800 XP

Deploy a cloud-native .NET microservice automatically with GitHub Actions and Azure Pipelines

34 min • Module7 Units

Intermediate | Developer | DevOps Engineer | Solution Architect | ASP.NET Core | Azure | Azure Container Registry | Azure Kubernetes Service (AKS) | GitHub

Use CI/CD pipelines to build a container image and deploy it to Azure Kubernetes Service (AKS).

Learning objectives

This module guides you through the following steps:

- Authenticate GitHub Actions to a container registry.
- Securely store sensitive information that GitHub Actions uses.
- Implement an action to build the container image for a microservice.
- Modify and commit the microservice code to trigger a build.
- Implement an action to deploy the updated container to an Azure Kubernetes Service (AKS) cluster.
- Revert the microservice to the previous deployment.
- Implement Azure Pipelines to build and deploy a microservice to Azure Kubernetes Service (AKS) cluster.

Start > ⊕ Add

Prerequisites

- Conceptual knowledge of DevOps practices.
- Conceptual knowledge of containers, Docker, and AKS.
- Access to an Azure subscription with **Owner** permissions.
- Access to a GitHub account.
- Access to an Azure DevOps organization.

- Ability to run *development containers* in Visual Studio Code or GitHub Codespaces, including Docker and the necessary Visual Studio Code extensions installed.

This module is part of these learning paths

Create cloud-native apps and services with .NET and ASP.NET Core

Introduction
1 min

Manually deploy your cloud-native app to Azure Kubernetes Service
8 min

Exercise - Create a GitHub action to build a container image
6 min

Exercise - Create a GitHub action to deploy to AKS
10 min

Explore different CI/CD approaches
2 min

Exercise - Create an Azure DevOps pipeline to deploy your cloud-native app
5 min

Summary
2 min

100 XP

Introduction

1 minute

Imagine that you work as a software engineer for an online outdoor clothing retailer. You're responsible for deploying and updating the retailer's online storefront, a cloud-native, microservices-based .NET app.

To fulfill project requirements and enhance your team's agile development practices, you decide to compare continuous integration and continuous deployment (CI/CD) through GitHub Actions and Azure Pipelines. CI/CD pipelines use a series of automated steps to compile and deploy apps from build through all environments.

Because the current web has a microservices architecture, and each microservice deploys independently, you start by setting up CI/CD for a single service.

The .NET web API, named the **product service**, supports all the backend catalog features of the website. In this module, you'll implement a CI/CD pipeline for the product service.

This module guides you through the following steps:

- Authenticate GitHub Actions to a container registry.
- Securely store sensitive information that GitHub Actions uses.
- Implement an action to build the container image for a microservice.
- Modify and commit the microservice code to trigger a build.
- Implement an action to deploy the updated container to an Azure Kubernetes Service (AKS) cluster.
- Modify and commit a Helm chart to trigger the deployment.
- Revert the microservice to the previous deployment.

You use your own Azure subscription to deploy the resources in this module. If you don't have an Azure subscription, create a free account before you begin.

Important

To avoid unnecessary charges in your Azure subscription, be sure to delete your Azure resources when you're done with this module.

Prerequisites

- Conceptual knowledge of DevOps practices.
- Conceptual knowledge of containers, Docker, and AKS.
- Access to an Azure subscription with **Owner** permissions.
- Access to a GitHub account.
- Ability to run *development containers* in Visual Studio Code or GitHub Codespaces, set up as described in the following section.

Development container

This module includes configuration files that define a development container, or *dev container*. Using a dev container ensures a standardized environment that's preconfigured with the required tools.

The dev container can run in either of two environments. Before you begin, follow the steps in one of the following links to set up your environment, including installing Docker and the necessary Visual Studio Code extensions.

- Visual Studio Code and a supported Docker environment on your local machine.
- GitHub Codespaces (costs may apply).

✓ 100 XP

Manually deploy your cloud-native app to Azure Kubernetes Service

8 minutes

Before you can automate your website deployments, you need to deploy the existing eShop app manually to Azure Kubernetes Service (AKS). You create the Azure resources and deploy the app to AKS using Azure CLI commands and bash scripts. Finally, you create an Azure Active Directory (Azure AD) service principal to allow GitHub Actions to deploy to AKS and Azure Container Registry.

The commands create the following resources to deploy an updated version of the eShop app.

- Provision an Azure Container Registry (ACR) and then push images into the registry.
- Provision an AKS cluster and then deploy the containers into the cluster.
- Test the deployment.
- Create service principals to allow GitHub Actions to deploy to AKS and Azure Container Registry.

> ⓘ **Important**
>
> Make sure you've completed the **prerequisites** before you begin.

Open the development environment

You can choose to use a GitHub codespace that hosts the exercise, or complete the exercise locally in Visual Studio Code.

GitHub Codespaces Setup

Fork the https://github.com/MicrosoftDocs/mslearn-dotnet-cloudnative-devops repository to your own GitHub account. Then on your new fork:

1. Select **Code**.
2. Select the **Codespaces** tab.

3. Select the + icon to create your codespace.

GitHub takes several minutes to create and configure the codespace. When the process completes, you see the code files for the exercise.

Optional: Visual Studio Code Setup

To use **Visual Studio Code**, fork the https://github.com/MicrosoftDocs/mslearn-dotnet-cloudnative-devops repository to your own GitHub account and clone it locally. Then:

1. Install any system requiements to run Dev Container in Visual Studio Code.
2. Make sure Docker is running.
3. In a new Visual Studio Code window open the folder of the cloned repository
4. Press `Ctrl`+`Shift`+`P` to open the command palette.
5. Search: >**Dev Containers: Rebuild and Reopen in Container**
6. Visual Studio Code creates your development container locally.

Build containers

1. In the terminal pane, run this dotnet CLI command:

   .NET CLI

   ```
   dotnet publish /p:PublishProfile=DefaultContainer
   ```

Create the Azure resources

1. In the terminal pane, sign in to Azure with this Azure CLI command:

   Azure CLI

   ```
   az login --use-device-code
   ```

2. View the selected Azure subscription.

   Azure CLI

   ```
   az account show -o table
   ```

   If the wrong subscription is selected, use the az account set command to select the correct one.

3. Run the following Azure CLI command to get a list of Azure regions and the Name associated with it:

   Azure CLI

   ```
   az account list-locations -o table
   ```

   Locate a region closest to you and use it in the next step by replacing `[Closest Azure region]`

4. Run these bash statements:

   Bash

   ```
   export LOCATION=[Closest Azure region]
   export RESOURCE_GROUP=rg-eshop
   export CLUSTER_NAME=aks-eshop
   export ACR_NAME=acseshop$SRANDOM
   ```

   The previous commands create environment variables that you'll use in the next Azure CLI commands. You need to change the **LOCATION** to an Azure region close to you; for example, **eastus**. If you'd like a different name for your resource group, AKS cluster, or ACR, change those values. To view your new repositories in the Azure portal, assign yourself as **App Compliance Automation Administrator** in the **Access control (IAM)** of the container registry.

5. Run these Azure CLI commands:

   Azure CLI

   ```
   az group create --name $RESOURCE_GROUP --location $LOCATION
   az acr create --resource-group $RESOURCE_GROUP --name $ACR_NAME --sku
   Basic
   az acr login --name $ACR_NAME
   ```

   If you receive an authentication error when `az acr login --name $ACR_Name` is run, you need to turn on **Admin user** in the newly created **container register** in Azure under **Settings - Access Keys**. Azure prompts you to enter these credentials to continue. You could also need to authenticate again with `az login --use-device-code`.

   These commands create a resource group to contain the Azure resources, an ACR for your images, and then logins into the ACR. It can take a few minutes until you see this output:

```Console
    ...
  },
  "status": null,
  "systemData": {
    "createdAt": "2023-10-19T09:11:51.389157+00:00",
    "createdBy": "",
    "createdByType": "User",
    "lastModifiedAt": "2023-10-19T09:11:51.389157+00:00",
    "lastModifiedBy": "",
    "lastModifiedByType": "User"
  },
  "tags": {},
  "type": "Microsoft.ContainerRegistry/registries",
  "zoneRedundancy": "Disabled"
}
Login Succeeded
```

6. To tag your images and push them to the ACR you created, run these commands:

```Bash
docker tag store $ACR_NAME.azurecr.io/storeimage:v1
docker tag products $ACR_NAME.azurecr.io/productservice:v1

docker push $ACR_NAME.azurecr.io/storeimage:v1
docker push $ACR_NAME.azurecr.io/productservice:v1
```

You can check pushing the images completes successfully with this command:

```Bash
az acr repository list --name $ACR_NAME --output table
```

7. Create your AKS and connect it to the ACR with these commands:

```Bash
az aks create --resource-group $RESOURCE_GROUP --name $CLUSTER_NAME --
node-count 1 --generate-ssh-keys --node-vm-size Standard_B2s --network-
plugin azure --attach-acr $ACR_NAME

az aks get-credentials --name $CLUSTER_NAME --resource-group
$RESOURCE_GROUP
```

The above commands create a single node AKS cluster, connect it to the ACR, and then connect your local machine to the AKS cluster. The above commands can take a few minutes to complete.

