Seneca

Final Project Report

Two-Tier web application automation with Terraform, Ansible and GitHub Actions

Ammar Salmawy - 110561198
Behzad Rajabalipour - 160934212

Prof: Dhana Karuppusamy
ASC730

Date: 9 Dec 2024

Introduction

In this project, we will use Terraform, Ansible, and GitHub actions to deploy and configure a highly available website. First, Terraform will be used to deploy the networking infrastructure which includes VPC, route tables, security groups, and nat/internet gateways then we will deploy and configure the EC2 instances to host the website, auto-scaling group and load balancer. Two of the EC2 instances will be configured as web servers using user data as part of the EC2 initialization. Second, the Ansible playbook will be used to configure the Apache servers for the rest of the instances. Lastly, we will use GitHub and GitHub actions as a continuous integration tool to automate the deployment of the website.

Traffic flow Terraform

This report will share snippets of the code and explain its functionality and what goal it serves as the overall deployment will be in the demo recording.

First Networking deployment using Terraform

```
module "vpc-dev" {
  # source = "../../prod_network"
  source              = "git::https://github.com/Behzad-Rajabalipour/prod_network.git"     # it's be
  env                 = var.env
  vpc_cidr            = var.vpc_cidr
  public_cidr_blocks  = var.public_cidr_blocks
  private_cidr_blocks  = var.private_cidr_blocks
  prefix              = var.prefix
  default_tags        = var.default_tags
}
```

We used GitHub repo as the source of our code

```
# Create a new VPC
resource "aws_vpc" "main" {
  cidr_block       = var.vpc_cidr
  instance_tenancy = "default"
  tags = merge(
    local.default_tags, {
      Name = "VPC-${var.prefix}"
    }
  )
}
```

Creating a new VPC

```
# Create Internet Gateway
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main.id

  tags = merge(local.default_tags,
    {
      "Name" = "IGW-${var.prefix}"
    }
  )
}


# Create the NAT Gateway
resource "aws_nat_gateway" "nat_gw" {
  allocation_id = aws_eip.nat_eip.id
  subnet_id     = aws_subnet.public_subnet[0].id # Ensure this is your public subnet
  tags = {
    Name = "${var.prefix}-nat-gateway"
  }
}
```

Create an internet gateway and NAT gateway for internet access

The NAT gateway is deployed into a public subnet and its functionality is to provide access to the internet for the EC2 instances in the Private subnet to install the Apache server.

```
# Add provisioning of the public subnetin the default VPC
resource "aws_subnet" "public_subnet" {
  count             = length(var.public_cidr_blocks)
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.public_cidr_blocks[count.index]
  availability_zone = data.aws_availability_zones.available.names[count.index] # so these two subnets are in
  tags = merge(
    local.default_tags, {
      Name = "${var.prefix}-public-subnet-${count.index + 1}"
    }
  )
}
```

```
# Provision public subnets in custom VPC
variable "public_cidr_blocks" {
  default     = ["10.1.1.0/24", "10.1.2.0/24", "10.1.3.0/24", "10.1.4.0/24"]
  type        = list(string)
  description = "Public Subnet CIDRs"
}
```

Deploying 4 public subnets into 4 different availability zones

```
resource "aws_subnet" "private_subnet" {
  count             = length(var.private_cidr_blocks)
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.private_cidr_blocks[count.index]
  availability_zone = data.aws_availability_zones.available.names[count.index] #
  tags = merge(
    local.default_tags, {
      Name = "${var.prefix}-private-subnet-${count.index + 1}"
    }
  )
}
```

```
# Provision private subnets in custom VPC
variable "private_cidr_blocks" {
  default     = ["10.1.5.0/24", "10.1.6.0/24"]
  type        = list(string)
  description = "Public Subnet CIDRs"
}
```

Deploying 2 private subnets into the first 2 availability zones

```
# Route table to route add default gateway pointing to Internet Gateway (IGW)
resource "aws_route_table" "public_route_table" {
  vpc_id = aws_vpc.main.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }
  tags = {
    Name = "${var.prefix}-RT-public-subnets"
  }
}

# Route table for private subnets pointing to NAT Gateway
resource "aws_route_table" "private_route_table" {
  vpc_id = aws_vpc.main.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_nat_gateway.nat_gw.id
  }
  tags = {
    Name = "${var.prefix}-RT-private-subnets"
  }
}
```

Traffic is routed to the IGW for public subnets and to the NAT gateway from the private subnet

```
# Load Balancer
resource "aws_lb" "ALB" {
  name               = "ALB"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.lb_sg.id]
  subnets            = aws_subnet.public_subnet[*].id            # ids of both pu

  enable_deletion_protection = false

  tags = {
    Name = "MyApplicationLoadBalancer"
  }
}
```

Deploy the load balancer into two subnets

```
# Security Group for the Load Balancer
resource "aws_security_group" "lb_sg" {
  name        = "lb_sg"
  vpc_id      = aws_vpc.main.id
  description = "Allow HTTP and HTTPS traffic"

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]  # Allow HTTP from anywhere
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"  # Allow all outbound traffic
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Allow inbound traffic to port 80(HTTP) for the load balancer

```
# Listener for the Load Balancer
resource "aws_lb_listener" "listener" {
  load_balancer_arn = aws_lb.ALB.arn
  port              = 80
  protocol          = "HTTP"

  default_action {
    type = "forward"
    target_group_arn = aws_lb_target_group.TG.arn
  }
}
```

Create an HTTP listener on the ALB that forwards traffic to the target group

```
# it only can get output.tf of networkmodule
output "public_subnet_ids" {
  value = module.vpc-dev.public_subnet_ids # error => value = aws_subnet.public_subnet[*].id
}

# it only can get output.tf of networkmodule
output "public_subnet_cidrs" {
  value = module.vpc-dev.public_subnet_cidrs # error => value = aws_subnet.public_subnet[*].id
}

# it only can get output.tf of networkmodule
output "private_subnet_ids" {
  value = module.vpc-dev.private_subnet_ids # error => value = aws_subnet.public_subnet[*].id
}

output "public_route_table_id" {
  value = module.vpc-dev.public_route_table.id
}

output "vpc_id" {
  value = module.vpc-dev.vpc_id # error => value = aws_vpc.main.id
}

output "vpc_cidr" {
  value = module.vpc-dev.vpc_cidr
}

output "target_group_arn" {
  value = module.vpc-dev.target_group.arn
}

output "ALB_SG_id" {
  value = module.vpc-dev.ALB_SG_id
}
```

Output subnets, ABL, TG and VPC IDs so it can be used when deploying the web servers

