CI/CD with GitHub Actions & Branching Strategy – AWS Class Notes

1. Introduction to CI/CD

- CI/CD stands for Continuous Integration and Continuous Deployment/Delivery.
- GitHub Actions is a popular DevOps tool for automating workflows like testing, building, and deploying applications directly from GitHub.

2. Importance of Branching Strategy

- In teams with multiple developers, code changes are frequent and simultaneous.
- This increases the chances of:
    - Merge conflicts
    - Application-breaking bugs
- To avoid this, teams follow a branching strategy – a structured way to manage and merge code safely.

3. Common Branch Types

Branch	Purpose
`main` / `master`	The stable production-ready version of the code.
`develop`	Integrates features before they are pushed to main. Acts as a staging area.
`feature/*`	Individual developer branches for new features. Merged into develop.
`uat`	User Acceptance Testing. Used to test features in a near-prod environment.
`preprod`	Pre-production environment, just before release to main.

hotfix/*	For urgent production fixes. Merged into main and develop.

4. Summary of Benefits

- Reduced production bugs.
- Organized code flow.
- Easier collaboration among developers.
- Safer deployment through stages like uat and preprod.

AWS DevOps Class Notes: CI/CD with GitHub Actions + Kubernetes + ArgoCD

1. CI/CD Pipeline Overview (Kubernetes Edition)

Previous Flow (Basic CI/CD):

```
SCSS

Source (GitHub) → CodeBuild → Deployment (e.g., EC2)
```

Current Flow (with Kubernetes & ArgoCD):

```
Markdown

1. Developer pushes code →
2. GitHub Action is triggered →
3. Docker image is built →
4. Docker image is pushed to container registry →
5. Kubernetes manifest files are updated →
6. ArgoCD detects the change →
7. ArgoCD applies updated manifests →
8. New version is deployed on Kubernetes
```

2. Step-by-Step Class Activity

Step 1: Get Base Code

- Clone the docker_demo project from Subhasis Sir's GitHub repository.
- Inside this cloned folder:
  - Create a Dockerfile.
  - Paste the content from the ci_cd_argo repository's Dockerfile into your newly created Dockerfile.

Step 2: Add `nginx.conf`

- Copy the `nginx.conf` file from the `ci_cd_argo` repository into your `docker_demo/` directory.

3. GitHub Setup

Create a GitHub Repository

In your terminal, navigate into the `docker_demo` directory and execute the following commands:

```
Bash

git init
git add.
git commit -m "initial commit"
git remote add origin https://github.com/YOUR_USERNAME/YOUR_REPO_NAME.git
git branch -M main
git push -u origin main
```

4. Set Up GitHub Actions

Enable Actions in GitHub

- Go to your GitHub repository in your web browser.
- Click on the "Actions" tab.
- Select "Set up a workflow yourself".
- Create a file named `main.yaml` (you can also use names like `ci-cd.yaml`).

Pull Workflow into VS Code

- You can execute `git pull` from your local `docker_demo` directory to fetch the newly created workflow file.
- Alternatively, copy the content of the `.yaml` file from the GitHub Readme and paste it into your local `main.yaml` file.

Important: Update the repository name within the `.yaml` file to precisely match your GitHub repository details wherever required.

5. Kubernetes Manifests Setup

Create a `deploy` Folder

Inside your `docker_demo` directory, create a new folder:

```
Bash

mkdir deploy
```

Within this `deploy` folder, create the following two files:

- `deployment.yaml`
- `svc.yaml`

Copy From Sir's GitHub

- Obtain the manifest files (`deployment.yaml` and `svc.yaml`) from the `kubernetes` folder within Subhasis Sir's GitHub repository.
- Paste the content of these files into your corresponding files in the `deploy/` folder.

Update Manifests

- Change the `image` field within your `deployment.yaml` to reflect your DockerHub image name (e.g., `yourusername/docker_demo`).
- Optionally, update deployment names, application labels, or any other relevant fields as needed.