8. Check that the new AKS can pull images from the ACR with this command:

Bash

```
az aks check-acr --acr $ACR_NAME.azurecr.io --name $CLUSTER_NAME --
resource-group $RESOURCE_GROUP
```

You should see similar output to the following messages:

Console

```
[2023-10-19T13:33:09Z] Loading azure.json file from /etc/kubernetes/
azure.json
[2023-10-19T13:33:09Z] Checking managed identity...
[2023-10-19T13:33:09Z] Cluster cloud name: AzurePublicCloud
[2023-10-19T13:33:09Z] Kubelet managed identity client ID:
71588cd0-9229-4914-9c8e-1dc229d775c8
[2023-10-19T13:33:09Z] Validating managed identity existance: SUCCEEDED
[2023-10-19T13:33:09Z] Validating image pull permission: SUCCEEDED
[2023-10-19T13:33:09Z]
Your cluster can pull images from acseshop1251599299.azurecr.io!
```

You can now run **kubectl** commands against your new AKS cluster. Copy the full ACR URL from the output; for example, above the URL is **acseshop1251599299**.

9. Check the status of your AKS cluster:

Bash

```
kubectl get nodes -A
```

You should see similar output to the following messages:

Console

```
NAME                                STATUS   ROLES   AGE     VERSION
aks-nodepool1-37200563-vmss000000   Ready    agent   3h44m   v1.26.6
```

Configure the Kubernetes deployment manifest

Now the eShop images are in the ACR you can update the AKS deployment manifest to use these new images.

1. In Visual Studio Code, from the EXPLORER panel, select the **deployment.yml** file in the root of the project.

2. Replace on line 17:

   yml

   ```yml
   - image: [replace with your ACR name].azurecr.io/storeimage:v1
   ```

   Paste the copied ACR name from the previous step – the line should look similar to the following yaml:

   yml

   ```yml
   - image: acseshop1251599299.azurecr.io/storeimage:v1
   ```

3. Repeat these steps for line 65:

   yml

   ```yml
   - image: [replace with your ACR name].azurecr.io/productservice:v1
   ```

   Save the file with `CTRL`+`S`.

4. In the terminal pane, deploy an NGINX ingress controller with the following kubernetes command:

   Bash

   ```bash
   kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.9.3/deploy/static/provider/cloud/deploy.yaml
   ```

   The above `kubectl` command adds services and components to allow ingress into your AKS cluster. Check that the ingress is ready run using the following kubernetes command:

   Bash

   ```bash
   kubectl get services --namespace ingress-nginx
   ```

You should see similar output to the following messages:

Console

```
NAME                                TYPE           CLUSTER-IP
EXTERNAL-IP    PORT(S)                      AGE
ingress-nginx-controller            LoadBalancer   10.0.135.51
20.26.154.64   80:32115/TCP,443:32254/TCP   58s
ingress-nginx-controller-admission  ClusterIP      10.0.137.137   <none>
443/TCP                     58s
```

5. Deploy the eShop app with this command:

Bash

```
kubectl apply -f deployment.yml
```

The `kubectl` apply command deploys the eShop app, a front-end Blazor web app and back-end REST API product service, and an ingress rule to route traffic to the correct services to your AKS cluster. Rerun this command if you receive any error on deployments.

You should see similar output to the following messages:

Console

```
deployment.apps/storeimage created
service/eshop-website created
deployment.apps/productservice created
service/eshop-backend created
ingress.networking.k8s.io/eshop-ingress created
```

6. Check the two microservices are deployed with this command:

Bash

```
kubectl get pods -A
```

You should see similar output to the following messages:

Console

```
NAMESPACE       NAME                                          READY   STATUS
RESTARTS   AGE
default         productservice-7569b8c64-vfbfz                1/1
```

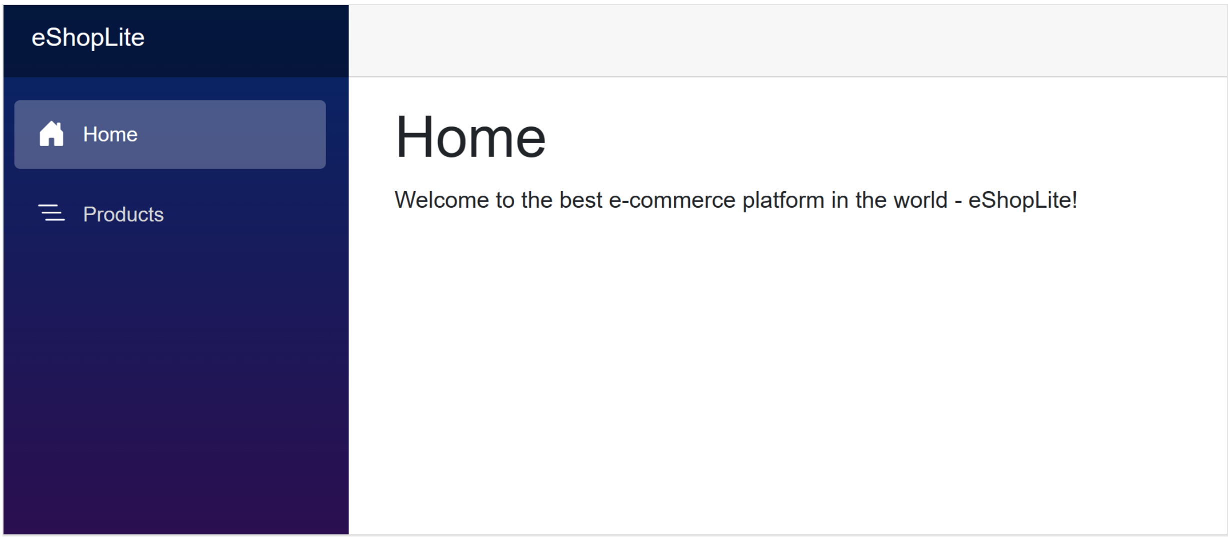
```
Running      0          3m56s
default         storeimage-6c7c999d7c-zsnxd                1/1
Running      0          3m56s
ingress-nginx   ingress-nginx-admission-create-szb8l       0/1
Completed    0          4m4s
ingress-nginx   ingress-nginx-admission-patch-czdbv        0/1
Completed    0          4m4s
ingress-nginx   ingress-nginx-controller-58bf5bf7dc-nwtsr   1/1
Running      0          4m4s
```

7. View the deployed eShop with this command:

Bash

```bash
echo "http://$(kubectl get services --namespace ingress-nginx ingress-
nginx-controller --output jsonpath='{.sta-
tus.loadBalancer.ingress[0].ip}')"
```

The above command returns the external IP address for the web app. Hold CTRL and click the link to open the app in a new tab.



Home

Welcome to the best e-commerce platform in the world - eShopLite!

Create a service principal for deploying from GitHub

GitHub Actions can publish container images to an Azure Container Registry. The GitHub runner therefore must have permissions to connect to Azure. The following steps create an Azure AD service principal to act as the GitHub Actions identity inside Azure.

