Published in Geek Culture

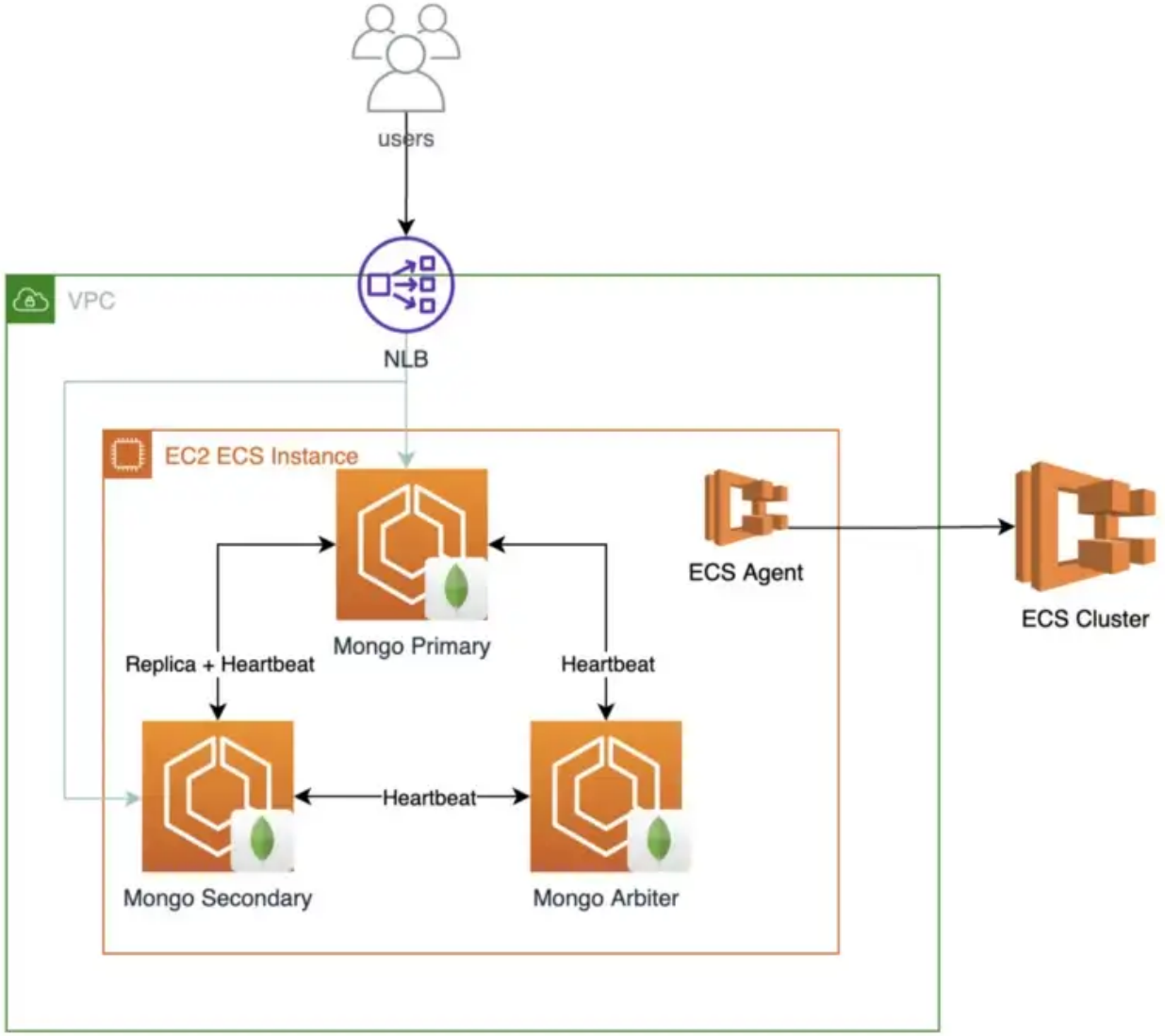
Jazz Tong Follow

May 14, 2021 · 8 min read · Listen

# MongoDB Replica Set in AWS ECS with Terraform

Today we shall go through the solution of using ECS to set up MongoDB replica Set. Although it is possible to set up MongoDB in ECS using the Fargate launch type, we will follow the best practices in ECS for persistent workload using the EC2 launch type. To allow reproducible artifacts for this solution, we will use Terraform for all the configurations. You are welcomed to reuse this template and modify it accordingly.

## High-level diagram




As illustrated in the diagram above, we will use EC2 as compute resource for our ECS Cluster. For simplicity's sake, only use 1 EC2 to run 3 MongoDB nodes, but you can add multiple EC2s to maximize the redundancy. Here is what we will build:-

1. 1 EC2 using ECS optimized image for ECS Instance
2. 3 MongoDB Services using Bitnami Image
3. 1 ECS Cluster
4. 1 NLB to route traffic to Primary and Secondary MongoDB service

## Why Bitnami MongoDB Image

To set up a replica set using docker, you are required to configure the replica set configuration and add nodes accordingly. MongoDB official image does not provide an out-of-the-box auto replica configuration, hence we will use the Bitnami MongoDB image, which allows users to configure the replica set using an environment variable.

## Prerequisite knowledge

The solution shared in this article requires an understanding of the following knowledge:-

1. Docker container
2. AWS IAM, EC2, Network load balancer, ECS Service and Task definition
3. Terraform scripting
4. MongoDB configuration

