Machine Learning Application Deployment using Cloud-Native Tools

Team Members -

1. Jack Gaul : jgaul@scu.edu
2. Pranav Bhendawade: pbhendawade@scu.edu
3. Laysa Malladi: smalladi2@scu.edu
4. Meghshanth Sara: msara@scu.edu

Abstract:

When it comes to deploying the Machine Learning Model, there are challenges like how to scale the model, how the model can interact with different services within or outside the application, how to achieve repetitive operations, etc. To overcome this problem, containerization technologies such as Docker and Kubernetes are the best fit.Our main objective is to explore different virtualization technologies learned in class and to learn how to deploy machine learning applications using Flask,Docker and Amazon EKS.

Introduction:

The applications in the real world are continuously growing, and the data, any application processes is enormous. As the applications' size increases, it becomes hard to manage, scale, monitor, and maintain them reliably. Many cloud-native tools and technologies, such as Docker, Kubernetes, and VMs (AWS EC2), are used widely in the industry to handle and run such massive applications. To get hands-on experience with these cloud-native tools and technologies, we are building an end-to-end pipeline to deploy a machine learning model, in any of the existing public cloud services like AWS.

Background and Related Work:

In recent years, machine learning has transitioned from a field of academic research interest to a field capable of solving real-world business problems. However, the deployment of machine learning models in production systems can present a number of issues. This project tries to implement the deployment of one such machine learning application deployment. We have designed an architecture to set up a kubernetes cluster for any machine learning application to use. The machine learning solutions in a variety of use cases, industries and applications and extracts practical considerations corresponding to stages of the machine learning deployment workflow. The main goal of our project is mainly on setting up the automated infrastructure for any kind of real-time machine learning application using our designed architecture.

To start with our project, we referred to various research papers and blogs to understand the issues and existing solutions in the area of machine learning model deployment. We found following three ways to deploy the machine learning model in the production environment -

1. Deploying machine learning models as a web service
2. Deploying machine learning models for batch prediction
3. Deploying machine learning models on edge devices as embedded models

After careful discussion, we decided to follow the first way mentioned above, deploying machine learning models as a web service, as suitable for our use case.

Approach:

Technologies Used:

- Flask
- GUnicorn Server
- Docker
- Amazon EC2
- Amazon Elastic Kubernetes Service(EKS)

1. **Machine Learning Application:**
   We used an existing Machine Learning Application that predicts the CO2 emission from vehicles.We built a flask web app framework to create a simple web application for predicting the CO2 emission using Machine Learning Model that takes input of values - 1. Engine Size 2. Cylinders 3. Fuel Consumption from the user through a simple Web UI. The LinerRegression() class from Sklearn library was used to build the linear regression model. We have used the `FuelConsumption.csv` file as a dataset to train the Machine Learning Model. The dataset which is used to train the model consists of 1067 rows and 4 columns. Engine Size, Cylinders, and Fuel Consumption are the three inputs given to the linear regression model which then predicts the CO2 emission.

2. **Flask:**
   Flask is a micro web framework, it's a Python module that lets you develop web applications easily. It has a small and easy-to-extend core: it's a microframework that doesn't include an ORM (Object Relational Manager) or such features.Flask creates a web server locally to test our application.We have used flask to test our machine learning web application on our local machine.

   The following code snippet show the `app.py`file which acts as server initialization file -

```python
#import libraries
import numpy as np
from flask import Flask, request, jsonify, render_template
import pickle

#Initialize the flask App
app = Flask(__name__)

#default page of our web-app
@app.route('/')
def home():
    return render_template('index.html')
```

3. **GUnicorn:**

   The standard web servers such as Apache, and NGINX don't know how to communicate with your Python applications. Python uses WSGI (Web Server Gateway Interface) for implementation of the web server. WSGI, is a set of rules which allow a WSGI compliant server to work with a WSGI compliant Python application. WSGI also handles scaling for web servers to be able to handle thousands of requests so you don't have to think about accepting multiple requests at a time. Gunicorn is one implementation of a WSGI server for Python applications. Gunicorn is a WSGI compliant web server for Python Applications that receives requests sent to the Web Server from a Client and forwards them onto the Python applications or Web Frameworks (such as Flask or Django) in order to run the appropriate application code for the request.

