

## Assignment-11(AWS-CI/CD Pipeline)

What is CI/CD?

CI/CD stands for Continuous Integration and Continuous Deployment/Delivery. It automates the process of code integration, testing, and deployment.

Example Workflow (Node.js App):

1. Developer pushes code to GitHub.
2. CodePipeline pulls the code.
3. CodeBuild builds a Docker image.
4. Image pushed to Amazon ECR (Elastic Container Registry).
5. Deployment done via ECS (Elastic Container Service) using Fargate.

5 Key Stages of AWS CI/CD (CodeSuite)

1. **Source** - Pull code from GitHub or other providers.
2. **Build** - Compile/build using AWS CodeBuild.
3. **Test** - Optional testing stage.
4. **Deploy** - Use ECS, Lambda, or Elastic Beanstalk to deploy.
5. **Monitor** - Use CloudWatch for logs and alerts.

Execution Modes in CodePipeline

Mode	Description
Superseded	Cancels old pipeline executions if a new one starts.
Queued	Queues all executions; runs one at a time.
Parallel	Runs multiple executions simultaneously.

ECS vs Fargate - Comparison

Feature	ECS (EC2 Launch Type)	Fargate
Server Management	You manage EC2 instances	No server management (serverless)
Cost Efficiency	May cost less with reserved instances	Pay per use (more efficient in most cases)

Scaling	Manual or auto-scaling with effort	Auto-scales automatically
Use Case	Better when full control over infra is needed	Ideal for microservices, quick setup
Recommendation	Suitable for large, consistent workloads	Recommended by AWS for most use cases

Export to Sheets

ECS Cluster Setup Using Fargate


- 1. Go to Amazon Elastic Container Service (ECS).
- 2. Click Create Cluster.
- 3. Cluster Name: devops-cluster.
- 4. Infrastructure: Select Fargate.
- 5. Leave other settings default.
- 6. Click Create.
- 7. Go to CloudFormation to verify stack is created.

✔ Cluster DevOps-Cluster-1 has been created successfully.


View cluster

✕

Clusters (1) [Info](#)


Last updated  June 24, 2025 at 19:31 (UTC+5:30)

Create cluster

< 1 > 

Cluster	Services	Tasks	Container instances	CloudWatch m
<a href="#">DevOps-Cluster-1</a>	0	No tasks running	0 EC2	✔ Default

DevOps-Cluster-01


Last updated  June 24, 2025 at 19:35 (UTC+5:30)

Update cluster

Delete cluster

Cluster overview

ARN

 arn:aws:ecs:ap-south-1:269855572870:cluster/DevOps-Cluster-01

CloudWatch monitoring

✔ Default

Services

Draining

-

Active

-

Status

✔ Active

Registered container instances

-

Tasks

Pending


-

Running

-

## Connecting GitHub to AWS CodePipeline


1. Go to AWS CodePipeline.
2. Click Settings → Connections.
3. Click Create Connection.
4. Provider: Select GitHub.
5. Connection Name: devops-connection.
6. Click Connect.
7. You'll be redirected to GitHub - Click Configure on your username.
8. Under Repository Access, choose Only select repositories.
9. Select the provided repo and click Save.
10. Back in AWS, click Connect to finalize the connection.

 Connection DevOps-Connection created successfully

[Developer Tools](#) > [Connections](#) > c0f18894-4c2b-43eb-a575-54924804461a

## DevOps-Connection

### Connection settings

Name	Provider	Status
DevOps-Connection	GitHub	 Available

**Arn**

arn:aws:codeconnections:ap-south-1:269855572870:connection/c0f18894-4c2b-43eb-a575-54924804461a

## Creating the CI/CD Pipeline

1. Go to Pipelines → Pipeline Create Pipeline.
2. Choose build custom pipeline.
3. Pipeline Name: devops-pipeline.
4. Execution Mode: Select Superseded.
5. Role is auto-generated.
6. Under Advanced Settings:
  - Choose Custom location for artifact store.
  - Select S3 bucket.
  - Check Allow AWS CodePipeline to create a service role.
  -
7. Click Next.