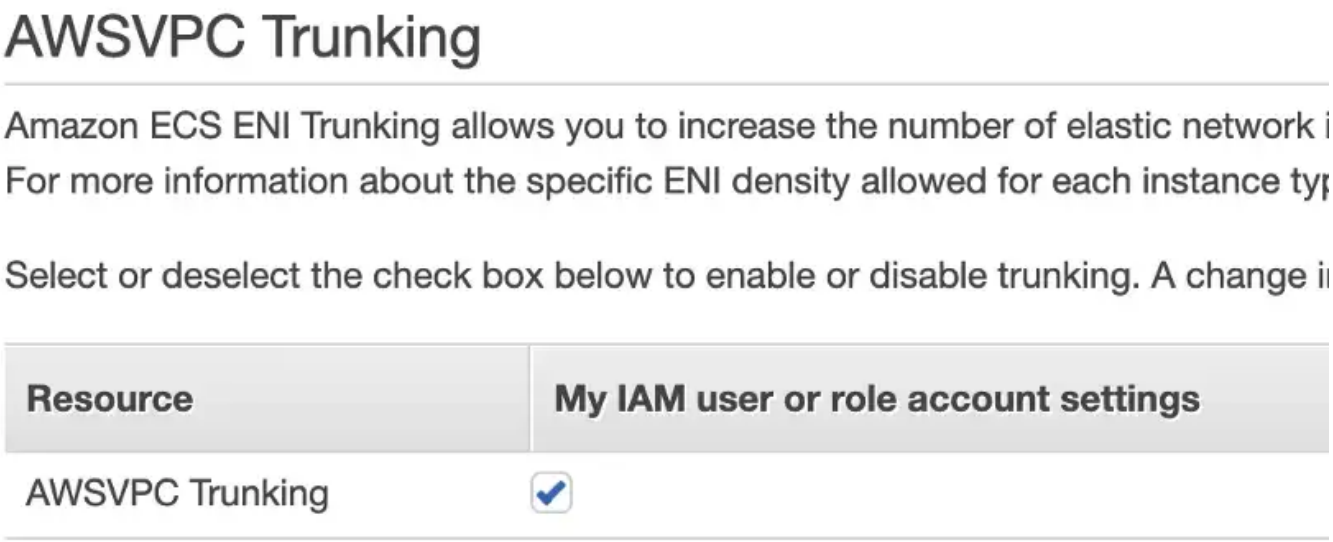
## Prerequisite Setup

You need the following tools to follow this solution:-

1. Visual Studio Code
2. AWS CLI Install
3. AWS Session Manager Plugin
4. MongoDB Compass
5. Terraform v0.15+

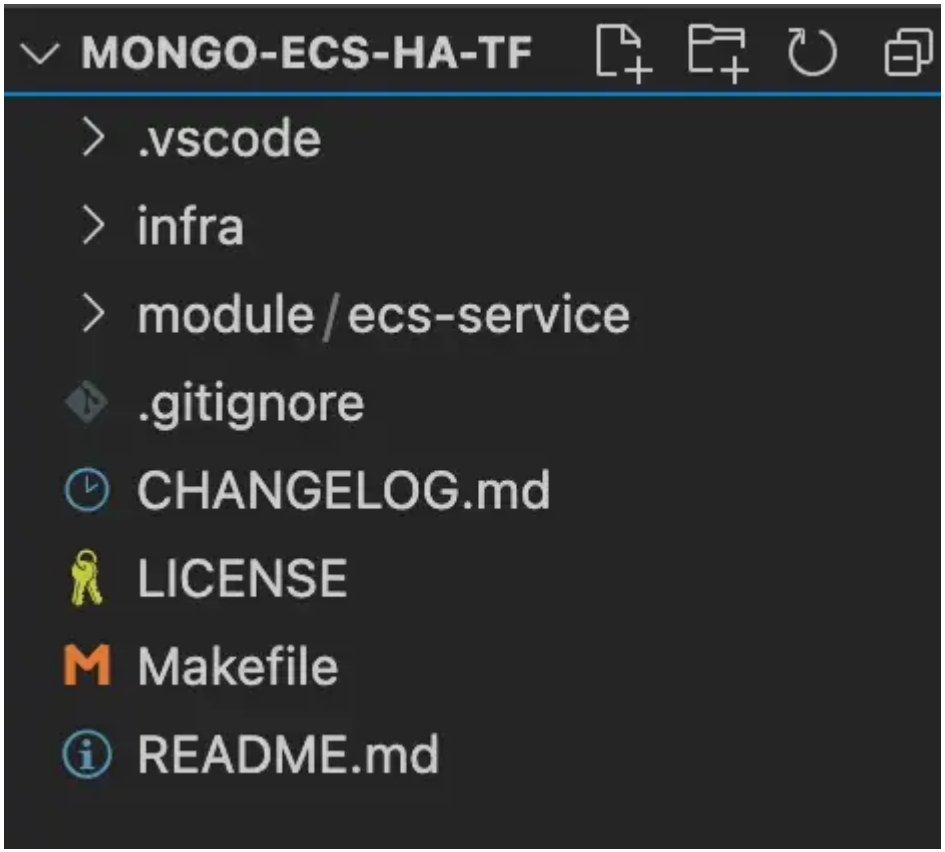
## VPC Trucking

In this solution, we will use `awsvpc` network mode for all the MongoDB Service. Refer here to the detailed `awsvpc` network mode. `awsvpc` consumes network interface or Elastic Network Interface (ENI) of each EC2, and there is a limited ENI for each EC2 type. VPC trucking is a new feature that allows us to launch twice as much task using `awsvpc` network mode with EC2 launch type. But this is the opt-in option, you need to enable this by following this configuration, or enable in account level setting in ECS console as below:-



## Initialize project

As the project is comprehensive to go through step by step, we will use the finished project for the explanation. Check out the source here and configure AWS credentials. The project structure looks like this:-



## ECS Service module

Inside the project, there is one module that allows users to provision ECS service with a predefined setting. We will go through the important resource for the module.

### aws_ecs_service resource

This is the `ECS` service resource that provision ECS service. We configure it to use `EC2` launch type by default and allows control of `load_balancer` settings. As MongoDB node cannot run more than one task at a time, we configure `deployment_maximun_percent` it to 100. As a result, any new deployment will immediately stop the existing task.

```
resource "aws_ecs_service" "this" {
  name            = var.name
  cluster         = var.cluster
  launch_type     = "EC2"

  task_definition      = aws_ecs_task_definition.this.arn
  force_new_deployment = false
  desired_count        = var.desired_count

  dynamic "load_balancer" {
    for_each = var.create_lb ? aws_lb_target_group.this : []
    content {
      target_group_arn = load_balancer.value.arn
      container_name   = var.name
      container_port   = var.containerPort
    }
  }

  network_configuration {
    subnets          = var.subnets
    security_groups  = var.security_groups
    assign_public_ip = false
  }

  service_registries {
    registry_arn = aws_service_discovery_service.this.arn
  }

  # Service only run one instance at a time
  deployment_maximum_percent         = 100
  deployment_minimum_healthy_percent = 0
}
```

module_ecs-service_service.tf hosted with ❤ by GitHub view raw

### aws_ecs_task_definition resource

We need task definition to control how our container run in ECS. The following resource allows dynamic volume configuration, where it is an important part to persist MongoDB data. To manage the volume using docker volume, we add `docker_volume_configuration` section to allow us to configure local or other volume drivers. The `placementConstraints` configuration also provides to allow us to control where our container/task should provision in which EC2 instance. You can create 3 different EC2 with different labels and ensure they launch in respective AZ and EC2.

```
resource "aws_ecs_task_definition" "this" {
  family                   = var.name
  memory                   = var.memory
  requires_compatibilities = ["EC2"]
  task_role_arn            = var.task_role_arn
  execution_role_arn       = var.execution_role_arn
  network_mode             = "awsvpc"


  dynamic "volume" {
    for_each = var.volumes == null ? [] : var.volumes
    content {
      name      = volume.value.name
      host_path = try(volume.value.host_path, null)
      dynamic "docker_volume_configuration" {
        for_each = can(volume.value.docker_volume_configuration) ? [try(volume.value.docker_volu
        content {
          scope         = try(docker_volume_configuration.value.scope, null)
          autoprovision = try(docker_volume_configuration.value.autoprovision, null)
          driver        = try(docker_volume_configuration.value.driver, null)
        }
      }
    }
  }

  container_definitions = jsonencode(
    [{
      "name" : var.name
      "image" : var.image,
      "portMappings" : [
        { containerPort = var.containerPort }
      ],
      "environment" : var.environment,
      "mountPoints" : var.mountPoints
```

```
      "placementConstraints" : var.placementConstraints,
      "volumes" : var.volumes,
      "logConfiguration" : {
        "logDriver" : "awslogs",
        "options" : {
          "awslogs-create-group" : "true",
          "awslogs-group" : "/ecs/${var.name}",
          "awslogs-region" : data.aws_region.this.name,
          "awslogs-stream-prefix" : "ecs"
        }
      },
    }]
  )
}
```

ecs_module_task_def.tf hosted with ❤ by GitHub view raw

```
#! /bin/bash
set -e
# Ouput all log
exec > >(tee /var/log/user-data.log|logger -t user-data -s 2>/dev/console) 2>&1

# Set ECS agent setting
cat <<'EOF' >> /etc/ecs/ecs.config
ECS_CLUSTER=${ECS_CLUSTER}
ECS_INSTANCE_ATTRIBUTES={"mongo": "primary"}
ECS_CONTAINER_STOP_TIMEOUT=2s
ECS_IMAGE_PULL_BEHAVIOR=prefer-cached
EOF

# Enable password authentication
sed 's/PasswordAuthentication no/PasswordAuthentication yes/' -i /etc/ssh/sshd_config
systemctl restart sshd
service sshd restart

# Add ecs-user
useradd ecs-user
usermod -aG wheel ecs-user
echo "${ECS_USER_PASSWORD}" | passwd --stdin ecs-user

# Report end
echo 'Done Initialization'
```

ecs_ha_tf_user_data.tmpl.sh hosted with ❤ by GitHub view raw

## mongo_primary resource

We will create a MongoDB node using ecs-service module. In this Mongo Primary node, we configure `Network Load Balancer` , Bitnami Mongo replica setting using an environment variable, volume and mount point to persist the data. Refer here for Bitnami Mongo Image configuration.

> `desired_count` *must set to 1 for successful provision,* `placementConstraints` *must also be configured to match EC2 label*

```
module "mongo_primary" {
  source = "../module/ecs-service"

  cluster       = aws_ecs_cluster.this.id
  name          = "${var.app_id}-primary"
  image         = var.image
  containerPort = 27017

  create_lb     = var.nlb_enabled
  lb_arn        = var.nlb_enabled ? aws_lb.this[0].arn : ""
  listener_port = 27017

  desired_count = var.primary_enabled ? 1 : 0
  memory        = var.memory
  environment = [
    { "name" : "MONGODB_ROOT_PASSWORD", "value" : var.mongo_password },
    { "name" : "MONGODB_ADVERTISED_HOSTNAME", "value" : "mongo-ecs-primary.ecs.demo" },
    { "name" : "MONGODB_REPLICA_SET_MODE", "value" : "primary" },
    { "name" : "MONGODB_REPLICA_SET_KEY", "value" : "replicasetkey123" }
  ]
  volumes = [
    {
      "name" : "primary-data",
      "docker_volume_configuration" : {
        "scope" : "shared",
        "autoprovision" : true,
        "driver" : "local"
      }
    }
  ]
  mountPoints = [
    {
      "containerPath" : "/bitnami/mongodb",
      "sourceVolume" : "primary-data"
```

```
      "sourceVolume" : "primary-data"
    }
  ]
  placementConstraints = [
    {
      "expression" : "attribute:mongo == primary",
      "type" : "memberOf"
    }
  ]
  discovery_namespace_id = aws_service_discovery_private_dns_namespace.this.id
  security_groups        = [aws_security_group.this.id]
  subnets                = data.aws_subnet_ids.this.ids
  task_role_arn          = aws_iam_role.task.arn
  execution_role_arn     = aws_iam_role.task.arn

  depends_on = [
    aws_lb.this
  ]
}
```