1. To save your Subscription ID in an environment variable, run the following command in the terminal:

Azure CLI

```
export SUBS=$(az account show --query 'id' --output tsv)
```

2. To create an Azure AD service principal to allow access from GitHub, run the following command:

Azure CLI

```
az ad sp create-for-rbac --name "eShop" --role contributor --scopes /sub-
scriptions/$SUBS/resourceGroups/$RESOURCE_GROUP --json-auth
```

A variation of the following output appears:

Console

```
Creating 'Contributor' role assignment under scope '/
subscriptions/00000000-0000-0000-0000-000000000000'

The output includes credentials that you must protect. Be sure that you do
not include these credentials in your code or check the credentials into
your source control. For more information, see https://aka.ms/azadsp-cli
 {
  "clientId": "00000000-0000-0000-0000-000000000000",
  "clientSecret": "abc1A~abc123ABC123abc123ABC123abc123ABC1",
  "subscriptionId": "00000000-0000-0000-0000-000000000000",
  "tenantId": "00000000-0000-0000-0000-000000000000",
  "activeDirectoryEndpointUrl": "https://login.microsoftonline.com",
  "resourceManagerEndpointUrl": "https://management.azure.com/",
  "activeDirectoryGraphResourceId": "https://graph.windows.net/",
  "sqlManagementEndpointUrl": "https://management.core.windows.net:8443/",
  "galleryEndpointUrl": "https://gallery.azure.com/",
  "managementEndpointUrl": "https://management.core.windows.net/"
}
```

3. Copy the JSON output and brackets to use in the next step.

Create the GitHub secrets

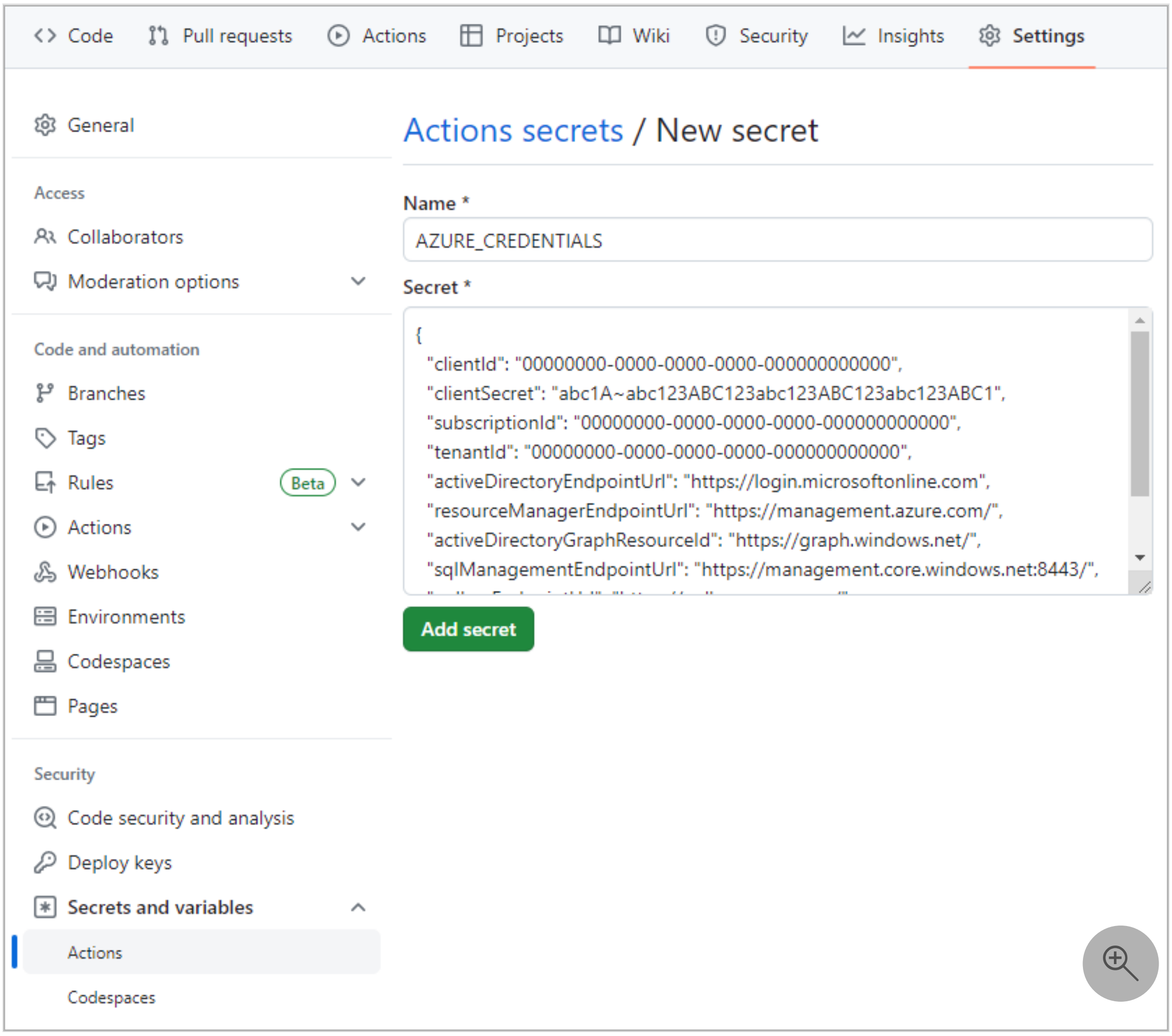
The GitHub Actions runner uses credentials to interact with Container Registry and AKS. The service principal and the credentials for the container registry are sensitive information. It's best to store sensitive information as encrypted *secrets* in a secure location. GitHub provides a built-in location to store secrets and other variables.

Complete the following steps to securely store the sensitive information as environment variables in your repository. Repository administrators should manage the secrets that the

GitHub Actions runner can access.

1. In your forked GitHub repository, go to **Settings** > **Secrets and variables** > **Actions**.
2. On the **Actions secrets and variables** page, select **New repository secret**.
3. On the **New secret** page, under **Name**, enter *AZURE_CREDENTIALS*, and under **Secret**, enter the JSON output you copied from the terminal.

   The settings should look similar to the following screenshot:



Actions secrets / New secret

Name *

AZURE_CREDENTIALS

Secret *

```
{
  "clientId": "00000000-0000-0000-0000-000000000000",
  "clientSecret": "abc1A~abc123ABC123abc123ABC123abc123ABC1",
  "subscriptionId": "00000000-0000-0000-0000-000000000000",
  "tenantId": "00000000-0000-0000-0000-000000000000",
  "activeDirectoryEndpointUrl": "https://login.microsoftonline.com",
  "resourceManagerEndpointUrl": "https://management.azure.com/",
  "activeDirectoryGraphResourceId": "https://graph.windows.net/",
  "sqlManagementEndpointUrl": "https://management.core.windows.net:8443/",
```

4. Select **Add secret**.

You'll use this GitHub secret in the next section to create a GitHub action to build the container image.

100 XP

Exercise - Create a GitHub action to build a container image

6 minutes

In this unit, you'll complete the following tasks:

- Create a GitHub action to implement a build pipeline.
- Modify the coupon service code to trigger the build workflow.
- Monitor the build workflow's progress in real time.

Create the build action

The YAML code in this procedure defines a GitHub action that:

- Triggers when a commit is pushed to the coupon service's source code or unit tests in the `main` branch.
- Defines step-specific environment variables.
- Has one *job*, or set of steps that execute on the same workflow runner, named `Build and push image to ACR`.

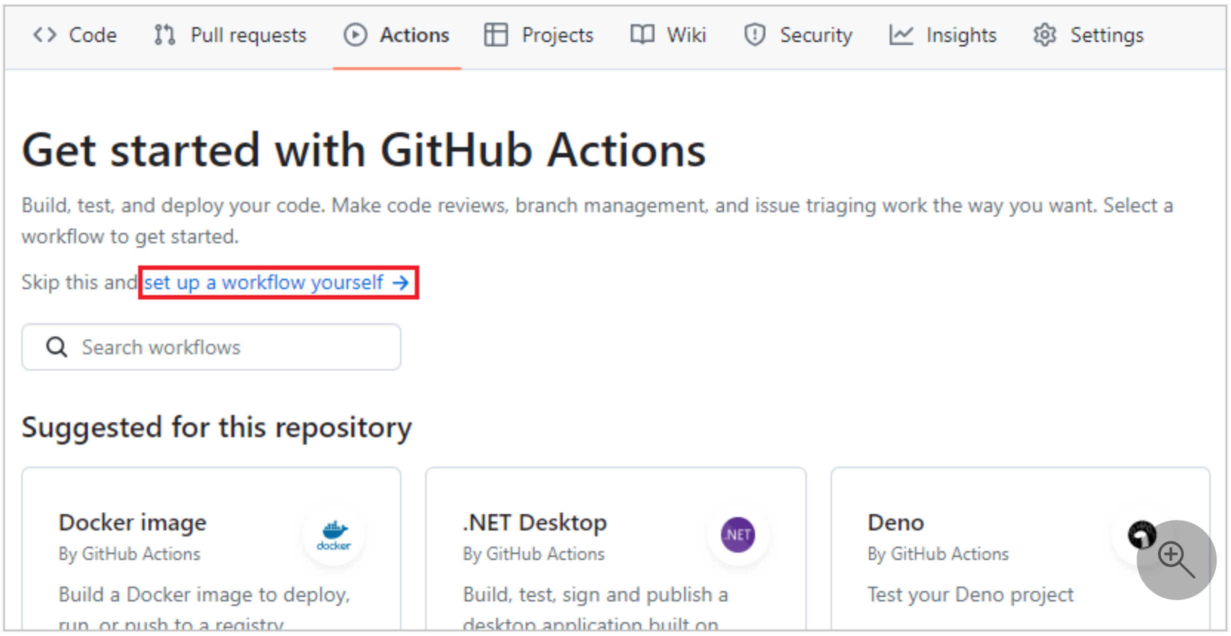
Important

Trigger conditions and other artifacts of GitHub Actions or workflows depend on the apps and environments. For ease of understanding, details are kept simple in this example. Both the build and the deploy workflows are scoped to product service changes because all the microservices are kept under a single repository. In an actual production scenario, each microservice would be kept in their own separate repository.

