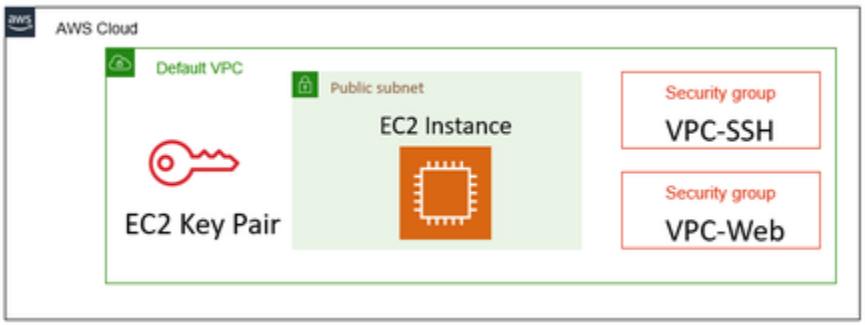
Hand-On Projects

Others:

- **Credentials migration from local to AWS parameter store**
  - credentials were stored local and encrypted using GPG or Ansible
  - This was a security concern
  - employees were able the decrypt all DB passwords local and even leave the company with it if layoffs
- **Rotate all AWS access keys and secret access keys**
  - keys were more than 2 years old
  - Keys were used for DB backup (Cassandra, Postgres, RDS), Jenkins backup, testing, and others
    - Challenges:
      - locate where the is been used in the code
      - identified which team the using the key and where it is been used in the code
      - **PS:** never delete a key while going through a key rotation
      - disable the key to figure out where it is been used
- **Launch EKS cluster using Terraform:**
  - https://github.com/leonardtia1/tia-devops/tree/main/Kubernetes/EKS-MIX/aws-eks-terraform/EKS-TEST
  - https://github.com/leonardtia1/tia-devops/tree/main/Kubernetes/EKS-MIX/aws-eks-terraform/eks-terraform-pure-simple
- **Upgrade EKS cluster:**
  - EKS Upgrade to (X.XX)
  - PS: When we upgrade the EKS cluster, we cannot revert it back to the previous versions. Even the AWS support team cannot
    - Challenges:
      - Deprecated Kubernetes
      - Some charts may not work properly
      - Helm charts upgrade to the latest version
- **Terraform state file issues**
  - time out (resource group in Azure, not resource group in AWS and we have to delete everything manually)
  - state lock (Unlock through CLI and not from the console)
  - ctrl + c (cancel execution)
  - exceed a resource limit (try to create, failed, and mess up the state file)
  - change instance size in production and it failed due to exceeding a resource limit in production deployment (This was to change the Redis size in all environments)
    - Rollback
    - Submit a request to AWS support for a resource quota limit increase in the region

Goal Here:

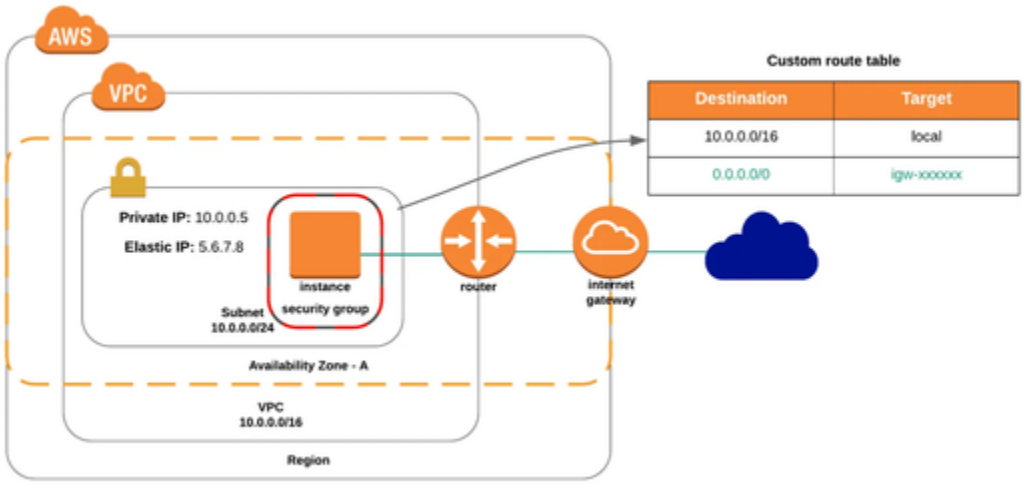
- Create a keypair through the AWS console
- Create 2 security groups. One for SSH and another for the webserver
- Launch and EC2 in the default public subnet and host the website using the user data script
- Make sure to connect using a keypair