mongo_ecs_ha_mongo-primary.tf hosted with ❤ by GitHub view raw

```
module "mongo_secondary" {
  source = "../module/ecs-service"

  cluster       = aws_ecs_cluster.this.id
  name          = "${var.app_id}-secondary"
  image         = var.image
  containerPort = 27017

  create_lb     = var.nlb_enabled
  lb_arn        = var.nlb_enabled ? aws_lb.this[0].arn : ""
  listener_port = 27018

  desired_count = var.secondary_enabled ? 1 : 0
  memory        = var.memory
  environment = [
    { "name" : "MONGODB_ADVERTISED_HOSTNAME", "value" : "mongo-ecs-secondary.ecs.demo" },
    { "name" : "MONGODB_REPLICA_SET_MODE", "value" : "secondary" },
    { "name" : "MONGODB_INITIAL_PRIMARY_HOST", "value" : "mongo-ecs-primary.ecs.demo" },
    { "name" : "MONGODB_INITIAL_PRIMARY_ROOT_PASSWORD", "value" : var.mongo_password },
    { "name" : "MONGODB_REPLICA_SET_KEY", "value" : "replicasetkey123" }
  ]
  placementConstraints = [
    {
      "expression" : "attribute:mongo == primary",
      "type" : "memberOf"
    }
  ]
  discovery_namespace_id = aws_service_discovery_private_dns_namespace.this.id
  security_groups        = [aws_security_group.this.id]
  subnets                = data.aws_subnet_ids.this.ids
  task_role_arn          = aws_iam_role.task.arn
  execution_role_arn     = aws_iam_role.task.arn

  depends_on = [
```

```
module "mongo_arbiter" {
  source = "../module/ecs-service"

  cluster       = aws_ecs_cluster.this.id
  name          = "${var.app_id}-arbiter"
  image         = var.image
  containerPort = 27017

  desired_count = var.arbiter_enabled ? 1 : 0
  memory        = var.memory
  environment = [
    { "name" : "MONGODB_ADVERTISED_HOSTNAME", "value" : "mongo-ecs-arbiter.ecs.demo" },
    { "name" : "MONGODB_REPLICA_SET_MODE", "value" : "arbiter" },
    { "name" : "MONGODB_INITIAL_PRIMARY_HOST", "value" : "mongo-ecs-primary.ecs.demo" },
    { "name" : "MONGODB_INITIAL_PRIMARY_ROOT_PASSWORD", "value" : var.mongo_password },
    { "name" : "MONGODB_REPLICA_SET_KEY", "value" : "replicasetkey123" }
  ]
  placementConstraints = [
    {
      "expression" : "attribute:mongo == primary",
      "type" : "memberOf"
    }
  ]
  discovery_namespace_id = aws_service_discovery_private_dns_namespace.this.id
  security_groups        = [aws_security_group.this.id]
  subnets                = data.aws_subnet_ids.this.ids
  task_role_arn          = aws_iam_role.task.arn
  execution_role_arn     = aws_iam_role.task.arn

  depends_on = [
    aws_lb.this
  ]
}
```

mongo_ecs_ha_mongo-arbiter.tf hosted with ❤ by GitHub view raw

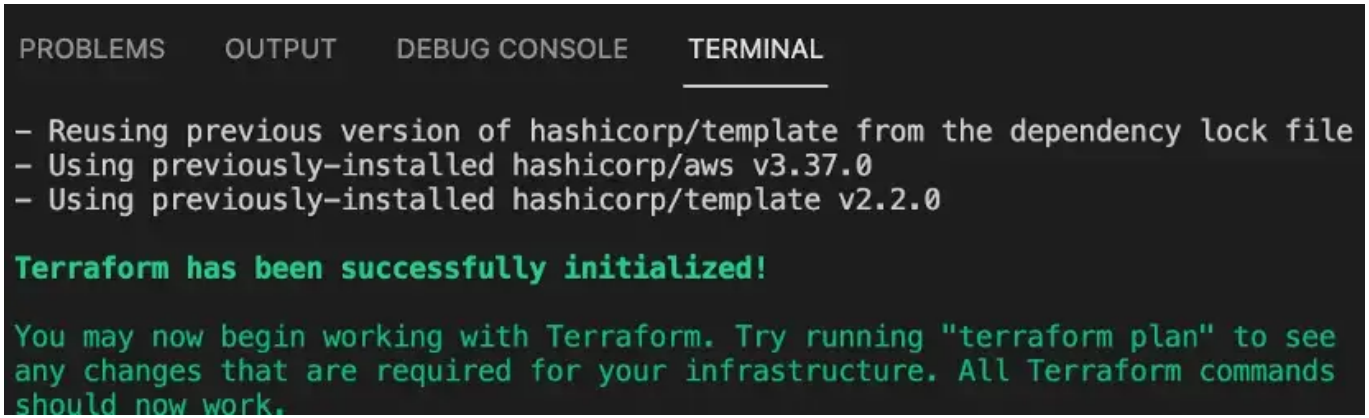
## Start provision the solution

We already walk through some of the important setups in this solution, let's start provision the solution. Open the checkout source in Visual Studio Code.

> *We will use Makefile for all the configuration. You can use the native script in the Makefile to run if your machine does not support Makefile*
>
> *Make sure you configure aws credential before start the provisioning*

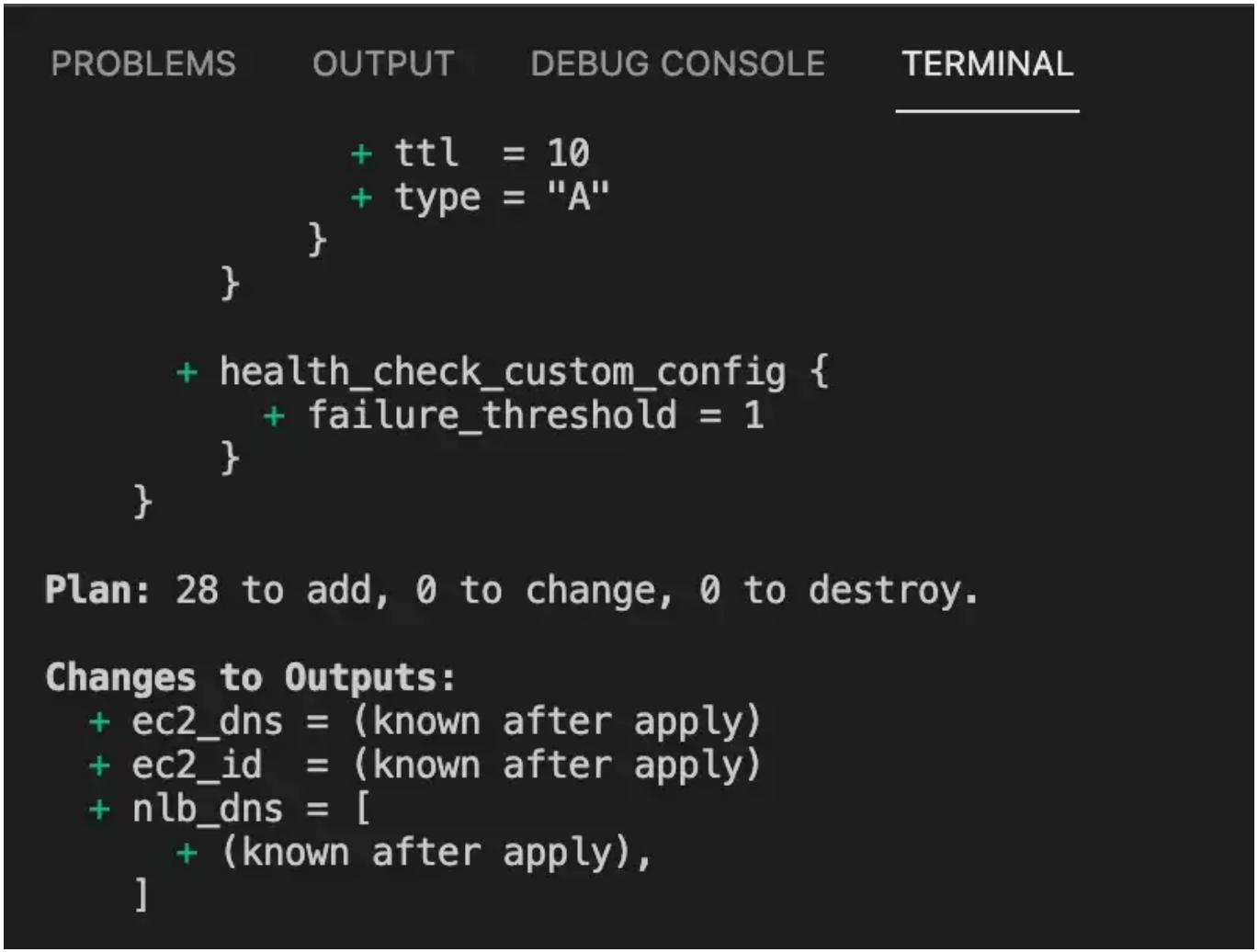
### terraform init

Run `make init` in the project root to initialize the Terraform project.



