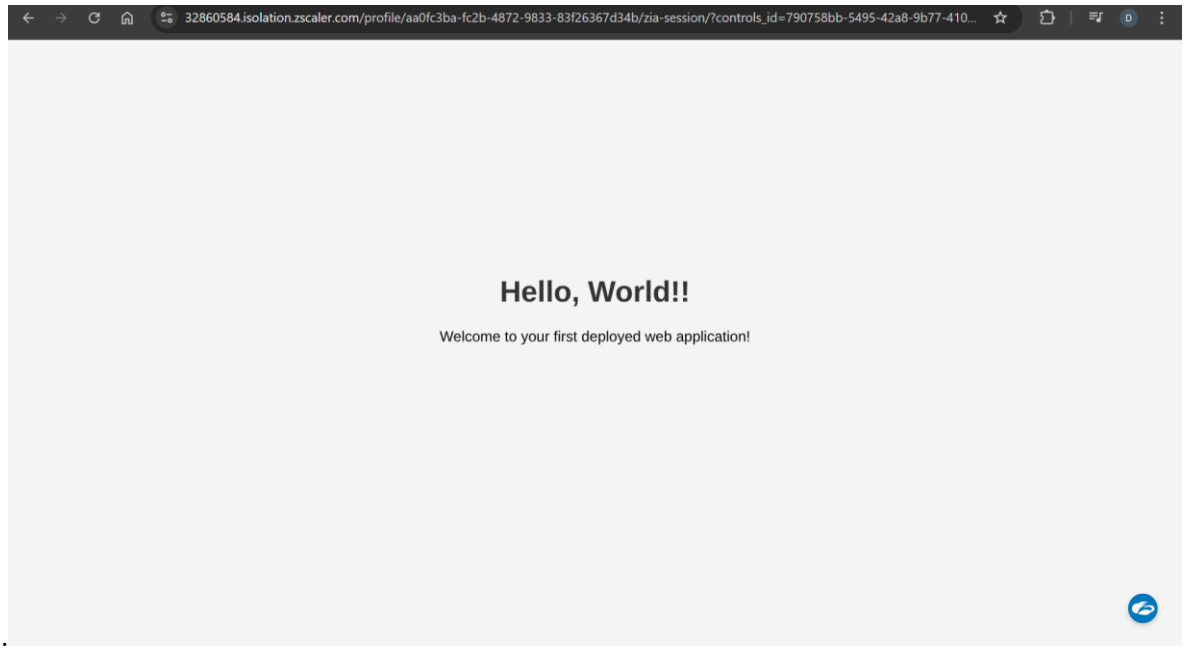


After deploying your app, perform a health check on the AKS cluster using kubectl get nodes. Verify all nodes show **Ready** status, and check pods with kubectl get pods for running state. Ensure your services are accessible and the app is responding properly via the external IP. Once everything is validated, proceed with the final build or release step in your pipeline. This confirms the Kubernetes environment is healthy and ready for production deployment.**3. final**

output:



In the Azure Portal, go to your **AKS cluster** and open **Services and Ingress**. Find the service exposing your app and note its **External IP** address. Copy the IP and open it in your web browser (e.g., http://<external-ip>). If port mapping is correct (like container port 80), your web app will load. This confirms the app is live and accessible via the external IP.

