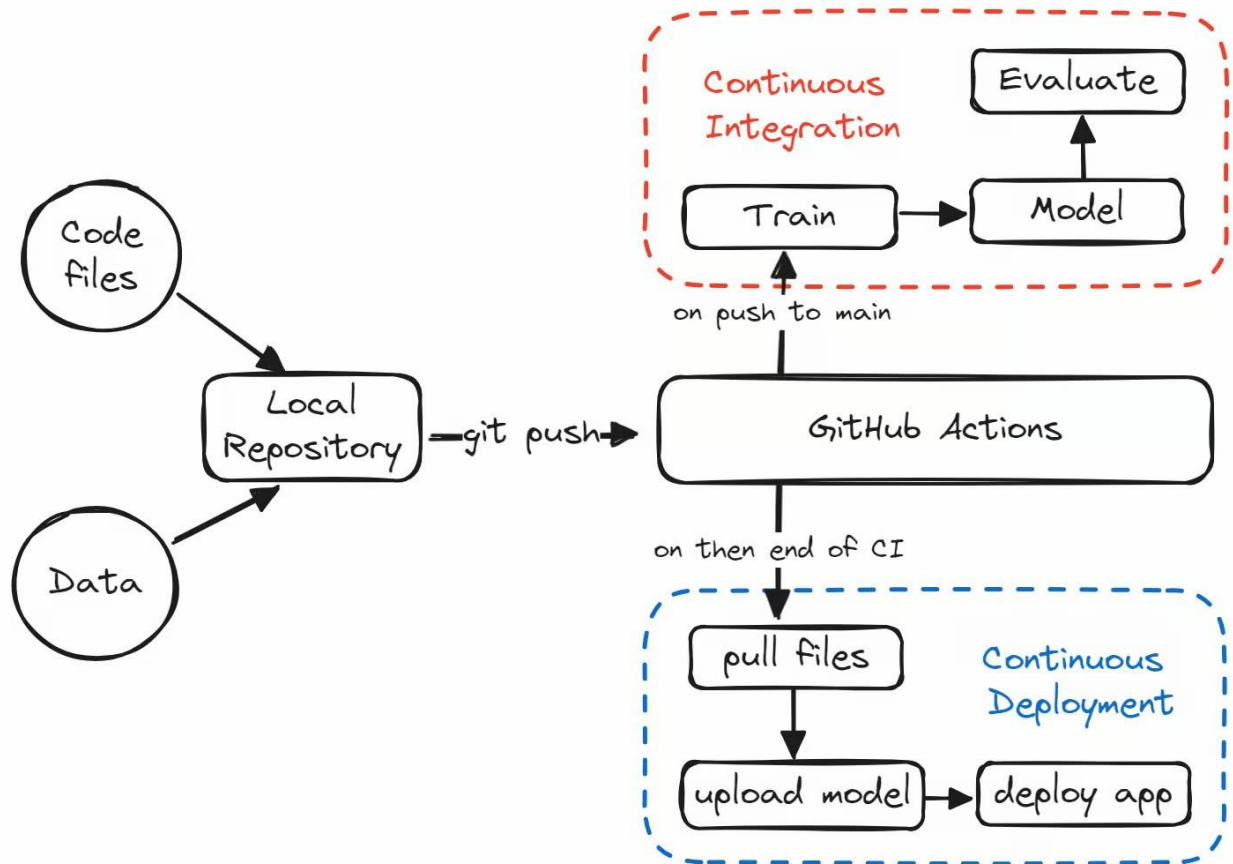


# A Guide to CI/CD Using GitHub Actions and Deployment to Azure Containers



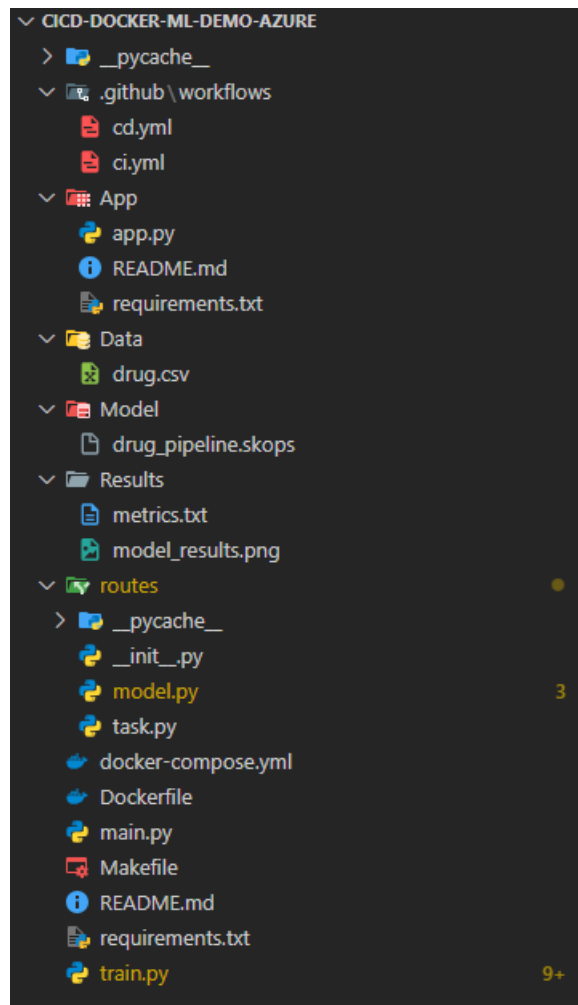
## Overview

This document provides a step-by-step guide on integrating CI/CD (Continuous Integration and Continuous Deployment) using GitHub Actions and deploying the application to an Azure Docker Container Service. It demonstrates the deployment of a machine learning application built with Fast API, which is containerized using Docker. Finally, the project will be pushed to GitHub, where GitHub Actions will be used to automate testing and enable seamless CI/CD.

This document will refer from GitHub repo: <https://github.com/guisiraST/CICD-docker-ML-demo-azure> You can try to play with this repositories and do step by step by following document.

Note: This document focuses only on CI/CD, so we will not cover anything related to machine learning.

File structure:



## Folder Explanation

- .github/workflows/: Contains the CI/CD workflow files (ci.yml and cd.yml). These scripts automate the Continuous Integration and Continuous Deployment (CI/CD) process using GitHub Actions.
- App/: This folder contains the UI-related files that integrate with Gradio. However, **this guide focuses on the API, which is stored in the routes/ folder.**
- Data/: Stores the dataset (drug.csv) used to train the machine learning model for drug classification. The dataset is sourced from [Kaggle](#).
- Model/: Contains the trained model weights.
- Results/: Stores performance metrics, visualizations, and other evaluation results. These outputs will be displayed in GitHub pull requests after the CI process is completed.

- routes/: Contains all API-related files and logic, which are integrated with FastAPI.
- Makefile: Defines scripts for automation, such as running commands in the ci.yml workflow.
- train.py: The script used for training the machine learning model.

## Training and Evaluating the Drug Classification Model

As mentioned earlier, we will not go into the details of training the model. However, before setting up CI/CD testing, you need to running the code for checking:

1. Start the Docker container by running: `docker-compose up -d`
