Multi-Cloud Infrastructure Automation with Terraform and Ansible

AWS, GCP, Terraform, Ansible

Mohammed Amir

Project Overview

Created a multi-cloud lab environment using **Terraform**, where infrastructure is provisioned simultaneously on **AWS** and **GCP**, with support for multiple workspaces (`Dev`, `Stage`, `Prod`).This setup provides a foundation for practicing **Ansible automation**.

The project provisions the following components:

- **AWS Setup:** A secure environment is created in the default VPC with a Terraform-generated Key Pair and Security Group. Three EC2 instances (Ubuntu and Red-Hat based) are deployed, enabling a practical Ansible master–node setup. Public IPs are output for inventory configuration.
- **GCP Setup:** A dedicated VPC network is provisioned with firewall rules allowing SSH and HTTP. Two Compute Engine VMs (Debian and Ubuntu based) are deployed with injected SSH keys for secure access. Public IPs are exported for Ansible usage.

Key Skills

- **Terraform (IaC):** Automated provisioning of multi-cloud resources with reproducibility and workspace-based environments.
- **AWS (EC2, VPC, Security Groups):** Deployed secure instances tagged by environment.
- **GCP (Compute Engine, VPC, Firewall):** Provisioned secure, networked VMs with SSH access.
- **Ansible Integration:** Infrastructure ready for configuration management across clouds.

Introduction to Ansible

Ansible is an open-source automation tool used for **configuration management**, **application deployment**, and **orchestration**. It allows you to automate repetitive tasks, ensuring consistency across servers and reducing manual errors.

Unlike other tools, Ansible is **agentless** - it does not require any software to be installed on the managed nodes. It uses **SSH** (Secure Shell) to connect to servers, making it lightweight and easy to integrate with existing infrastructure.

Why use Ansible?

- **Simple and Human-Readable:** Uses YAML-based playbooks that are easy to understand.
- **Agentless:** No agents or daemons required on target machines.
- **Scalable:** Can manage a few servers or thousands efficiently.
- **Cross-Platform:** Works with Linux, Windows, cloud platforms (AWS, GCP, Azure), and containers.

Important Components of Ansible

- **Inventory:** A list of target machines (with IPs or hostnames) where tasks will be executed.
- **Playbooks:** YAML files that define tasks, roles, and configurations to apply.
- **Modules:** Predefined building blocks used to perform tasks such as installing packages, copying files, or starting services.
- **Roles:** A structured way to organize playbooks, variables, files, and handlers for reusability.
- **Tasks:** Individual actions executed on managed nodes (e.g., install Nginx).
- **Handlers:** Special tasks triggered only when notified (e.g., restart a service after a config update).

In this project, the infrastructure is provisioned on AWS and GCP using Terraform, while Ansible will be used as the **configuration management layer** to automate application setup, package installation, and system configuration across these multi-cloud servers.

1 Introduction to Terraform

Terraform is an open-source tool developed by HashiCorp that enables **Infrastructure as Code (IaC)**. It allows you to define, provision, and manage cloud resources (such as servers, networks, and storage) using simple configuration files written in HashiCorp Configuration Language (HCL).

Instead of manually creating resources through cloud dashboards, Terraform automates the process, ensuring infrastructure is reproducible, consistent, and version-controlled.

Key features include :-

- **Multi-Cloud Support:** Works with AWS, GCP, Azure, and many other providers.
- **Declarative Syntax:** You describe the desired state, and Terraform ensures infrastructure matches it.
- **Execution Plan:** Shows a preview of changes before applying them.
- **State Management:** Keeps track of deployed resources for updates and deletions.
- **Reusability:** Modules allow sharing and standardizing configurations across projects.

Core commands:

terraform init: Initialize working directory and download providers.

terraform plan: Show execution plan before applying changes.

terraform apply: Provision infrastructure as defined.

Terraform was installed on WSL (Linux) to enable infrastructure provisioning. For installation, refer to the official documentation page

Figure 1: Terraform Installation on WSL

terraform –version

2 INFRASTRUCTURE SETUP

In this project, Terraform was used to provision resources on both **AWS** and **GCP**. The infrastructure is defined using modular files such as `aws.tf`, `gcp.tf`, and `variables.tf`. Terraform workspaces (`dev`, `stage`, `prod`) were used to manage multiple environments consistently.

2.1 AWS Infrastructure

The AWS portion provisions:

- An SSH Key Pair for secure login.
- A default VPC and Security Group with inbound rules for SSH (22) and HTTP (80).
- Three EC2 instances (Ubuntu and RedHat) using AMIs defined in variables.

```
# ------------------ Key Pair ------------------
# Used for secure SSH login into EC2 instances
resource "aws_key_pair" "Key_new" {
  key_name   = "terra-key-ansible"
  public_key = file("terra-key-ansible.pub") # Reads local public key
}

# ------------------ Default VPC ------------------
# Using the default AWS VPC for simplicity
resource "aws_default_vpc" "default" {}

# ------------------ Security Group ------------------
# Allows SSH (22) and HTTP (80) inbound traffic
resource "aws_security_group" "my_ansible_security_group" {
  name        = "ansible-sg-${terraform.workspace}"
  description = "Terraform generated Security Group"
  vpc_id      = aws_default_vpc.default.id

  # Inbound rules
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"] # Open for demo (not recommended in
       prod)
    description = "SSH open"
  }
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
    description = "HTTP open"
  }
```