Goal Here:

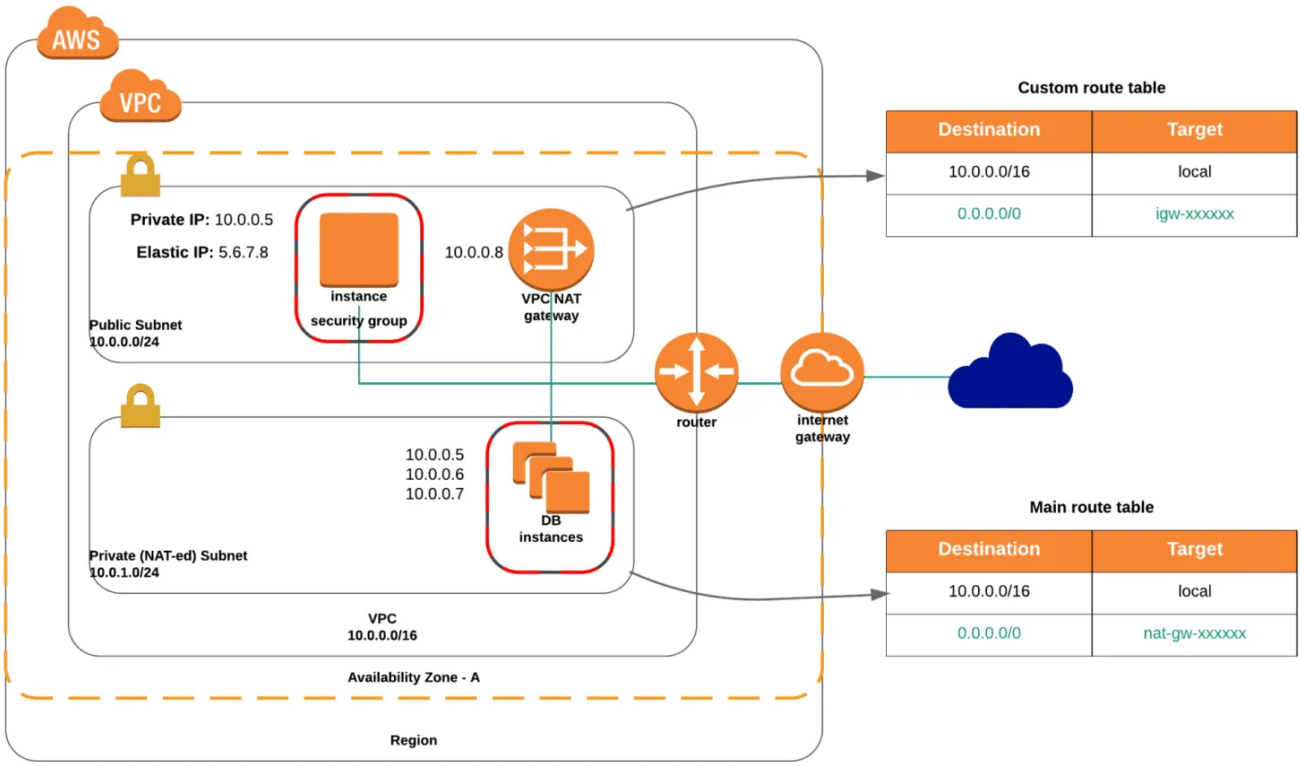
- Create a keypair through the AWS console
- Create EC2 SG
- Create a VPC with 1 public subnet
- Launch an EC2 instance in the public subnet
- Host a website and test

