

School: CLOUDBOOSTA	Course: DEVOPS
Name: AKOSA SAMUEL ONYEJEKWE	Module: JENKINS
	Class 3: Assignment_ Jenkins
Student No: CBALC0168	Issue Date: May 9, 2026

Contents

GitHub Repo Link for this Exercise.....	3
Learning Objectives.....	3
Case Study	4
Task 1: Creating the Infrastructure-as-Code (IaC) Repository.....	7
Terraform Project File Structure and Purpose	10
Task 2: Setting Up Local Git Setup.....	15
Task 3: Creating the GitHub Repository.....	15
Task 4: Storing Credentials in GitHub Secrets.....	16
Task 5: Connecting Local Repo, and Pushing to GitHub	19
5.1: Watching Workflow Run.....	19
5.2: Fixing Terraform Backend Lock Table Error.....	20
5.2.1: The Fix.....	22
5.3: Fixing Error After Running Terraform workflow directory Workflow	26
Task 6: Creating deploy.sh and destroy.sh files, and pushing these files to GitHub....	29
deploy.sh.....	30
destroy.sh.....	30
Adding deployment and destroy automation scripts.....	30
Adding Terraform destroy workflow.....	32
Added Terraform destroy workflow.....	33
Task 7: Provisioning and configuring a Jenkins Server.....	35
7.1: Create a Custom VPC and Subnets.....	35
7.2: Launch an instance within the custom public subnet	39
Task 8: Jenkins Initial Configuration and Administrator Setup	42
Task 9: GitHub Integration	45
9.1: Generating Token from GitHub.....	45
9.2: Creating a GitHub Credential in Jenkins	47
9.3: Generating a Public Key from Jenkins Instance and adding it to GitHub	49

Task 10: Creating Jenkins Pipeline Job.....	54
10.1: Create New Jenkins Job.....	54
Task 11: Connecting GitHub Repository to Jenkins Pipeline	55
Task 12: Creating Jenkinsfile for Terraform Pipeline	57
12.1: Create Jenkinsfile Locally	57
Task 13: Push Jenkinsfile to GitHub	57
Task 14: Configuring AWS Credentials for Jenkins User.....	58
14.1: Install AWS Credentials Plugin.....	58
14.2: Add AWS Credentials to Jenkins	59
14.3: Enter AWS Keys.....	60
Task 15: Test Pipeline Manually	61
15.1: Build Now.....	61
Task 16: Realistic Infrastructure Test	62
16.1: Modified Terraform	63
Task 17: Pushing Terraform Change	63
Task 18: Configure GitHub Webhook.....	66
Task 19: Enabling Webhook in Jenkins Job Configuration.....	68
Task 20: Testing the Webhook	69
20.1: In GitHub	71
20.2 In Jenkins	72
Task 21: Explaining Error that Occurred after Testing Webhook	72
21.1: Fixing the Error.....	74
21.2: Triggering Jenkins Again.....	79
Jenkins Build History Explained.....	81
Jenkinsfile Integration and CI/CD Automation	89
The GitHub Actions Flow	92
Terraform Layer.....	92
GitHub Actions Layer.....	92
Production Workflow	92
Terraform GitHub CI/CD Pipeline Workflow Runs.....	95
Conclusion	97

GitHub Repo Link for this Exercise

<https://github.com/samuelakosaonyekwe/terraform-s3-github-actions/actions>

Learning Objectives

Based on this Jenkins, Terraform, GitHub Actions, and AWS DevOps project, the following learning objectives were achieved:

