Deploying a NodeJS Application on ECS using GitHub Workflows Pipeline (Task 12)

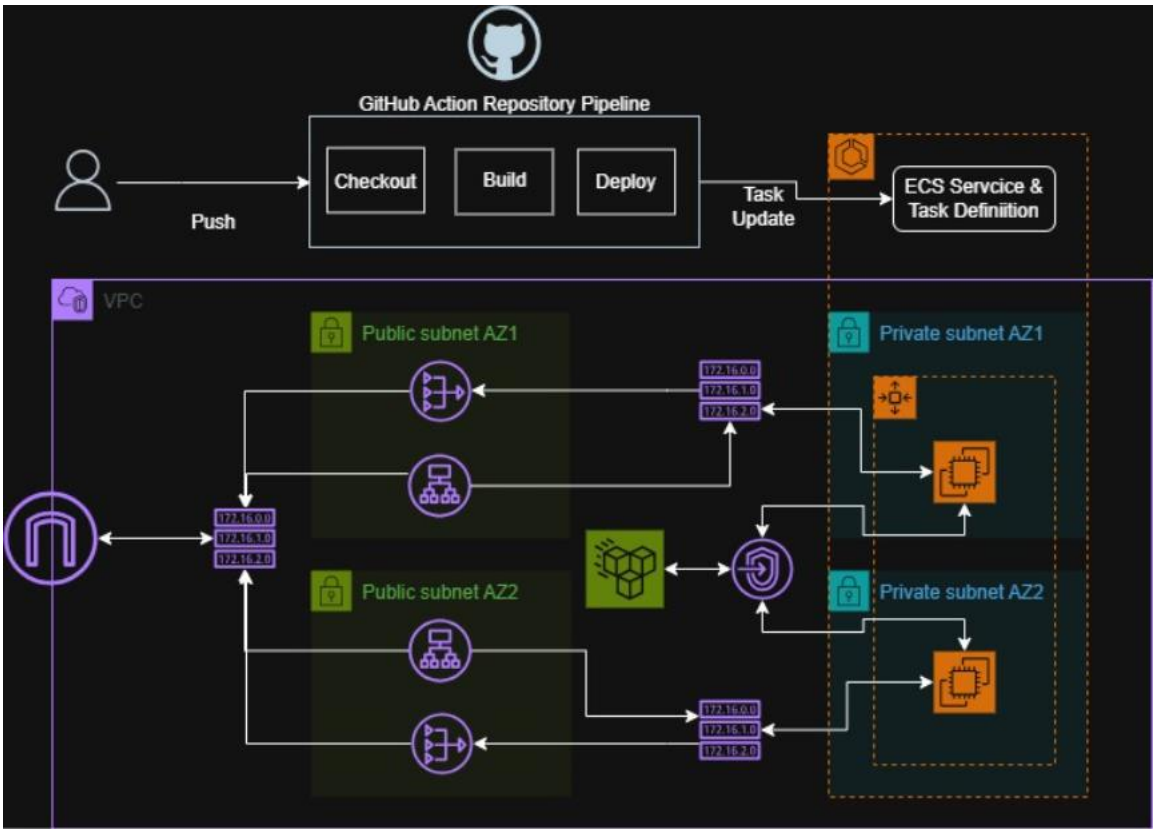
Zaeem Attique Ashar

Cloud Intern

Task Description:

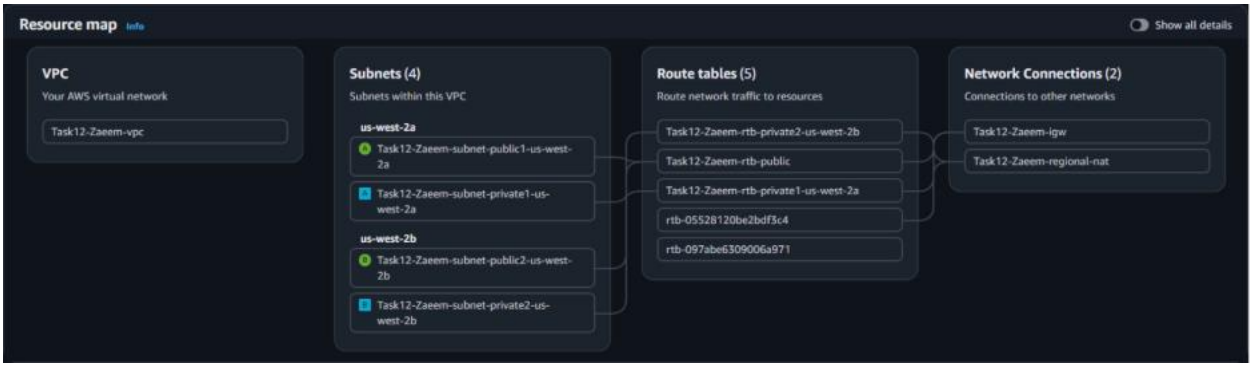
This project involves deploying a simple Node.js application using AWS Elastic Container Service. The application will be turned into a Docker Image using a Docker File stored in the repository. This image will be pushed to the ECR. An ECS Fargate Cluster and Task will be setup to run the application. After pulling the image from the ECR, it will be deployed on the ECS Fargate Cluster.

