Bachelor of Technology

Computer Engineering Sem : 7

01CE0717 – DevOps Essentials

Marwadi University
Marwadi Chandarana Group

NAAC A+

Unit : 5 – Docker & Devops Pipeline Using AWS

Content

- What is container technology?
- Docker introduction,
- How docker works,
- Networking in Docker,
- DockerHub,
- Create Docker images using yaml,
- Upload DockerImages to DockerHub,

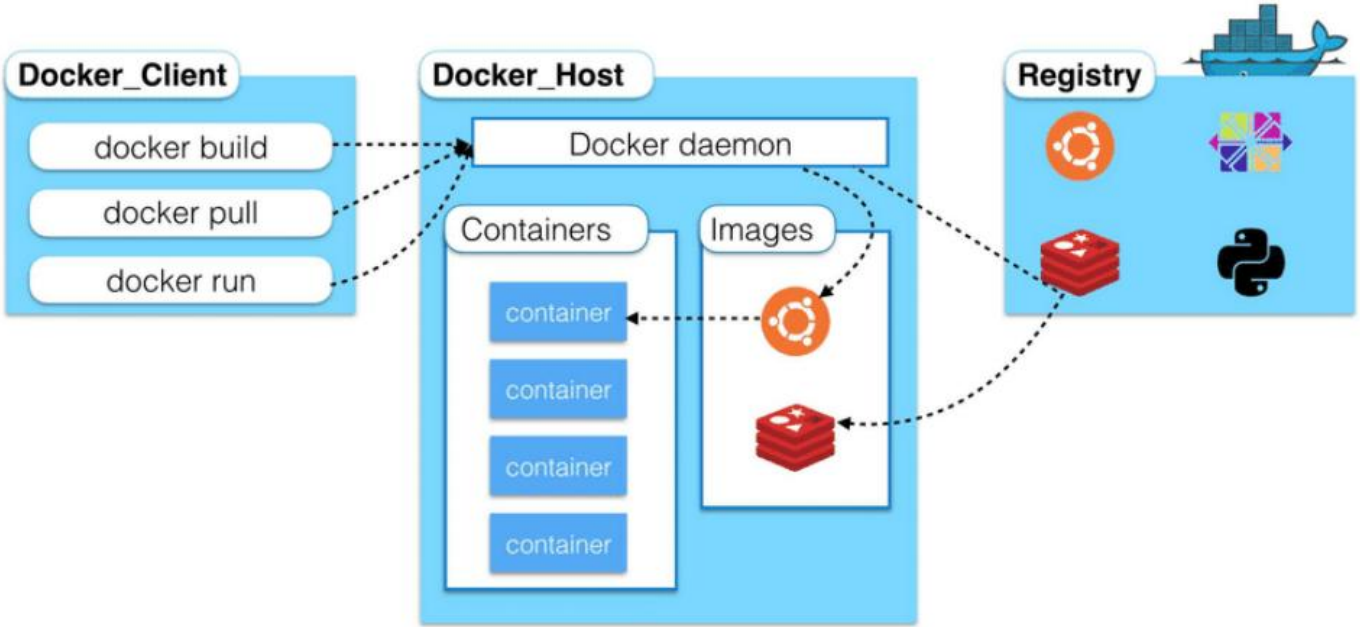
Content

- Create Github/AWS Account,
- Create Repository,
- Create a new pipeline,
- Build a sample code,
- Modify AWS code pipelines,
- Deploy from github actions to EC2.

Content

- **Definition Containerization**
- A form of operating system-level virtualization that allows software applications to run in isolated user spaces called containers, in any computing environment.
- **Functionality**
- Containers simulate different software applications running isolated processes by bundling related configuration files, libraries, and dependencies, while sharing a common operating system kernel.
- **Benefits**
- Provides a fully functional and portable computing environment for applications, keeping them independent of other environments running in parallel.

Content


Docker_Client
docker build
docker pull
docker run
Docker_Host
Docker daemon
Containers
container
container
container
container
Images
Registry

Content

- **Key Benefits:**
- Portability: Containers can be moved seamlessly between different environments (e.g., from a developer's machine to a production server).
- Efficiency: Containers share the host operating system kernel, making them lightweight compared to virtual machines (VMs), which require separate OS instances.
- Speed: Containers can start and stop quickly, allowing for rapid scaling and deployment.
- Isolation: Each container runs in its own isolated environment, reducing conflicts between applications.