- Understand the principles of Infrastructure-as-Code (IaC) using Terraform for automated cloud infrastructure provisioning.
- Develop and manage a production-style Terraform repository structure using modular and reusable configurations.
- Configure and automate cloud infrastructure deployment on Amazon Web Services using Terraform modules and remote state management.
- Implement CI/CD automation workflows using GitHub Actions for Terraform validation, planning, deployment, and destroy operations.
- Understand the importance of Terraform backend configuration using S3 remote state storage and DynamoDB state locking mechanisms.
- Troubleshoot Terraform backend and state-locking issues including DynamoDB lock conflicts and backend bucket configuration errors.
- Apply Linux command-line operations within an Ubuntu WSL environment for DevOps automation and infrastructure management.
- Configure secure credential management using GitHub repository secrets and Jenkins credential storage mechanisms.
- Build reusable deployment automation scripts using Bash shell scripting for Terraform deployment and infrastructure destruction workflows.
- Provision and configure a Jenkins server on AWS EC2 for CI/CD pipeline automation.
- Integrate Jenkins with GitHub repositories using SSH authentication, Personal Access Tokens (PATs), and GitHub webhooks.
- Create and configure Jenkins Pipeline jobs for automated Terraform infrastructure deployment.
- Develop and manage a Jenkinsfile for Infrastructure-as-Code CI/CD automation pipelines.
- Configure AWS credentials securely within Jenkins for Terraform execution and infrastructure provisioning.

- Understand webhook-based automation workflows between GitHub and Jenkins for real-time CI/CD triggering.
- Identify and resolve concurrency and Terraform state-lock issues caused by multiple CI/CD systems accessing the same backend resources simultaneously.
- Gain practical experience with DevOps workflow orchestration involving Terraform, GitHub Actions, Jenkins, AWS, Git, and Linux automation tools.
- Understand enterprise-grade DevOps practices including modularity, automation, infrastructure lifecycle management, state protection, and CI/CD pipeline integration.
- Demonstrate the ability to provision, modify, validate, deploy, and destroy cloud infrastructure resources automatically using modern DevOps methodologies.
- Develop hands-on experience in implementing scalable, reusable, and production-aligned Infrastructure-as-Code and CI/CD automation workflows.

Case Study

This project presents a real-world DevOps case study focused on implementing a production-style Infrastructure-as-Code (IaC) and CI/CD automation pipeline using Terraform, GitHub Actions, and Jenkins for automated AWS infrastructure provisioning and management.

The organisation scenario assumes a cloud engineering team responsible for automating AWS infrastructure deployments in a secure, scalable, and enterprise-aligned manner. Traditionally, infrastructure provisioning was performed manually through the AWS Management Console, resulting in:

- inconsistent deployments,
- configuration drift,
- reduced scalability,
- lack of auditability,
- increased deployment time,
- and higher risk of human error.

To solve these problems, a modular Terraform-based IaC repository was designed and integrated with GitHub Actions and Jenkins CI/CD pipelines to fully automate infrastructure deployment, validation, testing, and destruction workflows.

The solution architecture consists of:

- Terraform modules for reusable AWS infrastructure provisioning,
- GitHub repository integration for version control,
- GitHub Actions workflows for automated Terraform execution,

- Jenkins pipeline integration for enterprise CI/CD orchestration,
- remote Terraform backend using AWS S3,
- DynamoDB state locking for concurrency protection,
- deployment and destroy automation scripts,
- webhook-based automation triggers,
- and AWS credential management through secure secrets handling.

The primary infrastructure resource provisioned in this implementation was AWS S3 buckets created dynamically through Terraform modules. The Terraform configuration was structured using a production-grade repository hierarchy separating:

- reusable modules,
- workflow automation,
- scripts,
- backend configuration,
- provider configuration,
- variables,
- outputs,
- and CI/CD pipeline definitions.

The GitHub Actions pipeline automated:

- terraform init,
 - terraform validate,
 - terraform plan,
 - terraform apply,
 - and terraform destroy
- operations whenever code changes were pushed to the repository. Jenkins was later integrated as an enterprise-grade CI/CD orchestration platform capable of automatically triggering Terraform deployments using GitHub webhooks.

During implementation, several real-world DevOps and infrastructure challenges were encountered and resolved, including:

- Terraform backend configuration failures,
- missing DynamoDB lock table issues,
- remote state management problems,

- S3 bucket naming conflicts,
- GitHub authentication integration,
- Jenkins SSH credential configuration,
- webhook triggering problems,
- and Terraform state-lock concurrency conflicts between Jenkins and GitHub Actions.

