**STEPS:**

Key components used in this setup

1. [A Ruby on rails application](https://guides.rubyonrails.org/getting_started.html)
2. [Docker](https://www.docker.com/)
3. [Terraform](https://www.terraform.io/)

**First we will start with a basic rails\_terraform ROR application and dockerize it with Docker**

Create new Rails application using the following command

$ rails new rails\_terraform

Lets add a route to homepage for this application

In config/routes.rb add the following

get ‘/’, :to => ‘home#index’

In app/controllers folder create home\_controller.rb and add the following code

class HomeController < ApplicationController  
 def index;end  
end

In app/views, create a folder name home and add a index.html.erb file with the following code.

<p>  
 Hi, Rails app is running successfully <br>  
 <br>  
 AWS Infrastructure for this app is spin using terraform, docker and AWS codebuild<br><br>  
</p>

Running Rails s and navigate to [http://localhost:3000](http://localhost:3000/) to verify that the app is running successfully rendering index page.

**Dockerize the rails\_terraform Rails application**

Since we created a rails 6 application it while creating the docker image we will have to do the following steps in Dockerfile.

1. Fetch Ruby image as base for ruby version used in application.
2. Install all the packages needed to run the rails application in container like binutils, curl, git, gnupg, cmake, python, python-dev, postgresql-client, supervisor, tar, tzdata etc.
3. Install stable version of yarn(without cmdtest) from the yarnpkg.
4. Create application directory in container.
5. Copy application code into application directory in docker container.
6. Set work directory to application directory in docker container.
7. Run bundler install.
8. Run yarn install.
9. Run rake assets precompile bundle command.
10. Expose port 3000 of docker container.
11. Start rails server bind the server to 0.0.0.0

Following is the Dockerfile for the Rails rails\_terraform application

FROM ruby:2.5.3ENV PATH /root/.yarn/bin:$PATHARG build\_without  
ARG rails\_env  
RUN apt-get update -qq && apt-get install -y binutils curl git gnupg cmake python python-dev postgresql-client supervisor tar tzdata  
RUN apt-get install -y apt-transport-https apt-utils  
RUN curl -sL <https://deb.nodesource.com/setup_10.x> | bash - && apt-get install -y nodejs  
RUN curl -sS <https://dl.yarnpkg.com/debian/pubkey.gpg> | apt-key add -  
RUN echo "deb <https://dl.yarnpkg.com/debian/> stable main" | tee /etc/apt/sources.list.d/yarn.list  
RUN apt-get update && apt-get install -y yarn  
RUN mkdir /rails\_terraform\_docker  
COPY . /rails\_terraform\_docker  
WORKDIR /rails\_terraform\_dockerRUN gem install bundler -v 2.0.2RUN bundle install  
RUN yarn installRUN RAILS\_ENV=production NODE\_ENV=production SECRET\_KEY\_BASE=not\_set OLD\_AWS\_SECRET\_ACCESS\_KEY=not\_set OLD\_AWS\_ACCESS\_KEY\_ID=not\_set bundle exec rake assets:precompile# Add a script to be executed every time the container starts.  
COPY entrypoint.sh /usr/bin/  
RUN chmod +x /usr/bin/entrypoint.sh  
ENTRYPOINT ["entrypoint.sh"]  
EXPOSE 3000# Start the main process.  
CMD ["rails", "server", "-b", "0.0.0.0"]

Run the following command in the application root directory on your local machine to verify that Dockerfile is able to successfully build image for the application

$ docker image build -t rails\_terraform:latest .

After this command success run the following command to list all available docker images on your machine which should also list the docker image we just build for our application.

$ docker image list

This should list the image rails\_terraform with tag latest.

**Setting up terraform to build AWS infrastructure and application deployment**

First we need local installation of terraform

**Networking**

First we will to create a VPC with will have 2 subnets (1 public and 1 private) in each Availability Zone. We will create our application in us-east-1 region with two availability zones(us-east-1a, us-east-1b). Each Availability Zone is a geographically isolated region. Keeping our resources in more than one zone is the first thing to achieve high availability. If one physical zone fails for some reason, your application can answer from the others.

resource "aws\_vpc" "vpc" {  
 cidr\_block = “${var.vpc\_cidr}”  
 enable\_dns\_hostnames = true  
 enable\_dns\_support = truetags = {  
 Name = "${var.environment}-vpc"  
 Environment = "${var.environment}"  
 }  
}

Now we will add aws\_internet\_gateway and aws\_nat\_gateway. An internet gateway is a horizontally scaled, redundant, and highly available VPC component that allows communication between instances in your VPC and the internet.