Content

- **Common Technologies:**
- **Docker**: The most popular containerization platform that simplifies the process of creating, deploying, and managing containers.
- **Kubernetes**: An orchestration tool for managing containerized applications across clusters of machines, facilitating automated deployment, scaling, and management.
- **Other Tools**: Additional tools include Apache Mesos, rkt (pronounced "rocket"), and various container management solutions like Red Hat OpenShift.

Content

- **Use Cases:**
- **Microservices Architecture**: Containers are ideal for deploying microservices, where each service can be developed and scaled independently.
- **Cloud Migration:** Legacy applications can be encapsulated in containers for easier migration to cloud environments without extensive rewrites.
- **IoT Devices**: Containerization can simplify application deployment across resource-constrained IoT devices.

Content

- **Security Considerations**: While containers provide many benefits, they also present security challenges due to shared OS kernels. If a vulnerability exists in the host kernel, it can affect all running containers. Security measures include using signing infrastructure to prevent unauthorized containers from running and employing security tools that monitor container behavior.

Content

- **Docker** is an open-source containerization platform that enables developers to build, deploy, and manage applications in isolated environments called containers. Containers encapsulate an application and all its dependencies, ensuring that it runs consistently across different computing environments.

Content

- **Key Benefits of Docker**
- Portability: Containers can run on any system that supports Docker, allowing applications to be easily moved between development, testing, and production environments.
- Isolation: Each container runs in its own environment, ensuring that applications do not interfere with each other.
- Efficiency: Containers share the host operating system's kernel, making them lightweight compared to virtual machines.
- Scalability: Docker makes it easy to scale applications by deploying multiple containers quickly.
- Consistency: Applications behave the same way regardless of where they are deployed.

