Alex Cohen, Deepak Kowshika and Mithun Shriyan

NSSA 605

**Title:** Provisioning an Auto Scaling Infrastructure on Amazon Web Services using Ansible

1. **Abstract:**

Amazon Web services or AWS is very popular cloud infrastructure as a service (IaaS) provider. One feature of AWS is Auto Scaling Groups. These are groups of EC2 Linux virtual machine instances, built from an AMI image, which is essentially a clone of a prebuilt Linux instance. The main feature of Auto Scaling Groups is their ability to deploy or terminate new instances automatically to facilitate horizontal scaling in response traffic spikes. Ansible is a configuration management software platform that can be used for many aspects of server provisioning. In particular Ansible has several modules that can automate the process of deploying new architectures on AWS. For this project we will be using Ansible to automate the process of creating an auto scaling group and the other necessary architectures needed to create this auto scaling group.

1. **Introduction:**

Amazon Web Services (AWS) offers variety of cloud computing services. The cloud computing service can be described as an on-demand delivery of IT application and resources using the internet with a pay-as-you-go pricing for infrastructure as a service (IaaS). This service provides an easy way for individuals to access databases, servers, storage and other applications over the internet. AWS owns and maintains the hardware required for these services. AWS offers various services such as global compute, storage, database, analytics, deployment and application services. These services help reduce IT costs, scale applications and help organizations move faster. AWS services are trusted by a wide range of enterprises and start-ups to help power variety of workloads such as IoT, mobile and web applications, game development, data warehousing and processing, storage and archive (Amazon Web Services, 2016).

The AWS Auto Scaling Groups help in maintaining the availability of the applications and infrastructure by allowing for more dynamic clustering with the ability to scale capacity up or down automatically. ASG’s are groups built from EC2 instance image clones (called AMI’s) that have the ability to deploy or terminate new instances automatically to facilitate horizontal scaling in response traffic spikes allowing an infrastructure to maintain performance under high traffic and decrease capacity during hiatus to reduce costs. The AWS auto scaling is well suited for applications that have a stable demand or an application that has hourly, daily or weekly variability of usage that experience regular traffic spikes. It simplifies the web scale cloud computing making life easier for developers and provides a simple web service interface that allows us to configure capacity with minimal friction, providing developers with the tools to build more resilient infrastructures (Amazon Web Services, 2016).

Ansible is an open source automation software that seamlessly unites the workflow orchestration with provisioning, configuration management and deployment of application in one simple platform. It can also run tasks in sequence and create a chain of events that should happen on several different servers or devices, providing continuous integration. It is an extremely simple, easy to learn and scalable “engine that automates cloud provisioning, configuration management, application deployment, intra-service orchestration, and many other IT needs.” (Ansible, 2016).

We are going to utilize specific Ansible tasks designed for AWS within a playbook that can create all the necessary architectures for an amazon web services auto scaling group automatically.

1. **Purpose, Problems and Potential benefits:**

The goal of automation is to reduce the repetition of tasks whenever possible. Ansible provides this in many regards by providing a scalable automation solution that seamlessly integrates into many of the more common server architecture platforms available. One of these platforms is Amazon Web Services for which specific Ansible modules have been created that automate the process of creating many of the architectures available within the sphere of Amazon Web services (Martin 2014).

The process of creating an auto scaling group is a repetitive and multistep process that involves creating: a launch configuration, an auto scaling group, auto scaling metric alarms, auto scaling policies based on those alarms and then attaching all of the instances in that auto scaling group to an elastic load balancer made for the instances contained in that auto scaling group. All of this makes ASG architecture an ideal candidate for automation through Ansible. The concept of this project is that an Ansible playbook would be able to accomplish all of those tasks at once only needing an AMI image for input, thus making the creation of an auto scaling group from an AMI image a one step process.

1. **Literature Review:**

One of the most important concepts in cloud computing is scalability. To support their websites and service, Amazon, Microsoft, and Google have all built a ton of computing infrastructure. The data centers these companies use are bigger and more efficient than most data center for other companies that do not focus only on server hosting. These giants now rent some of this capacity to developers and companies anywhere in the world. A developer or a company can swipe a credit card and get access to fundamentally unlimited computing power. It means that software can run at much larger scales, for much less expense, at a higher rate of availability and performance, without anyone having to worry about maintaining a data center (Amazon Web Services, 2016).

