Blue Green Deployment

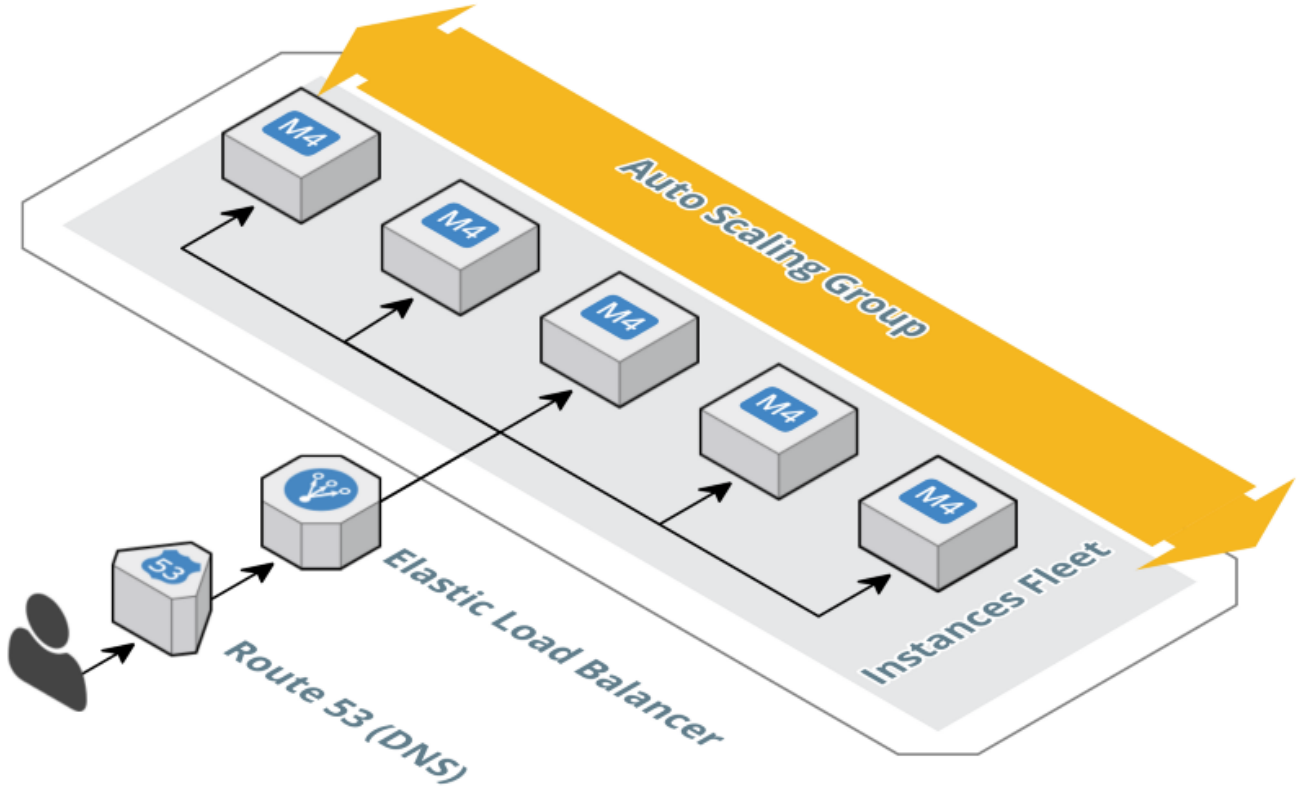
Using Amazon AWS and Python.

https://medium.com/practo-engineering/blue-green-deployment-on-amazon-aws-38b820518411

Introduction to components

In Blue green deployment approach we create replica of the current production infrastructure and route all the traffic to the replica. The current infrastructure is termed as **Blue** while the replica is **Green**. Of course we make all code updates in the green fleet. This ensure high availability during deployments.

This article is about implementing blue green deployment using Amazon AWS and Python. We will use Elastic Compute Cloud(EC2), Autoscaling Groups, Launch Configurations(LC), Elastic Load Balancer(ELB), Route 53, Python Boto and Python Fabric.