   In this project, we have used a Gunicorn server to deploy our web application in a production-like environment. After we tested the application using Gunicorn server locally and in AWS EC2 instance we decided to containerize our application to maintain and deploy it as a micro-services.

4. **Docker**

   Docker helps you create a single deployable unit for your application. This unit, also known as a container, has everything the application needs to work. This includes the code (or binary), the runtime, the system tools and libraries. Packing all the requirements into a single unit ensures an identical environment for the application, wherever it is deployed. It also helps to maintain identical development and production setups. Containers also eliminate a whole class of issues caused by files being out of sync or due to subtle differences in the production environments. We have used the docker platform to containerize our web application. Following is the Dockerfile we have written to create a docker image of our application -

```
FROM python:3.9.15-slim

RUN pip install pipenv

WORKDIR /app

COPY ["Pipfile", "Pipfile.lock", "./"]

RUN pipenv install --deploy --system

COPY . .

ENTRYPOINT ["gunicorn", "--bind=0.0.0.0:9696", "app:app"]
```

We have build the docker image using following command -

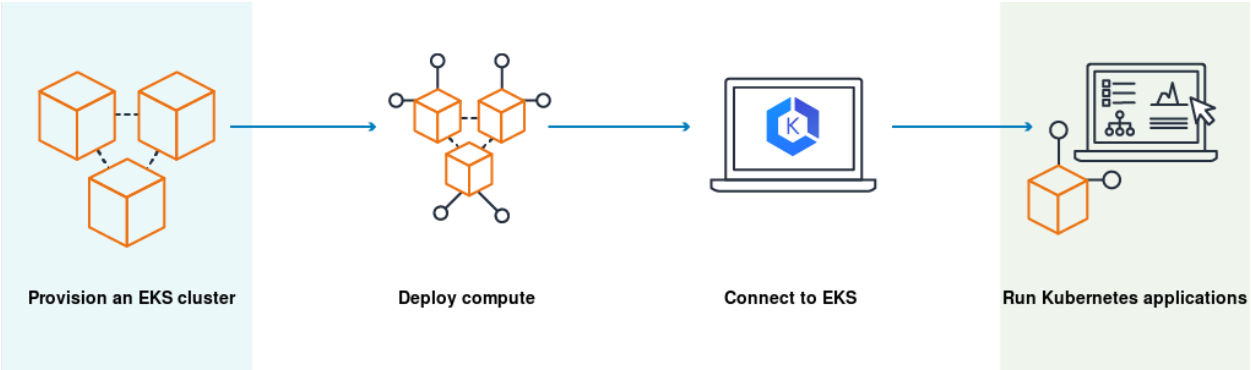
$ docker build -t flask-app:1.1 Dockerfile

After building the docker image using the above command, we pushed the docker image on the Docker Hub (https://hub.docker.com/). Here's the link for the docker image of our application - https://hub.docker.com/repository/docker/pranav2306/ml-model-app.