Architecture Diagram:



Task12.1: Create basic networking infrastructure

- Create and configure a VPC
    - CIDR Block: 10.0.0.0/16
- Create and configure Subnets
    - Public Subnet A (us-west-2a), CIDR: 10.0.1.0/24
    - Private Subnet A (us-west-2a), CIDR: 10.0.2.0/24
    - Public Subnet B (us-west-2b), CIDR: 10.0.3.0/24
    - Private Subnet A (us-west-2a), CIDR: 10.0.4.0/24
- Create and configure NAT Gateways
    - NAT Gateway A in Public Subnet A
    - NAT Gateway B in Public Subnet B
- Create and configure Internet Gateway
    - Create and attach to the project's VPC
- Create and configure Route Tables
    - Public Route Table, Outbound rule: 0.0.0.0/0 -> IGW, attach to Public SN A&B
    - Private Route Table A, Outbound Rule: 0.0.0.0/0 -> NGW attach to Private SN A
    - Private Route Table B, Outbound Rule: 0.0.0.0/0 -> NGW attach to Private SN B



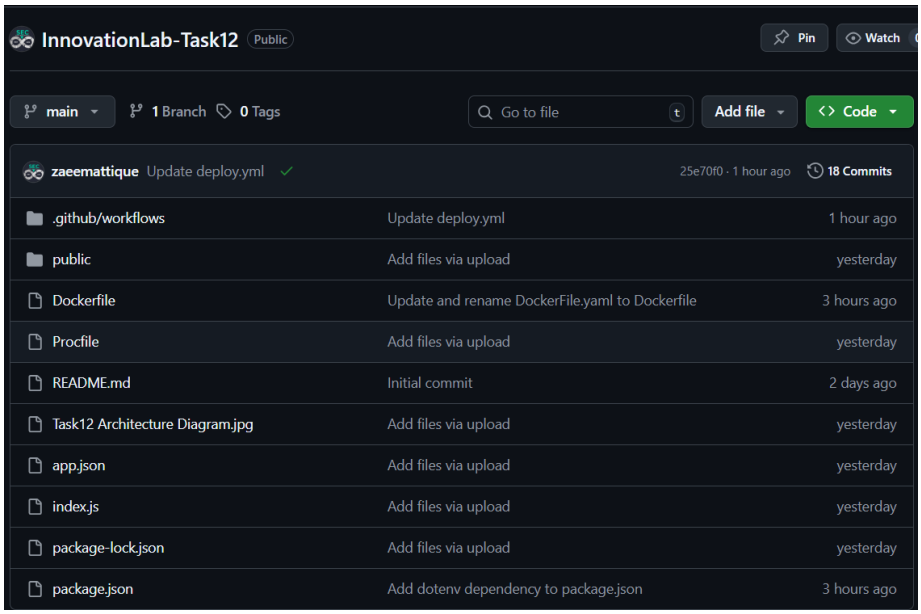
Task12.2: Prepare NodeJS Application and Dockerfile

- Remove node packages as they will be installed later in the build stage of the pipeline. Node packages are in the node_modules directory of the application.
- Write the Dockerfile which includes the steps to build the application. Here are the steps:
    - Uses node:18-alpine as the base image for the application image.
    - Set usr/src/app as the working directory.
    - Copy all the package file onto the node.
    - Use npm install to install all dependencies for the application.
    - Copy all the files from the app directory to the image.
    - Expose the container port 5000 for HTTP communication.
    - Use the “node” or “index.js” command to run the application.

