GitHub Actions Workflow Documentation for AWS ECS Deployment

Problem Statement

Configure a GitHub Actions workflow to automate the deployment of a simple web application on GitHub to AWS ECS (Elastic Container Service). The pipeline should include deployment, integration tests, and rollback functionality.

Steps Overview

1. Checkout code from GitHub repository
2. Build optimized Docker image with multistage build to serve with Nginx as a reverse proxy
3. Push image to ECR (Elastic Container Registry)
4. Deploy on ECS
5. Perform integration tests and implement rollback functionality in case of failure

Prerequisites

- A GitHub repository with the source code of the web application
- An AWS account with access to ECS, ECR, and IAM roles configured for GitHub Actions
- Docker installed locally for testing the build process
- AWS CLI configured with necessary permissions
- Required secrets added to your GitHub repository (AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, AWS_ECR_REGISTRY, AWS_REGION, ECS_CLUSTER_NAME, ECS_SERVICE_NAME, ECS_TASK_DEFINITION)

Setting Up the Environment

Before we start, ensure you have the following:

- A GitHub repository with your web application code
- An AWS account with ECR and ECS configured
- Necessary AWS credentials and permissions(IAM)
- Node.js and npm installed for Docker image building
- Docker installed for building the image

Step 1: Pushing My local application to GitHub Repository

Step 2: Build Optimized Docker Image with Multistage Build

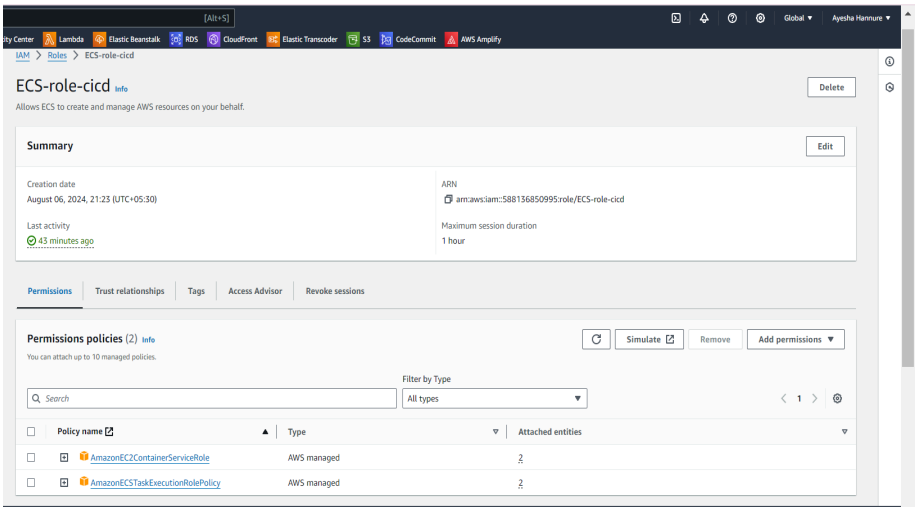
Add steps to build the Docker image using a multistage build to serve the application with Nginx.

```
FROM node:alpine AS build



WORKDIR /app



COPY package*.json ./



RUN npm install



COPY . .



RUN npm run build  # Assuming you have a build script in package.json