2. Access the container using: `docker exec -it <container_name> bash`
3. Run the training script: `python train.py`

This will train the model, save the trained weights, and store performance metrics in the Model and Results folders.

## Continuous Integration Section

Continuous Integration (CI) is a development practice where developers frequently merge their code changes into a shared repository, often multiple times a day. Each integration is automatically verified by building the project and running automated tests to detect errors early in the development cycle.

For Continuous Integration (CI), we will focus on the ci.yml file, which contains the script for running CI in GitHub Actions. Additionally, we will explain the Makefile, which works alongside ci.yml to automate tasks within the CI pipeline.

ci.yml

```
name: Continuous Integration
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
  workflow_dispatch:
```

- Runs automatically when code is pushed to the main branch or when a pull request is created.
- Can be triggered manually using workflow\_dispatch.

```
permissions: write-all
```

- Grants write access to all resources (e.g., for logging results or pushing updates).

```
jobs:  
  build:  
    runs-on: ubuntu-latest
```

- The job runs in a Ubuntu-based environment.

```
- uses: actions/checkout@v3
```

- This action checks-out your repository under GITHUB\_WORKSPACE, so your workflow can access it.

```
- uses: iterative/setup-cml@v2
```

- Installs CML (Continuous Machine Learning) for model evaluation and reporting.

```
- name: Start Docker  
  run: make build-docker
```

- Runs make build-docker (defined in the Makefile) to start a Docker container.

```
- name: Train  
  run: make run-train-docker
```

- Executes make run-train-docker to run the training script inside the Docker container.

```
- name: Evaluation  
  env:  
    REPO_TOKEN: ${ secrets.GITHUB_TOKEN }  
  run: make eval
```

- Runs make eval, which generates a report with model performance metrics.
- Uses CML to post the results as a comment on the GitHub pull request.

```
- name: Stop Docker  
  run: make stop-docker
```

- Runs make stop-docker to shut down the Docker container.