Our infrastructure contains a fleet of EC2 instances behind an ELB managed by an ASG. Route 53 acts as a gateway for users to interact with our service. All traffic to our domain reaches the ELB which distributes the load to the EC2 instances. ASG manages our cluster of instances. ASG can add an instance or remove it based on the configurable policies. ASG use Launch Configurations (LC) and Amazon Machine Images (AMI). Launch configurations define which virtual image(AMI) to use while scaling out.

Python Boto and Fabric are being used to automate the process of deployments. Boto is a library to access Amazon AWS via python scripts. An alternative to Boto is Amazon AWS CLI. Fabric is

another python library that comes handy to run deployment tasks on remote servers via SSH (concurrently!).

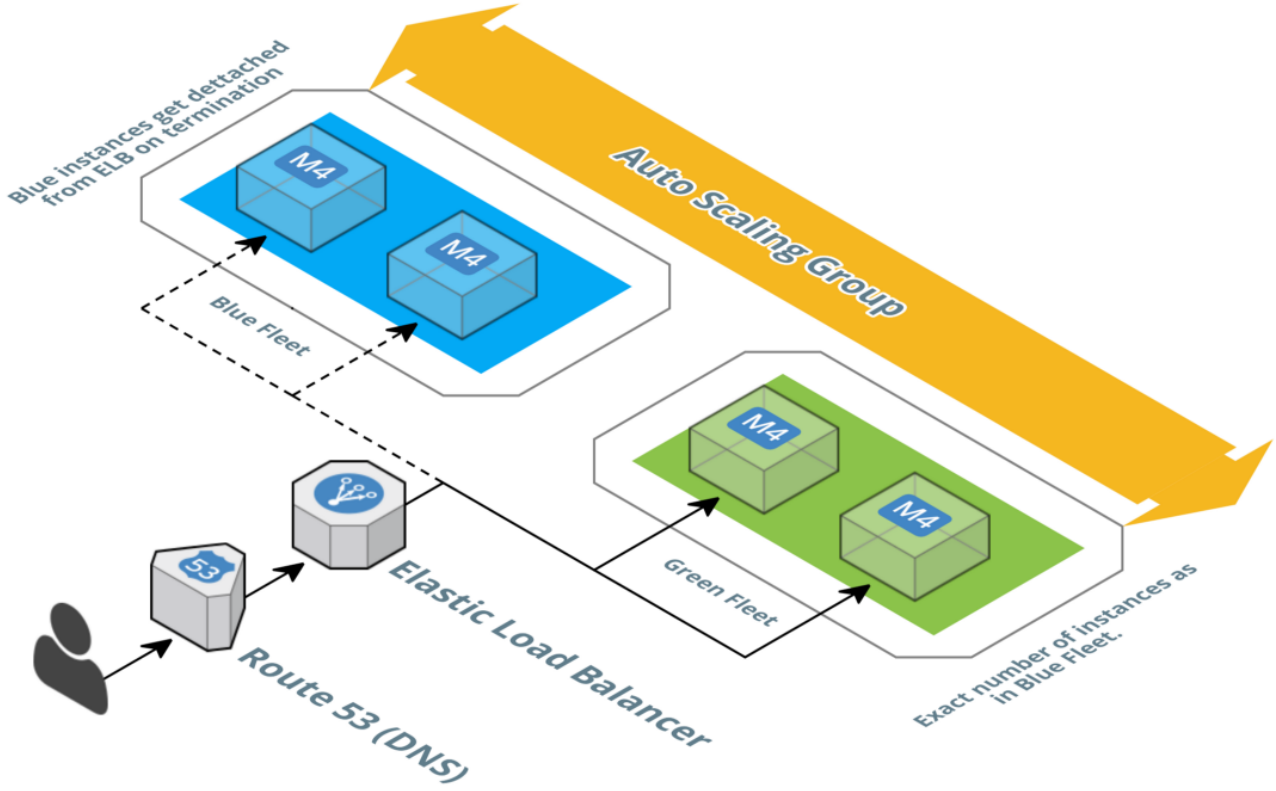
Talk is cheap, show me the CODE!

