Pason Effective Infrastructure Management

Group Members:
Akinwale Akinsete
Craig Martis
Yujia Cui

September 12, 2019

Contents

Abstract

This report Amazon Web Services is one of the most well-known cloud computing services in the world. Terraform is a commonly-used, open-source tool that facilitates the management of cloud infrastructure as code. Terraform and AWS are commonly used together. The team has chosen to work on the Pason's Effective Infrastructure management project which involves the usage of AWS and Terraform. The main objective of the project was to develop a solution for Pason Systems Corporation in Terraform that would be able to automatically accomplish a blue-green deployment on AWS's EC2 instances in Auto Scaling Groups so that the infrastructure would exhibit zero downtime of the hosted web services during deployments and have rollback capabilities, load matching current capacity or exceeding it, and integrated health checks. In this report, we give a quick-start user guide as well as a detailed explanation of the configuration that was developed. The report also details several 'gotchas' for both Terraform and AWS that were discovered and used in the development of this Terraform configuration. Finally, it summarizes some lessons learned from alternative solutions that were attempted but did not result in adequate solutions.

1 Overall Infrastructure

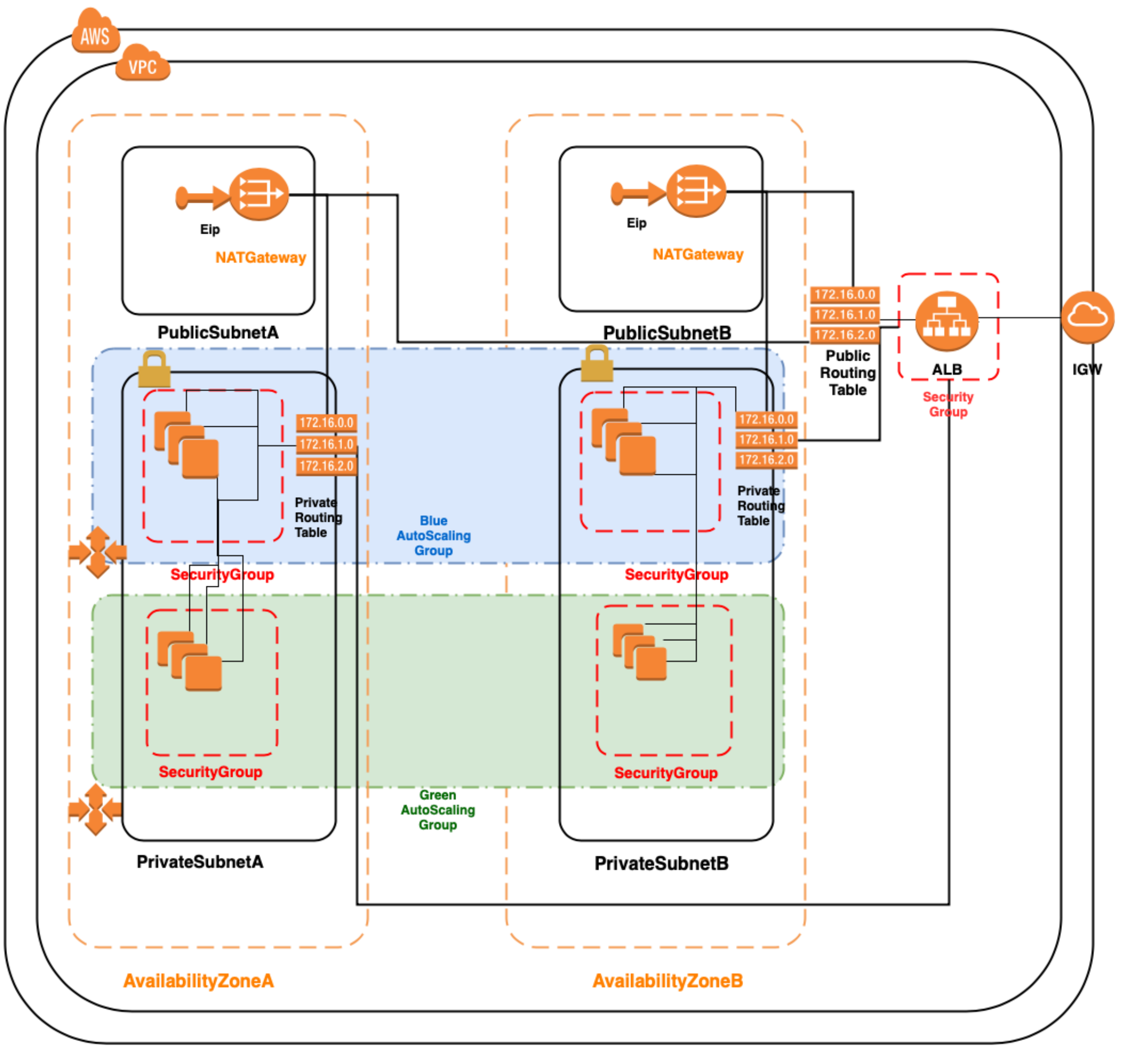


Figure 1: Infrastructure Diagram

The figure above is the schematic diagram of our overall infrastructure diagram. The VPC is spread over two AZs, each containing a private and a public subnet. The EC2 instances are located in the private subnets only and two ASGs each spans the two private subnets to contain all of the EC2 instances in both. In each public subnet is a NAT gateway and NAT EIP that provide NAT for the instances in the private subnet in the same AZ. Each private subnet has a route table. One of their rules is to route traffic to the NAT. There is a route table for the entire VPC which has one of its rules to route traffic from the NAT to the

IGW. Traffic originating from the outside world, first arrives at the IGW and then is passed on to the ALB, which send it to the appropriate EC2 instance in the ASG. A security group is attached to EC2 instances inside each private subnet, and another security group is attached to the ALB. The usage of security groups is for protection of the instances from unknown traffic that tries to access the information in the subnet.

Both ASGs should exist on the cloud at all times. Only on ASG is active at a given time, containing all the EC2 instances that host the service. The other ASG should be empty.

2 *Quick-Start Guide*

2.1 General sequence of operation

During a usual deployment of an update to a web service, a new launch configuration having those updates in new user data is first created and attached to both ASGs. Then the deployment proceeds as follows:

1. The inactive, empty ASG provisions up instances to its maximum capacity first. These instances thus contain the updated service from the new launch configuration.

2. Once Terraform has detected that all instances have been provisioned and have completed their load balancer health checks, it zeroes the old ASG.

3. From Terraform's perspective, the deployment finishes at this point; however, on the cloud, as the formerly-active ASG is zeroed, its the old instances take some time to drain before detaching from the target group and begin termination. This last step is crucial in ensuring the zero-downtime nature of the deployment.

2.2 First-time deployment

To deploy the infrastructure for the first time from a new machine, the following must be customized:

- In variables.tf, ensure the variable, "*deployment_name*", is customized to a unique string of choice to ensure infrastructure components receive a unique name and do not conflict with other, similar components on the cloud.