```
resource "aws_instance" "public_instance" {
  count                       = length(data.terraform_remote_state.prod_net_tfstate.outputs.public_subnet_ids)
  ami                         = data.aws_ami.latest_amazon_linux.id
  instance_type               = lookup(var.instance_type, var.env)
  key_name                    = aws_key_pair.keyName.key_name    # because we used count in key_name so we have to give it's index
  security_groups             = [aws_security_group.public_instance_SG.id]
  subnet_id                   = data.terraform_remote_state.prod_net_tfstate.outputs.public_subnet_ids[count.index] # it use s3/dev/network/terraform.tfstate => outputs.
  associate_public_ip_address = true
  user_data = count.index == 0 || count.index == 2 ? templatefile("${path.module}/install_httpd.sh.tpl",
    {
      env    = upper(var.env),
      prefix = upper(var.prefix),
      name   = "Behzad Rajabalipour"
    }
  ) : null

  root_block_device { # it will encrypt it if it is in env=test
    encrypted = var.env == "test" ? true : false
  }

  lifecycle { # lifecycle will create a new instance before destroy previous one
    create_before_destroy = true
  }

  # Dynamic tags
  tags = merge(
    local.default_tags,
    count.index == 1
      ? {
          Name = "Bastion VM webserver${count.index + 1}",
          Role = "Bastion VM"
          Type = "type1"
        }
      : {
          Name = "webserver${count.index + 1}",
          Type = contains([3], count.index) ? "type1" : null
        }
  )
}
```

Deployment of EC2 instance into the public subnets

Using a count to iterate to all public subnets then using user data to install httpd to the first and third EC2 instances index 0 and 2. then using type1 tag for instances 2 and 4 (index 1 and 3). The tag will be used to target these instances and install/configure the web server using Ansible

```
lifecycle { # lifecycle will create a new instance before destroy previous one
  create_before_destroy = true
}
```

Terraform ensures a new resource is created before an existing resource is destroyed

This ensures the high availability of the websites it avoids service interruptions by maintaining the old resource until the new one is fully provisioned and ready

```
resource "aws_security_group" "private_instance_SG" {
  name        = "allow_ssh_icmp"
  description = "Allow SSH and ICMP traffic from a specific VM"
  vpc_id      = data.terraform_remote_state.prod_net_tfstate.outputs.vpc_id

  # Ingress rule for SSH
  ingress {
    description = "Allow SSH from specific VM"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["${aws_instance.public_instance[1].private_ip}/32"]  # private_ip(IPV4 pri

  }

  # Egress rule to allow all outbound traffic
  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"    # Allow all egress traffic
    cidr_blocks      = ["0.0.0.0/0"]
  }

  tags = {
    Name = "Allow SSH and ICMP Security Group"
  }
}
```

Using the IP address of web server 2 to allow SSH into the EC2 instances in the private subnet

```
# Register EC2 Instances to Target Group
resource "aws_lb_target_group_attachment" "tg_attachment" {
  count            = length(aws_instance.public_instance[*].id) - 1  # exclude the last public instance (We
  target_group_arn = data.terraform_remote_state.prod_net_tfstate.outputs.target_group_arn
  target_id        = aws_instance.public_instance[count.index].id
  port             = 80
}
```

Attach the first 3 web servers to the target group

```
resource "aws_autoscaling_policy" "cpu_scale_out" {
  name                   = "target-tracking-scale-up-policy"
  autoscaling_group_name = aws_autoscaling_group.public_instance_asg.name
  policy_type            = "TargetTrackingScaling"

  target_tracking_configuration {
    target_value             = 60  # Trigger scale-out when CPU usage exceeds 60%
    predefined_metric_specification {
      predefined_metric_type = "ASGAverageCPUUtilization"
    }
    disable_scale_in         = true  # Disable scaling in
  }
}

#-------------------------------------------
# Simple Scaling policy
# we can have many simple scaling policy

# create cloudwatch alaram with CPUUtilization metric
resource "aws_cloudwatch_metric_alarm" "scale_in_alarm" {
  alarm_name          = "$scale-in-alaram"
  comparison_operator = "LessThanOrEqualToThreshold"        # <=

  evaluation_periods  = 2  # data points, Number of periods to evaluate
  datapoints_to_alarm = 2  # data points, Number of data points that must breach the threshold

  metric_name         = "CPUUtilization"           # metric name for this alaram
  namespace           = "AWS/EC2"
  period              = 60                 # 60 sec for alaram interval
  statistic           = "Average"
  threshold           = 40                   # CPUUtilization < 40%

  alarm_description   = "Trigger scale-in when CPU utilization is less than 40%"
  dimensions = {
    AutoScalingGroupName = aws_autoscaling_group.public_instance_asg.name
  }
  actions_enabled = true

  alarm_actions = [aws_autoscaling_policy.scale_in_policy.arn]  # Link to scaling policy
}