## Makefile

```
build-docker:  
    docker compose -f docker-compose.yml up -d  
    docker ps -a  
    docker images
```

- Starts the Docker container in detached mode (-d).
- Lists running containers and available images.

```
run-train-docker:  
    docker exec cisd-docker-ml-demo-azure-core-1 python train.py
```

- Executes train.py inside the Docker container.

eval:

```
echo "## Model Metrics" > report.md
cat ./Results/metrics.txt >> report.md
echo '\n## Confusion Matrix Plot' >> report.md
echo '![Confusion Matrix](./Results/model_results.png)' >> report.md
cml comment create report.md
```

- Extracts model performance metrics and confusion matrix.
- Posts the report as a GitHub comment using CML.

stop-docker:

```
docker compose -f docker-compose.yml down
```

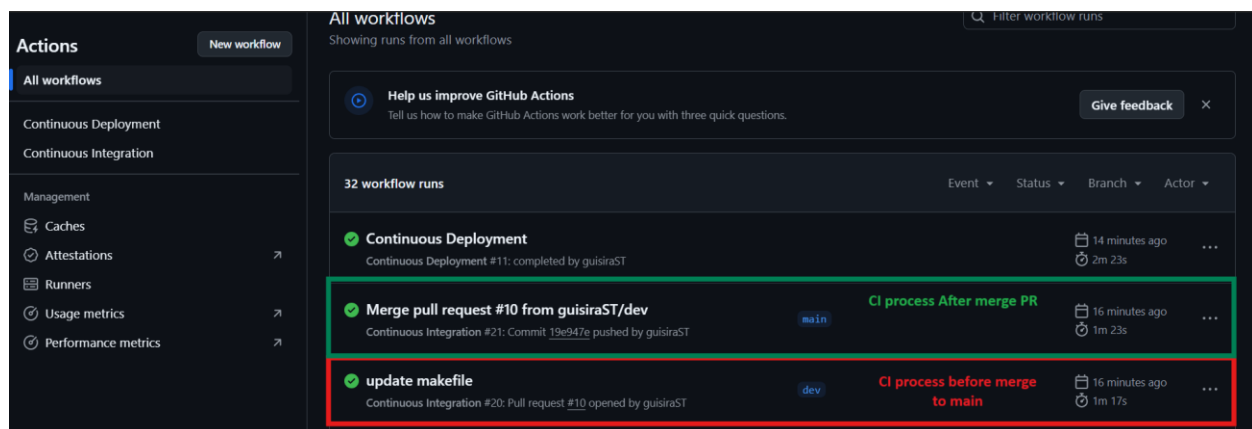
- Stops and removes the Docker container.

Note: To test the CI process by following these steps:

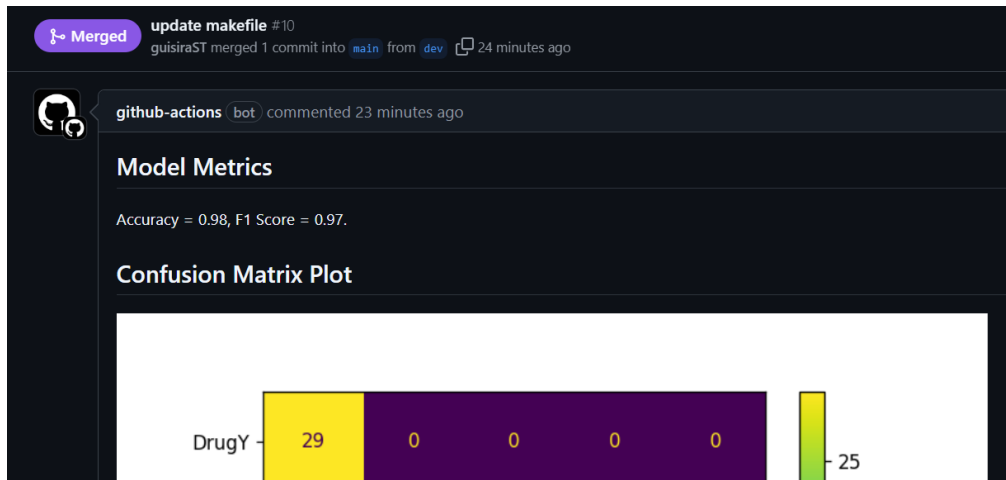
1. Comment out all the script in cd.yml.
2. Create a new branch in GitHub.
3. Push the code to the new branch.
4. Create a pull request to merge from new branch to the main branch.

Once the pull request is created, GitHub Actions will automatically run the script in ci.yml. You can check the status of the CI process under the GitHub Actions tab.

After the workflow is completed, GitHub Actions will post the results and visualization metrics in the conversation section of the pull request.



GitHub Action tab (Green = Complete)



Pull request tab in conversation section

## Continuous Deployment Section

Continuous Deployment (CD) is a software development practice where code changes that pass automated testing are automatically released into production without manual intervention. This ensures that updates, bug fixes, and new features reach users quickly and efficiently.

For Continuous Deployment (CD), we will explain the `cd.yml` script, which runs in GitHub Actions, and the setup process for deploying to Azure Container Service.