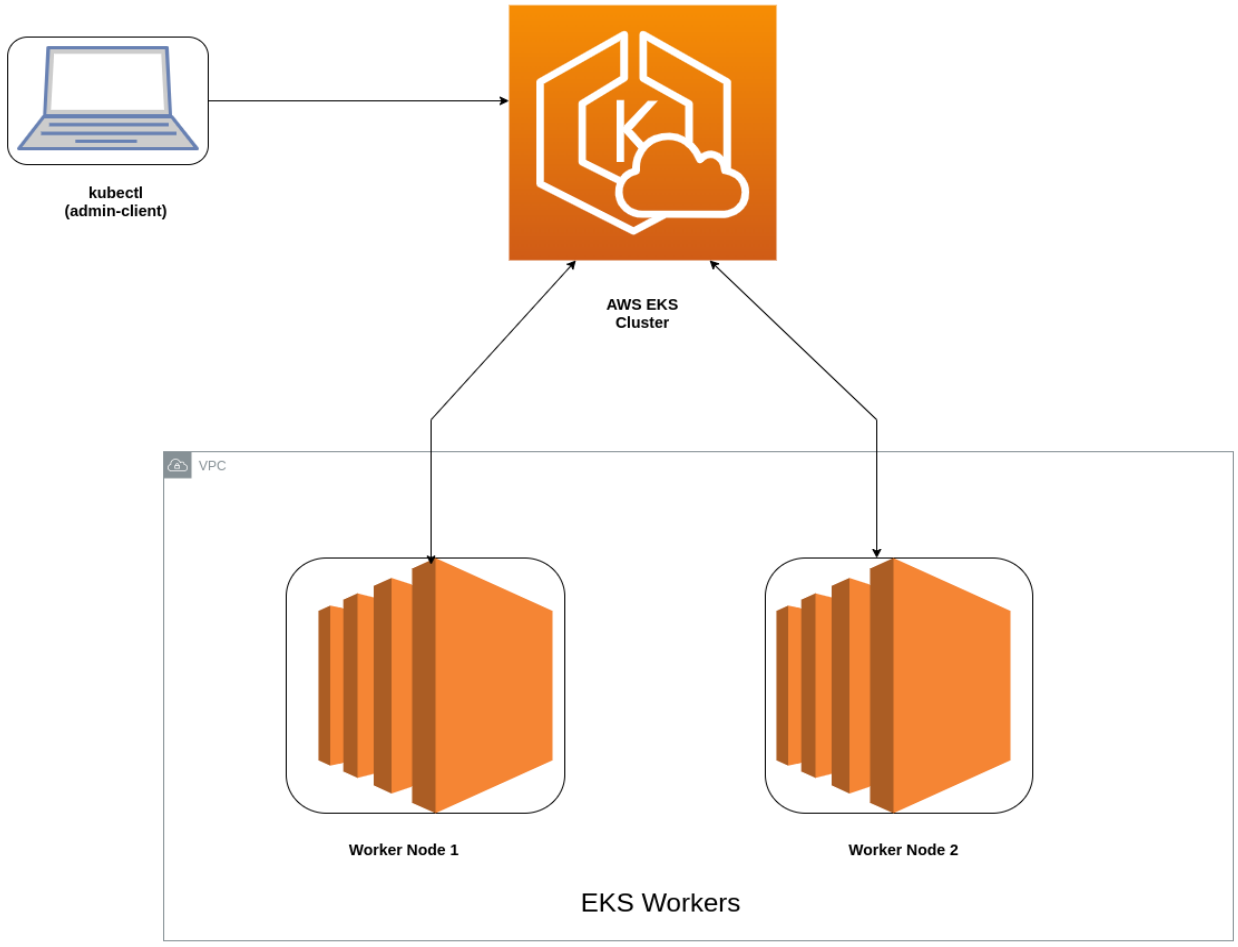
5. **Amazon EC2 -**
   Amazon Elastic Compute Cloud (Amazon EC2) provides scalable computing capacity in the Amazon Web Services (AWS) Cloud. Using Amazon EC2 eliminates your need to invest in hardware up front, so you can develop and deploy applications faster. We have used two EC2 instances as a worker node in the kubernetes cluster launched by using AWS EKS (Elastic Kubernetes Services).

6. **AWS Elastic Kubernetes Services (EKS) -**
   Amazon Elastic Kubernetes Service (Amazon EKS) is a managed service that you can use to run Kubernetes on AWS without needing to install, operate, and maintain your own Kubernetes control plane or nodes. Kubernetes is an open-source system for automating the deployment, scaling, and management of containerized applications. Amazon EKS runs and scales the Kubernetes control plane across multiple AWS Availability Zones to ensure high availability.

