GROUP 5 [VAIBHAV KHURANA & ROSS BARBA]

2/19/24

ML OPS PROJECT REPORT

PROJECT SCOPE

The overarching goal of this project was twofold: educational and application oriented. From an educational standpoint, the aim was to acquire hands-on experience with modern deployment technologies and methodologies, including containerization with Docker, cloud services via Azure, and container orchestration using Kubernetes. The project sought to deepen understanding of these technologies' principles, workflows, and best practices in a real-world application scenario.

Actual Project Objectives:

- **To develop an image classifier** using a pretrained ResNet model that can accurately categorize images.
- **To learn and implement containerization** by encapsulating the Flask-based application in Docker containers, facilitating portability and consistency across development, testing, and production environments.
- **To utilize cloud services for deploying the containerized application**, specifically focusing on Azure Container Registries for image storage and management.
- **To orchestrate containers using Azure Kubernetes Services (AKS)** for automated deployment, scaling, and management of the application, ensuring high availability and resource efficiency.
- **To apply these technologies in addressing a hypothetical business problem**, such as creating a scalable online image classifier service. This service aims to offer real-time image classification capabilities to users or businesses through a web interface.

EXPLORING POSSIBLE SOLUTIONS

Before finalizing the chosen solution, several deployment strategies were evaluated:

- **On-Premises Deployment:** Considered for its direct control over infrastructure but ruled out due to its lack of scalability and higher maintenance requirements.
- **Cloud-Based Virtual Machines:** While offering scalability, the lack of containerization would not provide the desired consistency across environments or the hands-on experience with Docker.
- **Containerization with Docker on a Single Cloud Provider:** A simpler approach that introduces containerization but lacks the orchestration and scalability advantages of Kubernetes.
- **Containerization with Docker and Orchestration with Kubernetes on Azure:** *Selected* for its educational value in learning advanced deployment techniques and its alignment with the project's practical objectives.

BUSINESS PERSPECTIVE

Our project focuses on developing a scalable online image classifier that provides instant classification services for various images uploaded by users. This service aims to cater to diverse needs, from educational purposes to assisting businesses in categorizing their digital assets. The integration of this classifier into a web-based platform allows for easy access and utilization, offering a valuable tool for real-time image analysis.

---- Below is a comparison summary of the project, before and after execution. ----

BEFORE

Pros:

- **Educational Value:** Provides an in-depth learning opportunity in deploying and managing cutting-edge software applications, specifically focusing on real-time image classification services.
- **Scalability and Flexibility:** Expected to efficiently address the need for scaling an online image classifier service, accommodating varying user demands and traffic spikes without compromising performance.
- **Operational Efficiency:** Predicted enhancements in deployment speed and environmental consistency, enabling rapid iterations and deployments of the image classifier across different platforms.

Cons:

- **Complexity and Learning Curve:** The integration of Docker, Azure services, and Kubernetes presents a complex learning landscape, potentially slowing initial progress as students navigate through the intricacies of each technology.
- **Upfront Investment:** Requires a significant investment in terms of time and resources for acquiring the necessary skills, setting up the cloud infrastructure, and configuring the deployment pipeline.

AFTER

Pros:

- **Achievement of Learning Objectives:** The project successfully met its educational goals, providing hands-on experience with Docker, Azure, and Kubernetes, and understanding how these technologies can be leveraged to deploy an online service.
- **Business Problem Solving:** The developed image classifier service showcases the potential to offer a valuable tool for users and businesses alike, enabling real-time image classification through a web-based interface, thus enhancing user engagement and operational capabilities.
- **Cost and Operational Efficiency:** Benefited from AKS's ability to dynamically scale resources, effectively managing operational costs while ensuring the service remains responsive to user demand.

Cons:

- **Ongoing Complexity:** Continuous learning needed for managing evolving cloud and container technologies.
- **Variable Costs:** Active cost management required to mitigate fluctuating expenses.
- **Security and Compliance:** Constant attention to security and regulatory compliance is essential in cloud deployments.

PROJECT COMPONENTS

We will delve into each file, examining their individual use cases and how they integrate to establish a workflow. This overview will illuminate the distinct functionalities each component contributes to the project, from facilitating user interactions to orchestrating deployment and service management, ensuring a robust and scalable image classification solution.

