DevOps Shack

GITHUB ACTIONS GUIDE

By DevOps Shack

Click here for DevSecOps & Cloud DevOps Course

DevOps Shack

GitHub Actions Step By Step Guide:

From Code to Cloud Using Docker, EC2 & GitHub Actions

🔁 Step-by-Step Workflow

✅ 1. Initialize Project → Push to GitHub

→ Set up your application locally (e.g., Node.js, Python, etc.)
→ Initialize a Git repository
→ Push the code to a GitHub repository for version control

✅ 2. Write Tests → Ensure App Quality

→ Add unit tests to verify core functionalities
→ Run tests locally to make sure the app behaves as expected

✅ 3. Create Dockerfile → Containerize the App

→ Define how the app should run inside a Docker container
→ Add a .dockerignore to keep the image clean and optimized

✅ 4. Build Docker Image Locally → Test in Container

→ Package your app into a containerized environment
→ Run the container locally to ensure it behaves correctly

✅ 5. Push Docker Image → Upload to Docker Hub

→ Authenticate with Docker Hub
→ Push the Docker image to your public/private repository for cloud access

✅ 6. Set Up GitHub Actions → Automate CI

→ Create a GitHub Actions workflow
→ Automatically run tests and build your Docker image on each push

✅ 7. Launch EC2 Instance → Prepare Hosting Environment

→ Set up an Ubuntu-based EC2 server on AWS
→ Install Docker and configure the environment for deployment

✅ 8. Deploy Docker Image to EC2 → App Goes Live

→ Pull the Docker image from Docker Hub onto EC2
→ Run the container and expose it publicly via EC2's IP

✅ 9. Optional: Attach Domain + HTTPS → Professional Touch

→ Connect a custom domain to the EC2 instance
→ Use NGINX + Certbot to add free HTTPS security

✅ 10. Automate Deployment → Full CI/CD Pipeline

→ Securely connect GitHub Actions with EC2 via SSH
→ On every push to main, automatically:

- Pull the latest image
- Restart the running container
  → No manual steps needed—just push code to go live!

🎯 Outcome

A fully automated, secure, scalable deployment pipeline:
GitHub → Docker Hub → EC2 → LIVE App — all hands-free after setup!

✅ Step 1: Set Up Your Project Repository

This is the foundational step where you create your application code and initialize it with Git so it can be pushed to GitHub.

🧰 Prerequisites

- Git installed
- A GitHub account
- Basic knowledge of CLI or terminal

🛠️ Steps

1.1 Create a new project folder

```
mkdir demo-app
cd demo-app
```

1.2 Initialize Git

```
git init
```

This creates a .git directory and makes your project a Git repository.

1.3 Create your application (Example: Node.js)

You can use any language, but here's a simple **Node.js** example:

```
touch index.js
```

Add some basic content:

```
// index.js
console.log("Hello from Dockerized App!");
```

Then create package.json:

```
npm init -y
```

Update it like this:

```
{
 "name": "demo-app",
 "version": "1.0.0",
 "main": "index.js",
 "scripts": {
  "start": "node index.js"
 }
}
```

1.4 Add a .gitignore file

Create .gitignore to avoid committing node_modules:

```
touch .gitignore
```

Contents:

```
node_modules
.env
```

1.5 Commit your code

```
git add .
git commit -m "Initial commit - Node.js app"
```

1.6 Create a GitHub repo and push code

Go to GitHub, create a new repository (e.g., demo-app), **without** a README.

Then push your code:

```
git remote add origin https://github.com/your-username/demo-app.git
git branch -M main
```

```
git push -u origin main
```

✅ Done! You now have a Node.js project set up in a GitHub repo and ready to build your CI/CD pipeline.

🐳 Step 2: Create a Dockerfile for Your Application

In this step, you'll containerize your app using **Docker** by writing a Dockerfile. This lets us package the app with its dependencies into a consistent environment that runs anywhere.

🚀 What is a Dockerfile?