To enable access to or from the internet for instances in a VPC subnet, you must do the following:

* Attach an internet gateway to your VPC.
* Ensure that your subnet’s route table points to the internet gateway.
* Ensure that instances in your subnet have a globally unique IP address (public IPv4 address, Elastic IP address, or IPv6 address).
* Ensure that your network access control and security group rules allow the relevant traffic to flow to and from your instance.

/\* Internet gateway for the public subnet \*/  
resource "aws\_internet\_gateway" "ig" {  
 vpc\_id = "${aws\_vpc.vpc.id}" tags = {  
 Name = "${var.environment}-igw"  
 Environment = "${var.environment}"  
 }  
}/\* Elastic IP for NAT \*/  
resource "aws\_eip" "nat\_eip" {  
 vpc = true  
 depends\_on = ["aws\_internet\_gateway.ig"]  
}/\* NAT \*/  
resource "aws\_nat\_gateway" "nat" {  
 allocation\_id = "${aws\_eip.nat\_eip.id}"  
 subnet\_id = "${element(aws\_subnet.public\_subnet.\*.id, 0)}"  
 depends\_on = [“aws\_internet\_gateway.ig”] tags = {  
 Name = "${var.environment}-${element(var.availability\_zones, 1)}- nat"  
 Environment = "{var.environment}"  
 }  
}/\* Public subnet \*/  
resource "aws\_subnet" "public\_subnet" {  
 vpc\_id = "${aws\_vpc.vpc.id}"  
 count = "${length(var.public\_subnets\_cidr)}"  
 cidr\_block = "${element(var.public\_subnets\_cidr, count.index)}"  
 availability\_zone = "${element(var.availability\_zones, count.index)}"  
 map\_public\_ip\_on\_launch = true tags = {  
 Name = "${var.environment}-${element(var.availability\_zones, count.index)}-public-subnet"  
 Environment = "${var.environment}"  
 }  
}/\* Private subnet \*/  
resource "aws\_subnet" "private\_subnet" {  
 vpc\_id = "${aws\_vpc.vpc.id}"  
 count = "${length(var.private\_subnets\_cidr)}"  
 cidr\_block = "${element(var.private\_subnets\_cidr, count.index)}"  
 availability\_zone = "${element(var.availability\_zones, count.index)}"  
 map\_public\_ip\_on\_launch = false tags = {  
 Name = "${var.environment}-${element(var.availability\_zones, count.index)}-private-subnet"  
 Environment = "${var.environment}"  
 }  
}/\* Routing table for private subnet \*/  
resource "aws\_route\_table" "private" {  
 vpc\_id = "${aws\_vpc.vpc.id}" tags = {  
 Name = "${var.environment}-private-route-table"  
 Environment = "${var.environment}"  
 }  
}/\* Routing table for public subnet \*/  
resource "aws\_route\_table" "public" {  
 vpc\_id = "${aws\_vpc.vpc.id}" tags = {  
 Name = "${var.environment}-public-route-table"  
 Environment = "${var.environment}"  
 }  
 }resource "aws\_route" "public\_internet\_gateway" {  
 route\_table\_id = "${aws\_route\_table.public.id}"  
 destination\_cidr\_block = "0.0.0.0/0"  
 gateway\_id = "${aws\_internet\_gateway.ig.id}"  
}resource "aws\_route" "private\_nat\_gateway" {  
 route\_table\_id = "${aws\_route\_table.private.id}"  
 destination\_cidr\_block = "0.0.0.0/0"  
 nat\_gateway\_id = "${aws\_nat\_gateway.nat.id}"  
}/\* Route table associations \*/  
resource "aws\_route\_table\_association" "public" {  
 count = "${length(var.public\_subnets\_cidr)}"  
 subnet\_id = "${element(aws\_subnet.public\_subnet.\*.id, count.index)}"  
 route\_table\_id = "${aws\_route\_table.public.id}"  
}resource "aws\_route\_table\_association" "private" {  
 count = "${length(var.private\_subnets\_cidr)}"  
 subnet\_id = "${element(aws\_subnet.private\_subnet.\*.id, count.index)}"  
 route\_table\_id = "${aws\_route\_table.private.id}"  
}/\*====  
VPC’s Default Security Group  
======\*/  
resource "aws\_security\_group" "default" {  
 name = "${var.environment}-default-sg"  
 description = "Default security group to allow inbound/outbound from the VPC"  
 vpc\_id = "${aws\_vpc.vpc.id}"  
 depends\_on = ["aws\_vpc.vpc"] ingress {  
 from\_port = "0"  
 to\_port = "0"  
 protocol = "-1"  
 self = true  
 } egress {  
 from\_port = "0"  
 to\_port = "0"  
 protocol = "-1"  
 self = “true”  
 }tags = {  
 Environment = "${var.environment}"  
 }  
}