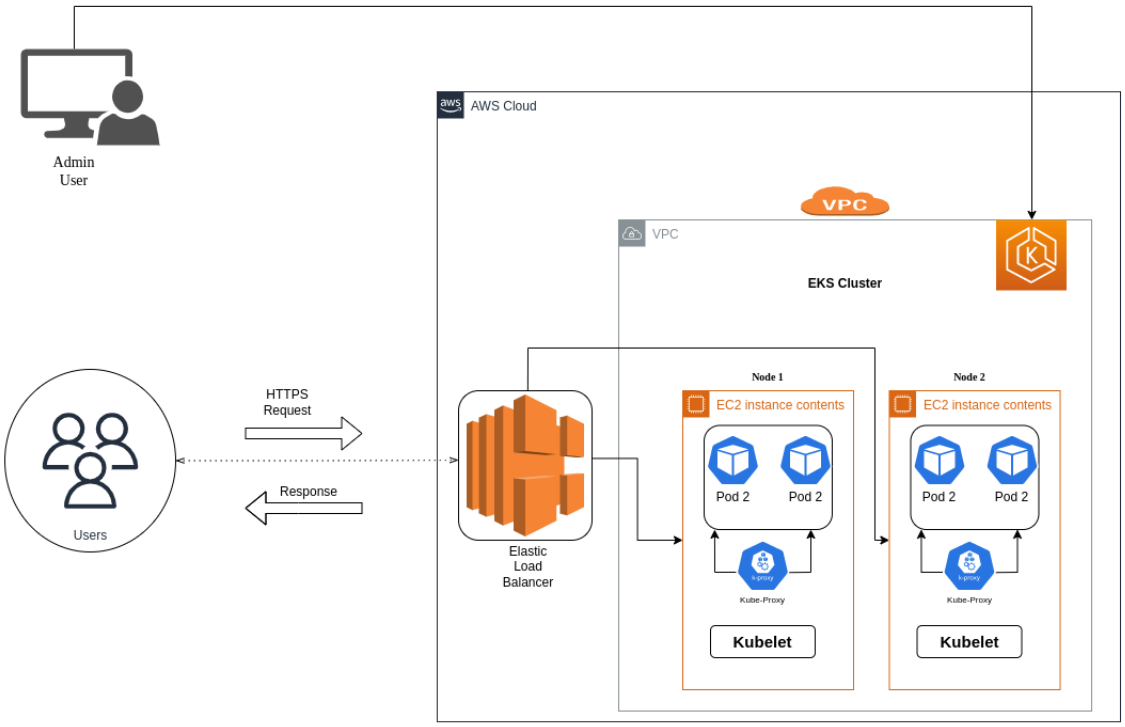
   Following diagram shows the basic working of AWS EKS -



We have used EKS to set up a kubernetes cluster of two EC2 worker nodes. The below diagram shows the architecture of a kubernetes cluster set up using AWS EKS -



Software Architecture:



Outcome:

In this project we used AWS services for all our cloud technologies and we used docker as our container technology. I will first go over our container that runs our application.

We built our application using docker image “python:3.9.15-slim” as a base image. This enabled us to run our flask-based machine learning web application with flask being a python module.

For our VMs that we used for our worker nodes, we used EC2. We ran these nodes using the predefined Ubuntu Server 22.04 LTS (HVM), SSD Volume Type. This is on a 64-bit architecture. The specific instance type was a t2.micro. The t2.micro is one of the lowest tiers offered by AWS but was more than enough for our application and ML model. They had 1 vCPU and 1 GiB of memory. We just gave each of these a storage volume of 8 GiB. For our current implementation, we were not storing any kind of state information so we didn't need any sort of large DB or storage capacity.