A Dockerfile is a script that contains a list of instructions Docker uses to build an image of your application.

Think of it like a recipe 👨‍🍳 for building your app into a portable box.

📁 Folder Structure So Far

Before we begin, your project directory should look like this:

```
demo-app/
├── index.js
├── package.json
├── .gitignore
```

We'll now add the Dockerfile here.

🧱 Create the Dockerfile

Run:

touch Dockerfile

Then open it and add the following:

Dockerfile

CopyEdit

Use an official Node.js base image

FROM node:18-alpine

```
# Set working directory inside the container
WORKDIR /app

# Copy package files and install dependencies
COPY package*.json ./
RUN npm install

# Copy the rest of your app files
COPY . .

# Define the port your app runs on (optional, for docs)
EXPOSE 3000

# Command to run your app
CMD ["npm", "start"]
```

🔍 Explanation of Each Line

Line	Meaning
FROM node:18-alpine	Uses a lightweight Node.js image as the base
WORKDIR /app	Sets /app as the working directory inside the container
COPY package*.json ./	Copies package.json and package-lock.json (if exists)
RUN npm install	Installs Node.js dependencies

Line	Meaning
COPY . .	Copies the rest of your project files into the container
EXPOSE 3000	(Optional) Documents the port the app will listen on
CMD ["npm", "start"]	Tells Docker how to run your app when the container starts

💡 **Note:** If your app runs on a different port (like 8080 or 5000), update the EXPOSE line accordingly.

✅ Validate the Dockerfile (Optional but good practice)

Run this to check if it's correctly written:

```
docker image build -t test-dockerfile .
```

If everything goes well, it should build without errors.

🧪 Step 3: Test Your Docker Image Locally

Before pushing your app into a CI/CD pipeline or deploying it to the cloud, it's critical to make sure your **Dockerfile works properly**, and your app runs successfully in a containerized environment.

This helps you catch any issues with dependencies, ports, or file structure early on.

🛠️ What You'll Do in This Step:

- Build the Docker image from your Dockerfile
- Run the image to start a container
- Test that your app runs successfully
- Debug if anything breaks

✅ Step-by-Step Instructions

3.1 Build the Docker Image

Use the docker build command:

```
docker build -t demo-app .
```

Let's break this down:

- docker build — Tells Docker to build an image
- -t demo-app — Names your image demo-app
- . — Specifies the build context (current folder)

If successful, you'll see output similar to:

```
Step 1/7 : FROM node:18-alpine
 ---> c92c00334ec6
Step 2/7 : WORKDIR /app
 ---> Using cache
...
```

```
Successfully built 123456789abc
Successfully tagged demo-app:latest
```

3.2 Run the Container

Once the image is built, start a container from it:

```
docker run --rm -it demo-app
```

Explanation:

- --rm — Removes the container after it exits
- -it — Interactive mode (optional for simple apps)
- demo-app — The name of the image you built

If your app is a simple "Hello World" app like in index.js, you should see:

```
Hello from Dockerized App!
```

🎉 Congrats! Your app just ran **inside a container**, isolated from your host machine.

3.3 Run in Detached Mode (Optional)

If your app is a server (e.g., Express or Flask), you can run it in detached mode:

```
docker run -d -p 3000:3000 demo-app
```

Explanation:

- -d — Detached mode (runs in background)
- -p 3000:3000 — Maps container's port 3000 to your local machine's port 3000

Then go to http://localhost:3000 and test the app.

3.4 View Running Containers (Optional)

```
docker ps
```

You'll see active containers with their ID, image name, ports, and more.