The project demonstrated practical enterprise DevOps concepts including:

- Infrastructure-as-Code (IaC),
- CI/CD automation,
- GitOps workflows,
- Terraform state management,
- modular infrastructure design,
- deployment automation,
- remote backend configuration,
- infrastructure lifecycle management,
- webhook-driven automation,
- Jenkins pipeline orchestration,
- and secure credential management.

The final implementation successfully achieved:

- automated AWS infrastructure provisioning,
- automated infrastructure destruction,
- GitHub-integrated CI/CD workflows,
- Jenkins-triggered Terraform pipelines,
- secure Terraform state locking,
- production-style repository organisation,
- and enterprise-aligned DevOps automation practices.

This case study demonstrates how modern organisations can adopt DevOps methodologies, Terraform Infrastructure-as-Code, GitHub Actions, and Jenkins automation to achieve scalable, reliable, repeatable, and production-ready cloud infrastructure management on AWS.

Task 1: Creating the Infrastructure-as-Code (IaC) Repository

I implemented a modular Terraform IaC repository integrated with GitHub Actions CI/CD automation for automated AWS S3 provisioning and teardown.

As such, I created the IaC files, modular Terraform configuration files, shell automation scripts, and GitHub Actions workflow files for automated AWS S3 provisioning.

Each file (empty) was created at once by running Linux bash commands on the Ubuntu terminal, as can be seen in the following screenshot.

```
akosa@AKOSA:~/terraform-s3-github-actions$ # CREATE PROJECT ROOT
# =====

mkdir -p ~/terraform-s3-github-actions

cd ~/terraform-s3-github-actions

# =====
# ROOT FILES
# =====

touch README.md .gitignore

# =====
# TERRAFORM CORE STRUCTURE
# =====

mkdir -p terraform

cd terraform

touch main.tf \
variables.tf \
outputs.tf \
providers.tf \
backend.tf \
versions.tf \
terraform.tfvars

# =====
# TERRAFORM MODULES
# =====

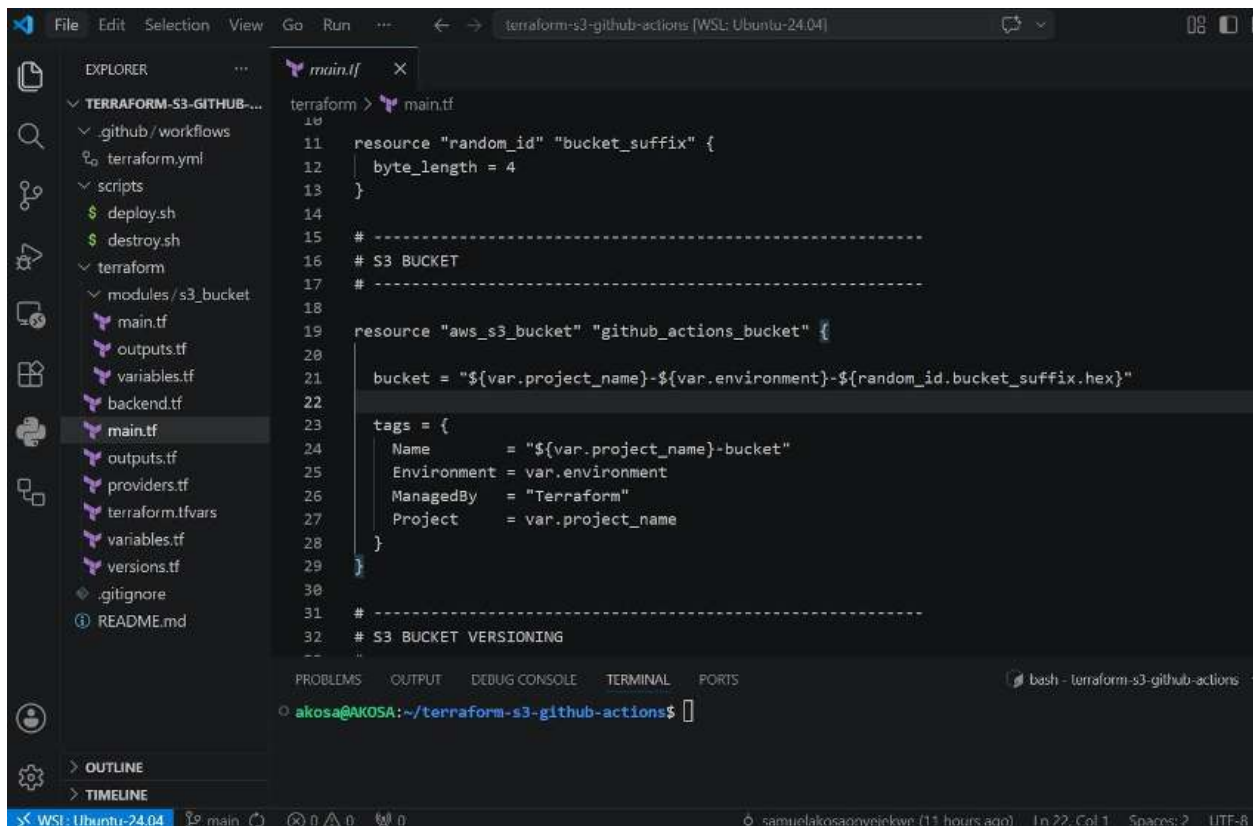
touch .github/workflows/terraform.yml=====
akosa@AKOSA:~/terraform-s3-github-actions$ |
```

