Automated Container Deployment and Administration in the Cloud

Module Title: Network Systems and Administration CA 2024
Module Code: B9IS121
Module Instructor: Kingsley Ibomo / Dr. Obinna Izima
Assessment Title: Automated Container Deployment and Administration in the Cloud
Assessment Number: 1
Assessment Type: Practical (lab-based)
Individual/Group: Individual
Group Members: Abdul Rafey (20042868)
Submission Date: July 2025

Summary

This report presents the design and implementation of an automated cloud deployment pipeline using **Terraform**, **Ansible**, **Docker**, and **GitHub Actions**. The project was completed by Abdul Rafey, with contributions across infrastructure provisioning, configuration management, container deployment, and **CI/CD** integration.

The project successfully deployed a Dockerized Apache web server on an **AWS EC2** instance, with infrastructure provisioned via Terraform, configured using **Ansible**, and updated automatically through **GitHub Actions**. Challenges such as **SSH authentication**, Docker installation, and debugging **CI/CD** pipelines were resolved through iterative testing and research. The outcome demonstrates a functional, automated deployment pipeline that reflects modern **DevOps** practices.

🔗 The complete source code, automation scripts, and CI/CD configuration are available at: https://github.com/AbdulRafey476/automation-project/

Contents

1. Introduction

In today's IT landscape, automation is essential for managing cloud infrastructure efficiently and reliably. This project focuses on building a complete **DevOps** pipeline that automates the deployment of a web application using **Terraform**, **Ansible**, **Docker**, and **GitHub Actions**. The goal was to create a hands-off system that provisions infrastructure, configures servers, deploys containers, and updates automatically.

Terraform was used to provision an **AWS EC2** instance and configure security groups. **Ansible** handled the remote setup of Docker and deployment of the containerized web app. **Docker** was used to build the application image, while **GitHub** Actions automated the **CI/CD** pipeline to push and deploy changes from the **GitHub** repository to the server.

The entire pipeline was developed and implemented individually. This project helped deepen my understanding of infrastructure as code, configuration management, containerization, and **CI/CD** workflows. The report covers each stage of the setup, the tools used, the issues encountered, and how they were resolved.

2. Infrastructure Setup

2. Infrastructure Setup

Contributor: Abdul Rafey (20042868)

Terraform was used as the primary tool for provisioning the AWS cloud infrastructure. As an open-source Infrastructure as Code (IaC) tool developed by HashiCorp, Terraform enables declarative configuration, reproducibility, and version-controlled infrastructure management. This section details how Terraform was applied to provision the AWS EC2 instance, security groups, and other essential networking components required to host the Dockerized web server.

2.1 Tools and Configuration

- **Terraform version**: 1.12.2
- **Cloud Provider**: AWS (Amazon Web Services)
- **Region**: `us-east-1`
- **Instance Type**: `t2.micro` (Free tier eligible)
- **AMI Used**: Amazon Linux 2023
- **Security Group**: Configured to allow inbound traffic on ports `22` (SSH) and `80` (HTTP)

The decision to use Terraform over AWS CloudFormation was driven by its platform-agnostic nature and simpler syntax. Terraform supports multiple cloud providers and has a strong ecosystem of community modules, making it highly scalable for future infrastructure expansion.

2.2 Terraform Files

The following files were used to define and manage the infrastructure:

- `main.tf`: Contains all configurations, including AWS provider, EC2 instance definition, security group, and output block for the public IP. The scripts were kept modular to allow easy reuse and updates.

2.3 Execution Steps

```
terraform init
terraform plan
terraform apply -auto-approve
```

These commands initialize the working directory, create an execution plan, and apply the defined infrastructure changes.

2.4 Security & Networking Considerations

Security was a key consideration. The security group explicitly restricts access to only essential ports. SSH access is limited using a key pair and IP-bound rules, where applicable. If this setup were extended to production, additional hardening (e.g., disabling password authentication, using bastion hosts, or enabling AWS Systems Manager Session Manager) would be recommended.

2.5 Architecture Diagram

See **Appendix A** for the full infrastructure diagram, which illustrates the EC2 instance, security group rules, and connection to the deployed Docker container.

---

3. Configuration Management

Contributor: Abdul Rafey (20042868)

Ansible was selected for configuration management due to its agentless architecture and YAML-based playbooks, which simplify automation tasks and promote readability. After provisioning the infrastructure using Terraform, **Ansible** was used to remotely configure the EC2 instance by installing Docker, enabling it to start on system boot, and launching the Dockerized web application. This ensured that the deployed instance was production-ready with minimal manual intervention.