Project Files:

- **app.py:** Defines a Flask web application that employs a pretrained ResNet101 model from the PyTorch library for image classification. It uses torchvision and PIL for image preprocessing and classification.
- **class_labels.txt:** Contains mappings from model output indices to human-readable class labels, enabling the application to provide understandable predictions.
- **credentials.txt:** Likely holds authentication credentials for accessing cloud services or APIs, although its specific contents weren't reviewed.
- **Deployment.yaml:** Kubernetes deployment configurations specifying how the Flask application and its services are deployed and managed within an AKS environment along with load balancer for HTTP.
- **Dockerfile:** Instructions for building a Docker container image of the Flask application, ensuring consistency across development, testing, and production environments.
- **requirements.txt:** Lists all Python dependencies required by the Flask application, ensuring they are installed within the Docker container or any virtual environment.
- **index.html:** The HTML template rendered by the Flask application on the homepage, providing a user interface for uploading images to classify.

Integration and Workflow:

- The Flask application (app.py) serves as the core, leveraging a ResNet101 model for image classification tasks.
- Users interact with the application through a web interface (index.html), where they can upload images to be classified.
- The application processes these images, utilizing the ResNet101 model to predict the image's class, and maps these predictions to human-readable labels using class_labels.txt.
- The Dockerfile outlines the steps to containerize the Flask application, ensuring that the environment is consistent and that all dependencies listed in requirements.txt are included.
- The Kubernetes configuration file (Deployment.yaml) detail how the containerized application is deployed and managed on AKS, facilitating scalability, high availability, and efficient resource utilization.

Cloud Platform Integration

Starting with the chosen technologies and infrastructure, we aimed to leverage the best practices in development and deployment, ensuring our application's efficiency, scalability, and robustness in a cloud environment.

Key deployment technologies used:

- Flask: Python web framework for the backend and model serving.
- Docker: Container platform for application consistency and deployment ease.
- Azure Container Registry (ACR): Service for storing Docker images.
- Azure Kubernetes Service (AKS): Kubernetes-based service for container orchestration.

Docker and Azure's cloud services were pivotal in our project infrastructure, ensuring our application's consistent performance across different environments and enhancing scalability and manageability. By encapsulating the application in Docker's lightweight containers and leveraging Azure's ACR and AKS for deployment, we achieved a streamlined and robust deployment pipeline, optimizing our application for the cloud environment.

Deployment Workflow Enhancement through Azure

- The project's deployment workflow began with the creation of an Azure Container Registry, a secure repository for Docker images. This step was pivotal in managing and storing the application's container images in a centralized cloud location. Subsequently, Azure Kubernetes Services was configured to orchestrate the application's deployment, leveraging container orchestration to enhance availability and scalability.

- The Dockerfile, detailing the application's environment setup, played a crucial role in building the Docker image. This image encapsulation was then tagged to associate it with the Azure Container Registry, preparing it for cloud deployment. Following this, the image was pushed to ACR, ensuring its availability for deployment across various environments.

- To bridge the deployment process, AKS was authorized to access the images stored in ACR. This authorization was critical for a smooth deployment, allowing AKS to pull the necessary images directly from ACR. The deployment phase was executed using Kubernetes YAML configuration file, specifying the deployment and service details. These files directed AKS to deploy the application according to predefined specifications, ensuring the application was deployed efficiently and managed effectively in the cloud.

- This cloud platform integration showcases a streamlined approach to deploying and managing a web-based image classification service. By leveraging Azure services, the project achieved enhanced scalability, security, and ease of deployment, underscoring the value of cloud technologies in modern software development and deployment strategies.

DETAILED FLOW OF DEPLOYMENT

This section outlines the step-by-step process used for deploying our image classification application using Azure services, Docker, and Kubernetes. Before execution, certain prerequisites must be met:

- **Azure Account:** An active Azure subscription is necessary to access Azure services such as Azure Container Registry (ACR) and Azure Kubernetes Service (AKS).
- **Azure CLI:** The Azure Command Line Interface must be installed and configured on your local machine to execute Azure service commands.
- **Docker:** Docker needs to be installed to build, tag, and push Docker images to the Azure Container Registry.
- **Kubernetes CLI (kubectl):** This command-line tool allows you to run commands against Kubernetes clusters, essential for deploying and managing applications on AKS.
- **Knowledge of YAML:** Understanding YAML syntax is important for creating and modifying the Kubernetes deployment and service configuration files.

Once you've met the requirements there are two ways to get started, either using Azure CLI or using the Azure portal dashboard.

For Dashboard use this:

Step 1: Create a Resource Group

- Sign in to the Azure Portal.
- Navigate to Resource groups from the Azure services dashboard.
- Click + Create, fill in the details (Subscription, Resource group name, and Region), and then click Review + create.

Step 2: Create Azure Container Registry (ACR)

- In the Azure Portal, search for Container registries and select it.
- Click + Create. Choose the previously created resource group, enter a unique name for your ACR, and select a location and SKU (e.g., Basic).
- Review the settings and click Create.