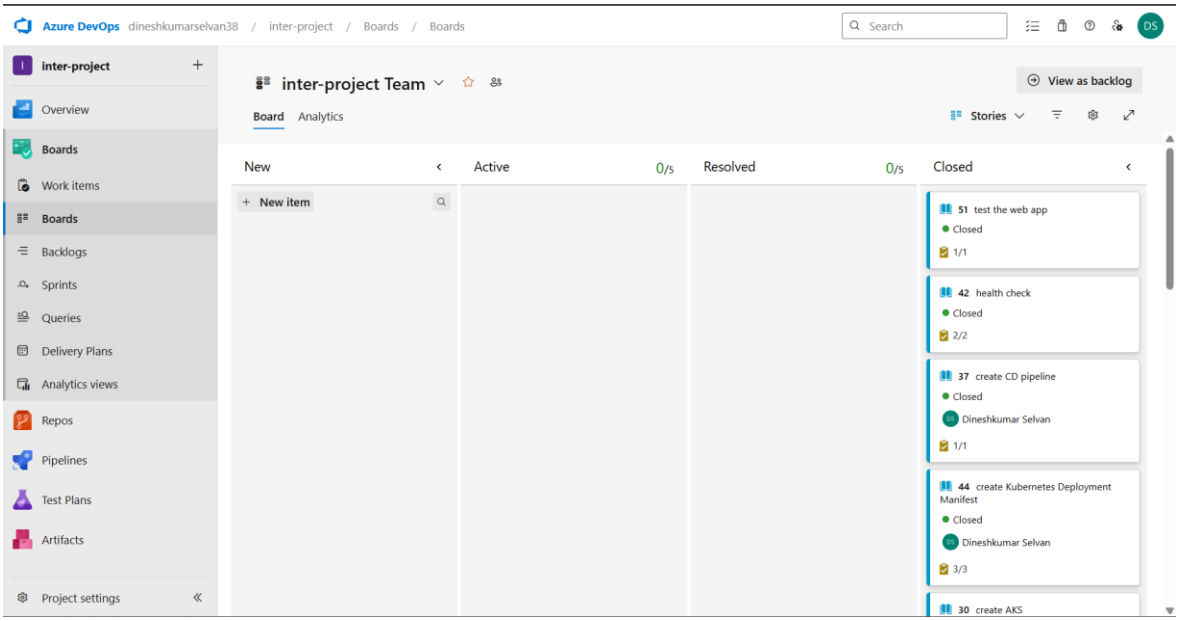


.

Conclusion:

Through this hands-on project, you've not only built and deployed a web application using Azure's free-tier services, but also navigated the core pillars of modern DevOps—including infrastructure provisioning, containerization with Docker, orchestration using AKS, and CI/CD automation with Azure Pipelines. These steps reflect real-world workflows used by DevOps engineers in professional environments.

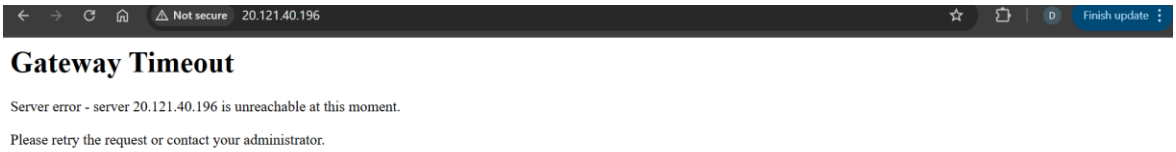
By implementing this end-to-end pipeline, you've gained essential experience in managing cloud-native applications efficiently and securely. More than just a technical exercise, this project lays a solid foundation for scaling your skills toward more advanced cloud architecture and DevOps automation challenges in the future. You're now better equipped to contribute confidently to modern software delivery lifecycles.