`cd.yml`

```
on:
  workflow_run:
    workflows: ["Continuous Integration"]
    types:
      - completed
    branches: [ "main" ]
  workflow_dispatch:
```

- This workflow runs automatically after the Continuous Integration (CI) workflow has completed successfully.
- It is triggered only for the main branch.
- It also allows for manual deployment through GitHub Actions UI using `workflow_dispatch`.

```
env:
  AZURE_CONTAINER_REGISTRY: cicddemoazure
  CONTAINER_APP_NAME: ccd-docker-ml-demo-azure
  RESOURCE_GROUP: cicddemo
```

- Defines environment variables used throughout the workflow:

- AZURE\_CONTAINER\_REGISTRY: The name of the Azure Container Registry (ACR).
- CONTAINER\_APP\_NAME: The name of the Azure Container App where the model will be deployed.
- RESOURCE\_GROUP: The Azure resource group managing the container app.

```
jobs:
  deploy:
    runs-on: ubuntu-latest
```

- Runs on Ubuntu.
- Handles the deployment process.

```
- uses: actions/checkout@v3
```

- Pulls the latest project files from GitHub into the workflow runner.

```
- name: Log in to ACR
  uses: docker/login-action@v3
  with:
    registry: ${ env.AZURE_CONTAINER_REGISTRY }.azurecr.io
    username: ${ secrets.AZURE_REGISTRY_USERNAME }
    password: ${ secrets.AZURE_REGISTRY_PASSWORD }
```

- Logs into Azure Container Registry (ACR) using credentials stored in GitHub Secrets (AZURE\_REGISTRY\_USERNAME & AZURE\_REGISTRY\_PASSWORD).
- This allows the workflow to push Docker images to ACR.

```
- name: Build and push Docker image to ACR
  uses: docker/build-push-action@v6
  with:
    push: true
    tags: ${ env.AZURE_CONTAINER_REGISTRY }.azurecr.io/${ env.CONTAINER_APP_NAME }:${{
github.sha }}
    file: Dockerfile
```

- Builds a Docker image from the Dockerfile.
- Pushes the built image to Azure Container Registry (ACR).
- The image is tagged with `\${ github.sha }` (commit hash) to ensure version tracking.

```
- name: Azure Login
  uses: azure/login@v2
  with:
    creds: ${ secrets.AZURE_CREDENTIALS }
```

- Authenticates to Azure using credentials stored in GitHub Secrets (AZURE\_CREDENTIALS).
- Allows the workflow to deploy the container to Azure.

```
- name: Deploy to Azure Container Apps
  uses: azure/container-apps-deploy-action@v1
  with:
    imageToDeploy: ${ env.AZURE_CONTAINER_REGISTRY }.azurecr.io/${ env.CONTAINER_APP_NAME }:${{
github.sha }}
    resourceGroup: ${ env.RESOURCE_GROUP }
```

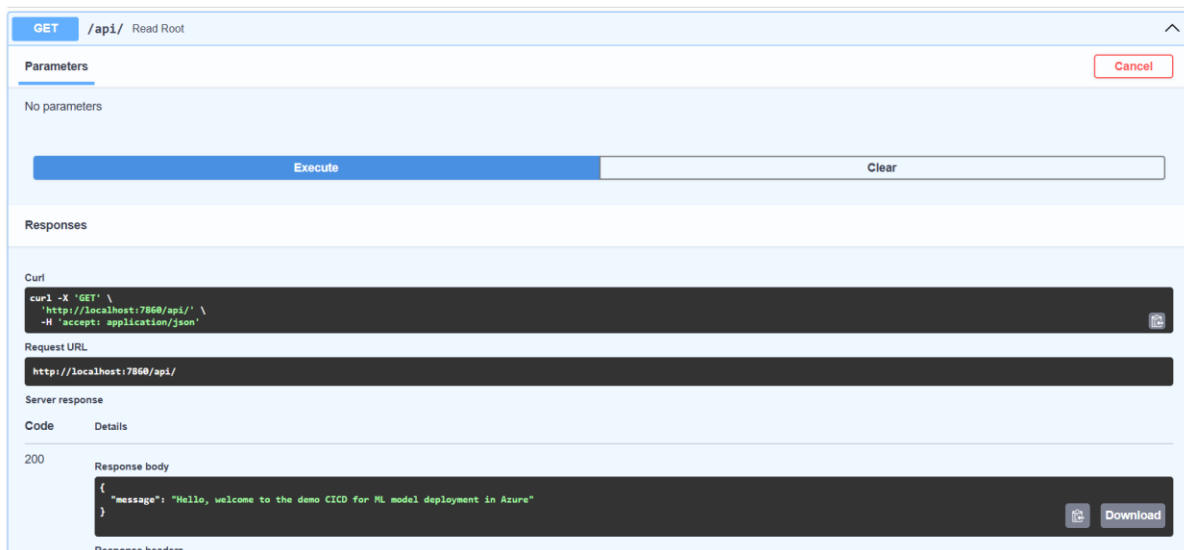
```
containerAppName: ${ env.CONTAINER_APP_NAME }}
```

- Deploys the Docker image to Azure Container Apps.
- Uses:
  - The container image from ACR.
  - The resource group to locate the Azure Container App.
  - The container app name for deployment.

