Week 4 Lab

# Terraform Modules, Functions, Conditions

# Deployment of Fault-Tolerant Web Application with Terraform

## The Goals

In this lab, we will firstly introduce more Terraform features, such as Terraform functions, conditions, and state manipulation commands. (Tasks 1-3)

As a next step, we will use all the knowledge acquired so far to improve the Terraform config we created in Week 3. We will deploy a static webserver on Amazon EC2.

We will continue to improve the reusability, modularity, and extensibility of our Terraform code. In addition to Terraform improvements, we will move away from the use of Default VPC and will create our own Virtua Private Cloud (VPC) in AWS and make our web site deployment fault tolerant.

The modularity of our code will support deployment into multiple environments, such as Test and Prod.

## Task 1: Use of functions and conditions – Support of Multiple Environments with Terraform

## Week 3 Architecture

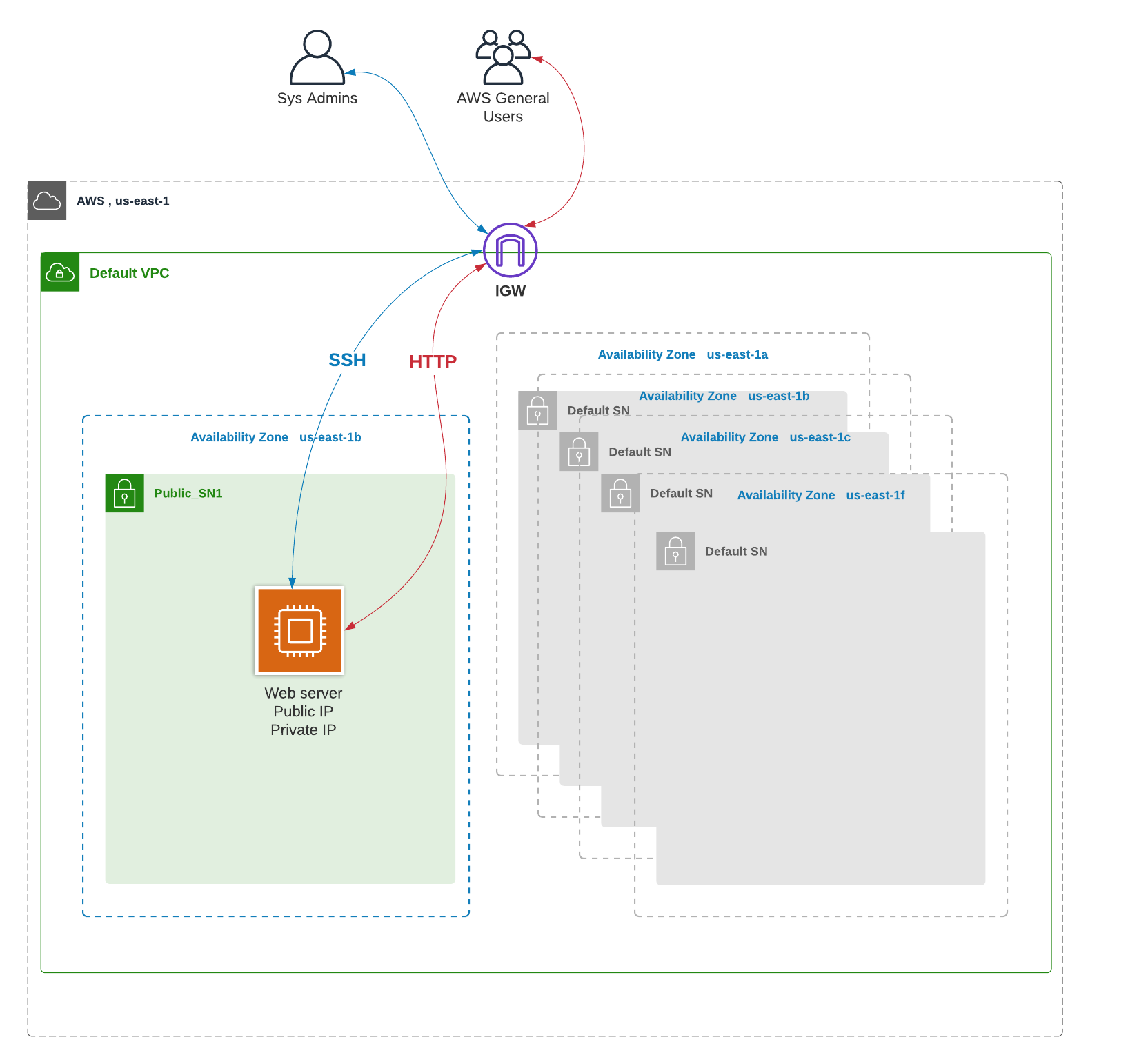


Figure 1Week 3 Architecture

Start with the code provided in BB, Learning Content => Week 4 => Lab =>week4-terraform\_code => Session 1 => week4\_task1\_start.

Our goal in Task 1 is to use the implement different deployments of our infrastructure into Test and Prov environments. We will use Terraform input parameters and conditions to create code that supports deployments in Prod and Test environments.

Prod environment is more evolved and has the requirements below:

* Root volume is encrypted in Prod only
* Additional data volume is deployed in Prod only
* EC2 instance type is t2.medium; all the other environments should be using t2.micro

**Note**: you can see the solution in week4\_task1\_result yet I recommend you build it out by yourself

1. Create an S3 bucket to store the remote Terraform state. The bucket should be created manually. Update config.tf files to reference this bucket.

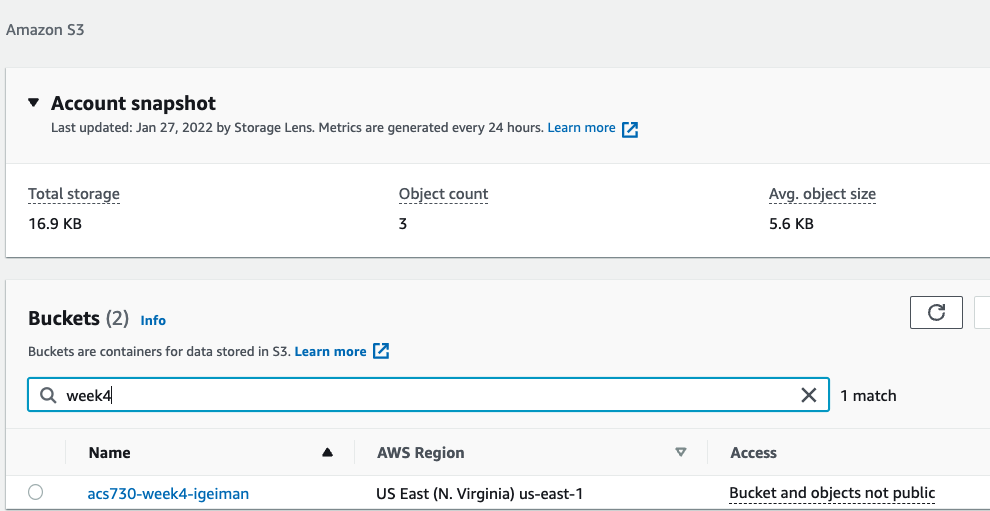


Figure 2 S3 bucket to store the state

1. Change directory to week4\_terraform\_code/Session 1/01\_task1\_start/01-Networking folder and create a new public subnet in default VPC.