# create simple scaling policy
resource "aws_autoscaling_policy" "scale_in_policy" {
  name                   = "$simple-scaling-in-policy"
  scaling_adjustment     = -1  # Reduce by 1 instance
  adjustment_type        = "ChangeInCapacity"  # Adjust the number of instances
  cooldown               = 70  # Wait 70 seconds between scaling activities
  autoscaling_group_name = aws_autoscaling_group.public_instance_asg.name
}
```

Creating a scale-in and scale-out policy for the auto-scaling group
This also creates an alarm for the cloud watch to monitor the EC2 instance

Ansible traffic flow

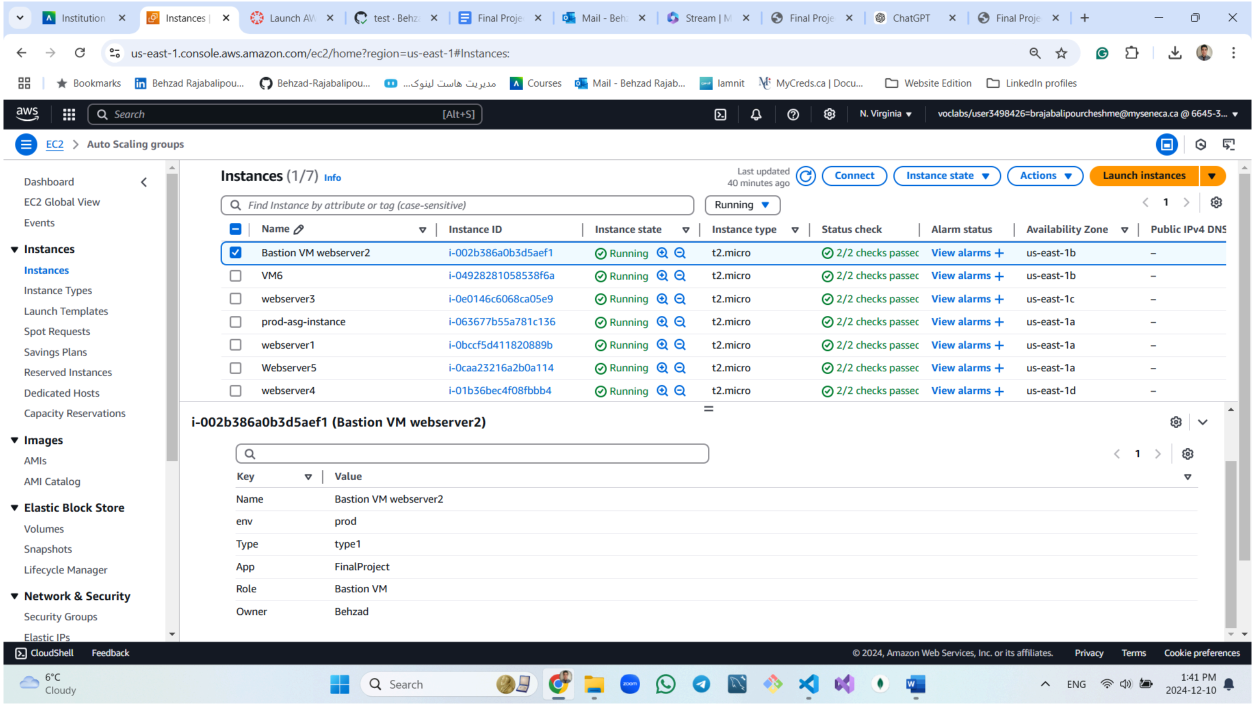
Ansible will be used to configure instances 2 and 4 (tagged: type1) we will use playbook and jinja2 templates for this purpose.

```yaml
plugin: aws_ec2
regions:
  - us-east-1
keyed_groups:
  - key: tags.Type
    prefix: tag         # Adds the prefix tag_ to the group names, like tag_
filters:
  instance-state-name : running
compose:
  ansible_host: public_ip_address     # ansible_host: is mandatory like ansi
  ansible_hostname: public_ip_address   # i used it in index.j2 file
  # ansible_user                  : ec2-user
  # ansible_ssh_private_key_file : ~/.ssh/prodKey     # it will store in the
```

Dynamic Inventory Setup: this will group instances based on their tags

```yaml
- hosts: tag_type1
  gather_facts: True
  become: yes

  vars:
    source_file: ./index.j2
    dest_file: /var/www/html
```

Targeting type in the playbook



As you see in the picture, Webserver2 and 4 has a tag name: Type = type1
So ansible inventory will dynamically take these two instances IP

```yaml
tasks:
  - name: Install Apache Web Server for RPM
    yum: name=httpd state=latest
    when: ansible_os_family == "RedHat"

  - name: Print Linux Family
    debug: var=ansible_os_family

  - name: Get the local IPv4 address
    shell: curl http://169.254.169.254/latest/meta-data/local-ipv4
    register: myip
    args:
      warn: false
```

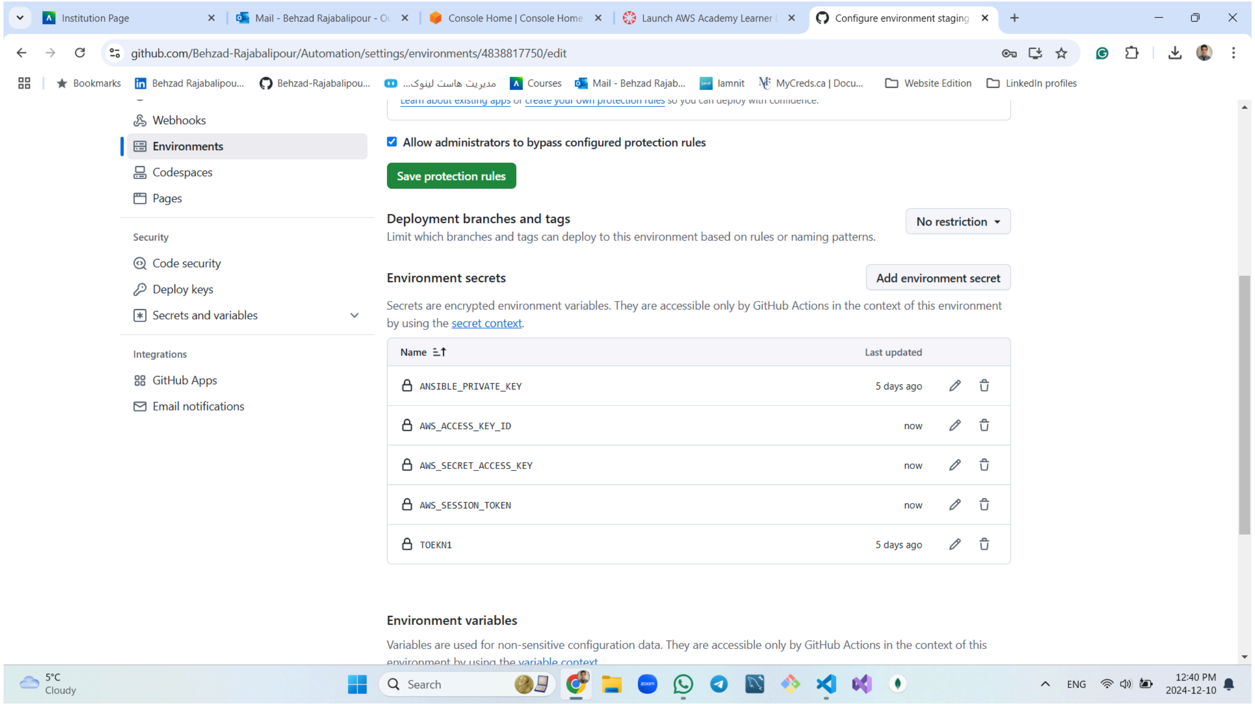
Ansible playbook tasks to install httpd

```yaml
- name: Generate index.html from jinja2 template and copy to the remote host
  template: src={{ source_file }} dest={{ dest_file }}/index.html mode=0555
  notify: Restart Httpd
  when: ansible_os_family == "RedHat"