To enable communication over the internet for IPv4, your instance must have a public IPv4 address or an Elastic IP address that’s associated with a private IPv4 address on your instance. Your instance is only aware of the private (internal) IP address space defined within the VPC and subnet. The internet gateway logically provides the one-to-one NAT on behalf of your instance, so that when traffic leaves your VPC subnet and goes to the internet, the reply address field is set to the public IPv4 address or Elastic IP address of your instance, and not its private IP address. Conversely, traffic that’s destined for the public IPv4 address or Elastic IP address of your instance has its destination address translated into the instance’s private IPv4 address before the traffic is delivered to the VPC.

**Database**

We will create a RDS database. It will be located on the private subnet. Allowing only the public subnet to access it.

Also we will create a db\_access\_sg which will enable access of ecs application to rds instance.

/\* subnet used by rds \*/  
resource "aws\_db\_subnet\_group" "rds\_subnet\_group" {  
 name = "${var.environment}-rds-subnet-group"  
 description = "RDS subnet group"  
 subnet\_ids = "${var.subnet\_ids}"  
 tags = {  
 Environment = "${var.environment}"  
 }  
}/\* Security Group for resources that want to access the Database \*/  
resource "aws\_security\_group" "db\_access\_sg" {  
 vpc\_id = "${var.vpc\_id}"  
 name = "${var.environment}-db-access-sg"  
 description = "Allow access to RDS" tags = {  
 Name = "${var.environment}-db-access-sg"  
 Environment = "${var.environment}"  
 }  
}resource "aws\_security\_group" "rds\_sg" {  
 name = "${var.environment}-rds-sg"  
 description = "${var.environment} Security Group"  
 vpc\_id = "${var.vpc\_id}"  
 tags = {  
 Name = "${var.environment}-rds-sg"  
 Environment = "${var.environment}"  
 }// allows traffic from the SG itself  
 ingress {  
 from\_port = 0  
 to\_port = 0  
 protocol = "-1"  
 self = true  
 }//allow traffic for TCP 5432  
 ingress {  
 from\_port = 5432  
 to\_port = 5432  
 protocol = "tcp"  
 security\_groups = ["${aws\_security\_group.db\_access\_sg.id}"]  
 }// outbound internet access  
 egress {  
 from\_port = 0  
 to\_port = 0  
 protocol = "-1"  
 cidr\_blocks = ["0.0.0.0/0"]  
 }  
}resource "aws\_db\_instance" "rds" {  
 identifier = "${var.environment}-database"  
 allocated\_storage = "${var.allocated\_storage}"  
 engine = "postgres"  
 engine\_version = "9.6.6"  
 instance\_class = "${var.instance\_class}"  
 multi\_az = "${var.multi\_az}"  
 name = "${var.database\_name}"  
 username = "${var.database\_username}"  
 password = "${var.database\_password}"  
 db\_subnet\_group\_name = "${aws\_db\_subnet\_group.rds\_subnet\_group.id}"  
 vpc\_security\_group\_ids = ["${aws\_security\_group.rds\_sg.id}"]  
 skip\_final\_snapshot = true  
 #snapshot\_identifier = "rds-${var.environment}-snapshot"  
 tags = {  
 Environment = "${var.environment}"  
 }  
}

As you can see aws\_security\_group.db\_access\_sg\_id is added to ingress inbound rules to allow access to database with db\_access\_sg inside our VPC. Later this security group will be added in network configuration security groups.

**ECS**

Now lets create our ECS service using terraform.First we create the repository to store our built images.

resource "aws\_ecr\_repository" "rails\_terraform\_app" {  
 name = "${var.repository\_name}"  
}

Next, we need our ECS cluster. Even using Fargate (that doesn’t need any EC2), we need to define a cluster for the application.

resource "aws\_ecs\_cluster" "cluster" {  
 name = "${var.environment}-ecs-cluster"  
}

Now, we will define 2 task definitions.

* Web: Contains the definition of the web app itself.
* Db Migrate: This task will only run the command to migrate our database and will die. Since it is a single run task, we don’t need a service for it.