The content of each file was created from scratch on Visual Studio Code, running Ubuntu through WSL (Windows Subsystem for Linux).

The following screenshot shows a production-grade Terraform Infrastructure-as-Code repository structure for the assignment.

The current tree structure is:

```
terraform-s3-github-actions/  
├── .github/  
│   └── workflows/  
│       └── terraform.yml  
├── scripts/  
│   ├── deploy.sh  
│   └── destroy.sh  
├── terraform/  
│   ├── backend.tf  
│   ├── main.tf  
│   ├── outputs.tf  
│   ├── providers.tf  
│   ├── terraform.tfvars  
│   ├── variables.tf  
│   └── versions.tf  
│   └── modules/  
│       └── s3_bucket/  
│           ├── main.tf  
│           ├── outputs.tf  
│           └── variables.tf  
├── .gitignore  
└── README.md
```



This structure is considered:

- Modular
- Reusable
- CI/CD-ready
- Production-style
- Enterprise-aligned

because it separates:

- automation (.github/workflows)
- scripts (scripts/)

Terraform core configs (terraform/)

- reusable modules (modules/)
- documentation (README.md)
- git exclusions (.gitignore)

Terraform Project File Structure and Purpose

terraform-s3-github-actions/ - Project Root

- Main project folder
- Contains:
 - Terraform infrastructure code
 - GitHub Actions CI/CD
 - Deployment scripts
 - Documentation

Root Files

README.md

- Project documentation
- Explains:
 - architecture
 - deployment steps
 - usage
 - workflow
 - technologies used

.gitignore

- Prevents unnecessary/sensitive files from being pushed to GitHub
- Common ignored files:
 - .terraform/
 - terraform.tfstate
 - secrets
 - cache files

scripts/ Directory

- Stores shell automation scripts

scripts/deploy.sh

- Automates Terraform deployment
- Typically runs:
 - terraform init
 - terraform validate
 - terraform plan
 - terraform apply

scripts/destroy.sh

- Automates infrastructure destruction

- Typically runs:
 - `terraform destroy -auto-approve`

`terraform/` Directory

- Main Terraform infrastructure folder

`terraform/main.tf`

- Core infrastructure definitions
- Creates AWS resources/modules

`terraform/variables.tf`

- Defines Terraform input variables
- Makes code reusable

`terraform/outputs.tf`

- Displays useful outputs after deployment
- Example:

- bucket name
- ARN
- endpoint

`terraform/providers.tf`

- Configures AWS provider
- Connects Terraform to AWS

`terraform/backend.tf`

- Configures remote Terraform state backend
- Usually uses S3

`terraform/versions.tf`

- Defines:
 - Terraform version
 - provider versions

`terraform/terraform.tfvars`

- Stores actual variable values
- Example:
 - project name
 - environment
 - region

