11/9/2025

Automated Container Deployment and Administration in the Cloud

Network Systems and Administration CA_One

Module Instructor: Kingsley Ibomo
Word Count: [Main text word count here]
Summary Word Count: [Summary word count here]

Mayur Govinda Dhake
20078257

Table of Contents

1. Summary

This report goes into a lot of detail on analysing and setting up a cloud automation setup. It pulls together Terraform, Ansible, Docker, and GitHub Actions right inside the AWS world. The main goal here is to handle the whole process of deploying a Flask web app in containers. That means everything from setting up the basic infrastructure to keeping integration and delivery going smoothly. All of it follows those IaC and DevOps ideas pretty closely.

The work shows off how mixing these free tools leads to deployments that you can repeat easily. They stay secure too, and scale up without much trouble. Terraform handles the cloud setup in a clear, declarative way. Ansible takes care of managing configs and pulling things together. Docker keeps the app steady by packing it into containers. GitHub Actions runs the automated checks and releases for CI/CD. Then there are some extra shell scripts that connect all these parts. They let you run the full thing from just one command in the terminal.

In the end, you get a workflow that is modular and can handle errors well. It cuts down on people doing things by hand. Plus, it makes sure everything repeats reliably. Those are the big ideas in DevOps these days.

2. Introduction

Cloud computing really changed the game for designing, deploying, and keeping up modern applications. Manual deployment seems simple enough in smaller setups. But it turns error-filled and slow when you scale things up. Automation steps in to fix those issues. It does this through tools like Infrastructure as Code, or IaC. Configuration Management, known as CM, plays a part too. Then there is Continuous Integration and Continuous Deployment, or CI/CD. All of these shift daily operations into code-based flows that run reliably.

This project puts that idea to work in a real pipeline. It pulls together Terraform, Ansible, Docker, and GitHub Actions. They coordinate to roll out a basic web app on AWS. The setup feels solid and ready to go.

The whole design lines up with DevOps principles pretty well. DevOps stresses teamwork between developers and operations folks. It pushes for ongoing tweaks and dependable systems. Terraform handles provisioning those EC2 instances. Ansible takes care of environment configs. Docker packages up the application code neatly. GitHub Actions manages the integration and deployment automatically. This mix shows how different tools link up for one main goal. That goal is fast and steady delivery to the cloud.

The research objectives cover a few key areas. One is to automate provisioning infrastructure on AWS with Terraform. Another involves using Ansible for system setup and installing Docker. There is also the task of containerizing a Flask web app to make deployments more

reliable. Integration comes next with GitHub Actions for CI/CD processes. The plan includes looking at other tech options and explaining why these choices fit best. Finally, it calls for full documentation on the architecture, the steps involved, and a solid evaluation.

This effort serves practical needs. It delivers actual automation components you can run. At the same time, it has an academic side. That includes comparing approaches and building on DevOps theory.

3. Theoretical Background

Automation in cloud computing draws from several core concepts. They connect in ways that support each other. Infrastructure as Code plays a big role here. Configuration Management adds to the mix. Containerization brings its own strengths. CI/CD pipelines tie things together overall. Each element contributes uniquely. That approach helps nail down continuous delivery in practice.

3.1 Infrastructure as Code (IaC)

IaC treats infrastructure elements like servers and networks as pieces of code. You can version control them just like regular scripts. This shifts away from hands on manual processes. Tools including Terraform or AWS CloudFormation come in handy. They let you specify the desired state clearly. The underlying system then ensures everything aligns with that plan. Reproducibility becomes straightforward as a result. Rollbacks happen without much hassle. Peer reviews on infrastructure changes get easier too. Those features used to apply mainly to software code alone.

Terraform supports IaC in this project setup. It maintains a state file to monitor existing resources. Updates roll out incrementally that way. Dependencies get handled automatically. Setting up a server turns traceable. It mirrors committing code to a repository in feel.

3.2 Configuration Management (CM)

Once infrastructure stands ready, configuration steps follow closely. Installing software takes priority. Services need setup. Environments require tuning for optimal performance. CM tools such as Ansible, Puppet, and Chef handle these tasks. They convert operations into scripts like playbooks or manifests. Consistency holds across repeated runs. Idempotence describes that reliable behaviour.

Ansible handles Docker installation in this project. It manages system packages effectively. User accounts fall under its scope too. The tool operates agentless. SSH provides the

communication link to servers. Extra processes avoid installation on target machines. Maintenance drops lower. Security concerns ease up noticeably.