FROM nginx:alpine

COPY --from=build /app/build /usr/share/nginx/html



# Copy custom Nginx configuration if needed

#COPY nginx.conf /etc/nginx/conf.d/default.conf



EXPOSE 80



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

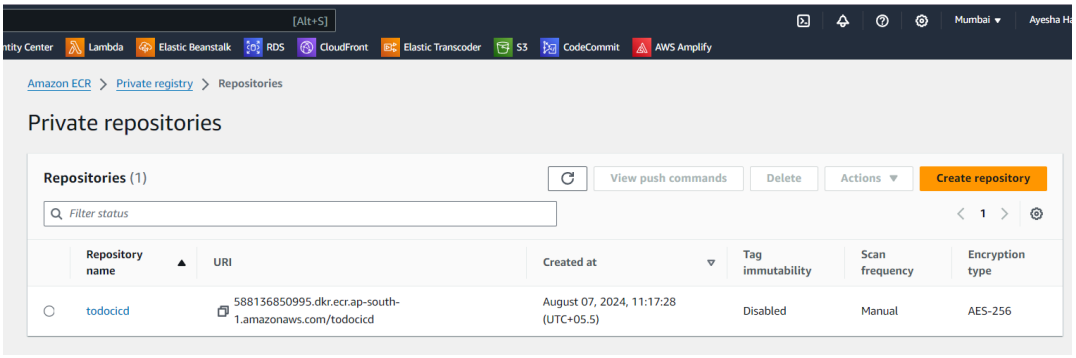
Step 3- Go to Your AWS Account :

Configure Necessary AWS credentials and permissions(IAM) for Role

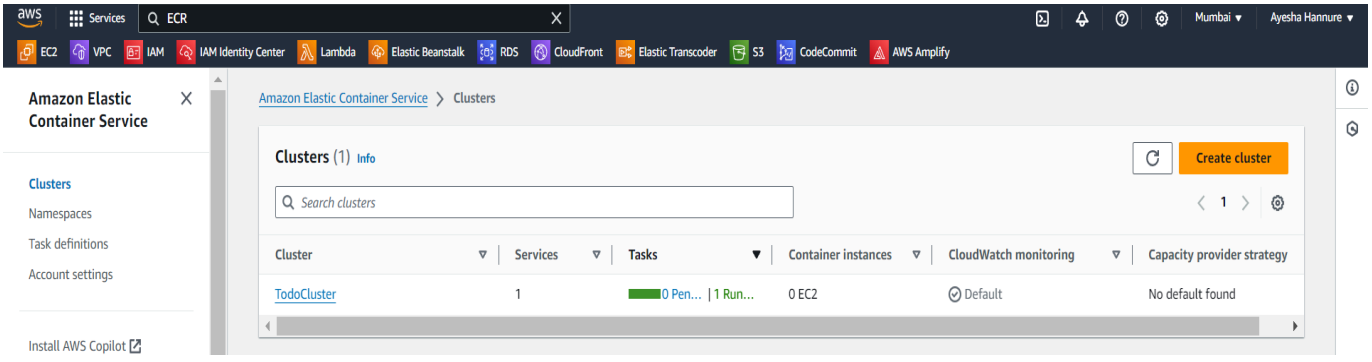
- For Role Give Permission to AmazonEC2ContainerServiceRole & AmazonECSTaskExecutionRolePolicy



Step 4: Go To ECR - Create One Repository



Step 5: Go to ECS and create 1 Cluster



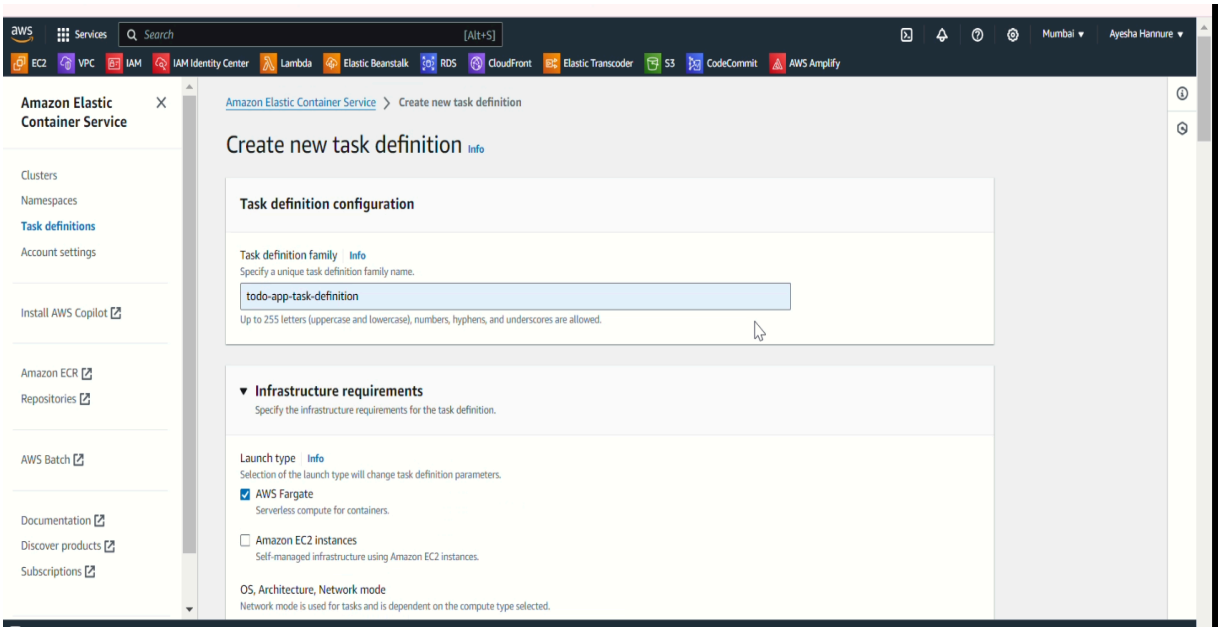
Step 6: Create Task Defination

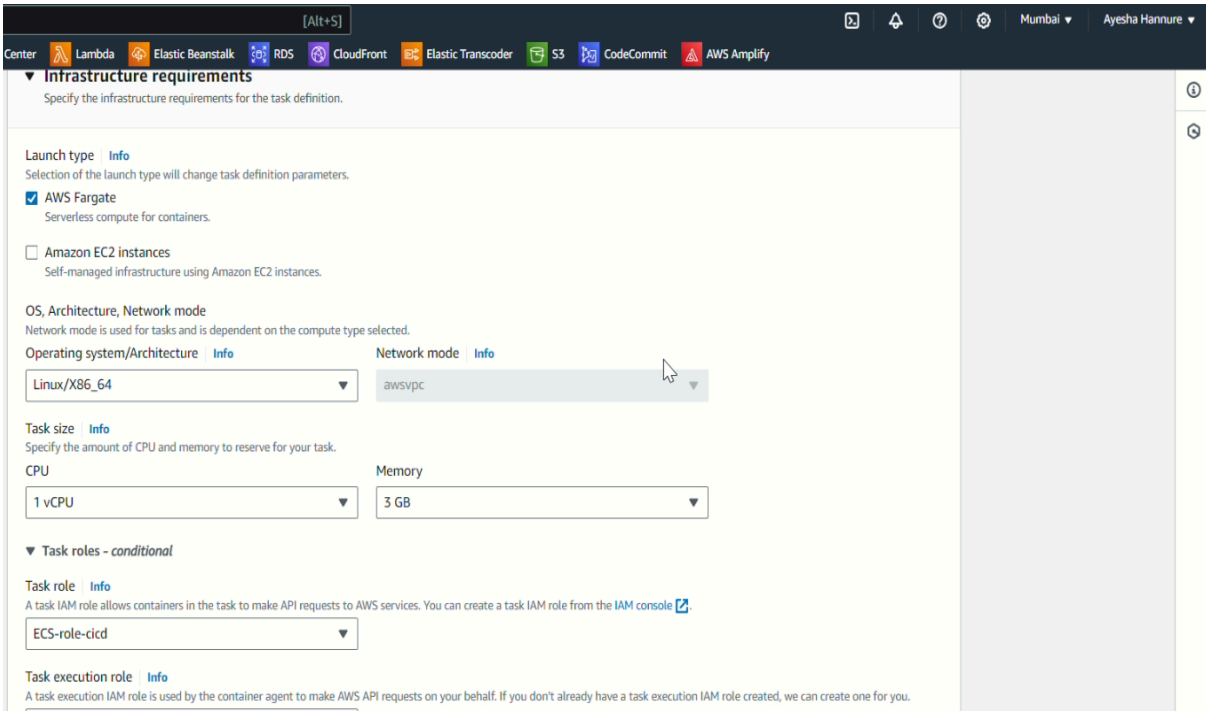
Purpose of the task Defination file -
Container Configuration,Task Configruation,Resource Allocation Resource allocation,Deployment Management and Scalling Optimization.

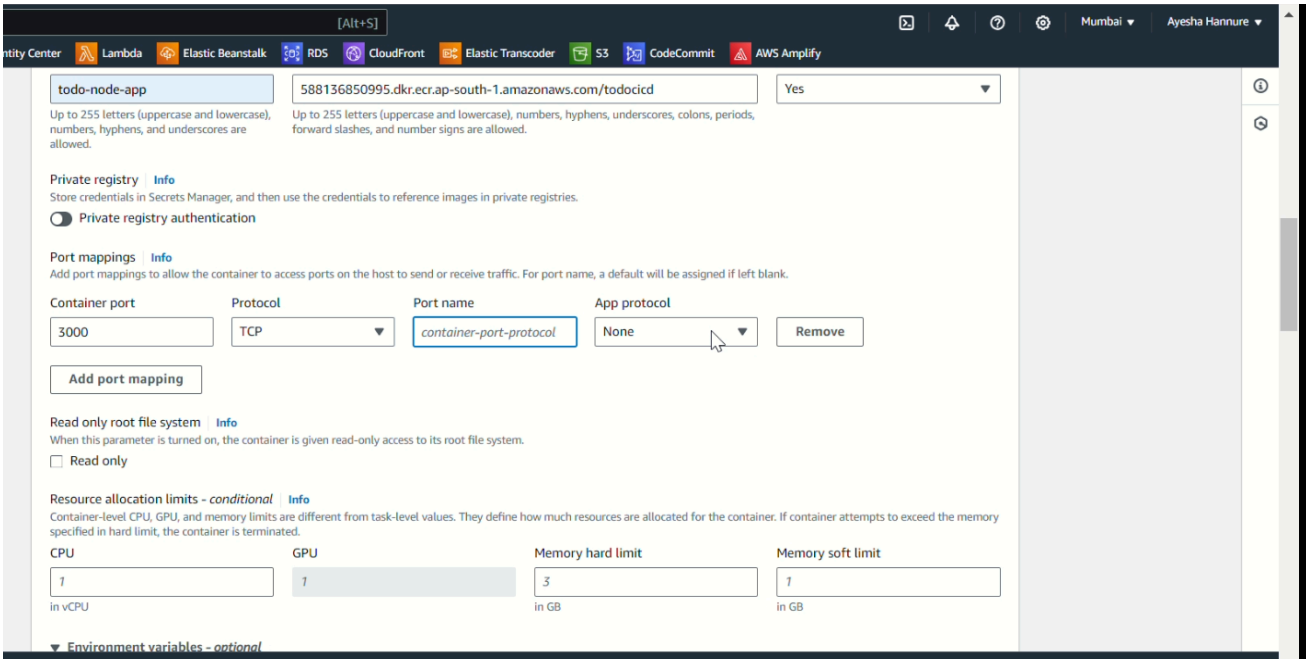