- Make sure to connect using a keypair



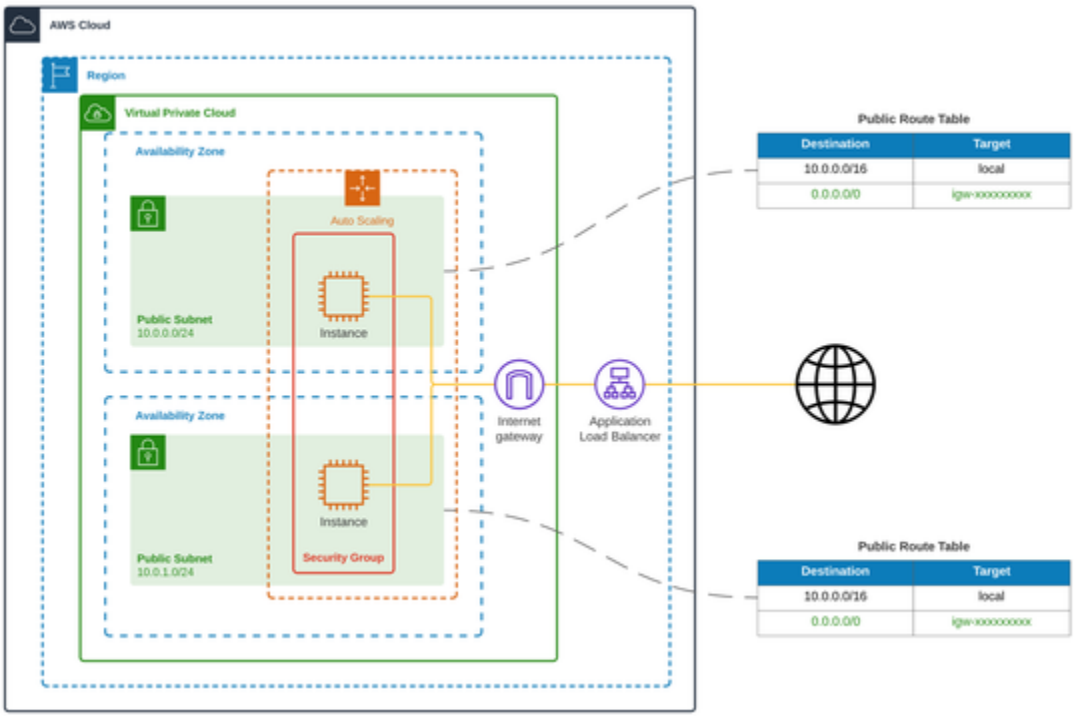
Goal Here:

- Create a key pair through the AWS console
- Create EC2 SG
- Create a VPC with 1 public and 1 private subnet
- Launch bastion host instance in the public subnet
- Launch an EC2 instance in the Private subnet
- Launch a DB instance in the Private subnet
- Test the DB connection from the bastion
- Login into the EC2 in the private subnet and run `yum update`
- Make sure to connect using a keypair



Goal Here:

- Create a key pair through the AWS console
- Create 3 security groups. One for SSH, another for the webserver, and the last for ALB
- Create a VPC with 2 public subnets
- Create a launch configuration
- Create a Load Balancer (ALB)
- Create the auto-scaling group
- Make sure to connect using a keypair