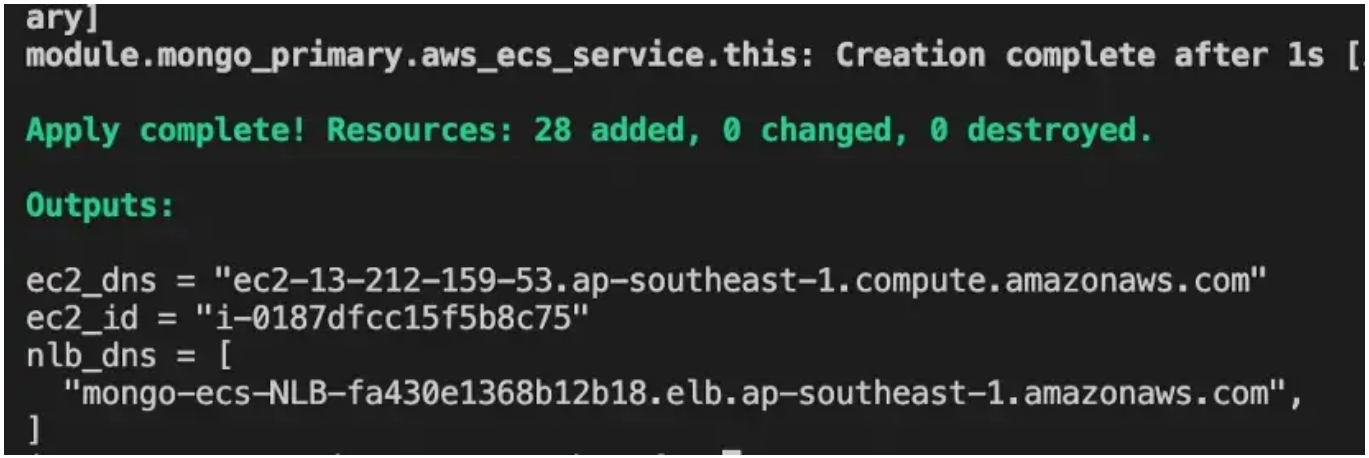
### terraform plan

Run `make plan` in the project root to verify the Terraform plan.




### terraform apply

After Terraform plan success, we can provision it into our environment using terraform plan. Run `make apply` to provision the Terraform script.




## Explore your result

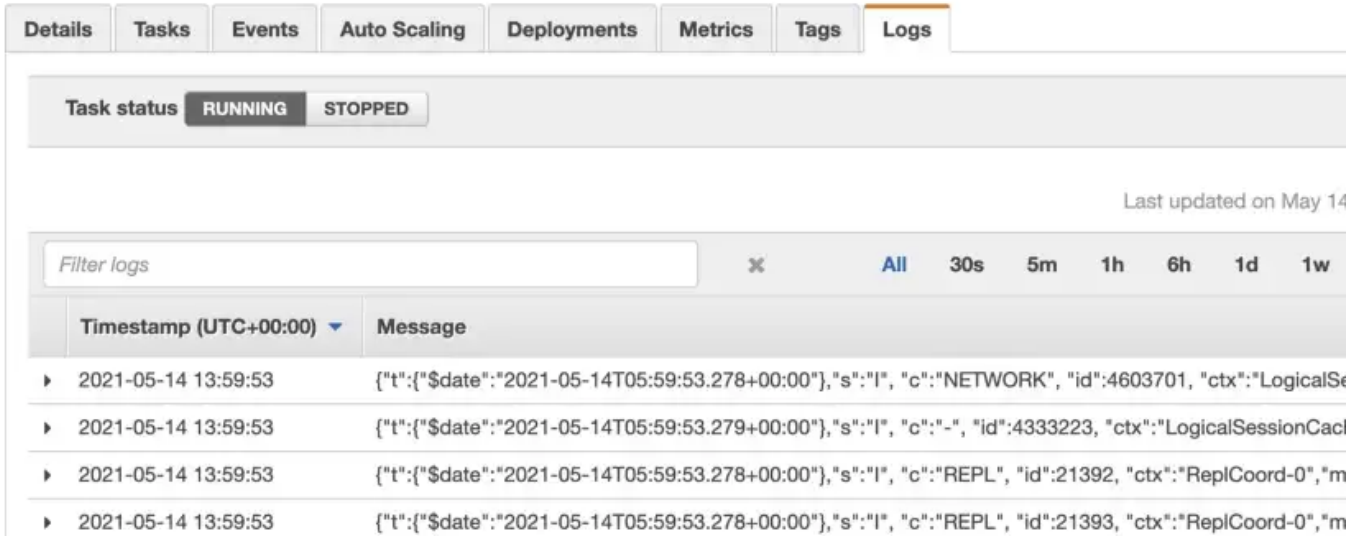
Great, you follow me. At this point, you should successfully provision the entire replica set into your account. Let's check your AWS environment to verify the result.

## Explore your Cluster

Open your ECS home, and you should found `mongo-ecs-cluster` provision with 3 Services, 3 Tasks, and 1 Container instances.

