**Internship Report for Summer Internship at Electromech**

**Phase 1: Study of AWS Foundational Services**

* **Network**:

**VPC**-Amazon Virtual Private Cloud (Amazon VPC) enables you to launch AWS resources into a virtual network that you have defined. This virtual network closely resembles a traditional network that you would operate in your own data centre, with the benefits of using the scalable infrastructure of AWS.

**VPC endpoints**-A VPC endpoint enables you to privately connect your VPC to supported AWS services and VPC endpoint services powered by AWS PrivateLink without requiring an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection.

* **Storage in AWS:**

**Object storage**-Object storage provides massively scalable, cost-effective storage to store any type of data in its native format. With AWS object storage solutions like Amazon Simple Storage Service (Amazon S3) and Amazon Glacier, you manage your storage in one place with an easy-to-use application interface.

* **Compute:**

**EC2**-Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides secure, resizable compute capacity in the cloud.

**Containers**-You choose if you want to manage servers or not. You choose AWS Fargate if you want serverless compute for containers and Amazon EC2 if you need control over the installation, configuration, and management of your compute environment. Second, you choose which container orchestrator to use : Amazon Elastic Container Service (ECS) or Amazon Elastic Kubernetes Service (EKS).

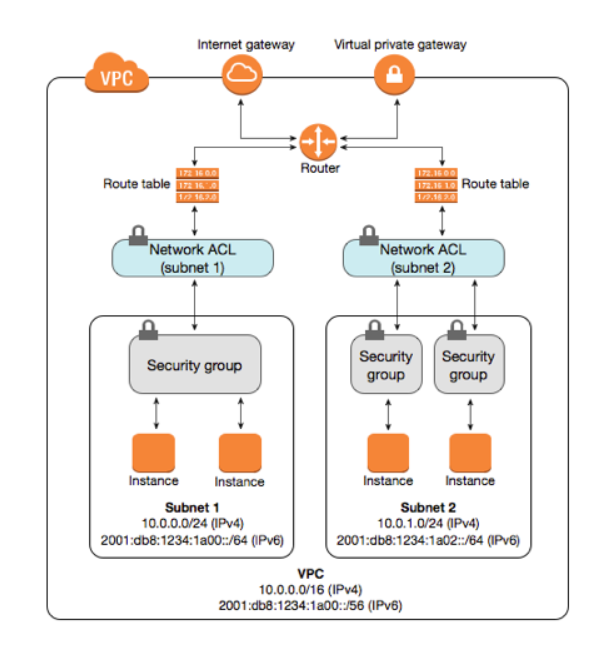
**Lambda-**AWS Lambda lets you run code without provisioning or managing servers. You pay only for the compute time you consume.

* **Security in AWS:**

**IAM-**AWS Identity and Access Management (IAM) enables you to manage access to AWS services and resources securely. Using IAM, you can create and manage AWS users and groups, and use permissions to allow and deny their access to AWS resources.

**Security Group-**AWS Security Groups act like a firewall for your Amazon EC2 instances controlling both inbound and outbound traffic. When you launch an instance on Amazon EC2, you need to assign it to a particular security group.

**NAC-**Network Access control list works similar to security group but at different level.

See figure to find difference in their both working. 

**Data at rest and Data in transit-** Security configurations offer settings to enable security for data in-transit and data at-rest in Amazon Elastic Block Store (Amazon EBS) volumes and EMRFS on Amazon S3. You protect your data in transit by defining your requirements and implementing controls, including encryption, reduces the risk of unauthorized access or exposure. You protect your data at rest by defining your requirements and implementing controls, including encryption, to reduce the risk of unauthorized access or loss.