Step 3: Create Azure Kubernetes Service (AKS) Cluster

- Search for Kubernetes services in the Azure Portal and select it.
- Click + Create. Select the same resource group, provide a name for your AKS cluster, and choose the region.
- In the Node size, you can select the default size or choose a different size based on your needs. Then, specify the number of nodes.
- Review and adjust any other settings as necessary, then click Review + create.

For Azure CLI use this:

Step 1: Create Resource Group

- Command: az group create --name <ResourceGroupName> --location <Location>
- Establish a foundational resource group in Azure to manage all related resources collectively.

Step 2: Create Azure Container Registry (ACR)

- Command: az acr create --resource-group <ResourceGroupName> --name <ACRName> --sku Basic
- This step initializes a private Docker container registry within Azure for storing and managing our Docker images.

Step 3: Create Kubernetes Cluster

- Command: az aks create --resource-group <ResourceGroupName> --name <ClusterName> --node-count 1 --generate-ssh-keys --attach-acr <ACRName>
- A Kubernetes cluster is created to manage the deployment and scaling of our containerized application.

We will be using Azure CLI for further steps

Step 4: Accessing the Resources

- Commands:
    - az aks get-credentials --resource-group <ResourceGroupName> --name <ClusterName>
    - docker login <ACRLoginServer>
- These commands configure access to the Kubernetes cluster and the Azure Container Registry.

Step 5: Logging into ACR using Docker.

- Utilize the username and password from the Azure container registry dashboard for Docker login.

Step 6: Tagging Docker Image for ACR

- Command: docker tag <DockerImageName> <ACRLoginServer>/<ImageName:tag>
- Tag the Docker image with the ACR's login server address to prepare it for pushing to the registry.

Step 7: Verify Tagged Docker Image

- Command: docker images
- Confirm the successful tagging of the Docker image in the list of local Docker images.

Step 8: Pushing Docker Image to ACR

- Command: docker push <ACRLoginServer>/<ImageName>
- Push the tagged Docker image to the Azure Container Registry to make it available for deployment.

Step 9: Deploy Using YAML File

- Command: kubectl apply -f <DeploymentYAMLFile>
- Deploy the application to the Kubernetes cluster using the specified YAML configuration file.

Step 10: Monitoring Deployment Status

- Utilize Kubernetes commands such as kubectl get pods or kubectl get services.

These steps collectively form the deployment pipeline, ensuring a smooth transition from development to production in a cloud environment, leveraging the integrated capabilities of Azure, Docker, and Kubernetes.

OBSTACLES AND ERRORS

During the deployment process, we encountered several challenges that served as valuable learning experiences. Two primary issues were the difficulties in resolving accurate dependencies for our `requirements.txt` file and a misconfiguration in our Kubernetes YAML files, specifically the omission of a load balancer, which led to an `ImagePullBackOff` error in Azure Kubernetes Service (AKS). Additionally, the critical step of authenticating AKS to use Azure Container Registry (ACR) was initially overlooked, further complicating the deployment.

- ***Resolving Dependencies in `requirements.txt`***
  - The `requirements.txt` file is crucial for ensuring that all necessary Python packages are available for our application to run. We encountered errors due to version conflicts and the unavailability of some dependencies for the specific versions listed. This obstacle was overcome by meticulously testing each package version in a local development environment before finalizing the `requirements.txt` file, ensuring compatibility and stability across all dependencies.

- ***Missing Load Balancer Configuration***
  - The initial deployment to AKS failed to expose our application to the internet, resulting in an `ImagePullBackOff` error. This was traced back to a missing load balancer configuration in our Kubernetes YAML deployment file. Kubernetes services require a load balancer to distribute incoming traffic among pods. By updating our `Deployment.yaml` file to include a load balancer, we resolved the connectivity issue, allowing AKS to successfully pull the Docker image from ACR and expose the service externally.

- ***Authenticating AKS to Use ACR***
  - A critical step in the deployment process is ensuring AKS has permission to access and pull Docker images from ACR. Initially, our Docker image did not reflect in the AKS deployment due to missing authentication between AKS and ACR. This was resolved by configuring the proper role-based access control (RBAC) permissions, allowing AKS to authenticate with ACR. Utilizing the Azure CLI, we assigned the AKS service principal the necessary roles to access ACR, effectively resolving the `ImagePullBackOff` error and enabling the seamless deployment of our Dockerized application.

These obstacles underscored the importance of thorough planning, testing, and configuration in the deployment process. Each challenge was a learning opportunity that improved our understanding of cloud-based deployment, containerization, and the intricacies of integrating various Azure services.

***** Important Lesson Learnt:*** *Reading the cloud provider's documentation and using available resources is vital for effective deployment. These guides offer insights into best practices, troubleshooting, and service usage, ensuring smoother project execution and problem resolution.* ********