To stop a running container:

docker stop <container-id>

🧯 Common Troubleshooting

Issue	Solution
Error: Cannot find module	Ensure all files are copied correctly in Dockerfile
App doesn't start or crashes	Check CMD or ENTRYPOINT command in Dockerfile
Port not accessible	Make sure you exposed the port correctly and mapped it with -p flag
App runs locally but not in Docker	Could be missing environment variables or incorrect working directory

🧼 Clean Up (Optional)

To free up space:

- Remove all stopped containers:

docker container prune

- Remove unused images:

docker image prune

✅ What You've Achieved

- ✅ You built a Docker image of your app
- ✅ You successfully ran your app in a container
- ✅ You tested the app inside an isolated, reproducible environment

This means your app is now **container-ready** and you can confidently move to CI/CD!

☁️ Step 4: Push Your Project to GitHub

This step is all about **getting your code from your local machine into the cloud** via GitHub. This is important because GitHub will serve as the central source for your CI/CD workflow in GitHub Actions.

📦 What You'll Do:

- Create a new GitHub repository
- Link your local project to that repo
- Push all your code (including the Dockerfile)
- Verify everything is in place

🛠️ Step-by-Step Instructions

4.1 Create a New Repository on GitHub

1. Go to https://github.com
2. Click on **+** in the top-right → **New repository**
3. Fill in details:
    - **Repository name**: demo-app
    - **Description**: *A simple Dockerized Node.js app with CI/CD*
    - **Public** or **Private** (your choice)
    - **Uncheck** the option to add a README (you already have files locally)
4. Click **Create repository**

GitHub will now show you instructions to push an existing project. Keep this page open!

4.2 Link Local Repo to GitHub

In your local terminal (inside the demo-app folder):

git remote add origin https://github.com/your-username/demo-app.git

Replace your-username with your actual GitHub username.

4.3 Set Main Branch and Push

git branch -M main

git push -u origin main

This sets main as the default branch and pushes all your local commits to GitHub.

4.4 Confirm on GitHub

Go to your GitHub repository URL in the browser. You should now see:

📁 index.js

📁 package.json

📁 Dockerfile

📁 .gitignore

All your project files should be there, perfectly synced.

💡 Bonus Tip: Add a README

While optional, it's a good idea to help others (and future-you) understand the repo.

Create a README.md file:

echo "# Demo App 🚀" > README.md

git add README.md

git commit -m "Add README"

git push

✅ What You've Achieved

- ✅ Your local project is now live on GitHub
- ✅ It's ready to trigger workflows through GitHub Actions
- ✅ It acts as a **single source of truth** for your application

Now you're all set for the magic of automation.

⚙️ Step 5: Create a GitHub Actions Workflow File

This step is where your **CI/CD pipeline begins**. You'll define an automated workflow that builds your Docker image, tests it (if needed), and optionally pushes it to a container registry.

GitHub Actions uses a special folder and YAML files to manage workflows.

🎯 What You'll Do:

- Create a .github/workflows/ folder in your project
- Add a workflow YAML file (e.g., ci.yml)
- Set up a basic pipeline to:
  - Trigger on every push to main
  - Set up Node.js
  - Build the Docker image

🛠️ Step-by-Step Instructions

5.1 Create Workflow Folder & File

In your project directory:

```
mkdir -p .github/workflows
touch .github/workflows/ci.yml
```

5.2 Add Workflow YAML Content

Open ci.yml and paste the following:

```
name: CI - Docker Build

on:
```

```
  push:
    branches:
      - main
  pull_request:

jobs:
  build:
    runs-on: ubuntu-latest

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

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'

      - name: Install dependencies
        run: npm install

      - name: Run app test (optional)
        run: echo "No tests yet..."

      - name: Build Docker image
        run: docker build -t demo-app .
```

🔍 What This Workflow Does:

Section	Purpose
on:	Triggers the workflow on push to main or on pull requests
jobs:	Defines a job named build
runs-on:	Uses an Ubuntu runner (virtual machine)
actions/checkout	Pulls your latest code from GitHub
setup-node	Installs Node.js so it can run npm install
docker build	Builds your Docker image using the Dockerfile

🚀 Push the Workflow to GitHub

```
git add .github/workflows/ci.yml
git commit -m "Add GitHub Actions workflow"
git push
```

As soon as you push, GitHub will trigger this workflow. You can watch it run:

1. Go to your repo on GitHub
2. Click on the **Actions** tab
3. You'll see your workflow running in real-time 🚦

📘 Optional: Add Tests Later

If your app has tests (like using Jest for Node), you can add a test command:

```
- name: Run tests
  run: npm test
```

For now, we've added a placeholder step for simplicity.