* **Programmatic access with API:**

python or node.js

**Phase 2: AWS Automation using Terraform**

**Prerequisite Topics:**

* **Cloudwatch -** Amazon CloudWatch is a monitoring and observability service built for DevOps engineers, developers, site reliability engineers (SREs), and IT managers. CloudWatch collects monitoring and operational data in the form of logs, metrics, and events, providing you with a unified view of AWS resources, applications, and services that run on AWS and on-premises servers.
* **Cloudtrail -** AWS CloudTrail is a service that enables governance, compliance, operational auditing, and risk auditing of your AWS account. With CloudTrail, you can log, continuously monitor, and retain account activity related to actions across your AWS infrastructure.
* **AWS Config -** AWS config is compliance rule engine. AWS Config is a service that enables you to assess, audit, and evaluate the configurations of your AWS resources.
* **Security HUB -** AWS Security Hub gives you a comprehensive view of your high-priority security alerts and security posture across your AWS accounts. When enabled, Security Hub will begin aggregating and prioritizing findings and conducting security checks.
* **Terraform -** Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions. Terraform Use Infrastructure as Code to provision and manage any cloud, infrastructure, or service

**Task 1: Create 80% cpu usage alarm using Terraform.**

**Prerequisites**

Make sure you have the following software and services installed and configured:

* Terraform
* AWS CLI

Make sure you have a public and private key created at the following paths (we’ll use these to SSH to our instances):

* Private key: ~/.ssh/id\_rsa
* Public key: ~/.ssh/id\_rsa.pub

And lastly, all of the commands in this blog post assume you’re using a Mac or Linux. These commands will probably work on Windows, but they’ll probably require some modifications.

**Step 1: Set up Terraform**

We need to setup Terraform before we can create anything. By default, it doesn’t know anything about our AWS account, and it stores state locally. Let’s fix that.

First, let’s initialize a new Terraform project:

mkdir cloudwatch-demo

cd cloudwatch-demo

terraform init

Now let’s create a bucket to store Terraform state. I’ll name it terraform-artifacts-bucket for this tutorial, but you’ll have to pick something unique, since AWS requires globally unique names for buckets.

aws s3api create-bucket --acl private --bucket terraform-artifacts-bucket

Terraform recommends enabling bucket versioning, so that in case of a failure we can recover. Let’s do that as well:

aws s3api put-bucket-versioning --bucket terraform-artifacts-bucket --versioning-configuration Status=Enabled

Great. Now we can use the bucket for storing Terraform artifacts. Create a file named terraform.tf in the root of your project, and write the following in it:

terraform {

backend "s3" {

bucket = "terraform-artifacts-bucket"

key = "cloudwatch-demo/terraform.tfstate"

region = "us-east-1"

}

}

Now run the following command to initialize the backend:

terraform init

It should create a .terraform folder in the root of your project.

Now that we have a backend configured, let’s configure our project to use our AWS user. Add the following to terraform.tf:

provider "aws" {

access\_key = "${var.access\_key}"

secret\_key = "${var.secret\_key}"

region = "${var.region}"

}

This tells Terraform to use the access key and secret key from our local project variables. We’ll have to define those, since they don’t exist yet.

Create a file named variables.tf with the following contents:

variable "access\_key" {}

variable "secret\_key" {}

variable "region" {

default = "us-east-1"

}

variable "alarms\_email" {}

Tell Terraform what values to use by creating a file terraform.tfvars  with the following contents:

access\_key = "your-aws-access-key-here"

secret\_key = "your-aws-secret-key-here"

alarms\_email = "your@email.here"

We’ll have to tell Terraform to initialize the aws provider by running the following command:

terraform init

Lastly, just in case you’re storing this project in Git (you should be!), let’s tell Git to ignore our sensitive Terraform files by creating a file named . gitignore  with the following contents:

\*\*/.terraform/\*

\*.tfstate

\*.tfstate.\*

crash.log

\*.tfvars

Make sure you do not check any API keys into your repository! For simplicity, we’ve stored sensitive keys in a . tfvars  file. Terraform recommends storing them in environment variables.