Complete the following steps to create the GitHub Actions build action:

1. Go to your forked repository in GitHub, select the **Actions** tab.

2. On the **Get started with GitHub Actions** page, select the **set up a workflow yourself** link.



Get started with GitHub Actions

Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started.

Skip this and set up a workflow yourself →

Suggested for this repository

Docker image
By GitHub Actions
Build a Docker image to deploy, run, or push to a registry

.NET Desktop
By GitHub Actions
Build, test, sign and publish a desktop application built on

Deno
By GitHub Actions
Test your Deno project

3. On the next page, paste the following YAML code into the editor.

YAML

```yaml
name: Build and deploy an app to AKS

on:
  push:
    branches: ["main"]
  workflow_dispatch:

env:
  # Local environment variables used later in the workflow
  AZURE_CONTAINER_REGISTRY: 'name of your Azure Container Registry'
  CONTAINER_NAME: 'productservice'
  RESOURCE_GROUP: 'rg-eshop'
  CLUSTER_NAME: 'aks-eshop'
  DEPLOYMENT_MANIFEST_PATH: './product.yml'
  DOCKER_PATH: './DockerfileProducts'

jobs:
  buildImage:
    permissions:
      contents: read
      id-token: write
    runs-on: ubuntu-latest
    steps:
      # Checks out the repository this file is in
      - uses: actions/checkout@v3

      # Logs in with your Azure credentials stored in GitHub secrets
      - name: Azure login
        uses: azure/login@v1.4.6
        with:
          creds: '${{ secrets.AZURE_CREDENTIALS }}'
    # Builds and pushes an image up to your Azure Container Registry
    - name: Build and push image to ACR
      run: |
        az acr build --file ${{ env.DOCKER_PATH }} --image ${{
env.AZURE_CONTAINER_REGISTRY }}.azurecr.io/${{ env.CONTAINER_NAME }}:${{
github.sha }} --registry ${{ env.AZURE_CONTAINER_REGISTRY }} -g ${{
env.RESOURCE_GROUP }} .
```

Replace the **name of your Azure Container Registry** with the ACR name you created in the previous unit; for example, **acseshop186748394**.

4. Replace the default workflow **main.yml** file name with *azure-kubernetes-service.yml*, and then select **Commit changes**.

5. On the **Commit changes** screen, select **Commit directly to the main branch** and then select **Commit changes**.

   You've finished creating the build workflow for your CI/CD pipeline.

6. In your terminal, run this command to view the current versions of the product service stored in the ACR:

   Bash

   ```
   az acr repository show-tags -n AZURE_CONTAINER_REGISTRY --repository prod-
   uctservice --orderby time_desc --output table
   ```

   Replacing the **AZURE_CONTAINER_REGISTRY** with the name of your ACR, you should see output similar to the following:

   Bash

   ```
   Result
   ----------------------------------------
   v1
   ```

Trigger the build

The build workflow triggers automatically as soon as you commit the workflow file. You can also trigger the build manually.

1. In your repository, select the **Actions** tab.

2. On the left, under **All workflows**, select the **Build and deploy an app to AKS** workflow, and then select **Run workflow**.

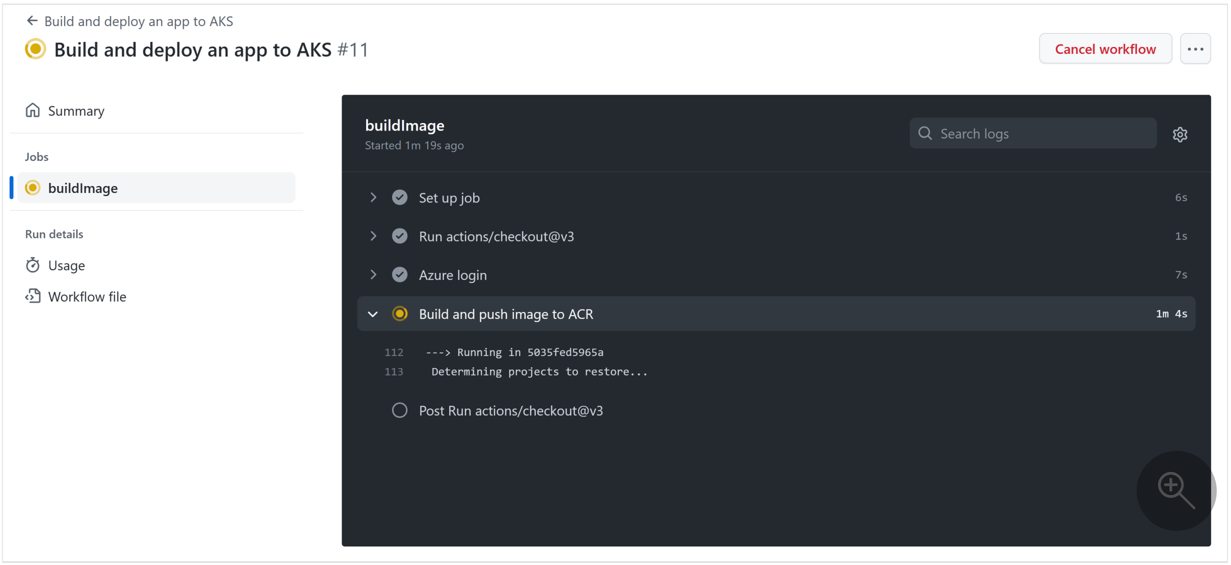
Monitor the build

To view the real-time progress of the build:

1. In your repository, select the **Actions** tab.

2. Select the most recent workflow run listed for the **Build and deploy an app to AKS** workflow. The name of the run is the commit message you used in the previous step.

3. Select the **buildImage** job.



← Build and deploy an app to AKS
Build and deploy an app to AKS #11
Cancel workflow
Summary
Jobs
buildImage
Run details
Usage
Workflow file
buildImage
Started 1m 19s ago
Search logs
Set up job 6s
Run actions/checkout@v3 1s
Azure login 7s
Build and push image to ACR 1m 4s
Post Run actions/checkout@v3

```
---> Running in 5035fed5965a
 Determining projects to restore...
```

4. If you wait a few minutes, the steps in the job should complete successfully.

5. In your terminal, run this command again to view the versions of the product service stored in the ACR.

Bash

```bash
az acr repository show-tags -n AZURE_CONTAINER_REGISTRY --repository prod-
uctservice --orderby time_desc --output table
```

You should see output similar to the following, that shows a new image version has been added to the ACR from the GitHub workflow:

Bash

```
Result
----------------------------------------
```

```
8c75edb7a349ec570bd4eac397015bc3c547186e
v1
```

200 XP

Exercise - Create a GitHub action to deploy to AKS

10 minutes

In this exercise, you'll complete the following tasks:

- Enhance the existing GitHub action to include a deployment job.
- Verify that the changes deploy to the Azure Kubernetes Service (AKS) cluster.
- Roll back the deployment.

Update the Kubernetes manifest for the product service

To deploy new versions of the eShop product service, edit the **product.yml** file to point at the Azure Container Registry (ACR) you used in the previous unit.

1. In your forked repository, select the **code tab**, then select the *product.yml* file.

2. To edit the file, select the edit icon (pencil).

3. Edit the line:

   yml

   ```yml
   containers:
     - image: [replace with your ACR name].azurecr.io/productservice:latest
   ```

   Replace the `[replace with your ACR name]` with the name of your ACR; for example, **acseshop186748394**.