3.3 Containerization

Containerization isolates applications along with their dependencies. Lightweight runtimes make this possible. Docker emerges as the primary choice for implementation. It replaces traditional virtual machines with portable containers. Startup times drop to milliseconds. Containers bundle everything an app needs. Runtimes, libraries, and config files all fit inside. The application behaves identically on a developer's laptop. Production environments match without issues.

This project packages the Flask app into a Docker image. A concise Docker file guides the process. Environment stability remains consistent everywhere. Rollbacks simplify as a direct benefit.

3.4 Continuous Integration and Continuous Deployment (CI/CD)

CI/CD streamlines the full cycle of code handling. Building occurs automatically. Testing runs frequently. Deployment pushes updates to production seamlessly. CI focuses on integrating changes and executing tests regularly. CD manages the release automation side. Tools like Jenkins, Azure DevOps, and GitHub Actions configure these pipelines. Declarative setups make workflows straightforward.

GitHub Actions activates on code pushes to the repository in this project. Docker images build during the process. They upload to Docker Hub afterward. Deployment follows on AWS via SSH. App changes appear live almost immediately. Manual redeploys become unnecessary.

3.5 DevOps as a Cultural and Technical Practice

DevOps extends beyond mere tooling. Feedback loops gain emphasis. Monitoring integrates deeply. Shared responsibilities spread across teams. Automation facilitates these elements. Yet it serves as a means, not the end itself. This project combines various open-source tools. The DevOps lifecycle unfolds completely here. Coding leads to building. Deployment and operations complete the loop. An educational focus shapes the design. Real world industry practices align closely with it too.

4. Tool Analysis and Integration

Automation in DevOps setups really picks up speed when you get various specialized tools working together across the software delivery process. These tools each bring their own approaches to design, what they handle day to day, and how they connect with others. For this project, we picked four main ones. Terraform, Ansible, Docker, and GitHub Actions fit together nicely to build out a full automated deployment setup.

4.1 Terraform Infrastructure as Code

- Overview

  Terraform comes from HashiCorp. It is an open-source tool for Infrastructure as Code. This framework lets users spell out cloud setups in a declarative way. It skips over all those step-by-step commands for building things. Instead, engineers just describe the end goal for the system. Terraform works with lots of cloud options through plug in providers. Think AWS, Azure, or GCP. That setup makes it pretty flexible and easy to expand compared to other IaC options out there.

- Theoretical Foundation

  Terraform builds on declarative configs and a way to converge toward the right state. It checks the current setup, which gets saved in a state file, against what you want. From there, it figures out exactly what to do. That could mean creating stuff, changing it, or even tearing it down. The whole thing stays idempotent. It also cuts down on drift that happens when people manage things by hand.

- Implementation

  This project uses Terraform to set up an EC2 instance. It also makes a security group and creates an SSH key pair. Here is a basic look at the config.

```
region = var.aws_region
}

resource "aws_key_pair" "deployer" {
 key_name   = "deployer-key"
 public_key = file(var.public_key_path)
}

resource "aws_security_group" "web_sg" {
 name_prefix = "web-sg-"
 description = "Allow SSH and HTTP"
 ingress {
   from_port   = 22
   to_port     = 22
   protocol    = "tcp"
   cidr_blocks = [var.my_ip_cidr]    # tighten to your IP
 }
 ingress {
   from_port   = 80
   to_port     = 80
   protocol    = "tcp"
   cidr_blocks = ["0.0.0.0/0"]
 }
 egress {
   from_port   = 0
   to_port     = 0
   protocol    = "-1"
   cidr_blocks = ["0.0.0.0/0"]
 }
}

resource "aws_instance" "web" {
#  ami                         = var.ami_id
 ami                         = "ami-0a7d80731ae1b2435"
 instance_type               = var.instance_type
 key_name                    = aws_key_pair.deployer.key_name
 vpc_security_group_ids      = [aws_security_group.web_sg.id]
 associate_public_ip_address = true
 tags = { Name = "automation-web" }
}
```

Terraform then spits out the public IP for that instance. Ansible picks that up later to start SSH config work. The way these two link up shows how you can chain automation steps without much hassle.

- Comparative Analysis

  Other IaC tools exist as alternatives. AWS CloudFormation stays within AWS and ties in tight. Still, it does not handle multiple clouds well. Pulumi lets you use regular coding languages for IaC. That adds flexibility. But it ramps up the complexity too. Chef Habitat zeros in on app lifecycles. It fits less for straight infrastructure work.