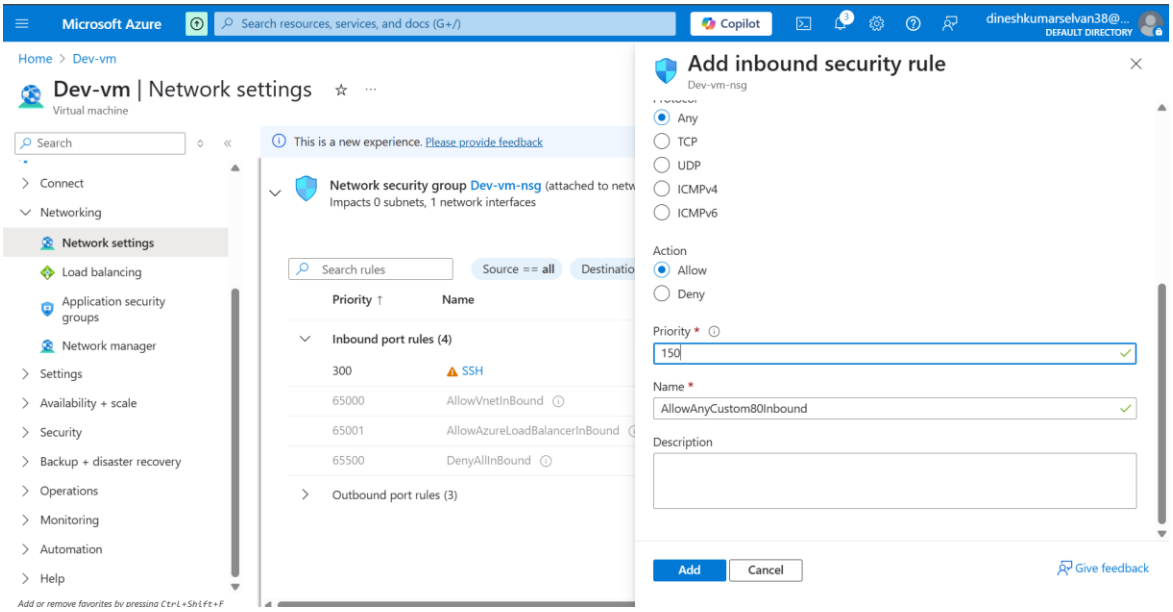
In Azure DevOps, navigate to **Boards** and select your project. Go to the **Work Items** tab to view all user stories and tasks assigned to you. Open each item, update its status to **Closed**, and add relevant comments if needed. Use filters to confirm all assigned work items are completed and closed properly. This ensures your sprint or milestone reflects accurate progress in the board.

TROUBLESHOOTING

1. unable to access the web page through port 80.



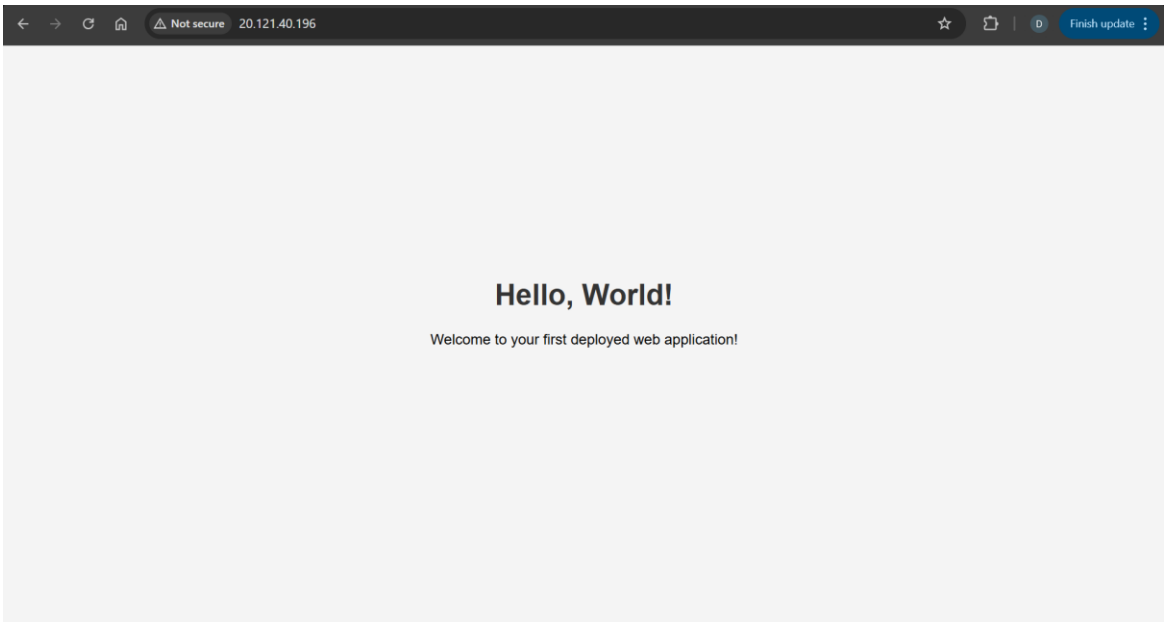
A **504 Gateway Timeout** typically means your browser couldn't get a timely response from the containerized app. First, make sure the container is running using docker ps on your Dev-vm. Confirm the app inside the container is actively listening on **port 80** or the expected port. Check your docker run command includes correct port mapping, like -p 80:80. Ensure that **NSG inbound rules** for the VM allow traffic on port 80. Also verify the VM's public IP is being used correctly in the browser (e.g., http://<public-ip>). If all checks are correct and the issue persists, try restarting the container and VM.