## Setting Up Source Stage

1. Source Provider: GitHub App.
2. Connection: Choose devops-connection.
3. Repository: Select the repo.
4. Branch: Select main.
5. Uncheck Enable automatic retry on stage failure.
6. Webhook Events Type: Push.
7. Filter Type: Branch.
8. Pattern: main.
9. Click Next.

## Setting Up Build Stage

1. Choose Build Provider: Other build providers.
2. Select AWS CodeBuild.
3. Under Project Name: Click Create Project.
4. You will be redirected to CodeBuild.

## Continuing Build Stage Configuration (AWS CodeBuild)

1. Project Name:
  - Set to devops-project.
2. Environment Image:
  - Use the latest image (usually selected by default).
3. Enable Docker Builds:
  - In Additional Configuration, scroll to Privileged section.
  - Turn on the flag: Enable this flag if you want to build Docker images.
  - **Explanation:** This is required because Docker-in-Docker (building Docker images inside the build environment) needs elevated privileges.
4. Environment Variables:
  - Go to the Environment variables section.
  - Paste the following key-value pairs:

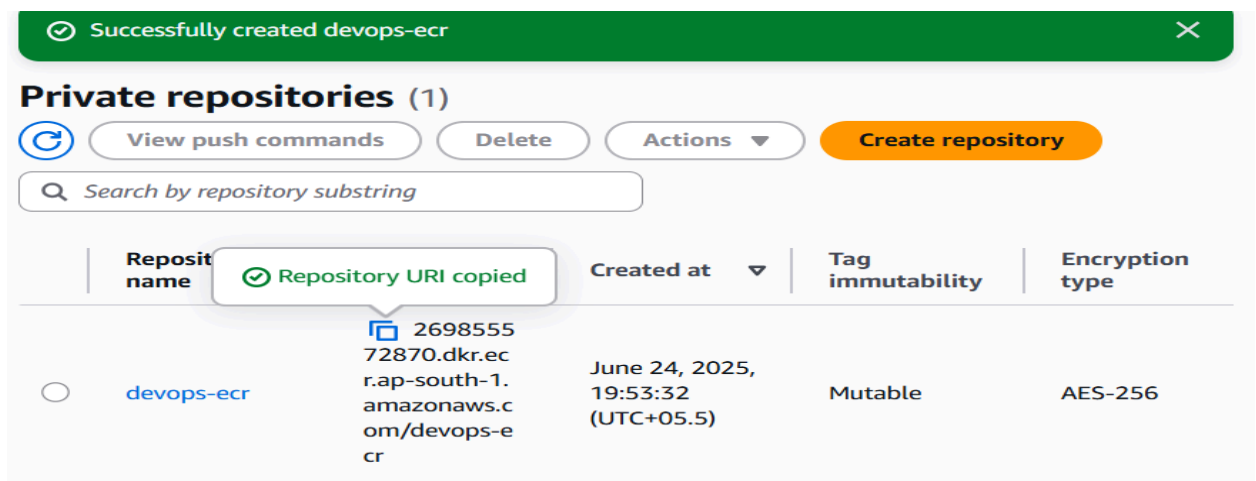
Key	Value
AWS_ACCOUNT_ID	your AWS account ID
AWS_DEFAULT_REGION	ap-south-1
IMAGE_REPO_NAME	devops-ecr (create a repository with this name)
REPOSITORY_URI	[Paste the ECR repository URI]
IMAGE_TAG	latest

## Export to Sheets

- Type: Keep all variables as Plaintext unless explicitly specified otherwise.

## Creating ECR Repository

1. In a new tab, open Amazon ECR (Elastic Container Registry).
2. Click Create Repository.
3. Repository Name: devops-ecr.
4. Click Create.
5. Copy the Repository URI.
6. Go back to CodeBuild, and paste it in the REPOSITORY\_URI variable.