  Terraform strikes a good balance overall. It keeps things simple, supports various providers, and stays modular. That makes it stand out in school projects or real multi cloud DevOps work.

4.2 Ansible. Configuration Management and Orchestration

- Overview

  Ansible handles config management and orchestration. It automates provisioning systems and deploying apps with easy-to-read YAML files known as playbooks. Red Hat developed it. The tool runs without agents. It talks to nodes over SSH. All that keeps it light and simple to slot into setups that already exist.

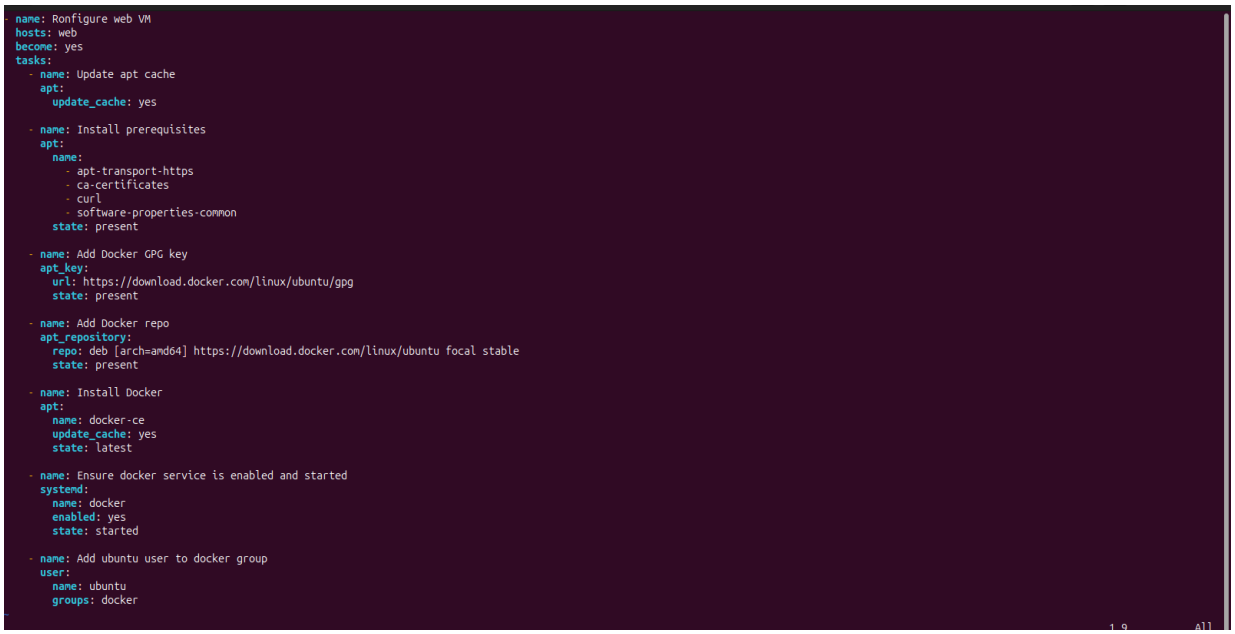
- Theoretical Foundation

Ansible rests on idempotent state handling and declarative ways to orchestrate tasks. Playbooks lay out steps that push the target system to the config you aim for. The modular build means you can reuse a lot through roles and inventories.

- Implementation

  Ansible takes care of three key steps in this project. First, it installs Docker and any needed dependencies. Next, it tweaks user permissions. That means adding the ubuntu user to the Docker group. Finally, it deploys and runs the Flask web app in a container.

  Here is a sample from the playbook.



```yaml
- name: Ronfigure web VM
  hosts: web
  become: yes
  tasks:
    - name: Update apt cache
      apt:
        update_cache: yes

    - name: Install prerequisites
      apt:
        name:
          - apt-transport-https
          - ca-certificates
          - curl
          - software-properties-common
        state: present

    - name: Add Docker GPG key
      apt_key:
        url: https://download.docker.com/linux/ubuntu/gpg
        state: present

    - name: Add Docker repo
      apt_repository:
        repo: deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable
        state: present

    - name: Install Docker
      apt:
        name: docker-ce
        update_cache: yes
        state: latest

    - name: Ensure docker service is enabled and started
      systemd:
        name: docker
        enabled: yes
        state: started

    - name: Add ubuntu user to docker group
      user:
        name: ubuntu
        groups: docker
```

4.3 Docker. Containerization and Application Deployment

- Overview.