- name: Start Apache Web Server
  service: name=httpd state=started enabled=yes
  when: ansible_os_family == "RedHat"


andlers:
 name: Restart Httpd
 service: name=httpd state=restarted
 when: ansible_os_family == "RedHat"
```

Generate a webpage using jinja2 and other configurations to the webpage

GitHub actions

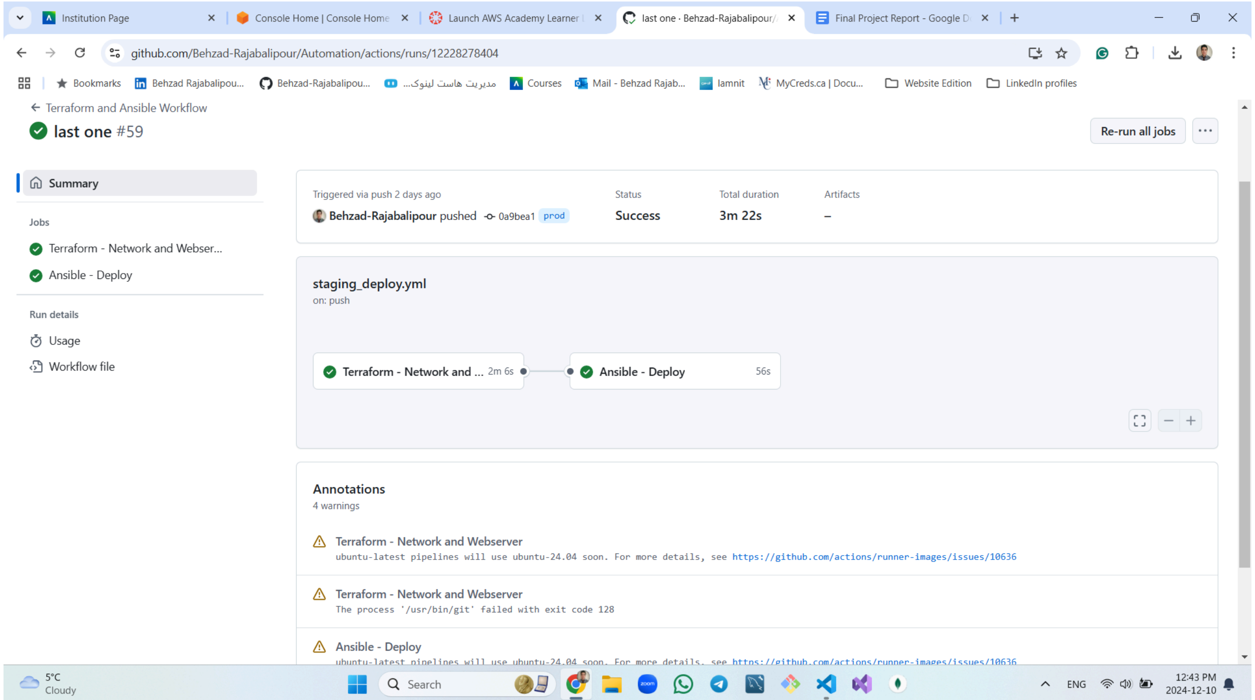
```yaml
jobs:
  terraform:
    name: "Terraform - Network and Webserver"
    runs-on: ubuntu-latest
    environment: staging
    env:
      GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      AWS_SESSION_TOKEN: ${{ secrets.AWS_SESSION_TOKEN }}
```



Providing the access credentials to GitHub

```yaml
# --- Network Infrastructure ---
- name: Terraform Format (Network)
  id: fmt-network
  run: terraform fmt
  working-directory: ./Terraform/prod/Network
  continue-on-error: true

- name: Terraform Init (Network)
  id: init-network
  run: terraform init
  working-directory: ./Terraform/prod/Network

- name: Terraform Validate (Network)
  id: validate-network
  run: terraform validate -no-color
  working-directory: ./Terraform/prod/Network

- name: Terraform Plan (Network)
  id: plan-network
  run: terraform plan -input=false -no-color -out=tf.plan
  working-directory: ./Terraform/prod/Network

- name: Terraform Apply (Network)
  id: apply-network
  run: terraform apply -input=false tf.plan
  working-directory: ./Terraform/prod/Network
```

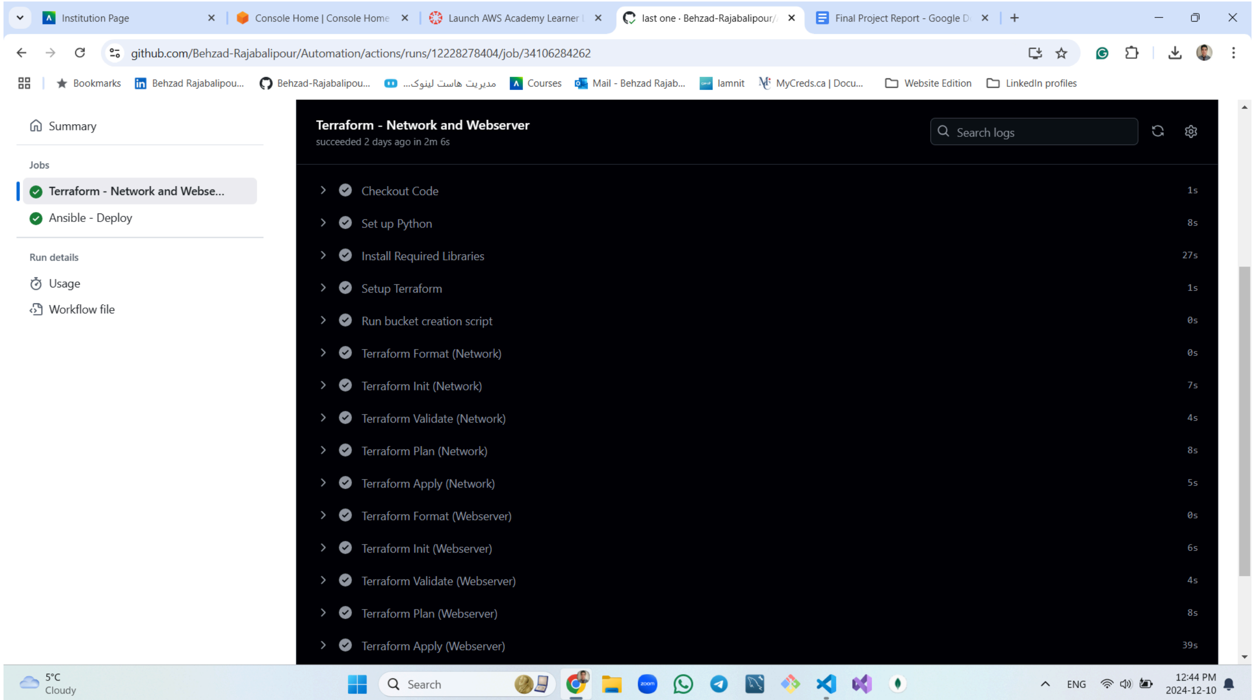
sequence of steps to format, initialize, validate, plan, and apply Terraform configurations for the network infrastructure



Picture shows Both Terraform and Ansible part installed correctly