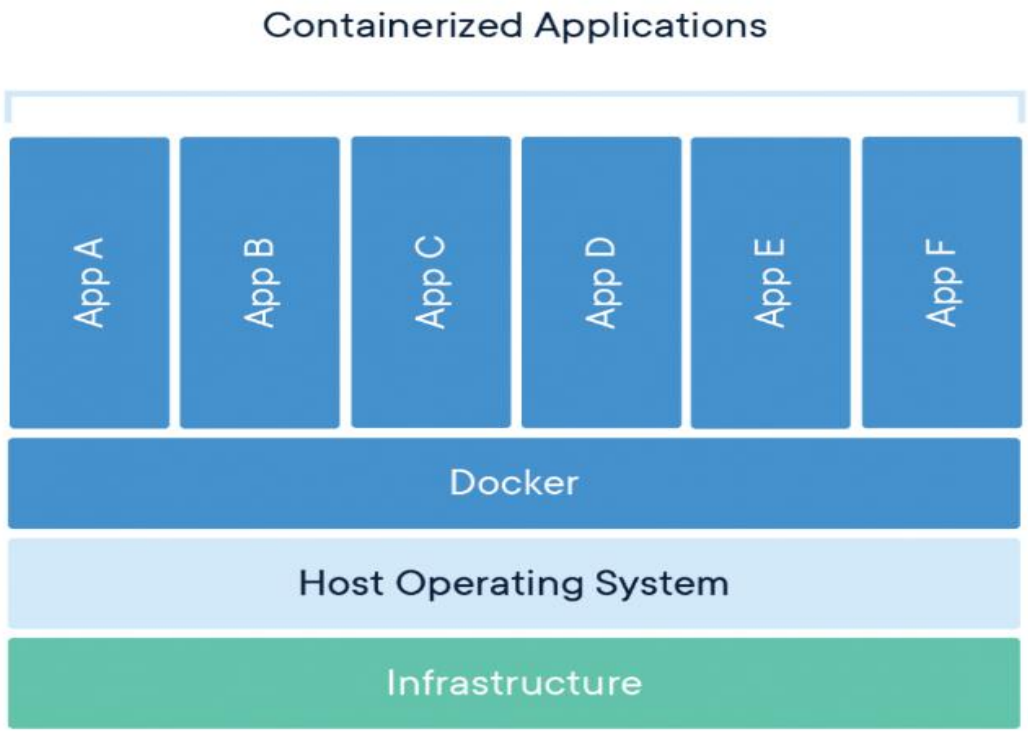
Content

- **Docker Architecture**
- Docker follows a client-server architecture consisting of several key components:
- Docker Client: The command-line interface (CLI) that allows users to interact with the Docker daemon. Users issue commands like docker run, docker build, etc., through the client.
- Docker Daemon (dockerd): The background service that manages Docker containers. It listens for API requests from the client and performs tasks such as building images, running containers, and managing networks.
- Docker Images: Read-only templates used to create containers. Images contain the application code, libraries, and dependencies required for the application to run.

Content

- Docker Containers: Running instances of Docker images. Containers are created from images and can be started, stopped, moved, or deleted.
- Docker Registry: A storage system for Docker images. Docker Hub is the default public registry where users can upload and download images.
- Docker Networks: Allow containers to communicate with each other and with external systems securely.
- Docker Volumes: Provide persistent storage for containers, allowing data to persist even when containers are stopped or removed.

Content


Containerized Applications
App A
App B
App C
App D
App E
App F
Docker
Host Operating System
Infrastructure

Content

- **Client-Server Architecture:**
- **Docker Client**: The command-line interface (CLI) that users interact with to issue commands such as docker run, docker build, and docker pull. The client communicates with the Docker daemon to perform these operations.
- **Docker Daemon (dockerd):** A background process that manages Docker containers. It listens for API requests from the Docker client and carries out tasks like building images, running containers, and managing networks.

Content

- **Docker Images:**
- An image is a read-only template used to create containers. It contains everything needed to run an application, including the code, libraries, and environment variables. Images are built from a set of instructions defined in a file called a Dockerfile.

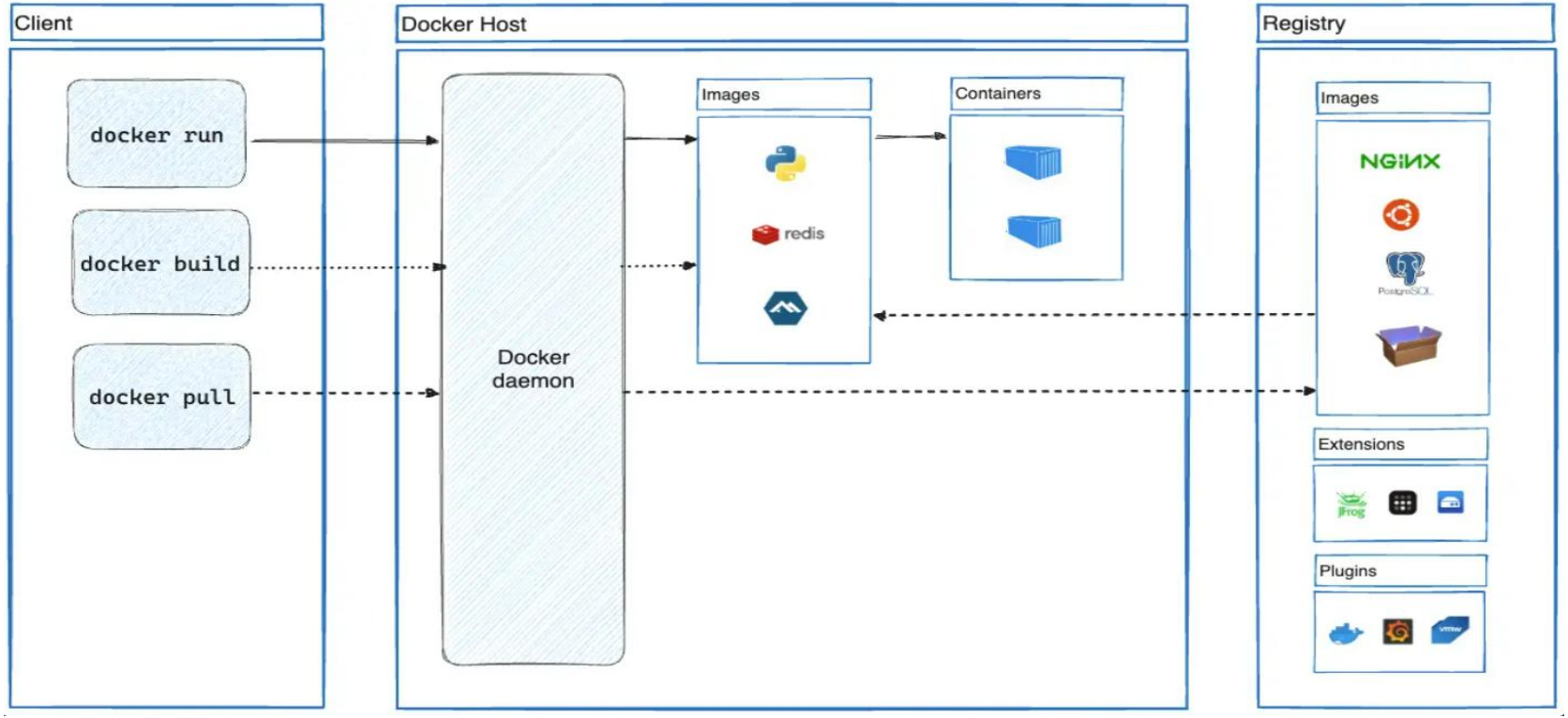
- **Docker Containers:**
- A container is a running instance of a Docker image. Containers are isolated from each other and the host system, ensuring that applications do not interfere with one another. They can be started, stopped, moved, and deleted independently.

