Guidebook: Building a CI/CD Pipeline for a Python App on AWS EKS

This guide details the process of creating an automated pipeline that takes a Python application from a Git commit to a live deployment on a Kubernetes cluster.

Phase 1: Local Setup & Source Control

1. **Create Project Directory & Files:**
   - Open a terminal and create a project folder:

     ```
     mkdir cicd-project
     cd cicd-project
     ```

   - Create `main.py` with your FastAPI application:

     ```
     from fastapi import FastAPI
     app = FastAPI()
     @app.get("/")
     def read_root():
         return {"message": "Hello, World! CI/CD Pipeline is running."}
     ```

   - Create `requirements.txt` for dependencies:

     ```
     fastapi
     uvicorn[standard]
     ```

2. **Set Up GitHub Repository:**
   - Go to GitHub and create a new, empty repository (e.g., `cicd-app-repo`).
   - In your local project folder, initialize Git and push your code. Replace the URL with your own.

     ```
     git init -b main
     git add .
     git commit -m "Initial commit"
     git remote add origin https://github.com/YOUR_USERNAME/cicd-app-repo.git
     git push -u origin main
     ```

Phase 2: Containerization with Docker

1. **Create Docker-related Files:**
   - In your project root, create a `Dockerfile`:

     ```
     # Start with an official lightweight Python image
     FROM python:3.9-slim

     # Set the working directory inside the container
     WORKDIR /app

     # Copy the dependencies file first to leverage Docker's layer caching
     COPY requirements.txt .

     # Install the dependencies
     ```

```
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application code into the container
COPY . .

# Command to run the application when the container starts
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

- Create a `.dockerignore` file to keep the image small:

```
__pycache__/
*.pyc
.git
.vscode
Venv
```

2. **Test Docker Image Locally:**
    - Build the image:
      ```
      docker build -t cicd-app .
      ```
    - Run the container:
      ```
      docker run -p 8000:8000 cicd-app
      ```
    - Verify it's working by opening `http://localhost:8000` in your browser.
3. **Commit Docker Files to GitHub:**
   ```
   git add Dockerfile .dockerignore
   git commit -m "feat: Add Docker containerization"
   git push origin main
   ```

Phase 3: Initial AWS Setup (IAM & ECR)

1. **Create an IAM User for CLI Access:**
    - In the AWS Console, go to **IAM** -> **Users** -> **Create user**.
    - Name the user (e.g., `cli-user`) and attach the `AdministratorAccess` policy for this project. **Note:** In a real production environment, you would grant more restrictive permissions.
    - Create the user and go to the **Security credentials** tab. Create an **access key** and save the `Access Key ID` and `Secret Access Key` securely.
2. **Create ECR Repository:**
    - Go to the **Amazon ECR** service.
    - Create a **private** repository named `cicd-app`.

Phase 4: Building the CI Pipeline

1. **Create the Build Specification (`buildspec.yml`):**
    - In your local project, create `buildspec.yml`:

```
version: 0.2

phases:
  pre_build:
    commands:
      - echo Logging in to Amazon ECR...
      - aws ecr get-login-password --region $AWS_DEFAULT_REGION | docker login
--username AWS --password-stdin
$AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com
  build:
    commands:
      - echo Build started on `date`
      - echo Building the Docker image...
      - docker build -t $IMAGE_REPO_NAME:$IMAGE_TAG .
      - docker tag $IMAGE_REPO_NAME:$IMAGE_TAG
$AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com/$IMAG
E_REPO_NAME:$IMAGE_TAG
  post_build:
    commands:
      - echo Build completed on `date`
      - echo Pushing the Docker image to ECR...
      - docker push
$AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com/$IMAG
E_REPO_NAME:$IMAGE_TAG
```

- Commit and push this file to GitHub.