Code Blame 14 lines (7 loc) · 159 Bytes

```
1   FROM node:18-alpine
2
3   WORKDIR /usr/src/app
4
5   COPY package*.json ./
6
7   RUN npm install --production
8
9   COPY . .
10
11  EXPOSE 5000
12
13
14  CMD ["node", "index.js"]
```

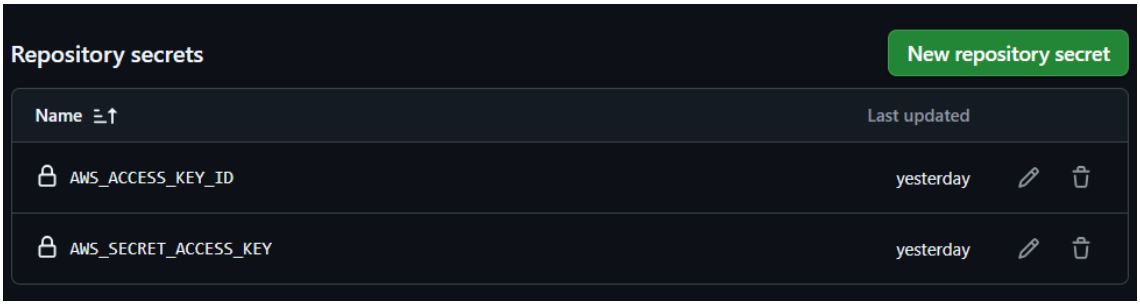
- Upload the rest of the application code to the GitHub repository for the dockerfile to use.



Task12.3: Configure AWS Access Keys as Secrets.

To safely store and access the AWS Access Keys to the use account and keep them safe from showing up in the logs, we must set them as secret variables in the GitHub repository. These keys can be used in the long term.

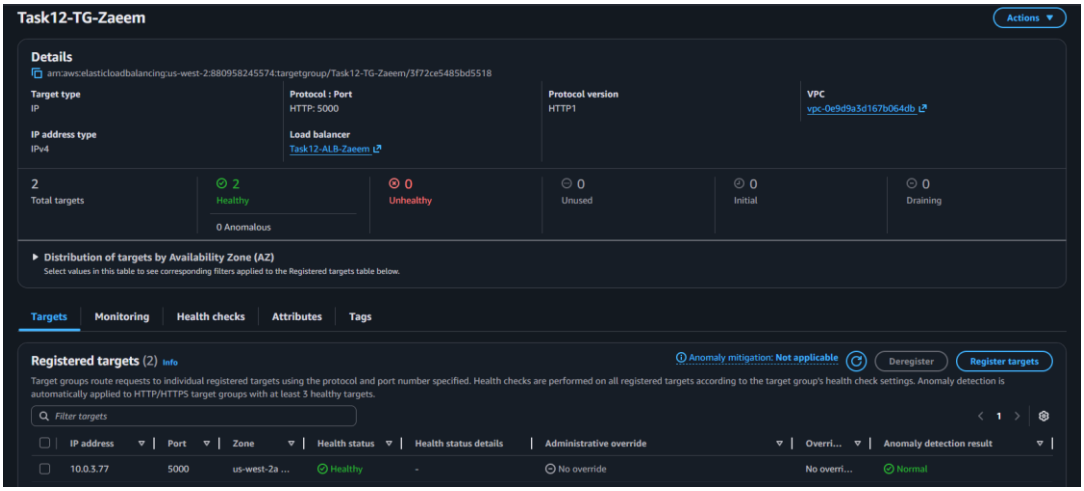
- Head over to the GitHub repository home page and navigate over to settings option on the top bar.
- Scroll down and find the Secrets and Variables option in the left panel and select Actions
- Scroll down and click on the 'New Repository Secret' button.
- Enter the name of the secret variable and then enter the secret value and save.
- Do the same for both the Access Key and the Secret Access Key.