`terraform/modules/`

- Stores reusable Terraform modules
 - `terraform/modules/s3_bucket/`
 - Reusable S3 bucket module

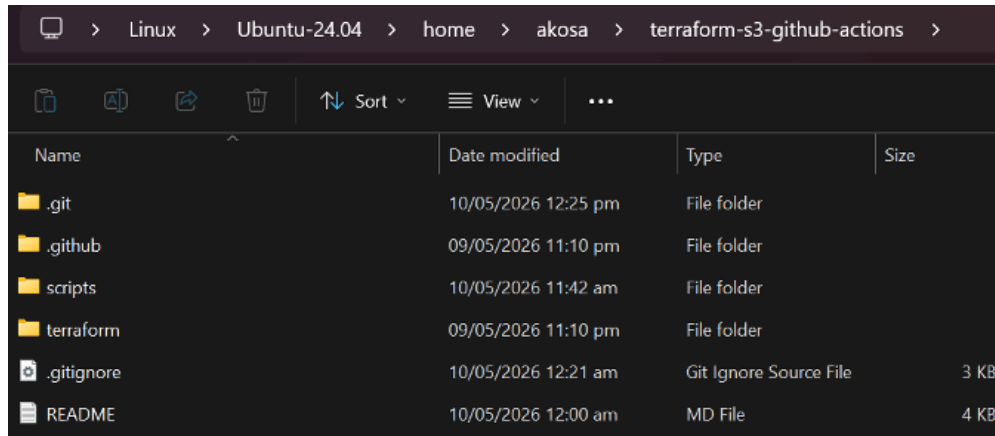
`terraform/modules/s3_bucket/main.tf`

- Contains S3 bucket resource logic
 - `terraform/modules/s3_bucket/variables.tf`
 - Defines module input variables
 - `terraform/modules/s3_bucket/outputs.tf`
 - Exposes S3 outputs to root Terraform

`.github/`

- GitHub configuration directory
 - `.github/workflows/`
 - Stores GitHub Actions workflows
 - `.github/workflows/terraform.yml`
 - Main CI/CD pipeline
 - Automates:
 - Terraform Init
 - Validate
 - Plan
 - Apply
 - Destroy
 - Triggered by:
 - push
 - pull request
 - manual workflow dispatch

The following screenshot shows that the IaC repository files and folders were successfully created and saved in the local Linux home directory within the Ubuntu WSL environment.

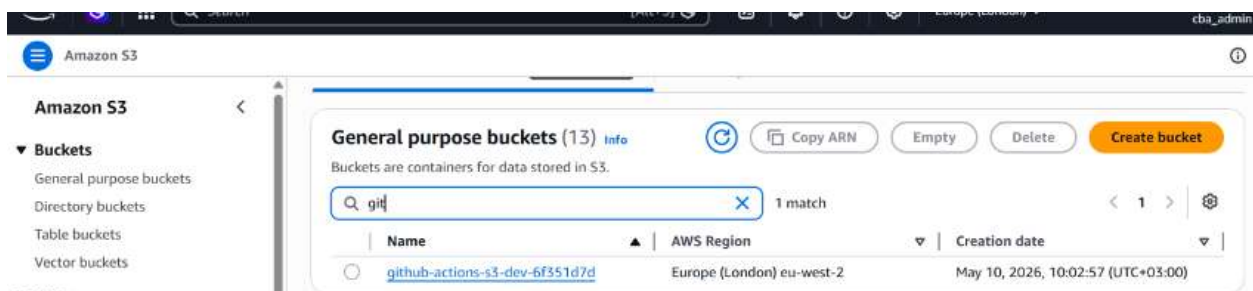


For this assignment, as shown in the following screenshot, I defined the main application S3 bucket name in the terraform/terraform.tfvars file as my preferred bucket prefix name:

```
project_name = "github-actions-s3"
```