The basic game plan for all cloud-computing vendors is to make these resources available to both independent software developers and big companies. For cloud based infrastructure as a service (IaaS) providers go, Amazon web services are an industry standard and offer a wide variety of highly specialized and well-engineered IT infrastructure services.

One of these are auto scaling groups, which can be used to detect impaired EC2 instances and unhealthy applications even if there are thousands of Amazon EC2 instances running. Using this feature, we can easily detect and replace these instances without human intervention. This would ensure that an application will receive the computing capacity that it needs at any point in time, saving system administrators time and effort. Auto scaling also lets you adjust according to the demand curve for the applications. This reduces the need to manually provision the EC2 capacity in advance (Amazon Web Services, 2016).

Ansible is simple, easy to use, and highly scalable. It has the ability to handle many complex operations such rolling continuous integration and multi-tiered server provisioning (Ansible, 2016). Specifically Ansible has many easy to use modules that integrate with AWS to provide the automation of infrastructure creation for many specific architectures including: Auto Scale Groups (ec2\_asg), Auto Scaling Policies (ec2\_scaling\_policy), CloudWatch Alarms (ec2\_metric\_alarams), Elastic Load Balancers (ec2\_elb\_lb) and Launch Configurations (ec2\_lc) (Martin, 2014).

1. **Methodology:**

1. Create an amazon EC2 instance to serve as our Ansible machine and install Ansible 2.0.0 on it (as this is the version required for some of the Ansible tasks we will use in this project).
2. Create an amiKeys.yml file containing our AMI access key and AMI secret key, which we will encrypt using Ansible vault.
3. Create a playbook utilizing the following Ansible tasks to create our infrastructure.
4. The ec2\_elb\_lb task will create and launch an elastic load balancer (ELB).
5. The ec2\_lc task will create a launch configuration that will be used to create our auto scaling group. For this I will use the AMI image I have built of an Ubuntu Linux machines running an apache web server that connects to a MySQL database on an RDS to form a distributed LAMP architecture.
6. The ec2\_asg task will create my auto scaling group from the launch configuration created in step b) and attach it to the load balancer created in step a).
7. The ec2\_metric\_alarm task will create two automatic cloud watch alarms for the auto scaling group with.

* One alarm called CPUup that will activate if the CPU utilization of my auto scaling group is over 20%.
* And a second alarm called CPUdown that will activate if the CPU utilization of my auto scaling group is under 10%.

1. The ec2\_scaling\_policy task will create two auto scaling policies using the metric alarms created in step d).

* One of these policies will automatically increase my ASG instance amount by 1 if CPUup activates.
* The other will automatically decrease my ASG instance amount by 1 if CPUdown activates.

1. Test the architecture by visiting the DNS address of the load balancer created.
2. Test the auto scaling policies by inducing high and low CPU utilization with the tool “stress”
3. Test the load balancing capabilities by editing the HTML index files of the servers in my auto scaling groups to be slightly different and attempt to visit the DNS address of my load balancer several times in order to see if it is alternating between my servers properly.
4. **Results:**

**NOTE: most of the values have been changed so that exact info from our AWS account is not revealed**

1. First we created an Ubuntu server as an amazon EC2 instance to serve as our Ansible machine and we installed Ansible 2.0.0 on it (as this is the version required for some of the Ansible tasks we will use in this project).

How to install Ansible 2.0.0:

sudo apt-get install software-properties-common

sudo apt-add-repository ppa:ansible/Ansible

sudo apt-get update

sudo apt-get install Ansible

1. Then we created an amiKeys.yml file containing our AMI access key and AMI secret key, which we encrypted using Ansible vault.

---

ami\_access: "JGVKLHYGLL"

ami\_secret: "iliuahsdflHDHFKJYY&/ID+"

We encrypted it with the command:

ansible-vault encrypt amiKeys.yml

This means that a password will be required whenever we use this file in another playbook and that using a playbook that takes variables from a vault file will require the following flag at the end of an ansible-playbook command:

--ask-vault-pass

1. Then we created a playbook to create our infrastructure called autoscale.yml

Our playbook begins like a regular playbook that is run on the local machine:

---

- hosts: localhost

connection: local

gather\_facts: false

1. Then we utilized a prompt to allow the user to enter in specific information to the playbook so that this playbook can easily be reused to create an automated multi-tiered architecture for different AMI images that we create in the future. The values entered will then be saved as variables that we can reference later on in the playbook.