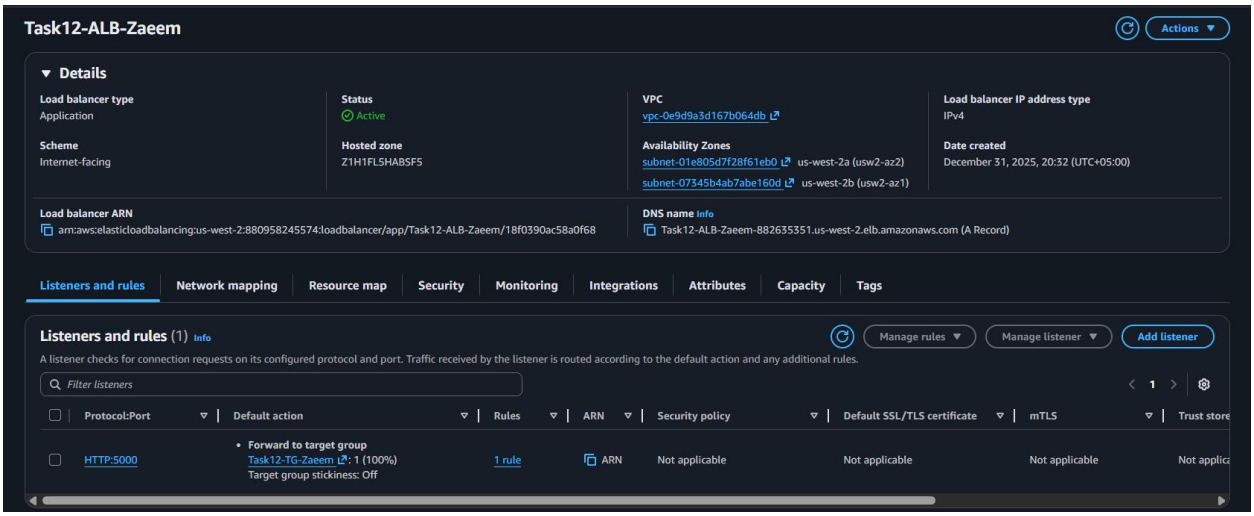
Task12.4: Create Target Group and Load Balancer

Now back to aws, we need to create an Application Load Balancer and a Target group that we will pass over to ECS when we create the service.

- Head over to the Target Group panel in the EC2 dashboard.
- Select Create Target Group and configure the following:
    - Target Type: IP addresses
    - Name: Task12-TG-Zaeem
    - Protocol: HTTP, Port: 5000
    - IP Type: IPv4
    - Protocol Version: v1
    - Health Checks: Default
    - Click next twice and then create Target Group

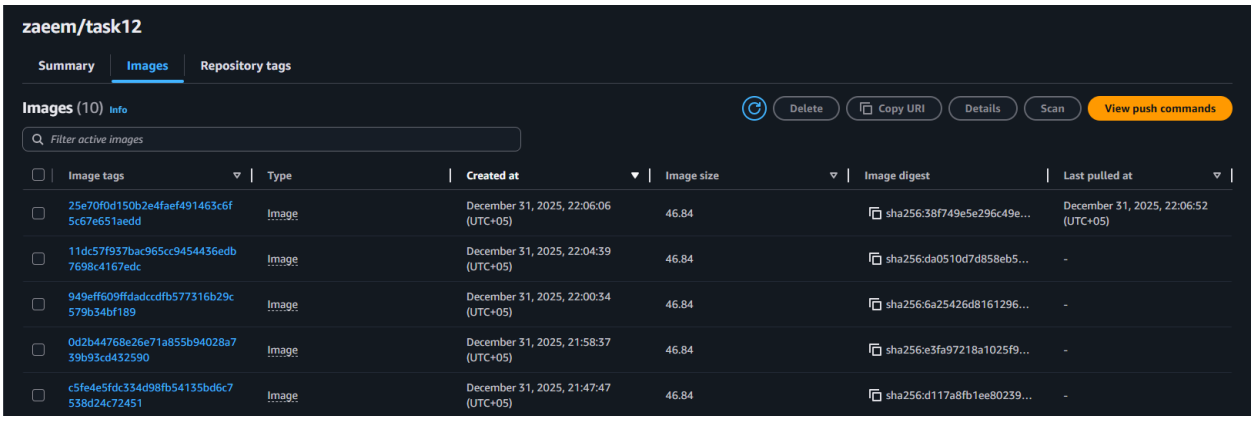


- Head over to the Load Balancer panel in the ECS dashboard.
- Select create Load Balancer and configure the following:
    - Load Balancer type: Application
    - Name: Task12-ALB-Zaeem
    - Scheme: Internet Facing
    - Load balancer IP Type: IPv4
    - VPC: Task12-VPC-Zaeem (as previously created)
    - Select Both Availability Zones and select the public subnets in each.
    - Create a new SG with allowing inbound traffic at port 5000 and attach to ALB
    - Listner Protocol: HTTP, Listener Port: 5000
    - Default action: forward traffic to target group
    - Select the target group previously created
    - Click on Create Load Balancer