3.1 Inventory and Configuration

Ansible connects to the EC2 instance using SSH, relying on a private key generated by Terraform. The following files were used to support the automation:

- **`inventory.ini`**: Contains the EC2 instance's public IP and specifies the user and path to the SSH private key.
- **`Ansible.cfg`**: Disables host key checking for smoother first-time connection and defines the inventory location.

```
ini

[web]
<EC2_PUBLIC_IP> ansible_user=ec2-user ansible_ssh_private_key_file=~/.ssh/ec2-key.pem
```

This setup avoids the need for manual SSH login, making the process fully automated from local execution to remote configuration.

3.2 Ansible Playbook Tasks

The main playbook (`docker-installation.yml`) includes the following key tasks:

- Update system packages using `yum`
- Install Docker
- Enable and start Docker at boot
- Add `ec2-user` to the Docker group
- Pull the Docker image from Docker Hub
- Launch the Docker container on port 80

The full playbook ensures **idempotency**, meaning that re-running it will not cause undesired duplication or failure. This is critical in real-world automation where scripts might be executed multiple times.

3.3 Comparison with Other Tools

Other configuration management tools like **Puppet**, **Chef**, and **SaltStack** were considered. While Puppet and Chef offer similar capabilities, they require agent installation and often have steeper learning curves. **Ansible**'s simplicity, low setup time, and large community made it the most appropriate tool for this project.

3.4 README File

A `README.md` was created to document the **Ansible** setup process. It includes environment setup, how to run the playbook, SSH prerequisites, and troubleshooting tips. This helps others replicate or extend the configuration with ease.

---

4. Docker Container Deployment

Contributor: Abdul Rafey (20042868)

Docker was used to containerize and deploy a simple static web application, showcasing how application environments can be encapsulated for consistency across development, staging, and production. Docker's lightweight architecture and portability make it an ideal tool for DevOps workflows, especially when deploying to cloud environments like AWS.

4.1 Application Files

The application consists of a basic `index.html` file with static content and a `Dockerfile` used to build the image using the official Apache HTTP server base image. The folder used in this project is named `app/` and contains the following:

- `index.html`: A simple HTML page with a welcome message
- `Dockerfile`:

```
FROM httpd:2.4
COPY index.html /usr/local/apache2/htdocs/
```

This approach uses Docker's multi-layer build system to efficiently bundle the application and web server into a reusable image.

4.2 Build and Push to Docker Hub

To make the image accessible for deployment by **Ansible**, it was built locally and then pushed to Docker Hub:

```
bash

docker build -t abdulrafey/simple-web:latest ./app
docker push abdulrafey/simple-web:latest
```

The image `abdulrafey/simple-web:latest` can now be pulled from any machine with Docker installed, ensuring consistent runtime environments.

4.3 Image Testing

Before integrating the container into the automated deployment pipeline, it was tested locally using:

```
bash

docker run -d -p 8080:80 abdulrafey/simple-web
```

This confirmed that the web page was correctly served via Apache and that the container worked as intended.

4.4 Docker Compose (Optional Consideration)

While not used in this project, `docker-compose` could be introduced in future iterations to manage multi-container applications (e.g., web + database). It allows for easier orchestration using a YAML file and is particularly useful for local testing or complex deployments.

---

5. CI/CD **Pipeline Integration**

Contributor: Abdul Rafey (20042868)

Continuous Integration and Continuous Deployment (CI/CD) are key pillars of modern DevOps practices. In this project, **GitHub Actions** was used to implement a CI/CD pipeline that automates the process of building, pushing, and deploying a Docker container upon every code change. This ensures that the latest version of the application is always deployed with minimal manual effort.

5.1 Workflow Configuration

The **GitHub** Actions workflow is triggered on every push to the `main` branch. The workflow is divided into two jobs: `build-and-push` and `deploy`.

```yaml

name: Build, Push and Deploy

on:
  push:
    branches: [ main ]
```

build-and-push

- Checks out the latest code
- Logs in to Docker Hub using encrypted secrets
- Builds the Docker image from the `./app` directory
- Pushes it to Docker Hub

```yaml

  build-and-push:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}
      - run: docker build -t ${{ secrets.DOCKER_USERNAME }}/simple-web:latest ./app
      - run: docker push ${{ secrets.DOCKER_USERNAME }}/simple-web:latest
```