- In variables.tf, ensure that the variable, "*user_data_file_string*", has the appropriate path and file name of the user data to be uploaded to EC2 instances. If no user data is to be uploaded, set this to 'null'.
- In variables.tf, be sure to update the variable "*ami*" is required. The default AMI is presently set to an empty Linux machine.
- If using on a Linux/Mac machine running Terraform, be sure to run "*chmod +x checktoProceed.sh*" first from the terminal within the working directory and then ensure the reference to "*checktoProceed.sh*" in the provisioners of the null resources in null_resources.tf is preceeded with "./" On a Windows machine, the "./" or running "*chmod +x checktoProceed.sh*" are not required

2.3 Regular usage

The most common commands are '*terraform plan*', '*terraform apply*', and '*terraform refresh*'.
To upload a change of AMI and/or user data, simply change these variables ("ami" and "user_data_file_string") and run '*terraform apply*'. The configuration automatically detects when there is a change in the launch configuration and triggers a deployment. To simply anticipate what Terraform will proceed to do, running '*terraform plan* ' will show this information. Running '*terraform refresh* ' with the several outputs in variable.tf uncommented, will simply show the current status of infrastructure and information since the last deployment. Note that '*terraform refresh*' only shows information from the **last deployment**, and, by itself, will **not** detect the presence a new launch configuration of a pending deployment.

2.4 Manual forced switching (forced deployment)

There might be cases when new instances have to be deployed without being triggered to do so by a detection of a new launch configuration. These would include when the infrastructure is recovering from a major, abnormal situation on the cloud. (Such situations are further detailed in other sections below.) This option is like a 'reset' button for the ASGs and instances. It is performed by running *terraform apply* with the variable "always_switch" set to true (*terraform apply -var always_switch=true*). This manual forced switch still takes place in a zero-downtime fashion.

2.5 Dealing with errors

Almost all errors can be easily fixed by running *terraform apply* a second time. If this does not resolve the problem, perform a manual forced switch by setting always_switch to true (run *terraform apply -var always_switch=true*). An output error message is displayed in the terminal in most cases when errors are detected and a manual forced switch is required.

3 Detailed Sequence of Operation

3.1 Normal or forced deployment

The following is a detailed sequence of events during a *terraform apply* command under normal conditions where both ASGs are present and in a correct configuration (one is empty and the other has healthy, live instances), and no change in ASG maximum and minimum capacities have taken place since the previous deployment.

1. Data resources draw information about the present state of ASG1, AGS2, and the launch configuration from the AWS cloud. This information is processed and stored in locals to be used to determine the course of action for that deployment. In particular, it is important to check which ASG is active and also whether:

   (a) a new launch configuration is about to be deployed (new user data or new AMI), or

   (b) a forced manual switch is requested from the user.

   Both of these scenarios would require the deployment of new instances, so the subsequent course of action for these two cases is the same.

2. Based on the data, a null resource called ”pre-update_ASG1_status” creates a hidden text file locally called ”.ASG1Active.txt” with either true or false depending on whether ASG1 has been detected on the cloud to be active or inactive/missing. The locals proceed to calculate the whether a change to ASG2 is required based on three factors: which ASG starts as active, whether there is a new launch configuration, or whether a forced switch is requested by the user. The combination of these factors and the resulting output is shown in the table below.

ASG1 active	New launch configuration	Forced switch	**Make ASG2 active**
true	true	false	**true**
false	true	false	**false**
true	false	false	**false**
false	false	false	**true**
true	true	true	**true**
false	true	true	**false**
true	false	true	**true**
false	false	true	**false**

These combinations can be summarized in the following logical statement:

$$change\_to\_ASG2 = (force\_switch == false \&\& ASG1\_is\_active == new\_LC) \\ || (force\_switch \&\& ASG1\_is\_active) \quad (1)$$

Based on the whether $change\_to\_ASG2$ is true or false, the values for the maximum, minimum, desired, and the wait_for_elb_capacity for each ASG can be determined and are automatically set for that particular deployment.

Make ASG2 Active	ASG name	Max	Min	Desired	Capacity to wait for
false	ASG1	max_value	min_value	max_value	max_value
	ASG2	0	0	0	0
true	ASG1	0	0	0	0
	ASG2	max_value	min_value	max_value	max_value

3. At this point the Auto Scaling policies attached to eash ASG are destroyed to prevent them from interfering with the deployment of new instances. The null resources, change_detected_ASG1 and change_detected_ASG2, are triggered (either by the launch configuration if a new launch configuration was detected or if the forced switch variable has been set to true).

4. Each of these null resources runs a provisioner that calls the bash file, *checktoProceed.sh*. The bash file script checks the value of the .ASG1Active.txt file and allows the ASG that is found to be inactive to proceed first, while the other one waits. This sequence is crucial to ensure zero downtime.

   - If the value in .ASG1Active.txt is true, (meaning that ASG1 is active) it would allow the provisioner of the change_detected_ASG2 null resource to complete first and change_detected_ASG1's provisioner would still continue to run, waiting for the value in the text file to change to false.

- If the value in .ASG1Active.txt is false, (meaning that ASG1 is inactive) it would allow the provisioner of the change_detected_ASG1 null resource to complete first and change_detected_ASG2's provisioner would still continue to run, waiting for the value in the text file to change to true.

The ASGs themselves depend on the completion of their respective 'change_detected' null resource. Thus, by allowing the null resource attached to the inactive ASG to complete first, the inactive ASG always goes first and deploys it instances before the active ASG is zeroed, whose 'change_detected' null resource is still waiting.

5. Terraform waits till the inactive ASG completes the deployment of all of its instances and AWS returns successful ELB health checks from all of its instances. Whenever deploying new instances, that value will always be the maximum capacity that has been set for that ASG.

6. ASG1 and ASG2 trigger two other null resources respectively, set_ASG1_post_status and set_ASG1_post_status, once either ASG completes their deployments. These null resources update the state of the .ASG1Active.txt, based on the deployment that has been made. Thus, once the inactive ASG has finished deploying its new instances and they are all deemed healthy, that ASG will trigger its 'set_post_status' null resource to flip the value of .ASG1Active.txt.

7. The change in value in .ASG1Active.txt will now cause the change_detected null resource of the previously-active ASG to complete, allowing that ASG to now be zeroed.

8. Once the previously-active ASG is zeroed, the Auto Scaling policies are recreated on the cloud. This ASG will also trigger its 'set_post_status' null resource. However, this will have no influence on the course of events and no change in the value of .ASG1Active.txt, since it was already changed by the 'set_post_status' null resource from other the ASG that was first to run.

9. The deployment is complete in Terraform. However, on the cloud, as the previously-active ASG is zeroed, AWS will begin draining and detaching each of its old instances from the target group. This draining process is another key action that must take place to ensure zero downtime. After

the last instance is drained and detached, the deployment is now entirely complete.

N.B. A small detail to note in the case of a forced switch (setting *always_switch=true*) is that on the subsequent *terraform plan* or *terraform apply*, minor local changes will be required to the null resources. No changes take place on the cloud. These changes to the null resource takes place because the triggers need to be revert from detecting the forced switch back to monitoring the launch configuration, which is the default value. This behaviour is further explained in future sections of the report.

Figure 2 summarizes the steps described thus far in a visual manner. In the case shown, ASG2 starts as inactive and ASG1 as active.