There can be two approaches to implement blue green deployments.

1. We can replace the blue(exiting) EC2 instance with green(with latest code) EC2 instances. In this approach, we create the same amount of instance in the existing infrastructure, update the instances with new code and add them behind the load balancer and kill blue instances.
2. Or, we can make the switch on DNS level using Route 53. In this approach we create an exact replica of our infrastructure with updated code i.e. a new ASG, new ELB and new set of instances that will contain the latest code.

In the next section we give implementation details for both the approaches.

Approach 1



Step 1. We get the details of the production ASG. We define a function `describe_asg` which takes the name of the ASG and gets details using Boto.

```
def describe_asg(asg):
    client = boto3.client('autoscaling')
    response = client.describe_auto_scaling_groups(
         AutoScalingGroupNames=[asg])
    auto_scaling_groups = response['AutoScalingGroups']
    if auto_scaling_groups:
         return auto_scaling_groups[0]
    return None
```

Step 2. Using `asg_details` we can get the number of instances and also the LC name. We use LC name to describe details like AMI id, instance type etc. Now we spun off instances with exact same configuration. We will update these instances with latest code.

```
def get_lc_by_name(name):
    """Get Launch Configuration by name
    """
    conn = boto.connect_autoscale()
    lcs = conn.get_all_launch_configurations()
    token = lcs.next_token
    while True:
        if token:
            r = conn.get_all_launch_configurations(next_token=token)
            token = r.next_token
            lcs.extend(r)
        else:
            break
    for lc in lcs:
        if lc.name == name:
            return lc
    else:
        raise AssertionError("Launch Configuration by name "
                             "'%s' doesn't exist" % name)

def start_deployment_instance(asg, instance_name):
    print("Starting deployment instance.")
    ec2_con = boto.connect_ec2()

    # Create instance as this will get attached to the Latest ASG.
    depl_instance = None
    lc = get_lc_by_name(asg["LaunchConfigurationName"])
    res = ec2_con.run_instances(image_id=lc.image_id,
                                security_group_ids=lc.security_groups,
                                instance_type=lc.instance_type,
                                ebs_optimized=False,
                                instance_profile_name=lc.instance_profile_name)
    depl_instance = res.instances[0]
    # Wait for instnance to get in running state.
    while depl_instance.update() == 'pending':
        print('.', end='')
        sys.stdout.flush()
        sleep(10)
    if depl_instance.state != 'running':
        raise Exception('Deployment Instance cannot be launched,\
                         Instance state =' + depl_instance.state)
    print('\nInstance started')
    print("Waiting 60 secs for instance reachability check to pass")
    sys.stdout.flush()
    sleep(60)

    depl_instance.add_tag('Name', instance_name)
    depl_instance.add_tag('service', 'cerebro')

    return depl_instance


old_instance_ids = map(lambda x: x["InstanceId"], asg_details["Instances"])
# Launch deployment instance
depl_instances = [
    start_deployment_instance(asg_details, depl_inst_name + "-" + str(i))
        for i in range(0, len(old_instance_ids))
```

```
]
```

Step 3. We use Python Fabric to update code in new instances. With Fabric we define `task` to `run` on remote machines. We use `env.rolesdef` to define machine roles. Any task attached to a role will only be executed in a machine with same role. Here we are defining `deployment` role. We will add all new instances to this role. We can define `tasks` like `update_code` , `validate_service` etc.

```
from fabric.api import env, run, roles, task, execute

def init_env():
    env.user = 'ubuntu' # Fabric will ssh into remote using this user
    env.use_ssh_config = True # Fabric will use ssh config to ssh.
    env.timeout = 30 # ssh attempt will fail after 30 secs
    env.connection_attempts = 4
    env.parallel = True # A task will be executed simaltaneously on all servers
    env.roledefs = LazyDict({ # Role that a remote server has.
        'deployment': lambda: []
    })

@task
@roles('deployment')
def update_code():
    print("Updating code.")
    run("""
      # Add your deployment script here
      # You can create multiple tasks in order to:
      # - Update dependencies
      # - Start the application
      # - restart your service
      # - Warmup the service in case its required
      # All it need is to create multiple functions like this
      # and execute them sequentially.
    """)

# Start deployment instance
deployemnt_instance = start_deployment_instance()
# Initialize fabric env
init_env()
# Add deployment instance ip/hostname to deployment role
env.rolesdef['deployment'].append(deployment_instance.public_ip)

# update code on deployment instances
execute(update_code)
# execute(update dependencies)
# execute(service_restart)
# execute(validate_service)
# execute(warmup_service)
```