* The value “image” will store the value of the ID for the AMI image that our auto scaling group is being built from.
* The value “architecture” will be used to name everything else we are creating.

vars\_prompt:

#user enters the name of the ami image to make an auto scaling group from

- name: image

prompt: "enter the AMI image id"

private: no

#user enters the name that will be used to name everything else in the architecture

- name: architecture

prompt: "enter the desired name for your infrastructure"

private: no

1. After this we set some variables that we will be reusing throughout our playbook, this includes:
   1. keypair: the name of the ssh key pair our instances will use.
   2. group: the ID of the security group our instances will use.
   3. regi: the name of the region much of our infrastructure will use.
   4. loadb: the name of the load balancer we will be creating.
   5. auto\_sc: the name of the auto scaling group we will be creating.
   6. lc: the name of the launch configuration we will be creating.

vars:

keypair: "asdfasdfaKeyPair"

group: "sg-asdassdee"

subnetID: "subnet-f0X2sfgaadsdb"

regi: "us-east-1"

loadb: "{{architecture}}-lb"

auto\_sc: "{{architecture}}\_asg"

lc: "{{architecture}}\_lc"

1. We also imported our AMI keys from the amiKeys.yml, this would let us use ami\_access or ami\_secret to reference our AMI key values without having to encrypt our autoscal.yml playbook

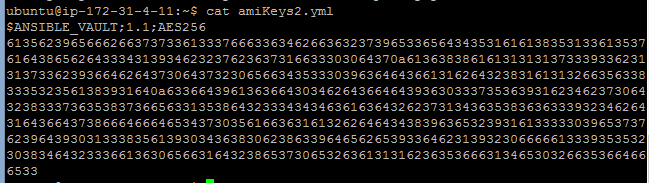
vars\_files:

- amiKeys.yml

By using this we are required to run our playbook with the extra flag --ask-vault-pass that requires the input of the password we set with our Ansible vault file. This is the full command that will be used to run our playbook:

ansible-playbook autoscale.yml --ask-vault-pass

After encryption our file becomes encoded gibberish:



1. Now we were ready to create our tasks under the following line:

tasks:

1. The first task we created was our ec2\_elb task that builds our elastic load balancer:

#load balancer

- ec2\_elb\_lb:

aws\_access\_key: '{{ ami\_access }}'

aws\_secret\_key: '{{ ami\_secret }}'

name: "{{ loadb }}"

region: "{{ regi }}"

zones:

- us-east-1a

- us-east-1b

- us-east-1c

- us-east-1e

state: present

listeners:

- protocol: http

load\_balancer\_port: 80

instance\_port: 80

1. The second task we created was our ec2\_lc task that creates a launch configuration that will build our auto scaling group.

#launch configuration

- ec2\_lc:

aws\_access\_key: '{{ ami\_access }}'

aws\_secret\_key: '{{ ami\_secret }}'

region: "{{regi}}"

name: '{{lc}}'

image\_id: "{{image}}"

key\_name: "{{ keypair }}"

security\_groups: '{{ group }}'

instance\_type: t2.micro

volumes:

- device\_name: /dev/sda1

volume\_size: 100

device\_type: io1

iops: 3000

delete\_on\_termination: true

1. The third task we created was our ec2\_asg task that builds our auto scaling group from the launch configuration created in the previous task and attaches it to the load balancer created in the task before that.

#auto scaling group

- ec2\_asg:

aws\_access\_key: '{{ ami\_access }}'

aws\_secret\_key: '{{ ami\_secret }}'

region: "{{ regi }}"

name: '{{ auto\_sc }}'

load\_balancers: '{{ loadb }}'

availability\_zones: 'us-east-1a'

launch\_config\_name: '{{ lc }}'

min\_size: 1

max\_size: 4

desired\_capacity: 2

vpc\_zone\_identifier: '{{ subnetID }}'

wait\_for\_instances: true

1. The fourth task we created was our ec2\_scaling\_policy task builds two auto scaling policies which can either add or remove an instance from our auto scaling group we created in the last task automatically. It also registers the output of the task so that our metric alarm can reference it.