/\* the task definition for the web service \*/  
data "template\_file" "web\_task" {  
 template = "${file("${path.module}/tasks/web\_task\_definition.json")}"vars = {  
 image = "${aws\_ecr\_repository.rails\_terraform\_app.repository\_url}"  
 secret\_key\_base = "${var.secret\_key\_base}"  
 database\_url = "postgresql://${var.database\_username}:${var.database\_password}@${var.database\_endpoint}:5432/${var.database\_name}?encoding=utf8&pool=40"  
 log\_group = "${aws\_cloudwatch\_log\_group.rails\_terraform.name}"  
 }  
}resource "aws\_ecs\_task\_definition" "web" {  
 family = "${var.environment}\_web"  
 container\_definitions = "${data.template\_file.web\_task.rendered}"  
 requires\_compatibilities = ["FARGATE"]  
 network\_mode = "awsvpc"  
 cpu = "256"  
 memory = "512"  
 execution\_role\_arn = "${aws\_iam\_role.ecs\_execution\_role.arn}"  
 task\_role\_arn = "${aws\_iam\_role.ecs\_execution\_role.arn}"  
}/\* the task definition for the db migration \*/  
data "template\_file" "db\_migrate\_task" {  
 template = "${file("${path.module}/tasks/db\_migrate\_task\_definition.json")}"vars = {  
 image = "${aws\_ecr\_repository.rails\_terraform\_app.repository\_url}"  
 secret\_key\_base = "${var.secret\_key\_base}"  
 database\_url = "postgresql://${var.database\_username}:${var.database\_password}@${var.database\_endpoint}:5432/${var.database\_name}?encoding=utf8&pool=40"  
 log\_group = "rails\_terraform"  
 }  
}resource "aws\_ecs\_task\_definition" "db\_migrate" {  
 family = "${var.environment}\_db\_migrate"  
 container\_definitions = "${data.template\_file.db\_migrate\_task.rendered}"  
 requires\_compatibilities = ["FARGATE"]  
 network\_mode = "awsvpc"  
 cpu = "256"  
 memory = "512"  
 execution\_role\_arn = "${aws\_iam\_role.ecs\_execution\_role.arn}"  
 task\_role\_arn = "${aws\_iam\_role.ecs\_execution\_role.arn}"  
}

The tasks definitions are configured in a JSON file and rendered as a template in Terraform.  
This is the task definition of the web app:

[  
 {  
 “name”: “web”,  
 “image”: “${image}”,  
 “portMappings”: [  
 {  
 “containerPort”: 80,  
 “hostPort”: 80  
 }  
 ],  
 “memory”: 300,  
 “networkMode”: “awsvpc”,  
 “logConfiguration”: {  
 “logDriver”: “awslogs”,  
 “options”: {  
 “awslogs-group”: “${log\_group}”,  
 “awslogs-region”: “us-east-1”,  
 “awslogs-stream-prefix”: “web”  
 }  
 },  
 “environment”: [  
 {  
 “name”: “RAILS\_ENV”,  
 “value”: “production”  
 },  
 {  
 “name”: “DATABASE\_URL”,  
 “value”: “${database\_url}”  
 },  
 {  
 “name”: “SECRET\_KEY\_BASE”,  
 “value”: “${secret\_key\_base}”  
 },  
 {  
 “name”: “PORT”,  
 “value”: “80”  
 },  
 {  
 “name”: “RAILS\_LOG\_TO\_STDOUT”,  
 “value”: “true”  
 },  
 {  
 “name”: “RAILS\_SERVE\_STATIC\_FILES”,  
 “value”: “true”  
 }  
 ]  
 }  
]

Similarly we will have a json definition of db:migrate task which will run only once to run database migrations.

[  
 {  
 “name”: “db-migrate”,  
 “image”: “${image}”,  
 “command”: [“bundle”, “exec”, “rake”, “db:migrate”],  
 “memory”: 300,  
 “logConfiguration”: {  
 “logDriver”: “awslogs”,  
 “options”: {  
 “awslogs-group”: “${log\_group}”,  
 “awslogs-region”: “us-east-1”,  
 “awslogs-stream-prefix”: “db\_migrate”  
 }  
 },  
 “environment”: [  
 {  
 “name”: “RAILS\_ENV”,  
 “value”: “production”  
 },  
 {  
 “name”: “DATABASE\_URL”,  
 “value”: “${database\_url}”  
 },  
 {  
 “name”: “SECRET\_KEY\_BASE”,  
 “value”: “${secret\_key\_base}”  
 },  
 {  
 “name”: “RAILS\_LOG\_TO\_STDOUT”,  
 “value”: “true”  
 }  
 ]  
 }  
]

Before creating the Services, we need to create the load balancers. They will be on the public subnet and will forward the requests to the ECS service.

