Cloud Computing Assignment 2

Elastic Web Service Auto-Scaling

Introduction

This report documents the steps followed to implement an elastic web service with auto-scaling using Amazon EC2. I first launched a VM using t2.small as instance type and AMI Linux 2 AMI as image. I enabled the ports 80 and 22. After I connected to my VM, I ran the following commands to do updates, and install PHP and Apache:

sudo yum update -y

sudo yum install httpd

sudo yum install php

I then ran the command **sudo vi /etc/rc.local** and added the line **sudo /usr/sbin/httpd -k start** inside the file. Afterwards, I downloaded the index.php file given on Moodle and moved it into my VM. I used the command **sudo /usr/sbin/httpd -k start** to start the Apache server. Lastly, using the public DNS of my VM, I checked if my instance was running the php file.

After making sure that it was working, I created an AMI from my instance, and I created two instances from the AMI that I created. When I set the number of instances as 2, AWS gave me the option to create an Auto Scaling Group. I set the minimum and desired number of instances as 1, maximum number of instances as 3. In the configuration of Auto Scaling Group, I created an application load balancer and attached it to my Auto Scaling Group and instances. I then updated the ping path for the health check as /index.php. I tested my load balancer by accessing its URL from my browser and saw that it was working. Afterwards, I created scaling policy to add or remove one instance based on average CPU utilization using Amazon CloudWatch. I used default values for parameters, target value as 50 and instance warmup as 300 seconds.

The preparations were finally done. I then installed httpref using Ubuntu and started to generate load on instances. I used two different commands in total, which are as follows:

httperf --server ec2-52-59-56-175.eu-central-1.compute.amazonaws.com --uri=/index.php --wsess=600,5,2 --rate 1 --timeout 5

httperf --server ec2-52-59-56-175.eu-central-1.compute.amazonaws.com --uri=/index.php --wsess=1000,8,2 --rate 1 --timeout 5

The first one was the same command given in the assignment file. I then increased the number of total sessions and number of calls for each session for the second command.

Monitoring the Results

I monitored the CPU utilization and the number of active instances during the load testing periods. I recorded my screen while doing that, the link to the video is given below:

https://youtu.be/IkzTIrkGDnQ

In Figure 1, the graph showing the CPU utilization during several load testing periods is given. When I used the first command, the utilization rate was barely able to reach 50%. However, second command, which created 1000 sessions each having 8 calls resulted in utilization rates almost 100%.

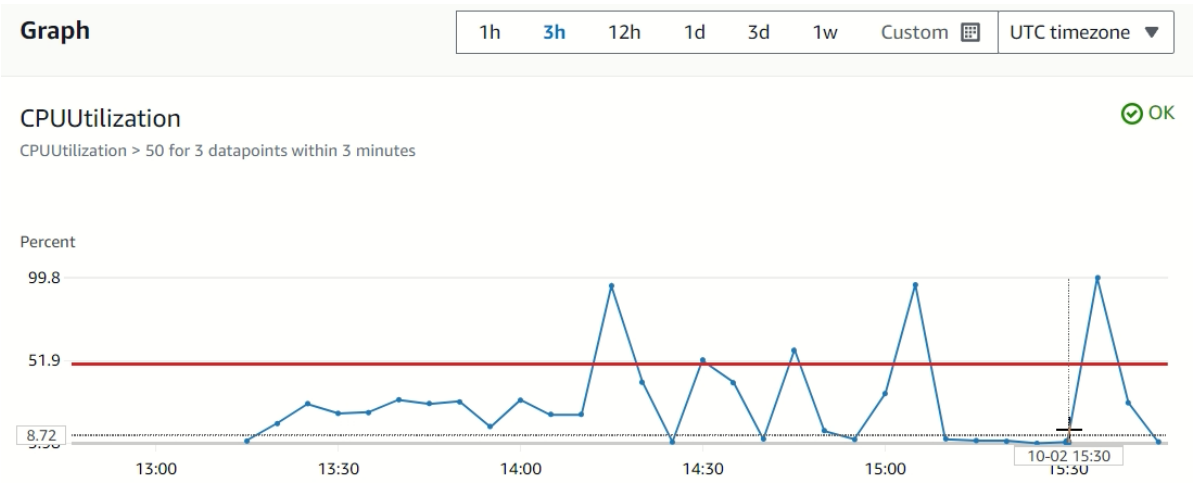

Figure 1. CPU Utilization Over Time

First command took 10 minutes while the latter took 16 minutes due to the differences in the number of sessions and the number of calls per session. In the first command, there were 600 sessions, each containing 5 calls, spaced 2 seconds apart, with 1 session generated every second. This resulted in a total runtime of 600 seconds. On the other hand, the second command

involved 1000 sessions, each containing 8 calls, also spaced 2 seconds apart, but with 1 session per second. The increased number of sessions and calls per session resulted in a total runtime of 1000 seconds. The longer time in the second command is directly related to the higher workload introduced by the additional sessions and calls.

Figure 2 below shows the total number of instances during the test loads. As it can be seen, it started with one instance since that is the initial desired capacity. Then, we see that it went back and forth between 2 and 3 instances. These frequent changes indicate dynamic scaling based on real-time demand, with the Auto Scaling group adding or removing instances based on load spikes and dips. The stronger load however makes the system to stay in three instances for a longer time than the weaker one does. In one of the tests, we see that the total number of instances remained at 3 during the whole test. However, even though I used the same command, AWS did only launch one instance during my last two tests. In overall, the graph shows the elastic nature of the Auto Scaling group, responding to changes in demand. When the load increases, the Auto Scaling group adds instances, and when demand decreases, it removes instances.