4. In the top right, select **Commit changes...** then, in the dialog, select **Commit changes**.

Create the deployment action

The YAML code adds a GitHub step that:

Has one step that deploys new images. Here's the steps in an `ubuntu-latest` runner:

1. Checks out the repository this file is in.
2. **Azure Login** signs in to Azure with the service principal credentials.
3. **Set up kubelogin for non-interactive login** configures the kubeconfig file for Azure authentication.
4. **Get K8s context** set context sets the Azure Kubernetes Service (AKS) credentials in the runner's *.kube/config* file.
5. **Deploys application** deploys the application to AKS, using the image built in the previous step and the Kubernetes manifest file you edited earlier.

Complete the following steps to create a GitHub action that deploys the coupon service:

1. In your forked repository, on the **code tab**, select the **.github/workflows** tab.

2. Select *azure-kubernetes-service.yml*.

3. To edit the file, select the edit icon (pencil).

4. At the bottom of the file, paste the following YAML code into the editor:

YAML

```yaml

  deploy:
    permissions:
      actions: read
      contents: read
      id-token: write
    runs-on: ubuntu-latest
    needs: [buildImage]
    steps:
      # Checks out the repository this file is in
      - uses: actions/checkout@v3

      # Logs in with your Azure credentials
      - name: Azure login
        uses: azure/login@v1.4.6
        with:
          creds: '${{ secrets.AZURE_CREDENTIALS }}'

      # Use kubelogin to configure your kubeconfig for Azure auth
      - name: Set up kubelogin for non-interactive login
        uses: azure/use-kubelogin@v1
        with:
          kubelogin-version: 'v0.0.25'

      # Retrieves your Azure Kubernetes Service cluster's kubeconfig file
      - name: Get K8s context
        uses: azure/aks-set-context@v3
        with:
          resource-group: ${{ env.RESOURCE_GROUP }}
          cluster-name: ${{ env.CLUSTER_NAME }}
          admin: 'false'
          use-kubelogin: 'true'

      # Deploys application based on given manifest file
      - name: Deploys application
        uses: Azure/k8s-deploy@v4
        with:
          action: deploy
          manifests: ${{ env.DEPLOYMENT_MANIFEST_PATH }}
          images: |
            ${{ env.AZURE_CONTAINER_REGISTRY }}.azurecr.io/${{
env.CONTAINER_NAME }}:${{ github.sha }}
          pull-images: false
```

5. In the top right, select **Commit changes...**, then in the dialog select **Commit changes**.

Trigger a deployment

Updating the **azure-kubernetes-service.yml** file and committing the changes automatically triggers another deployment. Now see how making a code change triggers another deployment.

You have a new product your marketing team would like to add to the catalog.

1. In your forked repository, on the **code tab**, select the **Products** folder.

2. Select the **Data** folder.

3. Select the **ProductDataContext.c** file.

4. To edit the file, select the edit icon (pencil).

5. At the bottom of the file, add a new product to the **products** array:

```C#
new Product {  Name = "Camping Tent 2", Description = "This updated tent
is improved and cheaper, perfect for your next trip.", Price = 79.99m,
ImageUrl = "product9.png" },
```

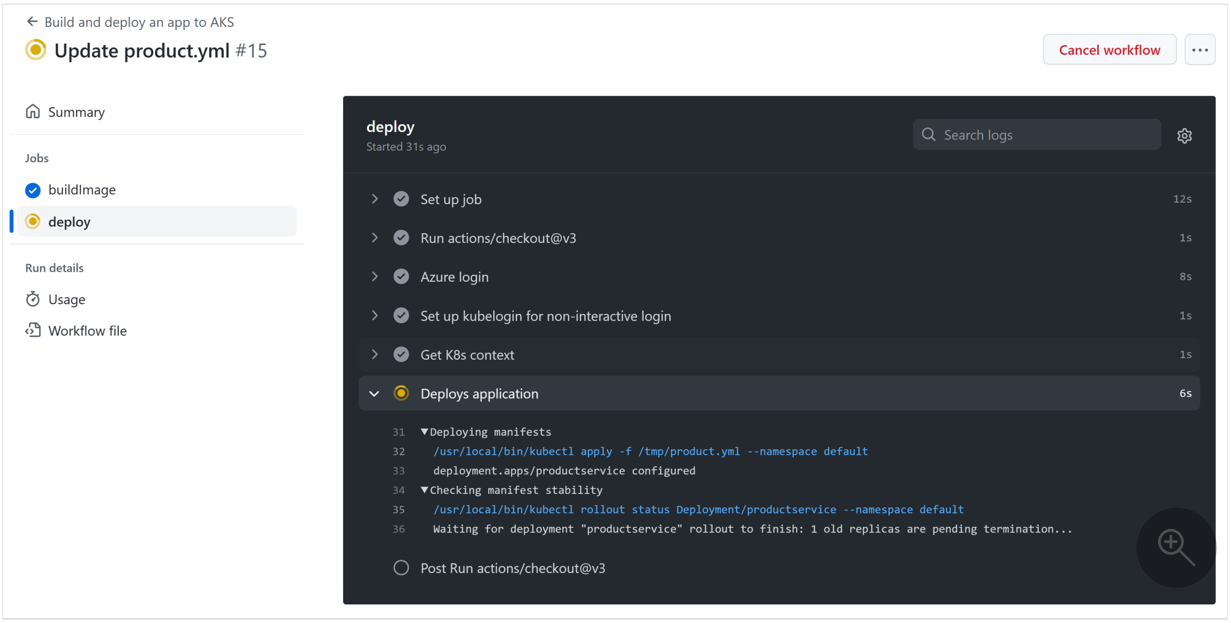
6. In the top right, select **Commit changes...** then, in the dialog, select **Commit changes**.

Monitor the deployment

1. To monitor the deployment's progress, select the **Actions** tab.

2. Select the most recent workflow run listed for the **Build and deploy an app to AKS** workflow. The run's name is the commit message you used in the previous step.

3. Select the **deploy** job to see details for this workflow run.



```
▼Deploying manifests
  /usr/local/bin/kubectl apply -f /tmp/product.yml --namespace default
  deployment.apps/productservice configured
▼Checking manifest stability
  /usr/local/bin/kubectl rollout status Deployment/productservice --namespace default
  Waiting for deployment "productservice" rollout to finish: 1 old replicas are pending termination...
```

4. In the terminal, run the following command to monitor the coupon service pods in your AKS cluster. The `--selector` flag filters the list to only pods for the coupon service, and the `--watch` flag instructs `kubectl` to watch for changes.

Bash

```bash
kubectl get pods --selector=app=productservice --watch
```

During the deployment, a variation of the following output appears:

Console

```
NAME                             READY   STATUS    RESTARTS   AGE
productservice-7979d4c47-xlcrr   1/1     Running   0          17m
productservice-ff98b6d8d-7wmsh   0/1     Pending   0          0s
productservice-ff98b6d8d-7wmsh   0/1     Pending   0          0s
productservice-ff98b6d8d-7wmsh   0/1     ContainerCreating   0          0s
productservice-ff98b6d8d-7wmsh   1/1     Running             0          4s
productservice-7979d4c47-xlcrr   1/1     Terminating         0          
19m
```

In the preceding output, notice that a new **productservice** pod is created. When the new pod is ready, the old one is terminated. This process makes the transition to the new version as smooth as possible.

Verify the app

Complete the following steps to verify that your app still works:

- View the deployed eShop by running this command in the terminal:

  Bash

  ```bash
  echo "http://$(kubectl get services --namespace ingress-nginx ingress-nginx-controller --output jsonpath='{.status.loadBalancer.ingress[0].ip}')"
  ```

  The above command returns the external IP address for the web app. Hold CTRL and select the link to open the app in a new tab.

Go to the products page to view the new tent listed at the bottom of the page.

Roll back the deployment

One common mitigation for production issues is to revert to a known good deployment. Kubernetes maintains a deployment history that you can use to roll back to a previous version of your app.

In your terminal, run this command to remove the new tent you just added to the website:

Bash

```bash
kubectl rollout undo deployment/productservice
```

You should see this console message:

Console

```console
deployment.apps/productservice rolled back
```

Refresh the products page in your browser, and the new tent should no longer be listed.