**Step 2: Create an application server**

Monitoring isn’t very useful without a real server to monitor, so let’s use Terraform to create one. I won’t go detail about how this works, since it’s not the point of this post. Create a file named application.tf with the following contents:

resource "aws\_key\_pair" "ssh" {

key\_name = "default"

public\_key = "${file("~/.ssh/id\_rsa.pub")}"

}

resource "aws\_security\_group" "web" {

name = "webserver"

description = "Public HTTP + SSH"

ingress {

from\_port = 22

to\_port = 22

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

from\_port = 80

to\_port = 80

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

egress {

from\_port = 0

to\_port = 65535

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

resource "aws\_instance" "web" {

ami = "ami-2757f631"

instance\_type = "t2.micro"

key\_name = "${aws\_key\_pair.ssh.id}"

vpc\_security\_group\_ids = [ "${aws\_security\_group.web.id}" ]

provisioner "remote-exec" {

connection {

type = "ssh"

user = "ubuntu"

private\_key = "${file("~/.ssh/id\_rsa")}"

timeout = "5m"

agent = true

}

inline = [

"sudo apt-get update -y && apt-get upgrade -y",

"sudo apt-get install nginx -y"

]

}

}

output "web\_public\_dns" {

value = "${aws\_instance.web.public\_dns}"

}

Create the instance by running the following command:

terraform apply

Lastly, test that it’s working:

open "http://$(terraform output web\_public\_dns)"

A browser should open to your new server running a default Nginx service.

**Step 3: Set up CloudWatch alarms**

Now that we’ve set everything up, we can actually setup some alarms. We’ll do this by setting up a topic and a subscription within Amazon Simple Notification Service (SNS), and then tying a CloudWatch alarm to that topic.

SNS is essentially an event system within Amazon. Topics are used for listening to events within the Amazon ecosystem, and subscriptions tie topics to actual endpoints – an email, a web server, etc. We’ll create a topic for the alarms to fire, and then we’ll create a subscription to send you an email.

Create a new file named alarms.tf with the following contents:

resource "aws\_sns\_topic" "alarm" {

name = "alarms-topic"

delivery\_policy = <<EOF

{

"http": {

"defaultHealthyRetryPolicy": {

"minDelayTarget": 20,

"maxDelayTarget": 20,

"numRetries": 3,

"numMaxDelayRetries": 0,

"numNoDelayRetries": 0,

"numMinDelayRetries": 0,

"backoffFunction": "linear"

},

"disableSubscriptionOverrides": false,

"defaultThrottlePolicy": {

"maxReceivesPerSecond": 1

}

}

}

EOF

provisioner "local-exec" {

command = "aws sns subscribe --topic-arn ${self.arn} --protocol email --notification-endpoint ${var.alarms\_email}"

}

}

Terraform doesn’t allow creating email subscriptions, so we have to use a provisioner instead. This code runs on your local machine, and uses your local AWS CLI to create the email subscription. Once you run this, you’ll have to open your email and confirm the subscription creation.

Now that we have the subscription working, we can setup some alarms. Let’s start with two basic alarms:

* CPU usage
* Health check failures

Add the following to alarms.tf:

resource "aws\_cloudwatch\_metric\_alarm" "cpu" {

alarm\_name = "web-cpu-alarm"

comparison\_operator = "GreaterThanOrEqualToThreshold"

evaluation\_periods = "2"

metric\_name = "CPUUtilization"

namespace = "AWS/EC2"

period = "120"

statistic = "Average"

threshold = "80"

alarm\_description = "This metric monitors ec2 cpu utilization"

alarm\_actions = [ "${aws\_sns\_topic.alarm.arn}" ]

dimensions {

InstanceId = "${aws\_instance.web.id}"

}

}

resource "aws\_cloudwatch\_metric\_alarm" "health" {

alarm\_name = "web-health-alarm"

comparison\_operator = "GreaterThanOrEqualToThreshold"

evaluation\_periods = "1"

metric\_name = "StatusCheckFailed"

namespace = "AWS/EC2"

period = "120"

statistic = "Average"

threshold = "1"

alarm\_description = "This metric monitors ec2 health status"

alarm\_actions = [ "${aws\_sns\_topic.alarm.arn}" ]

dimensions {

InstanceId = "${aws\_instance.web.id}"

}

}

Run:

terraform apply

And that’s it!