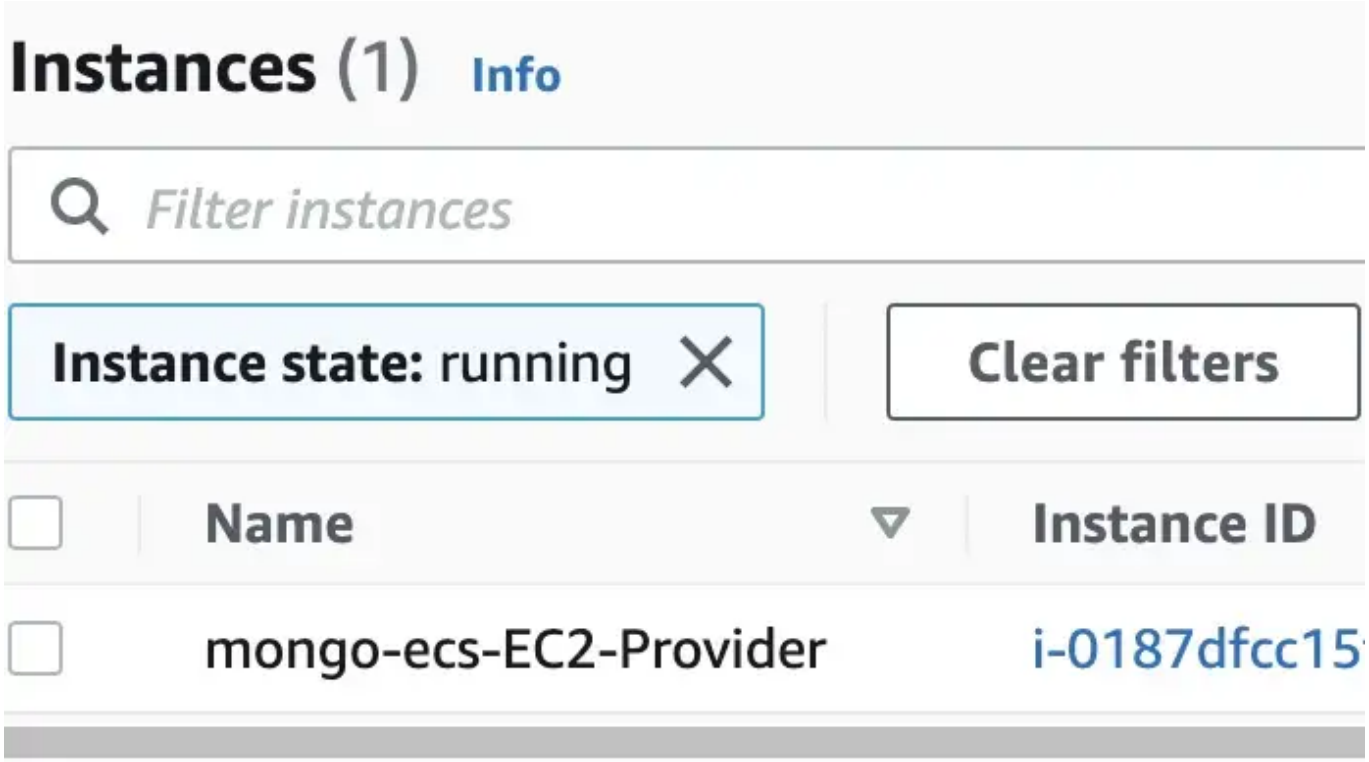

## mongo-ecs-primary service

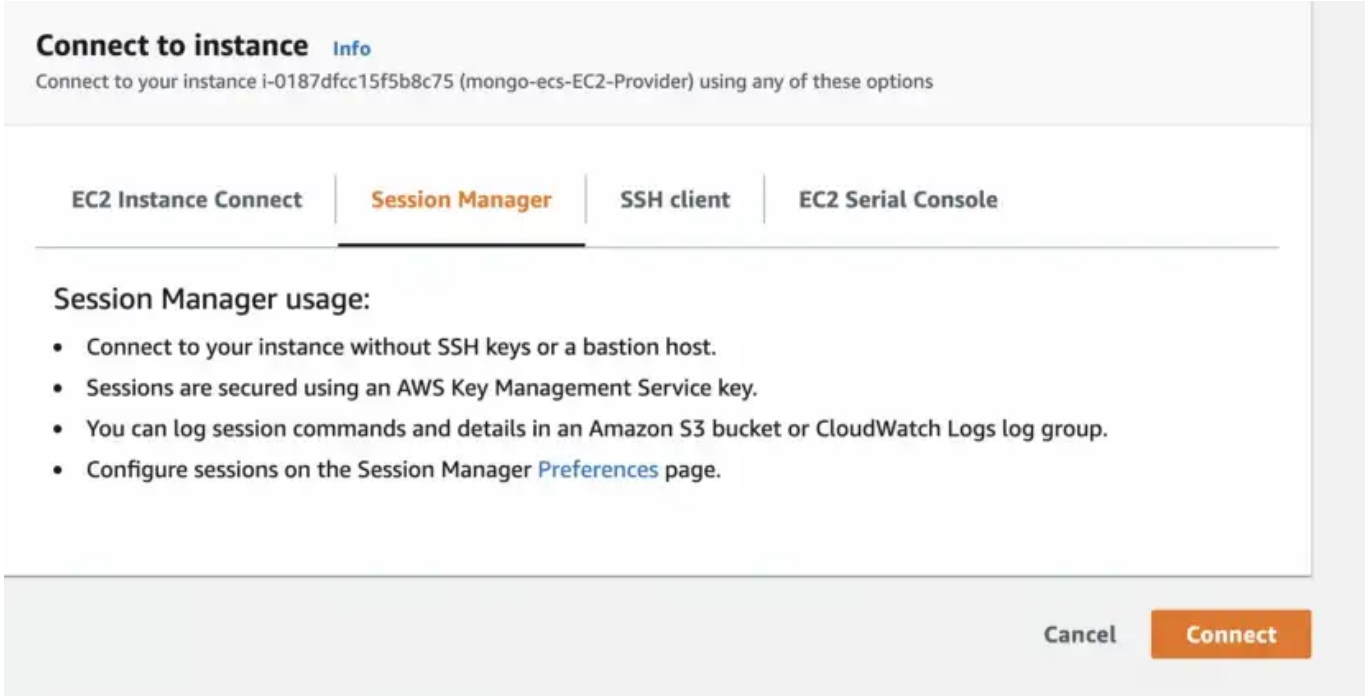
Open `mongo-ecs-primary` service and check the log is starting correctly as below:-

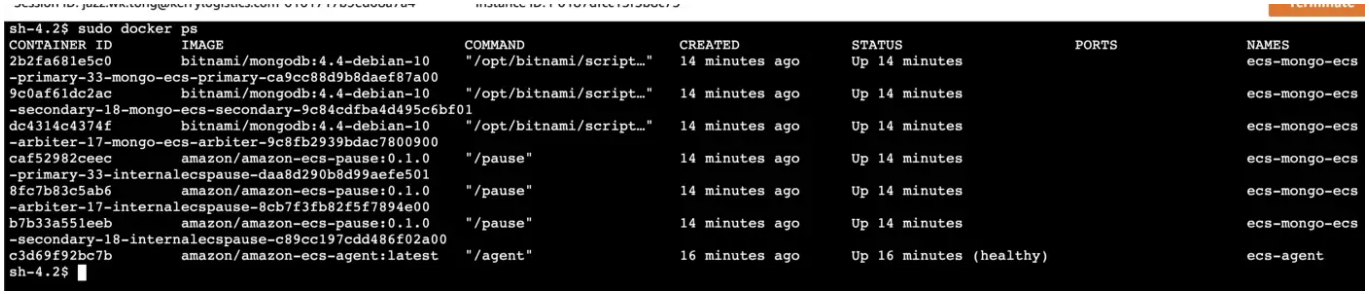



## EC2 instance

Explore your EC2 home console, and you should see one EC2 provisioned. Login to the EC2 using Session Manager, and run `sudo docker ps` , and you will see the current running containers, including the ECS agent container.

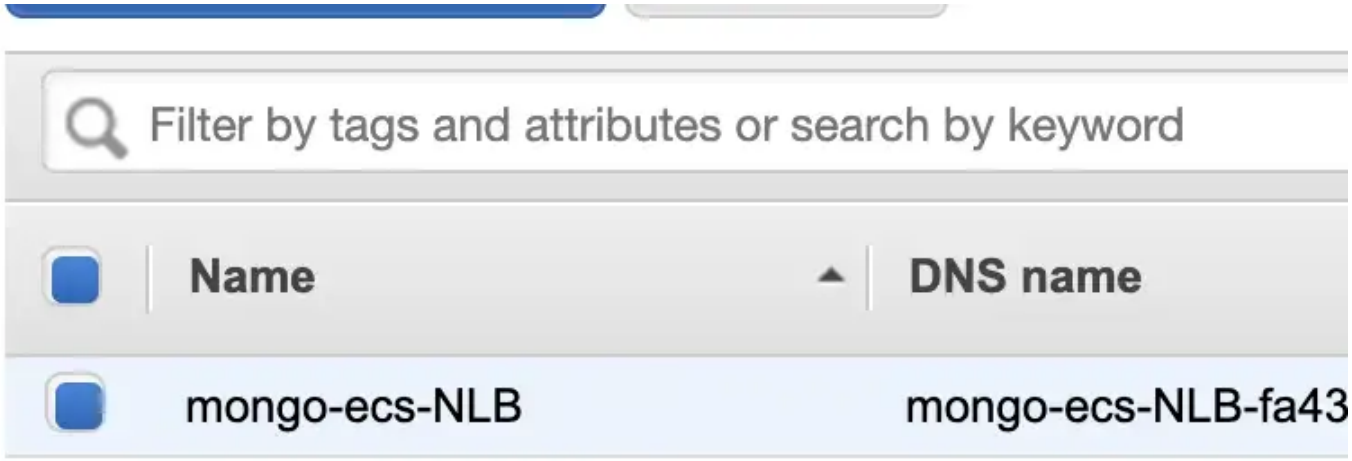





## Network Load Balancer

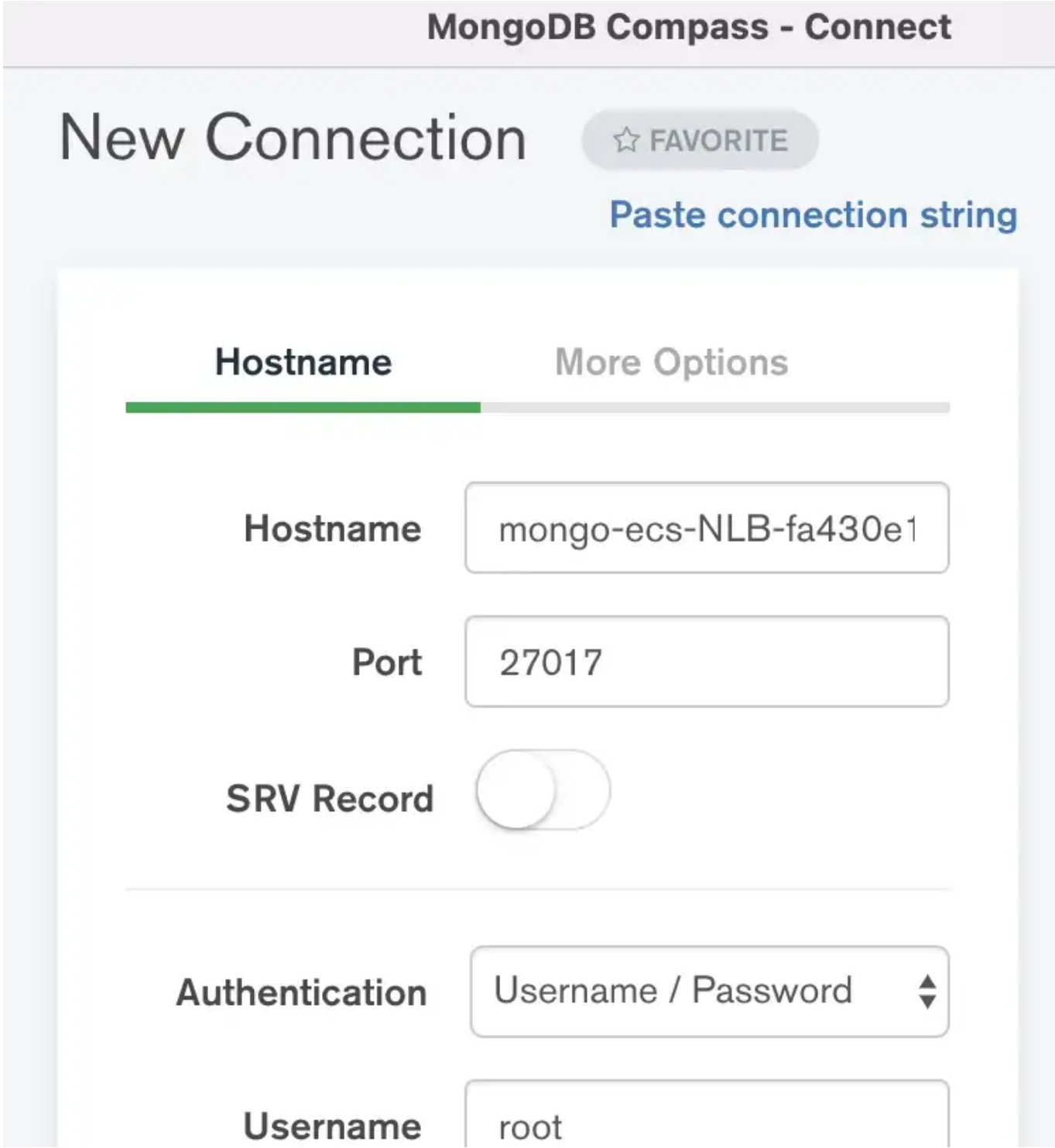
Open Load balancer page, you should see one `Network Load Balancer` provisioned.