Content

- **Docker Registry:**
- A registry is a repository for Docker images. Docker Hub is the default public registry where users can share and download images. Organizations can also set up private registries to store their own images.

- **Networking and Storage:**
- Docker provides networking capabilities that allow containers to communicate with each other and with external systems. It also supports persistent storage through volumes, allowing data to persist even when containers are stopped or removed.

Content


Client
docker run
docker build
docker pull
Docker Host
Docker daemon
Images
redis
Containers
Registry
Images
NGINX
Extensions
Plugins

Networking In Docker

- **Network Drivers**: Docker supports several types of network drivers, each serving different purposes:
- **Bridge**: The default network driver. Containers on the same bridge network can communicate with each other using their IP addresses. This driver is suitable for standalone applications that do not need to interact with the host's network directly.
- **Host:** This driver allows containers to share the host's network stack. Containers will use the host's IP address and ports directly, providing low latency but removing isolation between the host and containers.

Networking In Docker

- **Overlay:** Used for multi-host networking, allowing containers running on different Docker hosts to communicate with each other. This is essential for Docker Swarm setups where services need to scale across multiple hosts.

- **Macvlan**: This driver allows you to assign a MAC address to a container, making it appear as a physical device on the network. It is useful for scenarios where containers need to interact with existing network infrastructure.

- **None:** This disables all networking for a container, isolating it completely from external networks.

Networking In Docker

- Here are some essential commands for managing Docker networks:
- **List Networks:**
- bash
- docker network ls

- **Create a Network:**
- bash
- docker network create --driver bridge my_bridge_network

Networking In Docker

- **Connect a Container to a Network:**
- bash
- docker network connect my_bridge_network my_container

- **Disconnect a Container from a Network:**
- bash
- docker network disconnect my_bridge_network my_container

Networking In Docker

- **Inspect a Network:**
- bash
- docker network inspect my_bridge_network

- **Remove a Network:**
- bash
- docker network rm my_bridge_network

Networking In Docker

- **Example of Docker Networking**
- **Creating a Bridge Network:**
- bash
- docker network create my_bridge_network