**Step 4: Testing it out**

Let’s make sure this works. SSH into your instance:

ssh ubuntu@$(terraform output web\_public\_dns)

Now we’ll force the CPU to spike:

yes > /dev/null

Let that run for ten minutes. You should receive an email that looks something like this:

You are receiving this email because your Amazon CloudWatch Alarm “web-cpu-alarm” in the US East (N. Virginia) region has entered the ALARM state, because “Threshold Crossed: 1 datapoint [99.8064516129032 (01/07/18 03:56:00)] was greater than or equal to the threshold (80.0).”

Press “Control+C” to stop destroying your instance.

Well done: you have alarms!

**Step 5: Clean up**

All of the instances and resources we created cost money. Make sure to destroy all of them by running the following command:

terraform destroy

**Task 2: Create Billing Alerts using terraform at 100$**

**We are going to create three resources in our Terraform template. The first will be an account level billing alarm just in case you don’t already have one. Then we will create two budget resources for our EC2 and RDS forecasted spend.**

**Our account level billing alarm:**

provider "aws" {

region = "us-east-1"

}

resource "aws\_cloudwatch\_metric\_alarm" "account-billing-alarm" {

alarm\_name = "account-billing-alarm"

comparison\_operator = "GreaterThanOrEqualToThreshold"

evaluation\_periods = "1"

metric\_name = "EstimatedCharges"

namespace = "AWS/Billing"

period = "21600"

statistic = "Average"

threshold = "100"

alarm\_description = "Billing alarm by account"

alarm\_actions = ["<your-sns-topic-arn-for-notification>"]

dimensions {

Currency = "USD"

LinkedAccount = "<your-aws-account-id>"

}

}

**Once our template is initialized we can apply our template to our AWS account.**

$ terraform apply

An execution plan has been generated and is shown below.

Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

+ aws\_cloudwatch\_metric\_alarm.account-billing-alarm

....

....

aws\_cloudwatch\_metric\_alarm.account-billing-alarm: Creating...

actions\_enabled: "" => "true"

alarm\_actions.#: "" => "1"

alarm\_actions.321893454: "" => "<your-sns-topic-arn-for-notification>"

alarm\_description: "" => "Billing alarm by account"

alarm\_name: "" => "account-billing-alarm"

arn: "" => "<computed>"

comparison\_operator: "" => "GreaterThanOrEqualToThreshold"

dimensions.%: "" => "2"

dimensions.Currency: "" => "USD"

dimensions.LinkedAccount: "" => "<your-aws-account-id>"

evaluate\_low\_sample\_count\_percentiles: "" => "<computed>"

evaluation\_periods: "" => "1"

metric\_name: "" => "EstimatedCharges"

namespace: "" => "AWS/Billing"

period: "" => "21600"

statistic: "" => "Average"

threshold: "" => "100"

treat\_missing\_data: "" => "missing"

aws\_cloudwatch\_metric\_alarm.account-billing-alarm: Creation complete after 1s (ID: account-billing-alarm)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

**We now have our account level billing alarm to notify us when our estimated bill is going to be greater than $100 USD. Let’s create our budgets so we can alarm when our forecasted spend on EC2 or RDS exceeds our expectation.**

**Add the following resources to your Terraform template.**

resource "aws\_budgets\_budget" "ec2-forecast-alarm" {

name = "budget-ec2-monthly"

budget\_type = "COST"

limit\_amount = "50"

limit\_unit = "USD"

time\_period\_start = "2020-01-01\_00:00"

time\_unit = "MONTHLY"

cost\_filters {

service = "Amazon Elastic Compute Cloud - Compute"