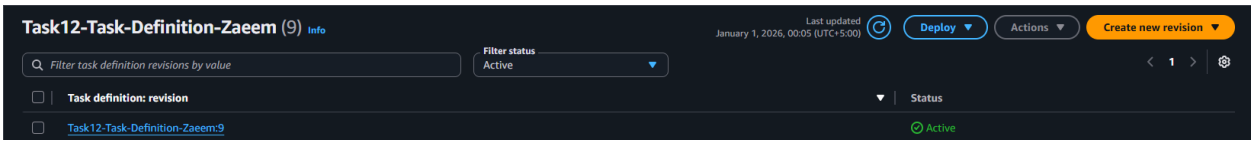
Task12.5: Create ECR Repository for the Docker Image

- Head over to the ECR Dashboard and click on create repository.
- Name: task12/zaeem
- Image Tag Mutability: Mutable
- Encryption Config: AES-256
- Click on the create button.

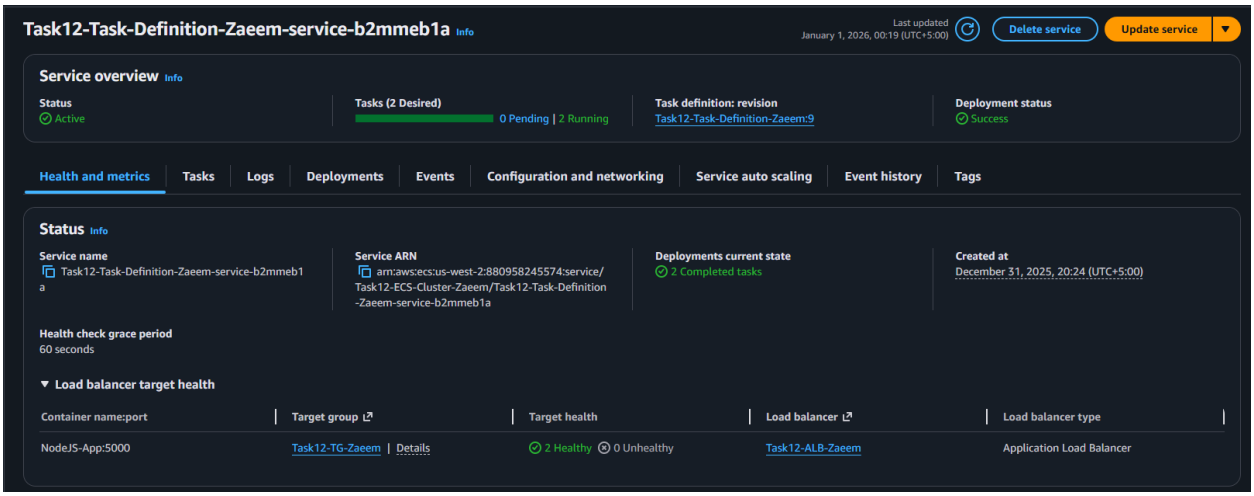


Task12.6: Create ECS Cluster, Task Definition and Service to deploy the application

- Head over to the ECS Dashboard and click on create cluster.
- Configure the following:
  - Name: Task12-ECS-Cluster-Zaeem
  - Infrastructure: Fargate only
  - Under monitoring, set the level of observability as container insights
  - Click on create
- Now head over to the Task Definitions panel in the left.
- Click on creating a new task definition button.
- Configure the following:
  - Task definition family name: Task12-ECS-TD-Zaeem
  - Launch type: AWS Fargate
  - OS Architecture: Default
  - Task Size: 2vCPU and 4GB Memory
  - Select task roles and execution roles.
  - Container name: NodeJS-App
  - Select Image URI from the created ECR repo
  - Port mapping: Container port 5000, Protocol TCP, Port name: App-port, HTTP
  - Click on Create



- Now head over to the services panel in the cluster we created previously and click on the create service button.
- Configure the following:
    - Task definition family: Task12-ECS-TD-Zaeem
    - Compute options: Launch Type, Fargate, Latest
    - Scheduling Strategy: Replica, Desired Tasks: 2
    - Deployment Option: Rolling Updates
    - Under load balancing: Load Balancer Type: Application
    - Select the container name from the drop-down menu.
    - Use an existing Load Balancer and choose the previously created ALB.
    - Use an existing Target Group and select the previously created TG.
    - Click on create service.



