DevOps – Assignment 2

David Kavanagh - 20057155

Waterford Institute of Technology
Applied Computing Year 3

Development Operations
Assignment 2
David Kavanagh – 20057155

Introduction:

The purpose of this report is to document the various steps involved in deploying a web application to a server placed in a personally defined virtual network. Using Amazon's Virtual Private Cloud (VPC) service, resources such as load balancers and auto scaling groups will be configured and used inside the VPC alongside the web application. These resources will allow for automated management of the application. Bash scripts will be used to demonstrate how these automated management features work. Once completed, there will be a structure in place that will give the application the ability expand and contract in reaction to network demands without the need for an administrator to oversee the operation.

Steps 1 - 7:

The first step completed was to configure and deploy a Virtual Private Cloud (VPC). By setting up the Amazon VPC, any AWS resources could be launched into a virtual network that was personally defined. The VPC was configured with two public subnets that were used for my application server, and two private subnets for the database servers. Following on from that, an Amazon EC2 Instance that would be the virtual application server was configured and launched. This virtual server was set up to run the web application that was about to be deployed.

Fig 1: VPC Setup

Fig 2: EC2 Instance Configuration

The next step to full deployment of the web app was to create a database server. For this, the Amazon Relational Database Service (RDS) was used. Amazon RDS automatically maintains a synchronous replica of the database server in a different availability zone. The two servers are kept in sync, allowing for redundancy in the case of an instance or availability zone failure. The server was configured with MYSQL as the database engine.

Once the configuration for the VPC and servers was complete, the next task was to deploy a web application on the EC2 instance. To do this, an SSH connection to the instance was opened. Firstly, Apache web server software was installed, followed by the latest version of Drupal. Drupal is an open source content management platform. It allows users to easily organise, manage and publish content online. For the purpose of this assignment, it would just be used to demonstrate auto scaling and load balancing within the VPC.

Fig 3: Drupal Web App Home Page

An Amazon Machine Image (AMI) was created from the instance configured with Apache and Drupal. This AMI would then be used to build identical instances when auto scaling policy is triggered. Following on from that, the load balancer and auto scaling group were configured. Auto scaling allows a user to scale the EC2 capacity up or down automatically according to a set of pre defined conditions. It ensures that there is always the desired amount of instances running in order to maintain performance during times of high traffic and decrease capacity during times of low traffic volume.

Fig 4: Load Balancer Setup

Fig 5: Auto Scaling Group Setup

For the purpose of this assignment, the metric used for executing the auto scaling policy was CPU utilisation. CPU usage is central to the operating of any server. Therefor the monitoring of CPU usage is of vitally important in order to stay on top of the networks performance. By monitoring CPU usage on the server, it is possible to spot server overloads before they can lead to poor performance or downtimes.

The auto scaling policy was set to execute when CPU utilisation was greater than 60% for 300 consecutive seconds. In order to trigger the policy to create a new instance, a bash script was copied to a running instance through an ssh connection. Contained in the script was an infinite loop. Executing the script caused the CPU usage to go above the specified threshold, in turn causing the scaling policy to execute. The result can be seen in figure 6 below.

Fig 6: Auto Scaling Policy Execution

Fig 7: Bash Script to Trigger Auto Scaling Policy

Step 8 – Generation of Test Traffic to the Load Balancer:

Curl (cURL):

Curl is an open source command line tool and library designed to transfer data using URL syntax to and from web servers using any of its supported protocols. It has been designed to run without any user interaction or any form of interactivity. Curl offers many useful features such as ftp transfer, proxy support and user authentication. It is used widely throughout modern software. It has been estimated that curl is used in software applications affecting more than one billion users. For the purpose of this assignment, curl was used in a script that would generate 2000 HTTP GET requests to the load balancer belonging to the web servers. The script can be seen below in figure 8.

```
testLoadBalancer.sh
#!/bin/bash

url="http://dk-loadbalancer-355490230.eu-west-1.elb.amazonaws.com"

i=0

while [ $i -le 2000 ]

do
  curl "$url/$i"
  i=$(( $i + 1 ))
done

```

Fig 8: Bash Script for Testing Load Balancer

Step 9 – Show Load Balancer distributing across more than one server:

