AWS LAB 1

October 17, 2022

Aura Kiiskinen - aura.kiiskinen@polymtl.ca
Cedric Helewaut - cedric.helewaut@polymtl.ca
Nick Reiter - nick.reiter@polymtl.ca
Victor Mesterton - victor.mesterton@polymtl.ca

Abstract

This paper presents the written elaboration of the first assignment of the Advanced Cloud Computing lecture of École Polytechnique de Montréal.

This assignment aims to set up automatically an Amazon Web Service (AWS) cloud computing architecture. The difficulty consists of connecting Ubuntu instances, target groups and an application load balancer in a suitable way. The practical draft of this assignment is developed in Python with the use of AWS' system development kit boto3.

1 Introduction

The target of the assignment was to create a single script that would set up automatically an Amazon Web Service (AWS) EC2 cloud computing architecture. This was done using python and AWS' system development kit Boto3. Boto3, a low-level Python library, is developed to create, configure, and manage in an object-oriented API manner AWS services.
In detail, this meant to install Flask on in advance created Ubuntu instances, which is done using a shell script. Then the instances were assigned to two target groups representing a cluster. These target groups should then be connected to an Application Load Balancer, which would assign and balance incoming `http` requests by means of a listener and its rules. The Performance of the clusters and instances were then supposed to be analysed using Cloud Watch.
The following sections of the documentation will explain each step in more detail. Firstly, the deployment and configuration of Flask on Ubuntu instances is given. Followed by the theoretical setup of the cloud computing architecture. Third, the results of the CloudWatch benchmarking metrics are evaluated. Lastly, the point by point instruction for running the code is explained.

2 Flask Application Deployment Procedure

To be able to install Flask on an EC2 Ubuntu instance, it is required to make use of scripting language commands that come in the form of a Unix shell (see Listing 1).
The first Unix command creates a project directory `flask_app` for the Flask application and navigates towards. Secondly, it all pre-existing packages of the instance are updated.

Furthermore, it is required to install the Python virtual environment library to be able to create a separate, independent developing environment. Thereupon, a virtual environment `venv` is compiled and activated. Flask is not comprised within the default stack of Python libraries and, hence, must be installed manually. `ec2_metadata` allows selecting the instance ID of the Ubuntu machine. To install a Flask server, a basic python script is coded that returns *Hello World from "Instance ID"*. Note, the Flask server is hosted on the IP address `0.0.0.0` and Port `80`. At last, the Python script is executed.

Listing 1: Flask Deployment on an EC2 Ubuntu instance

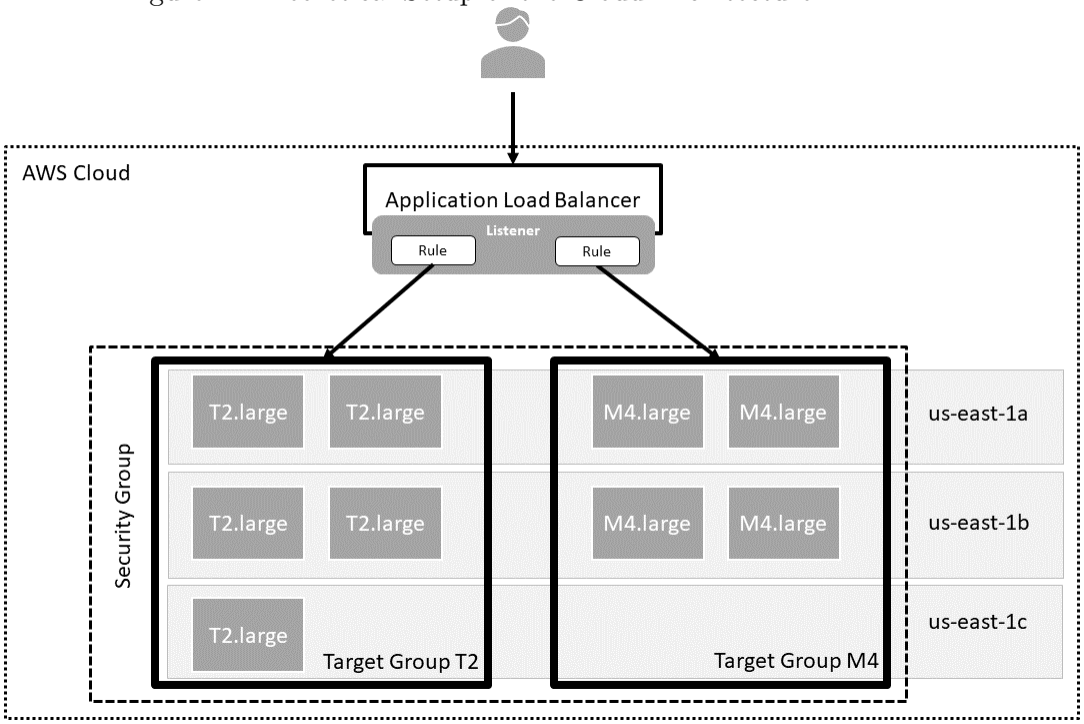
```
cd /home/ubuntu
mkdir flask_app
sudo apt-get update
yes | sudo apt-get install python3-venv
cd flask_app
python3 -m venv venv
source venv/bin/activate
pip install flask
pip install ec2_metadata

cat <<EOF >flask_app.py
from flask import Flask
from ec2_metadata import ec2_metadata
app = Flask(__name__)
@app.route('/')
def flask_app():
    return 'Hello, World from ' + ec2_metadata.instance_id
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=80)
EOF

flask --app flask_app run --host 0.0.0.0 --port 80
```