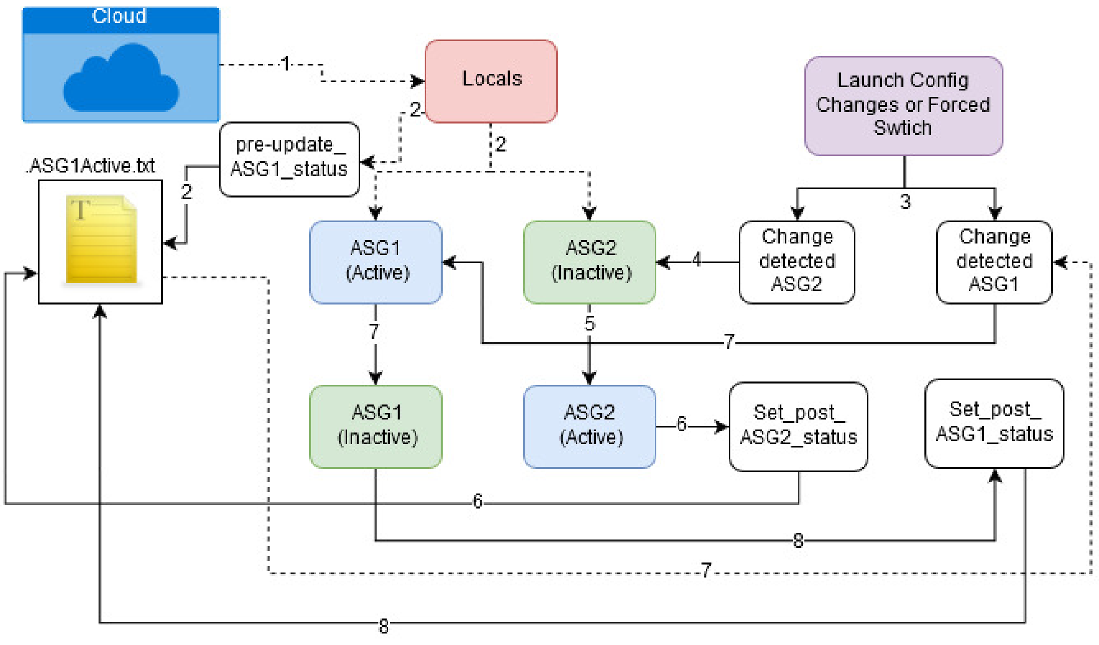


Figure 2: Sequence of Steps in Normal or Forced Deployment

3.2 Actions taken when no deployment is present

When there is no request to force a switch detected from the user or no change detected in the infrastructure (including changes to the launch configuration) the min, max, desired capacity, and wait_for_elb_capacity values for each ASG calculated by the locals do not change from the preceding deployment. Crucially, the only exception is the desired capacity is automatically set to null to ignore changes that might have occurred to the ASGs' capacities due to Auto Scaling by AWS since the last Terraform deployment. In this way, Terraform can continue to monitor the infrastructure in between new deployments for undesired drift.

3.3 Changing the capacity bounds of the ASGs

Changing the capacity (max and min) bounds of the ASGs do not cause the configuration to switch which the active ASG to inactive and vice versa. The only time any minor changes are required are when the maximum capacity is made lesser than the current capacity or the minimum capacity is made greater than the current capacity. In these two cases, instances either have to be deployed or destroyed to fall within the new ASG capacity bounds. Similar to the normal deployment process, the Auto Scaling polices are temporarily destroyed and the active ASG's desired capacity is set to the maximum. However, the active and inactive ASGs do not change.

3.4 Abnormal cases: missing ASGs, both active, or both zero

Terraform will ignore its provisioners in any abnormal situation.

- If the active ASG or both ASGs are missing, there are no live web services initially and so zero-downtime deployment is of no concern for that deployment. Thus, the order in which Terraform will deploy the infrastructure is not important and Terraform will be allowed to perform any pending deployment in whatever order it determines best.
- In the case, where the inactive ASG is missing or both ASGs are found active, live services exists on the cloud, but, due to the abnormal setup, Terraform cannot guarantee a zero-downtime deployment of any new user data in a single *terraform apply* step. Thus, to prevent interruption to the web service, the locals will ignore any pending deployment or forced switch and will first bring the infrastructure on AWS back to a normal. It will then prompt the terminal to immediately run a follow-up forced switch command, *terraform apply -var always_switch=true* to effect the desired changes to the web service in a subsequent deployment.

4 Terraform: Tips, Tricks, and Gotchas

4.1 Triggering resources based on changes in a resource or the state of a variable: Auto Scaling Policies and Forced Switching

To trigger a resource that is not a null resource, interpolate the name/id with the name of the triggering resource. A deletion or creation of the triggering resources will therefore cause the resource with the interpolated name to be triggered and recreated as well. If a resource is to be triggered based on an in-place update of another resource, this can be done by a null resources trigger being set to the attribute of the triggering resource and then interpolating the name/id of the triggered resource with the id of the null resource.
This was done to trigger the replacement and destruction of the Auto Scaling policy whenever a change was detected in the max or min values of that ASG.
Further, if a null resource needs to be triggered by the state of a variable, the following is one approach. The trigger must be conditionally set, based on whether the variable is true, to a Terraform function that always runs, such as $timestamp()$. This way, the null resource will always be called whenever that variable is true, which, in our case, is the variable dependent on whether $always\_switch$ is set to true. If the variable is false, it should default to an alternative trigger or null. In our case, this default trigger was the launch configuration ID.
One minor problem is that when the variable is reverted back to false, Terraform has to replace the null resource once more to revert the triggers from $timestamp()$ back to the default trigger value. However, if the provisioners can be made to be conditionally ignored in such a situation, this necessary step does not affect anything else.

4.2 About the create_before_destroy meta-argument

This lifecycle meta-argument is best explicitly set to true for all resources as a general rule, with the important exception of Auto Scaling Policies, where it should be set to false. Doing this, it was possible to cause the policies to be destroyed before and created after each deployment to avoid interference with the provisioning of new instances during a deployment. Without doing this, the scaling policy would interfere with the creation of new instances, preventing the ASG from reaching the maximum value in wait_for_elb_capacity and thus cause Terraform to deem the deployment incomplete.

4.3 Conditional provisioners

It might be important based on the design of the Terraform configuration to add conditionals to the execution of the provisioners, to prevent their execution in certain situations. This can be done by having a default scenario for the case when the condition to use the provisoners evaluates to false, such as *"echo blank step"*.
In the case of this configuration, the provisoners are ignored in cases where zero-downtime deployment does not apply. This includes when either ASG is missing, both are active or inactive, or simply when the locals do not forsee a switch in changing which ASG is active.

4.4 Difference in the behaviour of various optional arguments and conditionally setting the ASG's desired capacity

Resources in Terraform can have certain arguments as optional for the definition of the resource. However, it was found that different optional arguments behave differently, when it comes to setting or ignoring their values between deployments. Most optional arguments such the the $user\_data$ of the launch configuration or the $wait\_for\_elb\_capcity$ in the ASG behave in such a way that the change from a non-null value to the a null value would require the resource be updated on the subsequent *terraform plan* or *terraform apply*. Thus, to avoid repeated subsequent changes back and forth, it is best to keep these value as consistent as possible.
However, the $desired\_capacity$ argument in the ASGs (thankfully) does not behave this way. When it is set to null, Terraform simply ignores it. The value doesn't actually get set to null but reads the desired capacity value that is currently on the AWS cloud for that ASG. Since that argument reflects the current intended capacity of the ASG that is being actively scaling by AWS, it is very useful that changes to this value can be ignored simply by setting the value to null.
For this reason, it is a better choice to set and fix the $wait\_for\_elb\_capcity$ to the maximum value for the active ASG and change the desired capacity to match it, rather than changing the more static $wait\_for\_elb\_capcity$ value to match the dynamic $desired\_capacity$ argument. Consequently, this also explains why we scale the ASG to the maximum value whenever instances are destroyed or deployed and consequently, also why the Auto Scaling policies need to be temporarily destroyed during the course of the deployment and recreated at the end to prevent them from restraining the active ASG from reaching its $wait\_for\_elb\_capacity$ maximum capacity.

