**Set up Alarm at 80% of cpu utilization**

**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 named 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