LOG8415e
Advanced Concepts of Cloud Computing

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

Lab 1

Automn 2022

By:
Marc-Antoine Abou Jaoude
Roy Boutros
Anthony Sarkis
William Younanian

Presented to:
Vahid Majdinasab

Friday, October 7, 2022

Github link https://github.com/anthonysarkis/LOG8415-Assignment-1

1 Flask Application Deployment Procedure

We used a docker image to deploy the Flask application on the instances created on AWS. To do this, we used a docker file. First, we created a new repository and copied a requirements.txt file that contains all the dependencies we need with their respective versions. By running pip install with those requirements, we can download those dependencies in our environment. Our flask app is a simple app.py program that creates a webpage that shows the instanceId. When we run our Docker image on our instance, a CMD command allows us to automatically run our app.py program, which we can expose on the port 5000.

To simplify the deployment of our flask app, we published our Dockerfile on docker hub in the repository wayr/ec2_flask:latest.

Then, in our .sh file, we download docker. We then run the Docker image we pulled from wayr/ec2_flask:latest and pass it the instance id and instance type of the instance on which it is running. This allows us to display the instance id on our Flask application.

2 Cluster setup using Application Load Balancer

2.1 Terraform file

The code to create the infrastructure on AWS was made using Terraform. In our terraform file, we used "ressource" blocks to create our 9 instances. We specified the ami:ami-0149b2da6ceec4bb0, which corresponds to the ubuntu image that we use on those instances. We use the instance type to define 5 of our instances to be m4.large and four of our instances to be t2.large. We also use the availability_zone attribute to create our instances in two different physical regions, such as us-east-1c and us-east-1d. The userdata allows us to pass in the shell file (userdata.sh).

```
44  resource "aws_instance" "instance1" {
45    ami                    = "ami-0149b2da6ceec4bb0"
46    instance_type          = "m4.large"
47    vpc_security_group_ids = [aws_security_group.security_gp.id]
48    availability_zone      = "us-east-1c"
49    user_data              = file("userdata.sh")
      Marc-Antoine Abou Jaoude, 2 weeks ago | 1 author (Marc-Antoine Abou Jaoude)
50    tags = {
51      Name = "M4"
52    }
53  }
```

Figure 1: Creation of our instances of type m4.large

We used the default AWS Virtual Private Cloud (VPC). We also created a security group to determine who can access our instances, and where our instances can send messages to. We defined our security group's VPC to be the default VPC, and we allowed the input and output traffic to and from our instances to be anywhere.

Figure 2: Creation of our instances of type t2.large

The Application Load Balancer (ALB) is also created using the Terraform "ressource" block. We use the id of our security group created, and we define the subnets that our ALB can be attached to to all the subnets in our VPC, that we fetch with a "data" block.

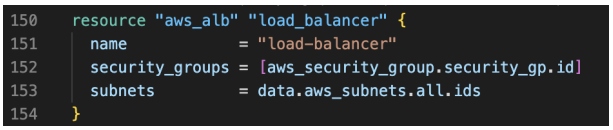

Figure 3: Creation of our ALB.large

We then create two different target groups, one for our T4 instances and one for our M4 instances, with a "ressource" block. This is also where we define our port 80. As such, when the load balancer receives a request, it will forward the request to the port 80 of a certain instance. (Within that instance, that request can then be forwarded to the port 5000, where our flask application is running).

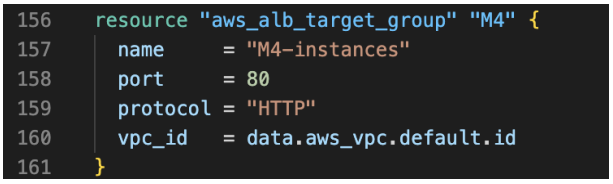

Figure 4: Creation of target group of m4 instances

Once our ALB and our target groups are created, we create our ALB listener ressource. This listener is linked to the ALB ressource we created before. By default, this listener will forward the requests received by the ALB to port 80 of the m4 instances. However, we then create two ressources to be the rule for m4 instances and the rule for t2 instances. The m4 rule will tell the ALB to forward requests received on the route finishing with /route1 to the m4 target group, and the t2 rule will tell ALB to forward requests received on the route finishing with /route2 to the t2 target group.

Next, we create the target group attachments, which allows us to attach each of our five M4 and our four T2 instances to their respective target group using the ARN, a name that uniquely identifies each of our target groups.

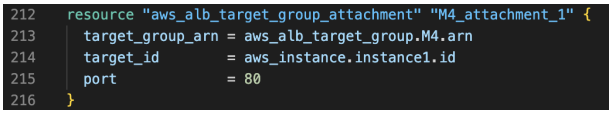

Figure 5: Attachment of instance1 to the M4 target group

2.2 Script.sh file

To automate the setup of our infrastructure and the sending of our requests, we use a shell file called script.sh. This file gets the AWS setup settings, such as the access key id, the secret access key and the session token, then deploys the infrastructure on AWS by running our Terraform file and saves the ALB DNS name using a Terraform output command. Our python script that is in charge of sending the requests on two threads is in a Docker image that is published on Dockerhub called wayr/ec2_requests. We can run a container based on this image and pass in the ALB DNS name, which allows the container to send the requests to our Application load balancer, which will then forward them to the respective target group.

2.3 Global docker image

The setup of our instances and load balancer was automated with a global docker image. This docker container allows us to install Terraform on the container before executing our script.sh. We do this to avoid the risk of the user not having Terraform installed on his computer, which would prevent the execution of our script.sh file.

3 Benchmark results

Benchmark results

4 Instructions to run the code

Instructions to run the code

To run the code, run the Docker container with the environment variables as... with the following command: ...