> **Note**
>
> In a real-life scenario, you deploy the build's artifacts to multiple environments. For

example, you might have development, testing, and staging environments. You can trigger deployment workflows by events like merging PRs. You can add quality or approval gates, such as a stakeholder's PR approval, to prevent unexpected deployments to production.

Check your knowledge

1. **What's the best place to store sensitive information, such as credentials, for GitHub Actions?** *

- ◯ *appsettings.json*
- ◯ *web.config*
- ◯ Azure Key Vault
- ◯ GitHub secrets

2. **What's the purpose of creating an Azure Active Directory service principal for GitHub Actions to use?** *

- ◯ The service principal is for storing application logs.
- ◯ The service principal is the name of the Azure Kubernetes Service (AKS) resource.
- ◯ The service principal is the GitHub Actions identity for doing tasks in Azure.
- ◯ The service principal is the web URL of the app.

3. **During upgrade deployment, why does AKS create a new container while the old one is still running?** *

- ◯ To avoid downtime.
- ◯ To scale up.
- ◯ For redundancy.
- ◯ To scale out.

Check your answers

100 XP

Explore different CI/CD approaches

2 minutes

So far, you've seen two approaches to support CI/CD for your cloud-native app. You manually deployed the app to AKS, and you used GitHub Actions to build and deploy the app. Microsoft supports a third approach, Azure Pipelines. Both the automated approaches are valid; you choose the one that best fits your needs.

How are GitHub actions and Azure Pipelines different?

Let's start by looking at how these two approaches are the same. GitHub Actions and Azure Pipelines are both CI/CD tools. They both support:

- Building and deploying your app.
- YAML files to define the steps to build and deploy your app.
- Triggers to start the build and deploy process.
- Monitoring the build and deploy process.
- Rolling back a deployment.

The different levels of support for features are where the two approaches vary. Let's look at these differences.

Expand table

GitHub Actions	**Azure Pipelines**
Free for public repositories	Free for open source projects
Free for up to **2000** minutes per month for private repositories	Free for up to **1800** minutes per month for private repositories
Limited to **20** concurrent jobs	Limited to **10** concurrent jobs

Azure Pipelines has an advantage over GitHub Actions as it supports many different source repositories. Azure Pipelines supports GitHub, GitHub Enterprise Server, Bitbucket Cloud, Azure Repos Git and TFVC, Subversion, and External Git. GitHub Actions only support GitHub.

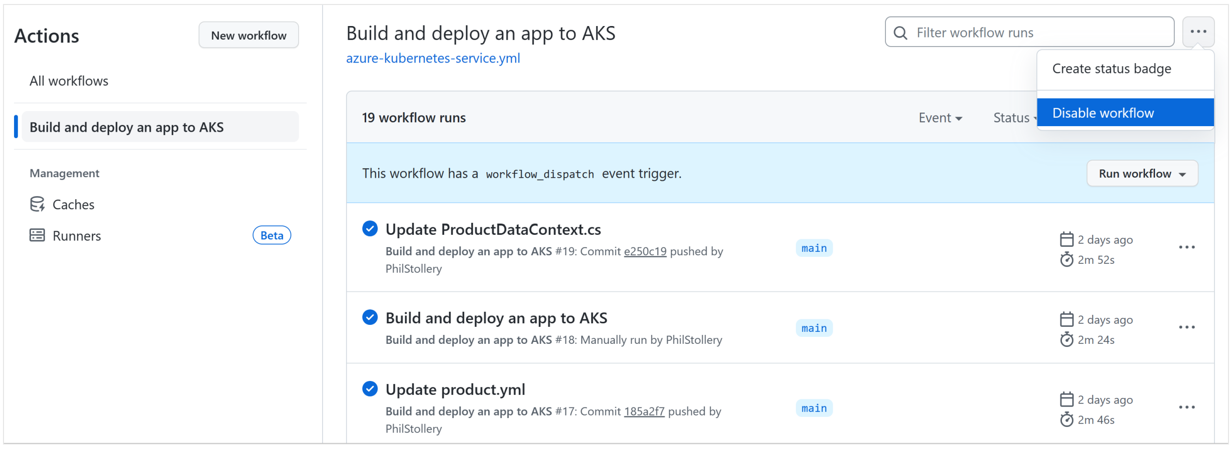
If you have more complex CD/CD workflows, Azure Pipelines can be scaled to support your needs. Azure Pipelines supports multiple stages, multiple jobs, and multiple steps. GitHub Actions only support a single job with multiple steps. This flexibility can be combined with automated testing scenarios.

> ⓘ **Note**
>
> When you create a new project in DevOps the free Azure Pipeline minutes might not be available. To request a free parallelism grant, fill out the parallelism request form. You'll need to do that before completing the next exercise.

Disable your GitHub Action

After reviewing the differences, you decide to change your app to use Azure Pipelines.

1. Go to your forked repository, on the **Actions** tab.
2. Select the **Build and deploy an app to AKS** workflow on the left.
3. Select the more options menu in the top right.



Actions
New workflow
All workflows
Build and deploy an app to AKS
Management
Caches
Runners Beta

Build and deploy an app to AKS
azure-kubernetes-service.yml
Filter workflow runs
Create status badge
Disable workflow

19 workflow runs
Event ▾ Status ▾

This workflow has a `workflow_dispatch` event trigger.
Run workflow ▾

Update ProductDataContext.cs Build and deploy an app to AKS #19: Commit e250c19 pushed by PhilStollery	main	2 days ago 2m 52s	···
Build and deploy an app to AKS Build and deploy an app to AKS #18: Manually run by PhilStollery	main	2 days ago 2m 24s	···
Update product.yml Build and deploy an app to AKS #17: Commit 185a2f7 pushed by PhilStollery	main	2 days ago 2m 46s	···

4. Select **Disable workflow**.

You've disabled the GitHub Action workflow. You'll now create an Azure Pipeline to build and deploy your app.

100 XP

Exercise - Create an Azure DevOps pipeline to deploy your cloud-native app

5 minutes

Your manager wants you to change the CI/CD for the companies eShop app to use Azure Pipelines. You'll now create an Azure DevOps Pipeline to build and deploy your products service.

Create an Azure DevOps Pipeline

Important

Before you begin you'll need to have an Azure DevOps account. If you don't have one, you can create one for free at dev.azure.com.

1. Sign in to dev.azure.com.
2. Select + **New project**.
3. For the **Project name**, enter **eShop deployment**.
4. Leave the **Visibility** set to **Private**, select **Create**.
5. On the left, select **Pipelines**, then select **Create Pipeline**.
6. On the **Connect page**, for **Where is your code?**, select **GitHub**.
7. If prompted, sign in to GitHub, and authorize Azure Pipelines to access your GitHub account.
8. For **Select a repository**, select your forked repository.
9. On the **Configure** page, select the **Deploy to Azure Kubernetes Service** option.
10. In the **Deploy to Azure Kubernetes Service** pane, select your Azure subscription, then select **Continue**.
11. If prompted, log in to your Azure subscription.
12. For the **Cluster**, select the AKS cluster you created in the previous unit **aks-eshop**.
13. For the **Namespace**, leave **Existing** selected, then select **default**.
14. For the **Container registry**, select the Azure Container Registry you created in the previous unit; for example, **acseshop186748394**.
15. For the **Image name**, enter **productservice**.
16. For the **Service Port**, enter **8080**.
17. Select **Validate and configure**.

Review the pipeline YAML file

Azure Pipelines uses YAML files to define the steps to build and deploy your app. The YAML file is stored in your GitHub repository and was created automatically for you, based on the information you provided.

Let's review the YAML file:

```yml
trigger:
- main

resources:
- repo: self

variables:

  # Container registry service connection established during pipeline creation
  dockerRegistryServiceConnection: '3bcbb23c-6fca-4ff0-8719-bfbdb64a89b1'
  imageRepository: 'productservice'
  containerRegistry: 'acseshop186748394.azurecr.io'
  dockerfilePath: '**/Dockerfile'
  tag: '$(Build.BuildId)'
  imagePullSecret: 'acseshop18674839414442d34-auth'

  # Agent VM image name
  vmImageName: 'ubuntu-latest'


stages:
- stage: Build
  displayName: Build 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: |
          $(tag)

    - upload: manifests
      artifact: manifests
```

```
- stage: Deploy
  displayName: Deploy stage
  dependsOn: Build

  jobs:
  - deployment: Deploy
    displayName: Deploy
    pool:
      vmImage: $(vmImageName)
    environment: 'PhilStollerymod9cloudnativeexercisecode-1959.default'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: KubernetesManifest@0
            displayName: Create imagePullSecret
            inputs:
              action: createSecret
              secretName: $(imagePullSecret)
              dockerRegistryEndpoint: $(dockerRegistryServiceConnection)

          - task: KubernetesManifest@0
            displayName: Deploy to Kubernetes cluster
            inputs:
              action: deploy
              manifests: |
                $(Pipeline.Workspace)/manifests/deployment.yml
                $(Pipeline.Workspace)/manifests/service.yml
              imagePullSecrets: |
                $(imagePullSecret)
              containers: |
                $(containerRegistry)/$(imageRepository):$(tag)
```

The **trigger** and **resources** sections define when the pipeline should run. In this case, the pipeline will run when a change is committed to the main branch of your repository.