7. Make a Change to Test the CI/CD Pipeline

From your local repository (e.g., `docker_demo` directory), make a minor code change (e.g., to `index.html`), then commit and push:

```
Bash

git add .
git commit -m "Made some changes"
git push origin main
```

- At this point, GitHub will detect the changes.
- However, the CI/CD pipeline might initially fail due to permission issues.

8. Fixing GitHub Actions Permissions

1. Go to your GitHub Repository in your web browser.
2. Navigate to: `Settings` → `Actions` → `General` → `Workflow permissions`.
3. Select the option "Read and write permissions".
4. Click "Save" to apply the changes.

9. Setting Up GitHub Secrets for DockerHub

These secrets will be securely accessed by your `main.yaml` GitHub Actions workflow file.

1. Navigate to: `Settings` → `Secrets and variables` → `Actions` → `New repository secret`.
2. You will find the required secret keys by examining your `main.yaml` (or your chosen workflow file). Typically, these are:

- DOCKER_USERNAME
- DOCKER_PASSWORD

10. DockerHub Setup for CI/CD

1. Go to DockerHub and log in to your account.
2. **Get Username:**
    - Your DockerHub username can be found on the top-right corner of the page or within your account settings.
    - Save this username as the value for the DOCKER_USERNAME secret key in GitHub.
3. **Generate Access Token:**
    - Navigate to: Account Settings → Security → New Access Token.
    - Configure the token with the following settings:
        - Name: github-ci
        - Access: Read, Write, Delete
        - Expiry: 30 days (or choose an appropriate duration)
    - Click "Generate".
    - **Crucially, copy the token immediately** (it will not be displayed again).
    - Save this token as the value for the DOCKER_PASSWORD secret key in GitHub.

11. Making a Minor Code Change to Trigger Pipeline

Make a small update to your code (e.g., edit index.html locally), then commit and push the changes:

```
Bash

git add .
git commit -m "Updated index.html to test pipeline"
git push origin main
```

Now, the GitHub Actions workflow will be automatically triggered:

- It will build the Docker image.
- It will push the Docker image to DockerHub.
- It will update the Kubernetes manifest files in your GitHub repository (if configured to do so).
- ArgoCD will detect these changes and redeploy your application to Kubernetes.

AWS DevOps Class Notes: CI → CD using GitHub Actions, DockerHub, EKS, ArgoCD, & Ingress

12. DockerHub Repo Note

- There is no need to manually create a DockerHub repository beforehand.
- The repository will be automatically created when the first Docker image is successfully pushed to it.

13. CD Phase Begins: Deploy to Kubernetes on AWS EKS

Goal:

Deploy the updated application using ArgoCD to an Amazon Elastic Kubernetes Service (EKS) cluster with proper ingress handling.

Step 1: Create EKS Cluster

This step should have been completed in Assignment 14. An example command used to create the cluster is:

```
Bash

eksctl create cluster --name testing-cluster ...

```

Step 2: Set Up Ingress with AWS Load Balancer Controller

Official Documentation Reference:

For detailed and up-to-date instructions, refer to the AWS Docs: Install AWS Load Balancer Controller.

Step-by-Step Setup for Load Balancer Controller

1. Create OIDC Provider

Execute the following command using PowerShell as Administrator:

```
Bash

eksctl utils associate-iam-oidc-provider \
    --region <your-region> \
    --cluster testing-cluster \
    --approve

```

Replace `<your-region>` with your AWS region.

2. Create IAM Role for the Load Balancer Controller

1. Go to the AWS Console, navigate to IAM, then Roles, and click "Create Role".
2. Select "Web Identity" as the trusted entity.
3. For "OIDC Provider", select the OIDC provider associated with your EKS cluster.
4. Set "Audience" to sts.amazonaws.com.
5. Attach the managed policy: AmazonEKSLoadBalancerControllerPolicy.
6. Name the role: AmazonEKSLoadBalancerControllerRole.
7. Click "Create Role".
8. **Copy the Role ARN** as you will need it in the next step.

3. Configure Kubernetes Service Account

Copy the content of the service-account.yaml file from the manifest → service and ingress folder.