Task12.7: Creating GitHub Workflow

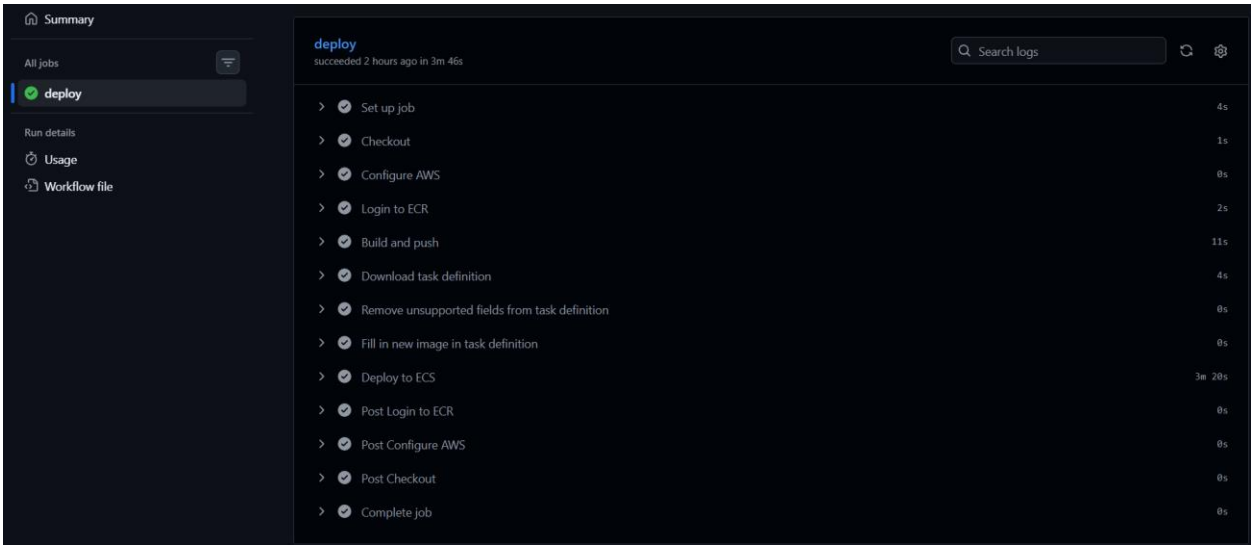
This will be a pipeline that will build and deploy the application code that we have uploaded into the repository by making use of the aws Credentials and the Dockerfile.