```
git add buildspec.yml
git commit -m "feat: Add buildspec for CodeBuild"
git push origin main
```

2. **Create the CodeBuild Project:**
    - In the AWS Console, go to **CodeBuild** -> **Create build project**.
    - **Project name:** `cicd-app-build`.
    - **Source:** Connect to your GitHub repository and select your `cicd-app-repo`. Enable the webhook to rebuild on every code push.
    - **Environment:** Use a **Managed image** (Amazon Linux 2, Standard). **Crucially, check the "Privileged" box.**
    - **Environment variables:** Add the following:
        - `AWS_ACCOUNT_ID`: *Your 12-digit AWS Account ID*
        - `AWS_DEFAULT_REGION`: *Your AWS region code (e.g., `ap-south-1`)*
        - `IMAGE_REPO_NAME`: `cicd-app`
        - `IMAGE_TAG`: `latest`
    - Create the build project.

3. **Grant ECR Permissions:**
    - After the project is created, a new IAM role will be generated. Go to the **IAM** console, find this role (`codebuild-cicd-app-build-service-role`), and attach the

AmazonEC2ContainerRegistryPowerUser policy.

4. **Create the CodePipeline:**
   - Go to **CodePipeline** -> **Create pipeline**.
   - **Name:** cicd-app-pipeline.
   - **Source stage:** Select **GitHub (Version 2)** and choose your repository and main branch.
   - **Build stage:** Select **AWS CodeBuild** and choose the cicd-app-build project you just created.
   - **Deploy stage: Skip** this stage for now.
   - Create the pipeline. It will run automatically and push an image to your ECR repository.

Phase 5: Kubernetes Cluster Setup (EKS)

1. **Create the EKS Cluster:**
   - Go to **Amazon EKS** -> **Create cluster**.
   - **Name:** cicd-cluster.
   - Follow the wizard to create the necessary **Cluster IAM role**.
   - In the **Compute** section, create a **Node group**. Let the wizard create the **Node IAM role**. Use t3.small instances and set the desired size to 2.
   - Create the cluster. This will take 10-20 minutes.
2. **Tag Public Subnets for Load Balancer:**
   - While the cluster is creating, go to the **VPC** console.
   - Find the VPC created for your cluster. Go to **Subnets**.
   - Identify your **public subnets** (those with a route to an Internet Gateway igw-).
   - For each public subnet, go to the **Tags** tab and add the following tag:
     - **Key:** kubernetes.io/role/elb
     - **Value:** 1

Phase 6: Connecting kubectl to EKS

1. **Configure Local CLI:**
   - Install the AWS CLI and kubectl on your machine.
   - Configure your local AWS CLI with the credentials of the IAM user you created in Phase 3.

   Bash
   ```
   aws configure
   ```
2. **Set Up Cluster Access (aws-auth):**
   - This step must be done from **AWS CloudShell**, which has initial admin access.
   - Open CloudShell and create a file named aws-auth-cm.yaml with the complete, correct configuration:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: aws-auth
  namespace: kube-system
data:
  mapRoles: |
    - rolearn: YOUR_NODE_IAM_ROLE_ARN
      username: system:node:{{EC2PrivateDNSName}}
      groups:
        - system:bootstrappers
        - system:nodes
  mapUsers: |
    - userarn: YOUR_CLI_USER_ARN
      username: cli-user
      groups:
        - system:masters
```

Replace `YOUR_NODE_IAM_ROLE_ARN` *and* `YOUR_CLI_USER_ARN` *with the actual ARNs.*

- Apply this configuration in CloudShell:

```
kubectl apply -f aws-auth-cm.yaml
```

3. **Connect Local `kubectl`:**
   - Back on your local machine, run the command to update your kubeconfig file:

```
aws eks update-kubeconfig --region YOUR_REGION --name cicd-cluster
```

   - Test the connection:

```
kubectl get nodes
```

You should see your two nodes listed.

Phase 7: Deploying the Application

1. **Create Manifest Files:**
   - In your local project folder, create `deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cicd-app-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cicd-app
  template:
    metadata:
      labels:
        app: cicd-app
    spec:
      containers:
      - name: cicd-app
        image: YOUR_AWS_ACCOUNT_ID.dkr.ecr.YOUR_REGION.amazonaws.com/cicd-app:latest
        ports:
        - containerPort: 8000
```

- Create service.yaml, including the annotation to ensure an internet-facing load balancer:

```
apiVersion: v1
kind: Service
metadata:
  name: cicd-app-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-scheme: internet-facing
spec:
  selector:
    app: cicd-app
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8000
```

Remember to replace the placeholders in deployment.yaml.

2. **Apply Manifests:**
   - From your local terminal, apply the manifests:

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

Phase 8: Verification

1. **Check Pods and Service:**

```
kubectl get pods
kubectl get service cicd-app-service
```

2. Wait a few minutes for the EXTERNAL-IP of the service to be assigned.
3. Copy the DNS name and paste it into your browser. You should see your application running.

Phase 9: Project Cleanup

To avoid costs, delete all the resources you created, starting with the resources in Kubernetes and then moving to AWS.

1. **Delete K8s LoadBalancer & Deployment:** `kubectl delete service cicd-app-service` and `kubectl delete deployment cicd-app-deployment`.
2. **Delete EKS Cluster:** In the AWS Console.
3. **Delete ECR Repository:** In the AWS Console.

4. **Delete CodePipeline & CodeBuild Project:** In the AWS Console.
5. **Delete S3 bucket** created by CodePipeline.
6. **Delete IAM Roles** created during the setup.