Step 4. Now that we have the latest revision of code available in the deployment instances we need to bake an image. This image will be used by the ASG in case of a scale out. AWS provides a way to create image without the need for a server to reboot. We are using this option so that we don’t need to wait for service to be up and running before we attach it to the ASG in next steps.

```
def create_image(instance_id, image_name, no_reboot=False):
```

```
    ec2_conn = boto.connect_ec2()
    image_id = ec2_conn.create_image(instance_id,
                                     image_name,
                                     'Image created using deplex',
                                     no_reboot=no_reboot)
    sleep(5)  # Sleep till EC2 API becomes consistent

    # Set Name tag for the image being created
    image = ec2_conn.get_image(image_id)
    image.add_tag('Name', image_name)

    print('Waiting for image to be done ', end='')
    while image.update() == 'pending':
        print('.', end='')
        sys.stdout.flush()
        sleep(15)
    if image.state != 'available':
        print("\nImage cannot be created. Image state =", image.state)
        return
    print('\nImage created')

    return image_id
```

Step 5. Now we have to attach the deployment instances to the ASG and ELB and terminate all the old instances. We have to attach the instances to ASG first and then to ELB as this is a precondition for ELB. While attaching the the ASG we may have to increase the max capacity of instances ASG can increase to scale out. When we attach the instances to ASG it increases the desired capacity. ASG throws an error if we try to add more instances than the max capacity. Also, we must check that instances in ELB pass health checks i.e. they are in service. If instances are not in service it indicates that some error has occurred and the deployment should **fail**.

```
elb_name = asg_details['LoadBalancerNames'][0]
def attach_with_asg(asg_details, instance_ids):
    client = boto3.client('autoscaling')
    asg_name = asg_details['AutoScalingGroupName']
    # Update ASG max capacity.
    max_capacity = asg_details['MaxSize']
    desired_now = asg_details['DesiredCapacity']
    new_max_capacity = max(len(instance_ids) + desired_now, max_capacity)
    if len(instance_ids) > max_capacity:
        response_asg_update = client.update_auto_scaling_group(
            AutoScalingGroupName=asg_name,
            MaxSize=new_max_capacity
        )
    # Attach instances to ASG
    instances = [{"InstanceId": id} for id in instance_ids]
    response_asg_attach = client.attach_instances(
        InstanceIds=instance_ids,
        AutoScalingGroupName=asg_name
    )

def attach_with_elb(elb_name, instance_ids):
    client = boto3.client('elb')
    response_lb = client.register_instances_with_load_balancer(
        LoadBalancerName=elb_name,
        Instances=instances)

def check_if_instances_in_service(elb_name, instance_ids):
    elb_client = boto3.client('elb')
```

```
    retries = 0
    are_instances_healthy = False
    instances = [{"InstanceId": id} for id in instance_ids]
    while retries < 30 and not are_instances_healthy:
        instance_health = True
        response = elb_client.describe_instance_health(
            LoadBalancerName=elb_name, Instances=instances)
        elb_instances_health = response['InstanceStates']
        for instance in elb_instances_health:
            current_instance_in_service = instance['State'] == 'InService'
            instance_health = instance_health and current_instance_in_service
        if instance_health:
            are_instances_healthy = True
            break
        sleep(10)
        retries = retries + 1
    return are_instances_healthy
```