By using the internet, I found the cause of this error. The inbound rule I applied to the NSG in Azure was not accepted by the VM's firewall, so when I tried to run the container on port 80, I got the error. I then visited the Azure portal and added the same inbound rule for port 80 that I had previously added to the NSG.

What is 504 Gateway Timeout?

A 504 Gateway Timeout error means a server acting as a gateway or proxy didn't receive a timely response from another server it needed to complete a request. It's a server-side issue, indicating a problem with the server's communication with another server. This often happens when a server is overloaded, down, or experiencing network connectivity issues.



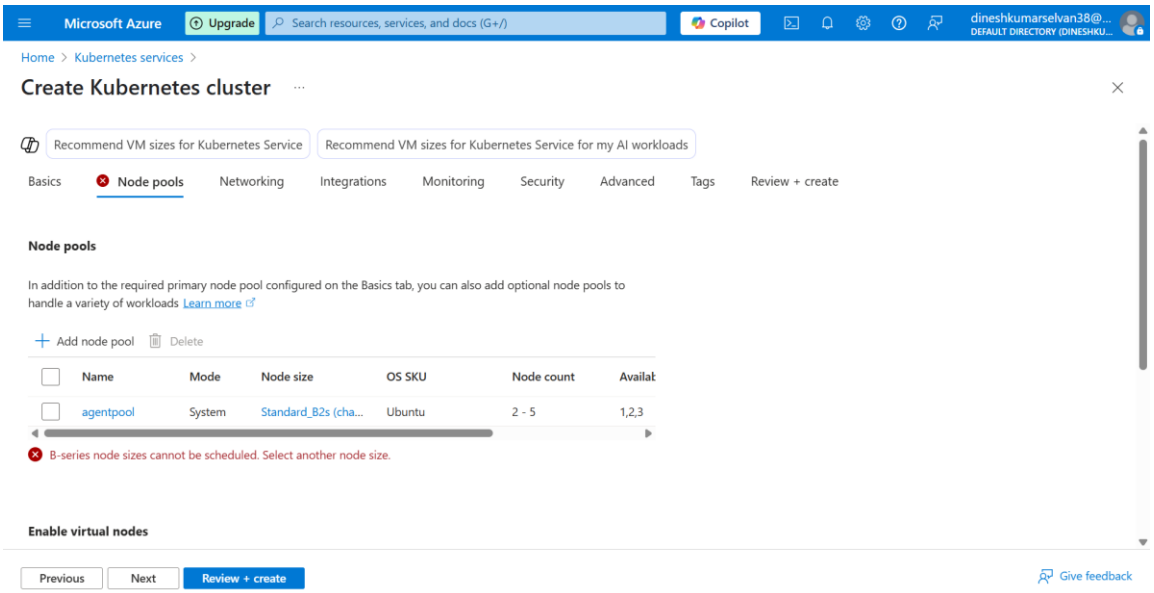
After adding the inbound rule for the VM in VM networking settings.

2. try to use b series size in AKS node.

System Node Pool Requirements: Azure AKS has specific requirements for system node pools, which are responsible for running critical Kubernetes components like CoreDNS, metrics-server, and other essential services. To ensure the reliability and stability of the AKS cluster, these system node pools have minimum resource requirements.