```yaml
# --- Webserver Infrastructure ---
- name: Terraform Format (Webserver)
  id: fmt-webserver
  run: terraform fmt
  working-directory: ./Terraform/prod/Webserver
  continue-on-error: true

- name: Terraform Init (Webserver)
  id: init-webserver
  run: terraform init
  working-directory: ./Terraform/prod/Webserver

- name: Terraform Validate (Webserver)
  id: validate-webserver
  run: terraform validate -no-color
  working-directory: ./Terraform/prod/Webserver

- name: Terraform Plan (Webserver)
  id: plan-webserver
  run: terraform plan -input=false -no-color -out=tf.plan
  working-directory: ./Terraform/prod/Webserver

- name: Terraform Apply (Webserver)
  id: apply-webserver
  run: terraform apply -input=false tf.plan
  working-directory: ./Terraform/prod/Webserver
```

sequence of steps to format, initialize, validate, plan, and apply Terraform configurations for the webserver infrastructure



Github Action workflow deployed both Network and Webserver directory

```yaml
ansible:
  name: "Ansible - Deploy"
  runs-on: ubuntu-latest
  needs: terraform
  environment: staging

  env:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
    AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
    AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    AWS_SESSION_TOKEN: ${{ secrets.AWS_SESSION_TOKEN }}

  steps:
    - name: Checkout Code
      uses: actions/checkout@v3

    - name: Set up Python (for Ansible)
      uses: actions/setup-python@v2
      with:
        python-version: "3.10.8"

    - name: Install Required Libraries
      run: |
        python -m pip install --upgrade pip
        pip install ansible==2.9 boto3 botocore "Jinja2<3.1"

    - name: Verify Inventory File
      run: |
        cat ./ansible/inventories/aws_ec2.yaml

    - name: Add SSH Key
    # it will store in the GitHub Action Ubuntu server
```

GitHub Actions workflow to deploy infrastructure using Ansible

The steps here include setting up Python and installing required libraries verifying the Ansible inventory and SSH key which will be used to access the targeted EC2 instances for configuration

```
Source: github-workflow.json
- name: Run Ansible Playbook
  run: |
    ansible-playbook -i ./ansible/inventories/aws_ec2.yaml ./ansible/playbook_jinja2.yaml
  env:
    ANSIBLE_HOST_KEY_CHECKING: "false"
```

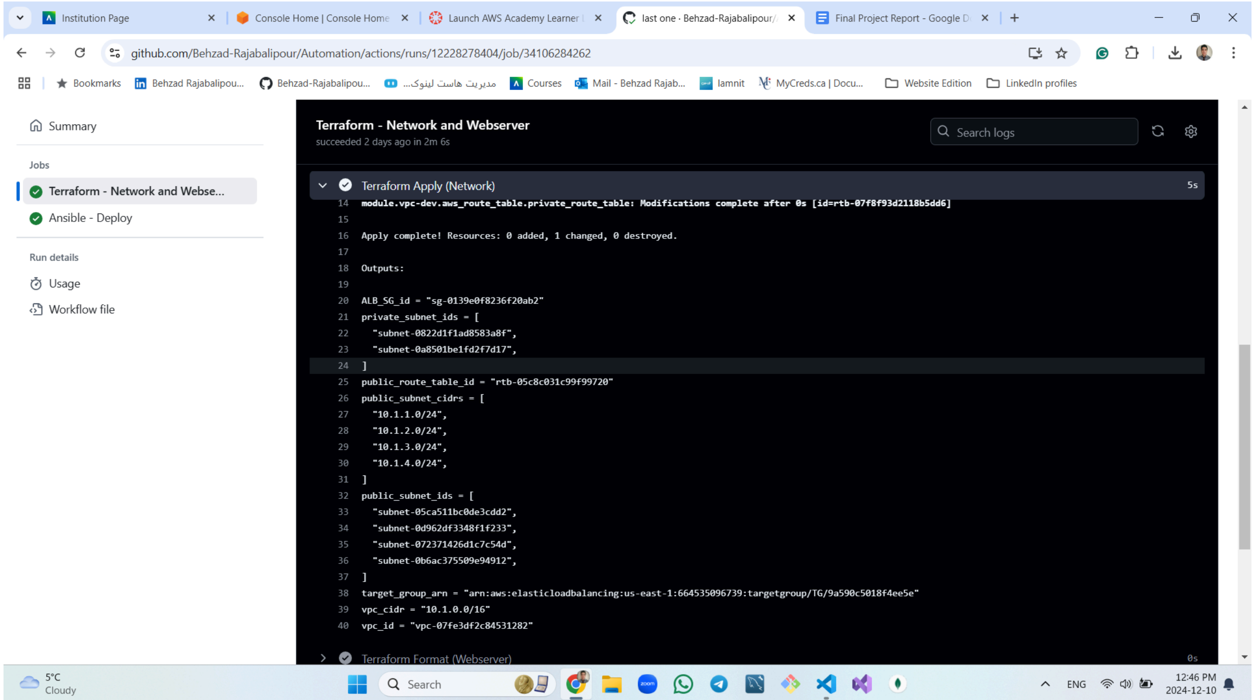
Run the Ansible playbook

Deployment

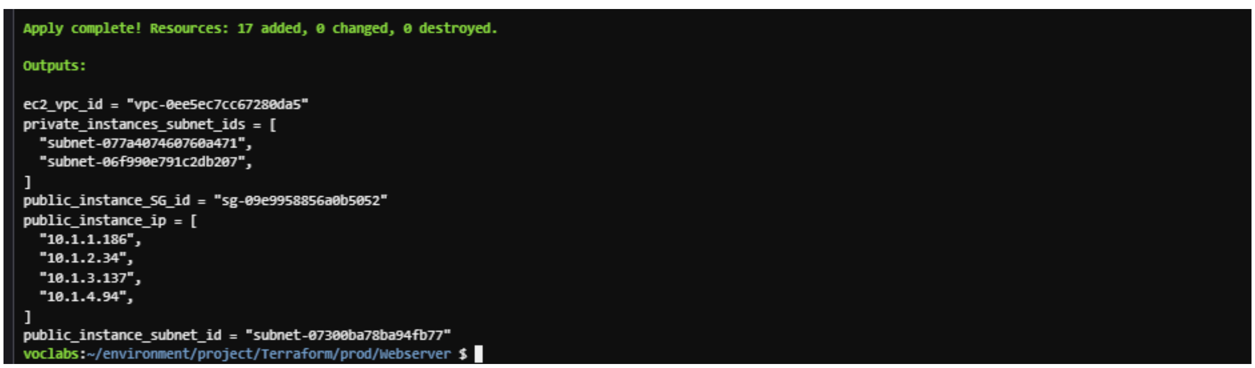
Here are some screenshots of the successful deployment of the website

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

Outputs:

ALB_SG_id = "sg-007153cab24d1ede1"
private_subnet_ids = [
  "subnet-077a407460760a471",
  "subnet-06f990e791c2db207",
]
public_route_table_id = "rtb-0278ff3538f0c061f"
public_subnet_cidrs = [
  "10.1.1.0/24",
  "10.1.2.0/24",
  "10.1.3.0/24",
  "10.1.4.0/24",
]
public_subnet_ids = [
  "subnet-0d5e170f2aa801f89",
  "subnet-07300ba78ba94fb77",
  "subnet-08ad85a07b606657f",
  "subnet-09e45886882ae142f",
]
target_group_arn = "arn:aws:elasticloadbalancing:us-east-1:142390386045:targetgroup/TG/e42fed72659dbce7"
vpc_cidr = "10.1.0.0/16"
vpc_id = "vpc-0ee5ec7cc67280da5"
voclabs:~/environment/project/Terraform/prod/Network $
```