Eventually, my final bucket name created by Terraform was this:

github-actions-s3-dev-6f351d7d as can be verified in AWS console. Please see screenshot just below,



because Terraform automatically adds:

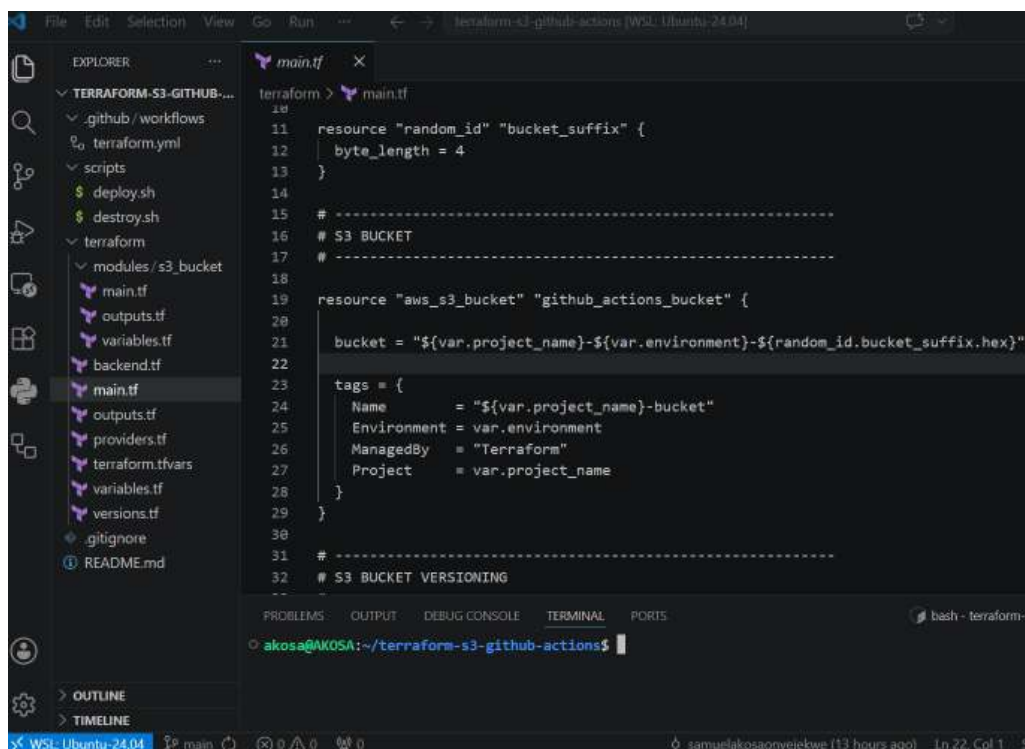
- environment
- random suffix

using this:

```
bucket = "${var.project_name}-${var.environment}-${random_id.bucket_suffix.hex}"
```

as can be seen in lines 11 and 21 of each of the two respective screenshots that follow.

```
terraform.tfvars X
terraform > terraform.tfvars
1  # =====
2  # TERRAFORM VARIABLES VALUES
3  # FILE: terraform.tfvars
4  # PROJECT: S3 Bucket with GitHub Actions Automation
5  # =====
6
7  # -----
8  # PROJECT CONFIGURATION
9  # -----
10
11 project_name = "github-actions-s3"
12
13 environment = "dev"
14
15 # -----
16 # AWS CONFIGURATION
17 # -----
18
19 aws_region = "eu-west-2"
```