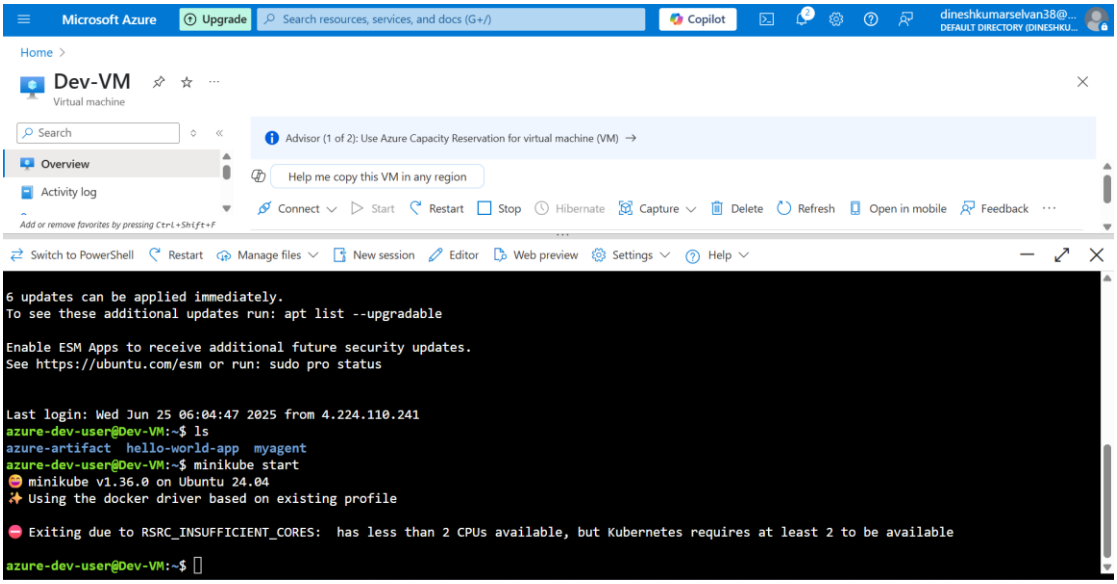
- **Minimum vCPU and Memory:** System node pools generally require a VM SKU with at least **4 vCPUs and 4GB memory**. Many of the smaller B-series VMs might not meet these minimums.
- **Consistent Performance:** The burstable nature of B-series VMs can lead to performance throttling if CPU credits are exhausted. This inconsistency is undesirable for core Kubernetes services, which need predictable and reliable performance to maintain cluster health.

So, I use d series size which is Standard_D2ps_v6 for AKS node.