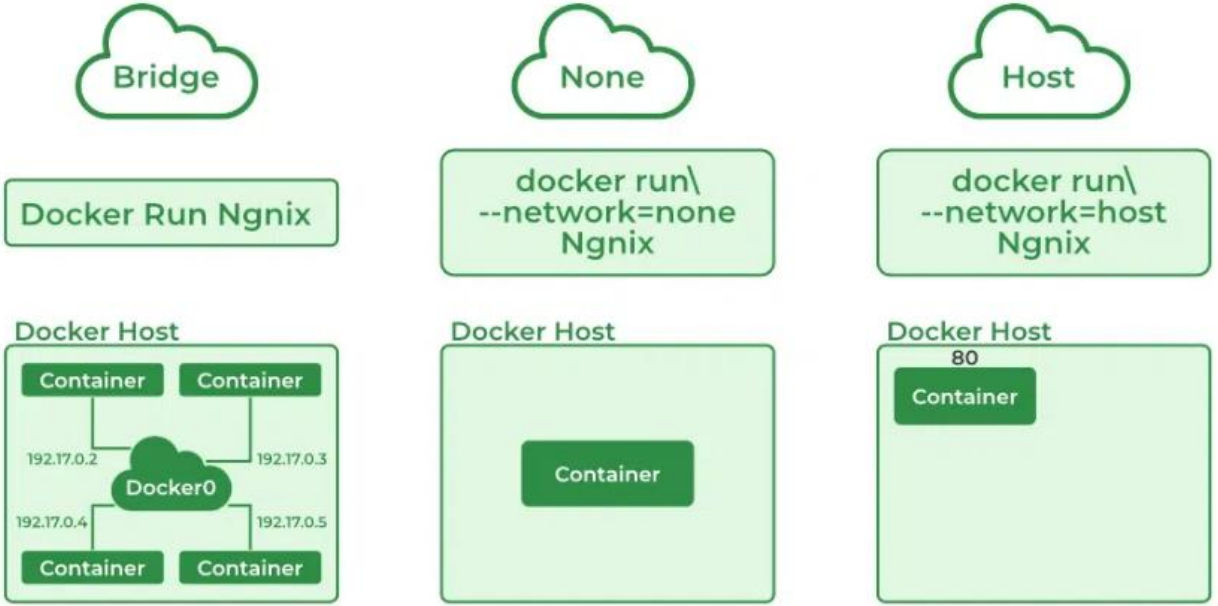
- **Running Containers on the Bridge Network:**
- bash
- docker run -d --name container1 --network my_bridge_network nginx
- docker run -d --name container2 --network my_bridge_network nginx

Networking In Docker

- **Testing Connectivity Between Containers:**
- bash
- docker exec -it container1 /bin/bash
- ping container2

- Note : You can exec into one container and ping another:

Networking In Docker



Networking In Docker

- **Docker Hub** is a cloud-based registry service provided by Docker for finding, sharing, and managing container images. It serves as the largest repository of container images, allowing developers to host public repositories for free or private repositories for teams and enterprises. Here's a detailed overview of Docker Hub, its features, and how it works.
- Overview of Docker Hub
- **Purpose**: Docker Hub allows users to store and share Docker images. It acts as a central repository where developers can publish their container images and pull images created by others.
- **Access**: Users can create an account on Docker Hub to manage their repositories, collaborate with teams, and access a vast library of community-contributed images.

Networking In Docker

- **Key Features of Docker Hub**
- **Public and Private Repositories:**
- Users can create public repositories accessible to everyone or private repositories that restrict access to specific users or teams.
- **Image Management:**
- Users can push and pull container images to and from their repositories. This facilitates easy sharing of applications across different environments.
- **Automated Builds:**
- Docker Hub supports automated builds that allow users to automatically build container images from source code hosted on GitHub or Bitbucket.

Networking In Docker

- **Webhooks:**
- Users can configure webhooks to trigger actions after a successful push to a repository, integrating Docker Hub with other services.
- **Docker Trusted Content:**
- Access verified content from Docker Official Images, Trusted Partner Content, and Trusted Open Source Content, ensuring that the software you use is secure and reliable.
- **Community Engagement:**
- With over 20 billion monthly image downloads, Docker Hub fosters a global community where developers can publish, share, and contribute to open-source projects.

Networking In Docker

- **How Docker Hub Works**
- Creating an Account: Users sign up for a free account on Docker Hub to access its features.
- Building Images: Developers create Docker images locally using a Dockerfile, which contains instructions for building the image.
- Pushing Images: After building an image, users push it to their Docker Hub repository using the command:
- bash
- docker push <username>/<repository>:<tag>

Networking In Docker

- Pulling Images: Users can pull images from Docker Hub using:
- bash
- docker pull <username>/<repository>:<tag>

- Using Images: Once pulled, the images can be run as containers on any system with Docker installed.

Networking In Docker

- **Example Workflow**
- **Build an Image:**
- Create a Dockerfile for your application and build it:
- bash
- docker build -t myapp:latest .