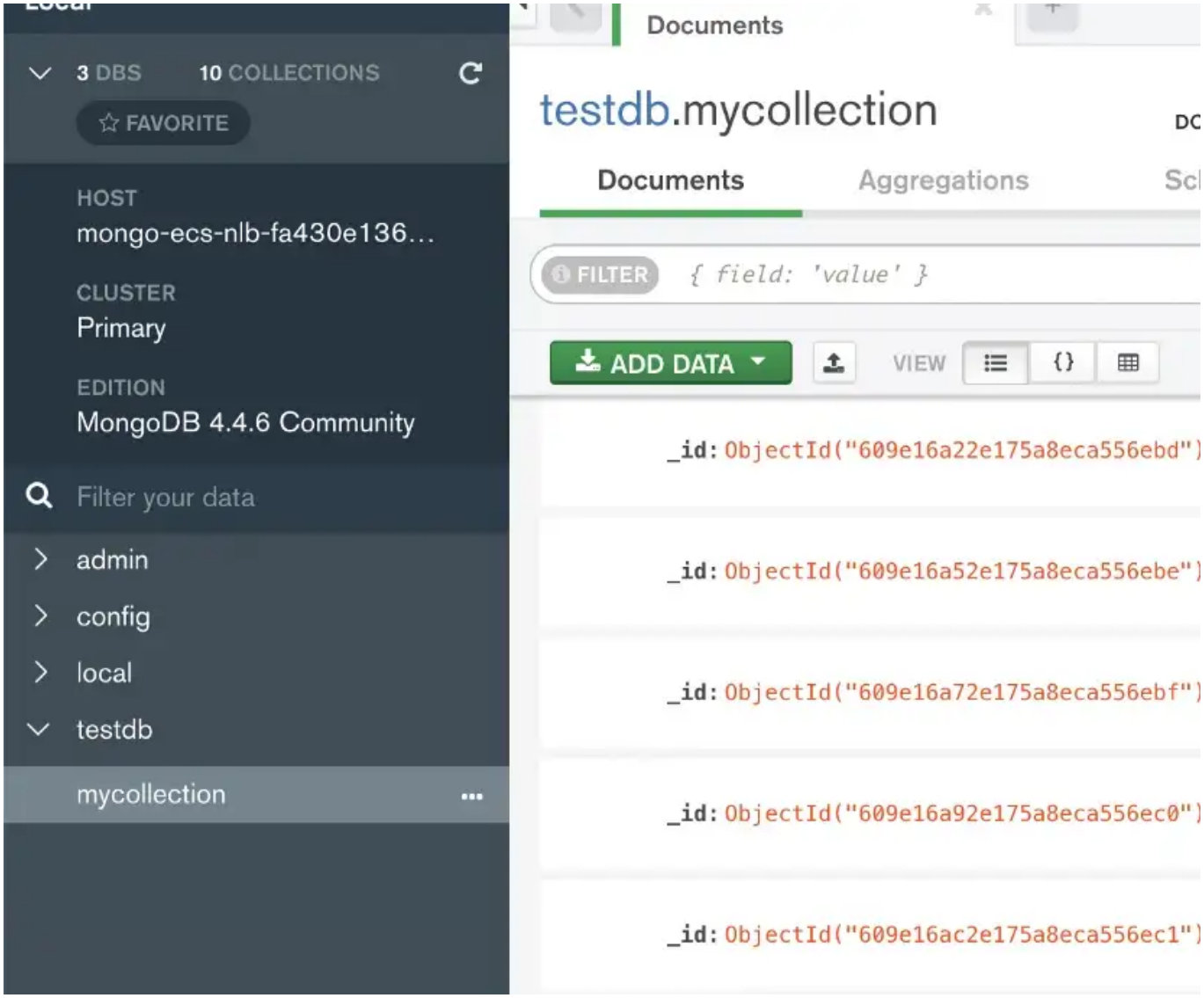



## Test Connection from NLB

Let's test the connection to your MongoDB primary node using MongoDB Compass. Use the NLB DNS in your terraform output and fill in the credential. Create a new DB, and document to verify it is working.

> *If you do not change the default value, the default username is* `root` *and password is* `mypassword`


MongoDB Compass - Connect
New Connection
☆ FAVORITE
Paste connection string
Hostname
More Options
Hostname
mongo-ecs-NLB-fa430e1
Port
27017
SRV Record
Authentication
Username / Password
Username
root

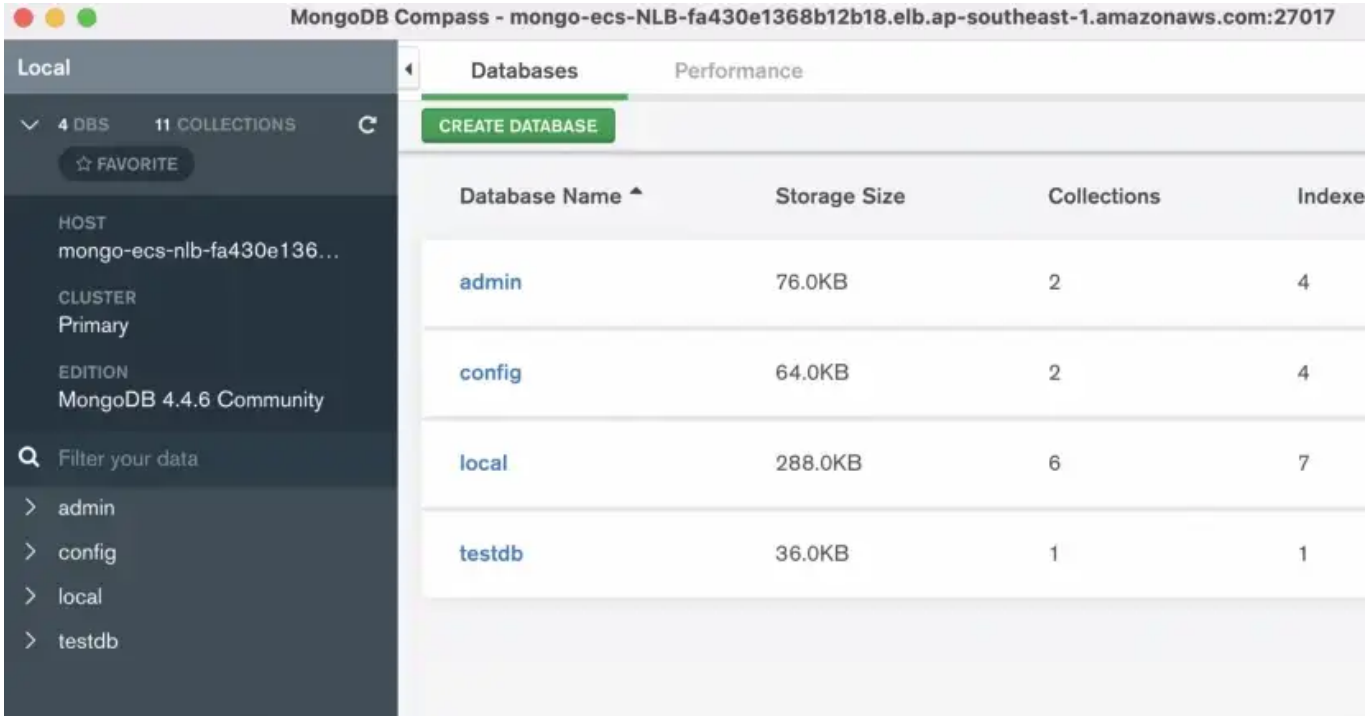



## Failover test

Use MongoDB Compass to connect to the primary and secondary node using NLB DNS. We will perform a failover test to bring down the primary and to ensure failover is working on the secondary node.