Step 6. If *Step 5.* succeeds we can now update the ASG to use the new image to spun off new instances. To update the ASG we will have to create a new LC which will be an updated copy of the old LC. We replace the image id with the newly formed AMI and attach new LC to the ASG.

```
def copy_lc_with_new_ami(old_lc_name, ami_id, new_lc_name):
  as_conn = boto.connect_autoscale()
  old_lc = get_lc_by_name(old_lc_name)
  lc = LaunchConfiguration(name=new_lc_name,
                           image_id=ami_id,
                           instance_type=old_lc.instance_type,
                           user_data=old_lc.user_data)
  as_conn.create_launch_configuration(lc)
  return lc.name

def update_asg_with_lc(asg_name, lc_name):
  asg = describe_asg(asg_name)
  asg.launch_configuration = lc_name
  asg.update()
```

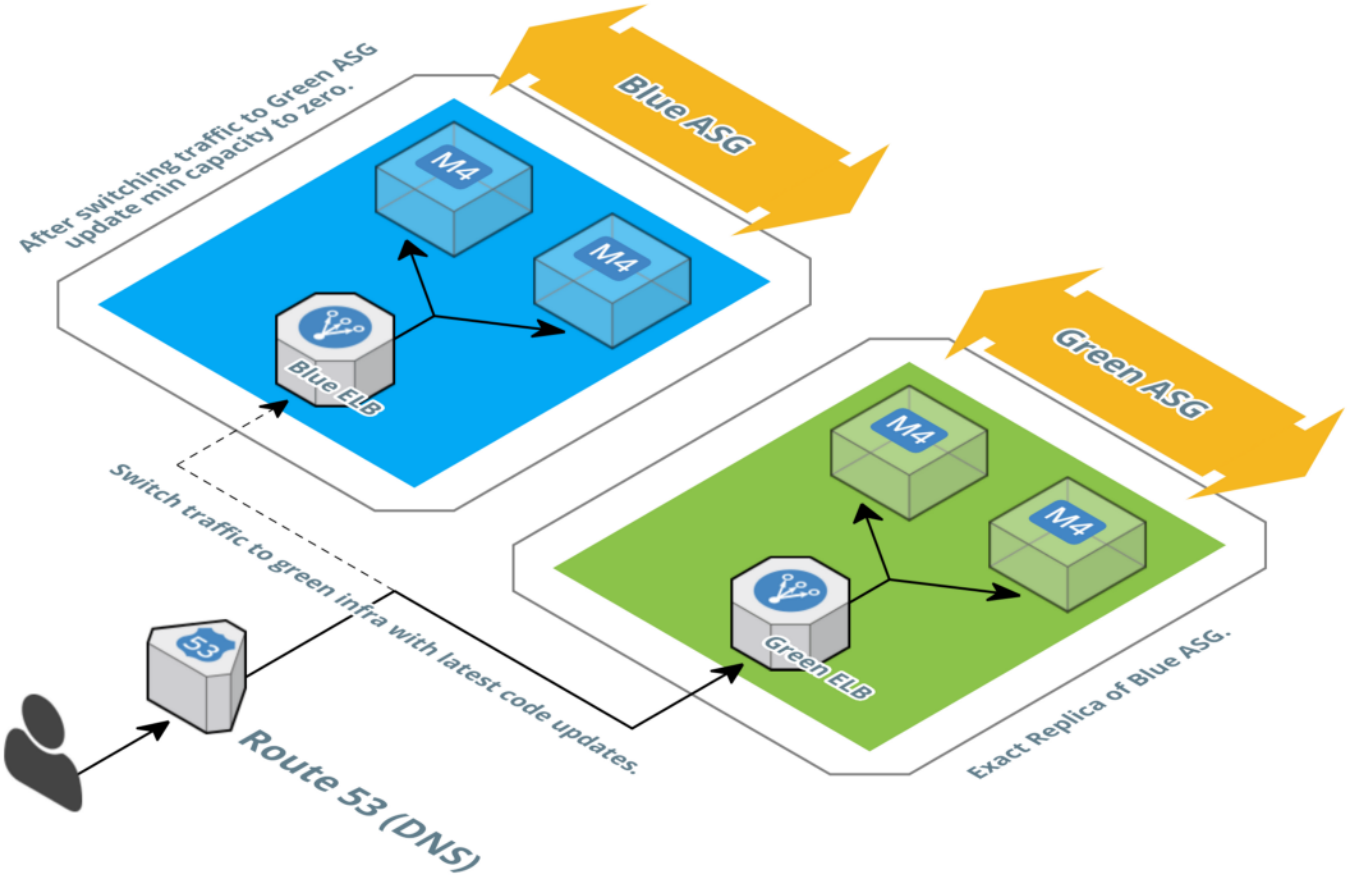
Step 7. If all the above steps succeed then we will have a set of instances with old code and a set with new code. Now we need to terminate the old instances.

```
  def terminate_instances(instance_ids):

    ec2_conn = boto.connect_ec2()
    ec2_conn.terminate_instances(instance_ids)
```