Networking In Docker

- **Tag the Image:**
- Tag the image for your Docker Hub repository:
- bash
- docker tag myapp:latest username/myapp:latest

- **Push the Image to Docker Hub:**
- Push your tagged image to your repository on Docker Hub:
- bash
- docker push username/myapp:latest

Networking In Docker

- **Pull the Image from Docker Hub:**
- On another machine or environment, pull the image using:
- bash
- docker pull username/myapp:latest

- **Run the Container:**
- Run a container from the pulled image:
- bash
- docker run -d username/myapp:latest

Create Docker images using yaml

- **Create a docker file**
- # Dockerfile
- FROM node:14
- WORKDIR /app
- COPY package*.json ./
- RUN npm install
- COPY . .
- EXPOSE 3000
- CMD ["npm", "start"]

Create Docker images using yaml

- **Create a docker-compose.yml file**
- version: '3'
- services:
- app:
- build:
- context: .
- dockerfile: Dockerfile

Create Docker images using yaml

- ports:
- - "3000:3000"
- volumes:
- - .:/app
- environment:
- NODE_ENV: development
- **To run :**
- docker-compose up --build

Upload Docker Images to DockerHub

- **Step 1: Login to Docker Hub**
- Login to Docker Hub using the Docker CLI:
- bash
- sudo docker login
- Enter your Docker Hub username and password when prompted. If successful, you'll see a message:
- Login Succeeded

Upload Docker Images to DockerHub

- **Step 2: Tag the Docker Image**
- Once your image is built, you need to tag it with your Docker Hub repository name.
- List your Docker images to find the image you want to upload:
- bash
- sudo docker images
- Example output:
- REPOSITORY TAG IMAGE ID CREATED SIZE
- myapp latest d123456abcde 10 hours ago 300MB

Upload Docker Images to DockerHub

- **Tag the image with your Docker Hub username and repository name:**
- bash
- sudo docker tag <IMAGE_ID> <DockerHubUsername>/<RepositoryName>:<Tag>
- Example:
- bash
- sudo docker tag d123456abcde myusername/myapp:latest
- Replace <IMAGE_ID> with the actual image ID, myusername with your Docker Hub username, and myapp with your repository name. You can use any tag (e.g., latest, v1.0).

Upload Docker Images to DockerHub

- **Step 3: Push the Docker Image to Docker Hub**
- Push the Docker image to Docker Hub:
- bash
- sudo docker push <DockerHubUsername>/<RepositoryName>:<Tag>
- Example:
- bash
- sudo docker push myusername/myapp:latest
- Wait for the image to upload. Once complete, the image will be available on Docker Hub.

Upload Docker Images to DockerHub

- **Step 4: Verify the Image on Docker Hub**
- Go to your Docker Hub account in a web browser:
- https://hub.docker.com
- Navigate to your repository. You should see the uploaded image.

Create Github /AWS Account

- **Go to GitHub:**
- Sign Up:
- Click the "Sign up" button in the upper-right corner.
- Enter your email address, then click "Continue".
- Create a password, then click "Continue".
- Choose a GitHub username (it must be unique), then click "Continue".
- Complete the puzzle to verify you are not a bot, and then click "Create account".
- Confirm Your Email:
- Check your email inbox for a confirmation email from GitHub and follow the link to verify your email.

Create Github /AWS Account

- **Sign Up AWS Account:**
- Click the "Create an AWS Account" button in the upper-right corner.
- Enter Account Information:
- Enter your email address, choose an account name (e.g., your name or organization), and set a password.
- Select Account Type:
- Choose whether your account is for Personal or Business use.
- Enter Contact Information:
- Provide your address, phone number, and country.
- Add Payment Method:

Create Github /AWS Account

- AWS requires a credit/debit card for billing. You won't be charged upfront; many services offer a free tier for new users.
- Identity Verification:
- AWS will send you a verification code via phone or text. Enter this to complete the verification process.
- Choose Support Plan:
- You can select the Basic Support (Free) plan if you're just starting out.
- Complete Setup:
- Once everything is filled out, click "Complete Sign Up".
- You can now log in to the AWS Management Console using your email and password.