#auto scaling policy

- ec2\_scaling\_policy:

aws\_access\_key: '{{ ami\_access }}'

aws\_secret\_key: '{{ ami\_secret }}'

region: '{{regi}}'

state: present

name: "{{item.pol\_name}}"

adjustment\_type: "ChangeInCapacity"

asg\_name: "{{auto\_sc}}"

scaling\_adjustment: "{{item.changes}}"

min\_adjustment\_step: 1

cooldown: 300

register: policies

with\_items:

- pol\_name: "cpuUP\_{{auto\_sc}}\_policy"

changes: +1

- pol\_name: "cpuDown\_{{auto\_sc}}\_policy"

changes: -1

To avoid creating two separate tasks we used the with\_items task option that will rerun the task twice replacing the inputs for our policy name and scaling adjustment. This allows the single task to create one policy that can add an instance to our ASG and one that can remove instances from our ASG, which makes our playbook cleaner.

1. The fifth and last task we created was our ec2\_metric\_alarm task that builds two automatic alarms for the auto scaling group and will activate one of our auto scaling policies by either adding or removing instances from our cluster. One alarm activates if the average CPU utilization of the ASG is over 10% and then terminates an instance in our ASG by calling one of our auto scaling policies. The other activates if it is under 20% and then adds an instance in our ASG by calling one of our auto scaling policies.

#cloud watch alarm for CPU Utilization

- ec2\_metric\_alarm:

aws\_access\_key: '{{ ami\_access }}'

aws\_secret\_key: '{{ ami\_secret }}'

state: present

region: "{{regi}}"

name: "{{item.names}}"

metric: "CPUUtilization"

namespace: "AWS/EC2"

statistic: Average

comparison: "{{item.compare}}"

threshold: "{{item.limits}}"

period: 60

evaluation\_periods: 1

unit: "Percent"

description: "{{item.desc}}"

dimensions: {'AutoScalingGroupName':'{{auto\_sc}}'}

alarm\_actions: "{{item.pol}}"

with\_items:

- names: "cpuUP\_{{auto\_sc}}"

compare: ">="

limits: "20.0"

desc: "This will alarm when the average cpu usage of the ASG is greater than 20% for 1 minute"

pol: "{{policies.results[0]['arn']}}"

- names: "cpuDown\_{{auto\_sc}}"

compare: "<="

limits: "10.0"

desc: "This will alarm when the average cpu usage of the ASG is less than 10% for 1 minute"

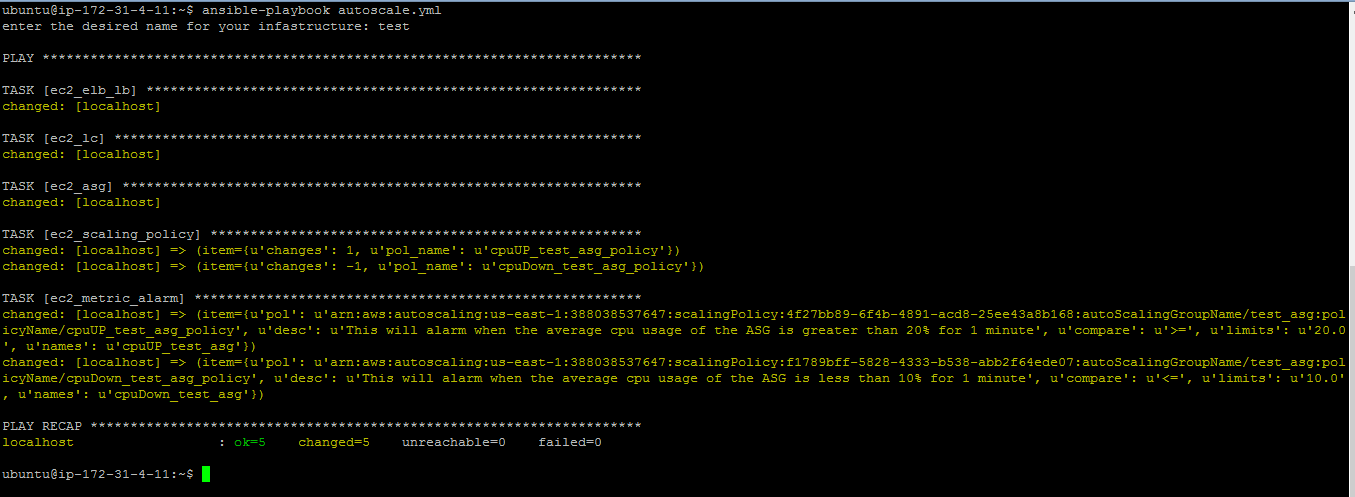
pol: "{{policies.results[1]['arn']}}"