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

  tags = {
    Name        = "ansible-demo-sg-${terraform.workspace}"
    Environment = terraform.workspace
  }
}

# ------------------ EC2 Instances ------------------
# Creates multiple EC2 instances based on AMI map
resource "aws_instance" "my_instance" {
  for_each      = var.aws_amis   # Loops through AMIs defined in
     variables.tf
  ami           = each.value     # Selects AMI
  depends_on    = [aws_security_group.my_ansible_security_group,
     aws_key_pair.Key_new]
  key_name      = aws_key_pair.Key_new.key_name
  security_groups = [aws_security_group.my_ansible_security_group.name]
  availability_zone = var.aws_az
  instance_type = "t2.micro"

  tags = {
    Name        = "${each.key}-${terraform.workspace}"
    Environment = terraform.workspace
  }
}
```

2.2 AWS Variables

AWS specific variables define credentials, region, availability zone, and AMIs. The AMI map is especially important for creating multiple instances in one loop.

```
variable "aws_access_key" {}  # AWS Access Key (Sensitive)
variable "aws_secret_key" {}  # AWS Secret Key (Sensitive)

variable "aws_region" {
  default = "ap-south-1"      # Mumbai Region
}

variable "aws_az" {
  default = "ap-south-1a"     # Availability Zone
}

# Map of AMIs used for creating multiple EC2 instances
variable "aws_amis" {
  default = {
    Ansible-Master = "ami-02d26659fd82cf299", # Ubuntu
    Ubuntu-Demo    = "ami-02d26659fd82cf299", # Ubuntu
    RedHat-Demo    = "ami-0cf8ec67f4b13b491", # RedHat
```

```
  }
}
```

2.3 GCP Infrastructure

The GCP portion provisions:

- A custom VPC network named `ansible-network`.
- Firewall rules for SSH (22) and HTTP (80).
- Two Compute Engine VMs (Debian and Ubuntu) with injected SSH keys.

```
# ------------------ Custom VPC ------------------
resource "google_compute_network" "vpc_network" {
  name = "ansible-network-${terraform.workspace}"
}

# ------------------ Firewall Rules ------------------
# Allow HTTP traffic
resource "google_compute_firewall" "allow-http" {
  name    = "allow-http-${terraform.workspace}"
  network = google_compute_network.vpc_network.name

  allow {
    protocol = "tcp"
    ports    = ["80"]
  }
  source_ranges = ["0.0.0.0/0"]
  target_tags   = ["ansible", terraform.workspace]
}

# Allow SSH traffic
resource "google_compute_firewall" "allow-ssh" {
  name    = "allow-ssh-${terraform.workspace}"
  network = google_compute_network.vpc_network.name

  allow {
    protocol = "tcp"
    ports    = ["22"]
  }
  source_ranges = ["0.0.0.0/0"] # For demo (better: restrict to AWS
     Master IP)
}

# ------------------ VM Instances ------------------
resource "google_compute_instance" "vms" {
  for_each     = var.gcp_images
  name         = "${each.key}-${terraform.workspace}"
  machine_type = "e2-micro"
  zone         = var.gcp_zone

  boot_disk {
    initialize_params {
```

```
      image = each.value
    }
  }

  network_interface {
    network       = google_compute_network.vpc_network.name
    access_config {} # Assigns external IP
  }

  metadata = {
    ssh-keys = "ubuntu:${file(var.ssh_public_key_path)}"
  }

  tags = ["ansible", terraform.workspace]
}
```

2.4 GCP Variables

GCP specific variables define region, zone, project, VM images, and SSH key paths.

```
variable "gcp_region" {
  default = "us-central1"
}

variable "gcp_zone" {
  default = "us-central1-a"
}

variable "gcp_project" {
  default = "terraform-469922"
}

# Map of VM images to create multiple instances
variable "gcp_images" {
  default = {
    gcp-vm1 = "debian-cloud/debian-11"
    gcp-vm2 = "ubuntu-os-cloud/ubuntu-2204-lts"
  }
}

# Public key used for secure SSH access
variable "ssh_public_key_path" {
  description = "Path to SSH public key"
  default     = "/home/amoomirr/Terraform-Ansible/terra-key-ansible.pub
     "
}

# Path to GCP service account credentials JSON
variable "gcp_credentials_file" {}
```

2.5 Connecting AWS and GCP in Terraform

Both AWS and GCP providers are declared in `providers.tf`, allowing Terraform to manage multi-cloud infrastructure from a single configuration. Workspaces ensure that separate environments (`dev`, `stage`, `prod`) are isolated and manageable.

2.6 Outputs

This module is used to fetch the public IPs of all EC2 and GCP VMs, which serve as the dynamic Ansible inventory for configuration management.