3 Cluster setup using Application Load Balancer

In this project report, a bottom up approach is used to create the cloud architecture in figure 1. So to say, the initialization begins with creating the Ubuntu instances, and ends by connecting the target groups with the listener rules.
To start, a security group is created that controls the traffic allowing to reach and leave the latter associated computing instances. The security group is assigned two inbound rules. One for the TCP protocol on Port 80 and the other one on Port 8080. Furthermore, all AWS availability zones, which are accessible from the current location, are read and the three following ones are chosen `us-east-1a, us-east-1b, us-east-1c` .
The two previously listed components, security groups and availability zones, are, thereupon, used to create five Ubuntu `T2.large` and four `M4.large` instances, which are equally distributed through the availability zones. All of these instances are assigned with the Unix shell code from listing 1.

The instances are used to create two clusters by assigning them to target groups. Generally speaking, a cluster consists of a homogenous composition of Ubuntu machines.
An application load balancers is created that automatically distributes with the help of a listener the incoming requests across the target groups and availability zones. First, the load balancer is compiled and then assigned to the three subnets. Second, a listener is initiated on port 80 of the load balancer. Through query string rules, the listener receives and forwards incoming HTTP queries sent by the user to the correct prevalent target groups.

Figure 1: Theoretical Setup of the Cloud Architecture

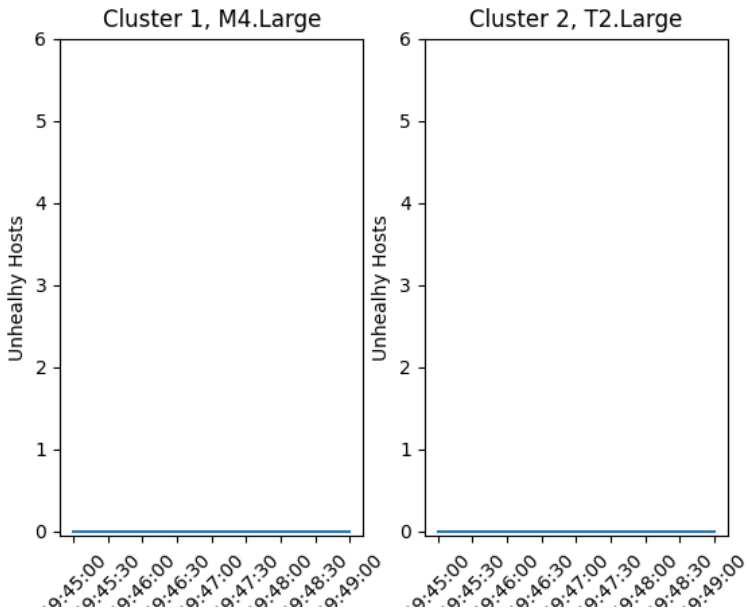


4 Results of benchmark

The developed automated script includes the surveillance with the help of an AWS CloudWatch metric and output on a web browser. 1200 queries are run against the different clusters and evaluated with CloudWatch's Unhealthy Host Count metric (figure 2). Throughout the whole time of the query period, none of the instances turned unhealthy. The satisfactory result can be traced back to strong computational power in contrast to the computing demand of the server requests. We exemplarily changed the Ubuntu instances to their micro and nano counterparts, which changed several times their states into unhealthy. It can be concluded that our approach is prepared for higher demanding computational tasks.

For further comparison, an extract of the cluster's metrics is given in figure 3. These can be retrieved from AWS' GUI console under target group and monitoring. Based on the insights, it is concluded that the response time and the request count per target of the `T2.Large` cluster instances are faster/higher compared to the ones of the `M4.Large` cluster. The differentiation

Figure 2: CloudWatch Metric: Unhealthy Host Count



of the metrics is derived by different distribution in terms of number of (`M4.Large` = 4 and `T2.Large` = 5)

5 Instructions to run code

This section provides a step-by-step instruction to run the given implementation with ease.

1. Download the AWS Command Line Interface and install it
2. Learner Lap ≫ AWS Details ≫ AWS CLI Show
3. Execute `aws configure` in the command line
4. Copy and paste the suiting credentials from step 2 and *region=us-east-1; output=json*
5. Unzip the compressed project file to yor location of wish
6. Navigate with your command line to the location where you stored the project folder
7. The script requires you to pip install `boto3, requests, matplotlib, tornado` via the command line. Ignore if already done.
8. Run the Python script with `python3 lab1_script.py` in your command line

Figure 3: CloudWatch Metrics: Extract from AWS' GUI console