Our kubernetes deployment was also done in AWS using their Elastic Kubernetes Service. The cluster we utilized used Kubernetes version 1.23. Our K8s Yaml can be seen below. In this file, we outline the app to be flask-ml-app and we launch it using the container hosted in Pranav’s

docker profile called: ml-model-app:1.0. For our implementation, we chose to invoke two replicas to show the ability of the load balancer.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-app-deployment
spec:
  selector:
    matchLabels:
      app: flask-ml-app
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: flask-ml-app
    spec:
      containers:
        - name: guniorn-server
          image: pranav2306/ml-model-app:1.0
          ports:
            - containerPort: 9696
```

To show how our project was scalable, which was a key feature and motivation for our architecture, we deployed the cluster with a load balancer service. The load balancer lives in its own node. This next yaml file outlines that service we created. Furthermore, it behaves as a proxy to change the port of the service. All the worker node's applications are listening for HTTP requests on port 9696. The load balancer here changes the destination port number to be the standard HTTP, port 80.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ml-model-service
spec:
  selector:
    app: flask-ml-app
  ports:
    - port: 80
      targetPort: 9696
  type: LoadBalancer
```

Analysis and Future Work:

1.Configure Prometheus to monitor, trace, and analyze the kubernetes cluster -

Currently, we do not have proper monitoring and alert system setup for our project to track events relevant to your application, such as memory consumption, network utilization, or individual incoming requests. Here, Prometheus can come to rescue. Prometheus stores events in real-time. Prometheus determines the current value of your metrics by using a pull-based data fetching mechanism. It'll periodically poll the data source that backs each metric, then store the result as a new event in the time-series database. We can configure the Prometheus to monitor, trace, and analyze our application's Kubernetes cluster.

2. Configure Grafana to visualize the charts, graphs, and alerts of the kubernetes cluster fetched by prometheus -

Grafana will connect to Prometheus. It is an open-source solution and also allows us to write plug-ins for integration with various data sources from scratch. The tool supports the study, analysis, and monitoring of data over a period of time called time-series analytics.
It helps track user behavior, application behavior, error frequency in production or in the pre-provided environment, the type of errors in which related data appears, and contextual scenarios. In addition to the core open-source solution, the Grafana Cloud & Enterprise team offers two other services for businesses. We are planning to configure Grafana to visualize the charts, graphs, and alerts of the kubernetes cluster fetched by prometheus.

3.Provision and manage Kubernetes clusters on AWS EKS and interact with your cluster using the Kubernetes Terraform provider -

While we are currently using the built-in kubectl and aws-cli for connecting to EKS clusters, Terraform, if used in future, will provide you with several benefits:

- Unified Workflow - You can also deploy applications into your EKS cluster using Terraform.
- Full Lifecycle Management - Terraform doesn't only create resources, it updates, and deletes tracked resources without requiring you to inspect the API to identify those resources.
- Graph of Relationships - Terraform understands dependency relationships between resources.

Conclusion :

Summary -

Overall, we were very satisfied with our project deployment. We set out to take a machine learning model and deploy it in a container. We wanted to deploy this model and container in the cloud so that it was easily accessible to the public internet. Furthermore, we wanted to explore how we could take a generalized model and build infrastructure to make an application scalable which we also achieved. This project allowed us to explore what AWS offered out of the box. We treated this project as a vehicle in which to explore and learn brand new conceptions and ideas that we covered in our class periods.

Key Learnings:

1. Integrating a machine learning model into a web application created on Flask.
2. Containerizing a web application using Docker.
3. Using Elastic Kubernetes Service to run kubernetes on AWS.

Limitations:

Our application is not really scalable. It is scalable in the sense that we can easily change the number of pods running the ML web application but this will be done in a static fashion. Meaning, our implementation is not dynamic in its computational resource requirements. A truly scalable version of our application would be able to dynamically increase the number of pods deployed, increasing the availability of the service.

Another limitation that we came across was cost. To run the application for a whole month was going to cost us $145 dollars. If you really wanted to scale up this service, the financial side of the project really needs to be considered because AWS costs can stack up extremely fast.

References:

1. https://github.com/NakulLakhotia/deploheroku
2.https://k21academy.com/docker-kubernetes/amazon-eks-kubernetes-on-aws/?utm_source=youtube&utm_medium=referral&utm_campaign=kubernetes17_july21_k21
3. `
4. https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html
5. https://docs.docker.com/get-started/02_our_app/