Again here we used the with\_items option to have this task create two separate cloud watch alarms

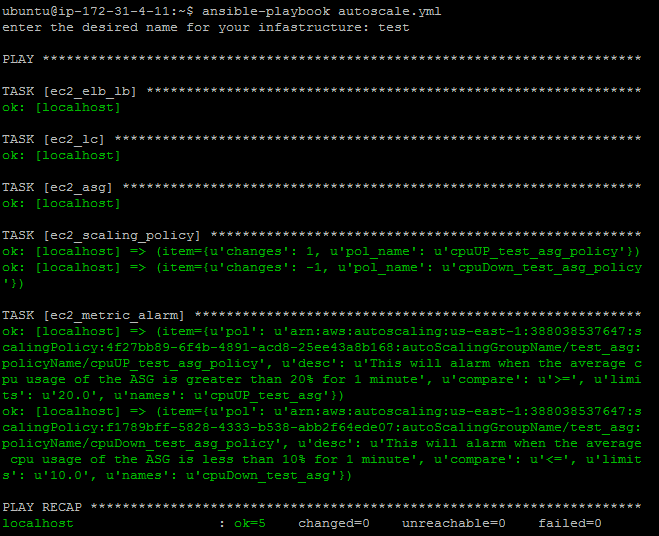
1. After the playbook has been created we tested it with an AMI image that we have taken of a Linux machine running an apache server that uses in python CGI to connect to an AWS RDS (relational database) that is running MySQL to create a distributed LAMP architecture. We named our architecture test:

Note that for the ease of testing, we set the image ID as a variable inside our playbook so that we would not have to paste in the exact AMI ID every time we wanted to test our playbook and we did not use an encrypted version of amiKeys.yml

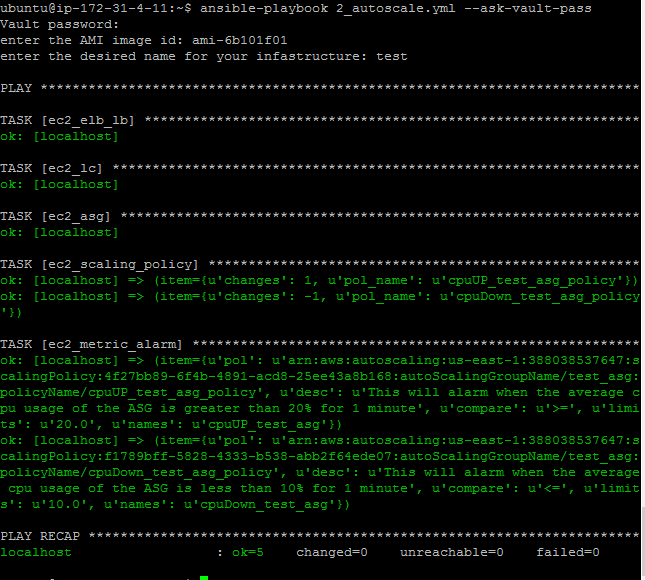
Ansible playbook runs successfully:



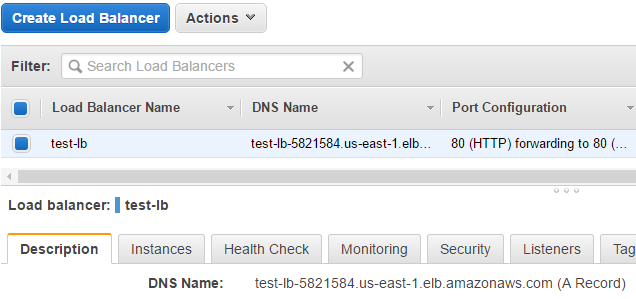
Play is idempotent:



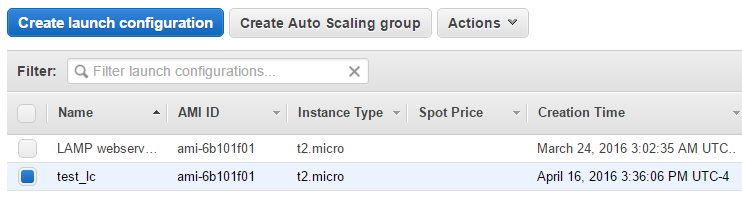
Another version of our playbook that takes the AMI image from user input and uses an amiKeys.yml file encrypted with Ansible vault works as well, we will be using this version of our playbook in our presentation:



Load balancer created:



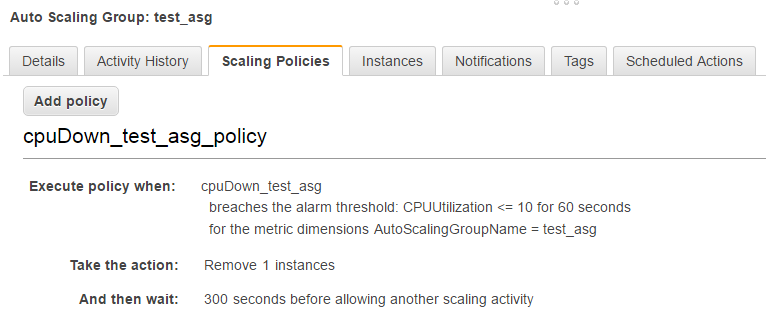
Launch configuration created:

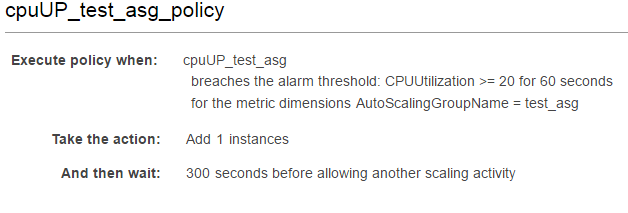


Auto scaling group created:

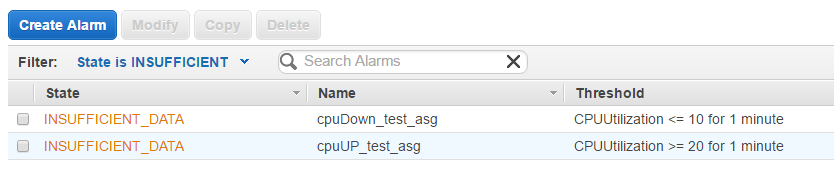


ASG policies created:

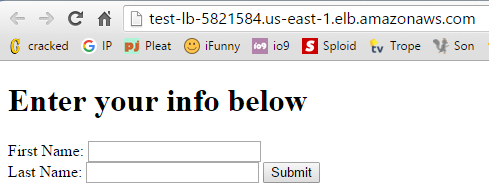




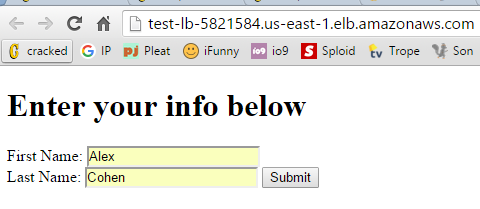
Cloud watch alarms created (they will have insufficient data for the first minute of operation after the instances go online):

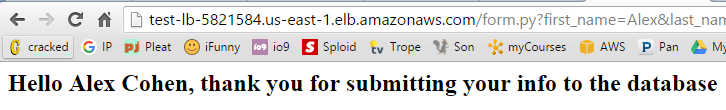


1. Then we tested the architecture by visiting the DNS address of the load balancer created.



It successfully brought us to the web page.





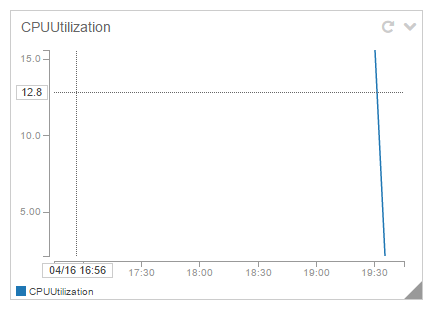
The web pages are active and can receive information.