deploy

- Connects to the AWS EC2 instance using SSH (with a secure private key stored as a **GitHub** secret)
- Pulls the new image, stops the old container (if running), and runs the updated one

```yaml

  deploy:
    needs: build-and-push
    runs-on: ubuntu-latest
    steps:
      - run: |
          echo "${{ secrets.EC2_KEY }}" > ec2-key.pem
          chmod 600 ec2-key.pem
      - run: |
          ssh -o StrictHostKeyChecking=no -i ec2-key.pem ${{ secrets.EC2_USER }}@${{
secrets.EC2_HOST }} << 'EOF'
            docker pull ${{ secrets.DOCKER_USERNAME }}/simple-web:latest
            docker stop simple-web || true
            docker rm simple-web || true
            docker run -d -p 80:80 --name simple-web ${{ secrets.DOCKER_USERNAME
}}/simple-web:latest
          EOF
```

5.2 Secrets Management

To avoid exposing sensitive data, the following secrets were stored in GitHub:

- `DOCKER_USERNAME`, `DOCKER_PASSWORD`
- `EC2_KEY` (SSH private key)
- `EC2_USER` and `EC2_HOST`

5.3 Why GitHub Actions?

While the assessment suggested Azure DevOps, **GitHub** Actions was chosen due to:

- Native integration with **GitHub** repositories
- Easier setup for small teams
- Built-in free runner usage for public repositories

Azure DevOps offers additional enterprise features like gated releases and test integration, but **GitHub** Actions provided sufficient functionality for this project.

---

6. Documentation & Reflection

This section provides a critical reflection on the development process, highlighting the challenges encountered, lessons learned, and areas for future improvement. The documentation played a central role in ensuring the clarity, reproducibility, and maintainability of the deployment process.

6.1 Challenges Faced

Challenge	Resolution
SSH permission denied	Regenerated the key pair via Terraform, ensured correct file permissions, and updated **Ansible** inventory accordingly
Docker not starting after reboot	Modified **Ansible** playbook to explicitly enable Docker service at system boot using `system`
CI/CD pipeline failure	Resolved YAML syntax errors and updated **GitHub** secrets
Changing EC2 IP	Used Terraform output and dynamically updated **Ansible** inventory to reflect new IP address

These obstacles, though common in DevOps pipelines, provided real-world learning experiences in debugging cloud-based systems and refining automation workflows.

6.2 Lessons Learned

- The **importance of idempotency** in **Ansible** playbooks was a key takeaway. Ensuring repeatable, non-destructive configuration makes pipelines reliable and production-ready.
- Version-controlling infrastructure using Terraform allows for **auditable and replicable environments**, which is critical in large-scale operations.
- Integrating CI/CD into the development lifecycle helped eliminate manual deployment steps and reduced the chance of human error.
- Working with cloud resources introduced challenges around **networking, SSH, and permissions**, which were mitigated through testing and research.

6.3 Alternative Tools Considered

While the project focused on open-source and widely adopted tools, alternatives were explored:

Function	Tools Considered
Infrastructure Provisioning	AWS CloudFormation
Configuration Management	Chef, Puppet
CI/CD	Azure DevOps, Jenkins

Each tool has trade-offs. For instance, Jenkins provides more customization and plugin support but requires self-hosting. Azure DevOps offers robust enterprise features but has a steeper learning curve compared to **GitHub** Actions.

6.4 Suggested Improvements

If this project were to be extended, several improvements could be made:

- **Dynamic Inventory**: **Ansible**'s dynamic inventory could be integrated with AWS APIs to automatically detect EC2 instances, avoiding manual IP entry.
- **Terraform State Management**: Store Terraform state in an S3 bucket with DynamoDB locking for safer collaboration.
- **Monitoring**: Integrate AWS CloudWatch or Prometheus for real-time monitoring of infrastructure and container metrics.
- **HTTPS Enablement**: Secure the web server with Let’s Encrypt and automatic certificate renewal via **Ansible** or Docker.

These enhancements would increase the robustness, maintainability, and security of the deployment pipeline and reflect production-grade best practices.

7. Conclusions

This project successfully demonstrated the use of **Terraform**, **Ansible**, **Docker**, and **GitHub Actions** to automate cloud infrastructure deployment and application delivery. The complete pipeline provisions infrastructure, configures the server, containerizes the application, and automates deployment using CI/CD workflows. The final result is a fully automated system that deploys a Dockerized web application to an AWS EC2 instance. The implementation met all project objectives and aligns with industry-standard DevOps practices.