```
output "aws_public_ips" {
  value = [
    for instance in aws_instance.my_instance : {
      name      = instance.tags.Name
      public_ip = instance.public_ip
    }
  ]
}

output "gcp_public_ips" {
  value = [
    for instance in google_compute_instance.vms : {
      name      = instance.name
      public_ip = instance.network_interface[0].access_config[0].nat_ip
    }
  ]
}
```

3 Multi-Cloud Outputs

Final Terraform outputs list AWS & GCP IPs. These are used in Ansible inventory.

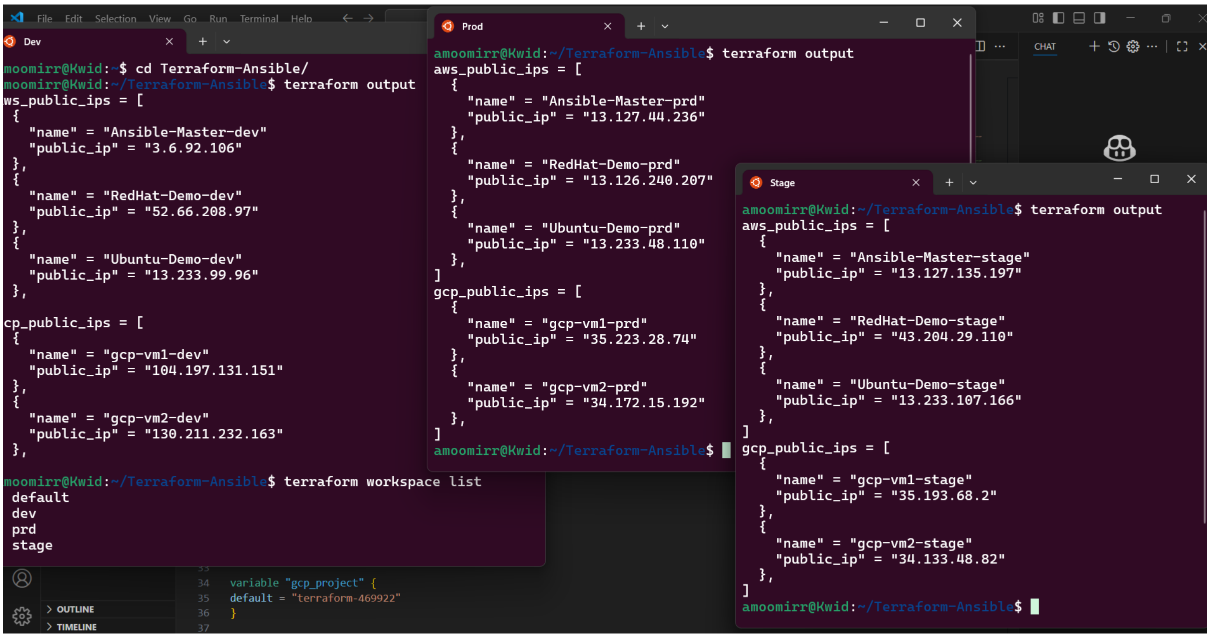


Figure 2: Terraform Outputs (Dev , Prod , Stage)

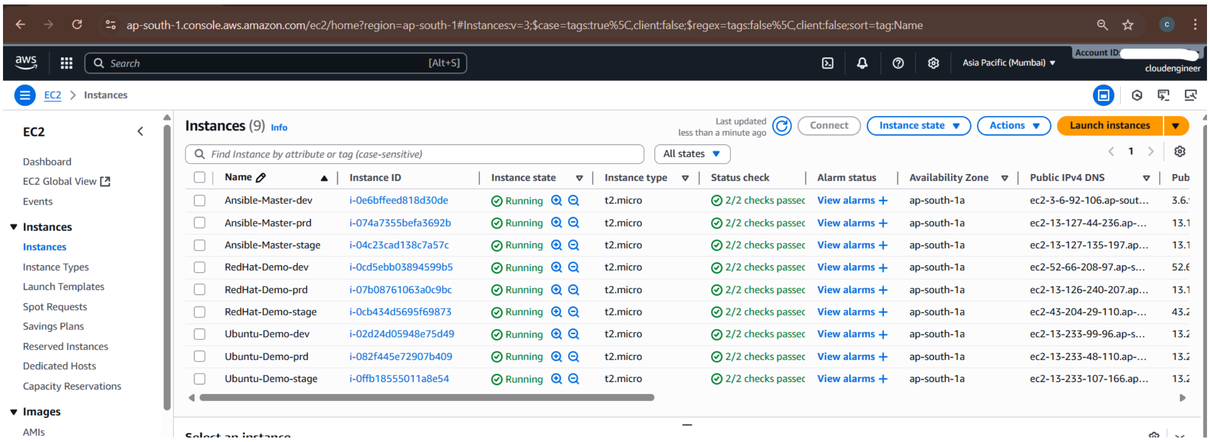


Figure 3: AWS DASHBOARD

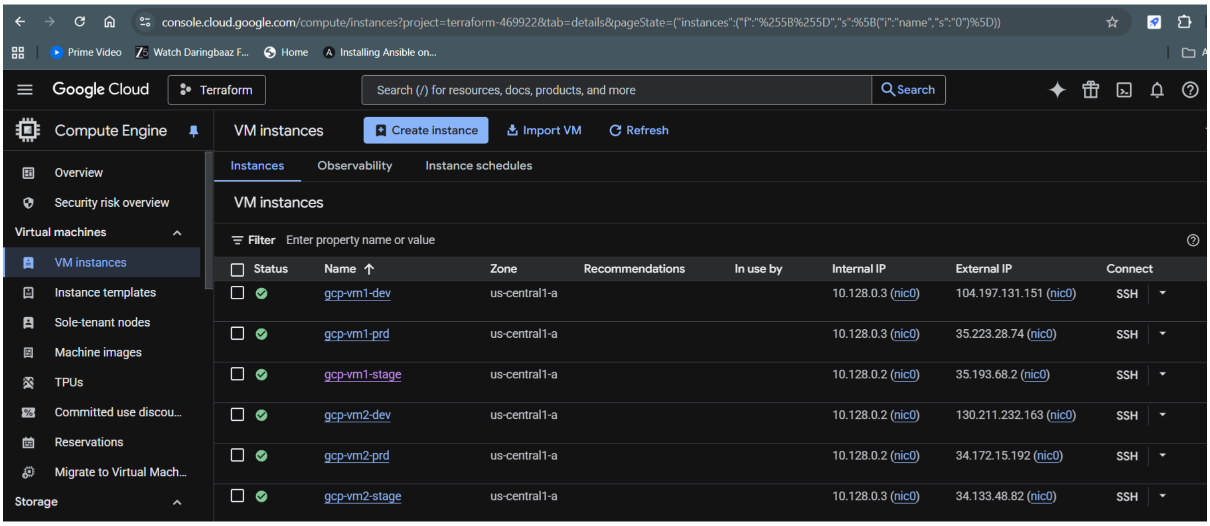


Figure 4: GCP DASHBOARD

4 Ansible Master Setup

After provisioning servers with Terraform, we configure an **Ansible Master** instance (Ubuntu EC2) to manage AWS and GCP servers.

4.1 Installing Ansible on Master

We first update the system and install Ansible using the official PPA:

```
$ sudo apt update
$ sudo apt install software-properties-common
$ sudo add-apt-repository --yes --update ppa:ansible/ansible
$ sudo apt install ansible
$ ansible --version
```

4.2 SSH Key Setup

To enable passwordless access from the Ansible Master to AWS and GCP servers:

- A private key `terra-key-ansible.pem` was placed inside the `.ssh/` directory.
- SSH open to 0.0.0.0/0 only for demo – should be restricted in production.
- Permissions were restricted with:

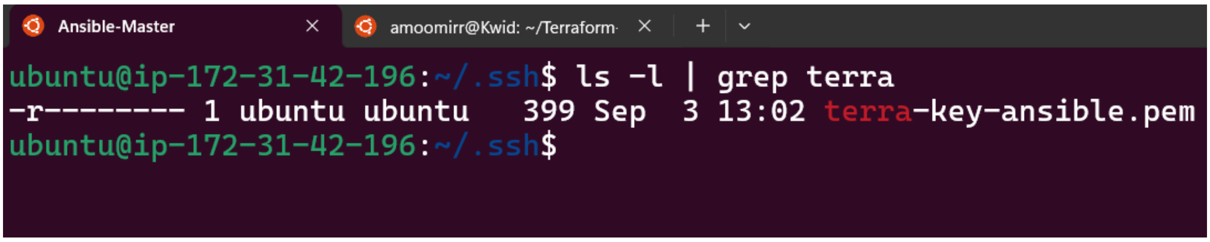


Figure 5: SSH PRIVATE KEY

```
chmod 400  /.ssh/terra-key-ansible.pem
```

This private key is referenced in the inventory file for connecting to target servers.

4.3 Ansible Project Structure

The following directory structure was created on the Ansible Master:

```
ansible-multi-env/
          inventory/
                dev.ini
                stage.ini
                prod.ini

          playbook/
                update.yml
                nginx-install.yml

          .ssh/
              terra-key-ansible.pem
```

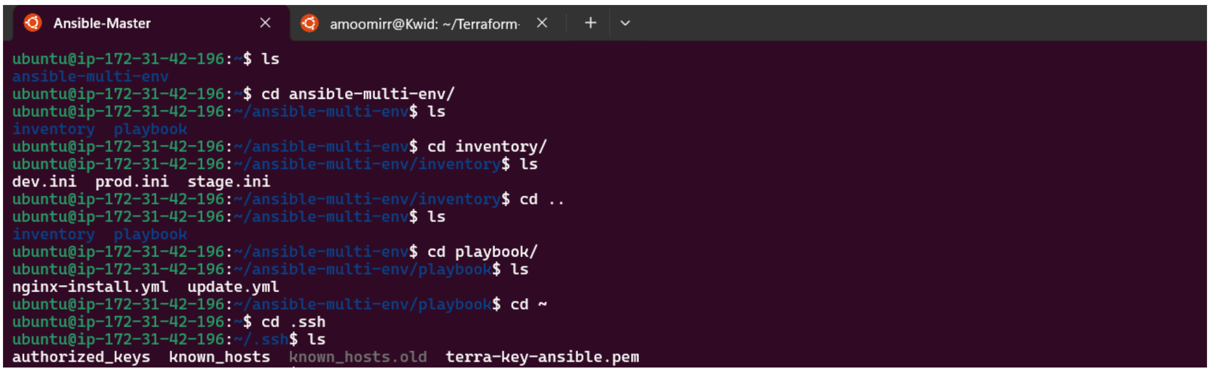


Figure 6: File Structure

4.4 Inventory File

The inventory file defines servers provisioned by Terraform. Each server entry includes its IP, user, and SSH key.