Create Repository

- Log in to your GitHub account.
- On the GitHub homepage, click the + icon in the top-right corner and select New repository.
- Fill out the repository details:
- Repository name: Enter a unique name for your repository.
- Description (optional): Provide a description of the project.
- Public or Private: Choose whether your repository is public (anyone can see it) or private (only you and invited collaborators can see it).
- Initialize repository: You can check this box to initialize the repository with a README file, which provides an overview of the project.
- Click Create repository.

Create a new pipeline

- Open Jenkins Dashboard:
- Go to your Jenkins URL (e.g., http://localhost:8080 or your Jenkins server address).
- Log in to your Jenkins account.
- Create a New Pipeline Job:
- On the Jenkins dashboard, click on New Item.
- Enter a name for your pipeline (e.g., "MyPipeline").
- Select Pipeline from the list and click OK.
- Configure Pipeline:
- In the pipeline configuration page, you can add a description if needed.
- Scroll down to the Pipeline section. You can define your pipeline script here.

Create a new pipeline

- Define Pipeline Script: You have two options:
- Pipeline Script (Inline): You can write the pipeline script directly in the Jenkins UI.
- Pipeline Script from SCM (Source Code Management): If your Jenkinsfile is stored in a GitHub or Git repository, select this option and configure the repository details.
- Save and Run the Pipeline:
- After adding the pipeline script, click Save.
- To run the pipeline, click on Build Now.

Create a new pipeline

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building...'
            }
        }

```

Create a new pipeline