  Docker gives developers an isolated setup for packing up and running apps. It helps cut down on those mismatches between dev setups and production ones. That way, apps act the same no matter the host machine.

  For this project, Docker wraps up a Flask web app into something reusable and easy to move around. You can drop it on any machine that runs Docker without much hassle.

- Implementation.

The Docker file lays out exactly how to build the container. It starts with a base Python image and sets up a working directory. Then it copies over the app files and installs all the needed packages through pip. After that, it exposes the port for the web server and sets the command to run the Flask app.

This approach makes sure dependencies get handled during the build. It cuts back on errors that might pop up later when things are running. After the build finishes, the image goes up to Docker Hub. That keeps it ready for pulling into CI/CD flows whenever needed.

- Theoretical Significance.

  On the theory side, containerization pulls the app layer away from the underlying infrastructure. It opens the door to microservices setups, keeps resources separate, and makes scaling straightforward. Compared to old school virtual machines, containers run quicker and use less overhead. They do this by sharing the host's OS kernel instead of duplicating everything.

- Alternatives.

  Podman stands out as a container runtime without a daemon. It runs rootless for better security focus.
  Kubernetes handles big scale orchestration. It goes beyond just single node setups.
  LXC offers lightweight containers. Still, it misses out on Docker's big ecosystem and easy portability.

This project picked Docker for its mix of straightforward use, standard practices, and solid ties to CI/CD tools.

4.4 GitHub Actions. Continuous Integration and Continuous Deployment

- Overview.

  GitHub Actions works as a built-in tool for automating workflows right inside the GitHub platform. Developers write these in YAML format. Triggers come from repo events like commits or pull requests.

  In this setup, GitHub Actions takes care of a few key steps. It builds the Docker image first. Then it pushes that image to DockerHub. Finally, it handles remote deployment over SSH to an AWS EC2 instance.

- Workflow Implementation.

The workflow file sits in the repo under dot github slash workflows. It triggers on pushes to the main branch. Jobs run in sequence, starting with checking out the code. Next comes building the Docker image with the right tags. After that, it logs into DockerHub and pushes the image. The last part sets up SSH and runs deployment commands on the EC2 server. Those commands include pulling the new image and restarting the container service.

Here is a quick look at some tools in this space, their upsides, and downsides. GitHub Actions fits natively with GitHub itself. No need for extra setups, and the free tier works for starters. But concurrency limits hit on that free plan. Jenkins lets you extend it in tons of ways. The catch involves hosting it yourself and tweaking configs. Azure DevOps brings full pipeline options to the table. Vendor lock in with Microsoft stuff is the main drawback though.

GitHub Actions made the cut here because it hooks straight into version control. Every code tweak kicks off an automated deployment that you can track easily.

4.5 Tool Interoperability

This system shines in how the tools play together without friction. Terraform sets up the infrastructure and spits out the EC2 IP address. Ansible grabs that IP to set up and roll out the software. Docker bundles the app so it stays consistent every time. GitHub Actions keeps the whole build and deploy loop running on autopilot.

All this layering creates a full automation chain for the app's life cycle. It needs almost no hands-on work. In a way, it captures that ongoing DevOps push toward total automation.

5. Ubuntu OS and Shell Script Justification

5.1 Choice of Ubuntu as the Operating System

We chose Ubuntu as the main operating system for development and deployment work. It stands out for its long-term stability and wide community backing. Plus, it integrates smoothly with various DevOps tools. Since AWS officially supports Ubuntu, users get preconfigured images that work well on EC2 instances. Setting up SSH connections and cloud-init scripts requires minimal effort. You also have access to the latest versions of packages like Terraform, Ansible, and Docker.

In an academic setting, Ubuntu provides a reliable space for DevOps experiments that can be reproduced easily. Its APT system handles package management in a predictable way. It pairs nicely with Python-based tools such as Ansible. The Long-Term Support versions of Ubuntu deliver five years' worth of updates. That feature suits enterprise-level automation needs pretty well.

Ubuntu sticks to standard Linux practices, which helps with scripting and smooth integrations. Take Ansible modules for apt and service, for example. They connect directly to Ubuntu's systemd setup and package system. All this makes configuration jobs simpler to handle. In the end, Ubuntu serves as a strong foundation for teaching purposes, research efforts, and even production environments.