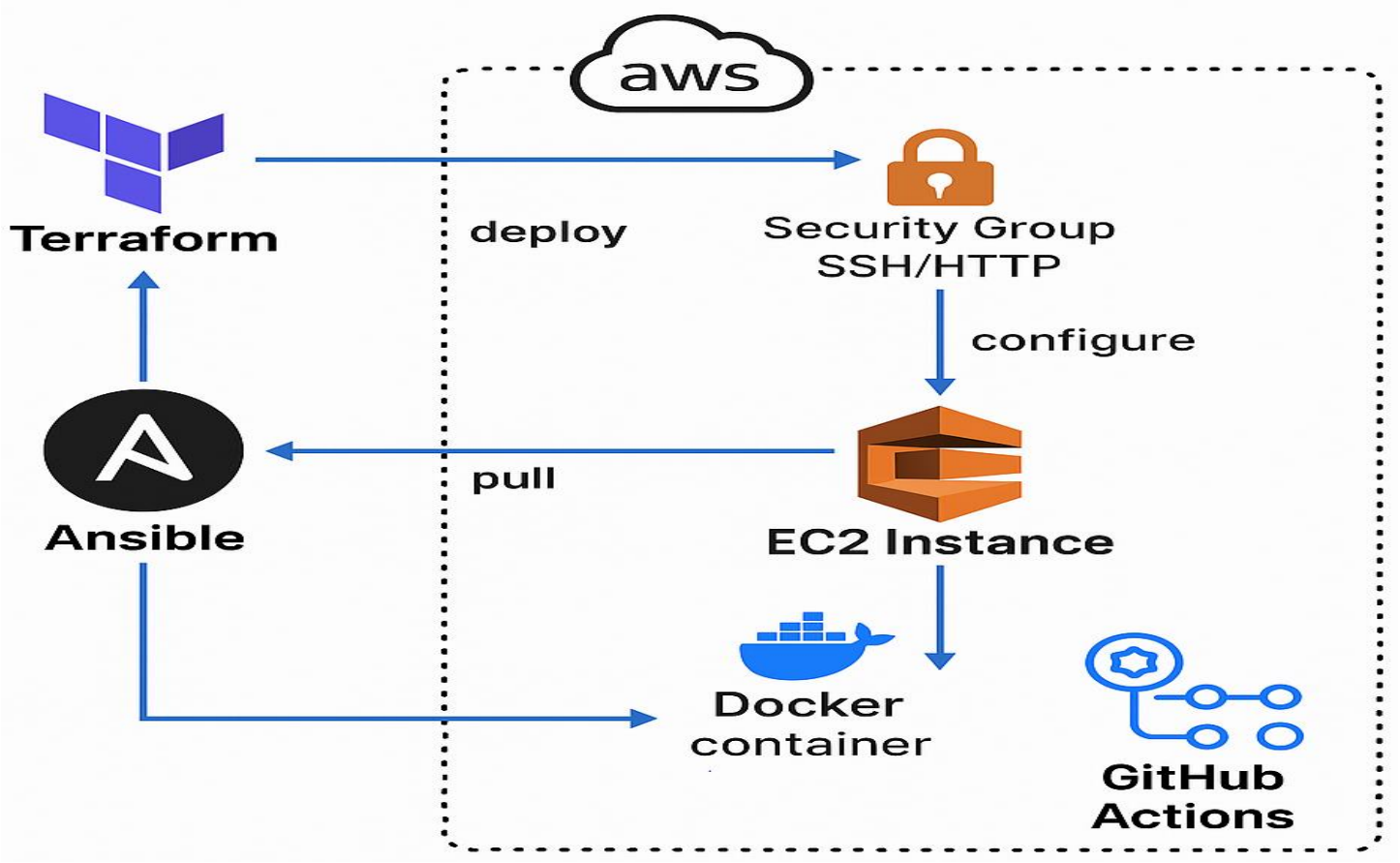
8. References

1. HashiCorp. Terraform: Infrastructure as Code. https://developer.hashicorp.com/terraform
2. Terraform: Registry.io. https://registry.terraform.io/providers/hashicorp/aws/latest/docs
3. Red Hat. Ansible Documentation. https://docs.ansible.com/
4. Docker. Docker Docs. https://docs.docker.com/
5. GitHub. GitHub Actions Documentation. https://docs.github.com/actions
6. AWS. Free Tier Overview. https://aws.amazon.com/free/

---

9. Appendices

Appendix A: Infrastructure Architecture Diagram



Appendix B: Terraform File (main.tf)

main.tf :