## Setup your image into Azure container app

Before testing the CI/CD pipeline on GitHub Actions, particularly the Continuous Deployment (CD) process, you need to set up the following:

1. Push Your Docker Image to Azure Container Registry (ACR)
  - Store your Docker container in Azure Container Registry (ACR) to make it available for deployment.
2. Configure Azure Container Apps
  - Set up Azure Container Apps with the correct configuration to match your container.
3. Start the Container App Service
  - Launch the container in Azure and ensure it's running correctly.
4. Test the Application
  - Access the Application URL to verify that the app is working properly before integrating it into the CI/CD pipeline.



Access the Application URL



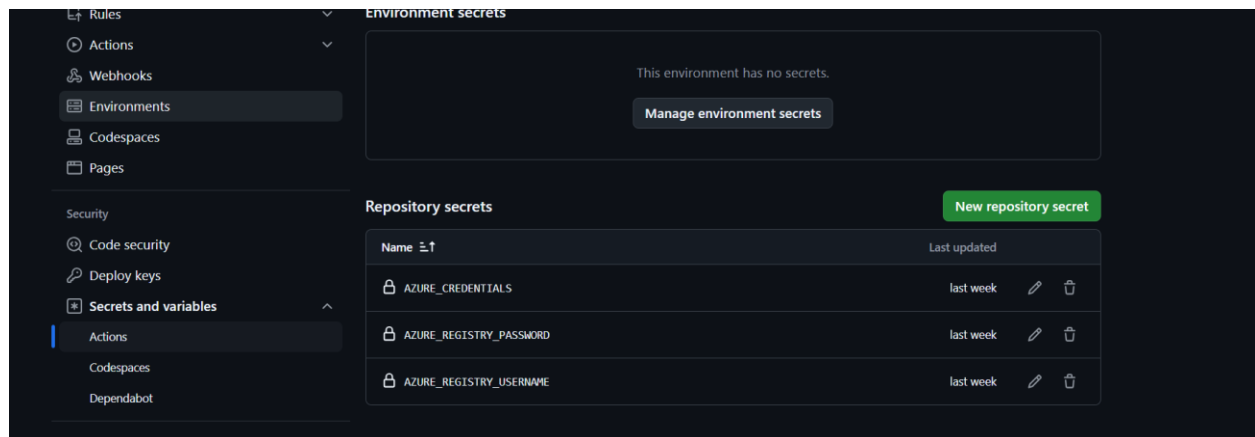
# Set up any secrets into Repository secrets

You need to configure repository secrets in GitHub to securely store credentials and API keys.

How to Set Up Repository Secrets:

1. Go to your GitHub repository.
2. Navigate to Settings.
3. Click on Secrets and Variables.
4. Add the necessary secrets (such as Azure credentials, container registry username, and password).

This ensures that your CI/CD pipeline can access secure credentials during the deployment process.



## Set up Repository secret page

For AZURE\_REGISTRY\_PASSWORD and AZURE\_REGISTRY\_USERNAME, you can retrieve these credentials from Azure Container Registry (ACR) by following these steps:

How to Get Your ACR Credentials:

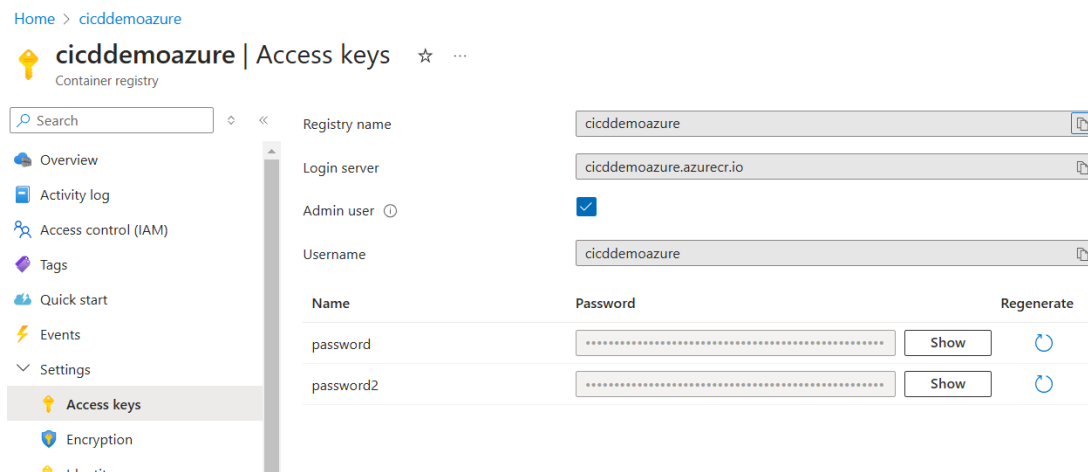
1. Go to your Azure Portal.
2. Navigate to Container Registry and select your registry.
3. Click on Access keys in the left sidebar.
4. Enable the Admin user option (if not already enabled).
5. Copy the Username and Password displayed.