## Configuring Buildspec

1. In Buildspec section, select:
  - "Use a buildspec file" from the source code (usually named buildspec.yml).
2. Click Create Project.

## Returning to CodePipeline

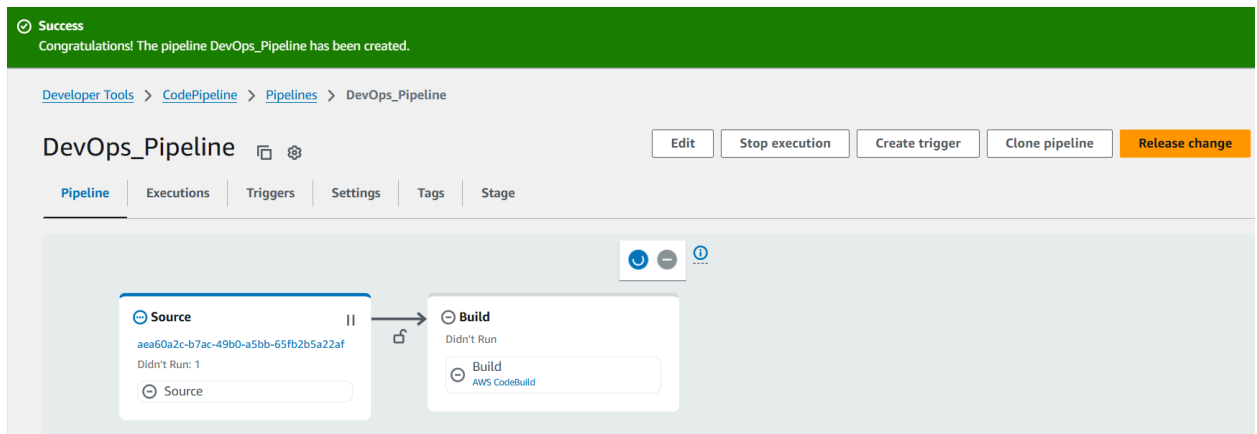
1. You'll be automatically redirected back to CodePipeline.
2. It should now select the newly created build project: devops-project.

## Test & Deploy Stages (Skipped)

1. Click Next.
2. Uncheck:
  - Enable automatic retry on stage failure (for both stages).
3. In Deploy stage, uncheck all options - since we're skipping deploy for now.

## Final Step - Create the Pipeline

1. Click Next to reach the Review page.
2. Review your settings.
3. Click Create Pipeline.



### Pipeline Execution & Build Error (Expected)

1. Once the pipeline is created, you'll see the build diagram.
2. The build will start automatically - give it a few minutes to complete.
3. **Purposeful Build Error:** You will see the build fail intentionally to demonstrate a common real-world issue.
4. Click on the Build stage → Logs to view the error details.
5. The error is typically related to insufficient permissions to push to Amazon ECR.

### Fixing the Error - Granting ECR Permissions to CodeBuild

1. Go to AWS CodeBuild Build Projects.
2. Click on your project (devops-project).
3. Scroll to the "Service role" section and click on the role name.
  - This takes you to IAM → Roles.

### Add Inline Policy to Service Role

1. In the IAM Role page:
  - Click Add permissions → Create inline policy.

Go to the JSON tab and paste the following:

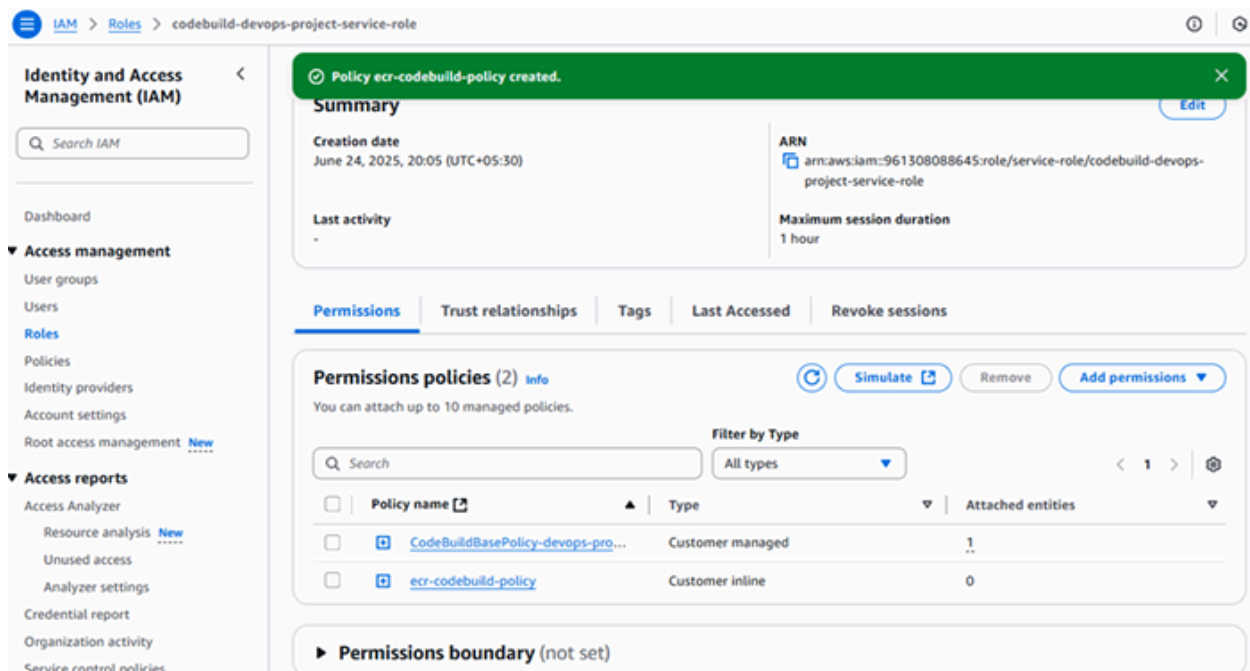
```
JSON
{
  "Statement": [
    {
      "Action": [
```

```

    "ecr:BatchCheckLayerAvailability",
    "ecr:CompleteLayerUpload",
    "ecr:GetAuthorizationToken",
    "ecr:InitiateLayerUpload",
    "ecr:PutImage",
    "ecr:UploadLayerPart",
    "ecr:*"
  ],
  "Resource": "*",
  "Effect": "Allow"
}
],
"Version": "2012-10-17"
}