In practice, this means that the *desired_capacity* argument is conditionally set to the ASG's maximum capacity whenever there is a new launch configuration, a forced switch, or a change in the range of its capacity's bounds beyond it current capacity. Otherwise it is set to null and is ignored by Terraform.

4.5 Wait_for_elb_capacity behaviour

It is important to note that this parameter is the exact number of instances that the ASG must reach in order for Terraform to deem a deployment complete. However, it only works when an ASG is being update in-place and not when it is being created. This is the reason why, when the inactive ASG might be found to be missing, Terraform could not attempt a zero-downtime deployment until that missing inactive ASG had been recreated. Only after this, could Terraform perform zero downtime deployments.

4.6 Handling Cycles and Nulls with Data Resources

A way to break cycle errors in ASG data resources is to use the hard-coded name (usually from a local) of the ASG that is being queried rather than retrieving that name from the resource. This setup can run the risk of returning a null, if the ASG with that name has been destroyed via the console or not yet created. To prevent this from throwing an error and stopping the deployment, another data resource capable of handling nulls should be used, namely *data.aws_autoscaling_groups*. If an ASG having the desired name is returned, the count of another data resource that specifically queries that ASG must be changed to 1. Otherwise it should be set to zero. In this way, data from the ASG can be retrieved and used to modify the ASG during the same apply action, or proceed to simply create that ASG if it does not exist.

4.7 Handling indexed (data) resources when count=0

It was found that if the index *count = 0*, any lines of code that would depend on references to those resources would not be executed. If the results of these lines were required for further ternary computations, those would also be null. The end result was that certain arguments in certain resources that used those values would also be null, giving, in our case, a run-time error, such as with the command argument in the provisioner of the null resource, *change_detected*, being empty.
To prevent this, it was necessary to put any uses of the indexed data resources in a ternary that first checks whether the data resources count is greater than

zero before attempting to retrieve information from the data resource, or else set that local to a default value (usually false).

4.8 About the Depends_on meta-argument

Depends_on must reference resources in the form of a static list. An error will result if a static list is not used. Secondly, it must be a direct internal reference to another resource in the state. Otherwise the configuration will not understand the reference, if, for example, simply the string of the name/id of that resource is used, and the desired dependency will not take effect, during the apply phase.

4.9 Breaking the cycle for the dynamic dependency between ASGs: The reason for several null resources

For our configuration to guarantee zero downtime during deployments, it is necessary that the *depends_on* argument changes between ASG1 depending on ASG2 (if ASG2 started empty) and ASG2 depending on ASG1 (if ASG1 started empty). However, this results in a cycle error. While ternary conditions can be used for *depends_on* arguments to apparently steer the dependency chain one way or the other, they still do not prevent the cycle error. Further, the cycle cannot be broken by using a hard-coded value as before with data resources, since a *depends_on* must use the reference to the resource stored in the state. Thus a more complex solution was employed.
An initial null resource would run its provisioner at the start of the deployment to create a text file with state of whether AGS1 was presently active or not. Then each ASG was made to depend on a respective null resource that would run a provisioner, which would read in the status of ASG1 from the text file.

- If ASG1 was listed as active, the null resource tied to ASG2 would exit immediately and allow ASG2 to deploy first. ASG1 would wait.
- Conversely if ASG1 was listed as inactive, the null resource tied to ASG1 would exit immediately and allow ASG2 to deploy first. ASG2 would wait.

Once the first ASG has executed it would trigger a null resource that updated the text file to flip the status, causing the waiting ASG and its null resource to proceed. In this way, the inactive ASG always proceeds first.

4.10 Change in maximum and minimum ASG limits

When an inputted change in the maximum and minimum ASG limits is detected, those values will updated on the next apply. The current setup runs so that any time there is a change in either maximum or minimum, it will simply proceed to change those values on the cloud for the active ASG or upcoming ASG. However, if a change in capacity limits results in the deployment or termination of new instances, the ASG must rescale the current capacity to the max value for that deployment. So whenever any instances are being added or removed, Terraform will wait till the ASG reaches the wait_for_elb capacity, which is always set to the maximum value. If it doesnt reach this value, the deployment within Terraform would be deemed incomplete and timeout. The two scenarios where this occurs are when the minimum value is set above or the maximum value is set below the current (desired) value of instances on the cloud. These cases must be detected and will change the change_capacity_lim to the maximum value so that.

4.11 Cases when to automatically cancel a planned deployment

Rogue manipulation through the console can cause the inactive ASG to be deleted or both ASGs to be made active. The infrastructure must be brought back to a normal state without interruption to any services, before deployments can be allowed. In these two cases, should a new deployment be detected when the infrastructure is in this defective state, it must be cancelled, the infrastructure must be rectified to a normal state, and then the deployment must take place on a second terraform apply cycle.
Let us examine each case of these two cases in further detail.

1. If both ASGs are initially active, one will be zeroed with a *terraform apply*. However, a deployment of a new launch configuration simultaneously would not be seen by the end-user. This is because in order for the new launch configuration to be launched on instances, they have to be brought up after the new launch configuration is added to the ASG. Any previous instances will continue to run the old launch configuration. When both ASGs are active, no new ASGs will be deployed. Thus a second, subsequent, manual, forced switch (setting always_switch=true) would be required for the new deployment to be visible.

2. If one ASG is active while the other (inactive one) has been destroyed, the situation is more delicate. Attempting to use the provisioners to ensure

zero downtime in a blue-green deployment does not work. In our Terraform configuration, when creating a new ASG, Terraform does not wait for the ELB health checks to complete. Thus, what results if that the active ASG proceeds to draining before the new instances are healthy. To prevent this from occurring, any deployments of new launch configurations are ignored if this scenario is detected on the cloud, and the apply is dedicated to first bringing back the empty ASG. The launch configuration would still change but since no new instances are being provisioned, it would not take effect. A second Terraform apply with a forced switch (setting always_switch=true) would be required.

4.12 Security groups: general tips

1. **Create all rules as separate resources otherwise they will not delete them when deleted in the code.** This is preferred instead of creating rules as blocks embedded in the security group resource because in the latter case, when the rules are deleted in the Terraform code, there are not deleted on the cloud. However, as individual Terraform resources, the rules can be deleted, created and updated as normal.

2. **Create before destroy = true** to avoid errors in Terraform. Certain errors resulted within Terraform when a security group rule resource was being replaced. The exact cause is unknown.

3. **Rules added via the console** will not be changed or tracked from Terraform (as expected). However, they might conflict with a duplicate rule that Terraform might provision. To fix this, those rules have to be manually deleted from the console.