- Select the create new file button in the github repository and set the new file name as follows: '.github/workflows/deploy.yml '.
- Write the YAML code of the pipeline which includes the configurations:
    - Name: Deploy to ECS
    - Trigger on push to main branch of the repository
    - Set the following env variables:
        - AWS_REGION: us-west-2
        - ECR_REGISTRY: 880958245574.dkr.ecr.us-west-2.amazonaws.com
        - ECR_REPOSITORY: zaeem/task12
        - ECS_SERVICE: Task12-Task-Definition-Zaeem-service-b2mmeb1a
        - ECS_CLUSTER: Task12-ECS-Cluster-Zaeem
        - CONTAINER_NAME: NodeJS-App
    - Create only one job named 'Deploy' that runs on ubuntu-latest runner.
    - First step checks out code and copies it to the runner using checkout@v3
    - Second, use configure-aws-credentials@v2 to use the keys to config aws cli
    - Third, use amazon-ecr-login@v1 to login to the ECR to push the image later
    - Fourth stage uses the dockerfile to build the app image and push it to ECR
    - Fifth stage collects the data of the Task Definition and turns it into JSON config
    - Sixth stage uses the amazon-ecs-render-task-definition@v1 action to removes unsupported fields and replaces the old image name with the fresh, newly built image which will now be used by the service
    - The seventh and last stage uses amazon-ecs-deploy-task-definition@v1 to deploy the new task definition onto the running ECS Service. The service deploys the new TD as configured (rolling updates).

```yaml
name: Deploy to ECS

on:
  push:
    branches: [main]

env:
  AWS_REGION: us-west-2
  ECR_REGISTRY: 880958245574.dkr.ecr.us-west-2.amazonaws.com
  ECR_REPOSITORY: zaeem/task12
  ECS_SERVICE: Task12-Task-Definition-Zaeem-service-b2mmeb1a
  ECS_CLUSTER: Task12-ECS-Cluster-Zaeem
  CONTAINER_NAME: NodeJS-App

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout
      uses: actions/checkout@v3

    - name: Configure AWS
      uses: aws-actions/configure-aws-credentials@v2
      with:
        aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
        aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        aws-region: ${{ env.AWS_REGION }}

    - name: Login to ECR
      id: login-ecr
      uses: aws-actions/amazon-ecr-login@v1

    - name: Build and push
      id: build-image
      env:
        ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
        IMAGE_TAG: ${{ github.sha }}
      run: |
        docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
        docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
        echo "image=$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG" >> $GITHUB_OUTPUT

    - name: Download task definition
      run: |
        aws ecs describe-task-definition --task-definition $(aws ecs describe-services --
    - name: Remove unsupported fields from task definition
      run: |
        jq 'del(.enableFaultInjection)' task-definition.json > task-definition-clean.json
        mv task-definition-clean.json task-definition.json

    - name: Fill in new image in task definition
      id: task-def
      uses: aws-actions/amazon-ecs-render-task-definition@v1
      with:
        task-definition: task-definition.json
        container-name: ${{ env.CONTAINER_NAME }}
        image: ${{ steps.build-image.outputs.image }}

    - name: Deploy to ECS
      uses: aws-actions/amazon-ecs-deploy-task-definition@v1
      with:
        task-definition: ${{ steps.task-def.outputs.task-definition }}
        service: ${{ env.ECS_SERVICE }}
        cluster: ${{ env.ECS_CLUSTER }}
        wait-for-service-stability: true
```

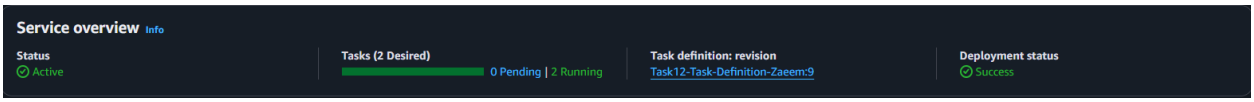
Task12.8: Triggering the GitHub Workflow

As we commit to the workflow file, a github workflow will run to implement the jobs and steps defined in the workflow file. It will output logs at each stage and output errors as well.

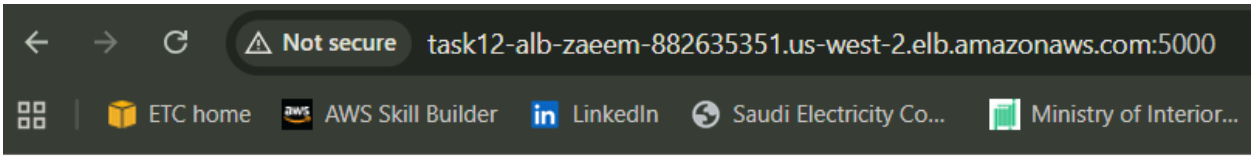


Task12.9: Testing the Deployment of the Application

- Head over to the cluster and open the service we created. There should be the desired number of tasks running that we mentioned in the Service Configuration. If not, the rolling updates have not been finished yet.



- Head over to the load balancer tab and use the ALB DNS to communicate with the frontend of the application through the browser.



- There are two ways to rollback an update that is deployed:
  - Fastest: Head over to the cluster dash and click on the update service button. In the configuration, select the previous version of the task definition and also check the force deployment button. This will update the service instantly.
  - Cleanest: Roll back to an older commit/ fix the code and the workflow will be automatically triggered upon commit. This will create a new task definition version and deploy it onto the service.

Task12.10: Problems faced and Solutions Learned

- We can wither write the infrastructure how we wish as IaC and use it inside the workflow so that when a new deployment is created, it will be according to our defined infrastructure.
- Alternatively, we can avoid using IaC by creating all the infrastructure such as cluster, task definition and service, and pulling the config into a JSON inside the workflow, modifying it at each run and deploying using that as the IaC.
- The target group of the ALB for an ECS cluster does not have any targets registered initially because it will auto register targets as they are created by Fargate.
- The package.json file must include the correct Node version and all the dependencies that are required for the application to run.