cd week4\_terraform\_code/Session 1/01\_task1\_start/01-Networking

terraform int

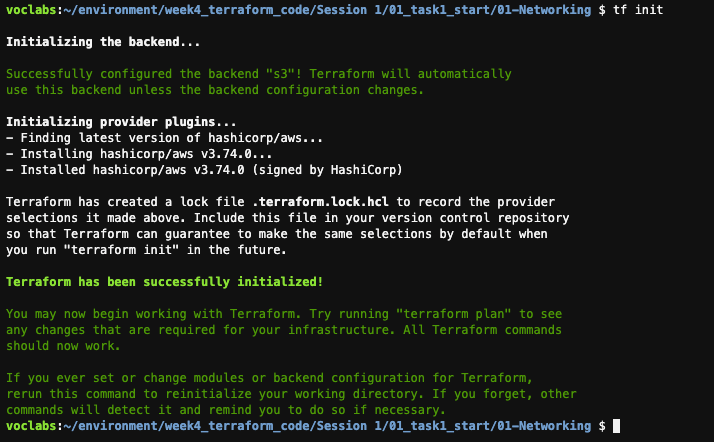


Figure 3 Terraform init in 01-Networking

terraform validate

terraform plan

terraform apply –auto-approve

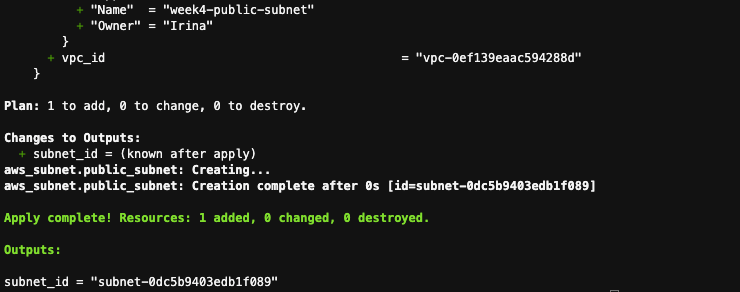


Figure 4 Results of terraform apply

1. Change directory to week4\_terraform\_code/Session 1/01\_task1\_start/01-Webserver and deploy the EC2 instance into the subnet we just created. Do not forget to update the bucket name in the file and to create a new ssh key pair for week4.

cd ../02-Webserver

ssh-keygen -t rsa -f week4

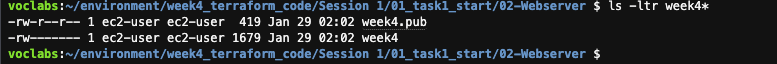


Figure 5 List files in the current folder

terraform int

terraform validate

terraform plan

terraform apply –auto-approve

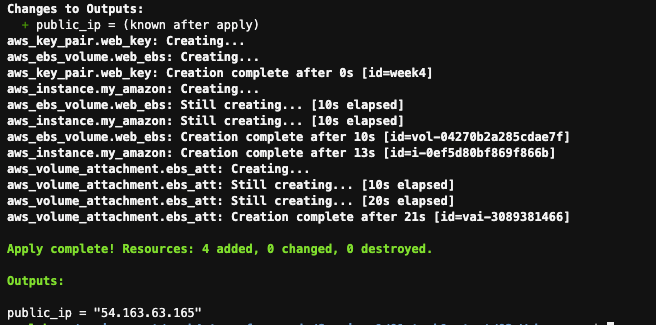


Figure 6 EC2 instance deployment output

1. Remain in the 01-Webserver folder and update the variable.tf and main.tf files to signal current environment (prod/test).   
   **Note:** reference the code in the 02\_task1\_completed folder to see the final implementation of this task.
2. Add “env” variable to variable.tf and add conditional deployment of EBS (Elastic Block Storage) volume as specified below.

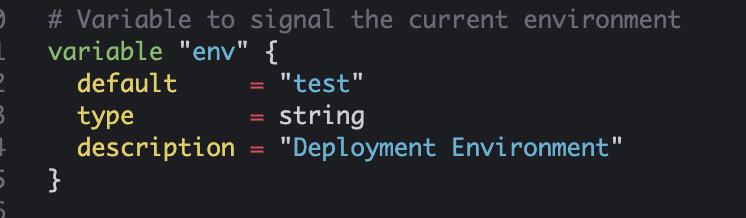


Figure 7 Define "env" variable to specify current environment



Figure 8 Add condition to EBS volume deployment

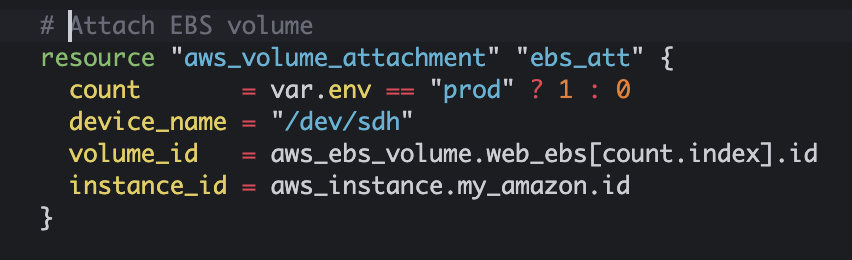


Figure 9 Add condition to EBS volume attachment

Conditional encryption of root volume:

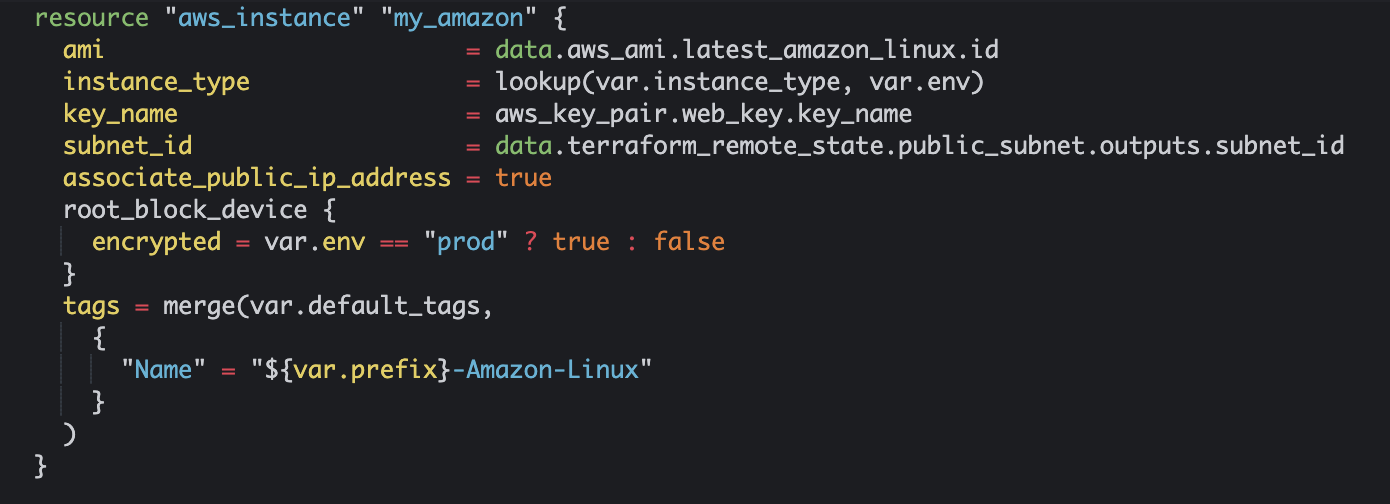
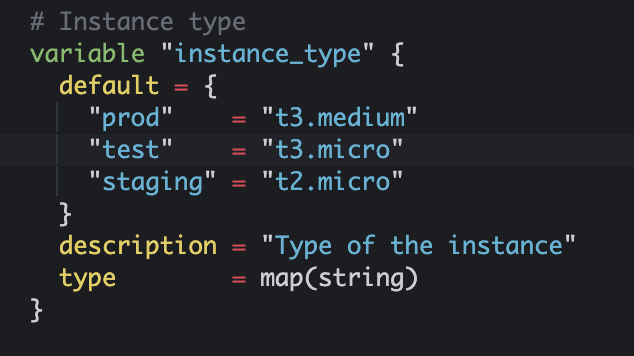