5 AWS: Tips, Tricks, and Gotchas

5.1 Security group rules

The setup that was used mostly followed the recommended setup given by AWS for a VPC with an internet-facing ALB and private instances. This involved an AB with traffic of various kinds open for ingress from all sources. In our case, we have HTTP, HTTPS and an ICMP ingress rules. Then a single egress rule from the ALB security group to the instance security group exists to direct outgoing traffic only to any resources that would have that security group. In our setup, this would only be the private EC2 instances. In our test code, we only

opened this to port 80 with TCP protocol, but this could be set to whichever protocol and port combinations required by the web service. On the security group of the instances, an ingress rule from the ALB security exists with the same ports/protocol as the egress of the ALB security group rule in order to receive the traffic from the ALB.
One deviation from the canonical setup suggested by AWS that we had to make was the addition of an egress from the instance security group, on port 80 with TCP protocol and an open CIDR block [0.0.0.0/0]. This egress access was required to allow an instance to communicate outside the VPC to complete its start-up process. After start-up such access is not longer required. As per system logs of these instances, this communication is evident in an error message when such access is prevented during instance start-up:

$$Cannot\ find\ a\ valid\ baseurl\ for\ repo:\ amzn-main/latest$$

$$Could\ not\ retrieve\ mirrorlist$$

$$http://repo.us-west-2.amazonaws.com/latest/main/mirror.list\ error\ was\ 12:$$

$$Timeout\ on\ http://repo.us-west-2.amazonaws.com/latest/main/mirror.list:$$

$$(28,\ 'Connection\ timed\ out\ after\ 5001\ milliseconds')$$

5.2 ASGs and Target Groups

Whenever instances are removed from a target group AWS always drains them for the set deregistration delay prior to detachment and then termination. This step is crucial in ensuring zero-downtime deployment. However, it was found that if the ASG of those instances was deleted and replaced by Terraform this would cause the instances to become detached from the target group before deployment. This was the particular case in our one of our alternative solutions involving a single ASG and AWS CLI. It was necessary to use AWS CLI to detach the ASg and force Terrafrom to wait for the time of the deregistration delay before Terraform would proceed to destroy the old ASG.

6 Alternative failed attempts at zero-downtime Blue/Green deployment

6.1 AWS CodeDeploy

CodeDeploy is an AWS service specifically designed for Blue-Green deployments on the AWS cloud. We can provision a CodeDeploy resource in Terraform which would have all the settings required to setup AWS CodeDeploy Deployment group that would be responsible for the deployment. The main motivation to use this approach is that CodeDeploy is an AWS tool specifically made to perform a Blue-Green Deployment of AWS infrastructure, with zero down-time guaranteed.

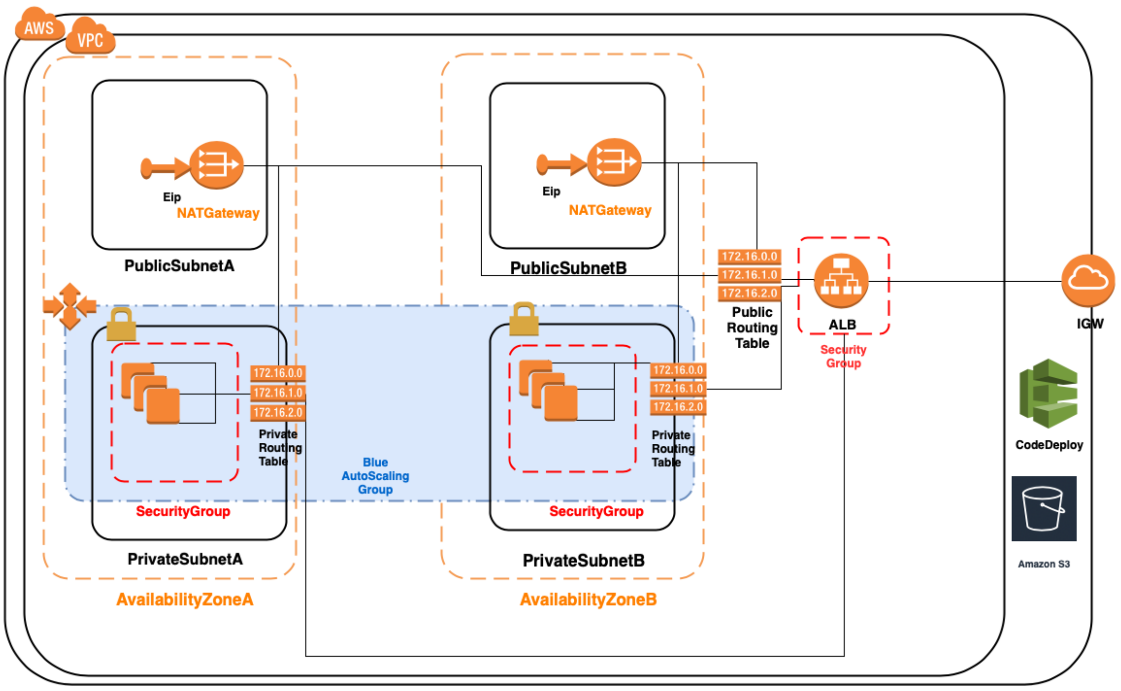
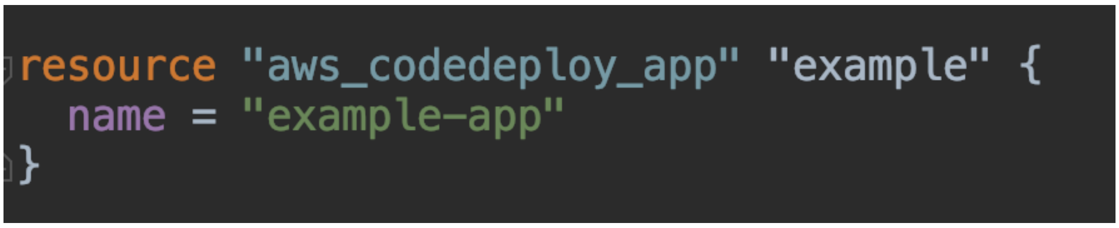