- Replace the placeholder ARN within this file with the IAM Role ARN you copied in the previous step.
  Apply the service account using:

```
Bash

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

Verify that the service account has been created:

```
Bash

kubectl get sa -n kube-system
```

You should see an entry for: aws-load-balancer-controller.

4. Install AWS Load Balancer Controller using Helm

If Helm is not already installed on your system, please install it first.

Commands (from AWS documentation):

```
Bash

helm repo add eks https://aws.github.io/eks-charts
helm repo update
helm install aws-load-balancer-controller \
   eks/aws-load-balancer-controller \
   -n kube-system \
   --set clusterName=testing-cluster \
   --set serviceAccount.create=false \
```

```
  --set region=<your-region> \
  --set vpcId=<your-vpc-id> \
  --set serviceAccount.name=aws-load-balancer-controller
```

Replace `<your-region>` and `<your-vpc-id>` with your actual AWS region and VPC ID. You can obtain your VPC ID by running `aws eks describe-cluster --name testing-cluster`.

AWS DevOps Class Notes (Final Part): Testing, Manual Curl & ArgoCD Setup

14. Verify Load Balancer Controller Installation

Confirm the successful deployment of the Load Balancer Controller:

```
Bash

kubectl get deploy -n kube-system
```

You should see a deployment named: `aws-load-balancer-controller`.

15. Pull the Latest Code to Local

Ensure your local repository is up-to-date with the latest changes from GitHub:

```
Bash

git pull origin main
```

16. Apply Kubernetes Manifests (Deploy)

Apply your application's Kubernetes manifests to the EKS cluster:

```
Bash

kubectl apply -f deploy/
```

Now, check the status of your application pods:

```
Bash

kubectl get pods
```

You should see your application pods running (e.g., `docker-demo-deployment-*`).

17. Manual Testing with a Test Pod

Create a temporary test pod:

```
Bash

kubectl run test-nginx --image=nginx --restart=Never -it --rm -- bash
```

Get the ClusterIP of your deployed service:

```
Bash

kubectl get svc
```

Example output:

```
docker-demo-service   ClusterIP   10.0.182.122   80/TCP
```

Inside the `test-nginx` pod, test using curl:

```
Bash

curl http://10.0.182.122/
```

You should receive the HTML output from your deployed container, indicating the service is reachable internally.

18. Setting Up ArgoCD for CD

Reference: Official ArgoCD Install Docs

Install ArgoCD

```
Bash

kubectl create namespace argocd
kubectl apply -n argocd -f
https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

Wait for the ArgoCD pods to be ready before proceeding:

```
Bash
```

```
kubectl get pods -n argocd
```

You should see pods such as: `argocd-server`, `argocd-repo-server`, `argocd-application-controller`, etc.

Expose ArgoCD Dashboard via LoadBalancer

First, get the current service information for `argocd-server`:

```
Bash

kubectl get svc -n argocd
```

You will likely see:

```
PostgreSQL & PL/pgSQL

NAME            TYPE        CLUSTER-IP     EXTERNAL-IP    PORT(S)
argocd-server   ClusterIP   10.0.XXX.XXX                  80/TCP
```

Now, edit the `argocd-server` service to change its type to `LoadBalancer`:

```
Bash

kubectl edit svc argocd-server -n argocd
```

This command will open a YAML editor (often in Notepad on Windows). Change the line:

```
YAML

type: ClusterIP
```

to:

```
YAML

type: LoadBalancer
```

Save and close the editor.

Run `kubectl get svc -n argocd` again:

```
Bash

kubectl get svc -n argocd
```

You will now see an `EXTERNAL-IP` assigned to `argocd-server`.

Access ArgoCD in Browser

- Copy the `EXTERNAL-IP` of `argocd-server` from the previous `kubectl get svc` output.
- Paste this IP address into your web browser.
- The ArgoCD UI login page should load.

19. Accessing ArgoCD Dashboard & Login

1. Get ArgoCD Admin Password

Retrieve the temporary administrator password for ArgoCD:

```
Bash

kubectl get secret argocd-initial-admin-secret -n argocd -o jsonpath="{.data.password}" | base64 --decode
```

This command will output your temporary ArgoCD admin password. Save it securely.