Figure 10 Add condition to root volume encryption

1. Add a map with the instance size per environment to variable.tf



Update main.tf to resolve instance type dynamically using lookup function.

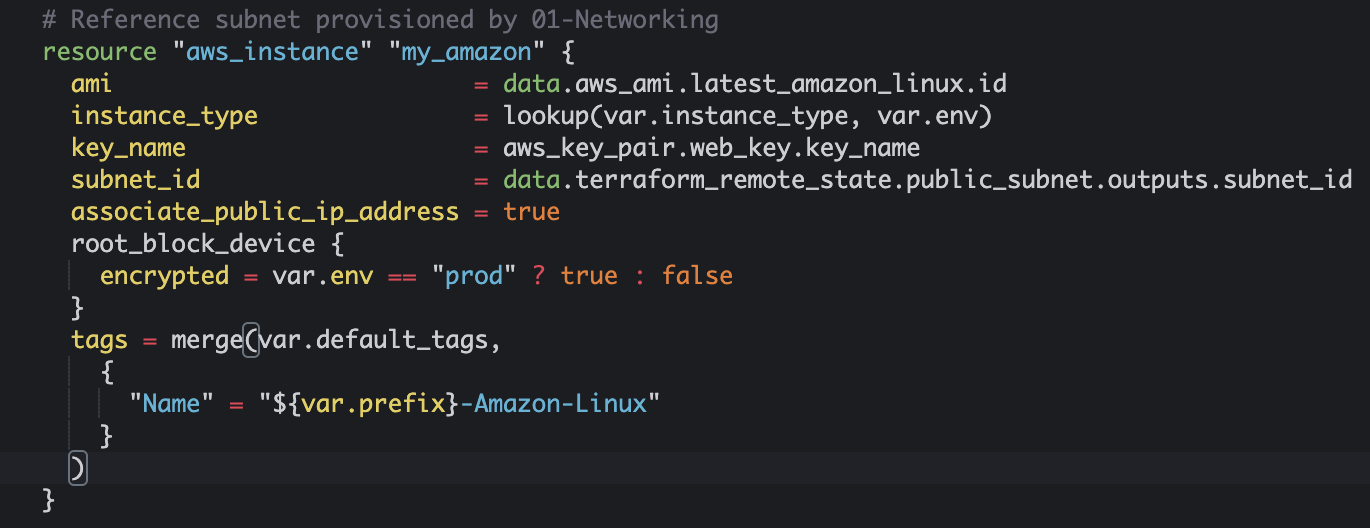


Figure 11 Specify map variable of instance type per environment

Validate and test your code with the commands below

terraform int

terraform validate

terraform plan –var env=test

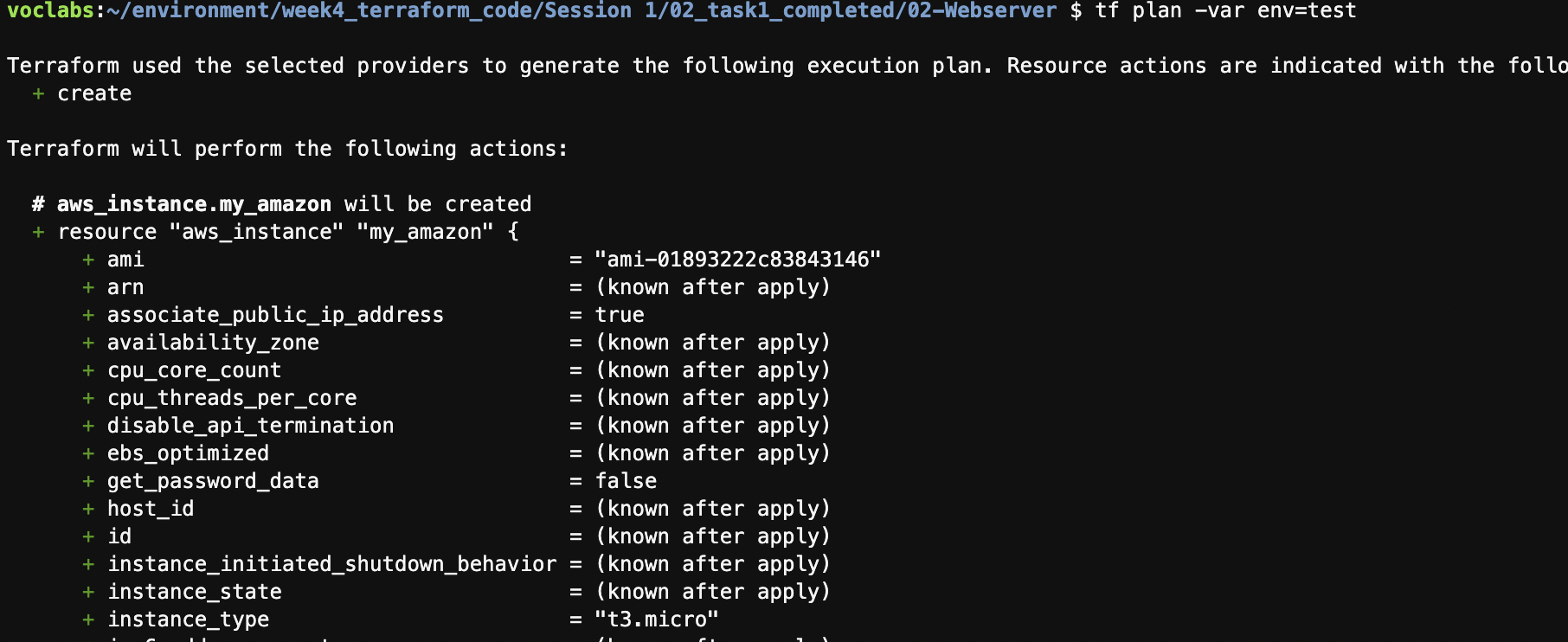


Figure 12 Terraform plan for test environment, instance type t3.micro

terraform plan –var env=prod