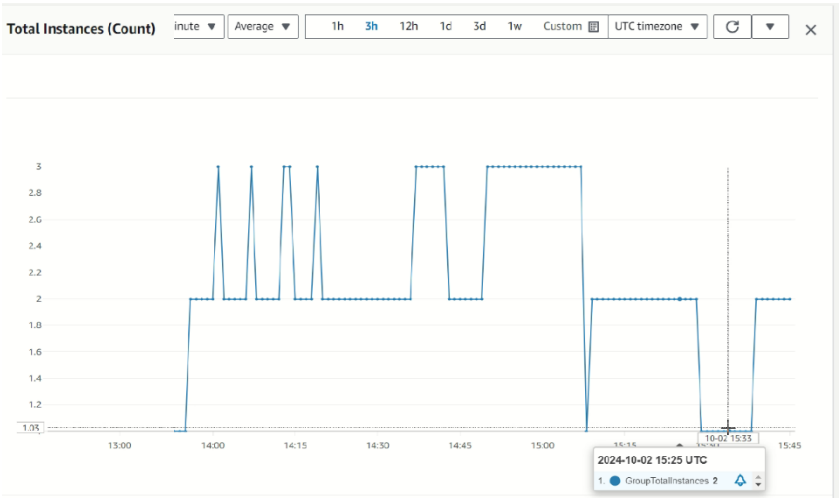

Figure 2. Number of Total Instances Over Time

Analysis

The web service scaled effectively in response to the httperf load. As the load increased, the number of active instances grew from 1 to 2 or 3 based on the demand, as indicated by the rising CPU utilization. When the load subsided, the system successfully terminated unnecessary instances, reducing the number of active instances. Specifically, we see that the number of

instances increases from 1 to 2 or 3 in several cycles between 14:30 and 14:45. This scaling out behavior indicates that the service is experiencing increased demand, causing the Auto Scaling group to provision additional resources to handle the higher load.

Similarly, when the load decreases, the Auto Scaling group reduces the number of running instances to optimize resource usage. This is reflected by the sharp drops in instance count, where the system scales back down to 1 instance, such as around 15:00 and 15:30. We can say that the system effectively reduces unnecessary costs by automatically terminating instances when they are no longer needed.

My tests also show some fluctuation in the number of instances between 1 and 3 at between 14:00 and 14:20, indicating that the load experienced by the web service may vary rapidly. These fluctuations suggest that the system is reactive to real-time load changes, provisioning or terminating instances in response to changing demands.

The period between load increase and the system's response by adding new instances was shorter than I expected, it usually took not more than 5 minutes. However, the scaling mechanism doesn't immediately scale up to 3 instances every time, despite running the same load test. I tried to find the reason for this, and found out that this variation in the number of instances could be attributed to the scaling thresholds not always being exceeded, or a variation in load distribution between instances.

Reactivity of my web service to the load tests should also be discussed. When the load increased, the load balancer metrics that I defined crossed the predefined threshold. Then, the auto scaling policy that I set triggered a scaling-out action to add instances. New instances were firstly provisioned, then initialized, which took 4 minutes in average. Finally, once the instances were initialized and marked healthy, they were added to the load balancer to distribute the incoming traffic. A very similar process happens when the load decreases. This time, when CPU utilization drops below the threshold (50% in my case), the policy triggers a scale-in action, and the system gradually terminates the unnecessary instances.

In overall, the Auto Scaling mechanism is highly elastic, that is able to adapt to traffic changes within a relatively short time. However, scaling in and out isn't instantaneous, and the delay could affect applications requiring immediate response to load changes.

Many web applications that experience variable traffic can benefit significantly from Auto Scaling for cloud resources. Below are several concrete examples of web applications that would particularly benefit from this mechanism:

- E-commerce websites usually experience traffic spikes during seasonal sales, special days like black friday and other promotional events. Auto scaling ensures that the platform can handle sudden surges in traffic by automatically adding resources when needed. When the traffic returns to normal after the event, the system scales down, reducing operational costs. As a concrete example, an online retailer like Amazon could leverage Auto scaling to handle peak shopping periods efficiently, ensuring customer satisfaction with minimal downtime or slow performance.
- Streaming platforms like Netflix or YouTube can experience fluctuating demand, during evenings or weekends, or specifically when a movie or a video that is most of the people are hyped to watch is released. If Auto scaling is used, the service can scale up during high-demand periods and scale down during off-peak hours to optimize resource usage.
- Social media platforms experience variable traffic patterns, with peaks during special events, user-generated content going viral, or in different time zones. The platforms can again use Auto scaling to dynamically adjust the number of servers based on user activity, maintaining a smooth user experience without over-provisioning during low-traffic periods. When a viral incident occurs, such as assasination attempt on Trump, platforms like X can benefit from scaling up.

Reflection

Everything I did after creating the instance was completely new to me; so I learned about the elastic nature of cloud resources and how Auto scaling dynamically adjusts the number of running instances based on load fluctuations. Moreover, the hands-on experience of using tools like httperf to simulate load and analyze the performance of my web service was salso omething new. I hadn't worked with load-testing tools in this context before, so learning to run these tests and interpret the results was enlightening.

I found the assignment challenging in general. I was confused about the chronological order of the steps I should have followed; therefore, I had to create the Auto scaling group for at least four times. The problem was not to create it, but I was not able to monitor the effects of the loads somehow. Finally, thanks to one of my friends, I was able to see the changing CPUUtilization based on the load tests.