2. Open the ArgoCD Dashboard

- Open your web browser.
- Paste the `EXTERNAL-IP` of the `argocd-server` service (obtained from `kubectl get svc -n argocd`).
- Enter the username: `admin`
- Paste the decoded password obtained in the previous step.

20. ArgoCD GitHub Repo & Application Setup

1. Connect GitHub Repo to ArgoCD

Inside the ArgoCD Dashboard:

- Go to `Settings` → `Repositories`.
- Click "Connect Repo".
- Choose the connection type: `HTTP/HTTPS`.
- Fill in the details:
    - Repo URL: `https://github.com/YOUR_USERNAME/docker_demo`
    - Leave username/password fields empty if your repository is public.
- Click "Connect".

2. Create ArgoCD Application

1. Go to `Applications` → `New App`.
2. Fill in the following fields:

Field	Value
Application Name	`devops_demo`
Project	`default`
Sync Policy	`Automatic`
Repository URL	Your connected GitHub repository URL
Revision	`main`
Path	`deploy`
Cluster URL	Use default (ArgoCD should detect your EKS cluster)
Namespace	`default`

- Click "Create".

21. Ingress Setup (Exposing App to Public)

This section covers the likely steps taken by the instructor to expose the application publicly via an AWS Application Load Balancer.

1. Edit `ingress.yaml`

- The `ingress.yaml` file needs to be configured with specific AWS-related annotations and possibly path-based routing rules. These are crucial for the AWS Application Load Balancer to correctly route incoming traffic to your Kubernetes service.
- You can refer to the official AWS Ingress Controller documentation or Subhasis Sir's repository for a ready-made `ingress.yaml` example. Ensure you include necessary subnet IDs and ALB annotations.

2. Apply the Ingress Manifest

Apply the configured Ingress resource to your Kubernetes cluster:

```
Bash

kubectl apply -f ingress.yaml
```

Check the status of your Ingress:

```
Bash

kubectl get ingress
```

This command will display the DNS address of the AWS Application Load Balancer once it has been provisioned and is ready.

3. Go to AWS Console to Verify Load Balancer

- Navigate to the AWS Management Console, then to `EC2`, and select `Load Balancers`.
- Click on your Application Load Balancer (ALB).
- Review the "Listeners" and "Rules" tabs to ensure proper routing configuration.
- Check the "Target Group" health and activity to confirm your EKS pods are registered and healthy.
- Once the ALB is ready and healthy, copy its DNS Name.
- Paste the DNS Name into your web browser – you should now see your deployed application accessible publicly!

22. Test Automatic CI/CD via Code Push

Change Something (e.g., `index.html`) in VS Code

Make a visible change to your application's `index.html` file, for example:

```
HTML

Devops CI/CD Yaaay!
```

Then in Terminal:

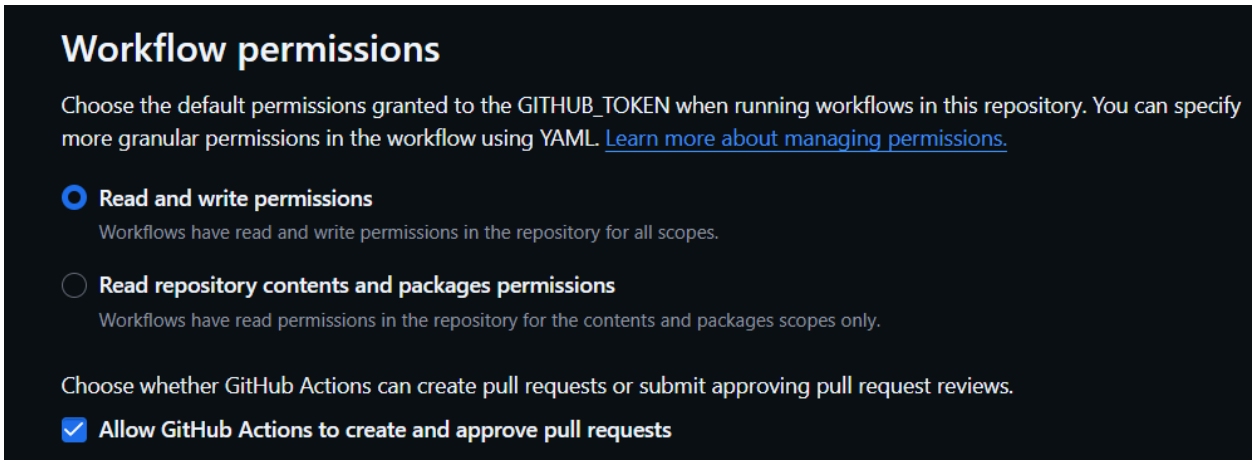
Commit and push your changes to the GitHub repository:

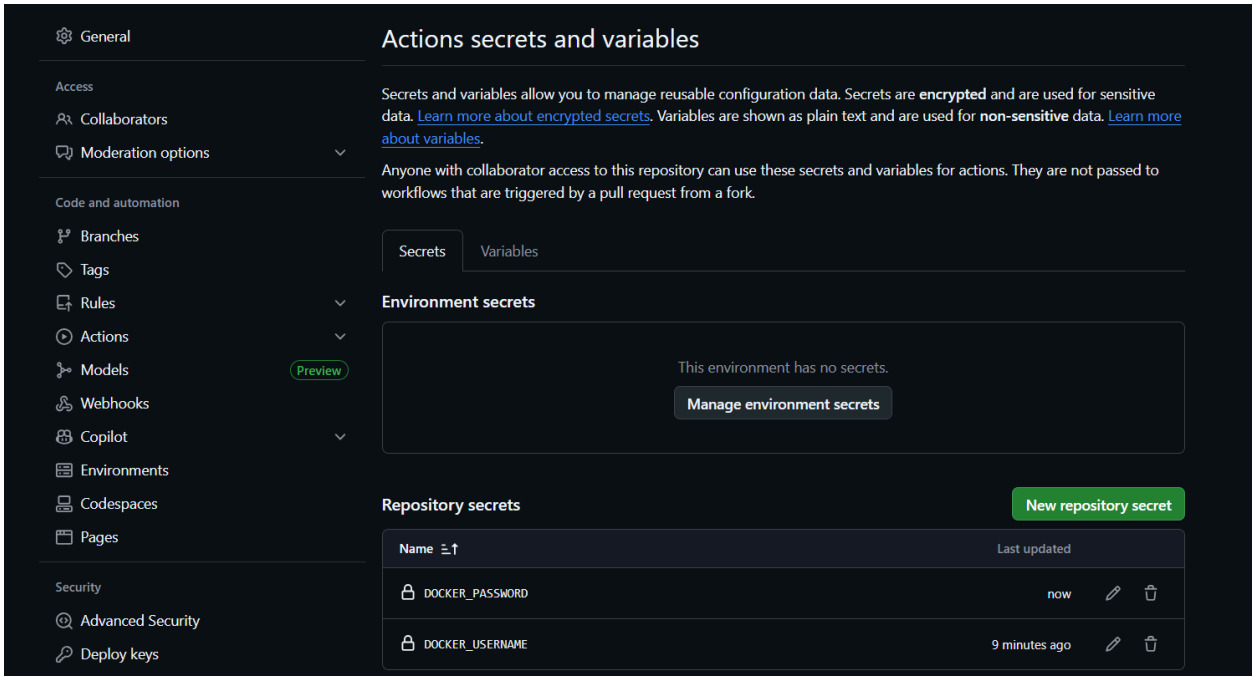
```
Bash

git add .
git commit -m "Updated index.html for CI/CD demo"
git push origin main
```