```

2. Click Next.
3. Name the policy: ecr-codebuild-policy.
4. Click Create Policy.



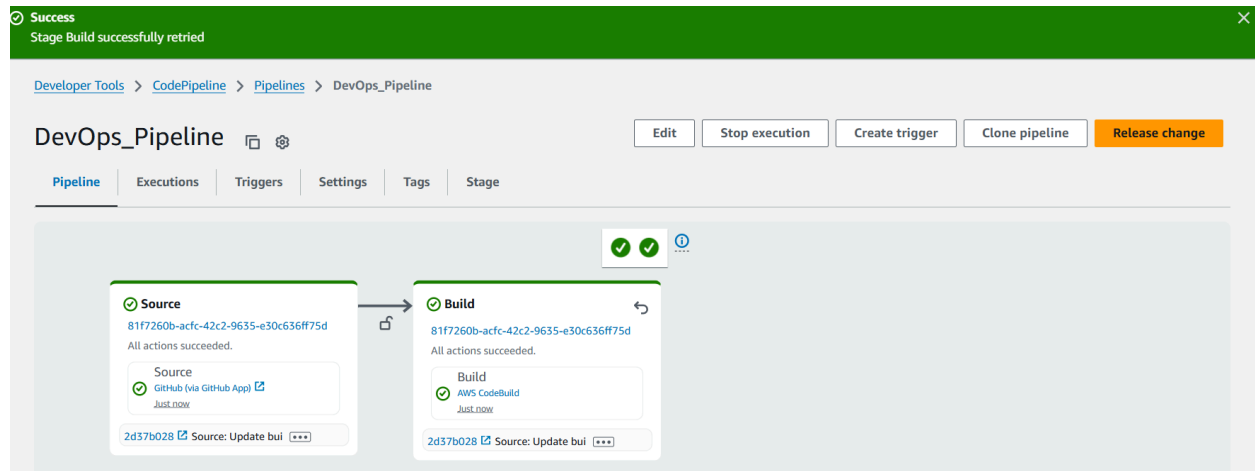
## Retrying the Pipeline Build

1. Return to AWS CodePipeline → Pipelines.
2. Select your pipeline (devops-pipeline).

3. In the failed stage, click Retry.

## Successful Build & ECR Login

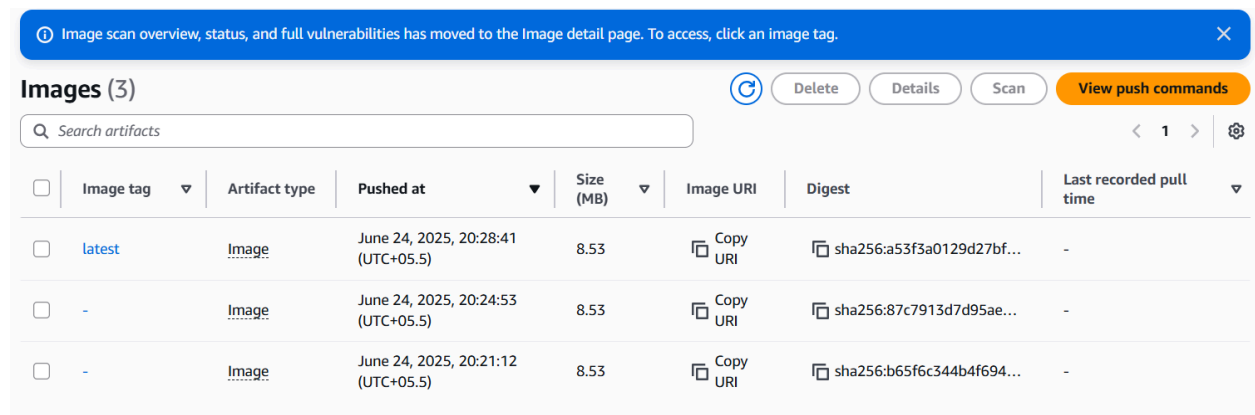
- After retrying:
  - Check the build logs.
  - There should be no errors now.
  - You will see confirmation that you're successfully logged into Amazon ECR.
- Your CI/CD pipeline has now built the Docker image and pushed it to ECR!



## Deploying Docker Image to ECS using Fargate

### Check Image in Amazon ECR

- Once the pipeline succeeds:
  - Go to Amazon ECR.
  - You'll see an image named: latest.
  - This is your built and pushed Docker image!







## Creating Task Definition in ECS

1. Go to Amazon ECS → Task Definitions.
2. Click Create new task definition.
3. Set Name: devops-task.
4. Launch Type: Choose AWS Fargate.
5. CPU and Memory: Select 2 vCPU and 4 GB RAM.

## Container Settings





1. Container name: devops-ecr.
2. Image URI:
  - Go to ECR, copy the repository URI, and paste it here.
3. Port Mappings:
  - Container port: 80.
  - Protocol: TCP.
  - Port Name: container-port.
  - App Protocol: HTTP.
4. Click Create Task Definition.
5. You can inspect task configuration by viewing the JSON.