Storing Credentials in GitHub Secrets:

1. Go to your GitHub repository.

2. Click on Settings → Secrets and variables → Actions.
3. Click New repository secret.
4. Add the following secrets:
  - AZURE\_REGISTRY\_USERNAME → Paste the copied username.
  - AZURE\_REGISTRY\_PASSWORD → Paste the copied password.

This allows GitHub Actions to authenticate and push Docker images to your Azure Container Registry securely during deployment.



For AZURE\_CREDENTIALS you can copy config by use this command for get the credentials: `az ad sp create-for-rbac --name "cicddemoazure" --role contributor --scopes /subscriptions/f2cd0c66-edbe-xxxx-xxxx-c004a199fe6d --sdk-auth` (Make sure you have already logged in to Azure on your PC using "az login".)

Breakdown of Parameters:

- `az ad sp create-for-rbac`: Creates a service principal with RBAC permissions.
- `--name "cicddemoazure"`: Assigns the name "cicddemoazure" to the service principal.
- `--role contributor`: Grants Contributor permissions, allowing full management access except for granting permissions to others.
- `--scopes /subscriptions/f2cd0c66-edbe-xxxx-xxxx-c004a199fe6d`: Limits the permission scope to the specified Azure Subscription (identified by the Subscription ID).
- `--sdk-auth`: Returns the credentials in a JSON format (useful for authentication in automated deployments).

```
C:\Users\milan>az ad sp create-for-rbac --name "refitapi" --role contributor --scopes
/subscriptions/f2cd0c66-edbe-47c7-9006-c004a199fe6d --sdk-auth
Option '--sdk-auth' has been deprecated and will be removed in a future release.
Creating 'contributor' role assignment under scope '/subscriptions/f2cd0c66-edbe-47c7-9006-c004a199fe6d'
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": "a0e90eba-3",
  "clientSecret": "k9A8Q~",
  "subscriptionId": "f2cd",
  "tenantId": "cf7cfe4c-8",
  "activeDirectoryEndpoint": "https://login.microsoftonline.com/tenantId",
  "resourceManagerEndpoint": "https://management.azure.com/",
  "activeDirectoryGraphEndpoint": "https://graph.microsoft.com/",
  "sqlManagementEndpoint": "https://management.azure.com/",
  "galleryEndpointUrl": "https://gallery.azure.com/",
  "managementEndpointUrl": "https://management.azure.com/"
}
```

## Actions secrets / New secret

Name \*

AZURE\_CREDENTIALS

Secret \*

```
"clientSecret": "k9A8Q~",
"subscriptionId": "f2cd",
"tenantId": "cf7cfe4c-8",
"activeDirectoryEndpoint": "https://login.microsoftonline.com/tenantId",
"resourceManagerEndpoint": "https://management.azure.com/",
"activeDirectoryGraphEndpoint": "https://graph.microsoft.com/",
"sqlManagementEndpoint": "https://management.azure.com/",
"galleryEndpointUrl": "https://gallery.azure.com/",
"managementEndpointUrl": "https://management.azure.com/"
```