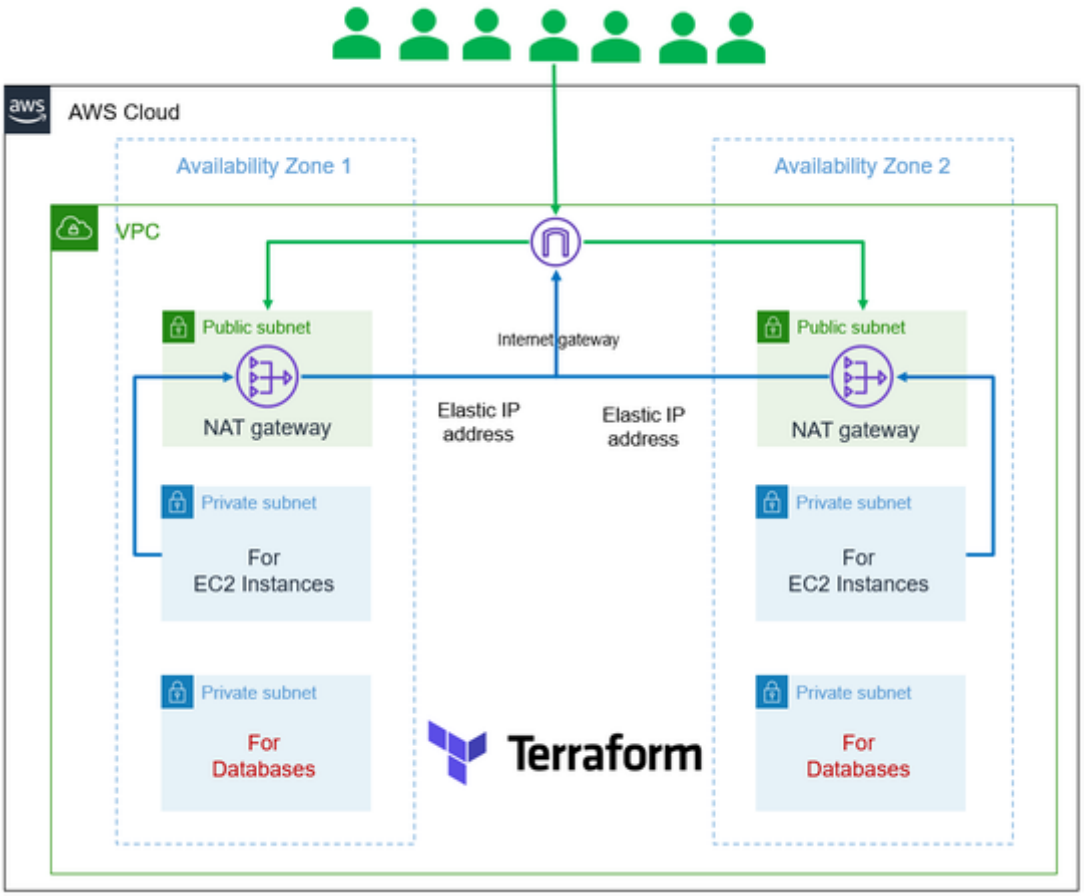


Create The Whole Infrastructure Using Terraform file the following:

- VPC
- Bastion host in the public subnet
- EC2 in private subnets or webserver
- ALB
- Auto Scaling and scaling policy
- Launch configuration
- DB in the private subnet

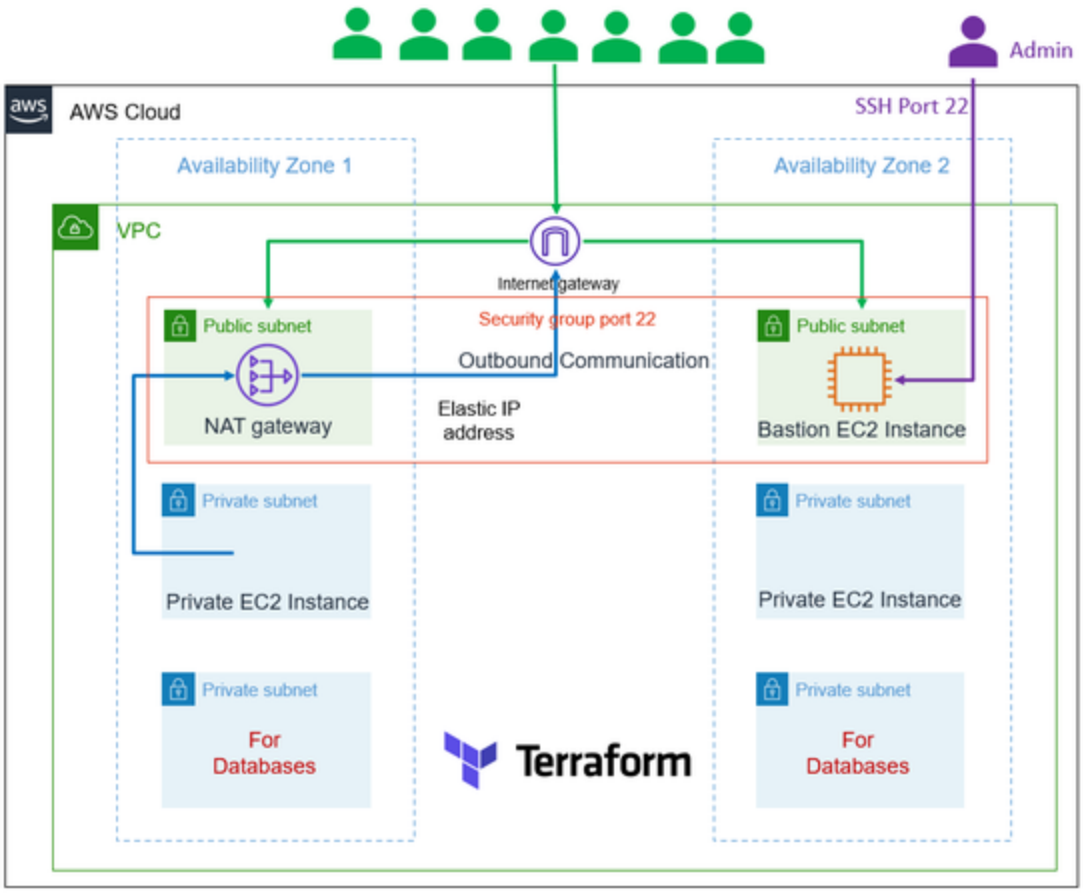
VPC Creation Steps:

- Create VPC
- Create Public and Private Subnets
- Create an Internet Gateway and Associate with the VPC
- Create NAT Gateway in each public Public Subnet for **high availability**
- Create Public Route Table, Add Public Route via Internet Gateway, and Associate Public Subnet
- Create a Private Route Table, Add Private Route via NAT Gateway, and Associate Private Subnet



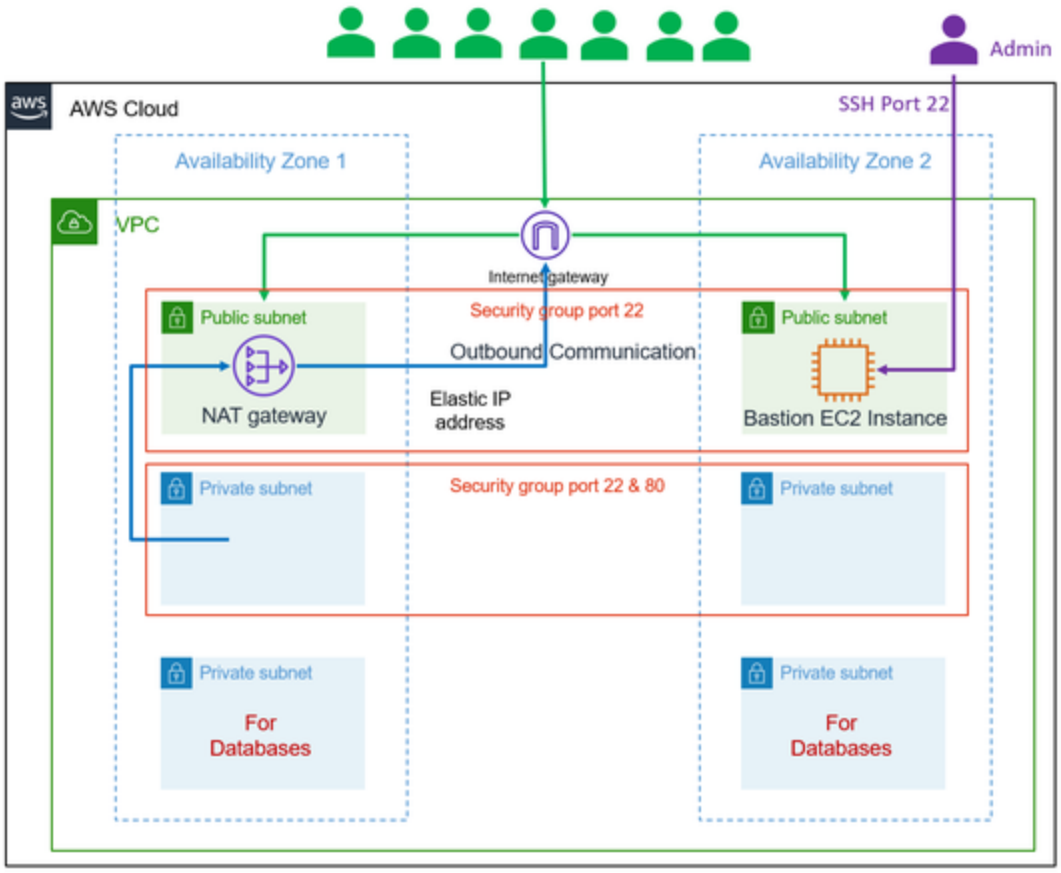
Bastion Host:

- Create a bastion host security group and enable ping capability
- Make sure you create the security group first before the bastion host
- Create a bastion host in the public subnet to access resources in the private subnets.
- Host a website in the bastion host user EC2 user data
- Use Terraform provisioner to copy the private into the bastion host