 Task definition successfully created  
DevOps-Task:1 has been successfully created. You can use this task definition to deploy a service or run a task.

[View task definition](#) 

**DevOps-Task:1** [Deploy](#) [Actions](#) [Create new revision](#)

**Overview** [Info](#)

<b>ARN</b>  <a href="#">arn:aws:ecs:ap-south-1:26985557:2870:task-definition/DevOps-Task:1</a>	<b>Status</b>  <b>ACTIVE</b>	<b>Time created</b> <a href="#">June 24, 2025 at 20:33 (UTC+5:30)</a>	<b>App environment</b> Fargate
<b>Task role</b> -	<b>Task execution role</b> <a href="#">ecsTaskExecutionRole</a> 	<b>Operating system/Architecture</b> Linux/X86_64	<b>Network mode</b> awsvpc
<b>Fault injection</b>  Turned off			

## Creating ECS Service for the Task

1. Go to Task Definition → Revision 1.
2. Click Create Service.

## Service Configuration

- Service Name: devops-svc.
- Launch Type: Fargate.
- Enable: Availability Zone Rebalancing.
- Health Check Grace Period: Keep default (0).

- Do not click create, further config required, follow instructions.

## Networking & Load Balancer Configuration

### Security Group

1. Under Networking, click Create a new Security Group.
2. Name: devops-sg.
3. Description: devops-sg.
4. Inbound Rules:
  - Type: All Traffic.
  - Source: Anywhere.

### Public IP

- Set Public IP to: ENABLED (for testing visibility).

### Load Balancer

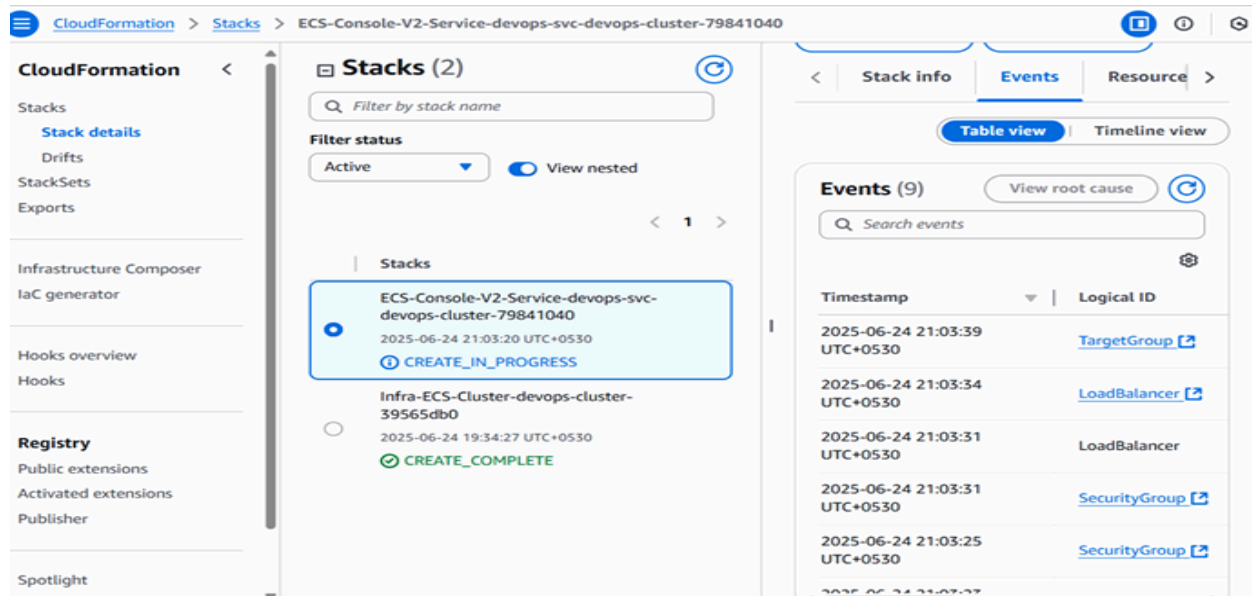
1. Choose: Use Load Balancer.
2. Type: Application Load Balancer.
3. Click Create a New Load Balancer.
  - Name: devops-lb.
  - Listener: Port 80, Protocol HTTP.