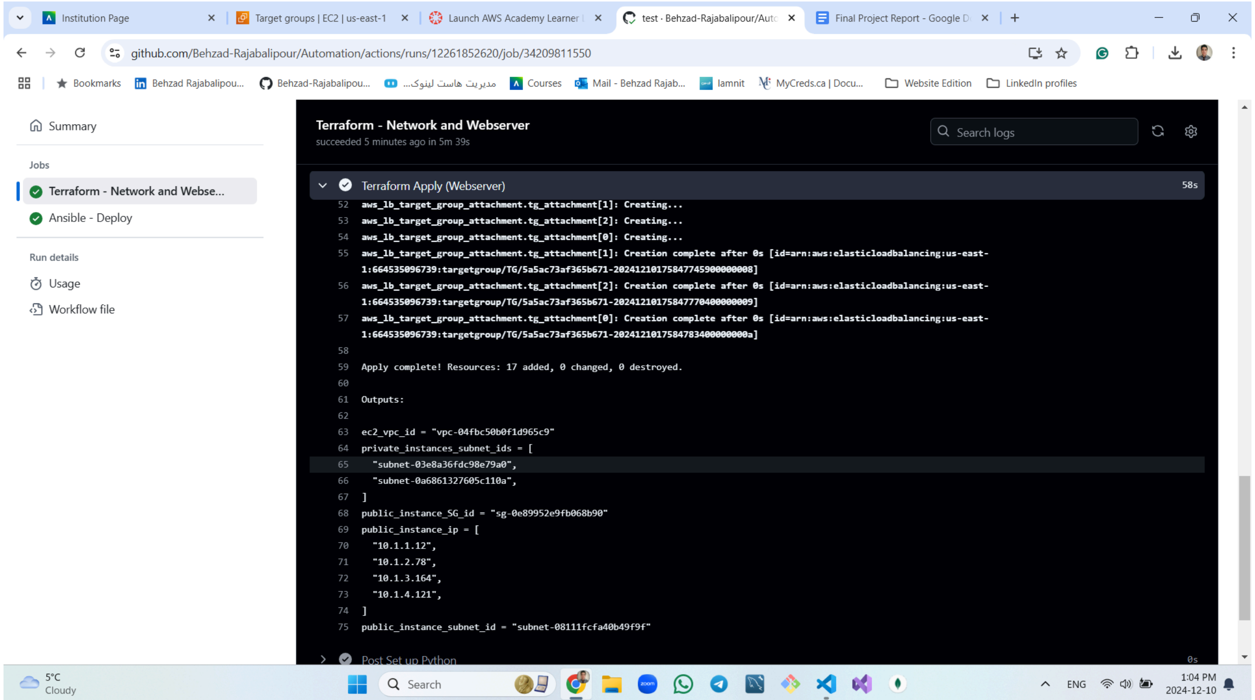
Networking deployment in Terraform



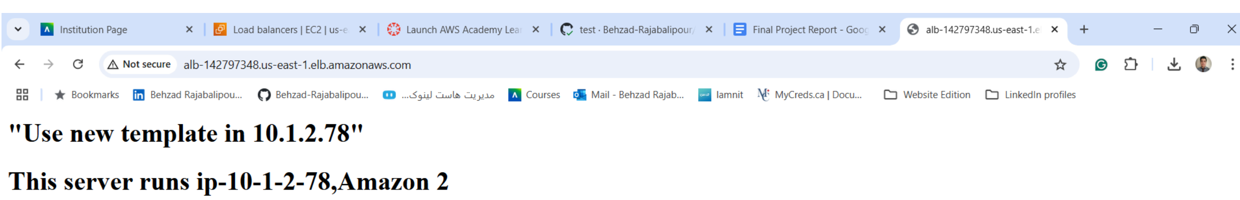
Networking deployment in GitHub Action



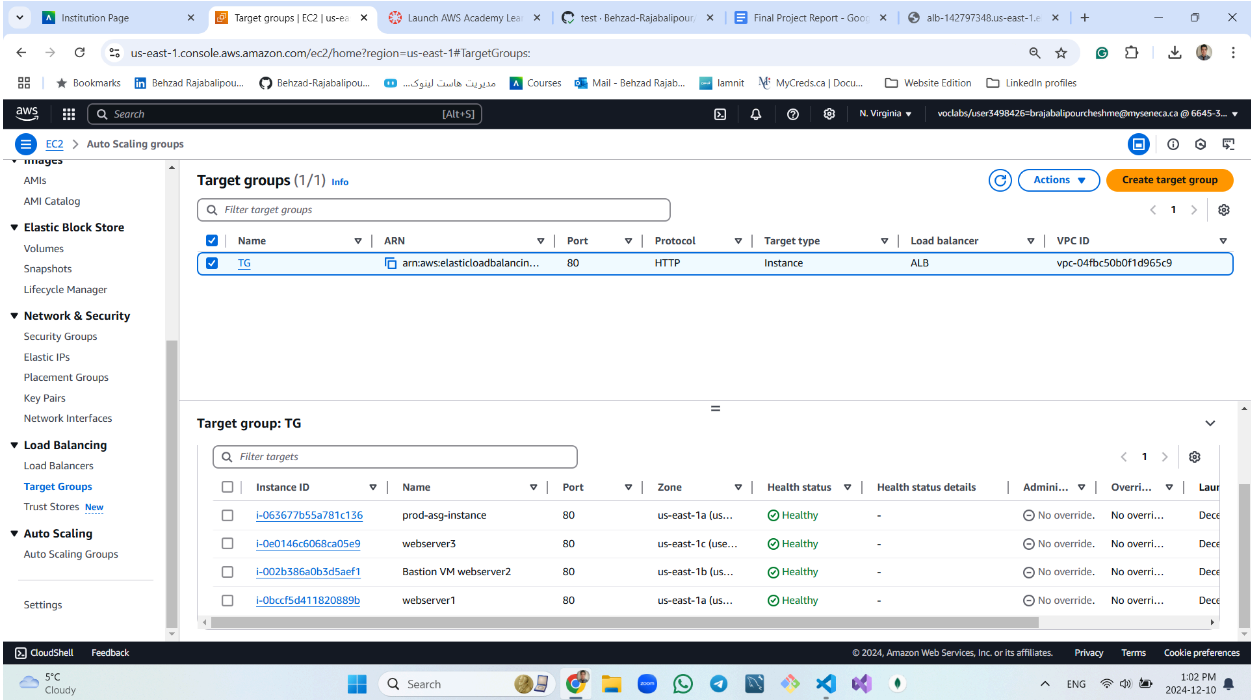
Web server deployment(EC2 and installing httpd) in Terraform



Web server deployment(EC2 and installing httpd) in Anisble