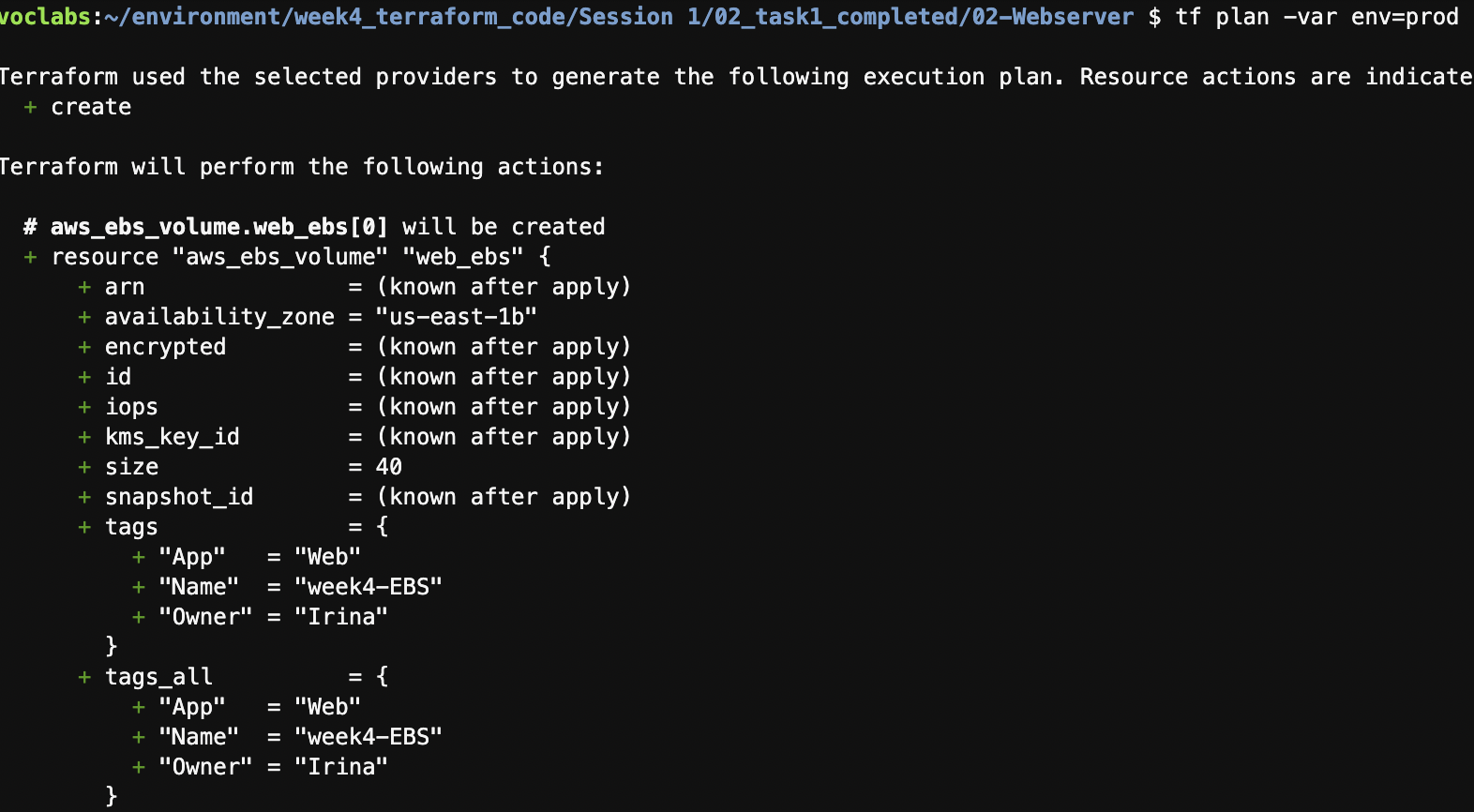


Figure 13 Terraform plan for prod environment, instance type t3.medium and extra volume attached, root volume encrypted

**Do you see the difference between planned deployment of “dev” and “prod” instances? What other changes would you have implemented?**

Deploy the “test” instance.

terraform apply –var env=test

Verify the instance has one EBS volume only.

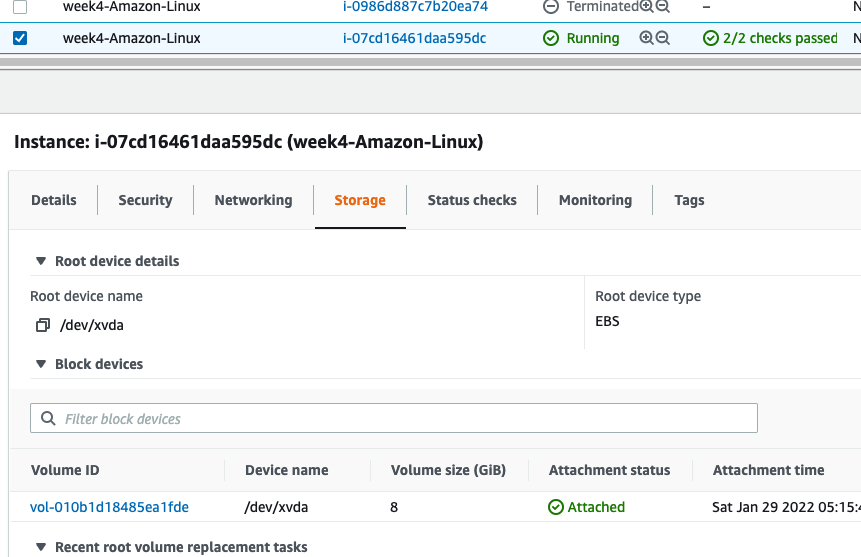


Figure 14 One EBS volume in “test” deployment

Deploy the “prod” instance.

terraform apply –var env=prod

Verify the instance has two EBS volumes.

