

LOG8415  
Advanced Concepts of Cloud Computing

**Cluster Benchmarking using EC2 Virtual Machines  
and Elastic Load Balancer (ELB)**

Lab 2

Fall 2023

**By:**

Samy Cheklat - 1937812  
Ming Xiao Yuan - 1949477  
Cassy Charles - 1947025

**Presented to:**

Vahid Majdinasab

Saturday, November 11th 2023

To access the Github repo, click [here](#).

The link is [https://github.com/ming-xiao-yuan/deploying\\_ML\\_on\\_aws](https://github.com/ming-xiao-yuan/deploying_ML_on_aws)

## 1 Introduction

This lab assignment involves practical application of container technology on AWS to host web applications. The tasks include installing Docker, deploying custom containers on instances, and running a Flask app performing ML model inference. The final deliverable is a report formatted in LaTeX summarizing results and analysis.

## 2 Deployment of Machine Learning on AWS

To simplify the deployment of our application, we've opted to utilize Terraform. This tool, categorized as infrastructure as code, enables us to articulate cloud resources through configuration files.

Upon the successful creation of our EC2 virtual machines, where we granted traffic access on port 80 within the security group, we proceeded to connect to the instance for deploying Flask. Upon launching our Docker image on the instance, a command facilitates the automatic execution of our app.py program, enabling its exposure on the port 5000.

## 3 Flask application

The Docker container experiments involve setting up a Flask application within the container for running ML inference using a pre-trained DistilBert model. This includes defining routes for health checks and handling incoming requests to perform inference on randomly generated text.

Our Flask application functions as a worker instance dedicated to running machine learning (ML) inference using a pre-trained DistilBert model. We set up a **"run\_model"** route is configured to handle incoming POST requests for performing inference on randomly generated text. The application utilizes the Hugging Face Transformers library to load a DistilBert tokenizer and a DistilBertForSequenceClassification model with two output labels. The **"health\_check"** route allows for basic health checking, indicating the worker instance's status.

Upon receiving a POST request to **"run\_model"** the application generates a random input text using the **'generate\_random\_text'** function. It then tokenizes the input text and runs it through the pre-trained DistilBert model. The model returns logits, which are converted into probabilities using softmax. The resulting probabilities are transformed into a list and returned as a JSON response, including the original input text.

The server starts on the port 5000, listening on all network interfaces. This worker instance showcases its ability to process ML inference requests, providing a flexible and dynamic approach to generating probabilities for randomly generated text inputs.

## 4 Setup of orchestrator

In order to efficiently handle machine learning model requests within our instances, we're tasked with implementing a cluster manager, also known as an orchestrator. The orchestrator is like the brain overseeing the deployment and operation of containers. It receives requests, keeps track of container availability, and ensures that tasks are assigned to the right containers in an efficient manner. Essentially, it helps in orchestrating the flow of operations within a distributed system, ensuring everything works harmoniously. 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.

The orchestrator is responsible for overseeing four worker instances, with each worker running a pair of identical containers. It monitors the processing status of each container within an instance through an internal status file. Upon receiving a new request, the orchestrator checks this file. If a container is available, it promptly directs the request to the idle container. In cases where no containers are available, the orchestrator queues the request and dispatches it to an available container as soon as one becomes free.

To setup the orchestrator, we specified the base image for the container in a Dockerfile. The container will listen on the port 5000. This is typically the default port for Flask applications. A json file was provided to us for us to update the status and the ips of the containers. In order to change the status of these containers, we put together a code that updates the json file whenever we send requests and a container is busy.

```
def update_test_json_with_ips(worker_ips):
    # For this example, we're assuming each IP is sequentially assigned to two containers
    with lock:
        with open("test.json", "r") as file:
            data = json.load(file)

        # Map each IP to two containers
        for i, ip in enumerate(worker_ips):
            data[f"container{i*2+1}"]["ip"] = ip
            data[f"container{i*2+2}"]["ip"] = ip

        # Write the updated data back to the file
        with open("test.json", "w") as file:
            json.dump(data, file, indent=4)

    # Print the updated test.json to the logs
    app.logger.info("Updated test.json with new IPs:")
    app.logger.info(json.dumps(data, indent=4))
```

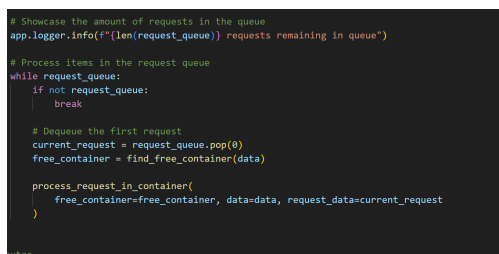
Figure 1: Updating the Json file

The code defines a route in the Flask application at the endpoint **receive\_ips\_from\_workers** with the HTTP method "POST". When a POST request is received at this endpoint, it retrieves the JSON data containing worker IP addresses. If worker IPs are present, it logs each IP, updates the JSON file with the received IPs and returns the result of the request. By processing that way, we are able to monitor which container is ready to handle incoming requests.

Part of the processing requests code was provided to us. However, we had to modify it to handle the queue. Additionally, the function logs information about the number of requests remaining in the queue using the Flask application's logger.