First we will create an alb\_target\_group in our VPCresource “aws\_alb\_target\_group” “alb\_target\_group” {  
 name = “${var.environment}-alb-target-group-${random\_id.target\_group\_sufix.hex}”  
 port = 80  
 protocol = “HTTP”  
 vpc\_id = “${var.vpc\_id}”  
 target\_type = “ip” lifecycle {  
 create\_before\_destroy = true  
 }  
}

Now we will create an security group for application load balancer which will direct traffic from port 80 to port 80 of ecs service.

resource “aws\_security\_group” “web\_inbound\_sg” {  
 name = “${var.environment}-web-inbound-sg”  
 description = “Allow HTTP from Anywhere into ALB”  
 vpc\_id = “${var.vpc\_id}” ingress {  
 from\_port = 80  
 to\_port = 80  
 protocol = “tcp”  
 cidr\_blocks = [“0.0.0.0/0”]  
 } ingress {  
 from\_port = 8  
 to\_port = 0  
 protocol = “icmp”  
 cidr\_blocks = [“0.0.0.0/0”]  
 } egress {  
 from\_port = 0  
 to\_port = 0  
 protocol = “-1”  
 cidr\_blocks = [“0.0.0.0/0”]  
 } tags = {  
 Name = “${var.environment}-web-inbound-sg”  
 }  
}

Now we create alb along with alb\_listener at port 80

resource “aws\_alb” “alb\_rails-terraform” {  
 name = “${var.environment}-alb-rails-terraform”  
 subnets = “${var.public\_subnet\_ids}”  
 security\_groups = “${concat(var.security\_groups\_ids, [aws\_security\_group.web\_inbound\_sg.id])}” tags = {  
 Name = “${var.environment}-alb-rails\_terraform”  
 Environment = “${var.environment}”  
 }  
}resource “aws\_alb\_listener” “rails\_terraform” {  
 load\_balancer\_arn = “${aws\_alb.alb\_rails-terraform.arn}”  
 port = “80”  
 protocol = “HTTP”  
 depends\_on = [“aws\_alb\_target\_group.alb\_target\_group”] default\_action {  
 target\_group\_arn = “${aws\_alb\_target\_group.alb\_target\_group.arn}”  
 type = “forward”  
 }  
}

We then create IAM service role with attached policies

data “aws\_iam\_policy\_document” “ecs\_service\_role” {  
 statement {  
 effect = “Allow”  
 actions = [“sts:AssumeRole”]  
 principals {  
 type = “Service”  
 identifiers = [“ecs.amazonaws.com”]  
 }  
 }  
}resource “aws\_iam\_role” “ecs\_role” {  
 name = “ecs\_role”  
 assume\_role\_policy = “${data.aws\_iam\_policy\_document.ecs\_service\_role.json}”  
}data “aws\_iam\_policy\_document” “ecs\_service\_policy” {  
 statement {  
 effect = “Allow”  
 resources = [“\*”]  
 actions = [  
 “elasticloadbalancing:Describe\*”,  
 “elasticloadbalancing:DeregisterInstancesFromLoadBalancer”,  
 “elasticloadbalancing:RegisterInstancesWithLoadBalancer”,  
 “ec2:Describe\*”,  
 “ec2:AuthorizeSecurityGroupIngress”  
 ]  
 }  
}/\* ecs service scheduler role \*/  
resource “aws\_iam\_role\_policy” “ecs\_service\_role\_policy” {  
 name = “ecs\_service\_role\_policy”  
 #policy = “${file(“${path.module}/policies/ecs-service-role.json”)}”  
 policy = “${data.aws\_iam\_policy\_document.ecs\_service\_policy.json}”  
 role = “${aws\_iam\_role.ecs\_role.id}”  
}/\* role that the Amazon ECS container agent and the Docker daemon can assume \*/  
resource “aws\_iam\_role” “ecs\_execution\_role” {  
 name = “ecs\_task\_execution\_role”  
 assume\_role\_policy = “${file(“${path.module}/policies/ecs-task-execution-role.json”)}”  
}resource “aws\_iam\_role\_policy” “ecs\_execution\_role\_policy” {  
 name = “ecs\_execution\_role\_policy”  
 policy = “${file(“${path.module}/policies/ecs-execution-role-policy.json”)}”  
 role = “${aws\_iam\_role.ecs\_execution\_role.id}”  
}

