# Deploying Webapps on Containers on AWS

Advanced Concepts of Cloud Computing

2025

**Abstract**

In this lab assignment, you will get hands-on experience running containers on AWS in order to serve web applications. In the first part of the assignment, you will have install docker. Next, you will deploy our custom containers on your instances in order to serve your web application. Your web application will be a simple flask app that runs inference on a ML model. You are asked to report your results and analysis by producing a report using LATEX format.

## 1 Introduction

Containerization is a lightweight virtualization technology that packages applications and their dependencies into isolated units called containers. These containers encapsulate everything needed to run an application, including code, runtime, system tools, and libraries, ensuring consistency across different computing environments. Containerization has gained widespread adoption in deploying microservices and web applications due to its efficiency, portability, and scalability. It allows developers to build, test, and deploy applications quickly and consistently across various platforms, from development laptops to production servers. In microservices architectures, each service can be containerized separately, enabling independent scaling and updates. For web applications, containerization simplifies deployment processes, improves resource utilization, and facilitates rapid scaling to meet fluctuating demand. Popular containerization platforms like Docker and container orchestration tools such as Kubernetes have further accelerated the adoption of this technology in modern software development and deployment practices.

For this assignment, you will use Docker to build your containers, deploy your containers on separate instances, and provide the deployed containers as web services that can be invoked by sending a request.

***It is important to stop EC2 instances when you are not working on them.*** You will have 50 CAD credits provided by AWS for three assignments of this course. You should manage your expenses carefully. The overall goals of this lab assignment are:

- Get hands-on experience running containers on AWS.
- Understand containerization technologies such as Docker.
- Run inference on ML models hosted on the cloud.

- Use the codebase you have developed for your 1st assignment to setup your instances.

# 2 Setting Up Docker

You will find detailed instructions about how to install and run Docker on an Ubuntu instance in here: Install Docker Engine on Ubuntu. You should use the "Install using the Apt repository" section. Then, build your own docker container using docker compose by following the instructions in here: Docker Compose overview. Follow the instructions in the "QuickStart" section in order to setup a single **flask** application in a container on your instance.

# 3 Experiments with Orchestration

As you are going to be using machine learning models inside your instances, you need to make sure to orchestrate which machine should respond to each request. A request should be sent to a machine that is not already processing a previous request. For this purpose, you will need to implement a simple cluster manager instance (orchestrator). The orchestrator is responsible for:

- Receiving the requests from users.
- Keeping track of which container is available for processing requests.
- Returning the results to the users.

## 3.1 Receiving requests from users

This is handled the same way as your previous assignment. You will run a simple Flask application on the orchestrator which accepts requests on port 80. The orchestrator will manage 4 worker instances. Each worker instance will be running two similar containers. The orchestrator will be monitoring which container in each instance is processing a request and which is free by using an internal status file. When a new request is received, the orchestrator will check the status file and if a container is free, will forward the request to the mentioned container. If no containers are free for processing an incoming request, the orchestrator will store the request in a queue and will forward it to a container as soon as one is free. Below is an example of a JSON file that is used to keep track of the status of workers:

```
1   {
2     "container1": {
3       "ip": "public_ip_of_worker1",
4       "port": "5000",
5       "status": "busy"
6     },
7     "container2": {
8       "ip": "public_ip_of_worker1",
```

```
        "port": "5001",
        "status": "free"
      },
      "container3": {
        "ip": "public_ip_of_worker2",
        "port": "5000",
        "status": "free"
      },
      "container4": {
        "ip": "public_ip_of_worker2",
        "port": "5001",
        "status": "free"
      },
      "container5": {
        "ip": "public_ip_of_worker3",
        "port": "5000",
        "status": "free"
      },
      "container6": {
        "ip": "public_ip_of_worker3",
        "port": "5001",
        "status": "free"
      }
    }
```

Listing 1: Orchestrator internal instance management example

Below is an example of the code you need to run on your orchestrator. Note that you need to write the code for calling your instance in the part that is written in red. Feel free to change or modify this example as you wish.

```
from flask import Flask, request, jsonify
import threading
import json
import time

app = Flask(__name__)
lock = threading.Lock()
request_queue = []

def send_request_to_container(container_id, container_info,
   incoming_request_data):
    print(f"Sending request to {container_id} with data: {
       incoming_request_data}...")
    # Put the code to call your instance here
    # This should get the ip of the instance, alongwith the port and
        send the request to it
    print(f"Received response from {container_id}.")