In order to demonstrate the load balancer distributing across more than one server, the bash script seen in figure 7 was used to trigger the auto scaling Policy to execute. The execution allowed for three instances to be used in this demonstration. An ssh connection was opened to each of the instances. Once connected, the idea was to use the script shown in figure 8 to generate traffic to the load balancer. In order to view the traffic coming to each web server, the access_log file was displayed on each terminal window. The tail command was used to stream the output of the end of the file. The output of the tail command was piped into the grep command. The grep command searches files for a given string of text. Using the –v option inverts the command so it will only select lines without the given text. The purpose of this was to exclude any of the health checker pings that are sent to each instance every 30 seconds, thus making the output easier to read.

```
[ec2-user@ip-10-0-1-197 ~]$ sudo tail -f /var/log/httpd/access_log | grep -v HealthChecker
```

Fig 9: Command to display the end of the access log file

In order to clearly see how the load balancer operates, each of the 2000 HTTP GET requests sent by the bash script was individually numbered. The load balancer can be seen in figure 10, almost evenly distributing the requests amongst the three web servers. For example, looking at the bottom terminal window shows the last request sent, i.e. GET /2000. By observing each numbered request above it can be seen that the numbers almost uniformly decrease by 3. This shows that each third request was sent to that particular server.

Fig 10: Load Balancer in operation

Step 10 - Use of your own script to monitor some activity on your server:

Monitoring:

To complete step 10 of this report, the boto API was used in a python script to monitor a chosen instance, as well as create a new CloudWatch alarm based on a different metric. In order to perform monitoring tasks, the instance id was used to connect directly to an already created instance. A user is presented with the option of five different metrics that can be monitored. A CloudWatch object is instantiated in the same region as the instance. The get_metric_statistics method is then called on the object, passing in the metric chosen by the user as one of the parameters. The result of this is to return the statistics for the metric, which are printed to the screen. This allows the user to keep up to date with the current activity of their instance.

```
#
# Allows the user to monitor a list of metrics. A cloudwatch object is initialised
# The cloudwatch object is used to get the statistics for a metric chosen by the user
# by passing it into the get_metric_statistics method.
#
def monitor_instance(instance_id):
    metrics = {'0' : 'Exit', '1' : 'CPUUtilization', '2' : 'NetworkIn', '3' : 'NetworkOut', '4' : 'DiskReadOps', '5' : 'DiskWriteOps'}
    descriptions = {'CPUUtilization' : 'The percentage of allocated EC2 compute units that are currently in use on the instance',
                    'NetworkIn':'This metric identifies the volume of incoming network traffic to an application on a single instance',
                    'NetworkOut':'This metric identifies the volume of outgoing network traffic to an application on a single instance',
                    'DiskReadOps':'Completed read operations from all ephemeral disks available to the instance in a specified period of time',
                    'DiskWriteOps':'Completed write operations to all ephemeral disks available to the instance in a specified period of time'}

    options = metrics.keys()
    options.sort()
    print '\nChoose a metric to monitor:'
    for entry in options:
        print entry, metrics[entry]
    input = raw_input('Input: ')

    if input in options:
        metric = metrics[input]

        cloudwatch = boto.ec2.cloudwatch.connect_to_region('eu-west-1')
        stats = cloudwatch.get_metric_statistics(300,
            datetime.datetime.utcnow() - datetime.timedelta(seconds=600),
            datetime.datetime.utcnow(),
            metric,
            'AWS/EC2',
            'Average',
            dimensions={'InstanceId' : instance_id})

        if stats:
            statistics = stats[0]
            print '\n', metric + ' ::'
            print descriptions[metric]
            print '--------------------------'
            for x,y in statistics.iteritems():
                print x +' : ', y
            print '--------------------------'
    else:
        print '\nInvalid option\n'
```

Fig 11: Method for monitoring the instance

```
Choose a metric to monitor:
0 Exit
1 CPUUtilization
2 NetworkIn
3 NetworkOut
4 DiskReadOps
5 DiskWriteOps
Input: 1

CPUUtilization ::
The percentage of allocated EC2 compute units that are currently in use on the i
nstance
--------------------------
Timestamp :  2014-11-24 19:55:00
Average :  0.634
Unit :  Percent
--------------------------
0 Exit
1 Create a Cloudwatch alarm
2 Monitor Instance
3 Trigger Alarm
Input:
```

Fig 12: Result of monitoring CPUUtilization

Creating an Auto Scaling Group:

Another option made available to the user by the script is to create a new CloudWatch alarm and to trigger that alarm so that its auto scaling policy will create a new instance using the same AMI as in steps 1 - 7. The script first creates a launch configuration using the same image id as before. An auto scaling connection is open and used to submit the LC to AWS. The new auto scaling group is created next. The launch configuration is used in the creation of the group by passing it in as one of the parameters.

```
# create an autoscaling group
autoscaling_group = AutoScalingGroup(name = 'DK-AutoScale-Boto',
    launch_config = launch_config,
    min_size = 1, max_size = 3,
    availability_zones = ['eu-west-1a', 'eu-west-1b'],
    vpc_zone_identifier = ['subnet-747ea503','subnet-bfcb34e6'],
    security_group_ids = ['sg-0edd5d6b'],
    connection = autoscaling_conn)
```