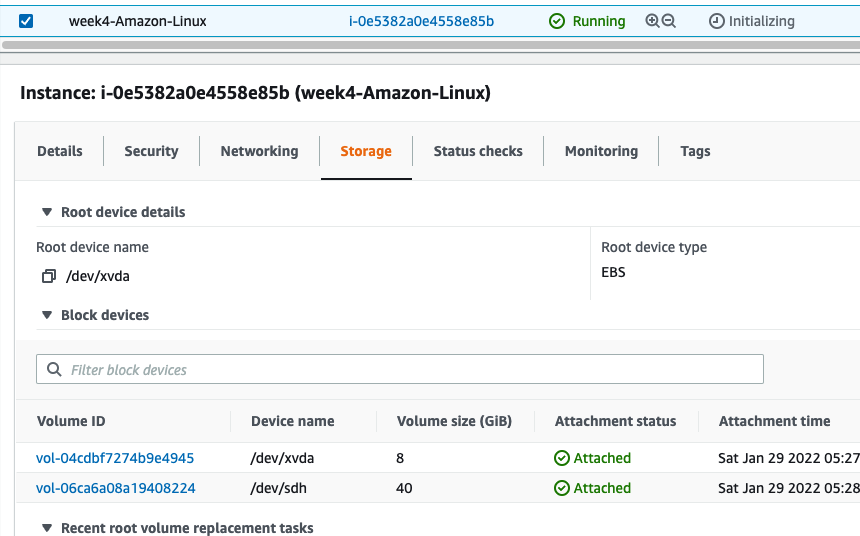


Figure 15 Two EBS volumes in "prod" deployment

1. Improve readability of your environment and add an “env” tag using local variables and re-deploy.

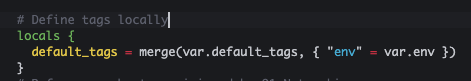


Figure 16 Define tags locally

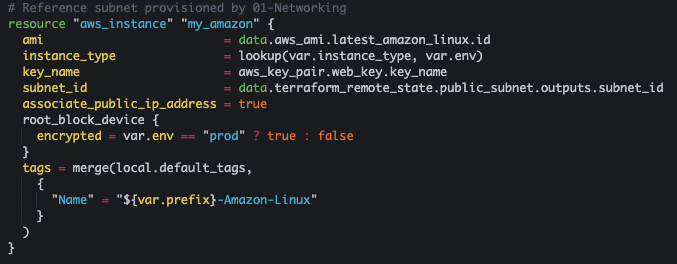


Figure 17 Reference local tags

## Task 2: Using “user\_data” and Terraform provisioners to install static website

In this task we will explore the use of “user\_data” section of “aws\_instance” Terraform and local-exec provisioner to install static web server on our EC2 instance.

1. Add a user\_data block to the EC2 deployment. The script in the block should update the package manager, install httpd webserver, start and enable the server.

Text

Description automatically generated

Figure 18 Inline user\_data block

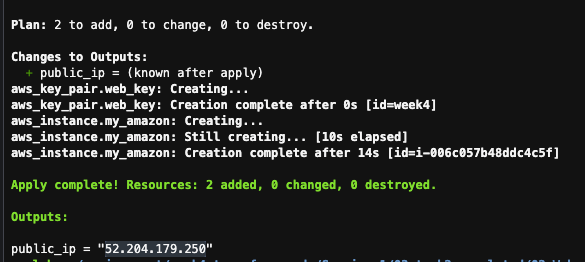


Figure 19 Note the public IP

1. Deploy the security group to allow ingress on ports 80 and 22 from everywhere.

Update 01-Networking Terraform configuration “output.tf” to expose VPC id.

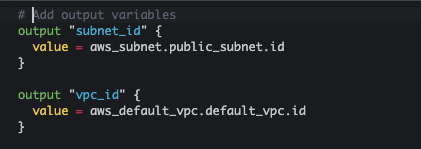


Figure 20 Add VPC id to output.tf in 01-Networking module

Redeploy the 01-Netwotking module. Did any of the resources get re-deployed?

cd ../01-Networking/

terraform plan

terraform apply

Add security group to your EC2 instance and redeploy 02-Webserver code. Notice the use of “terraform\_remote\_state” in the configuration below.



Figure 21 Security Group Definition, notice the use of the outputs

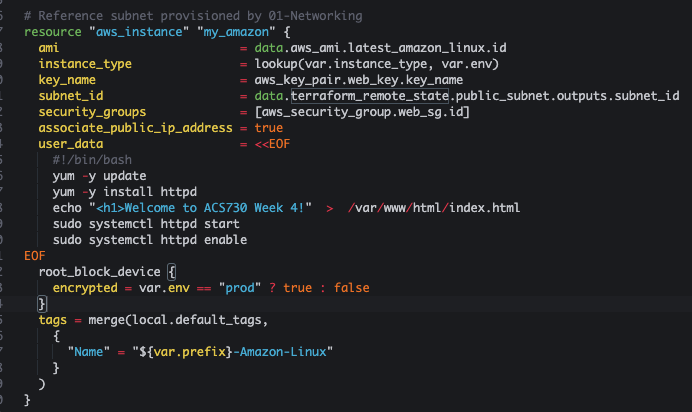


Figure 22 Add Security Groups reference to "aws\_instance" resource

cd -

terraform plan

terraform apply

1. Log into deployed EC2 instance and examine the logs related to the user\_data script execution.

ssh -i week4 [ec2-user@3.80.102.168](mailto:ec2-user@3.80.102.168)

sudo vi /var/log/cloud-init-output.log # Check initialization logs

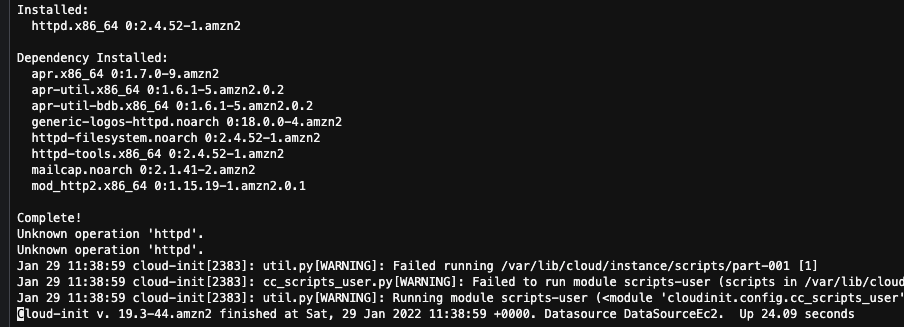


Figure 23 Is httpd installed?

sudo systemctl httpd status $ Is httpd running?

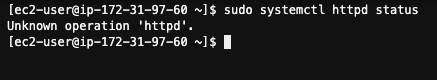


Figure 24 Is httpd running?

sudo rpm -q httpd # Is httpd installed?

Is httpd installed?

Figure 25 Is httpd installed?

sudo systemctl status cloud-final # What happen to the service?

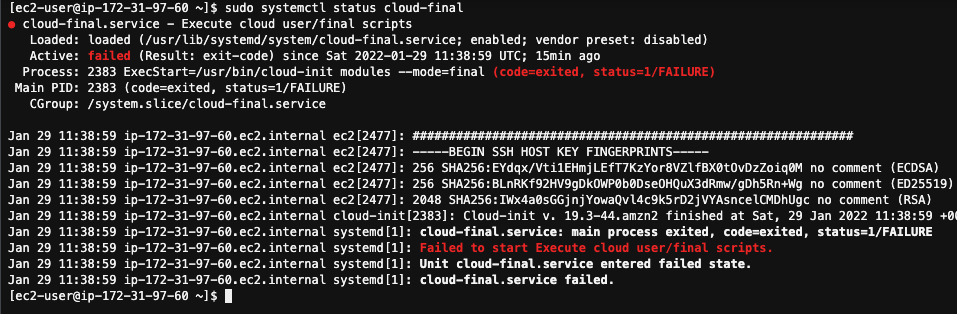


Figure 26 Cloud-final service failed to start

sudo journalctl -u cloud-final # Check service logs

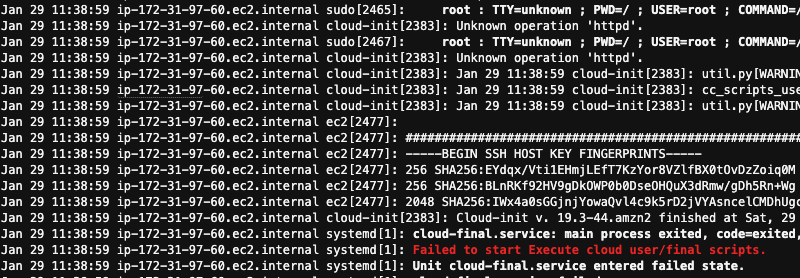


Figure 27 Linux service logs, note the “Unknown operation ‘httpd’”

1. Reproduce the problem locally on EC2.

sudo systemctl httpd start

Service start command errors out

Figure 28 Service start command errors out

1. Fix the service start problem

sudo systemctl start httpd

sudo systemctl status httpd

curl localhost

Text

Description automatically generated

Figure 29 Webservice started successfully



Figure 30 Webserver in the Web browser

1. We created configuration drift! Our deployment configuration is different than our local configuration on EC2. Let us fix it!