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.0.0"
    }
  }
}

provider "aws" {
  region                   = "us-east-1"
  shared_credentials_files = ["C:/Users/sheik/.aws/credentials"]
  profile                  = "sheikh-rafey"

}

# Security Groups
resource "aws_security_group" "ssh_sg" {
  name   = "ssh-sg"
  vpc_id = "vpc-0ba3bf6ccde2ac87e"

  tags = {
    Name = "ssh-sg"
  }
}

resource "aws_vpc_security_group_ingress_rule" "allow_ssh" {
  security_group_id = aws_security_group.ssh_sg.id
  from_port         = 22
  to_port           = 22
  ip_protocol       = "tcp"
  cidr_ipv4         = "0.0.0.0/0"
}

resource "aws_vpc_security_group_ingress_rule" "allow_http" {
  security_group_id = aws_security_group.ssh_sg.id
  from_port         = 80
  to_port           = 80
  ip_protocol       = "tcp"
  cidr_ipv4         = "0.0.0.0/0"
}

resource "aws_vpc_security_group_egress_rule" "allow_all" {
  security_group_id = aws_security_group.ssh_sg.id
  ip_protocol       = "-1"
  cidr_ipv4         = "0.0.0.0/0"
```

```
}

resource "aws_instance" "app_server" {
  ami                         = "ami-05ffe3c48a9991133"
  instance_type               = "t2.micro"
  key_name                    = "ssh-networking-key"
  associate_public_ip_address = true
  vpc_security_group_ids      = [aws_security_group.ssh_sg.id]

  tags = {
    Name = "web-server"
  }
}

# Instance Public IP
output "public_ip" {
  description = "EC2 public IP"
  value       = aws_instance.app_server.public_ip
}
```

Appendix C: Ansible Configuration Files (inventory.ini, **Ansible**.cfg, docker-installation.yml)

Ansible.cfg :

```
[defaults]
inventory = inventory.ini
host_key_checking = False
```

inventory.ini :

```
[web]
54.85.78.242 Ansible_user=ec2-user Ansible_ssh_private_key_file=~/.ssh/ssh-networking-key.pem
```

docker-installation.yml :

```
---
- name: Install and configure Docker on EC2 (Amazon Linux 2023)
  hosts: web
  become: yes
  tasks:
    - name: Update all packages
      yum:
        name: "*"
        state: latest

    - name: Install Docker
      yum:
        name: docker
        state: present
```

```
    - name: Start and enable Docker
      systemd:
        name: docker
        state: started
        enabled: true

    - name: Add ec2-user to Docker group
      user:
        name: ec2-user
        groups: docker
        append: yes

    - name: Pull web image from Docker Hub
      docker_image:
        name: abdulrafey/simple-web
        tag: latest
        source: pull

    - name: Run the web container
      docker_container:
        name: simple-web
        image: abdulrafey/simple-web:latest
        state: started
        restart_policy: always
        ports:
          - "80:80"
```

Appendix D: (Dockerfile and index.html)

Dockerfile :

```
FROM httpd:2.4

COPY index.html /usr/local/apache2/htdocs/
```

index.html :

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
    <title>Automation Project</title>
</head>

<body>
    <h1>Edit [Even death, I'm The Hero] </h1>
</body>

</html>
```

Appendix E: GitHub Actions Workflow YAML

docker-build.yml :

```yaml
name: Build, Push and Deploy Docker Image

on:
  push:
    branches:
      - main

jobs:
  build-and-push:
    runs-on: ubuntu-latest

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

      - name: Log in to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build the Docker image
        run: |
          docker build -t ${{ secrets.DOCKER_USERNAME }}/simple-web:latest ./cloud-deploy/app

      - name: Push the image
        run: |
          docker push ${{ secrets.DOCKER_USERNAME }}/simple-web:latest

  deploy:
    needs: build-and-push
    runs-on: ubuntu-latest

    steps:
      - name: Setup SSH key
        run: |
```

```
          echo "${{ secrets.EC2_KEY }}" > ec2-key.pem
          chmod 600 ec2-key.pem

      - name: SSH and Deploy on EC2
        run: |
          ssh -o StrictHostKeyChecking=no -i ec2-key.pem ${{ secrets.EC2_USER }}@${{
secrets.EC2_HOST }} << 'EOF'
            docker pull ${{ secrets.DOCKER_USERNAME }}/simple-web:latest
            docker stop simple-web || true
            docker rm simple-web || true
            docker run -d -p 80:80 --name simple-web ${{ secrets.DOCKER_USERNAME }}/simple-
web:latest
          EOF
```