✅ What You've Achieved

- ✅ You've created your first automated CI/CD pipeline with GitHub Actions
- ✅ The pipeline builds your Docker image every time you push code
- ✅ You've unlocked true automation and consistency in your workflow

🧪 Step 6: Define Build and Test Steps in the Workflow

Now that your GitHub Actions workflow builds your Docker image, it's time to **make it smarter** by adding **build validations** and **automated tests** (if you have them). This step helps catch bugs early—before they ever reach staging or production.

🎯 What You'll Do:

- Add test scripts (for Node.js or any language you're using)
- Update the GitHub Actions workflow to:
  - Run those tests
  - Fail fast if anything breaks
  - Separate build from test logic

🔍 Why Tests in CI Matter?

CI stands for *Continuous Integration*. It's all about:

- Ensuring **code quality**
- Making sure **new code doesn't break existing logic**
- Letting you move **fast and safely**

🛠️ Step-by-Step Instructions

6.1 Add a Simple Test (Optional for Node.js)

If you're using Node.js, let's use jest for a basic test setup.

First, install it:

npm install --save-dev jest

Update package.json:

```
"scripts": {
  "start": "node index.js",
  "test": "jest"
}
```

Create a test file:

```
mkdir tests
touch tests/basic.test.js
```

Inside tests/basic.test.js:

```
test('adds two numbers', () => {
  expect(1 + 2).toBe(3);
});
```

Run locally to verify:

```
npm test
```

6.2 Update GitHub Actions Workflow to Include Tests

Now go to .github/workflows/ci.yml and update it like this:

```
name: CI - Docker Build and Test

on:
  push:
    branches:
      - main
  pull_request:

jobs:
  build-and-test:
```

```
runs-on: ubuntu-latest

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

  - name: Set up Node.js
    uses: actions/setup-node@v3
    with:
      node-version: '18'

  - name: Install dependencies
    run: npm install

  - name: Run unit tests
    run: npm test

  - name: Build Docker image
    run: docker build -t demo-app .
```

💡 **How It Works Now:**

Step	Purpose
✅ Checkout	Gets your code from GitHub
✅ Setup Node.js	Sets the right Node version
✅ Install	Installs app dependencies

Step	Purpose
✅ Run Tests	Automatically runs your test suite
✅ Build Docker	Only happens **after tests pass**

So, if a test fails, your pipeline **stops immediately**. This saves time and prevents bad code from reaching staging or production.

🧠 (Optional) Add Code Coverage or Linting

You can extend this further:

```
- name: Run linter
  run: npm run lint


- name: Generate code coverage
  run: npm run test -- --coverage
```

✅ What You've Achieved

- ✅ You've added a test layer to your pipeline
- ✅ You've enforced automatic quality checks for all code pushes
- ✅ Your pipeline is now smart enough to protect your project

🐳 📦 Step 7: Build and Push Docker Image to a Container Registry

Now that your CI pipeline builds and tests your app, the next big move is to **publish your Docker image** to a container registry—so it can be pulled later by staging or production environments.

We'll use **Docker Hub** here for simplicity, but you can adapt it for GitHub Container Registry (GHCR), AWS ECR, GCP GCR, or Azure ACR as well.