```
ubuntu@ip-172-31-42-196:~$ cat ansible-multi-env/inventory/stage.ini
[ansible_servers]
server1 ansible_host=13.233.107.166 ansible_user=ubuntu ansible_ssh_private_key_file=/home/ubuntu/.ssh/terra-key-ansible.pem
server2 ansible_host=43.204.29.110 ansible_user=ec2-user ansible_ssh_private_key_file=/home/ubuntu/.ssh/terra-key-ansible.pem
server3 ansible_host=35.193.68.2 ansible_user=ubuntu ansible_ssh_private_key_file=/home/ubuntu/.ssh/terra-key-ansible.pem
server4 ansible_host=34.133.48.82 ansible_user=ubuntu ansible_ssh_private_key_file=/home/ubuntu/.ssh/terra-key-ansible.pem

[ansible_servers:vars]
ansible_python_interpreter=/usr/bin/python3
```

Figure 7: Stage.ini

4.5 Running Playbooks

Example playbooks used:

- **update.yml** – Updates packages on Debian/Ubuntu or RedHat servers.

```
Ansible-Master    amoomirr@Kwid: ~/Terraform
ubuntu@ip-172-31-42-196:~$ cat ansible-multi-env/playbook/update.yml
- name: Update system
  hosts: ansible_servers
  become: yes

  tasks:
    # Debian/Ubuntu Update
    - name: Update packages on Debian/Ubuntu
      apt:
        update_cache: yes
      when: ansible_os_family == "Debian"

    # RedHat Update
    - name: Update packages on RedHat family
      yum:
        name: '*'
        state: latest
      when: ansible_os_family == "RedHat"
ubuntu@ip-172-31-42-196:~$
```

Figure 8: Update.yml

- **nginx-install.yml** – Installs and starts Nginx on servers.

```
Ansible-Master          amoomirr@Kwid: ~/Terraform
ubuntu@ip-172-31-42-196:~$ cat ansible-multi-env/playbook/nginx-install.yml
- name: Install Nginx
  hosts: ansible_servers
  become: yes

  tasks:

    # Install Nginx on Debian/Ubuntu
    - name: Install Nginx on Debian/Ubuntu
      apt:
        name: nginx
        state: present
      when: ansible_os_family == "Debian"

    # Install Nginx on RedHat family
    - name: Install Nginx on RedHat family
      yum:
        name: nginx
        state: present
      when: ansible_os_family == "RedHat"

    # Start Nginx
    - name: Start Nginx
      service:
        name: nginx
        state: started
        enabled: yes
ubuntu@ip-172-31-42-196:~$
```

Figure 9: Nginx-Install.yml

```
Ansible-Master          amoomirr@Kwid: ~/Terraform
ubuntu@ip-172-31-42-196:~/ansible-multi-env$ ls
inventory  playbook
ubuntu@ip-172-31-42-196:~/ansible-multi-env$ ansible-playbook -i inventory/dev.ini playbook/nginx-install.yml

PLAY [Install Nginx] ******************************************************************************************

TASK [Gathering Facts] ****************************************************************************************
ok: [server2]
ok: [server1]
ok: [server3]
ok: [server4]

TASK [Install Nginx on Debian/Ubuntu] *************************************************************************
skipping: [server2]
ok: [server1]
ok: [server3]
ok: [server4]

TASK [Install Nginx on RedHat family] *************************************************************************
skipping: [server1]
skipping: [server3]
skipping: [server4]
ok: [server2]

TASK [Start Nginx] ********************************************************************************************
ok: [server1]
ok: [server2]
ok: [server3]
ok: [server4]

PLAY RECAP ****************************************************************************************************
server1                    : ok=3    changed=0    unreachable=0    failed=0    skipped=1    rescued=0    ignored=0
server2                    : ok=3    changed=0    unreachable=0    failed=0    skipped=1    rescued=0    ignored=0
server3                    : ok=3    changed=0    unreachable=0    failed=0    skipped=1    rescued=0    ignored=0
server4                    : ok=3    changed=0    unreachable=0    failed=0    skipped=1    rescued=0    ignored=0