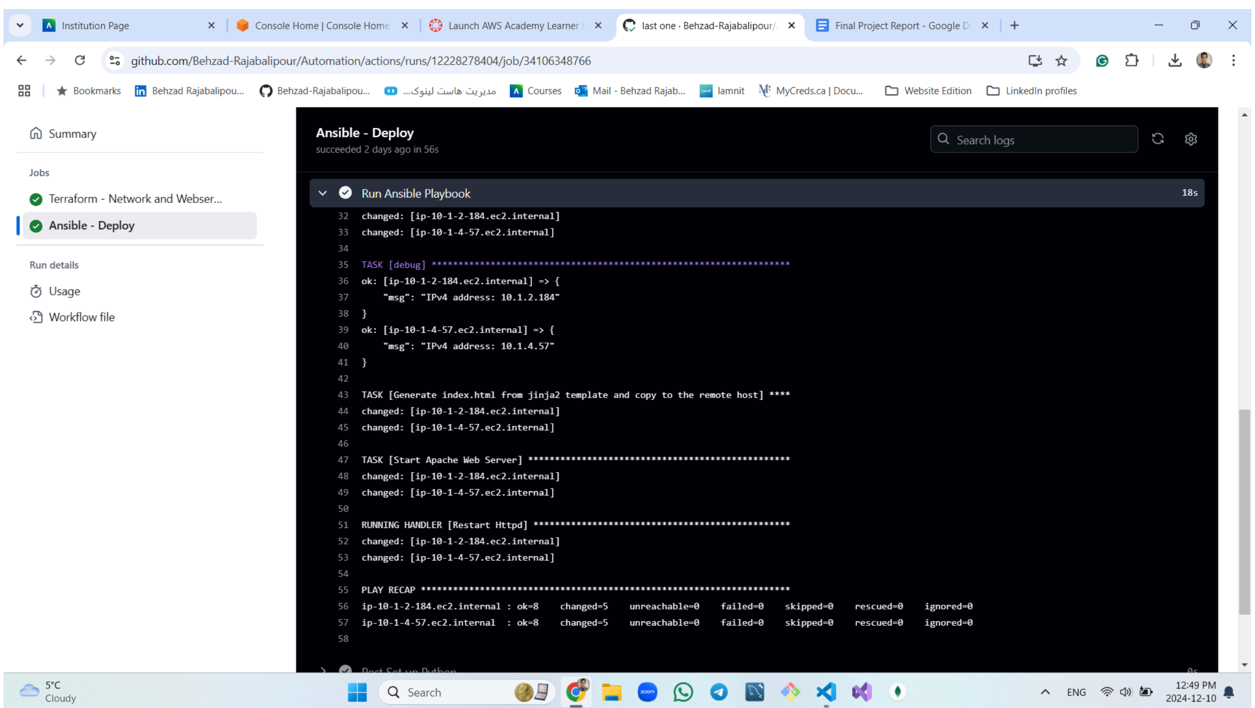


This is Webserver2 (Bastion Host) which installed http with Ansible

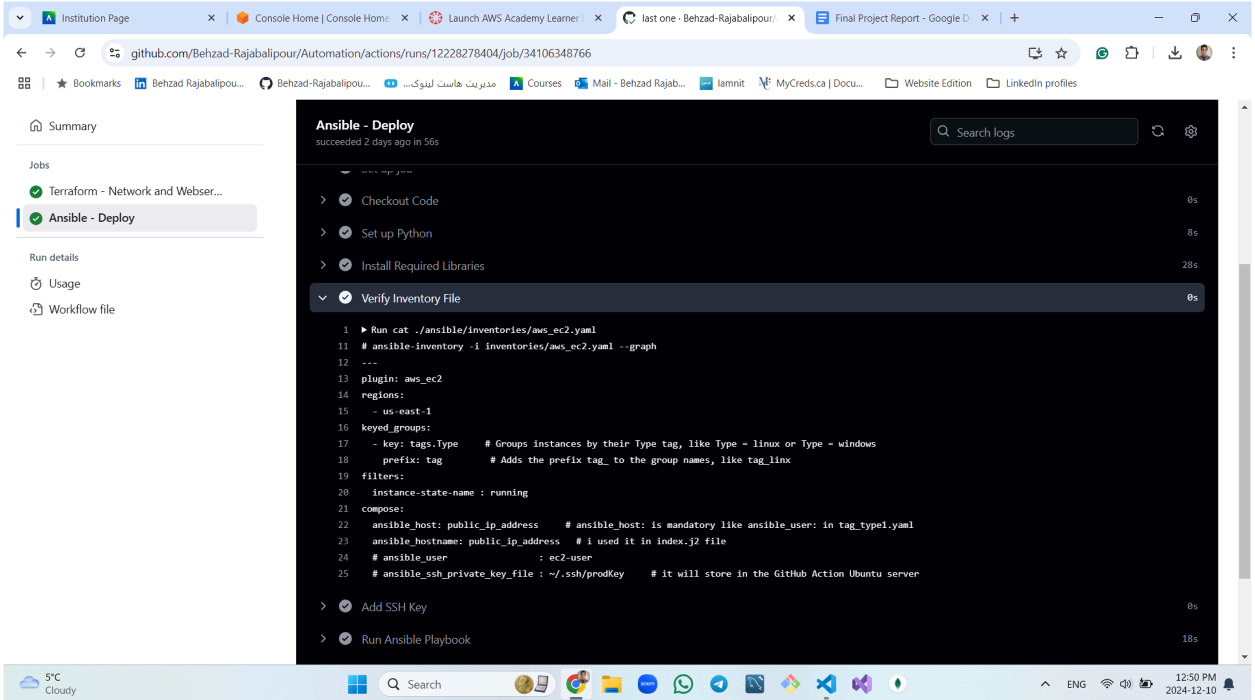


The target group at the final step, shows all 3 instances are healthy. One more instance (prod-asg-instance) which is created by ASG