Task Defination file is a bule-print of your container. The file tells ECS how to set up and run your containers.
You can define how many instances of your container you want to run. ECS will automatically adjust the number of instances based on the workload.ou can specify how much CPU, memory, and other resources each container needs. This helps optimize your costs and performance.

Create a Task Definition

- Navigate to the ECS console and select "Task Definitions".
- Click "Create new task definition".
- Choose a task definition family name and select the launch type (Fargate or EC2).
- Under "Container definitions", add a new container.
- Provide a container name and specify the image URI (e.g., `your-account.dkr.ecr.region.amazonaws.com/your-image:latest`).
- Configure other container properties as needed (CPU, memory, port mappings, environment variables, etc.).
- Click "Create".



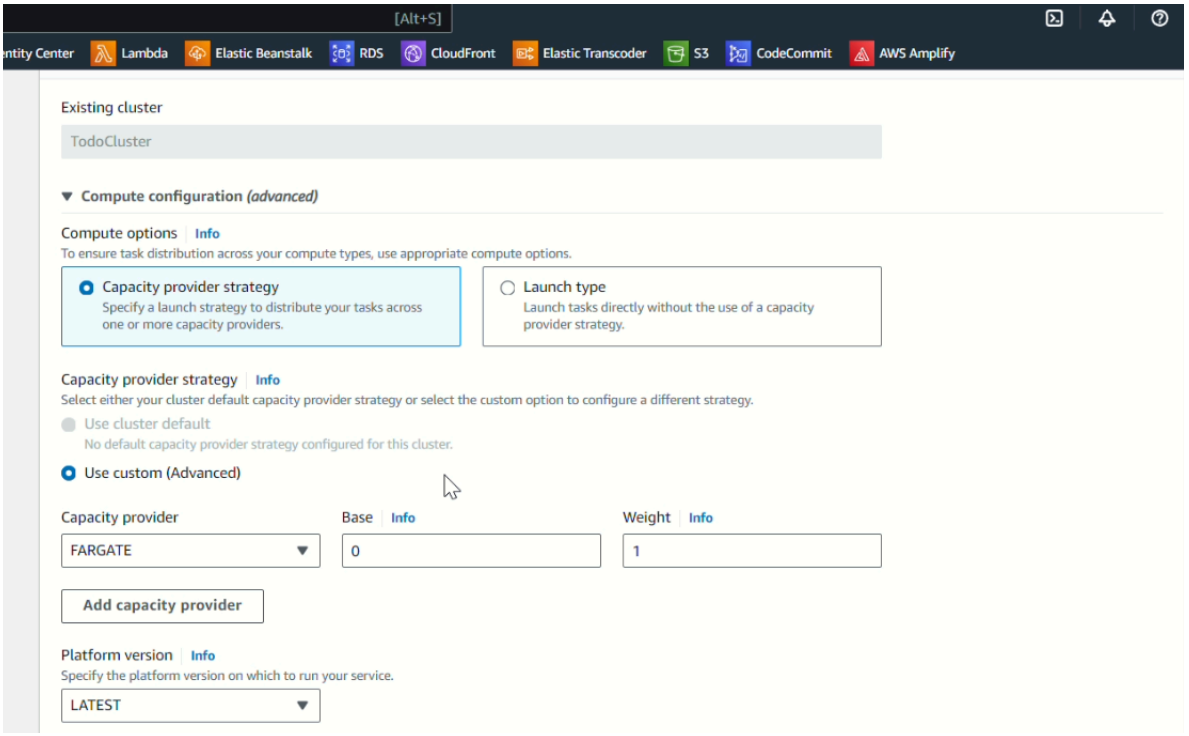


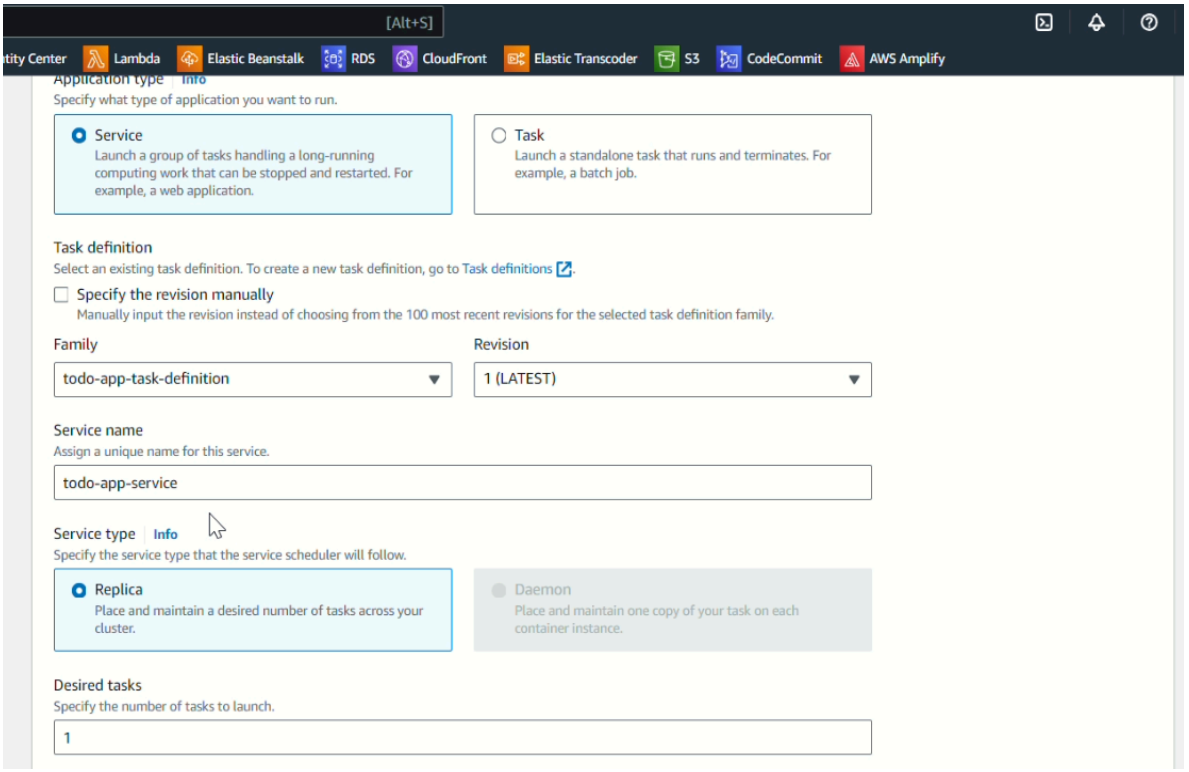


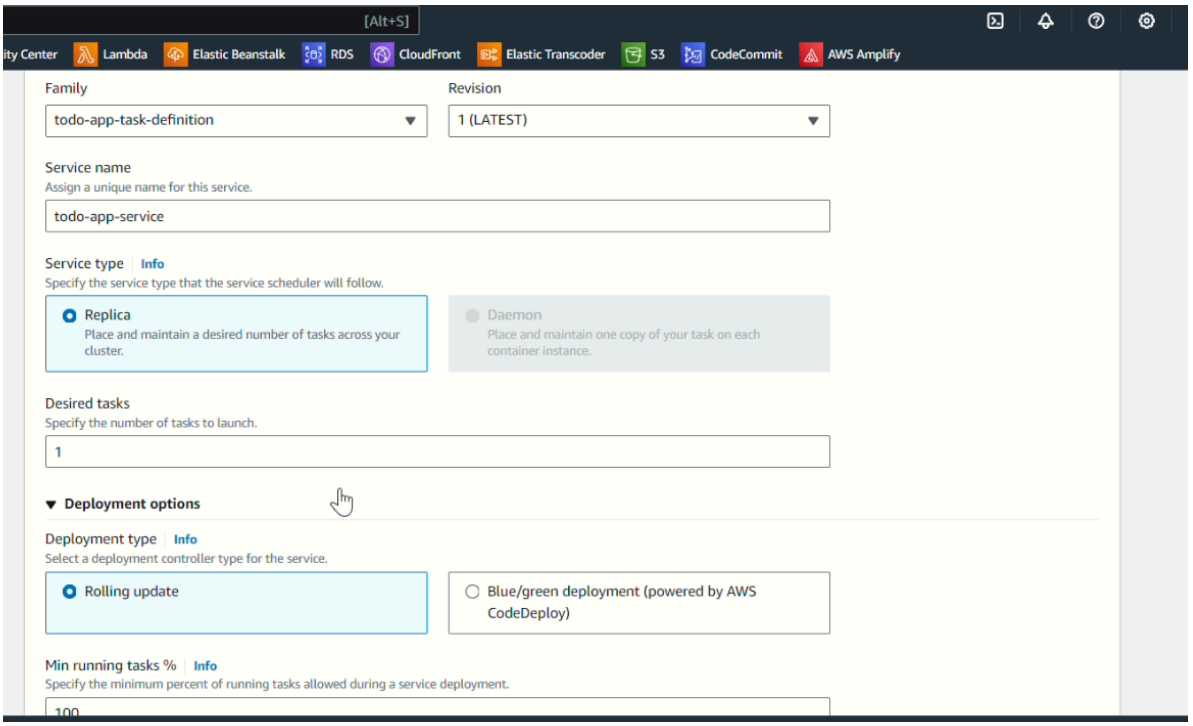
Step 7: Create Service

Create or Update a Service

- Navigate to the ECS console and select "Clusters".
- Choose your desired cluster.
- Click "Create service" or select an existing service to update.
- Provide a service name and select the task definition created in step 1.
- Configure desired number of tasks, load balancing, and other service settings.
- Click "Create" or "Save" to deploy the service.


[Alt+S]
ntity Center
Lambda
Elastic Beanstalk
RDS
CloudFront
Elastic Transcoder
S3
CodeCommit
AWS Amplify
Existing cluster
TodoCluster
Compute configuration (advanced)
Compute options Info
To ensure task distribution across your compute types, use appropriate compute options.
Capacity provider strategy
Specify a launch strategy to distribute your tasks across one or more capacity providers.