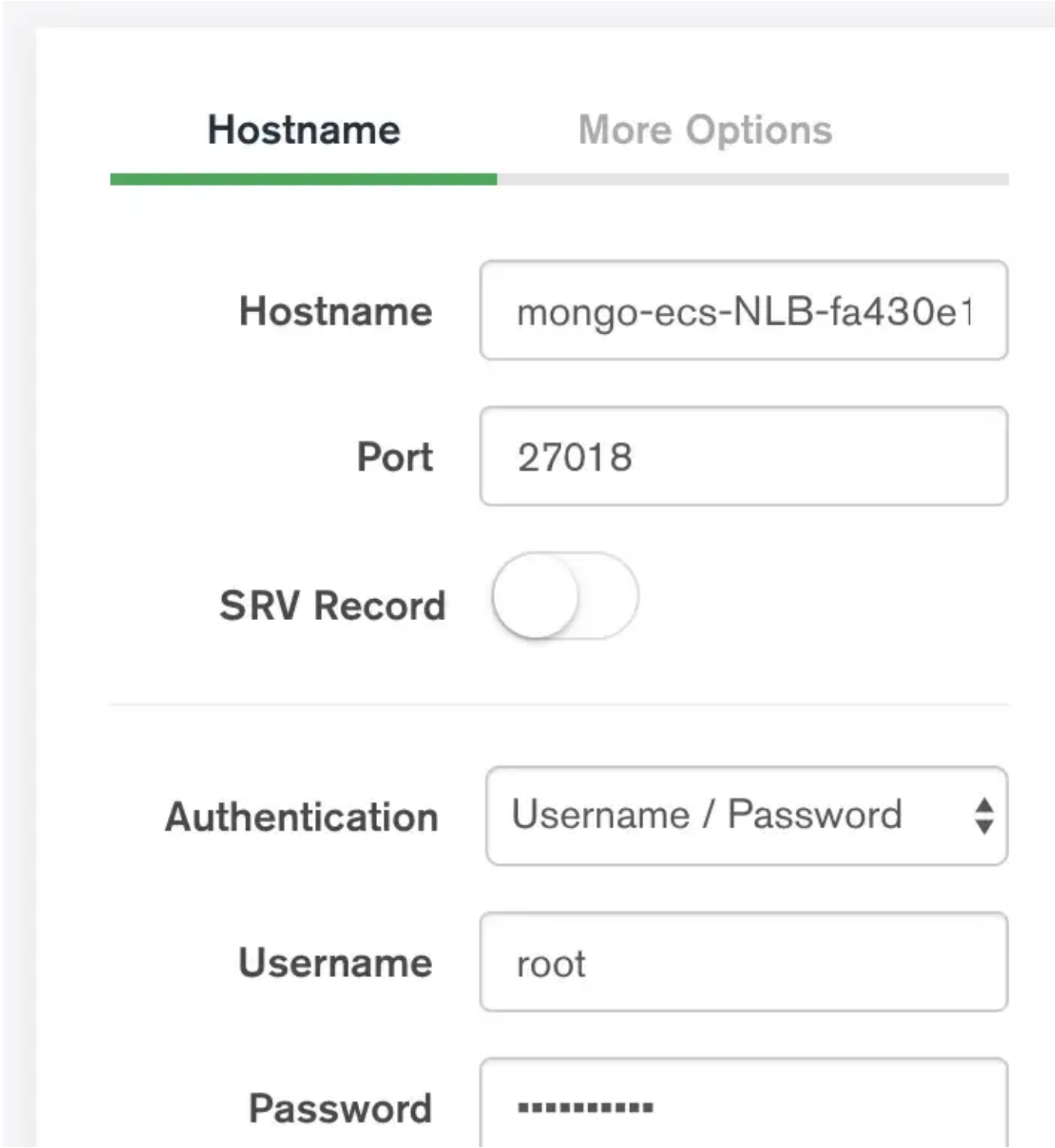
## Connect to Primary Node

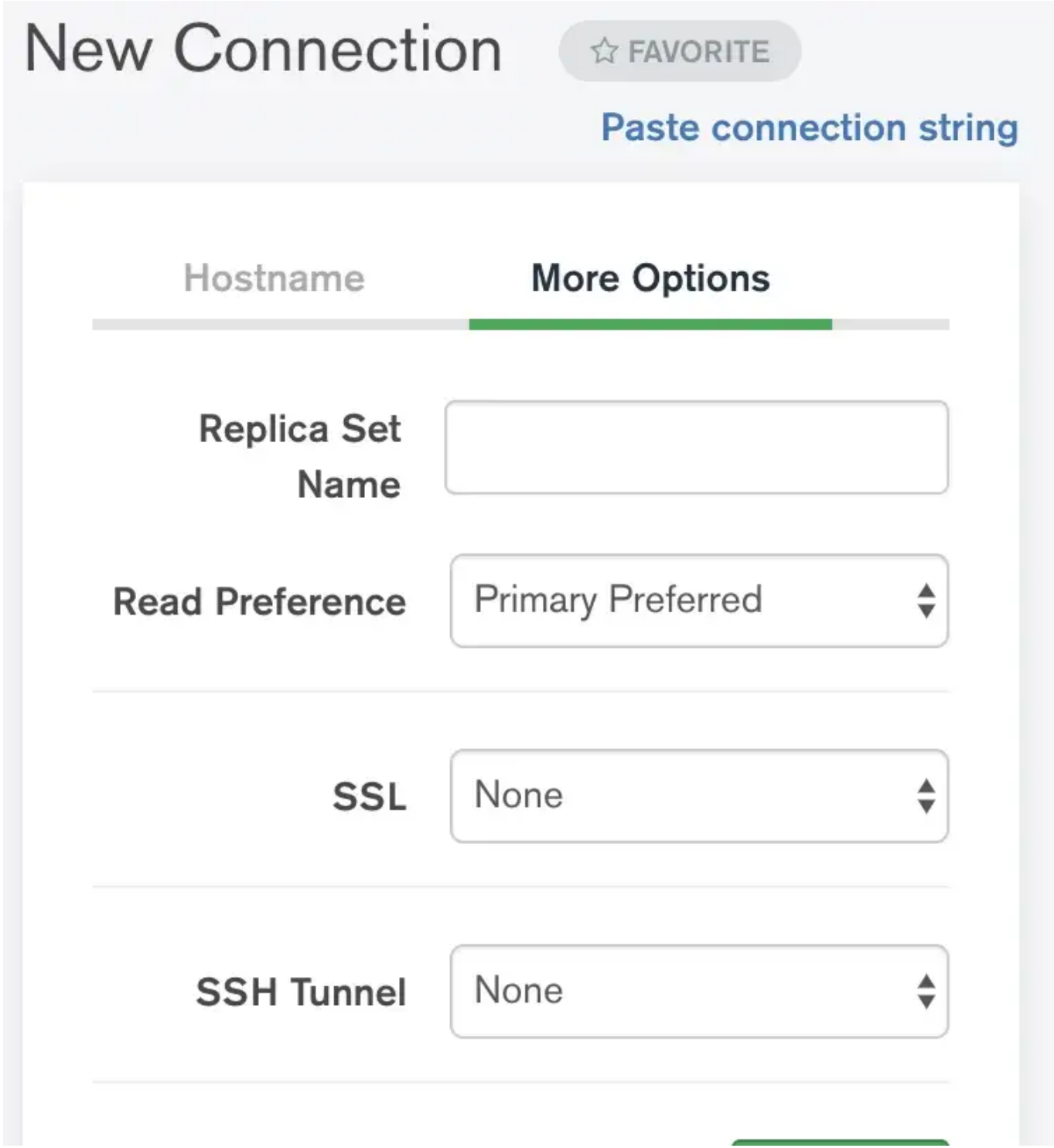
Use MongoDB Compass to connect to the primary node using NLB DNS as below.