The **variables** section defines the variables used in the pipeline. The variables are used to define the Azure Container Registry, and the Dockerfile to use.

The YAML then defines a **Build** job that uses the **ubuntu-latest** agent. The job uses the Docker task to build and push the image to the Azure Container Registry.

The last stage is to **Deploy** the updated product service to AKS. The job uses the **KubernetesManifest** task to deploy the image to AKS.

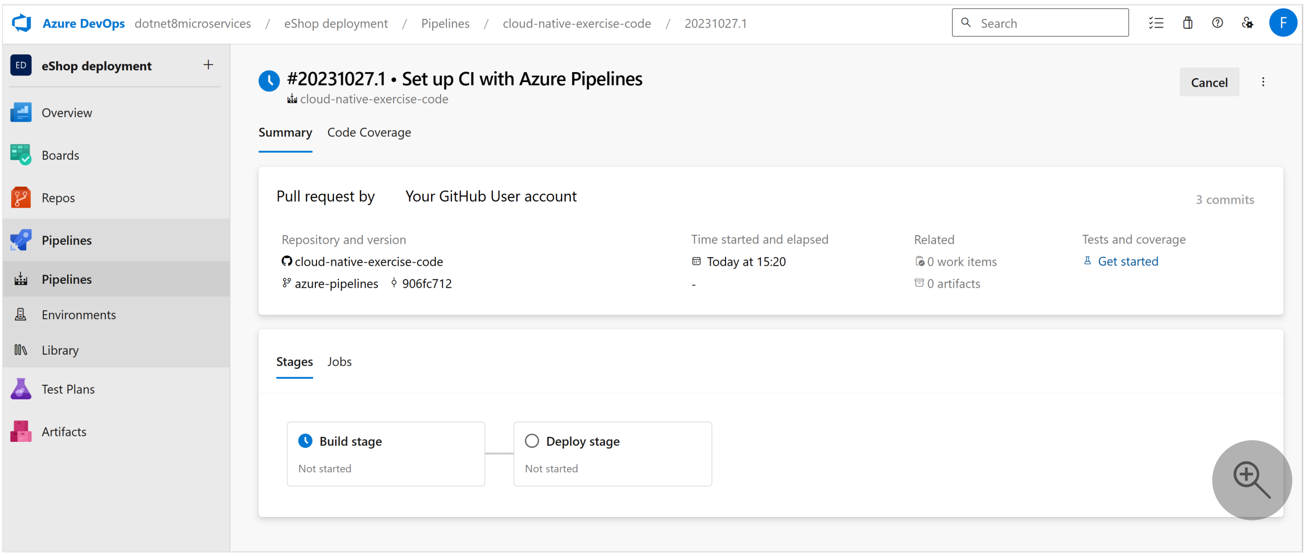
Run the pipeline

In the top right of the **Review your pipeline YAML** page, select **Save and run**. In the Save and run pane:

1. Select **Create a new branch for this commit**.
2. Leave all the other options set to their defaults.
3. Select **Save and run**.

Monitor and troubleshoot the pipeline

Azure Pipelines are monitored and managed from the Azure DevOps portal. Let's look at the output of running the pipeline you created.



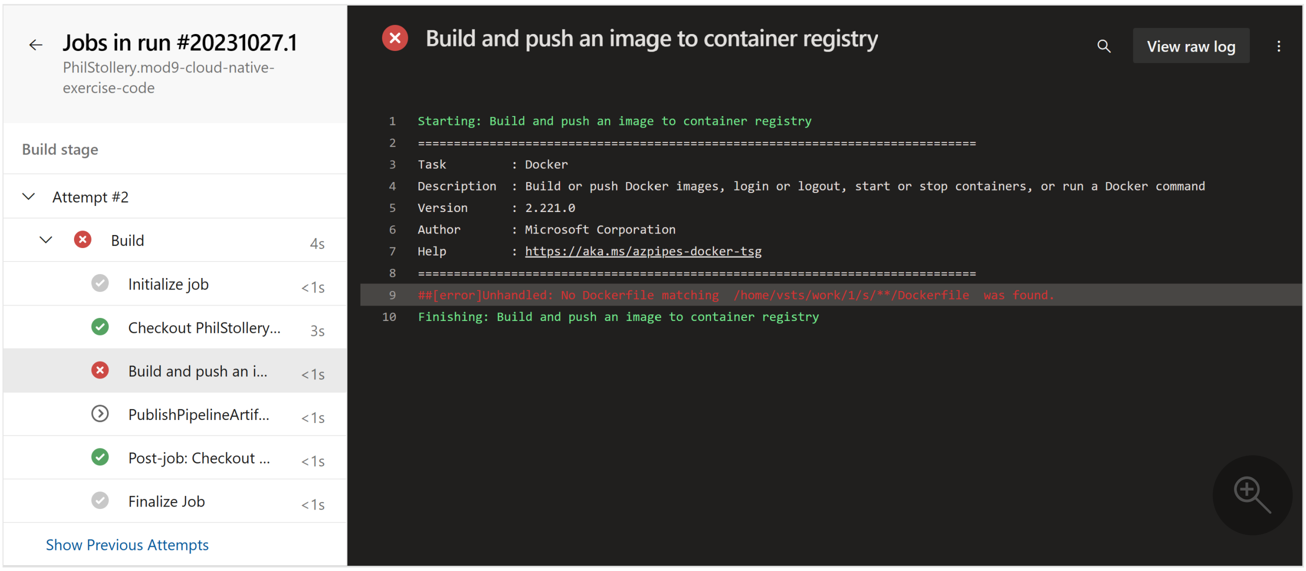
#20231027.1 • Set up CI with Azure Pipelines
cloud-native-exercise-code

Pull request by Your GitHub User account — 3 commits

Repository and version: cloud-native-exercise-code, azure-pipelines, 906fc712
Time started and elapsed: Today at 15:20, -
Related: 0 work items, 0 artifacts
Tests and coverage: Get started

Build stage — Not started
Deploy stage — Not started

The summary page shows you all the stages of your running pipeline. You can select a stage to view the steps in more detail. In a moment, you'll see that the pipeline has failed. Select the **Build** stage.


Jobs in run #20231027.1
PhilStollery.mod9-cloud-native-exercise-code
Build stage
Attempt #2
Build 4s
Initialize job <1s
Checkout PhilStollery... 3s
Build and push an i... <1s
PublishPipelineArtif... <1s
Post-job: Checkout ... <1s
Finalize Job <1s
Show Previous Attempts
View raw log

Build and push an image to container registry

```
Starting: Build and push an image to container registry
==============================================================================
Task         : Docker
Description  : Build or push Docker images, login or logout, start or stop containers, or run a Docker command
Version      : 2.221.0
Author       : Microsoft Corporation
Help         : https://aka.ms/azpipes-docker-tsg
==============================================================================
##[error]Unhandled: No Dockerfile matching  /home/vsts/work/1/s/**/Dockerfile  was found.
Finishing: Build and push an image to container registry
```

In the build stage, you can see that the build has failed. Select the **Build and push an image**

to Azure Container Registry step. The error in the log file shows:

Console

```
##[error]Unhandled: No Dockerfile matching  /home/vsts/work/1/s/**/Dockerfile
was found.
```

Fix the error

In DevOps, go back to the pipeline summary page. You're going to edit the created pipeline to fix the error.

1. In the top right, select the **More actions** menu, then select **Edit pipeline**.
2. Line 17 of the YAML file defines the Dockerfile to use, and by default the pipeline expects there to be a file named **Dockerfile** in the root of the repository.