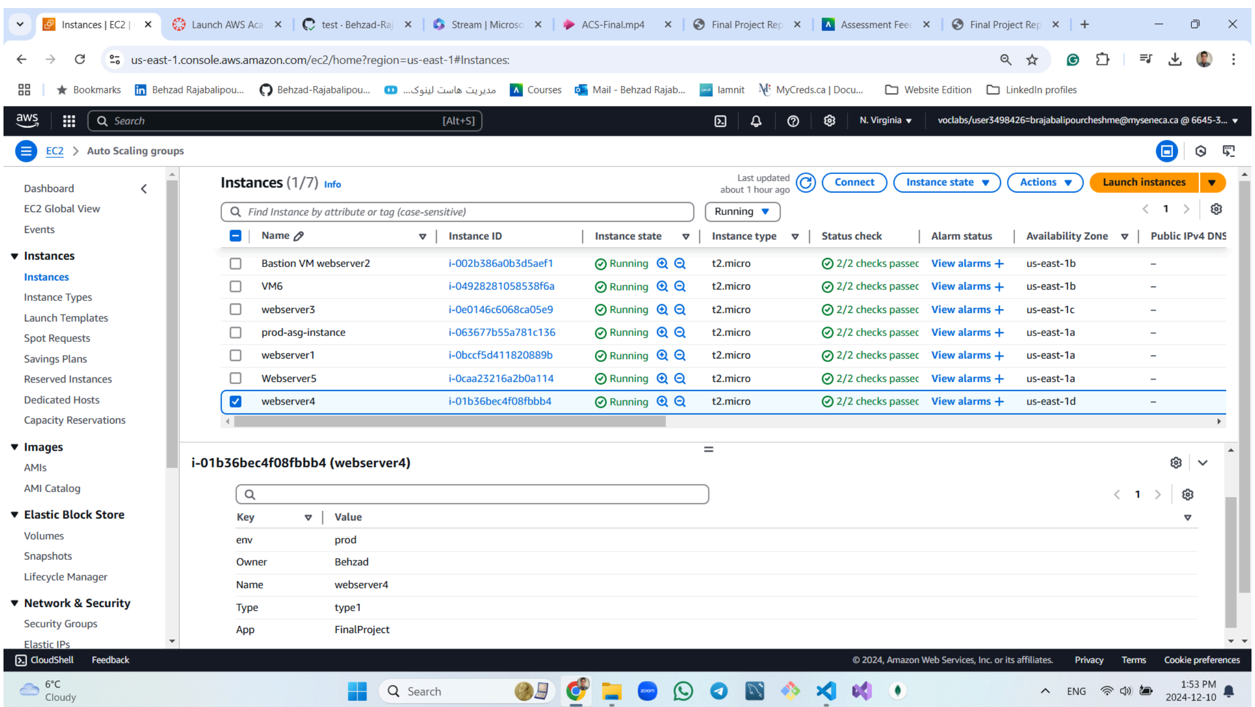
Accessing multiple instances via the Load balancer



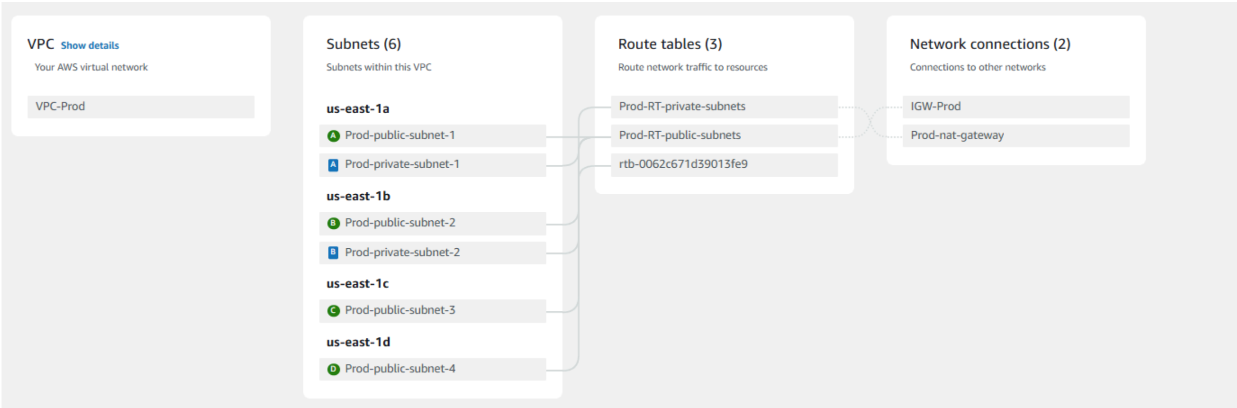
Webserver deployment in GitHub workflow (Run Ansible)



Ansible-Inventory file



4 instances in public subnets, 2 instances in private subnets



VPC with 4 availability zones

Challenges

The main challenge was coordination between the team members. As each step is dependent on the previous step, we had to wait and then build the next based on what we created. GitHub should have streamlined this process but with the limited time we had, it added more overhead. We eventually decided to divide up the project into three main steps: Terraform, Ansible, and GitHub actions. The real challenge came in integrating everything which felt like solving a puzzle.

One more challenge we got was about ASG. The Auto Scaling Group would create a new instance without considering how many instances are already inside the Target Group. We even used the same templates for webservers but ASG still create a new instance without considering there were enough instances inside the Target group.

Conclusion

In conclusion, this project demonstrates the seamless integration of Terraform, Ansible, and GitHub Actions to automate the deployment and configuration of a highly available website. Terraform for infrastructure provisioning, Ansible for server configuration, and GitHub Actions for continuous integration