Finally we create ECS service

resource “aws\_security\_group” “ecs\_service” {  
 vpc\_id = “${var.vpc\_id}”  
 name = “${var.environment}-ecs-service-sg”  
 description = “Allow egress from container” egress {  
 from\_port = 0  
 to\_port = 0  
 protocol = “-1”  
 cidr\_blocks = [“0.0.0.0/0”]  
 } ingress {  
 from\_port = 8  
 to\_port = 0  
 protocol = “icmp”  
 cidr\_blocks = [“0.0.0.0/0”]  
 } tags = {  
 Name = “${var.environment}-ecs-service-sg”  
 Environment = “${var.environment}”  
 }  
}data “aws\_ecs\_task\_definition” “web” {  
 task\_definition = “${aws\_ecs\_task\_definition.web.family}”  
 depends\_on = [ “aws\_ecs\_task\_definition.web” ]  
}resource “aws\_ecs\_service” “web” {  
 name = “${var.environment}-web”  
 task\_definition = “${aws\_ecs\_task\_definition.web.family}:${max(“${aws\_ecs\_task\_definition.web.revision}”, “${data.aws\_ecs\_task\_definition.web.revision}”)}”  
 desired\_count = 2  
 launch\_type = “FARGATE”  
 cluster = “${aws\_ecs\_cluster.cluster.id}”  
 depends\_on = [“aws\_iam\_role\_policy.ecs\_service\_role\_policy”, “aws\_alb\_target\_group.alb\_target\_group”]network\_configuration {  
 security\_groups = “${concat(var.security\_groups\_ids, [aws\_security\_group.ecs\_service.id])}”  
 subnets = “${var.subnets\_ids}”  
 }load\_balancer {  
 target\_group\_arn = “${aws\_alb\_target\_group.alb\_target\_group.arn}”  
 container\_name = “web”  
 container\_port = “80”  
 }#depends\_on = [“aws\_alb\_target\_group.alb\_target\_group”]  
}

we have specified launch type as FARGATE and in network configuration we have passed security\_groups\_ids along with security group id created for ecs service. In deploy.tf file we can verify that security\_group\_ids contain the security\_group\_id of RDS and default VPM security\_group\_id.

Now lets setup autoscaling for our FARGATE application. For this we need to define aws\_appautoscaling\_policy for up and down and define cloudwatch alarm actions to triggers these policies for up and down.

resource “aws\_iam\_role” “ecs\_autoscale\_role” {  
 name = “${var.environment}\_ecs\_autoscale\_role”  
 assume\_role\_policy = “${file(“${path.module}/policies/ecs-autoscale-role.json”)}”  
}resource “aws\_iam\_role\_policy” “ecs\_autoscale\_role\_policy” {  
 name = “ecs\_autoscale\_role\_policy”  
 policy = “${file(“${path.module}/policies/ecs-autoscale-role-policy.json”)}”  
 role = “${aws\_iam\_role.ecs\_autoscale\_role.id}”  
}resource “aws\_appautoscaling\_target” “target” {  
 service\_namespace = “ecs”  
 resource\_id = “service/${aws\_ecs\_cluster.cluster.name}/${aws\_ecs\_service.web.name}”  
 scalable\_dimension = “ecs:service:DesiredCount”  
 role\_arn = “${aws\_iam\_role.ecs\_autoscale\_role.arn}”  
 min\_capacity = 2  
 max\_capacity = 4  
}resource “aws\_appautoscaling\_policy” “up” {  
 name = “${var.environment}\_scale\_up”  
 service\_namespace = “ecs”  
 resource\_id = “service/${aws\_ecs\_cluster.cluster.name}/${aws\_ecs\_service.web.name}”  
 scalable\_dimension = “ecs:service:DesiredCount” step\_scaling\_policy\_configuration {  
 adjustment\_type = “ChangeInCapacity”  
 cooldown = 60  
 metric\_aggregation\_type = “Maximum” step\_adjustment {  
 metric\_interval\_lower\_bound = 0  
 scaling\_adjustment = 1  
 }  
 }depends\_on = [“aws\_appautoscaling\_target.target”]  
}resource “aws\_appautoscaling\_policy” “down” {  
 name = “${var.environment}\_scale\_down”  
 service\_namespace = “ecs”  
 resource\_id = “service/${aws\_ecs\_cluster.cluster.name}/${aws\_ecs\_service.web.name}”  
 scalable\_dimension = “ecs:service:DesiredCount” step\_scaling\_policy\_configuration {  
 adjustment\_type = “ChangeInCapacity”  
 cooldown = 60  
 metric\_aggregation\_type = “Maximum” step\_adjustment {  
 metric\_interval\_lower\_bound = 0  
 scaling\_adjustment = -1  
 }  
 }depends\_on = [“aws\_appautoscaling\_target.target”]  
}/\* metric used for auto scale \*/  
resource “aws\_cloudwatch\_metric\_alarm” “service\_cpu\_high” {  
 alarm\_name = “${var.environment}\_rails\_terraform\_web\_cpu\_utilization\_high”  
 comparison\_operator = “GreaterThanOrEqualToThreshold”  
 evaluation\_periods = “3”  
 metric\_name = “CPUUtilization”  
 namespace = “AWS/ECS”  
 period = “60”  
 statistic = “Maximum”  
 threshold = “70” dimensions = {  
 ClusterName = “${aws\_ecs\_cluster.cluster.name}”  
 ServiceName = “${aws\_ecs\_service.web.name}”  
 } alarm\_actions = [“${aws\_appautoscaling\_policy.up.arn}”]  
 ok\_actions = [“${aws\_appautoscaling\_policy.down.arn}”]  
}