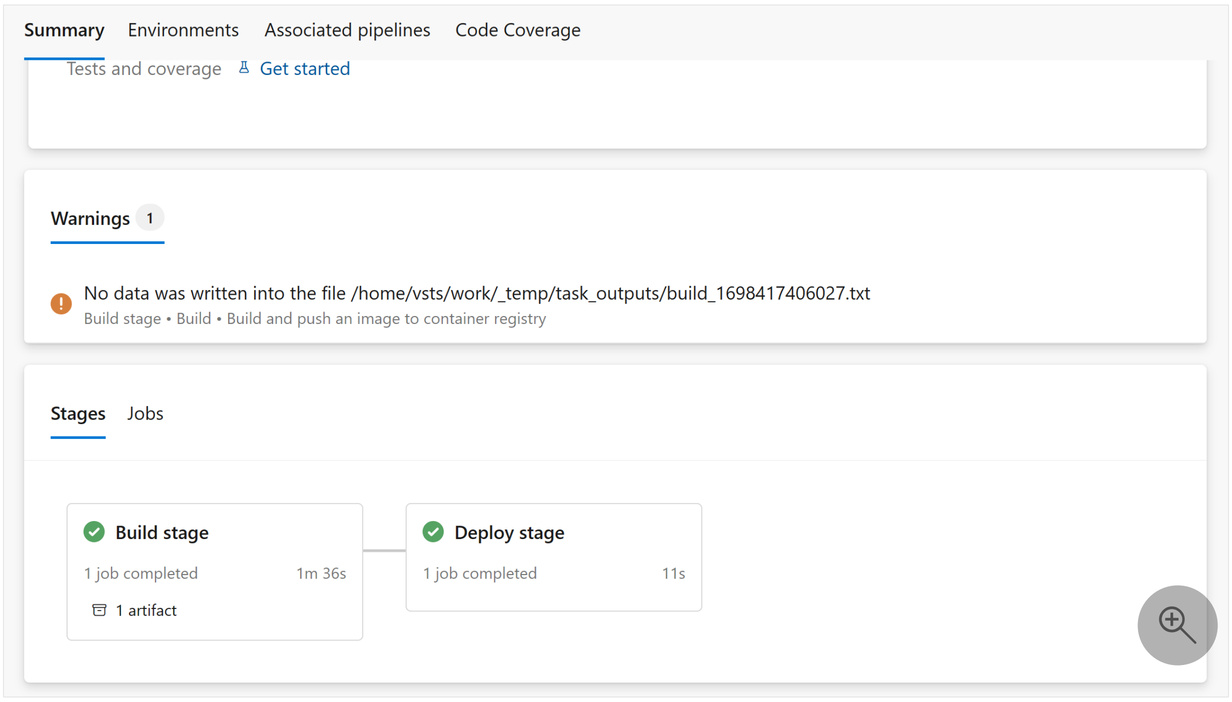
   The eShop uses a different docker file for the product service named **DockerfileProducts**. Edit Line 17 to be:

   YAML

   ```
     dockerfilePath: '**/DockerfileProducts'
   ```

3. Select **Save**.
4. In the **Save** pane, select **Save**.
5. Select **Run** then, in the **Run pipeline** pane, select **Run**.

   Watch the **Build stage** complete. The **Deploy stage** pauses until you select it and permit it to run.



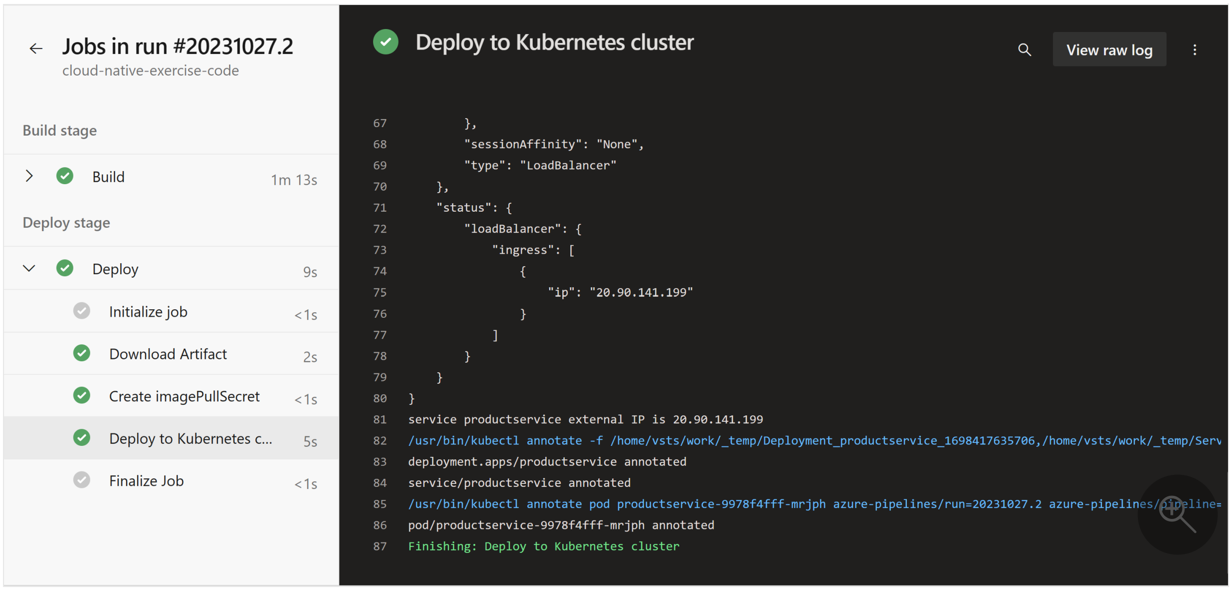
Warnings 1

No data was written into the file /home/vsts/work/_temp/task_outputs/build_1698417406027.txt
Build stage • Build • Build and push an image to container registry

Build stage
1 job completed 1m 36s
1 artifact

Deploy stage
1 job completed 11s

The pipeline completes successfully. Select the **Deploy** stage to view the steps.



Jobs in run #20231027.2
cloud-native-exercise-code

Build stage

Build	1m 13s

Deploy stage

Deploy	9s
Initialize job	<1s
Download Artifact	2s
Create imagePullSecret	<1s
Deploy to Kubernetes c...	5s
Finalize Job	<1s

Deploy to Kubernetes cluster

```
        },
        "sessionAffinity": "None",
        "type": "LoadBalancer"
    },
    "status": {
        "loadBalancer": {
            "ingress": [
                {
                    "ip": "20.90.141.199"
                }
            ]
        }
    }
}
service productservice external IP is 20.90.141.199
/usr/bin/kubectl annotate -f /home/vsts/work/_temp/Deployment_productservice_1698417635706,/home/vsts/work/_temp/Serv
deployment.apps/productservice annotated
service/productservice annotated
/usr/bin/kubectl annotate pod productservice-9978f4fff-mrjph azure-pipelines/run=20231027.2 azure-pipelines/pipeline=
pod/productservice-9978f4fff-mrjph annotated
Finishing: Deploy to Kubernetes cluster
```

✓ 100 XP

Summary

2 minutes

In this module, you:

- Authenticated GitHub Actions to an Azure Container Registry instance.
- Stored sensitive information that GitHub Actions uses.
- Implemented a GitHub action to build the product service's container image in Container Registry.
- Modified the product service adding a new product to trigger a build.
- Implemented a GitHub action to deploy the product service container to the Azure Kubernetes Service (AKS) cluster.
- Rolled back the product service to the previous deployment.

Remove Azure service principal

Earlier, you created an Entra service principal that allows GitHub to authenticate to Azure resources. To remove the service principal, you can use the Azure CLI.

1. Use the following Azure CLI command to return a list of service principal identifiers from Microsoft Entra ID:

   Azure CLI

   ```
   az ad sp list --show-mine --query "[?contains(displayName,'eShop')].appId"
   --output tsv
   ```

2. Filter the service principals to the following identifiers:

   - Owned by the current user.
   - Containing the string `eShop` in the display name.

3. Use the `az ad sp delete` Azure CLI command to remove each matching service principal.

4. Delete the Azure resource group `rg-eshop` to delete all the resources you created in previous units.

Cleanup Codespace

You can delete the codespace on GitHub under **By repository** where you see **MicrosoftDocs/mslearn-dotnet-cloudnative-devops**.

Learn more about microservices

- Architecting Cloud Native .NET Applications for Azure
- Video: Implement microservices patterns with .NET and Docker containers