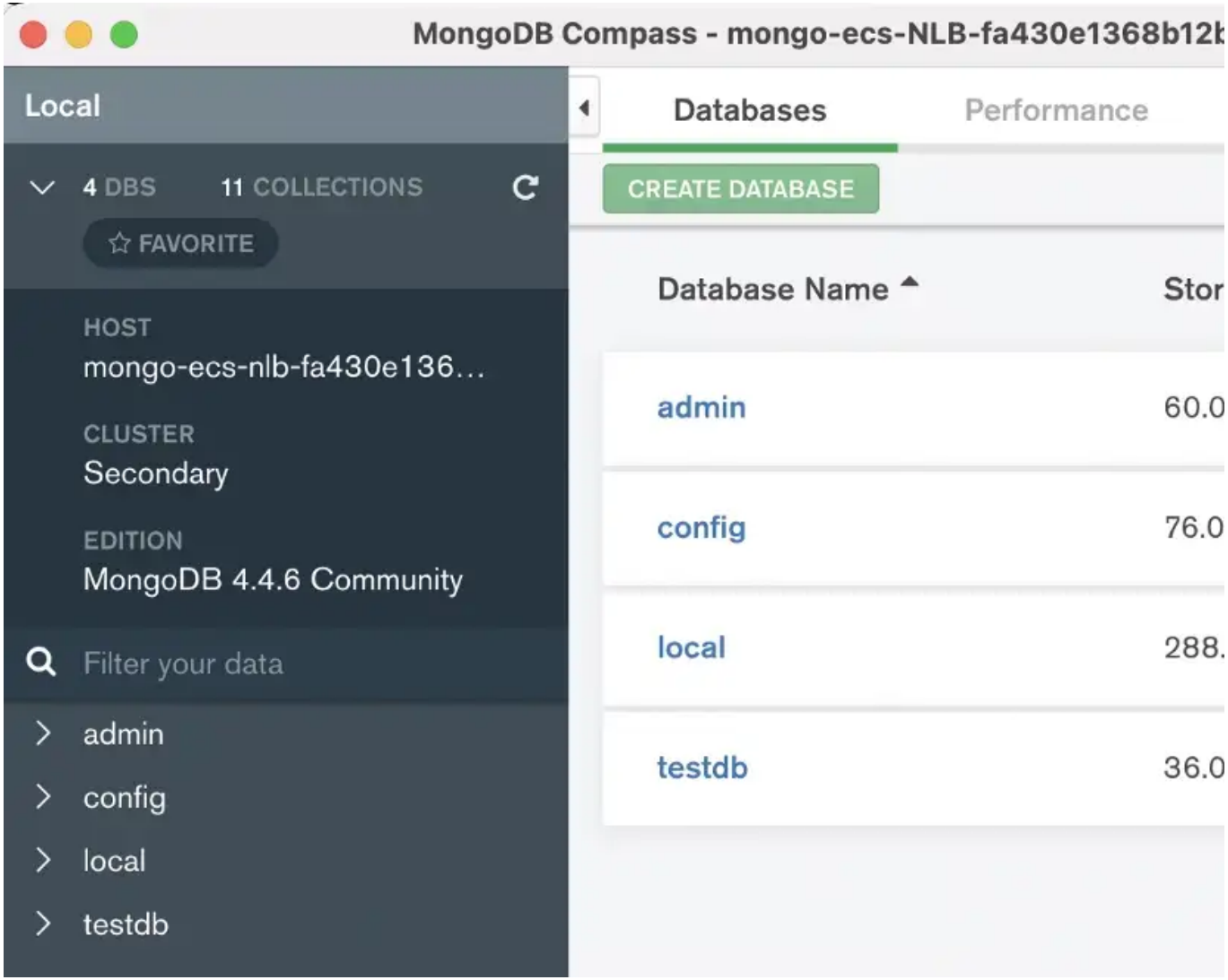



## Connect to Secondary Node

Use MongoDB Compass to connect to Secondary Node. Use the port `27018` as it configures a different port from NLB to route to the Secondary node. As it is a Secondary node, we need to specify Read Preference as `PrimaryPreffered` when we connect to it. Verify that the Secondary Node is not allowed to perform write action.

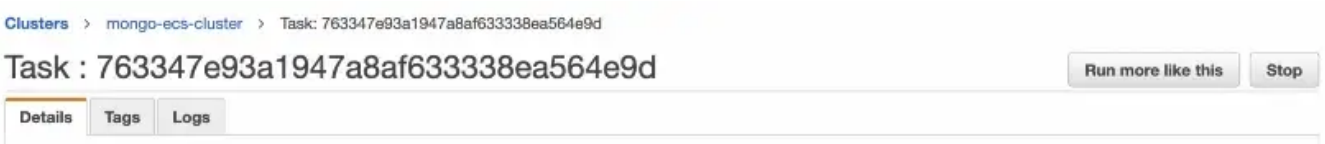

Hostname
More Options
Hostname
mongo-ecs-NLB-fa430e1
Port
27018
SRV Record
Authentication
Username / Password
Username
root
Password


New Connection
☆ FAVORITE
Paste connection string
Hostname
More Options
Replica Set Name
Read Preference
Primary Preferred
SSL
None
SSH Tunnel
None



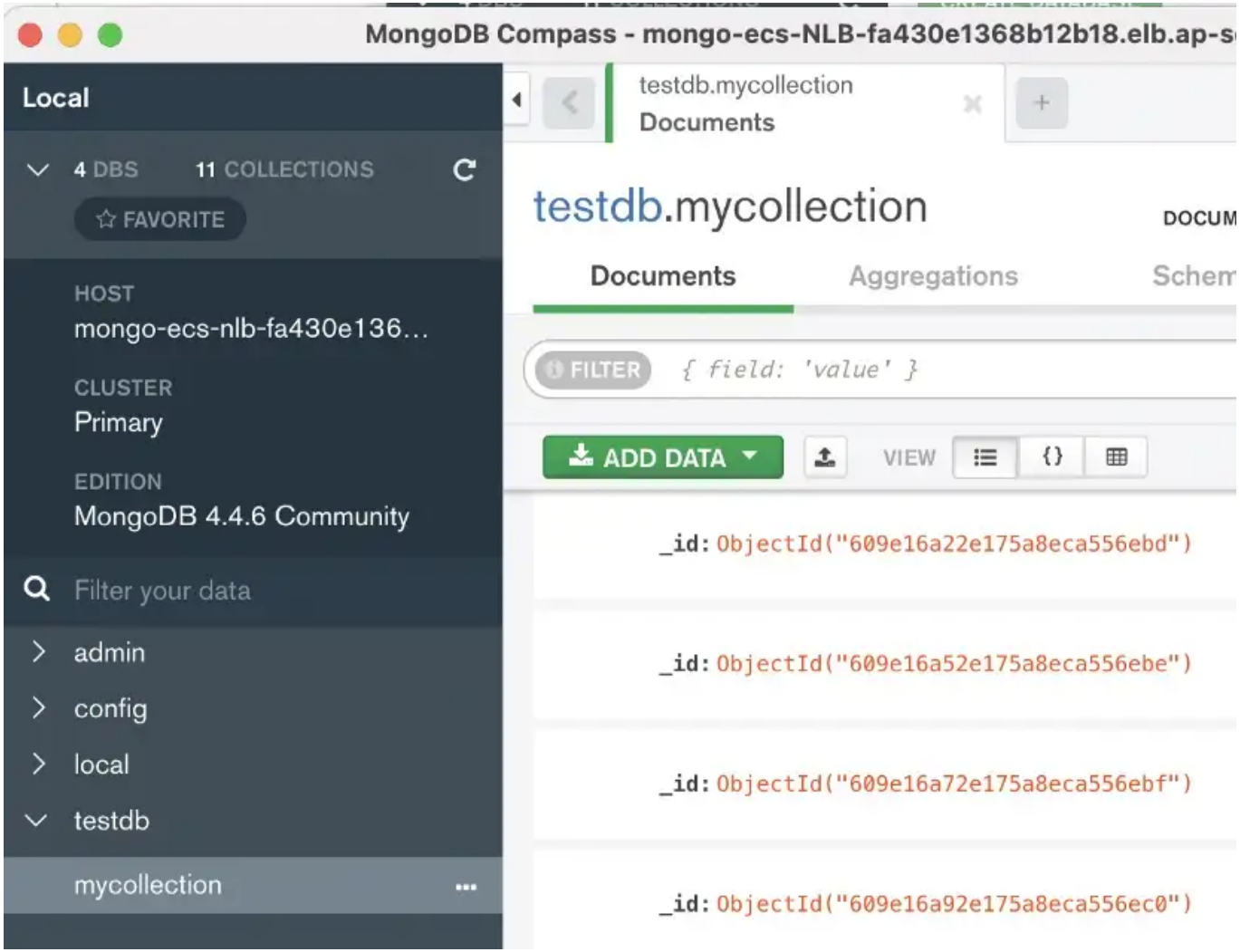

### Stop MongoDB primary task

Open ECS home, and look up the MongoDB Primary service. In the task list, stop the task to simulate the downtime of MongoDB primary node.




## Observe Secondary Node become Primary

Observe the Secondary node, the failover will happen within 1 min time. Secondary node will become primary and allow to perform write action.




## Old Primary Node recovering

One of the Bitnami image features is to recover the replica set to force the original Primary node to become Primary. Observe the MongoDB Primary ECS service until it creates a new task, and it will take over the primary node

when it is stable. It allows the system to go back to its original state without a change on the application side.

## Take away

We use Terraform to build up all the components, and it allows us to easily rebuild if something goes wrong. I hope you enjoy this solution, and let me know if there is any issue to use the template.

**jazztong/mongo-ecs-ha-tf**

Demonstrate Mongo ECS Replica setup with terraform, it will setup 3 MongoDb nodes in ECS with EC2 launch type Run...

github.com

Mongodb Terraform Ecs High Availability Mongodb Replica