Figure 3: Infrastructure Diagram

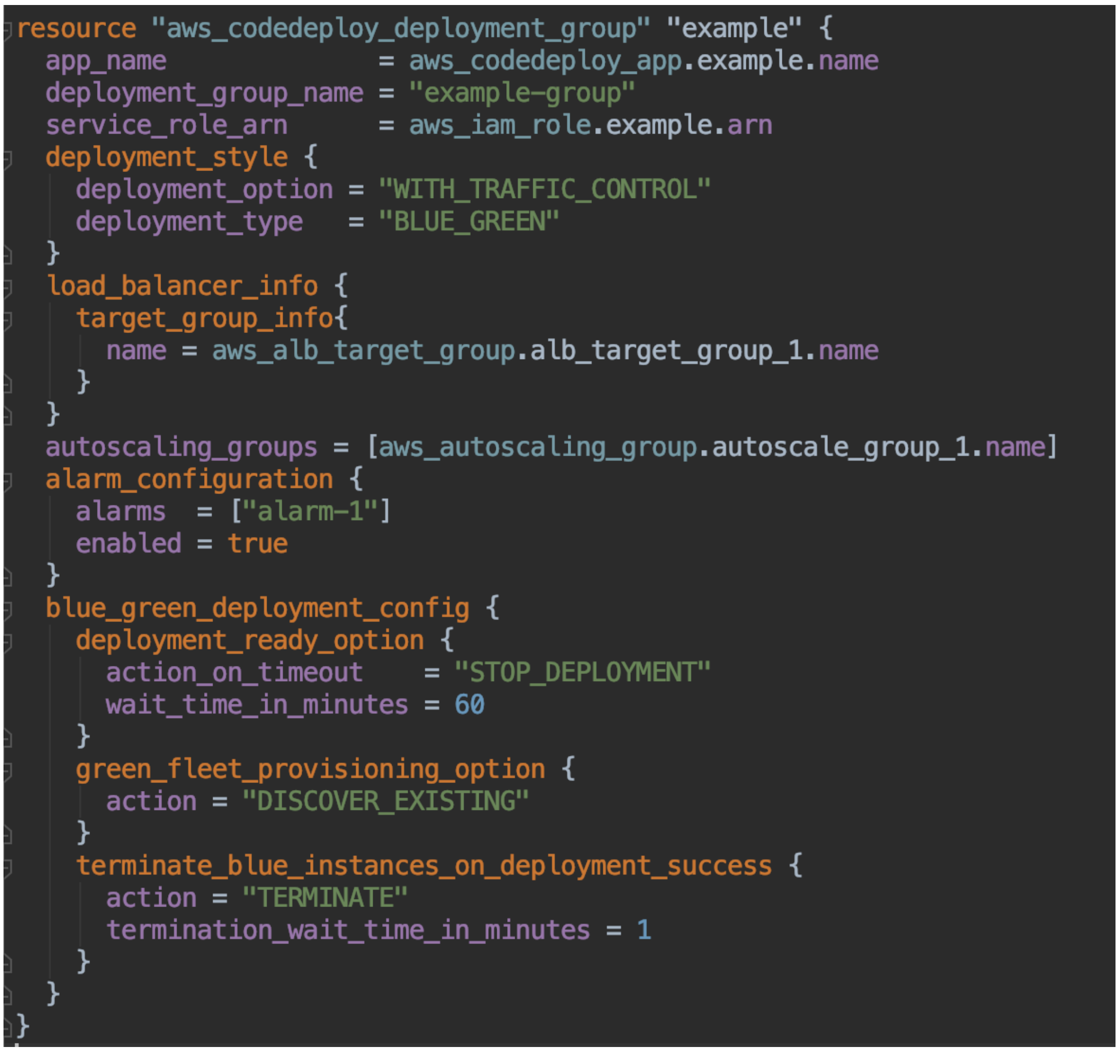
The way in which we tried this approach was by first deploying the entire blue infrastructure using Terraform as shown in the figure above. For this CodeDeploy setup, we would only use one autoscaling group with application load balancer.



```
resource "aws_codedeploy_app" "example" {
  name = "example-app"
}
```

Figure 4: Code Snippet

All the above settings are done using Terraform code. Then we would start the actual create deployment through the CodeDeploy console. In there we would specify the deployment group name, and indicate the revision location, in our case we stored our user data inside the S3 bucket. And then we would manually select on the autoscaling group that is currently in place. After all these parameters are set. We can start our blue green deployment.



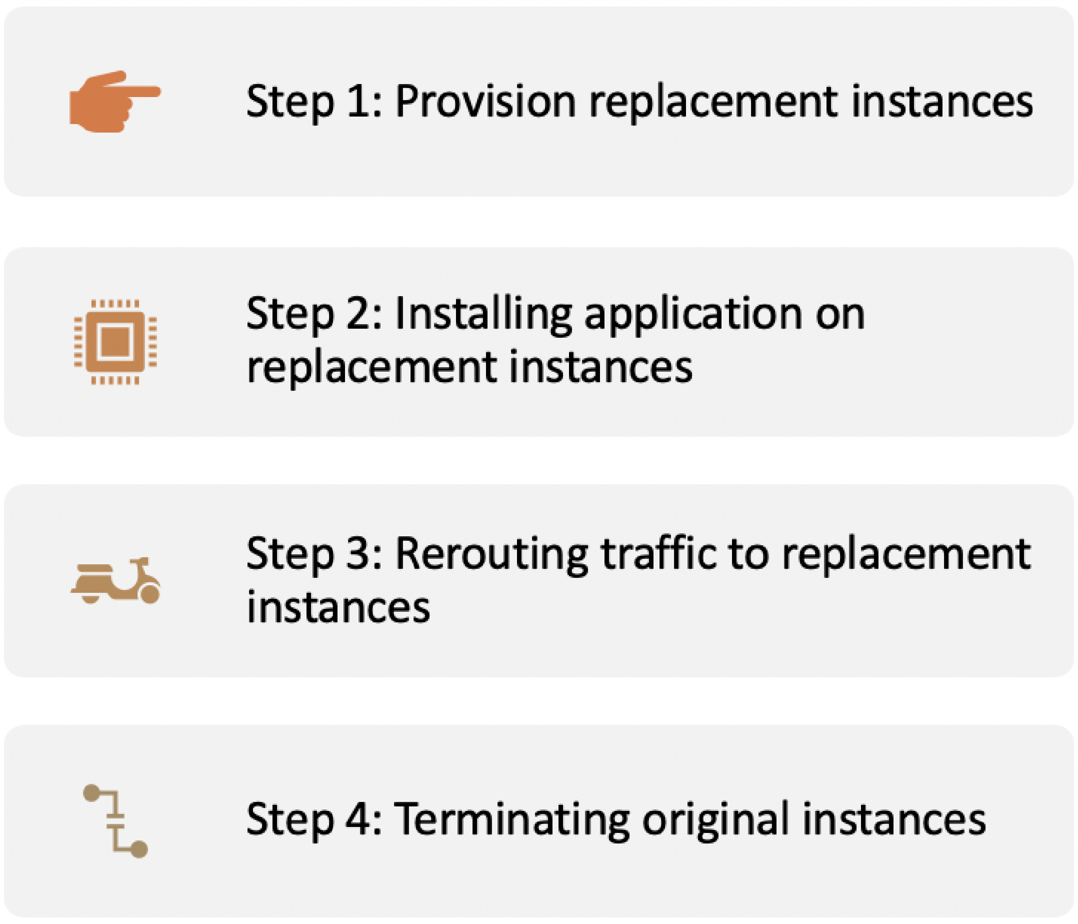
```
resource "aws_codedeploy_deployment_group" "example" {
  app_name               = aws_codedeploy_app.example.name
  deployment_group_name = "example-group"
  service_role_arn       = aws_iam_role.example.arn
  deployment_style {
    deployment_option = "WITH_TRAFFIC_CONTROL"
    deployment_type   = "BLUE_GREEN"
  }
  load_balancer_info {
    target_group_info{
      name = aws_alb_target_group.alb_target_group_1.name
    }
  }
  autoscaling_groups = [aws_autoscaling_group.autoscale_group_1.name]
  alarm_configuration {
    alarms  = ["alarm-1"]
    enabled = true
  }
  blue_green_deployment_config {
    deployment_ready_option {
      action_on_timeout    = "STOP_DEPLOYMENT"
      wait_time_in_minutes = 60
    }
    green_fleet_provisioning_option {
      action = "DISCOVER_EXISTING"
    }
    terminate_blue_instances_on_deployment_success {
      action = "TERMINATE"
      termination_wait_time_in_minutes = 1
    }
  }
}
```

Figure 5: Code Snippet

6.1.1 Zip file in S3 bucket

The zip file in S3 bucket contains a html file which displays user data on the web, a AppSpec.yml and buildspec.yml file. Within the AppSpec.yml file, html file location is indicated along with the hooks. Application stop tells the http server to stop, before install would allow the system to install the http service, and in after install, the http service will be started.