### Target Group

1. Create new Target Group:
  - Name: devops-tg.
2. Health Check Path: /.
3. Click Create.

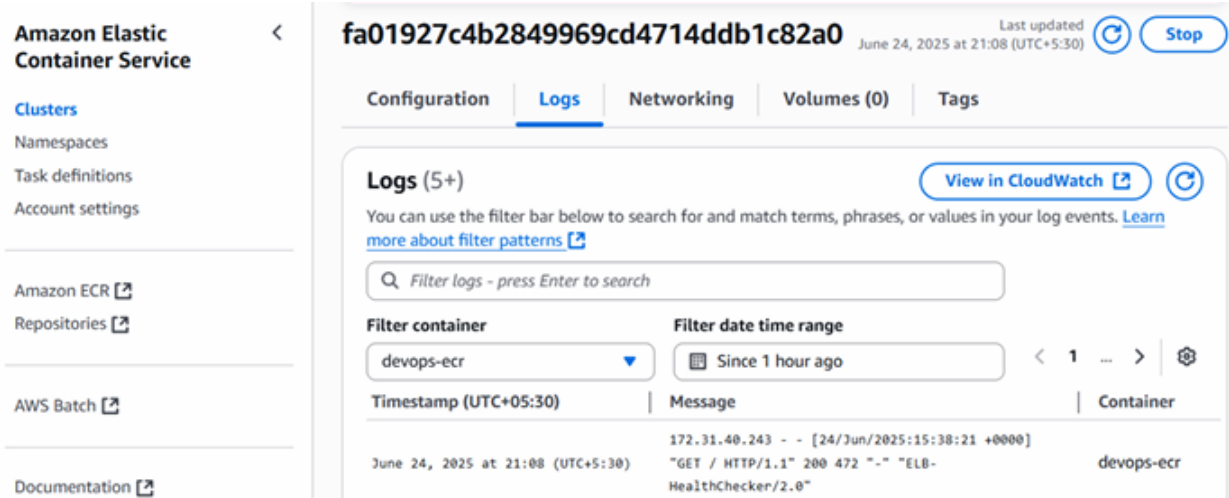
### Verifying Provisioned Resources

1. Go to AWS CloudFormation → Stacks.
2. You'll see multiple stacks created:
  - Security Group
  - Target Group
  - Load Balancer
  - Service
  - Task Definition
3. Wait for all stacks to complete.



## Monitor Task & Service Status

1. Go to ECS → Clusters → Services.
2. Click on your service (devops-svc).
3. Go to Tasks tab.
  - Wait until task status = RUNNING.
4. Click on the task → go to Logs.
  - Should display: Success Code 200.



## Accessing Your Application

1. Scroll to the Configuration section in task details.
2. Copy the Public IP.
3. Paste it into your browser: `http://<PUBLIC-IP>`.
4. Your Node.js application is live via Fargate!



## Adding ECS Deploy Stage to CodePipeline

1. Go to CodePipeline → Your Pipeline.
2. Click Edit.
3. Scroll to the bottom, click + Add Stage.
  - Name: Deploy.

### Add Action Group

1. Click Add Action Group under Deploy stage.
2. Action Group Name: Deploy.
3. Action Provider: Choose Amazon ECS.
4. Input Artifacts: Select BuildArtifact.
5. Select:
  - Cluster Name: devops-cluster.
  - Service Name: devops-svc.
6. Click Done, then Save pipeline.
7. Your pipeline now shows 3 stages: Source → Build → Deploy.