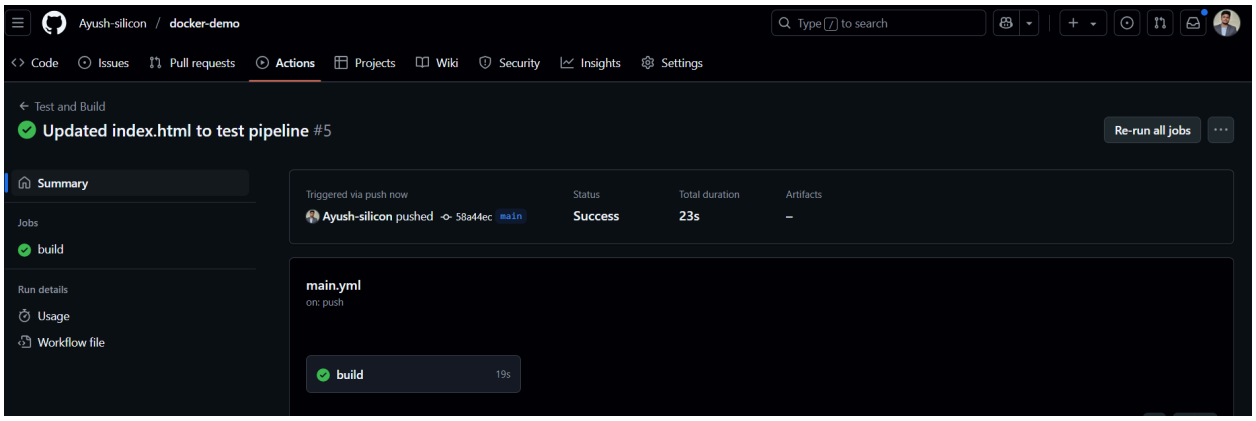
What Happens Next:

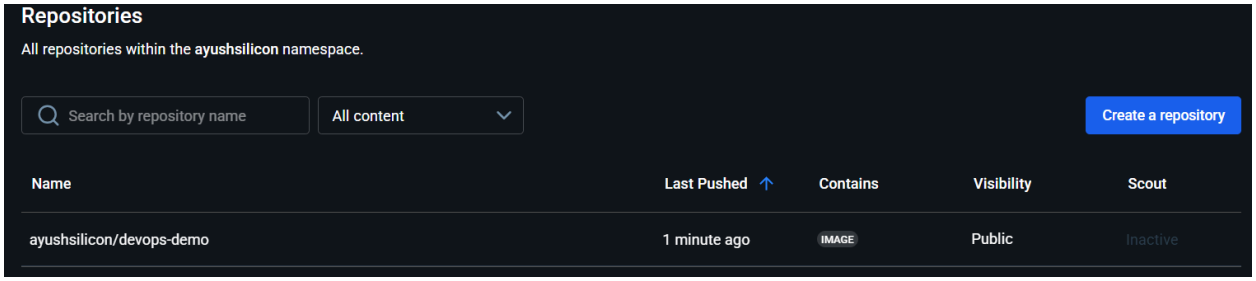
1. **GitHub Actions workflow is triggered:**
    - It will automatically rebuild the Docker image with your latest changes.
    - It will push the updated image to DockerHub.
    - It will update the Kubernetes manifests in your GitHub repository (if your workflow is configured to do so).
2. **ArgoCD detects changes in Git repo:**
    - ArgoCD, continuously monitoring your Git repository, will detect the updated YAML manifests.
    - It will automatically pull these updated manifests.
    - It will apply the changes to your EKS cluster, initiating a new deployment of your application.
3. **Refresh the ALB URL in browser:**
    - After a short period (allowing for the deployment to complete), refresh the Application Load Balancer (ALB) URL in your web browser.