5.2 Role of Shell Scripting in Orchestration

Tools such as Terraform and Ansible handle the more complex parts of infrastructure management. Still, shell scripts act as the connecting code between those different layers. In our project, the ec2_setup.sh script takes care of the full workflow from start to finish. It begins by initializing Terraform and applying the needed configurations. Then it pulls the EC2 instance's IP address right from Terraform's outputs. After that, it updates the Ansible inventory file on the fly. Finally, it runs the Ansible playbook to set up Docker and launch the container.

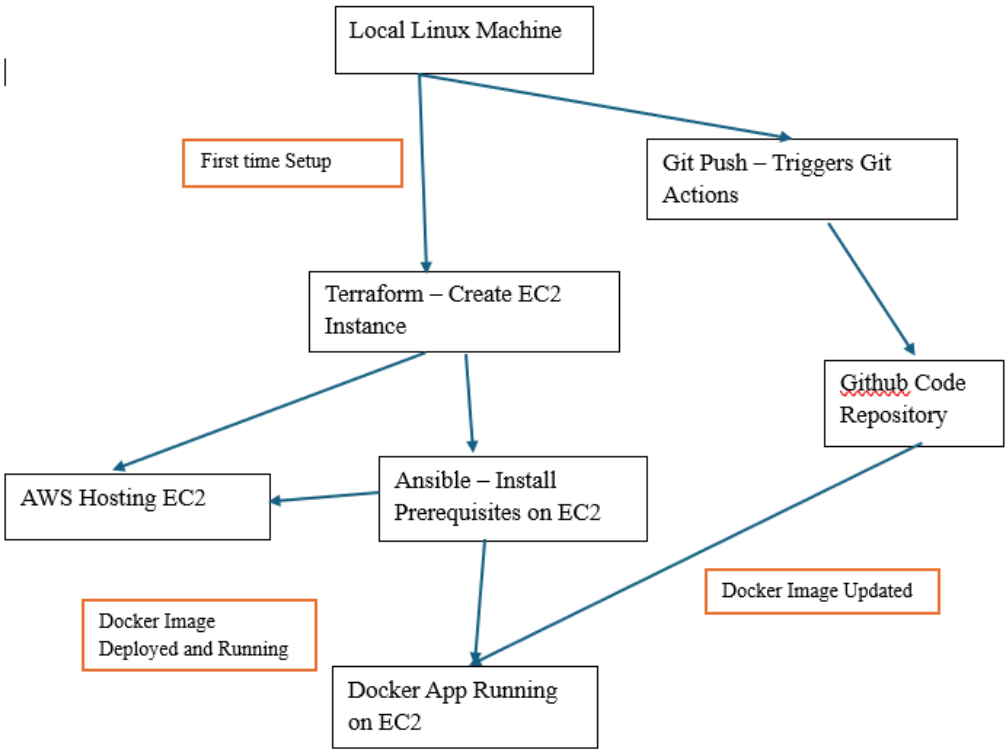
Here is a basic part of that script.

This method cuts out the hassle of manual changes or hunting down IP details. It keeps everything running together without hitches between the tools.

Shell scripts also build in some toughness for automation tasks. They manage error checks through if statements and exit codes. They handle the order of steps too. Sure, full CI/CD setups can take over from these scripts in bigger systems. Even so, the scripts prove useful for local tests, initial bootstrapping, and setting up repeatable processes.

6. Implementation Architecture and Workflow

This part breaks down the project's architecture and workflow in clear steps.



(Figure 1: Overall System Architecture Diagram)

6.1 Architectural Overview

The automation setup includes four key layers that work together.

The Infrastructure Layer uses Terraform to create the EC2 instance, along with the SSH key pair and security group.

The Configuration Layer relies on Ansible to install Docker, set up the environment, and get the container running.

The Application Layer involves Docker to run the Flask app inside a container, with port 80 open for access.

The Pipeline Layer draws on GitHub Actions to manage builds, pushes, and deployments whenever the repository changes.

6.2 Execution Flow

In Step 1, infrastructure creation happens through Terraform. It deploys an EC2 instance based on the given AMI and SSH key. The public IP address gets saved as an output variable for later use.

Step 2 focuses on server configuration. Ansible takes that IP address and uses it to install Docker. It also gets the system ready for the next parts.

During Step 3, container deployment kicks in. The Flask app starts up in its container on port 80. That setup serves a basic web interface to users.

Step 4 brings in CI/CD integration. GitHub Actions builds the Docker image each time code gets pushed. It updates Docker Hub with the new image. Then it redeploys everything to the EC2 instance through secure SSH connections.