Launch type
Launch tasks directly without the use of a capacity provider strategy.
Capacity provider strategy Info
Select either your cluster default capacity provider strategy or select the custom option to configure a different strategy.
Use cluster default
No default capacity provider strategy configured for this cluster.
Use custom (Advanced)
Capacity provider
FARGATE
Base Info
0
Weight Info
1
Add capacity provider
Platform version Info
Specify the platform version on which to run your service.
LATEST





Step 7 - Checkout Code in GitHub Repository

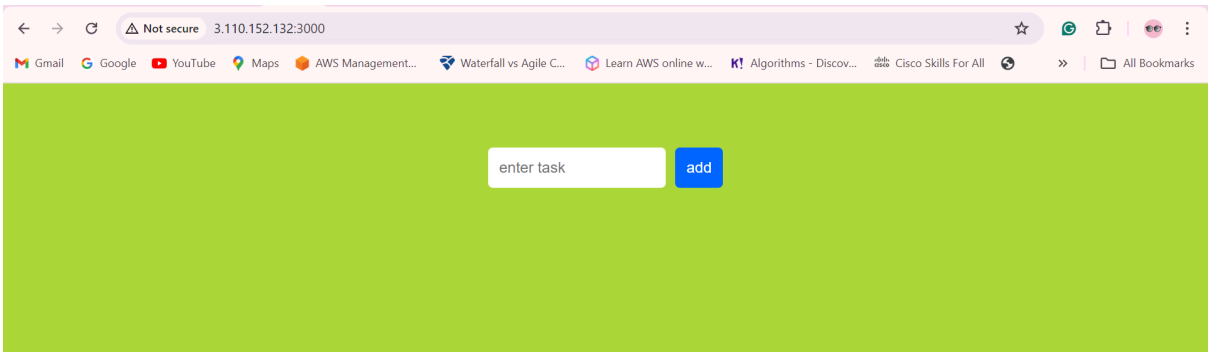
- Go to Github click on Action
- Create workflow File
- Work Flow file contain Configure AWS credential,Login to ECR ,Build Image to ECR and Push to Deploy in ECS,Run integration tests,Rollback on failure .

Here is the code snippet of my Workflow file.

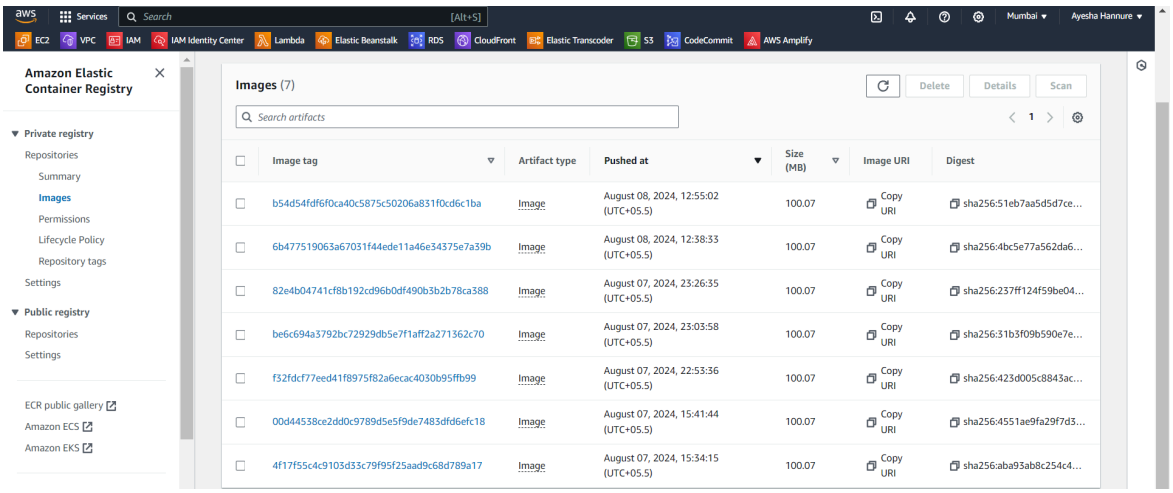
```yaml
name: CICD