Add secret

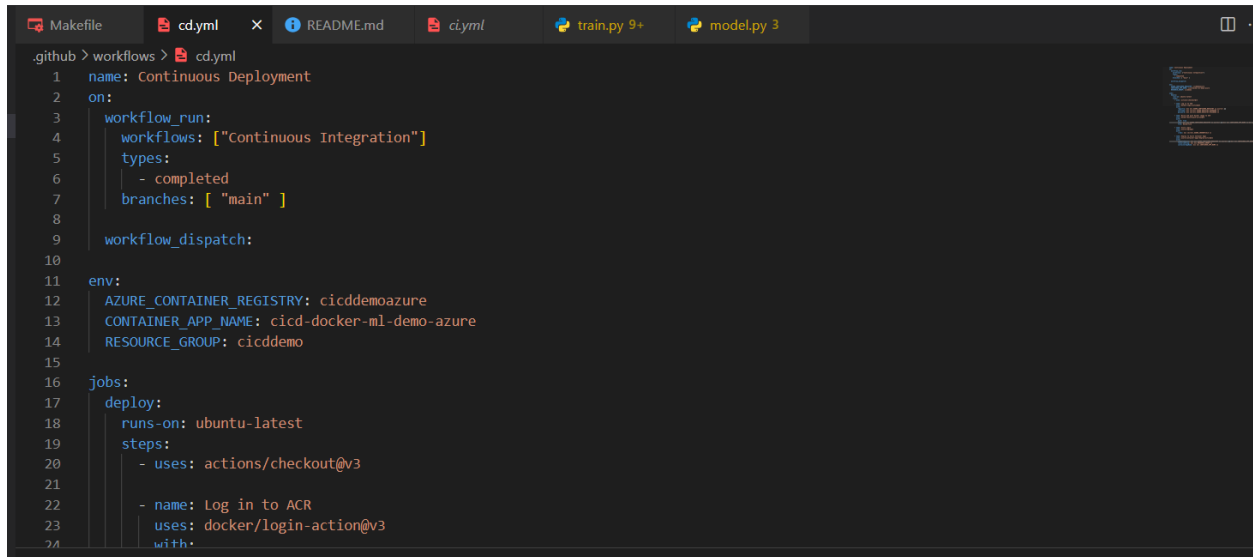
Azure\_credentials

Before completing the setup, don't forget to update your cd.yml file with the correct service names and environment variables to match your Azure configuration.

Here's what each variable represents:

- AZURE\_CONTAINER\_REGISTRY: cicddemoazure -> Your Azure Container Registry (ACR) name where the Docker images are stored.
- CONTAINER\_APP\_NAME: cicd-docker-ml-demo-azure -> The name of your Azure Container App where the application will be deployed.

- `RESOURCE_GROUP: cicddemo` -> The Azure Resource Group that manages your container and other related services.



```
.github > workflows > cd.yml
1  name: Continuous Deployment
2  on:
3    workflow_run:
4      workflows: ["Continuous Integration"]
5      types:
6        - completed
7      branches: [ "main" ]
8
9    workflow_dispatch:
10
11  env:
12    AZURE_CONTAINER_REGISTRY: cicddemoazure
13    CONTAINER_APP_NAME: cicd-docker-ml-demo-azure
14    RESOURCE_GROUP: cicddemo
15
16  jobs:
17    deploy:
18      runs-on: ubuntu-latest
19      steps:
20        - uses: actions/checkout@v3
21
22        - name: Log in to ACR
23          uses: docker/login-action@v3
24          with:
```

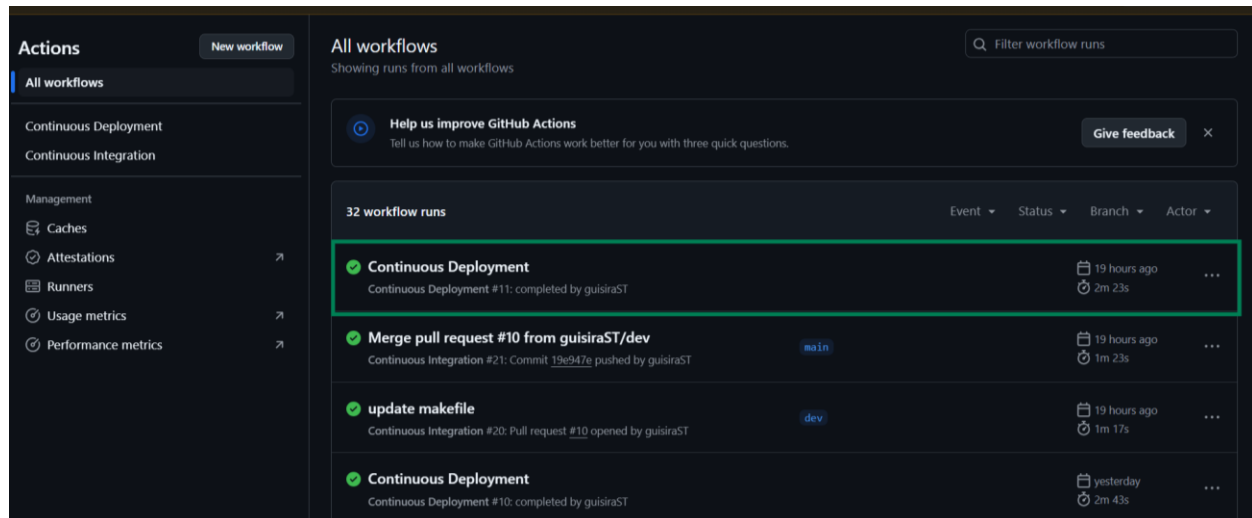
## Final Step: Testing the CI/CD Pipeline

Now that everything is set up, it's time to test if your CI/CD pipeline is working correctly. (Recheck your `cd.yml` file. If it is commented, just uncomment it.)