6.1.2 Blue/Green deployment Procedures



Step 1: Provision replacement instances
Step 2: Installing application on replacement instances
Step 3: Rerouting traffic to replacement instances
Step 4: Terminating original instances

Figure 6: CodeDeploy Steps

CodeDeploy uses four steps in performing blue-green deployment. CodeDeploy would first start by creating a new ASG containing same number of instances as originally ASG. Next, it would upload to each instance the appropriate user data. Next, it would start routing traffic from the original, blue ASG to the new, green ASG. Once this is all done, the shutdown of the original blue ASG can begin, along with its instances.

6.1.3 Advantages & Drawbacks

This method did indeed provide a zero-downtime, blue-green deployment but there were some pressing issues that could not be resolved to bring it to fulfill key requirements. One reason why we could not continue with this approach is that CodeDeploy would create a new, untracked Auto Scaling Group, for which changes could not be seen or controlled by Terraform. Thus, Terraform could not maintain that ASG with the original infrastructure. A second and more predominant drawback, was that CodeDeploy could not be triggered to start the deployment from Terraform itself, violating another key requirement. A final, third problem was that the code to be deployed has to be placed remotely on GitHub or in an AWS S3 Bucket. It could not be drawn from code in a local location such as from Pasons in-house systems. For these three reasons, we did not pursue the use of AWS CodeDeploy as part of an acceptable solution to zero-downtime Blue-Green deployment.

6.2 AWS Lambda

AWS Lambda is an AWS service that provides a way for code hosted on the AWS cloud to be run. Terraform can provision AWS Lambda functions which in turn could be used to perform Blue-Green deployments. Lambda was used in conjunction with AWS SNS that was used to communicate with AWS Lambda. The Lambda for triggering and controlling the deployment is developed in Python and communicates with the AWS infrastructure via the AWS SDK. It is capable of providing minute control over the AWS resources, allowing us to manipulate them to ensure zero-downtime of the infrastructure during a deployment. It is necessary to authorize the Lambda to make such changes by giving it the necessary permissions from AWS IAM policies and roles. The Figure 7 demonstrate how this was achieved.

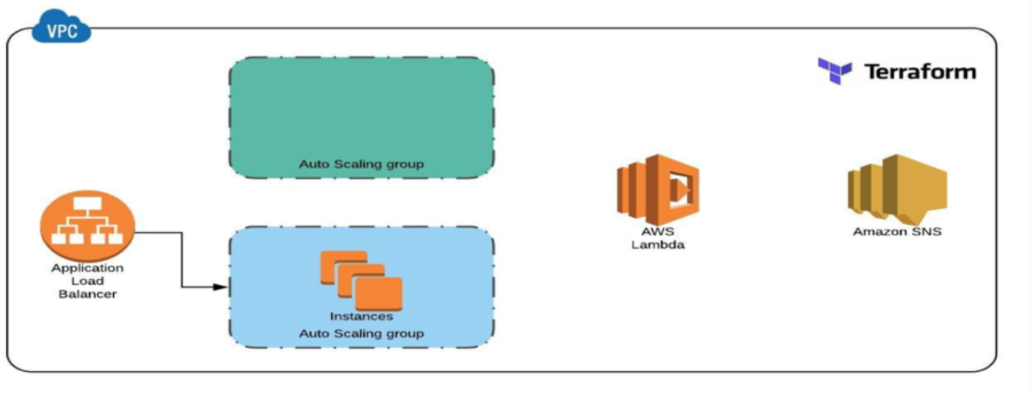


Figure 7: Lambda Deployment Initial Stages

First, AWS Lambda, AWS SNS, and all needed AWS resources needed to perform the blue-green deployment were provisioned on AWS by Terraform. This included two instead of one Auto Scaling Group (ASG) deployedone with minimum, maximum and desired instances greater than zero and the other with zero instances. Lambda and SNS are used to trigger/switch from the blue ASG to green. The architecture between Lambda and SNS mirrors an Observer architecture design pattern, in which the Lambda acts as the observer and SNS as the subject.
Sequence of steps:

1. Lambda subscribes to SNS topic

2. A message is published to SNS that triggers Lambda which runs the python code inside the Lambda function

3. The function load-matches from blue to green Auto Scaling Groups by creating same number of instances running in the blue deployment in the green Auto Scaling Group.

4. Health checks are performed on each instance in the green Auto Scaling Group as they are provisioned. The Lambda checks the status of the instances health in two ways. Firstly, instance the status check must be in running mode. Secondly, the system status check for network traffic must pass as okay, as well as checks for the disc (EBS volume) and CPU utilization, before the any of the instances created in the green ASG can be verified as health by the Lambda.

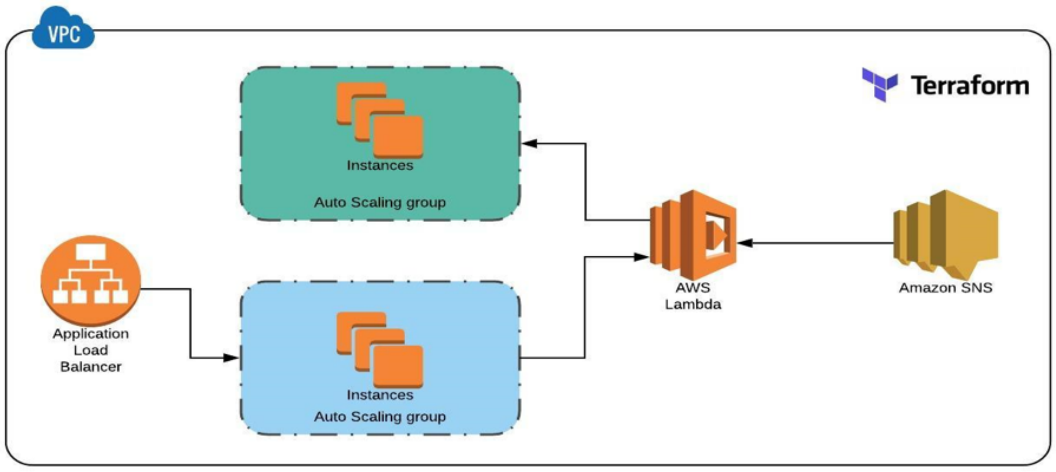


Figure 8: Lambda Deployment during deployment

5. if any of the instances are detected as unhealthy by the Lambda, the function will roll back to blue deployment and drain any instances in the green. This fulfills the condition of rollback automatically.
6. Alternatively, if all the instances deployed into the green ASG pass as healthy, the Lambda detaches the blue ASG from the load balancer.
7. While step 6 is ongoing, the instances in the blue ASG will start to drain. The transition between the blue to green deployment is so seamless that there is zero down time of the system.