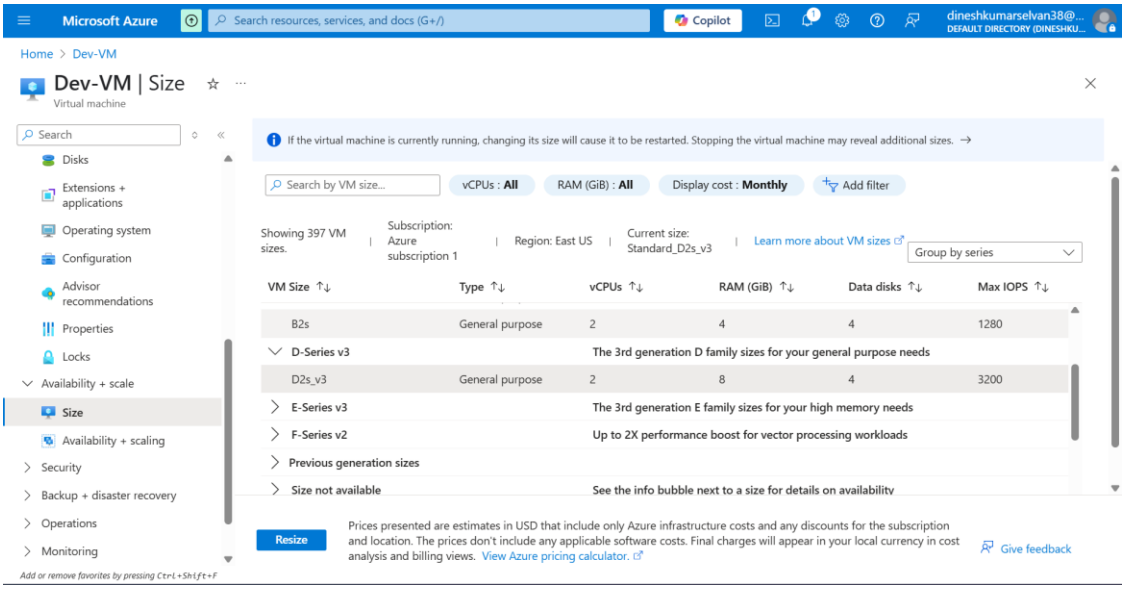


---

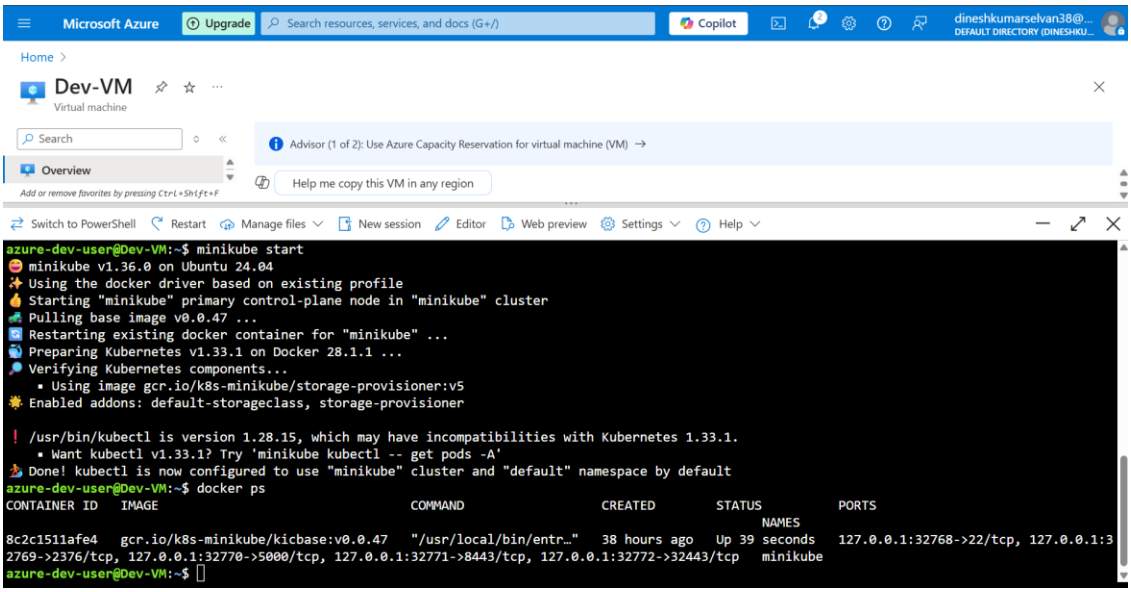
3. can't be able to start the minikube:



As per the documentation I use b1s size for the VM. Due to this I can't able to start the minikube.



In the Azure Portal, go to **Virtual Machines** and select your VM. Click on **Size** under the **Settings** section in the left menu. Browse available sizes and choose **Standard_D2s_v3** from the list. Click **Resize**, then confirm the operation to change the VM size. After resizing, restart the VM if needed to apply the changes.



After resizing the VM to a larger size like **Standard_D2s_v3**, resource constraints are resolved. Log into your VM and run minikube start to initialize the Kubernetes cluster. Minikube begins setting up the local cluster using the updated resources. Once completed, check cluster status with kubectl get nodes to confirm it's running. Your Minikube environment is now active and ready for local Kubernetes testing.