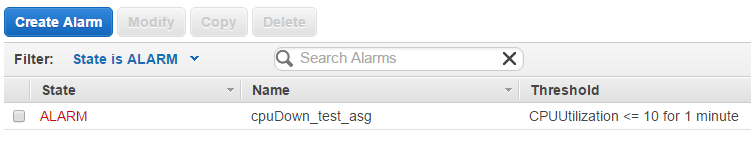
Using a python script we have made, that shows the contents of the table in our MySQL database on our RDS server we can see that an entry has been made for the information we entered and the LAMP architecture has been created successfully.

1. We then tested the auto scaling policies of our ASG, to see the average CPU utilization of our cluster we were able to go into cloud watch and create a dashboard graph of the average CPU utilization for our ASG.

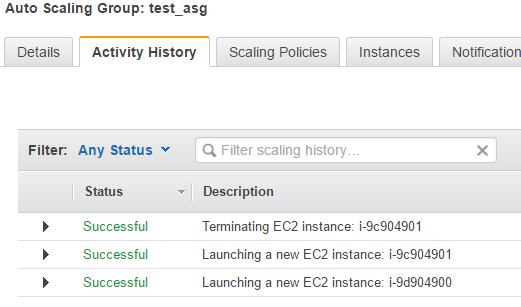
Immediately after the creation of this ASG the average CPU utilization dropped under 10%



This activated our alarm cpuDown\_test\_asg:



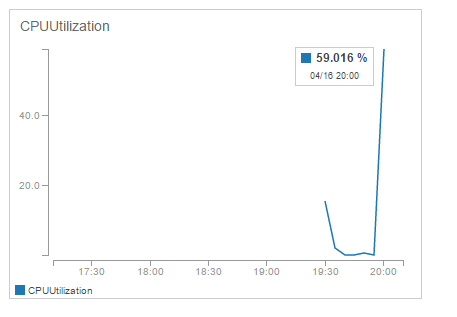
This then caused our auto scaling policy cpuDown\_test\_asg\_policy to activate and terminate an instance:



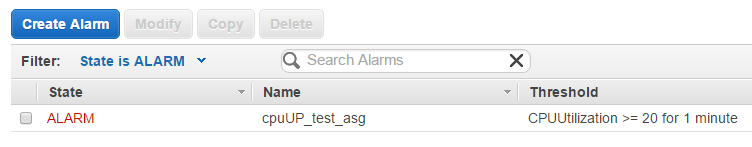
1. Next we used putty to ssh into the instance in the cluster and used the tool stress to stress the CPU utilization of the remaining instance:



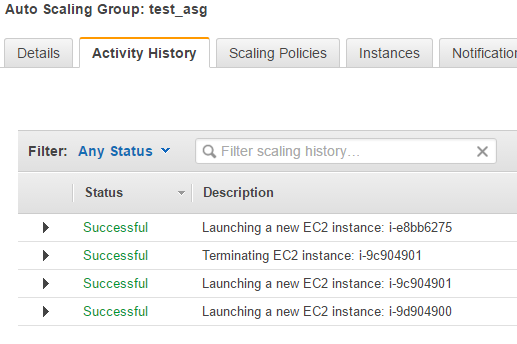
CPU utilization skyrockets:



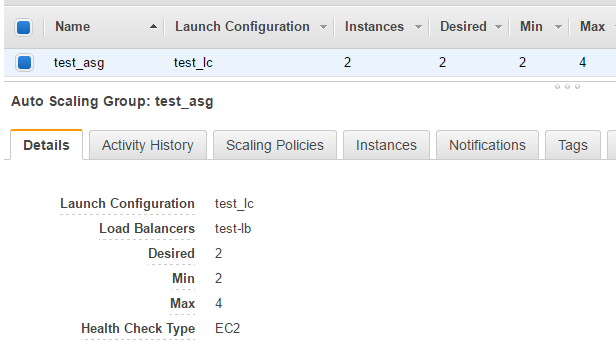
The alarm cpuUP\_test\_asg is activated:



A new instance in our auto scaling group is automatically created:



1. Last we wanted to test the load balancing capabilities by editing the HTML index files of the servers in the auto scaling groups to be slightly different. First to ensure we had at least 2 instances to load balance between we changed the minimum instances of our cluster from 1 to 2:



Our load balancer is set to the default of round robin so every other visit results in a new server:





Here we can see that it is successfully load balancing.