So, we have replaced the infrastructure with new code successfully.

Approach 2



This approach is a simpler one as we depend on Amazon ASG to create the instances and attach them to the ELB. All we got to do is create an AMI with updated code and create an ASG.

To create and AMI with updated code we can follow the *Step 1, 2, 3 and 4* of **Approach 1**. Note that in *Step 2* we don't need to create an exact number of instances we just need to create an AMI using only one deployment instance. Once that is done we may terminate the deployment instance using *Step 7* of **Approach 1**.

Creating ASG Replica

Step 1. First up we create a new LC using `copy_lc_with_new_ami`. We use the AMI with updated code in this LC.

Step 2. Now we create a new ELB that we will attach to the new ASG. This ELB has to be configured exactly as the older one. So we will copy the `listeners`, `security_groups`, `subnets` and `health_checks` .

```
def describe_elb(elb_name):
    client = boto3.client('elb')
    response = client.describe_load_balancers(
        LoadBalancerNames=[origin_elb_name])
```

```
    elbs = response['LoadBalancerDescriptions']
    if not elbs:
        raise Exception('No ELB by name ' + origin_elb_name)
    return elbs[0]

def get_elb_listeners(elb_details):
    listeners = []
    for ld in elb['ListenerDescriptions']:
        listeners.append(ld['Listener'])
    return listeners

def create_elb_replica(old_elb_name, new_elb_name):
    old_elb_details = describe_elb(old_elb_name)
    listeners = get_elb_listeners(old_elb_details)
    health_checks = old_elb_details["HealthCheck"]
    # Create new elb while copying listeners and tags and new name
    client.create_load_balancer(
        LoadBalancerName=new_elb_name,
        Listeners=listeners,
        Subnets=elb['Subnets'],
        SecurityGroups=elb['SecurityGroups'],
        Scheme=elb['Scheme'],
        Tags=origin_tags)
    client.configure_health_check(
        LoadBalancerName=new_elb_name,
        HealthCheck=health_checks)
```

Step 3. We create a new ASG using the newly created ELB and Launch configuration. We use `check_instances_in_service` to validate success of this step.

```
def create_asg_replica(old_asg_details, new_asg_name, new_lc_name,
new_elb_name):
    client = boto3.client('autoscaling')
    client.create_auto_scaling_group(
        AutoScalingGroupName=new_asg_name,
        LaunchConfigurationName=replica_lc,
        MinSize=asg['MinSize'],
        MaxSize=asg['MaxSize'],
        DesiredCapacity=asg['DesiredCapacity'],
        DefaultCooldown=asg['DefaultCooldown'],
        AvailabilityZones=asg['AvailabilityZones'],
        LoadBalancerNames=[new_elb_name],
        HealthCheckType=asg['HealthCheckType'],
        HealthCheckGracePeriod=asg['HealthCheckGracePeriod'],
        VPCZoneIdentifier=asg['VPCZoneIdentifier'],
        TerminationPolicies=asg['TerminationPolicies'],
        NewInstancesProtectedFromScaleIn=asg[
            'NewInstancesProtectedFromScaleIn']
    )
```