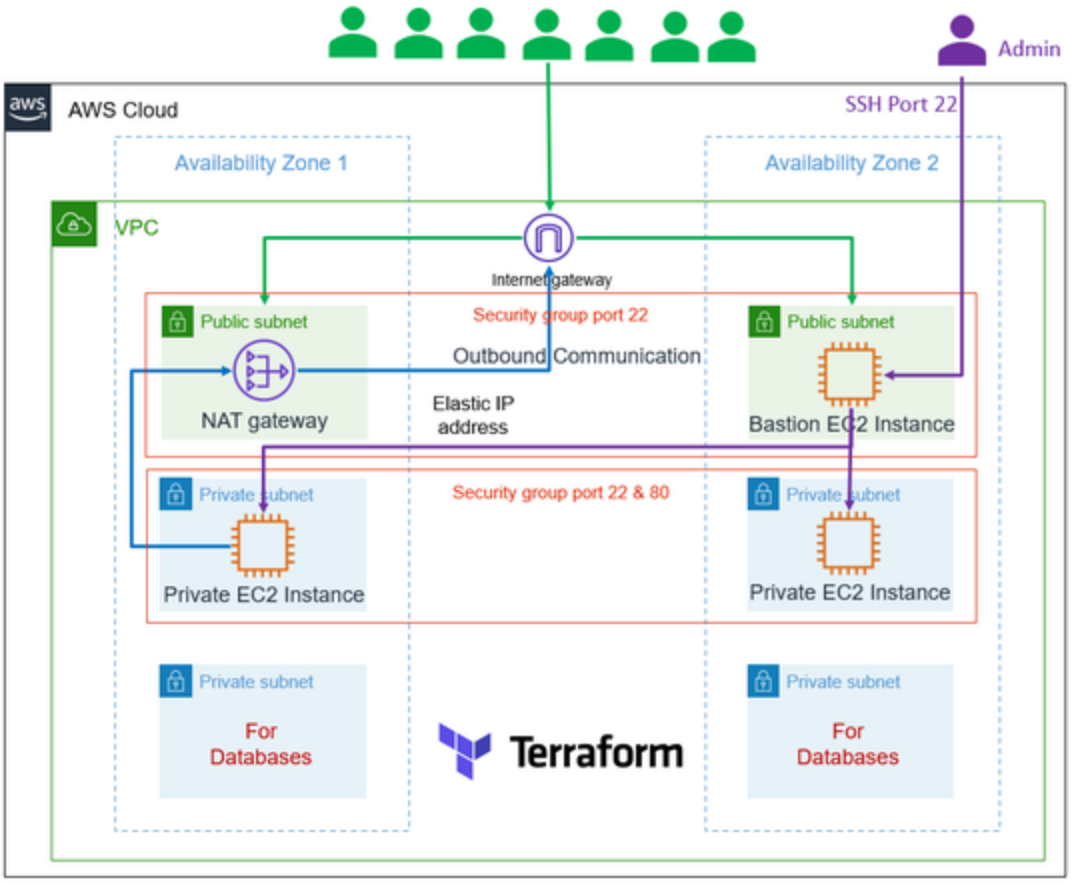
EC2 Instance security group:

- create a security group for EC2 instances within the private subnet
- Make sure that the security can only accept traffic from the VPC **CIDR block**
- Enable ping capability
- Also, create a port 80 that will be used for ELB later on



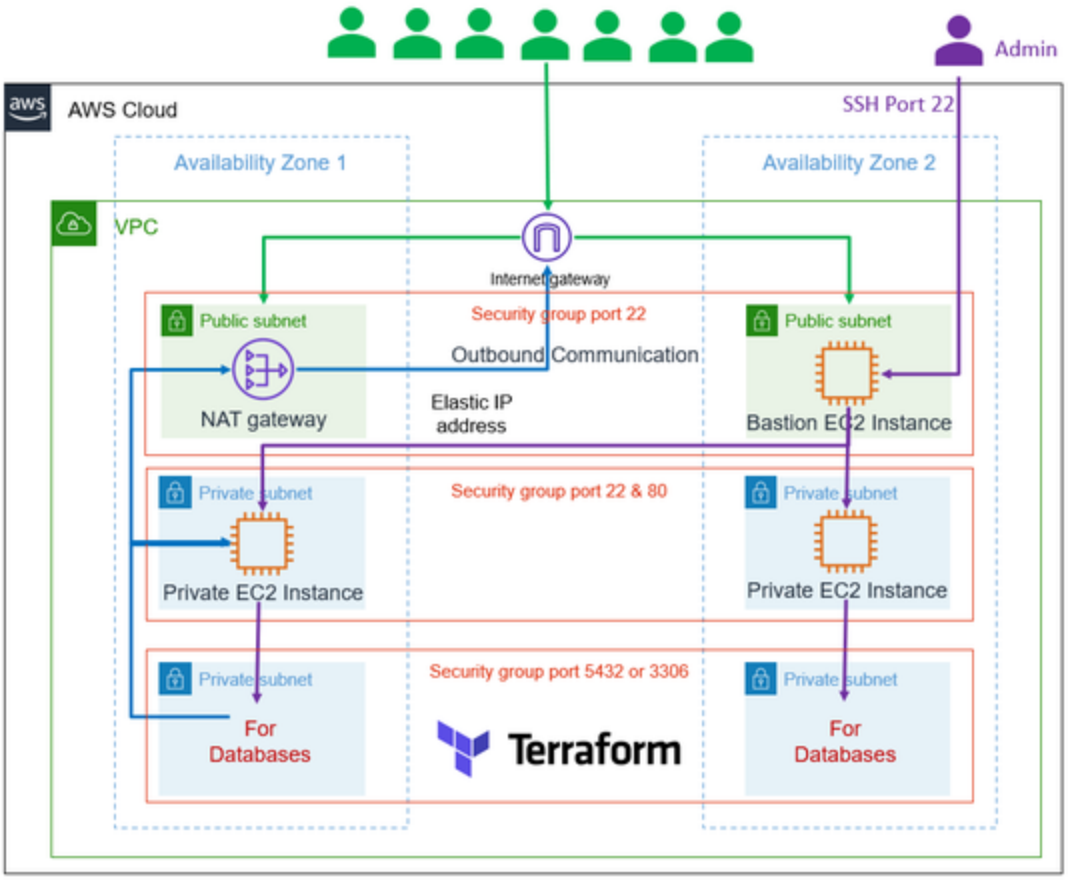
EC2 Instances in the private subnets:

- Create EC2 instances in the private subnet using count
- Login into any EC2 from the bastion host using the private key
- Test if EC2 in the private subnet can access the internet through the NAT gateway by pinging `www.google.com` or by running `yum update`



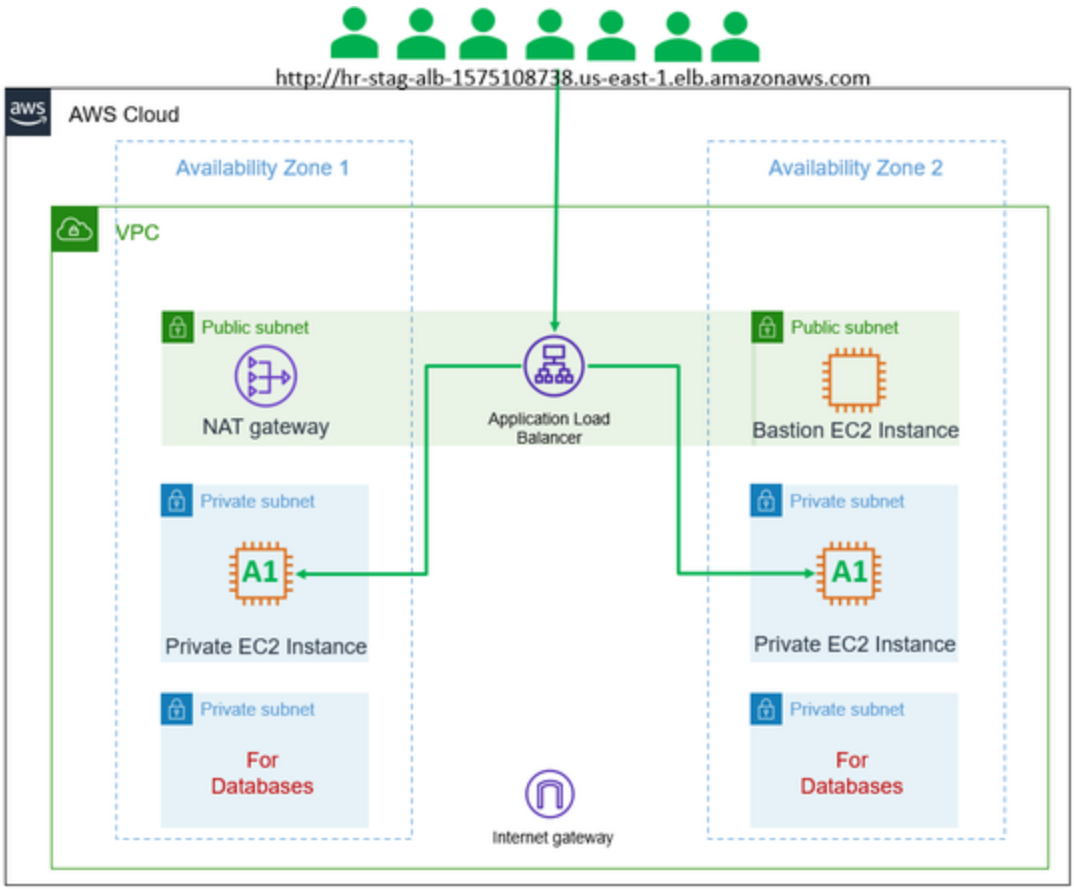
Database:

- Create a DB SG and make sure that it can only access traffic from the VPC CIDR block
- Create a database in the private subnet
- Make sure you protect the database username and password within the module
- install psql on the bastion host
- From the bastion host, login into the database within the private subnet
- List databases to test

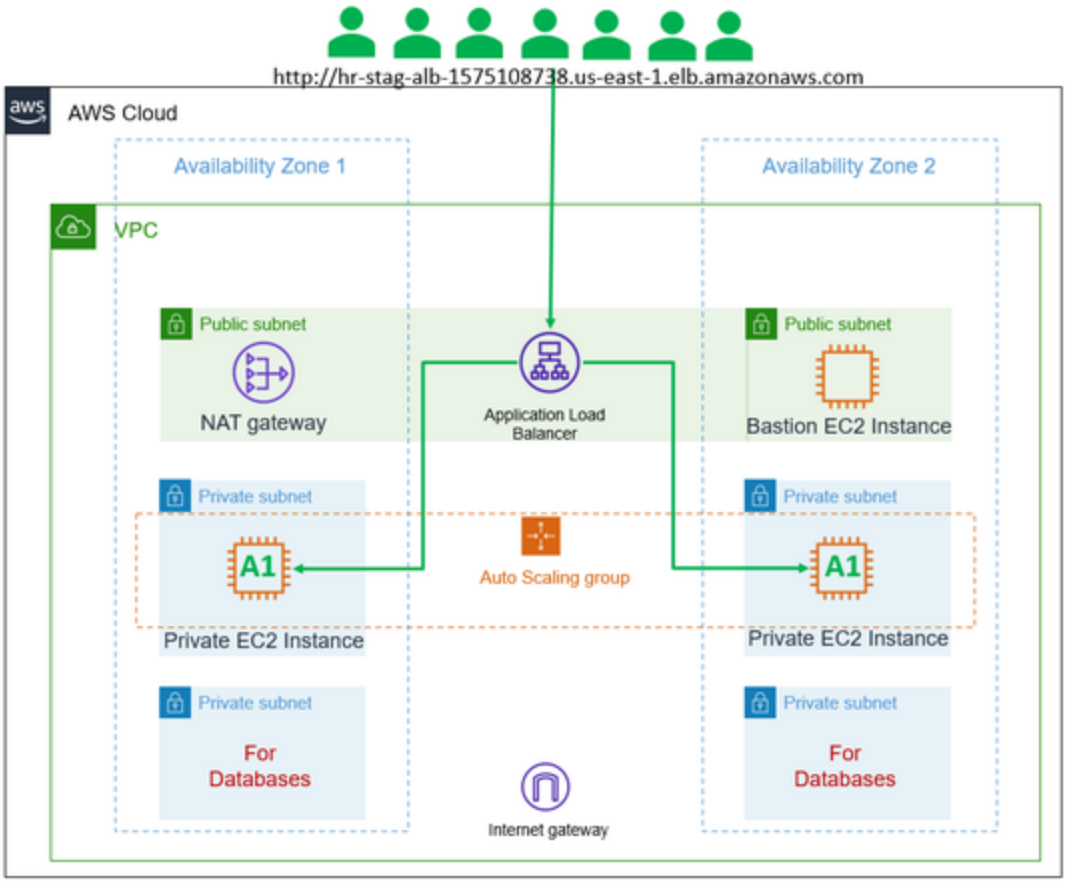


ELS:

- Create an ELB security group and make sure that it can accept traffic from anywhere
- Add the instances running in the private subnet as the target
- Host a website in those instances
- Make sure that those instances can accept traffic from the ELB
- Host a website in those instances
- Test if the website is reachable using the ELB DNS



Auto Scaling:



Terraform Module:

- Write a module that will launch the whole DEV environment
- Use S3 as backend
- Use AWS DynamoDB to lock the state file
- The whole DEV environment should have one state file

Terragrunt:

- Create 4 environments with the following:
  - DEV: with its own state file
  - QA: with its own state file
  - STAGE: with its own state file
  - PROD: with its own state file

PS: Use S3 as the backend and also use DynamoDB to lock the state file