🎯 What You'll Do:

- Create a Docker Hub repository
- Authenticate inside GitHub Actions
- Build your Docker image with a version tag
- Push it to the Docker Hub registry

🛠️ Step-by-Step Instructions

7.1 Create a Docker Hub Repository

1. Go to https://hub.docker.com
2. Log in and click **Repositories** > **Create Repository**
3. Name it something like:
   yourusername/demo-app
4. Set visibility (public/private) and click **Create**

7.2 Add Docker Hub Credentials to GitHub Secrets

In your GitHub repo:

1. Go to **Settings** > **Secrets and Variables** > **Actions**
2. Click **New repository secret**

3. Add the following secrets:
    - DOCKER_USERNAME: your Docker Hub username
    - DOCKER_PASSWORD: your Docker Hub password or a personal access token

7.3 Update GitHub Actions Workflow

Open .github/workflows/ci.yml and update the file to:

```
name: CI - Build, Test & Push Docker Image

on:
  push:
    branches:
      - main
  pull_request:

jobs:
  build-test-push:
    runs-on: ubuntu-latest

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

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'
```

```
    - name: Install dependencies
      run: npm install

    - name: Run unit tests
      run: npm test

    - name: Log in to Docker Hub
      run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{ secrets.DOCKER_USERNAME }}" --password-stdin

    - name: Build Docker image
      run: docker build -t ${{ secrets.DOCKER_USERNAME }}/demo-app:latest .

    - name: Push Docker image to Docker Hub
      run: docker push ${{ secrets.DOCKER_USERNAME }}/demo-app:latest
```

🔒 Why Use GitHub Secrets?

Secrets keep your credentials safe—GitHub masks them and ensures they aren't exposed in logs.

🧪 Test Your Pipeline

Push your changes:

```
git add .
git commit -m "Add Docker Hub image push to CI"
git push
```

Then go to **Actions** tab on GitHub → click your running workflow. You'll see it:

✅ Logs in to Docker
✅ Builds your image
✅ Pushes it to Docker Hub!

Check hub.docker.com to confirm. Your new image will be live there.

✅ **What You've Achieved**

- ✅ Your pipeline now publishes Docker images automatically
- ✅ You've enabled true **Continuous Delivery**
- ✅ Your Docker image is now usable in any cloud or server deployment

☁️🐳 Step 8: Deploy to a Cloud Platform — AWS EC2 (In Detail)

🚀 Goal:

Deploy your Docker image (from Docker Hub) to an EC2 server so it runs 24/7 and can be accessed via a public IP or domain.

🔧 Pre-requisites

- A Dockerized app pushed to Docker Hub
- AWS account with EC2 access
- A public key-pair (.pem) to SSH into your EC2 instance
- Basic CLI knowledge

🛠️ Step-by-Step Guide

8.1 Launch an EC2 Instance

1. Go to: https://console.aws.amazon.com/ec2
2. Click **"Launch Instance"**
3. Fill details:
    - **Name:** docker-deploy-server
    - **AMI:** Choose **Ubuntu 22.04 LTS** (or Amazon Linux 2)
    - **Instance type:** t2.micro (free tier eligible)
    - **Key pair:** Create new or select existing (.pem file)
    - **Allow HTTPS and HTTP traffic in Security Group**
4. Click **Launch Instance**

📝 *Wait 1-2 mins for instance to initialize.*

8.2 SSH Into EC2

ssh -i path/to/your-key.pem ubuntu@<EC2-Public-IP>

- Replace <EC2-Public-IP> with your instance IP (from AWS dashboard)
- Use ec2-user for Amazon Linux, ubuntu for Ubuntu

8.3 Install Docker

For **Ubuntu**:

sudo apt update

sudo apt install docker.io -y

sudo systemctl enable docker

sudo systemctl start docker

Add your user to Docker group (optional):

sudo usermod -aG docker ubuntu

Log out and back in if needed to apply group change.

8.4 Pull Docker Image from Docker Hub

Make sure your image is public or you have credentials.

docker pull yourusername/demo-app:latest

Confirm it's pulled:

docker images

8.5 Run Docker Container

docker run -d -p 80:3000 yourusername/demo-app:latest

- -d: detached mode
- -p 80:3000: maps public port 80 to internal app port (e.g., 3000)

Check if it's running:

```
docker ps
```

Now go to:

📡 http://<EC2-Public-IP> → You should see your app live!