```

```python
def update_container_status(container_id, status):
    with lock:
        with open("test.json", "r") as f:
            data = json.load(f)
        data[container_id]["status"] = status
        with open("test.json", "w") as f:
            json.dump(data, f)

def process_request(incoming_request_data):
    with lock:
        with open("test.json", "r") as f:
            data = json.load(f)
        free_container = None
        for container_id, container_info in data.items():
            if container_info["status"] == "free":
                free_container = container_id
                break

        if free_container:
            update_container_status(free_container, "busy")
            send_request_to_container(
                free_container, data[free_container],
                   incoming_request_data
            )
            update_container_status(free_container, "free")
        else:
            request_queue.append(incoming_request_data)

@app.route("/new_request", methods=["POST"])
def new_request():
    incoming_request_data = request.json
    threading.Thread(target=process_request, args=(
       incoming_request_data,)).start()
    return jsonify({"message": "Request received and processing
       started."})

if __name__ == "__main__":
    app.run(port=80)
```

Listing 2: Orchestrator

## 3.2 Deploying an ML application on your workers using flask

This part is similar to how deployed your flask apps in your first assignment. However, instead of FastAPI we will be using flask. Also, instead of returning a simple string, your flask app calls a function that is responsible for running inference on an ML model and returns a JSON response. Below, is an example of the code you can use on your worker

instances:

```
from flask import Flask, jsonify
from transformers import DistilBertTokenizer,
   DistilBertForSequenceClassification
import torch
import random
import string

app = Flask(__name__)

# Load the pre-trained model and tokenizer
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-
   uncased')
model = DistilBertForSequenceClassification.from_pretrained('
   distilbert-base-uncased', num_labels=2)

def generate_random_text(length=50):
    letters = string.ascii_lowercase + ' '
    return ''.join(random.choice(letters) for i in range(length))

@app.route('/run_model', methods=['POST'])
def run_model():
    # Generate random input text
    input_text = generate_random_text()

    # Tokenize the input text and run it through the model
    inputs = tokenizer(input_text, return_tensors='pt', padding=True
       , truncation=True)
    outputs = model(**inputs)

    # The model returns logits, so let's turn that into
       probabilities
    probabilities = torch.softmax(outputs.logits, dim=-1)

    # Convert the tensor to a list and return
    probabilities_list = probabilities.tolist()[0]

    return jsonify({"input_text": input_text, "probabilities":
       probabilities_list})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)  # Adjust the port as needed
       for your setup
```

Listing 3: Worker instance

DistilBert is a small model and should run without any problems on a T2.large instance. However, you are allowed to use even smaller models as the purpose of this assignment is

getting familiar with ML model deployment and microservies. You can find more information about this model and how to use the HuggingFace library in here Distilbert, Huggingface transformers.

# 4 Instructions

You will need to setup five instances in total. One T2.large instance to act as the orchestrator, and 4 other T2.large instances to act as the workers. Each worker must run 2 containers that listen on different ports for incoming requests. For this assignment, there is no need to setup a loadbalanacer and target groups. The figure below displays the architecture of your cluster:

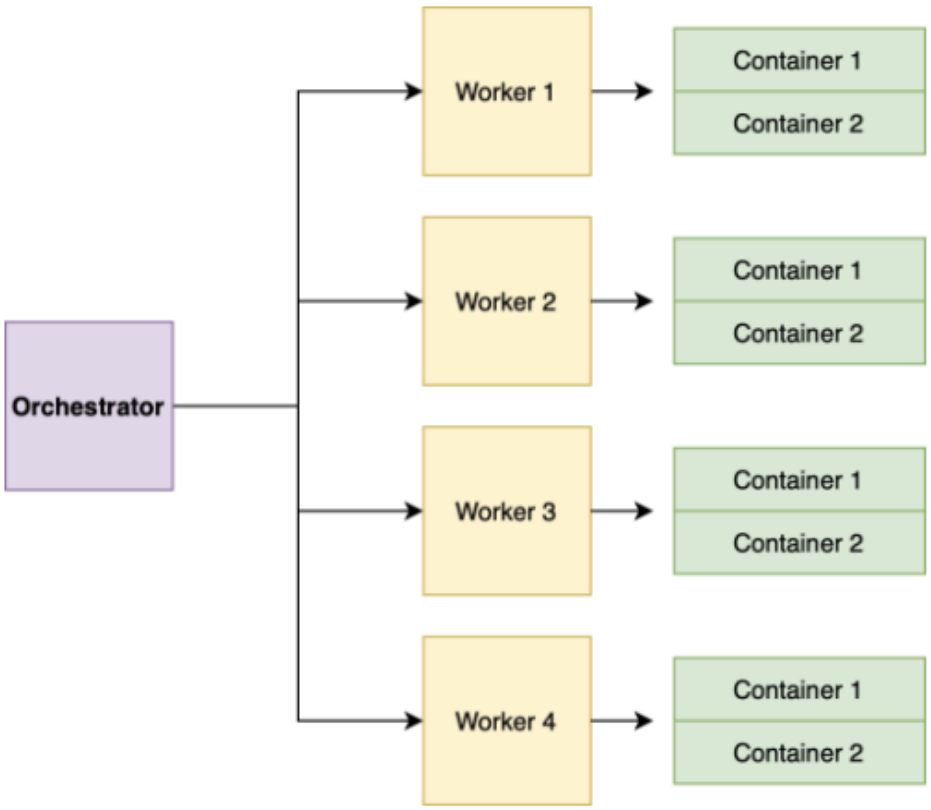


Figure 1: Architecture

# 5 How does this relate to your 1st assignment

In your first assignment, you practiced setting up your clusters and installing applications on them using code and running commands on your machines remotely. Here, the underlying principles are the same:

- Setting up the machines using code is the same.
- You have already used docker for your 1st assignment. Here, you need to customize your docker files to build the containers that are required for this assignment.
- You don't need to set up a loadbalancer.
- You will install the Flask, PyTorch, and HuggingFace libraries the same way that you installed them for your first assignment.
- You will send the requests to the orchestrator the same way that you did for your first assignment. Here, instead of a loadbalancer, the orchestrator will decide which instance and which container inside an instance should respond to the incoming request.

# 6 Automation and Infrastructure as Code

Your solution should be end-to-end automated. This means that during your demo, running the commands for downloading and setting up Docker, deploying the containers, sending the requests, and getting the results **SHOULD** be done by running a simple bash script. As such, you will need to develop your solutions using AWS' SDKs depending on the programming language that you prefer: AWS SDKs

# 7 Working in Groups

You should work in groups of two, three, or four.

# 8 Report

One submission per group is required for this assignment. To present your findings and answers questions, you have to write a lab report on your results using LATEX template. In your report should write:

- Experiments with your docker container.
- How did you compose your docker containers.
- How the orchestrator manages the queue when all instances are busy.
- How you deployed your flask application on the orchestrator.
- How did you deploy your flask apps on your workers.
- A sample of what your cluster returns for an incoming request.
- Summary of results and instructions to run your code.

One submission per group is required for this assignment. You must submit only one ZIPPED file named assignment1.zip which includes:

- report.pdf: Contains the name of the team members and all the material of the report. You can design the format and sections at your convenience.
- Your code: This will include all the necessary commands to execute and show the results of your benchmarking. It should have enough comments and descriptions to understand every single section of it.