```
stage('Test') {
            steps {
                echo 'Testing...'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying...'
            } } }
}
```

Build a Sample Code

- Step 1: Create a New React Application
- First, you need to create a new React application using create-react-app. Open your terminal and run:
- bash
- npx create-react-app my-react-app
- cd my-react-app

Build a Sample Code

- Step 2: Create a Dockerfile
- In the root directory of your React application (my-react-app), create a file named Dockerfile with the following content:
- text
- # Use the official Node.js image as a base
- FROM node:14-alpine
- # Set the working directory inside the container
- WORKDIR /app
- # Copy package.json and package-lock.json to the working directory
- COPY package*.json ./

Build a Sample Code

- # Install dependencies
- RUN npm install
- # Copy the rest of your application code to the container
- COPY . .
- # Expose port 3000 for the app to be accessible
- EXPOSE 3000

- # Command to run the app
- CMD ["npm", "start"]

Build a Sample Code

- Step 3: Create a .dockerignore File
- To optimize the build process, create a .dockerignore file in the root directory of your project. This file tells Docker which files and directories to ignore when building the image. Add the following content:
- text
- node_modules
- build
- .dockerignore
- Dockerfile
-

Build a Sample Code

- Step 4: Build the Docker Image
- Now that you have your Dockerfile ready, you can build your Docker image. Run the following command in your terminal (make sure you're in the my-react-app directory):
- bash
- docker build -t my-react-app .
- This command builds an image with the name my-react-app using the current directory (.) as context.
-

Build a Sample Code

- Step 5: Run the Docker Container
- Once the image is built, you can run it using:
- bash
- docker run -p 3000:3000 my-react-app

- This command runs your React application in a container and maps port 3000 on your host machine to port 3000 in the container.

Build a Sample Code

- Step 6: Access Your Application
- Open your web browser and navigate to http://localhost:3000. You should see your React application running.
- Summary of Commands
- Here’s a summary of all commands used:
- bash
- npx create-react-app my-react-app
- cd my-react-app

Build a Sample Code

- # Create Dockerfile and .dockerignore as described above.

- # Build the Docker image
- docker build -t my-react-app .

- # Run the Docker container
- docker run -p 3000:3000 my-react-app

Modifying aws code pipeline

- Modifying an AWS CodePipeline involves updating the pipeline's stages, actions, or configuration. Below is an overview of how you can modify an AWS CodePipeline for a sample Node.js application. This includes adding stages like Source, Build, and Deploy using services like GitHub, AWS CodeBuild, and AWS Elastic Beanstalk (for deployment).
- 1. AWS CodePipeline Basic Structure
- A typical AWS CodePipeline might include the following stages:
- Source Stage: Fetches the code from a GitHub or AWS CodeCommit repository.
- Build Stage: Uses AWS CodeBuild to build the project.
- Deploy Stage: Deploys the application to a service like AWS Elastic Beanstalk or ECS.
- Here’s how you can modify the AWS CodePipeline configuration via the AWS Management Console or AWS CLI.

Modifying aws code pipeline

- Modifying AWS CodePipeline Using AWS Console
- Navigate to CodePipeline:

- Go to the AWS Management Console.
- Open the CodePipeline service.
- Select the Pipeline:

- In the pipeline dashboard, select the pipeline you want to modify.
- Modify the Stages:

Modifying aws code pipeline

- In the pipeline view, you'll see the various stages like Source, Build, and Deploy. Click on Edit near the stage you want to modify.
- Updating Source Stage:
- If you want to change the repository (e.g., GitHub):
- Click Edit on the Source stage.
- Modify the repository URL, branch, or credentials.
- You can choose GitHub, Bitbucket, or AWS CodeCommit as your source provider.
- Updating Build Stage:
- To modify the build configuration (e.g., update the buildspec file or environment):

Modifying aws code pipeline

- Click Edit on the Build stage.
- Select the CodeBuild project.
- You can update the buildspec.yml or change the environment settings (Node.js version, etc.).
- Updating Deploy Stage:
- For deploying to AWS services like Elastic Beanstalk or ECS:
- Click Edit on the Deploy stage.
- You can update the Elastic Beanstalk environment, or choose a different service like ECS.
- Save the Changes:
- After editing each stage, click Save to apply the modifications.

Modifying aws code pipeline

- Step 1: Prepare Your EC2 Instance
- Launch an EC2 Instance:
- Go to the AWS Management Console and launch a new EC2 instance with the desired configuration (e.g., Amazon Linux, Ubuntu).
- Configure Security Groups:
- Ensure that your security group allows inbound traffic on the necessary ports (e.g., port 22 for SSH, port 80 for HTTP).
- SSH Access:

Modifying aws code pipeline

- Make sure you have SSH access to your EC2 instance. You'll need the private key associated with the instance.
- Install Necessary Software:
- SSH into your EC2 instance and install any necessary software (like Node.js, Nginx, etc.) required to run your application.

Modifying aws code pipeline

- Step 2: Set Up GitHub Secrets
- Access Your GitHub Repository:
- Go to your repository on GitHub.
- Add Secrets:
- Navigate to Settings > Secrets and variables > Actions.
- Add the following secrets:
- EC2_HOST: The public IP or DNS of your EC2 instance.
- EC2_USER: The username for SSH (e.g., ec2-user for Amazon Linux).
- EC2_KEY: Your private SSH key (ensure it’s in a single line format).

Modifying aws code pipeline

- Step 3: Create a GitHub Actions Workflow
- Create a new file in your repository at .github/workflows/deploy.yml with the following content:
- text
- name: Deploy to EC2
- on:
- push:
- branches:
- - main # Change this if you want to deploy from a different branch
- jobs:
- deploy:
- runs-on: ubuntu-latest

Modifying aws code pipeline

```
steps:
    - name: Checkout code
      uses: actions/checkout@v2
    - name: Set up SSH
      run: |
        mkdir -p ~/.ssh
        echo "${{ secrets.EC2_KEY }}" > ~/.ssh/id_rsa
        chmod 600 ~/.ssh/id_rsa
        ssh-keyscan -H ${{ secrets.EC2_HOST }} >> ~/.ssh/known_hosts
```

Modifying aws code pipeline

```
- name: Deploy to EC2
    run: |
      ssh ${{ secrets.EC2_USER }}@${{ secrets.EC2_HOST }} << 'EOF'
        cd /path/to/your/app   # Change this to your app's directory
        git pull origin main    # Pull the latest code from GitHub
        npm install            # Install dependencies if needed
        npm run build          # Build your application if it's a frontend app
        # Optionally restart your application server here
      EOF
```

Modifying aws code pipeline

- Step 4: Commit and Push Changes
- Commit your changes to the repository:
- bash
- git add .github/workflows/deploy.yml
- git commit -m "Add deployment workflow"
- git push origin main

Modifying aws code pipeline

- Step 5: Monitor Deployment
- After pushing changes, go to the "Actions" tab in your GitHub repository.
- You should see the workflow running. Monitor it for any errors.

THANK YOU