After, we create a CloudWatch metric based on the CPU. If the CPU usage is greater than 70% from 3periods, we trigger the alarm\_action that calls the scale-up policy. If it returns to the Ok state, it will trigger the scale-down policy.

**Codepipeline**

Most of modern day deployments happening today uses automated deployments and gone are the days when we manually update the code and restart the service again.

Here with terraform our infrastructure is ready and owr DOCKERFILE is also ready to create image for our application code.

Codepipeline and codebuild are two tools provided by AWS to automate deployment. We will create an AWS codepipeline which will pick our source code from github repository, build the docker image from the docker file, tag the docker image with image tag and ECS FARGATE task for our web application. Whenever we push the code to master branch it will repeat the same, sourcing the latest code, build image and launch ECS task.

For more detail on [AWS codepipeline](https://aws.amazon.com/codepipeline/) and [Codebuild](https://aws.amazon.com/codebuild/" \t "_blank).

In code\_pipeline folder in main.tf we create the aws\_codebuild\_project and aws\_codepipeline.

resource “aws\_s3\_bucket” “source” {  
 bucket = “rails-terraform-experiment-source”  
 acl = “private”  
 force\_destroy = true  
}resource “aws\_iam\_role” “codepipeline\_role” {  
 name = “codepipeline-role”assume\_role\_policy = “${file(“${path.module}/policies/codepipeline\_role.json”)}”  
}/\* policies \*/  
data “template\_file” “codepipeline\_policy” {  
 template = “${file(“${path.module}/policies/codepipeline.json”)}” vars = {  
 aws\_s3\_bucket\_arn = “${aws\_s3\_bucket.source.arn}”  
 }  
}resource “aws\_iam\_role\_policy” “codepipeline\_policy” {  
 name = “codepipeline\_policy”  
 role = “${aws\_iam\_role.codepipeline\_role.id}”  
 policy = “${data.template\_file.codepipeline\_policy.rendered}”  
}/\*  
/\* CodeBuild  
\*/  
resource “aws\_iam\_role” “codebuild\_role” {  
 name = “codebuild-role”  
 assume\_role\_policy = “${file(“${path.module}/policies/codebuild\_role.json”)}”  
}data “template\_file” “codebuild\_policy” {  
 template = “${file(“${path.module}/policies/codebuild\_policy.json”)}” vars = {  
 aws\_s3\_bucket\_arn = “${aws\_s3\_bucket.source.arn}”  
 }  
}resource “aws\_iam\_role\_policy” “codebuild\_policy” {  
 name = “codebuild-policy”  
 role = “${aws\_iam\_role.codebuild\_role.id}”  
 policy = “${data.template\_file.codebuild\_policy.rendered}”  
}data “template\_file” “buildspec” {  
 template = “${file(“${path.module}/buildspec.yml”)}” vars = {  
 repository\_url = “${var.repository\_url}”  
 region = “${var.region}”  
 cluster\_name = “${var.ecs\_cluster\_name}”  
 subnet\_id = “${var.run\_task\_subnet\_id}”  
 security\_group\_ids = “${join(“,”, var.run\_task\_security\_group\_ids)}”  
 }  
}resource “aws\_codebuild\_project” “rails\_terraform\_build” {  
 name = “rails\_terraform-codebuild”  
 build\_timeout = “10”  
 service\_role = “${aws\_iam\_role.codebuild\_role.arn}” artifacts {  
 type = “CODEPIPELINE”  
 }environment {  
 compute\_type = “BUILD\_GENERAL1\_SMALL”  
 // <https://docs.aws.amazon.com/codebuild/latest/userguide/build-env-ref-available.html>  
 image = “aws/codebuild/docker:1.12.1”  
 type = “LINUX\_CONTAINER”  
 privileged\_mode = true  
 }source {  
 type = “CODEPIPELINE”  
 buildspec = “${data.template\_file.buildspec.rendered}”  
 }  
}/\* CodePipeline \*/resource “aws\_codepipeline” “pipeline” {  
 name = “rails\_terraform-pipeline”  
 role\_arn = “${aws\_iam\_role.codepipeline\_role.arn}” artifact\_store {  
 location = “${aws\_s3\_bucket.source.bucket}”  
 type = “S3”  
 } stage {  
 name = “Source” action {  
 name = “Source”  
 category = “Source”  
 owner = “ThirdParty”  
 provider = “GitHub”  
 version = “1”  
 output\_artifacts = [“source”] configuration = {  
 Owner = “owner\_of\_repo”  
 Repo = “rails\_terraform”  
 Branch = “master”  
 OAuthToken = “\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*”  
 }  
 }  
 }stage {  
 name = “Build” action {  
 name = “Build”  
 category = “Build”  
 owner = “AWS”  
 provider = “CodeBuild”  
 version = “1”  
 input\_artifacts = [“source”]  
 output\_artifacts = [“imagedefinitions”] configuration = {  
 ProjectName = “rails\_terraform-codebuild”  
 }  
 }  
 }stage {  
 name = “Production” action {  
 name = “Deploy”  
 category = “Deploy”  
 owner = “AWS”  
 provider = “ECS”  
 input\_artifacts = [“imagedefinitions”]  
 version = “1” configuration = {  
 ClusterName = “${var.ecs\_cluster\_name}”  
 ServiceName = “${var.ecs\_service\_name}”  
 FileName = “imagedefinitions.json”  
 }  
 }  
 }  
}