on:
  push:
    branches: [ main ]

jobs:
  build-and-deploy:
    runs-on: [ ubuntu-latest ]
    steps:
      - name: Checkout source
        uses: actions/checkout@v3
      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v3
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: 'ap-south-1'
      - name: Login to Amazon ECR
        id: login-ecr
        uses: aws-actions/amazon-ecr-login@v1
        with:
          mask-password: 'true'

      - name: Build, tag, and push image to Amazon ECR
        id: build-image
        env:
          ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
          IMAGE_TAG: ${{ github.sha }}
          REPOSITORY: todocicd
        run: |
          # Build a docker container and
          # push it to ECR so that it can
          # be deployed to ECS.
          docker build -t $ECR_REGISTRY/$REPOSITORY:$IMAGE_TAG .
          docker push $ECR_REGISTRY/$REPOSITORY:$IMAGE_TAG
          echo "image=$ECR_REGISTRY/$REPOSITORY:$IMAGE_TAG" >> $GITHUB_OUTPUT
      - name: Fill in the new image ID in the Amazon ECS task definition
        id: task-def
        uses: aws-actions/amazon-ecs-render-task-definition@v1
        with:
          task-definition: todo-app-task-definition.json
          container-name: todo-node-app
          image: ${{ steps.build-image.outputs.image }}
      - name: Deploy Amazon ECS task definition
        uses: aws-actions/amazon-ecs-deploy-task-definition@v1
```

```
        with:
          task-definition: ${{ steps.task-def.outputs.task-definition }}
          service: todo-app-service
          cluster: TodoCluster
          wait-for-service-stability: true

      - name: Run integration tests
        id: integration-tests
        run: |
          curl -f http://127.0.0.1:2000
        continue-on-error: true