6.3 Security Considerations

For SSH key management, private keys stay safe in GitHub Secrets. Terraform only pulls in the public key when needed.

Network security comes from security groups that limit SSH access to trusted IP addresses. HTTP traffic stays open to everyone else.

Secret handling uses environment variables for things like Docker Hub credentials, the EC2 host details, and SSH keys. GitHub's secret management keeps them encrypted.

This setup with its layers creates a secure path for automated deployments. It draws clear lines between provisioning steps, configuration work, and actual deployment.

We dealt with several key challenges in setting up the infrastructure. The first one involved terraform state management. That local terraform.tfstate file carried risks of exposure along with desynchronization issues over time. To handle it, we excluded the file through .gitignore right away. We also set up a remote S3 backend specifically for production to ensure better safety overall.

Another problem came up with dynamic IP management. The EC2 public IP would shift around with every single deployment. This broke all the static references in our Ansible inventory pretty quickly. Our fix involved automating the IP substitution directly in Ansible's inventory. We used some shell scripting to make that process run smoothly without manual tweaks.

Docker permission errors popped up next. Docker needed root privileges just to carry out certain operations reliably. In the Ansible playbook, we added the ubuntu user straight to the Docker group. That change let things operate without always escalating to root level.

SSH access failures happened too, especially in GitHub Actions. Those workflows would occasionally fail at authentication steps during deployments. We configured the appleboy slash ssh-action to use PEM-based authentication instead. We adjusted the instance permissions accordingly to avoid those hiccups going forward.

Finally, cost optimization was a concern worth addressing. Keeping the EC2 instances running continuously just racked up unnecessary expenses day after day. We added a scheduled shutdown for the containers after about four hours. This relied on Ansible's at module to trigger the stop automatically when idle time hit.

7. Evaluation, and Future Improvements

7.1 Evaluation Metrics

The average full deployment using Terraform, Ansible, and Docker usually finishes in four to five minutes.

Multiple runs showed the same infrastructure and application behavior every time. This proves the setup works idempotently.

SSH access stays limited to known IP addresses. Secrets get protected through encryption methods.

Terraform and Ansible scripts come modularized. They support deploying multiple instances easily.

If one stage fails, like Ansible, you can retry it on its own. No need to tear down the whole infrastructure.

7.2 Academic and Practical Significance

This project puts DevOps theory into real practice. It shows how Infrastructure as Code and configuration management work together in actual settings.

On the practical side, it copies the deployment pipelines that big companies like Netflix, Amazon, and Google Cloud use.

The setup offers a repeatable template for teaching DevOps automation. Small teams can reach enterprise level using free open source tools.

7.3 Future Improvements

You could integrate with Kubernetes or ECS next. That would handle multi container orchestration for better scaling.

Add monitoring and logging tools like Prometheus and Grafana. They would let you watch the system live.

Bring in more testing, such as unit and integration tests. Fit them right into the CI/CD process.

Centralize secrets using AWS Systems Manager Parameter Store. That handles credential management securely.

For infrastructure, add auto scaling groups in Terraform. They respond to load changes with elasticity.

8. Conclusion

This project pulls off a full automation pipeline. It follows DevOps principles, Infrastructure as Code, and Continuous Delivery closely.

Terraform, Ansible, Docker, and GitHub Actions all tie together here. They cover the entire lifecycle from provisioning and setup to deploying apps and handling updates.

Academically, it applies software engineering and cloud computing ideas across fields. Terraform shows declarative Infrastructure as Code in action. Ansible handles procedural configuration management. Docker keeps runtime consistent. GitHub Actions manages the automation and feedback. All of this points to how automation and reliability come together in today’s DevOps world.

Future work might look at hybrid cloud setups or stronger security. It could also try serverless options. The projects modular design means it adapts well to microservices, ongoing testing, or watching infrastructure.

9. References

- HashiCorp. (2024). *Terraform Documentation*. https://developer.hashicorp.com/terraform
- Red Hat. (2024). *Ansible Documentation*. https://docs.ansible.com/
- Docker Inc. (2024). *Docker Reference Documentation*. https://docs.docker.com/
- GitHub. (2024). *GitHub Actions Developer Guide*. https://docs.github.com/en/actions
- Amazon Web Services. (2024). *AWS EC2 and Security Best Practices*. https://aws.amazon.com/ec2

Materials

Github Repo - https://github.com/mdhake1-dbs/network-sys-assessment.git

Video Demo -