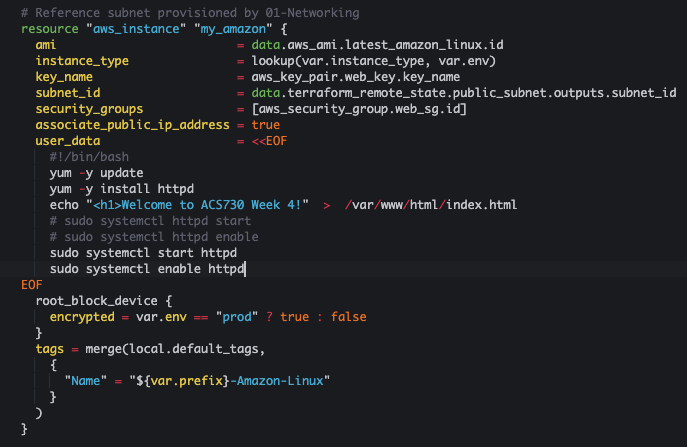


Figure 31 Fix the systemctl commands

**Important**! Even though we changed configuration script only, the EC2 instance is fully redeployed! Is it a good thing in production?

Mark the new public IP of the EC2 instance and see if the webserver is still available.

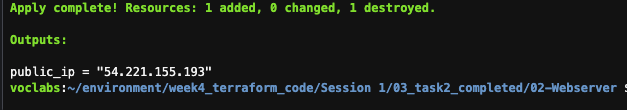


Figure 32 Instance redeployed with a new public IP

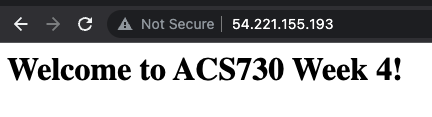


Figure 33 Web server is running

1. Refactor our “user\_data” section to reference file instead of the inline code.

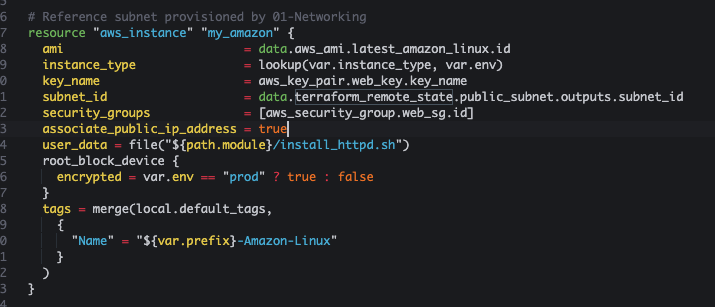


Figure 34 Replace inline code with the file reference

**Important** We changed the message in static HTML only, yet the instance is redeployed. Is it a good thing in production?

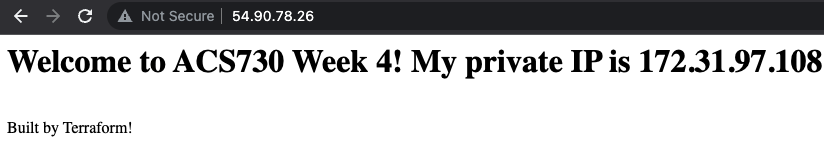


Figure 35 Updated message

1. Destroy all the infrastructure we have provided so far. Task 3 will start from the beginning.

terraform destroy –auto-approve

cd ../01-Networking

terraform destroy –auto-approve

## Task 3: Build a Webserver with Near Zero Downtime

In this task we will address a few out of the many challenges presented by the Week3 architecture and our implementation in Task2.

As we have seen already, every change in the static website configuration will result in the instance redeployment. It will render our site out of service until the new instance is created and configured. Moreover, every time the instance is redeployed or rebooted, the public IP of our site changes. It means that even with the DNS defined for our site, users will not be able to access it for an extended period.

Lastly, we can improve the security of our solution by creating a custom VPC in place of AWS provided VPC (we called it default VPC).

This task starts with the code of 03\_task2\_completed and progresses towards 01\_task3\_completed. You can use 01\_task3\_completed code as a reference throughout this workshop yet I highly recommend building out the solution and instead of deploying the provided code.

The solution is parametrized to support dev and prod deployments.

## Week 4 Architecture

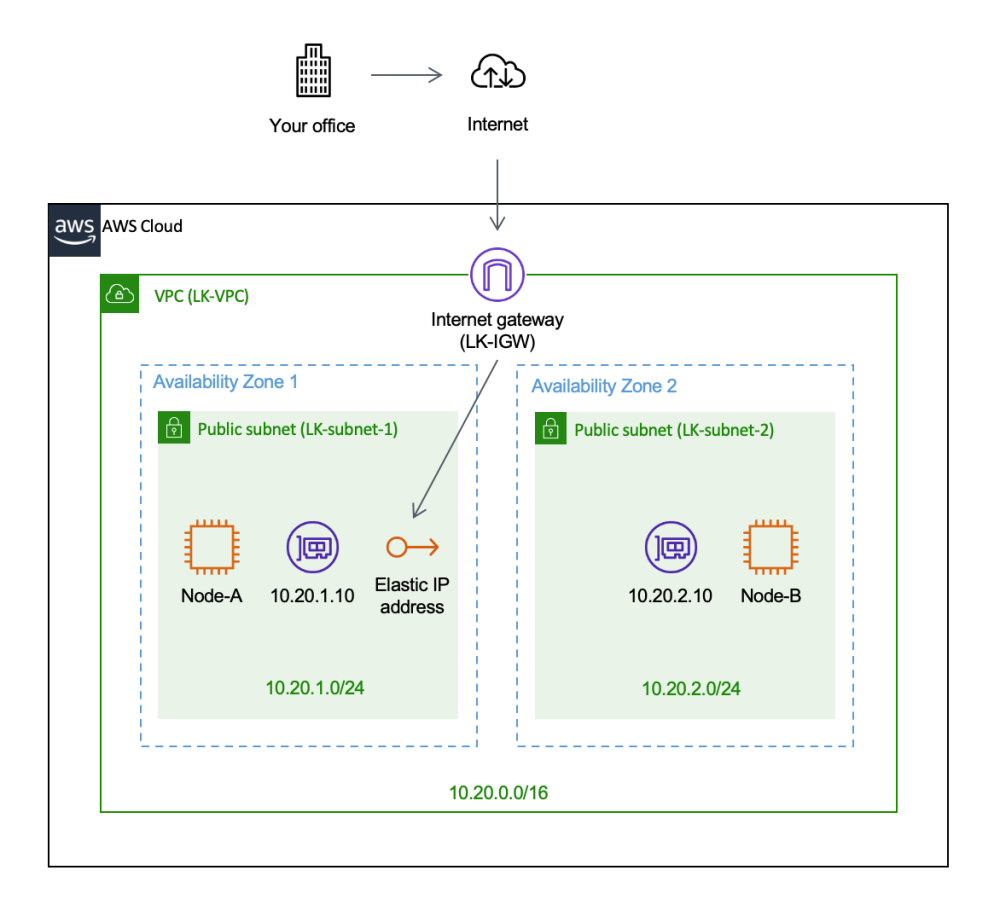


Figure 36 Fault tolerant design

In this task we will deploy all the underlying infrastructure required to host our static web application. The hosting solution will support near-zero downtime application’s updates and rollbacks. The web site will be hosted on a standalone AWS EC2 in a public subnet with the Elastic IP attached to the instance.