🛠️ Optional: Add Firewall Rule (If You Missed It)

If you didn't open port 80 earlier:

1. Go to AWS EC2 > **Security Groups**
2. Select the one attached to your instance
3. Click **Inbound Rules** > **Edit Rules**
4. Add:

Type	Protocol	Port Range	Source
HTTP	TCP	80	0.0.0.0/0
HTTPS	TCP	443	0.0.0.0/0

Save, then reload the browser.

💡 Optional Enhancements

➕ Add Docker Compose (For multi-service apps)

Create a docker-compose.yml:

```
version: "3"
services:
  app:
    image: yourusername/demo-app
    ports:
      - "80:3000"
```

Run it:

```
docker compose up -d
```

🔐 Add Domain & HTTPS (with NGINX + Certbot)

- Point your domain to EC2 IP via A record
- Install NGINX and Certbot
- Reverse proxy Docker container with HTTPS

Let me know if you want a **bonus guide** on this too 🔥

✅ What You've Achieved

- ✅ Created a scalable, production-like environment
- ✅ Deployed your Dockerized app to a live Linux server
- ✅ Learned how to run containers on cloud infra with root access

🌐 🔒 Step 9: Set Up Domain & HTTPS (Optional but Pro)

By default, platforms like Render or Railway give you a free subdomain like:

https://demo-app.onrender.com

That's cool for testing—but for a real project or product, you want:

https://www.yourbrand.com

And yes—**with HTTPS** (🔒 padlock)!

🎯 What You'll Do:

- Register or use an existing domain (from GoDaddy, Namecheap, Google Domains, etc.)
- Point it to your app hosted on Render (or other platform)
- Set up HTTPS automatically (free with Let's Encrypt)

🛠️ Step-by-Step Guide (Using Render)

9.1 Buy a Domain (If You Haven't)

You can buy a domain from any registrar. Some options:

- https://domains.google
- https://www.namecheap.com
- https://www.godaddy.com

Once you own a domain, you'll have access to:

- **DNS Management Panel**
- Ability to add **A Records** or **CNAME**

9.2 Add Custom Domain in Render

1. Go to your **Render Dashboard**

2. Open your deployed web service
3. Click **Settings > Custom Domains**
4. Add a domain like:
   www.yourdomain.com

Render will now give you instructions to add DNS records.

9.3 Add DNS Records in Your Domain Provider

In your domain registrar's DNS panel:

- If Render gives you a **CNAME**:
  - Point www to yourapp.onrender.com
- If Render gives you **A Records**:
  - Add two A records pointing to IP addresses they provide

Wait for DNS propagation (usually 5–15 mins, sometimes longer).

9.4 Enable HTTPS on Render

Render will automatically:

- Detect your domain
- Issue an SSL certificate using Let's Encrypt
- Redirect HTTP to HTTPS (you'll see 🔒)

You'll get:

https://www.yourdomain.com

Boom 💥 —live and secure!

✅ What You've Achieved

- ✅ You now have a **professional domain** name
- ✅ Your app is **secured with HTTPS**
- ✅ You can show off your project to the world like a pro 😎

🔄 Step 10: Automate Full CI/CD with GitHub Actions + EC2 Deployment

Goal: Every time you push code to GitHub, your EC2 server should automatically pull the latest image and restart the container. Fully hands-free deployment. ✨

🎯 What You'll Do:

1. Set up SSH access between GitHub Actions and EC2
2. Write a GitHub Actions workflow to:
    - Build/test your app
    - Push Docker image to Docker Hub
    - SSH into EC2
    - Pull updated image & restart container

🛠️ Step-by-Step Guide

10.1 Create SSH Key for GitHub Actions

On your **local machine** or any secure environment:

```
ssh-keygen -t rsa -b 4096 -f deploy-key
```

You'll get:

- deploy-key (private)
- deploy-key.pub (public)

10.2 Add Public Key to EC2 Instance

1. SSH into EC2:

```
ssh -i your-key.pem ubuntu@<EC2-IP>
```

2. Paste the public key into ~/.ssh/authorized_keys:

nano ~/.ssh/authorized_keys

Paste contents of deploy-key.pub and save.