In the above code, we create a CodeBuild project, using the following buildspec (build specifications file):

version: 0.2

phases:  
 pre\_build:  
 commands:  
 — pip install awscli — upgrade — user  
 — echo `aws — version`  
 — echo Logging in to Amazon ECR…  
 — $(aws ecr get-login — region ${region} — no-include-email)  
 — REPOSITORY\_URI=${repository\_url}  
 — IMAGE\_TAG=$(echo $CODEBUILD\_RESOLVED\_SOURCE\_VERSION | cut -c 1–7)  
 — echo Entered the pre\_build phase…  
 build:  
 commands:  
 — echo Build started on `date`  
 — echo Building the Docker image…  
 — docker build — build-arg build\_without=”development test” — build-arg rails\_env=”production” -t $REPOSITORY\_URI:latest .  
 — docker tag $REPOSITORY\_URI:latest $REPOSITORY\_URI:$IMAGE\_TAG  
 post\_build:  
 commands:  
 — echo Build completed on `date`  
 — echo Pushing the Docker images…  
 — docker push $REPOSITORY\_URI:latest  
 — docker push $REPOSITORY\_URI:$IMAGE\_TAG  
 — echo Writing image definitions file…  
 — printf ‘[{“name”:”web”,”imageUri”:”%s”}]’ $REPOSITORY\_URI:$IMAGE\_TAG > imagedefinitions.json  
 — echo upgrading db-migrate task definitions  
 — aws ecs run-task — launch-type FARGATE — cluster ${cluster\_name} — task-definition production\_db\_migrate — network-configuration “awsvpcConfiguration={subnets=[${subnet\_id}],securityGroups=[${security\_group\_ids}]}”  
artifacts:  
 files: imagedefinitions.json

We use Github as the CodePipeline source provider, you need to generate a token to access the repositories and replace that in code\_pipeline resource along with repository owner, repository name.

Also you will need to replace aws\_access\_key and secret of your account in deploy.tf file in aws\_provider.

Now, we need to import the modules and the provider library

After this you can apply the terraform to AWS to spin up the application infrastructure.

$ terraform apply --var-file=staging.tfvars

ou can keep environment specific variables values in production.tfvars and staging.tfvars and pass that file as variable in — var-file while applying terraform.

Above command will success with providing url pointing to application load balancer for the application and boom our application is deployed