ubuntu@ip-172-31-42-196:~/ansible-multi-env$
```

Figure 10: Nginx-Install.yml

5 Verification of Nginx Deployment

After running the Ansible playbooks (`update.yml` and `nginx-install.yml`), we verified the successful installation of Nginx on both AWS and GCP servers.

5.1 Verification on AWS

The AWS EC2 instances provisioned by Terraform were configured via Ansible. Once the playbook executed, the Nginx web server was accessible in the browser using the public IP of the instance.

- Instance Type: `t2.micro`
- AMI: Ubuntu (from Terraform variables)
- Public IP : `43.204.29.110`
- Result: Displayed the default Nginx welcome page.

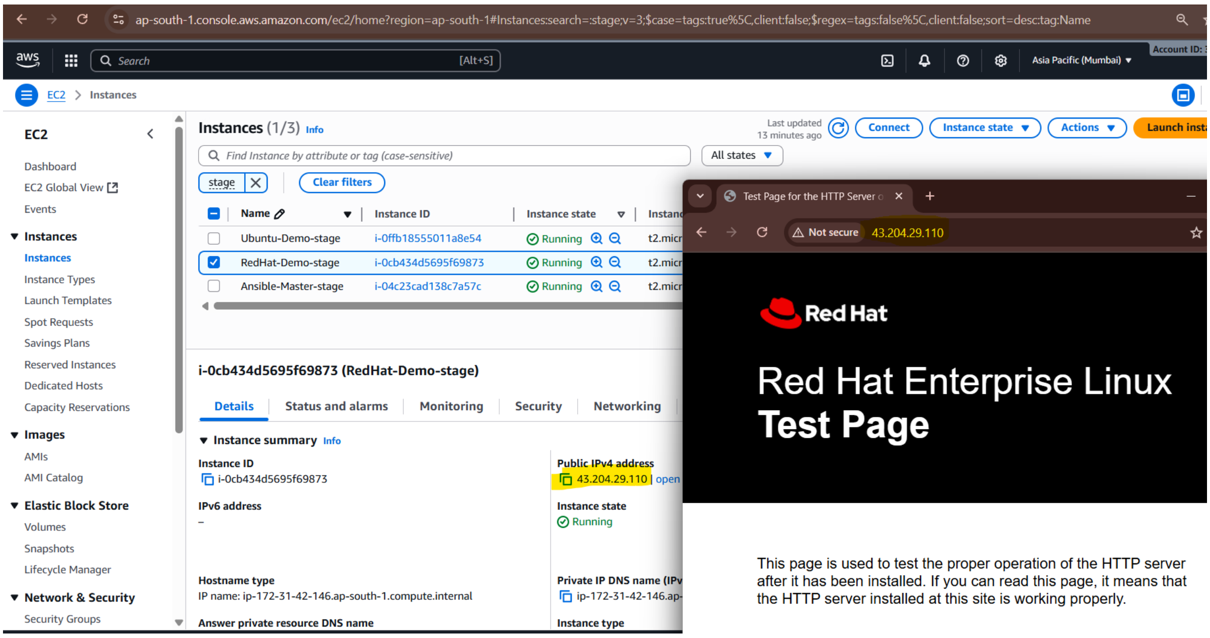


Figure 11: Nginx running on AWS EC2 instance (default welcome page)

5.2 Verification on GCP

Similarly, the GCP Compute Engine VMs were provisioned with firewall rules for ports 22 (SSH) and 80 (HTTP). After executing the Ansible playbook, Nginx was successfully installed and served the default page.