The seamless action is guaranteed by making sure that the green ASG is up and running before the draining of the instances in the blue ASG.

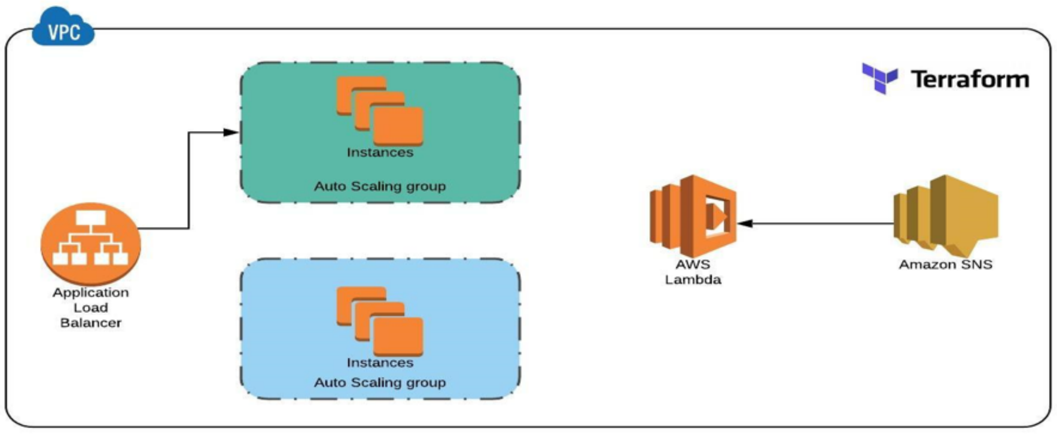


Figure 9: Lambda Deployment during draining

Pros:

1. AWS SDK gives great control of managing and provisioning AWS resources making blue green deployment easy to deploy with lambda function.
2. Load matching is easily done with this configuration.
3. Instance health check is recursively and thoroughly done on all instances.
4. Detaching the blue ASG and attaching the green ASG is also seamless.
5. SNS can notify via SMS or Email with the status of a deployment on whether it fails or passes.

Cons:

1. Terraform is not able to track the states of the resources as soon as Lambda takes over and provisions a deployment.
2. Lambda cannot be triggered from within Terraform to start the deployment. One would need AWS CLI (Command Line Interface) to publish a message in order to trigger Lambda or this can be done from the AWS Console in the browser.

With excellent control, instance health checking, and ability to automatically roll back in a failure, AWS Lambda was one of our better solutions. However, because it fails to deliver on provided Terraform initiation and Terraform control of infrastructure deployed by the Lambda, it was not an adequate solution based on Pasons requirements.

6.3 AWS Cloud Formation

AWS CloudFormation is an AWS service that is essentially an AWS version of Terraform, in that it provisions AWS infrastructure based on the infrastructure defined as code. CloudFormation stack (segments of CloudFormation code) can be defined and provisioned within Terraform. Once deployed, these CloudFormation stacks will automatically run to deploy whatever resources defined in that stack.

The whole motivation to use CloudFormation stacks instead of directly provisioning with Terraform was that CloudFormation has access to certain parameters in the definition of the AWS resources that Terraform does not. It is said that these parameters can ensure that when the replacement for that part of the infrastructure is deployed on AWS, it will redeploy in a zero-downtime fashion.

To try this, we attempted to provision most of the infrastructure using Terraform, but provisioned the ASG using CloudFormation, with settings that should allow deployments to take place in a blue-green manner with zero-downtime, when its instances switched. However, this immediately led to a problem. In order for CloudFormation to verify the deployment, it had to control more than just the ASG. Otherwise it would fail the deployment. Terraform would have to allow more of the infrastructure that it was provisioning and able to maintain, to be provisioned by CloudFormation. This was already a problem because by giving up control, Terraform could no longer track and maintain those resources, which was a fundamental requirement from Pason. Thus, the solution of CloudFormation was similarly not viable.

6.4 AWS CLI and Terraform

This was the first solution that was a Terraform-based system involved just one ASG. The differences that were added to achieve a zero downtime deployment were as follows:

- The name of the current active (blue) ASG was stored in a text file. This name was need and retrieved during the deployment process as described below.
- **Create_before_destroy** must be set to **false** for Auto Scaling Policy resource. This ensures that Terraform would destroy it before it began the rest of the deployment which would ensure that it does not interfere with the deployment. It was found that the Auto Scaling Policy might have been

scaling down on the new ASG, preventing it to reach its required initial capacity to ensure load matching.

- **Create_before_destroy** must be set to **true** for three resources that will be replaced: ASG, ASG attachment, and the Launch Configuration. They would be thus created first and pass health checks before the old versions are destroyed.
- A local exec provisioner is added on the ASG attachment. When the ASG is almost done creating and has almost registered with the Target Group, it runs a bash file that reads the name of the old (blue) ASG from the text file. It uses this name in an AWS CLI command to zero the old ASG and thus beginning the draining of the old ASG. The provisioner then has a *sleep* that will cause it to hold up Terraform for the max length of draining set for the instances, allowing them to drain before any further steps are taken.
- The draining complete, the old (blue) ASG, ASG attachment, and Launch Configuration are then destroyed by Terraform and a new for Auto Scaling Policy is created for the new ASG.

With these sequence of steps, the deployment was indeed found to ensure zero downtime of the hosted web service.

The benefits of this method were that first, being a Terraform-based solution, all the infrastructure deployed would be entirely owned and tracked by Terraform. Secondly, the zero down-time deployment was achieved within a regular *terraform apply* command, with the details transparent to the user. Thirdly, the infrastructure remained relatively simple and unchanged with the exception of the one local-exec provisioner and one bash file, as well as certain changes to meta-arguments of certain resources.

However, there was one critical issue with this solution that prevented it from fulfilling all of Pasons requirements. This was the need for to use an AWS CLI command embedded in the bash file. This would mean that any system running Terraform would have to also have AWS CLI installed which not the versatility Pason would prefer. It also meant that the procedure was not 100% controlled by Terraform. Another inconvenience was the need of having to persist the text file with the name of the old ASG name from the previous deployment. Thus, while zero-down time initiated from within Terraform was achieved, the use of AWS CLI meant the deployment was not 100% Terraform controlled, and thus it could not be accepted as the ideal solution.

7 References

Amazon. [Online]. Available: https://docs.aws.amazon.com/. [Accessed: 22-Jun-2019].

Classless Inter-Domain Routing,Wikipedia, 07-Jun-2019. [Online]. Available: https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing. [Accessed: 22-Jun-2019].

Documentation,Terraform by HashiCorp. [Online].
Available: https://www.terraform.io/docs/index.html. [Accessed: 22-Jun-2019].

Docs.aws.amazon.com. (2019). Configure Security Groups for Your Classic Load Balancer - Elastic Load Balancing. [online]
Available at: https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-security-groups.html [Accessed 12 Sep. 2019].

Udemy Ryan Kroonenberg Solution Architect Associate [Online]. Available: ”https://www.udemy.com/user/ryankroonenburg/” [Accessed: 20-May-2019]

Personal Collaboration:

Dr. Tony Schellenberg, Systems Engineer, Pason Systems Corporation