Now GitHub will be able to SSH into the EC2 box.

10.3 Add Private Key and EC2 Details to GitHub Secrets

In GitHub repo:

Secret Name	Value
EC2_SSH_KEY	Contents of deploy-key
EC2_HOST	ec2-user@<EC2-IP>
EC2_PORT	22
DOCKER_USERNAME	Docker Hub username
DOCKER_PASSWORD	Docker Hub password or token

10.4 GitHub Actions Workflow: deploy.yml

```
name: CI/CD to EC2

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
```

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

    - name: Set up Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'

    - name: Install dependencies
      run: npm install

    - name: Run tests
      run: npm test

    - name: Log in to Docker Hub
      run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{ secrets.DOCKER_USERNAME }}" --password-stdin

    - name: Build Docker image
      run: docker build -t ${{ secrets.DOCKER_USERNAME }}/demo-app:latest .

    - name: Push Docker image
      run: docker push ${{ secrets.DOCKER_USERNAME }}/demo-app:latest

    - name: Deploy on EC2
```

```
      uses: appleboy/ssh-action@v1.0.0
      with:
        host: ${{ secrets.EC2_HOST }}
        username: ubuntu
        key: ${{ secrets.EC2_SSH_KEY }}
        port: ${{ secrets.EC2_PORT }}
        script: |
          docker pull ${{ secrets.DOCKER_USERNAME }}/demo-app:latest
          docker rm -f demo-app || true
          docker run -d -p 80:3000 --name demo-app ${{
secrets.DOCKER_USERNAME }}/demo-app:latest
```

✅ Workflow Explanation

- GitHub runs CI tests
- Pushes the Docker image
- SSHes into EC2
- Pulls the latest image
- Restarts the running container

🔥 Bonus: Lock It Down

- Use a firewall (like UFW) to limit access to only port 80 & 22
- Use NGINX as a reverse proxy with HTTPS (Let me know if you want a walkthrough)
- Monitor with tools like Prometheus, Grafana, or basic docker logs

🏁 What You've Built

- ✅ Full CI/CD DevOps Pipeline
- ✅ Dockerized microservice
- ✅ Tests + Linting
- ✅ Automated deployment to your own cloud infrastructure
- ✅ Production-ready setup with custom domain and HTTPS

✅ Conclusion

In today's fast-paced software world, it's not enough to just write code—you also need to deliver it efficiently, reliably, and securely. This guide walked you through a complete DevOps pipeline, transforming a basic application into a cloud-deployed, production-ready service.

We started by organizing our source code with GitHub, establishing a foundation of version control and team collaboration. From there, we added unit tests to ensure that every piece of code works as expected—because quality matters from the start.

Next, we moved into containerization with Docker, allowing us to package our app with all its dependencies, ensuring that it runs the same everywhere—from your laptop to the cloud. With Docker Hub, we made that container available globally, so it could be pulled and run on any server.

We didn't stop at manual deployment. By launching an AWS EC2 instance, we took the first real-world step into the cloud—where your app becomes accessible to the world. We configured the environment, deployed our Docker container, and verified it was working live over the internet.

Then came the real magic: **automation**. Using GitHub Actions, we built a Continuous Integration/Continuous Deployment (CI/CD) pipeline that runs tests, builds Docker images, and deploys the latest version to EC2—all triggered by a simple code push. This kind of automation reduces human error, speeds up delivery, and allows teams to focus on building, not babysitting servers.

We even touched on the polish: mapping a custom domain, enabling HTTPS, and preparing for future scalability. With this setup, you now have a professional-grade delivery system similar to what top tech companies use to ship features and updates daily.