We will reuse the S3 bucket and ssh key we created in Task 1.

1. Add input variable for VPC CIDR range and adjust the rest of 01-Networking code accordingly.

Add input parameters to create a custom VPC and 2 subnets in this VPC. Note a new CIDR range 10.20.0.0/16.

Text

Description automatically generated

Figure 37 Add VPC CIDR to variables.tf and create a list of subnet CIDRs

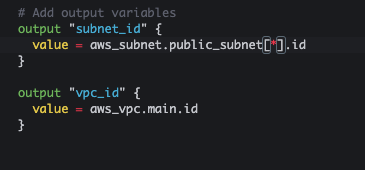


Figure 38 Modify Output variables in output.tf

Modify the main.tf to create the custom VPC and reference it in the subnets block.

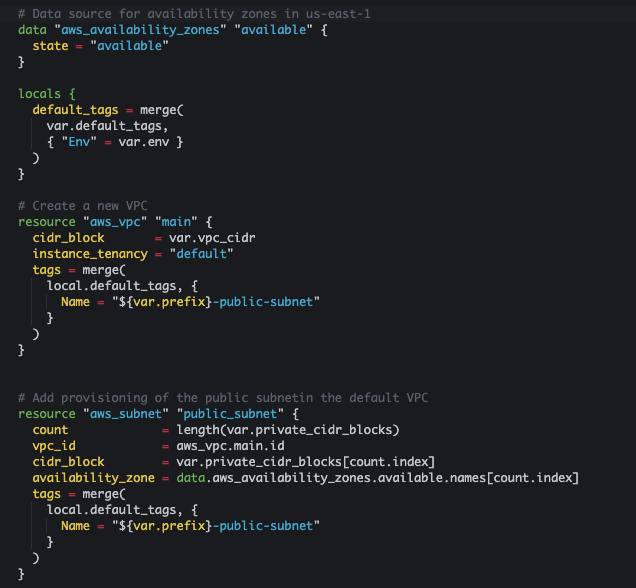


Figure 39 Modify main.tf in 01-Networking

Run the commands below:

terraform init

terraform plan

terraform apply

The output of the 01-Networking module deployment

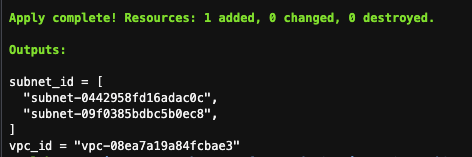


Figure 40 The output of the 01-Networking module deployment

1. Modify the 02-Webserver. Create an internet gateway. Create an Elastic IP and assign it to EC2 instance, remove public IP association, reflect the fact that private subnet is now a list.

cd ../02-Webserver/



Figure 41 Create Internet Gateway



Figure 42 Add Elastic IP to EC2 Instance

Text

Description automatically generated

Figure 43 Remove public IP association, add index to the public subnets reference and lifecycle.

Run the commands below

terraform init

terraform validate

terraform plan

terraform apply

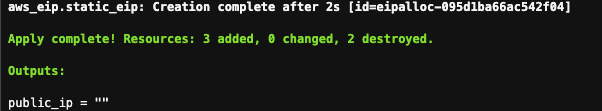


Figure 44 No public IP attached

The instance is created successfully but there is no public IP. What is the problem?

Update output.tf to reflect the EIP.

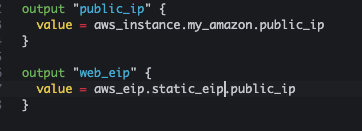


Figure 45 Add EIP to output.tf

Redeploy your Terraform config.

terraform apply

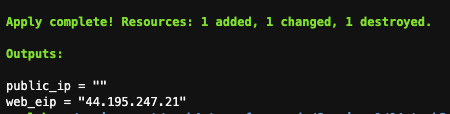


Figure 46 EIP was printed out

1. Debugging connectivity to EC2 instance using AWS SSM Session Manager.

No connectivity over port 80

Figure 47 No connectivity over port 80

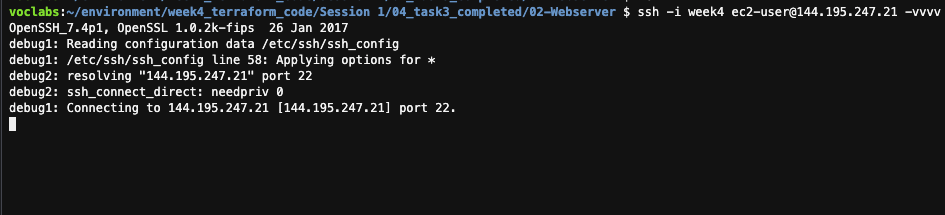


Figure 48 No SSH (Secure Shell) SSH (Secure Shell) connectivity over port 22

Checking our Security group and ingress rules – look good. Ingress is open for ports 80 and 22, all egress is open as well. How to access EC2 instance that has a Public IP but we not network connectivity.

Local network stack is functional since we used to connect to the instances already. We need to debug network connectivity on EC2 instance, yet we cannot log into it via SSH.

We need to check routing to see that the default route for our subnet is defined correctly.

There is no route for internet bound traffic! This is a perfect example of flowless yet non-functional deployment.