In order for ASG to scale in/out we need to configure the scaling policies. This can be done using AWS Cloudwatch Alarms and Simple Scaling Policies. So, we create a cloudwatch alarm which can monitor a metric and notify the scaling policy to perform the desired action.

```
def create_scaling_policy(asg_name, policy_name, scaling_adjustment):
    client = boto3.client('autoscaling')
    replica_scale_down_policy = client.put_scaling_policy(
```

```
        AutoScalingGroupName=asg_name,
        PolicyName=policy_name,
        PolicyType='SimpleScaling',
        AdjustmentType='ChangeInCapacity',
        ScalingAdjustment=scaling_adjustment, # defines instances to be added or
removed
        Cooldown=300 # wait time between 2 policy breach.
    )

# Cloudwatch alams can notify the listeners to perform desired actions.
# In this case we need to pass the policy arn to be configured as alarm action.
# Here we are creating an alarm on CPU utilization metric.
def create_alarm_for_policy(
    alarm_name,
    alarm_description,
    policy_arn,
    asg_name,
    evaluation_period # number of times the metric should be checked before
alarming.
):
    client = boto3.client('cloudwatch')
    client.put_metric_alarm(
        AlarmName=alarm_name,
        AlarmDescription=alarm_description,
        ActionsEnabled=True,
        AlarmActions=[policy_arm],
        Statistic='Average',
        Dimensions=[
            {
                'Name': 'AutoScalingGroupName',
                'Value': asg_name
            },
        ],
        MetricName='CPUUtilization',
        Namespace='AWS/EC2',
        Period=60,
        EvaluationPeriods=evaluation_period,
        Threshold=55,
        ComparisonOperator='GreaterThanOrEqualToThreshold' # Set as desired.
    )

scale_up_policy = create_scaling_policy(asg_name, 'asg_scale_up_policy', 1)
create_alarm_for_policy(
    'asg_scale_up_alarm',
    'description',
    scale_up_policy['PolicyARN'],
    'my_asg_name',
    2)
```

Step 4. Now that ASG is set up and there are instances ready to serve traffic behind the new ELB last step is to switch the traffic using Route53. We will create an A record entry in Route53 to redirect all traffic to our desired domain to reach to green ELB's DNS.

```
def switch_traffic(elb_name, dns_name, hosted_zone_id):
    client = boto3.client('elb')
    elb_details = client.describe_load_balancers(
        LoadBalancerNames=elb_name)['LoadBalancerDescriptions'][0]
    elb_dns = elb_details['DNSName']
    elb_hosted_zone = elb_details['CanonicalHostedZoneNameID']
    r53client = boto3.client('route53')
```

```
    r53client.change_resource_record_sets(
        HostedZoneId=hosted_zone_id,
        ChangeBatch={
            'Comment': 'Switching to new ELB',
            'Changes': [{
                'Action': 'UPSERT',
                'ResourceRecordSet': {
                    'Name': dns_name,
                    'Type': 'A',
                    'AliasTarget': {
                        'HostedZoneId': elb_hosted_zone,
                        'DNSName': elb_dns,
                        'EvaluateTargetHealth': False
                    }
                }
            }]
        }
    )
```

Step 5. As traffic is now being served by the Green ASG we do not need the Blue one. We will set the minimum capacity of Blue ASG to zero. This will scale down the Blue ASG gracefully.

```
def update_asg_min_capacity(asg_name, capacity):
    client = boto3.client('autoscaling')
    response = client.update_auto_scaling_group(
        AutoScalingGroupName=asg_name,
        MinSize=capacity
    )
```

Thus we have successfully updated our infrastructure in Blue Green Approach. We can delete the Blue ASG(old) once sanity of the Green ASG has been ensured.

Follow us on twitter for regular updates. If you liked this article, please hit the ❤ button to recommend it. This will help other Medium users find it.