When all instances are busy, the orchestrator efficiently manages the queue by storing incoming requests in a queue data structure. It checks the availability status of containers using an internal status file. If a container is free, the orchestrator forwards the request to the available container; otherwise, it queues the request. The queued requests are processed as soon as a container becomes available.

It then enters a loop to process any requests in the queue, dequeuing and assigning them to available containers until the queue is empty. This mechanism allows for efficient handling of requests, ensuring that they are promptly processed by available containers or queued for future processing when resources become available.



```
# Showcase the amount of requests in the queue
app.logger.info(f'{len(request_queue)} requests remaining in queue')

# Process items in the request queue
while request_queue:
    if not request_queue:
        break

    # Dequeue the first request
    current_request = request_queue.pop(0)
    free_container = find_free_container(data)

    process_request_in_container(
        free_container=free_container, data=data, request_data=current_request
    )
```

Figure 2: Processing requests

At first, we encountered issues when sending the environment variables at first within our orchestrator. However, we were able to simulate concurrent requests to an orchestrator using a `ThreadPoolExecutor`. It defines a function called **‘make\_request’** to send a POST request to a specified URL with a dummy payload. The orchestrator’s DNS is retrieved from an environment variable. We then print the orchestrator’s DNS for verification. It constructs a URL for making requests to the orchestrator’s **“new\_request”** endpoint, and the number of concurrent requests is set to 15.

To parallelize these requests, we set up a list of 15 identical URLs. It then utilizes the `ThreadPoolExecutor` to concurrently execute the **‘make\_request’** function for each URL. The results, including the URL and the corresponding HTTP status code, are printed once the requests are completed.

The composition of our docker containers involves defining a Dockerfile specifying the base image, setting the working directory, copying necessary files, installing dependencies, exposing ports, and running the Flask application. For the worker instances, the Docker container is configured to utilize the Hugging Face Transformers library to load a DistilBert tokenizer and model.

## 5 Summary of results

The cluster’s response to an incoming request is a JSON object containing information about the processed request. It includes the input text generated for inference and the corresponding probabilities generated by the pre-trained DistilBert model.

The results from the cluster operations demonstrate the comprehensive monitoring and coordination facilitated by the Orchestrator. Logs from the Orchestrator, specifically when accessing the

Finally, the interaction involving hitting **"receive\_ips\_from\_workers"** highlights the orchestration of obtaining and communicating worker IP addresses to the Orchestrator. Together, these results showcase the efficiency and coordination achieved by the Orchestrator in managing the distributed system.

[illegible]

```
Iteration #2 :
{
  "workers info": {
    "container1": {
      "ip": "3.88.17.243",
      "port": "5000",
      "status": "busy"
    },
    "container2": {
      "ip": "3.88.17.243",
      "port": "5001",
      "status": "busy"
    },
    "container3": {
      "ip": "3.81.52.210",
      "port": "5000",
      "status": "busy"
    },
    "container4": {
      "ip": "3.81.52.210",
      "port": "5001",

```

```
Waiting for Orchestrator service to start...
Orchestrator service is now available.

Waiting for Worker Container 1 at 54.210.710.184 service to start...
Worker Container 1 at 54.210.710.184 service is now available.

Waiting for Worker Container 2 at 54.210.710.184 service to start...
Worker Container 2 at 54.210.710.184 service is now available.

Waiting for Worker Container 1 at 52.91.9.251 service to start...
Worker Container 1 at 52.91.9.251 service is now available.

Waiting for Worker Container 2 at 52.91.9.251 service to start...
Worker Container 2 at 52.91.9.251 service is now available.

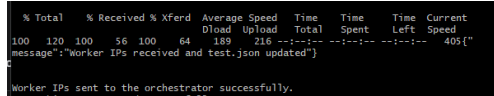
Waiting for Worker Container 1 at 18.212.116.110 service to start...
Worker Container 1 at 18.212.116.110 service is now available.

Waiting for Worker Container 2 at 18.212.116.110 service to start...
Worker Container 2 at 18.212.116.110 service is now available.

Waiting for Worker Container 1 at 52.55.52.247 service to start...
Worker Container 1 at 52.55.52.247 service is now available.

Waiting for Worker Container 2 at 52.55.52.247 service to start...
```

5



```

% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left     Speed
100 120 100    56 100    64    189    216  --:--:-- --:--:-- --:--:--    405("
message":"Worker IPs received and test.json updated")
}
Worker IPs sent to the orchestrator successfully.

```

Figure 6: Reception of ips from workers

## 6 Instructions to run the code

Prerequisite:

- Ensure that the source code is prepared on the local machine.
- Ensure that Docker is installed on the local machine.
- Ensure that terraform is installed on the local machine

### 6.1 From downloading the zip:

- Download the zip and unzip it.
- Within the folder, in a bash terminal, run the following command **bash scripts.sh**.

### 6.2 From cloning the repo :

- To execute the code, open Docker Desktop. Then, launch a bash command using the following command: **bash run.sh**
- To see the statuses of the workers, they are shown in the terminal of the docker container where the requests python application was ran.
- To see the logs of the orchestrator application on the EC2 instance, you can manually connect or through an ssh connection and enter the following command : **sudo docker logs "container-id"**.
- To execute a second time the requests application, in the **./scripts** folder, you can simply run **bash send\_requests.sh**. You can modify directly in the code the amount of concurrent requests you desire to send.