}

}

resource "aws\_budgets\_budget" "rds-forecast-alarm" {

name = "budget-rds-monthly"

budget\_type = "COST"

limit\_amount = "40"

limit\_unit = "USD"

time\_period\_start = "2020-01-01\_00:00"

time\_unit = "MONTHLY"

cost\_filters {

service = "Amazon Relational Database Service"

}

}

**Here we have our budgets for EC2 and RDS.**

**The cost\_filters are using the fully qualified names for each service, this is what AWS budgets via Terraform expect. We fire “apply” command again to create our new budgets.**

$ terraform apply

An execution plan has been generated and is shown below.

Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

+ aws\_budgets\_budget.ec2-forecast-alarm

id: <computed>

account\_id: <computed>

budget\_type: "COST"

cost\_filters.%: "1"

cost\_filters.Service: "ec2"

cost\_types.#: <computed>

limit\_amount: "50"

limit\_unit: "USD"

name: "budget-ec2-monthly"

name\_prefix: <computed>

time\_period\_end: "2089-06-15\_00:00"

time\_period\_start: "2020-01-01\_00:00"

time\_unit: "MONTHLY"

+ aws\_budgets\_budget.rds-forecast-alarm

id: <computed>

account\_id: <computed>

budget\_type: "COST"

cost\_filters.%: "1"

cost\_filters.Service: "rds"

cost\_types.#: <computed>

limit\_amount: "40"

limit\_unit: "USD"

name: "budget-rds-monthly"

name\_prefix: <computed>

time\_period\_end: "2089-06-15\_00:00"

time\_period\_start: "2020-01-01\_00:00"

time\_unit: "MONTHLY"

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

**We now have two budgets configured in our AWS account. The first tracks our spend on EC2 and the latter tracks our spend on RDS. However, we do not have notifications set up for these yet.**

**Adding notifications to our budgets:**

Because of the limitation of Terraform as of today, we need to add the notifications to our budgets outside of our template. We can quickly add notifications to each of our budgets by using the AWS CLI.

$ aws budgets create-notification --account-id <your-aws-account-id> --budget-name budget-rds-monthly --notification NotificationType=FORECASTED,ComparisonOperator=GREATER\_THAN,Threshold=100,ThresholdType=PERCENTAGE --subscribers SubscriptionType=EMAIL,Address=<your-email-address>

$ aws budgets create-notification --account-id <your-aws-account-id> --budget-name budget-ec2-monthly --notification NotificationType=FORECASTED,ComparisonOperator=GREATER\_THAN,Threshold=100,ThresholdType=PERCENTAGE --subscribers SubscriptionType=EMAIL,Address=<your-email-address>

**Task 3: Create Cloudconfig rules to find any security group is rule is open i.e. (0.0.0.0/0) other than 80/443 ports.**

- name: high-risk-security-groups

resource: security-group

description: |

Find any rule from a security group that allows 0.0.0.0/0 or ::/0 (IPv6) ingress

and notify the user.

mode:

type: cloudtrail

role: arn:aws:iam::987659872123:role/cloudcustodian

events:

- source: ec2.amazonaws.com

event: AuthorizeSecurityGroupIngress

ids: "requestParameters.groupId"

- source: ec2.amazonaws.com

event: RevokeSecurityGroupIngress

ids: "requestParameters.groupId"

filters:

- and:

- type: security-group

key: GroupId

#key: SecurityGroups[].GroupID

op: not-in

value:

- sg-0db5e1ab7s8323

- or:

- type: ingress

OnlyPorts: [80, 443]

Cidr:

value: "0.0.0.0/0"

op: in

- type: ingress

OnlyPorts: [80, 443]

CidrV6:

value: "::/0"

op: in

actions:

- type:

ingress: matched