- Instance Type: `e2-micro`
- Images: Debian 11 and Ubuntu 22.04 (from Terraform variables)
- Public IP : `35.193.68.2`
- Result: Accessible via browser showing the Nginx default page.

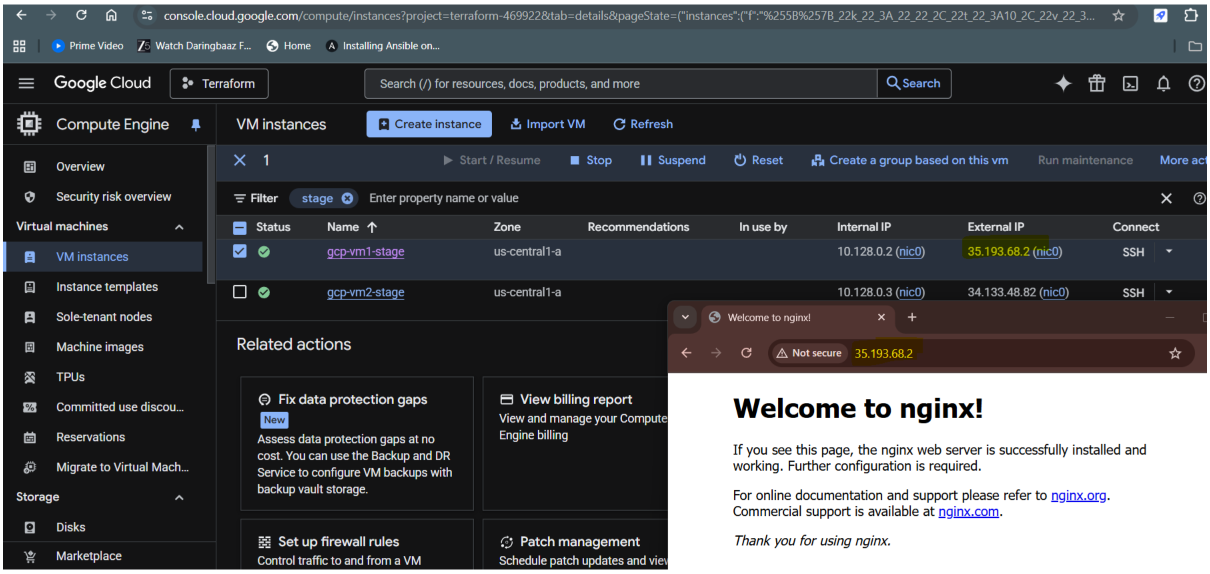


Figure 12: Nginx running on GCP Compute Engine VM (default welcome page)

5.3 Controlling Services (Start/Stop Nginx)

Using ad-hoc Ansible commands, we can control services remotely. For example:

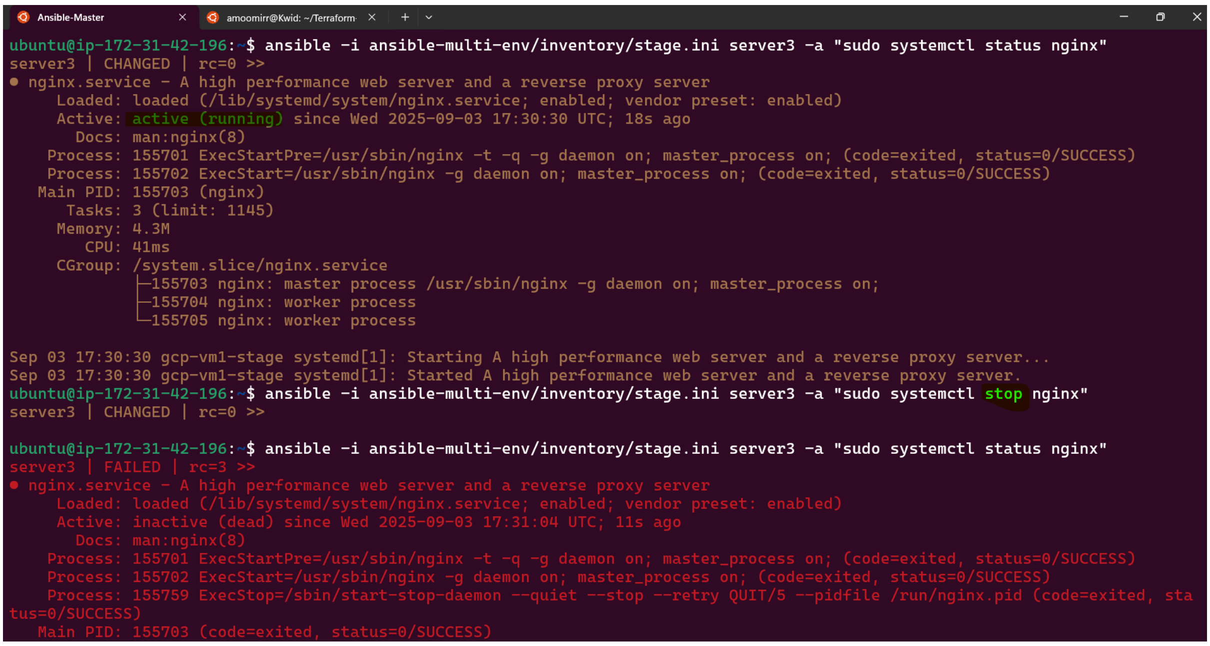


Figure 13: Service Status

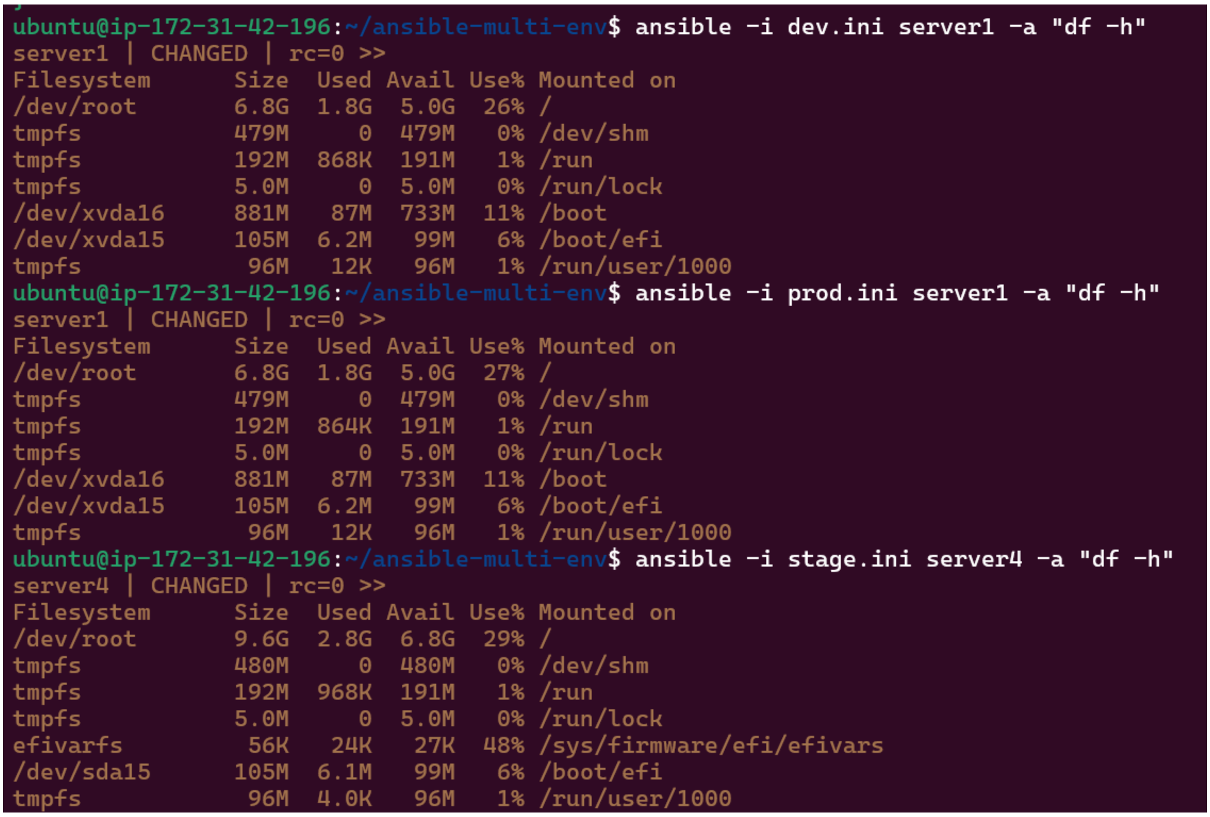
ansible -i ansible-multi-env/inventory/stage.ini server3 -a ”sudo systemctl start nginx”

ansible -i ansible-multi-env/inventory/stage.ini server3 -a ”sudo systemctl status nginx”

ansible -i ansible-multi-env/inventory/stage.ini server3 -a ”sudo systemctl stop nginx”

5.4 Monitoring Disk Utilization

We can also check resource usage directly from Ansible:



```
ubuntu@ip-172-31-42-196:~/ansible-multi-env$ ansible -i dev.ini server1 -a "df -h"
server1 | CHANGED | rc=0 >>
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       6.8G  1.8G  5.0G  26% /
tmpfs           479M     0  479M   0% /dev/shm
tmpfs           192M  868K  191M   1% /run
tmpfs           5.0M     0  5.0M   0% /run/lock
/dev/xvda16     881M   87M  733M  11% /boot
/dev/xvda15     105M  6.2M   99M   6% /boot/efi
tmpfs            96M   12K   96M   1% /run/user/1000
ubuntu@ip-172-31-42-196:~/ansible-multi-env$ ansible -i prod.ini server1 -a "df -h"
server1 | CHANGED | rc=0 >>
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       6.8G  1.8G  5.0G  27% /
tmpfs           479M     0  479M   0% /dev/shm
tmpfs           192M  864K  191M   1% /run
tmpfs           5.0M     0  5.0M   0% /run/lock
/dev/xvda16     881M   87M  733M  11% /boot
/dev/xvda15     105M  6.2M   99M   6% /boot/efi
tmpfs            96M   12K   96M   1% /run/user/1000
ubuntu@ip-172-31-42-196:~/ansible-multi-env$ ansible -i stage.ini server4 -a "df -h"
server4 | CHANGED | rc=0 >>
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       9.6G  2.8G  6.8G  29% /
tmpfs           480M     0  480M   0% /dev/shm
tmpfs           192M  968K  191M   1% /run
tmpfs           5.0M     0  5.0M   0% /run/lock
efivarfs         56K   24K   27K  48% /sys/firmware/efi/efivars
/dev/sda15      105M  6.1M   99M   6% /boot/efi
tmpfs            96M  4.0K   96M   1% /run/user/1000
```

Figure 14: Disk Utilization

ansible -i ansible-multi-env/inventory/stage.ini ansible$_servers - a''df - h''$

CONCLUSION

This project successfully demonstrates the integration of **Terraform** and **Ansible** to provision and manage a **multi-cloud lab environment** across AWS and GCP. Terraform provided automated, reproducible provisioning of compute instances, networking, and firewall rules, while Ansible acted as the configuration management layer for package updates, service installation, and operational tasks.

Key outcomes include:

- **Automated Multi-Cloud Infrastructure:** Terraform provisioned AWS EC2 instances and GCP Compute Engine VMs with a single codebase, ensuring reproducibility and consistency.
- **Environment Isolation:** Workspaces (`Dev`, `Stage`, `Prod`) enabled clean separation of environments for testing and production.
- **Seamless Ansible Integration:** Terraform outputs were directly used as Ansible inventory, allowing automated updates, package installations, and service configuration.
- **Cross-Platform Configuration Management:** Ansible installed and managed Nginx across multiple OS families (Ubuntu, Debian, RedHat) without manual intervention.
- **Remote Orchestration & Monitoring:** Services like Nginx could be started, stopped, and monitored, while disk utilization and system health were checked via Ansible ad-hoc commands.
- **Security & SSH Management:** Terraform-generated SSH keys and security groups/firewall rules enabled secure, controlled access across AWS and GCP.
- **Foundation for Hybrid-Cloud DevOps:** Demonstrates the power of combining IaC (Terraform) and Configuration Management (Ansible) for scalable, hybrid-cloud automation workflows.

In conclusion, this project highlights the power of combining **Infrastructure-as-Code (IaC)** with **Configuration Management** for building consistent, scalable, and cross-cloud workflows. It serves as a practical foundation for hybrid-cloud automation, enabling faster deployments and simplified operations.