How to Test:

1. Make a Small Change
  - Modify something in your project, such as changing a model parameter or text output, to ensure that it differs from the currently deployed version in your Azure Container App.
2. Push Your Changes to a New Branch
  - Create a new branch in your GitHub repository and push your changes there.
3. Create a Pull Request
  - Open a Pull Request (PR) to merge your changes into the main branch.
  - This triggers the CI process to check if the changes are valid.
4. Merge the PR If CI Passes
  - If the CI checks pass, merge the PR into the main branch.
  - This will trigger the CI/CD pipeline again, but this time on the main branch.
5. Verify the Deployment

- After the CI process completes, the CD process will start automatically.
- You can check the deployment status in GitHub Actions. If everything works correctly, it will display a green success indicator



## Final Verification of the CD Process

Now that your Continuous Deployment (CD) process is complete, you can verify that everything is working as expected.

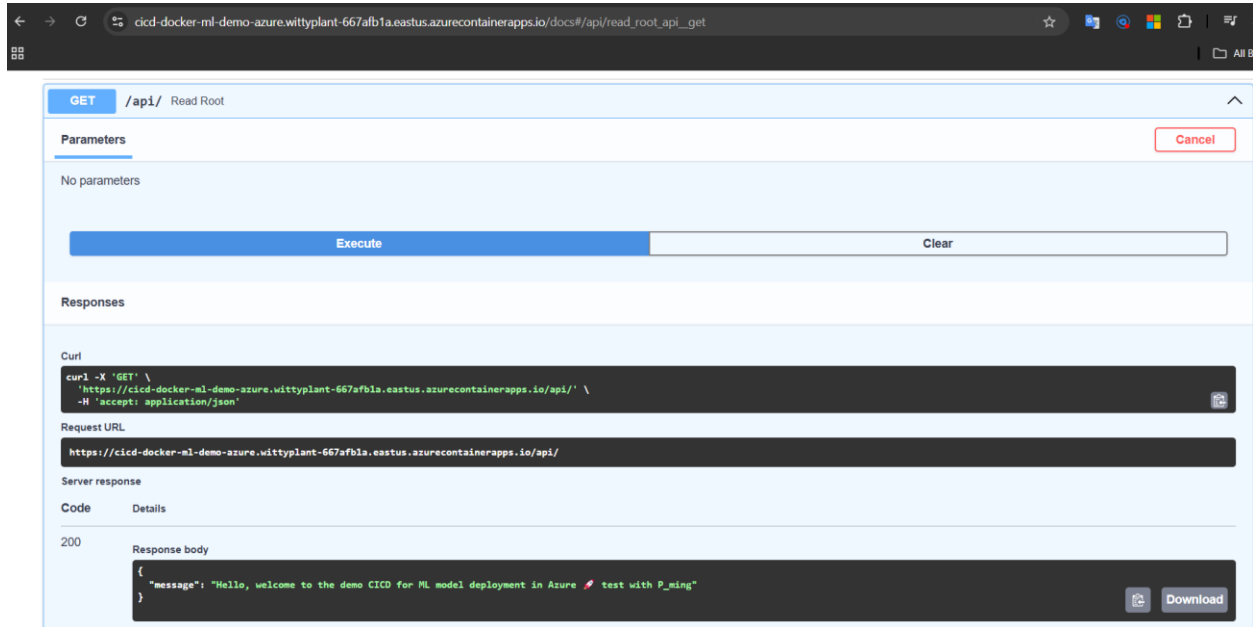
How to Verify Deployment:

1. Check Your Application URL
  - Go to your Azure Container Service and find the Application URL.
  - Open the URL in your web browser to see if your deployed application is running.
2. Confirm the Changes
  - Verify whether the changes you made (e.g., model parameters, text output) are reflected in the live application.
3. Successful Deployment
  - If you see the updated changes, congratulations!
  - Your GitHub Actions CI/CD pipeline is successfully set up and working.

If the changes do not appear, try the following:

- Check the GitHub Actions logs for errors.
- Ensure the CD pipeline ran successfully without any failures.
- Manually restart the container service in Azure if necessary.

Now, you have successfully deployed your machine learning model using CI/CD with GitHub Actions and Azure Container Apps



Reference:

Dataset

- <https://www.kaggle.com/datasets/prathamtripathi/drug-classification>

huggingface space

- [https://sirah-drug-classification-docker-fastapi.hf.space/docs#/api/predict\\_ml\\_api\\_predict\\_post](https://sirah-drug-classification-docker-fastapi.hf.space/docs#/api/predict_ml_api_predict_post)
- <https://huggingface.co/spaces/SiraH/Drug-classification-cicd>

Guild deploy docker to huggingface space

- <https://huggingface.co/blog/HemanthSai7/deploy-applications-on-huggingface-spaces>
- <https://www.datacamp.com/tutorial/ci-cd-for-machine-learning>

Guild deploy docker with github action to azure

- <https://www.youtube.com/watch?v=yjbzWq7bvNc>