1. **Challenges:**

Writing each task for the autoscale.yml playbook was a constant struggle. Every documentation page was lacking some information on necessary values needed for each task. Lucky for us that most of the error messages we received when running new tasks were clear enough to determine what else we needed to add. The only issue we could not find a solution to just by searching on the internet was how to attach an auto scaling policy to a cloud watch alarm due to the cloud watch alarm requiring the ARN syntax of the auto scaling policy. For this we posted a question about this on stack exchange and received an answer to our issue a day later.

We were going to maybe attempt to use the Ansible tower platform as another method of running our playbook, but we decided against this due to time constraints and other projects we were working on.

1. **Future Work:**

Since we did not get a chance to work with Ansible Tower that platform is definitely something we would like to try out in the future. There are also many other configuration management tools we would like to work with in the future such as chef, salt and puppet all of which have AWS modules as well.

Regarding Amazon Web Services there are dozens of other services and infrastructure options available that we would like to work with at a future point in time. There is an EC2 container service that may prove to be very useful for creating architectures with Docker. There is definitely more work that can be done with RDS such as importing databases, working with other database languages, snapshotting and cloning, large scale provisioning and deployments with Ansible and other configuration management software’s. There is also the option of working with DynamoDB servers which is the equivalent of an RDS for NoSQL databases. This would be particularly useful if we were ever to create distributed NoSQL architectures such as a MEAN stack. There are also many options in the storage and content delivery section of AWS, such as S3, that we have dealt very little with before. The developer tools section of AWS looks interesting as well; CodeCommit, CodeDeploy and CodePipeling seem to be interesting options of configuration management through a process designed specifically for AWS, although we know very little about them at this point in time they definitely would be a fun thing to play around with.

1. **Conclusion:**

In conclusion using an Ansible playbook to automate the process of creating an auto scaling group infrastructure was very successful and eliminated much of the repetition involved in infrastructure creation. With our playbook we were automatically able to create an ELB, launch configuration, ASG, auto scaling policies and cloud watch alarms. All of this only requires the user’s desired name for the infrastructure and the AMI image that everything will be built off of.

**References:**

**Amazon Web Services. (2008, October). Amazon Web Services. Retrieved April 14, 2016, from** [**http://dclug.tux.org/200611/AmazonEC2.pdf**](http://dclug.tux.org/200611/AmazonEC2.pdf)

**Amazon Web Services. (2016). AWS | Auto Scaling. Retrieved April 14, 2016, from** [**https://aws.amazon.com/autoscaling/**](https://aws.amazon.com/autoscaling/)

**Ansible. (2016). How Ansible Works | Ansible.com. Retrieved April 14, 2016, from** [**https://www.ansible.com/how-ansible-works**](https://www.ansible.com/how-ansible-works)

**Huckman, R. S., Pisano, G. P., & Kind, L. (2008, February). Amazon Web Services. Retrieved April 14, 2016, from** [**http://www.hbs.edu/faculty/Pages/item.aspx?num=36511**](http://www.hbs.edu/faculty/Pages/item.aspx?num=36511)

**Martin, J. (2014, September 11). Provisioning an Autoscaling Infrastructure using Ansible. Retrieved March 29, 2016, from** [**https://www.ansible.com/blog/autoscaling-infrastructures**](https://www.ansible.com/blog/autoscaling-infrastructures)

**Ogenstad, P. (2016). What is Ansible? - Open Source IT Automation. Retrieved April 14, 2016, from** [**https://networklore.com/ansible/**](https://networklore.com/ansible/)

**Weinberger, M. (2015, November 07). The cloud wars explained: Why nobody can catch up with Amazon. Retrieved April 14, 2016, from** [**http://www.businessinsider.com/why-amazon-is-so-hard-to-topple-in-the-cloud-and-where-everybody-else-falls-2015-10**](http://www.businessinsider.com/why-amazon-is-so-hard-to-topple-in-the-cloud-and-where-everybody-else-falls-2015-10)