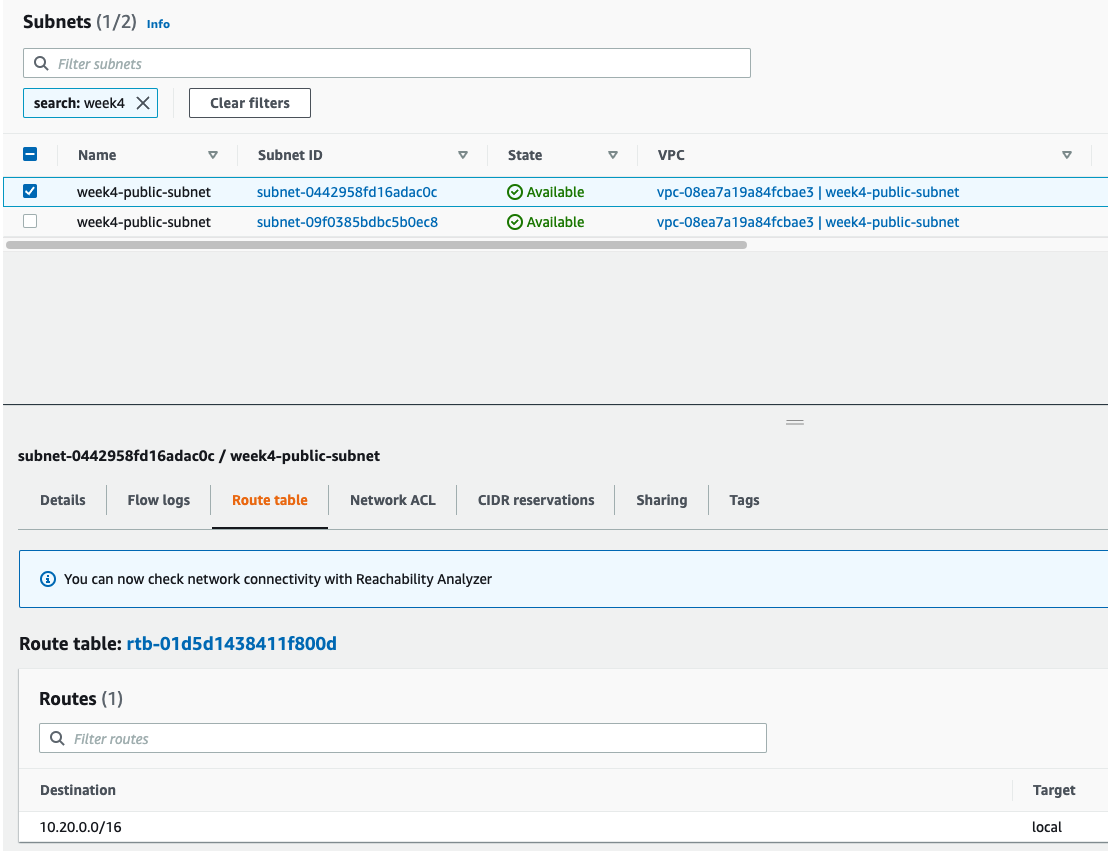


Figure 49 Default route is not defined

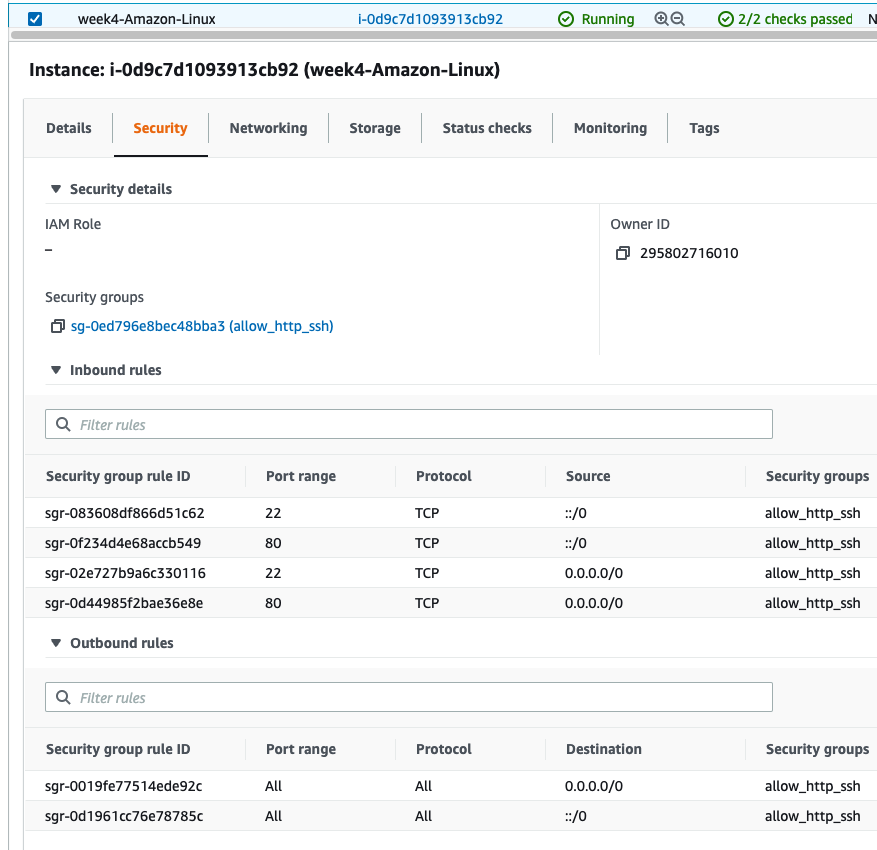


Figure 50 Security groups definition

Default VPC came with the route table provisioned by AWS, but custom VPC has local routes only in its default route table. We will fix it in the next step.

1. Provision route table, associate it with your public subnets

We need to refactor again!

Internet gateway should be provisioned in the Networking module, and it is id exposed via output variable. Move the IGW code to the Networking module and adjust output parameters of the Networking module.

Destroy the deployed Web server.

terraform destroy –auto-approve

Add the code below to the main.tf in 01-Networking folder.



Figure 51 Update main.tf in the 01-Networking folder

cd ../01-Networking

terraform validate

terraform plan

terraform apply

1. Redeploy Web server, check network connectivity

cd ../02-Webserver/

terraform apply

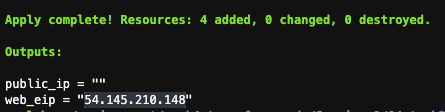


Figure 52 Webserver successfully redeployed



Figure 53 Webserver access via web browser

Change test in install\_httpd.sh file and rerun your deployment while refreshing your browser. Did the public IP change? How long was the downtime during redeployment?

1. Change the user data, run an update, check the downtime

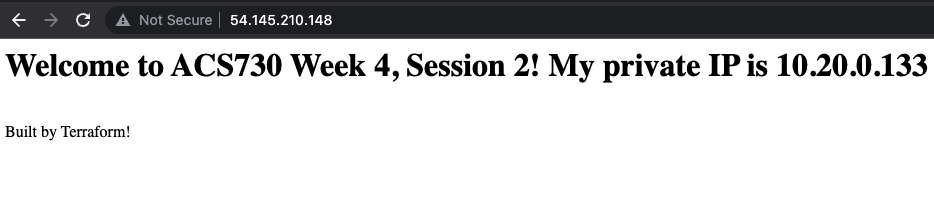


Figure 54 After redeployment, public IP preserved, private IP changed



# Optional Learning

## Terraform state commands

* Experiment with Terraform state commands:
  + terraform state [show| list| mv | pull| rm]
  + What are the dangers of Terraform state manipulations?
  + What will happen if we manually remove the resource from the state file?

## Reconciling the state

* Delete the EC2 instance provisioned by Terraform outside of the Terraform scope, either through AWS Management console or through AWS CLI
* Run terraform state show [aws\_ec2.name\_of\_the\_resource]. Why does Terraform think the EC2 is still deployed?
* Execute terraform refresh and re-run terraform state show [aws\_ec2.name\_of\_the\_resource].
* Re-run terraform apply. Why is EC2 instance re-created by Terraform?