      - name: Rollback on failure
        if: failure()
        run: |
          # Get the previous task definition from ECS service history or version control
          PREVIOUS_TASK_DEFINITION=$(aws ecs describe-services --cluster ${{ secrets.ECS_CLUSTER }} --services ${{ secrets.ECS_SERVICE }} --query "services[0].deployments[1].taskDefinition" --output text)
          aws ecs update-service --cluster ${{ secrets.ECS_CLUSTER }} --service ${{ secrets.ECS_SERVICE }} --task-definition $PREVIOUS_TASK_DEFINITION

      - name: Notify failure
        if: failure()
        run: |
          echo "Deployment failed and rolled back"
```

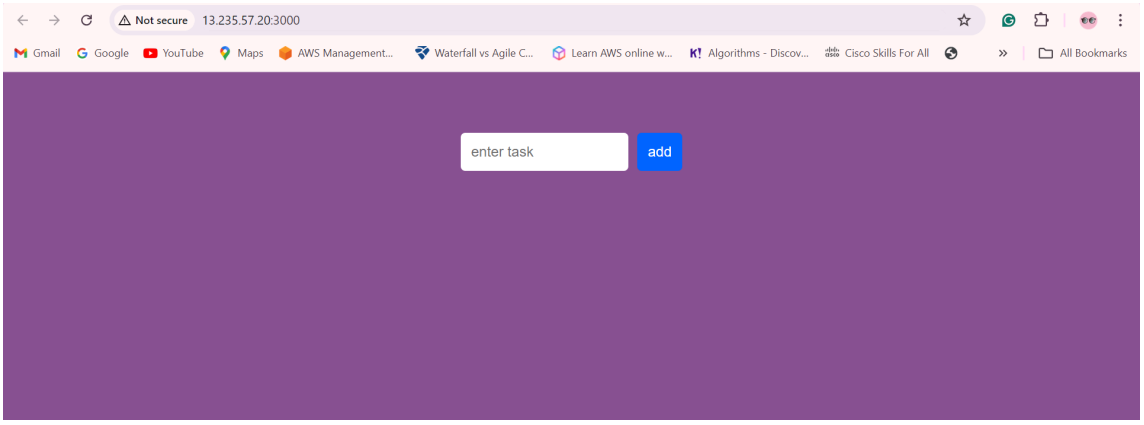
This is snippet of the after successful Deployment -



Snippet of the Multiple images -



This image after the Re-deployment -



Step 8- Create ngix.config File

- Nginx is use for Request handling Process .
- SSL/TLS Termination: Handles encryption and decryption of HTTPS connections, protecting data in transit.
- Web Server: Serves static content directly to clients without involving backend applications.

Nginx.config file explanation.

1. **server block**:

- Defines a virtual server configuration. In this case, it listens on port 3000 for incoming HTTP requests.

2. **location / block**:

- This block defines how to handle requests to the root URL `/`.
- `root /usr/share/nginx/html;` specifies the directory to serve files from.
- `index index.html index.htm;` sets the default files to look for when a directory is requested.
- `try_files $uri $uri/ /index.html;` tries to serve the requested file. If it doesn't exist, it falls back to serving `index.html`. This is important for handling client-side routing in single-page applications (SPAs) like React.

3. **location /api/ block**:

- This block defines how to handle requests to URLs starting with `/api/`.

- `proxy_pass http://backend:5000;` forwards these requests to the backend service. Replace `http://backend:5000` with your actual backend service URL.
- `proxy_set_header` directives add headers to the request forwarded to the backend. These headers help the backend service to correctly identify the original request details, such as the client's IP address and the protocol used.

Step 9: Mention Your Nginx Server to into Docker file .

Serve the App with Nginx

Base Image:

```
FROM nginx:stable-alpine
```

Explaination -

This line sets the base image to `nginx:stable-alpine`, a lightweight Nginx image.

Copy Built Assets:

```
COPY --from=build /app/build /usr/share/nginx/html
```

Explaination -

Copies the build artifacts from the `build` stage (`/app/build`) to the Nginx HTML directory (`/usr/share/nginx/html`). The `--from=build` part specifies that the source files are from the previous build stage.

Copy Custom Nginx Configuration:

```
COPY nginx.conf /etc/nginx/conf.d/default.conf
```

Explaination -

Copies the custom Nginx configuration file (`nginx.conf`) into the Nginx configuration directory.

Expose Port 80:

```
EXPOSE 3000
```

Explaination -

Exposes port 3000, which is the default HTTP port. This makes the application accessible on this port.

Start Nginx Server:

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

Explaination -

Sets the command to start the Nginx server. The `-g "daemon off;"` option tells Nginx to run in the foreground, which is required for Docker containers to stay alive.