Project Documentation: Building an End-to-End CI/CD Pipeline on AWS

Project Goal: To create a fully automated Continuous Integration (CI) and Continuous Deployment (CD) pipeline to build, containerize, and deploy a Python web application onto a Kubernetes cluster.

Core Technologies:

- **Application:** Python (FastAPI)
- **Source Control:** Git & GitHub
- **Containerization:** Docker
- **CI/CD:** AWS CodePipeline, AWS CodeBuild
- **Artifact Registry:** Amazon ECR (Elastic Container Registry)
- **Orchestration:** Kubernetes on Amazon EKS (Elastic Kubernetes Service)

Phase 1: Local Application Setup & Version Control

Step 1: Create the Python Application

- **Action:** A simple web application was created using the FastAPI framework in a file named `main.py`. A `requirements.txt` file was created to define dependencies (`fastapi`, `uvicorn`).
- **Goal:** To have a functional application that can be run locally.

Step 2: Initialize Git Repository

- **Action:** A local Git repository was initialized in the project directory. The initial application files were committed.
- **Hurdle:** The `git init` command was accidentally run in the main user directory (~) instead of the project folder. This caused Git to try and track all personal files.
- **Solution:** The incorrect `.git` folder was deleted from the home directory (`rm -rf ~/.git`), we navigated into the correct `cicd-project` folder, and then re-initialized the repository and committed the files successfully.

Phase 2: Containerization with Docker

Step 3: Create the Dockerfile

- **Action:** A `Dockerfile` was created to define the steps for building a container image of the Python application. It specified a Python base image, copied the code, installed dependencies, and set the command to run the app. A `.dockerignore` file was also created to exclude unnecessary files.

- **Goal:** To create a portable, self-contained artifact of the application.

Step 4: Build and Run the Docker Image

- **Action:** The `docker build` command was used to create the image, and `docker run` was used to test it locally.
- **Hurdle:** The application was not accessible in the browser when navigating to `0.0.0.0:8000`.
- **Solution:** The difference between a server listening address (`0.0.0.0`) and a client connection address (`localhost` or `127.0.0.1`) was clarified. Navigating to `http://localhost:8000` successfully accessed the application running in the container.

Phase 3: Cloud Setup and Continuous Integration (CI)

Step 5: Set Up Cloud Repositories

- **Action:** The plan was to create a repository in AWS CodeCommit for the source code and a repository in Amazon ECR for the Docker images.
- **Hurdle:** AWS CodeCommit was no longer available for new customers, preventing repository creation.
- **Solution:** We pivoted to using **GitHub** as the source code repository, which is a common industry practice. The local repository was linked and pushed to a new GitHub repository, and the Amazon ECR repository was created as planned.

Step 6: Build the CI Pipeline

- **Action:** The CI pipeline was constructed using **AWS CodePipeline** and **AWS CodeBuild**. A `buildspec.yml` file was created to define the build, tag, and push commands for the Docker image.
- **Hurdle 1:** The AWS UI for connecting to GitHub was slightly different than expected, causing initial confusion.
- **Solution 1:** We navigated the UI together, selecting the correct `GitHub (via OAuth app)` option to use the previously authorized connection.
- **Hurdle 2:** The initial pipeline run failed because the CodeBuild project's default IAM role lacked permission to push images to ECR.
- **Solution 2:** We navigated to the IAM console, found the role created for CodeBuild (`codebuild-cicd-app-build-service-role`), and attached the `AmazonEC2ContainerRegistryPowerUser` managed policy to grant the necessary permissions. The pipeline then succeeded, and a new image appeared in ECR.

Phase 4: Kubernetes Cluster Setup (EKS)

Step 7: Create the EKS Cluster

- **Action:** An Amazon EKS cluster was created using the AWS Management Console. This involved creating the EKS control plane, a new VPC, and a node group with `t3.small` instances.
- **Hurdle:** The wizard required an IAM role for the cluster, which didn't exist yet.
- **Solution:** We used the "Create recommended role" option, which redirected to the IAM console. We followed the wizard to create the `eks-cluster-role` with the `AmazonEKSClusterPolicy` and then returned to the EKS setup to select it.

Step 8: Configure Local `kubectl` Access

- **Action:** The goal was to connect the local machine's `kubectl` to the new EKS cluster.
- **Hurdle 1:** The `aws eks update-kubeconfig` command failed with an `AccessDeniedException` because the IAM user (`abhay-iam`) was not authorized to perform `eks:DescribeCluster`.
- **Solution 1:** An inline IAM policy was added directly to the `abhay-iam` user, granting the specific `eks:DescribeCluster` permission for our new cluster.
- **Hurdle 2:** After fixing the IAM permission, `kubectl get nodes` failed with an error "the server has asked for the client to provide credentials." This pointed to an issue with Kubernetes authorization, not AWS IAM.
- **Hurdle 3:** The `aws-auth` ConfigMap, which maps IAM users to Kubernetes users, was not found in the cluster.
- **Solution 2 & 3:** We used the AWS CloudShell, which had initial admin access to the cluster. We created a new `aws-auth-cm.yaml` file from scratch, ensuring it correctly mapped both the `eks-node-role` (to allow nodes to join) and the `abhay-iam` user (to grant admin access via `system:masters`). After applying this clean ConfigMap, the local `kubectl` connection finally succeeded.

Phase 5: Continuous Deployment (CD)

Step 9: Deploy the Application to Kubernetes

- **Action:** Two manifest files, `deployment.yaml` and `service.yaml`, were created. The deployment defined how to run the application pods using the image from ECR. The service was set to `type: LoadBalancer` to expose the application to the internet.
- **Hurdle:** After applying the manifests, the service's `EXTERNAL-IP` remained in a `<pending>` state for a long time. The `kubectl describe service` command revealed a `FailedBuildModel` error because the controller could not find subnets tagged for an `internal-elb`.
- **Solution:**

1. **Wrong Tag:** We first discovered the subnet tags were incorrect (key was `kubernetes.io` instead of `kubernetes.io/role/elb`).
2. **Explicit Annotation:** After correcting the tag, the issue persisted. The final fix was to explicitly tell Kubernetes to create an **external** load balancer by adding an annotation to the `service.yaml` file: `service.beta.kubernetes.io/aws-load-balancer-scheme: internet-facing`.
3. After applying the annotated service file, the AWS Load Balancer was successfully created, and an external IP was assigned.

Step 10: Verify the Full Pipeline

- **Action:** A code change was made to the `main.py` file and pushed to GitHub. The CI pipeline ran successfully, building a new image.
- **Hurdle:** The change did not appear in the browser after the CI pipeline finished.
- **Solution:** We identified that because the image tag `:latest` was being used, Kubernetes did not automatically pull the new version. The deployment was manually updated using `kubectl set image...`, and the pods were deleted (`kubectl delete pods --all`) to force a restart with the new image. This made the final changes appear in the browser, successfully validating the entire workflow.