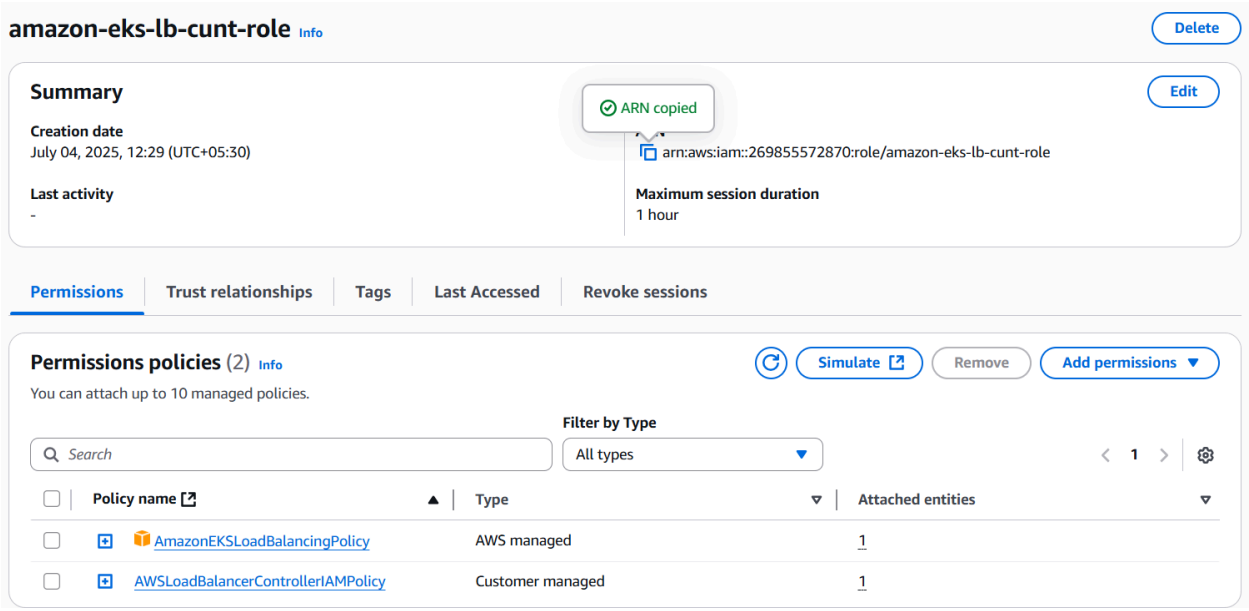
You will now see the updated HTML content live, demonstrating the complete end-to-end CI/CD pipeline!




General
Access
Collaborators
Moderation options
Code and automation
Branches
Tags
Rules
Actions
Models
Preview
Webhooks
Copilot
Environments
Codespaces
Pages
Security
Advanced Security
Deploy keys
Actions secrets and variables
Secrets and variables allow you to manage reusable configuration data. Secrets are encrypted and are used for sensitive data. Learn more about encrypted secrets. Variables are shown as plain text and are used for non-sensitive data. Learn more about variables.
Anyone with collaborator access to this repository can use these secrets and variables for actions. They are not passed to workflows that are triggered by a pull request from a fork.
Secrets
Variables
Environment secrets
This environment has no secrets.
Manage environment secrets
Repository secrets
New repository secret
Name
Last updated
DOCKER_PASSWORD
now
DOCKER_USERNAME
9 minutes ago


Ayush-silicon / docker-demo
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
Test and Build
Updated index.html to test pipeline #5
Re-run all jobs
Summary
Jobs
build
Run details
Usage
Workflow file
Triggered via push now
Ayush-silicon pushed 58a44ec main
Status
Success
Total duration
23s
Artifacts
–
main.yml
on: push
build
19s


Repositories
All repositories within the ayushsilicon namespace.
Search by repository name
All content
Create a repository
Name
Last Pushed
Contains
Visibility
Scout
ayushsilicon/devops-demo
1 minute ago
IMAGE
Public
Inactive




```
PS C:\Users\Ayush Singh\Downloads\DevOps-Assignment\devops-demo> kubectl apply -f svc.yaml
serviceaccount/aws-load-balancer-controller created
PS C:\Users\Ayush Singh\Downloads\DevOps-Assignment\devops-demo> kubectl get sa -n kube-syst
NAME                                 SECRETS   AGE
attachdetach-controller              0         2d18h
aws-cloud-provider                   0         2d18h
aws-load-balancer-controller         0         25s
aws-node                             0         2d18h
```