### Fixing Permissions for ECS Deployment

1. In CodePipeline → Settings, click the Service Role ARN link.

2. In IAM → Role page:
  - Click Add permissions → Attach policies.
3. Search: ECS.
4. Select: AmazonECS\_FullAccess.
5. Click Add permissions.
6. Go back to CodePipeline, and your deploy stage is now functional!

## Testing CI/CD – Triggering Deployment from GitHub

1. Go to your GitHub repository (the one connected to CodePipeline).
2. Navigate to: `src → app.jsx`.
3. Click Edit File.

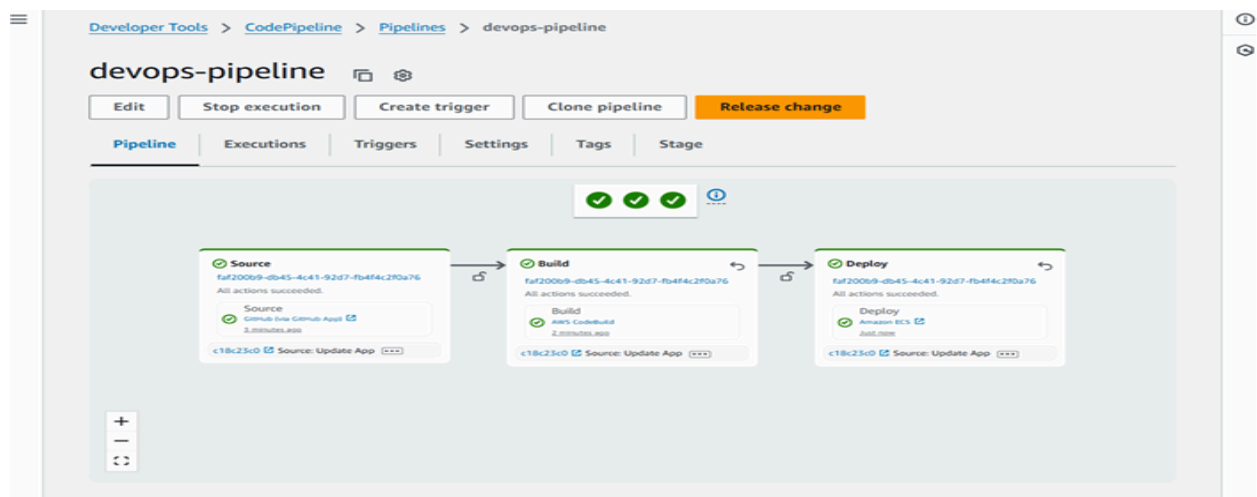
## Make a Code Change

Add the following line inside the JSX code: `<h1>This code is from the DevOps class!!!</h1>`

4. Scroll down and Commit the changes.

## Automatic CI/CD Trigger

1. As soon as the commit is made:
  - Go to AWS CodePipeline → Your Pipeline.
2. The pipeline will be automatically triggered via GitHub webhook.
3. The pipeline will:
  - Pull the updated source code.
  - Build the Docker image with the updated app.
  - Push it to ECR.
  - Deploy it to ECS Fargate using the defined service.



## Accessing the Deployed App via Load Balancer DNS

1. Go to AWS → EC2 → Load Balancers.
2. Find your Application Load Balancer (devops-lb).
3. Locate the DNS Name of the load balancer.
4. Copy the DNS name URL (something like):  
`http://devops-lb-xxxxxxxxxx.ap-south-1.elb.amazonaws.com.`
5. Paste the URL in your browser.

You'll See:

- The live Node.js application deployed via ECS.
- With the message: `<h1>This code is from the DevOps class!!!</h1>`.
- This confirms the entire CI/CD pipeline is working end-to-end: Code → GitHub → CodePipeline → Build → ECR → ECS → Load Balancer → Browser.

**This code is from the DevOps class!!!**

**This code is from the devops class**



**Vite + React**

count is 0

Edit `src/App.jsx` and save to test HMR