Fig 13: Creating an auto scaling group using boto API

Fig 14: Confirmation of AS group

Once the group is submitted to AWS, an auto scaling policy called 'scale-up' is created, stating that one new instance should be created in the event of the alarm being triggered. The policy is also submitted to AWS using the same auto scaling connection, however this time the get_all_policies method is called on the connection to retrieve the new policy. The reason for this is that extra properties need to be added to the policy by AWS in order for it to be configured correctly. Once the policy is retrieved back from AWS correctly configured, the Amazon Resource Name (ARN) of the policy is placed into a variable that will be used when creating the CloudWatch alarm. ARN's are used to uniquely identify individual Amazon resources.

Creating a CloudWatch Alarm:

To create an alarm using boto, a topic must first be created. Amazon uses its Simple Notification Service (sns) to alert users if a CloudWatch alarm has been triggered on one of their instances. Using an sns connection, a topic and a subscription are created. A topic is an "access point" for allowing recipients to dynamically subscribe for identical copies of the same notification. Multiple recipients can be grouped together to receive sns notifications consecutively. For the subscription, an accessible email address was used. This email will receive the notification.

```
sns = boto.sns.connect_to_region('eu-west-1')
sns.create_topic('DavidK_Network_Problem') # Create a topic
arn = 'arn:aws:sns:eu-west-1:808146113457:DavidK_Network_Problem' #Amazon Resource Name, uniquely identify AWS resources
sns.subscribe(arn, "email", "david_kav@hotmail.com") # subscribe my email to the topic
```

Fig 15: Creating the topic and subscription

A CloudWatch object is instantiated and used to get a list of metrics that the alarm will be based upon. In this case, the chosen metric was NetworkIn. This metric identifies the volume of incoming network traffic to an application on a single instance. The threshold was set at greater than or equal to 500,000 bytes for a period of 60 seconds. If that threshold is passed, the alarm will trigger. The ARN from the auto scaling group is used in the creation of the alarm to connect the group and its policy to the triggering of the alarm.

Fig 16: Alarm After Creation in the Script

Triggering Alarm:

The final option made available to the user is to trigger the alarm that has just been created. To do this a while loop was used to send repeated GET requests to the public URL of the instance using the curl command. The URL was constructed using the public DNS name of the instance. The curl request is sent to the instance 1000 times causing an increase in the incoming network activity. This increase triggers the CloudWatch alarm to take action, which in turn triggers the auto scaling policy to create a new instance, from the same AMI as the original.

```
#
# Uses a loop to send repeated curl requests to the url of the instance. This will cause the stated threshold
# to be passed and therefor trigger the cloudwatch alarm which creates a new instance
#
def trigger_alarm():
    global alarmed
    global instance

    dns = instance.public_dns_name
    url = '"http://' + dns + '"'
    if alarmed == True:

        i = 0
        cmd = 'curl ' + url
        print '\nRunning script to trigger alarm\n'
        while i < 1000:
            commands.getstatusoutput(cmd)
            i = i + 1

    else:
        print '\nNo alarm has been set'
        print 'Set a CloudWatch alarm first\n'
```

Fig 17: Method to Trigger CloudWatch alarm

Conclusion:

Amazon Web Services provides a highly reliable, scalable, low-cost infrastructure platform in the cloud that powers hundreds of thousands of businesses in 190 countries around the world. With data center locations in the U.S., Europe, Brazil, Singapore, Japan, and Australia, customers across all industries are taking advantage of the benefits allowed by the service. These benefits include a low cost pay-as-you-go service that are open, flexible and secure. AWS provides a global cloud infrastructure that allows users to quickly innovate, experiment and iterate. So new applications can be instantly deployed and scaled up and down depending on demand.

In order to successfully run a server or application from the cloud that involves changing network environments, the ability to monitor and adjust to changes in such environments is paramount. The configuration of resources such as virtual private networks, load balancers and auto scaling groups allows a user to configure their own network environment as they see fit. It also allows them to be assured that any changes in the network, such as large amounts of traffic, can be navigated automatically by the relevant resources without is causing any performance issues to the network in question. The cost saving and comparable ease of use means that services such as AWS are in many cases taking the place of expensive hardware, which in turn leads to more online services and innovations.