The screenshot shows the Visual Studio Code interface with a Terraform project. The Explorer sidebar on the left displays the project structure, including a 'modules/s3_bucket' directory containing 'main.tf', 'outputs.tf', 'providers.tf', 'terraform.tfvars', 'variables.tf', and 'versions.tf'. The main editor window shows the 'main.tf' file with Terraform code. The code defines a 'random_id' resource to generate a bucket suffix and an 'aws_s3_bucket' resource named 'github_actions_bucket' with a specific bucket name and tags. The terminal at the bottom shows the user 'akosa@AKOSA' in the directory '~/terraform-s3-github-actions'.

```
main.tf X
terraform > main.tf
11 resource "random_id" "bucket_suffix" {
12   byte_length = 4
13 }
14
15 # -----
16 # S3 BUCKET
17 # -----
18
19 resource "aws_s3_bucket" "github_actions_bucket" {
20   bucket = "${var.project_name}-${var.environment}-${random_id.bucket_suffix.hex}"
21
22   tags = {
23     Name       = "${var.project_name}-bucket"
24     Environment = var.environment
25     ManagedBy  = "Terraform"
26     Project    = var.project_name
27   }
28 }
29
30 # -----
31 # S3 BUCKET VERSIONING
32 # -----
```

This is effectively good production-grade practice because:

- S3 bucket names must be globally unique
- random suffix prevents collisions
- environment separation is clean
- avoids manual naming conflicts

Task 2: Setting Up Local Git Setup

On the Ubuntu terminal, I ran:

```
cd ~/terraform-s3-github-actions
git init
git add .
git commit -m "Initial commit"
```

This:

- initialises git locally
- tracks my files
- creates my first commit

Task 3: Creating the GitHub Repository

Following are steps taken to create the GitHub repo as seen in the screenshot below.

- Opened <https://github.com> and signed in to my GitHub account.
- Clicked the “+” icon at the top-right corner of the GitHub dashboard.
- Selected “**New repository**” from the dropdown menu.
- Under **Owner**, selected my GitHub account.
- Entered the repository name:
 terraform-s3-github-actions
- Entered a repository description explaining the purpose of the project.
- Chose the repository visibility:
 - **Public** → visible to everyone
- Left “**Add README**” unchecked because the README file was already created locally.
- Left “**.gitignore**” unchecked because the .gitignore file already exists locally.

- Left “**License**” as **No license**.
- Clicked the green “**Create repository**” button to create the GitHub repository.
- After creation, GitHub generated an empty remote repository ready to be linked to the local Terraform IaC repository using Git commands.

Create a new repository

Repositories contain a project's files and version history. Have a project elsewhere? [Import a repository](#).
Required fields are marked with an asterisk (*).

1 General

Owner * 👤 samuelakosaonyekwe / Repository name *
✔ terraform-s3-github-actions is available.

Great repository names are short and memorable. How about [verbose-waffle?](#)

Description
 182 / 350 characters

2 Configuration

Choose visibility * 📖 Public
 Choose who can see and commit to this repository

Add README Off
 READMEs can be used as longer descriptions. [About READMEs](#)

Add .gitignore No .gitignore
 .gitignore tells git which files not to track. [About ignoring files](#)

Add license No license
 Licenses explain how others can use your code. [About licenses](#)

Create repository

© 2026 GitHub, Inc. [Terms](#) [Privacy](#) [Security](#) [Status](#) [Community](#) [Docs](#) [Contact](#) [Manage cookies](#) [Do not share my personal information](#)

Task 4: Storing Credentials in GitHub Secrets

Following are steps I took to store my credentials in GitHub secrets.

- Opened the GitHub repository in my browser.
- Clicked the **Settings** tab at the top of the repository page.
- In the left sidebar, navigated to:
Security and quality → Secrets and variables → Actions
- Under the **Secrets** tab, located the **Repository secrets** section.
- Clicked the green “**New repository secret**” button.

- Created the first secret:
 - Name:
AWS_ACCESS_KEY_ID
 - Value:
My AWS Access Key ID
- Clicked “**Add secret**” to save it.
- Repeated the process for the second secret:
 - Name:
AWS_SECRET_ACCESS_KEY
 - Value:
My AWS Secret Access Key
- Clicked “**Add secret**” again.
- Repeated the process for the third secret:
 - Name:
AWS_REGION
 - Value:
My AWS deployment region (eu-west-2)
- Clicked “**Add secret**” to save the region value.
- After completion, the repository secrets became securely stored and encrypted within GitHub Actions.
- These secrets are then automatically used by the CI/CD workflow (terraform.yml) during Terraform deployment and infrastructure automation.

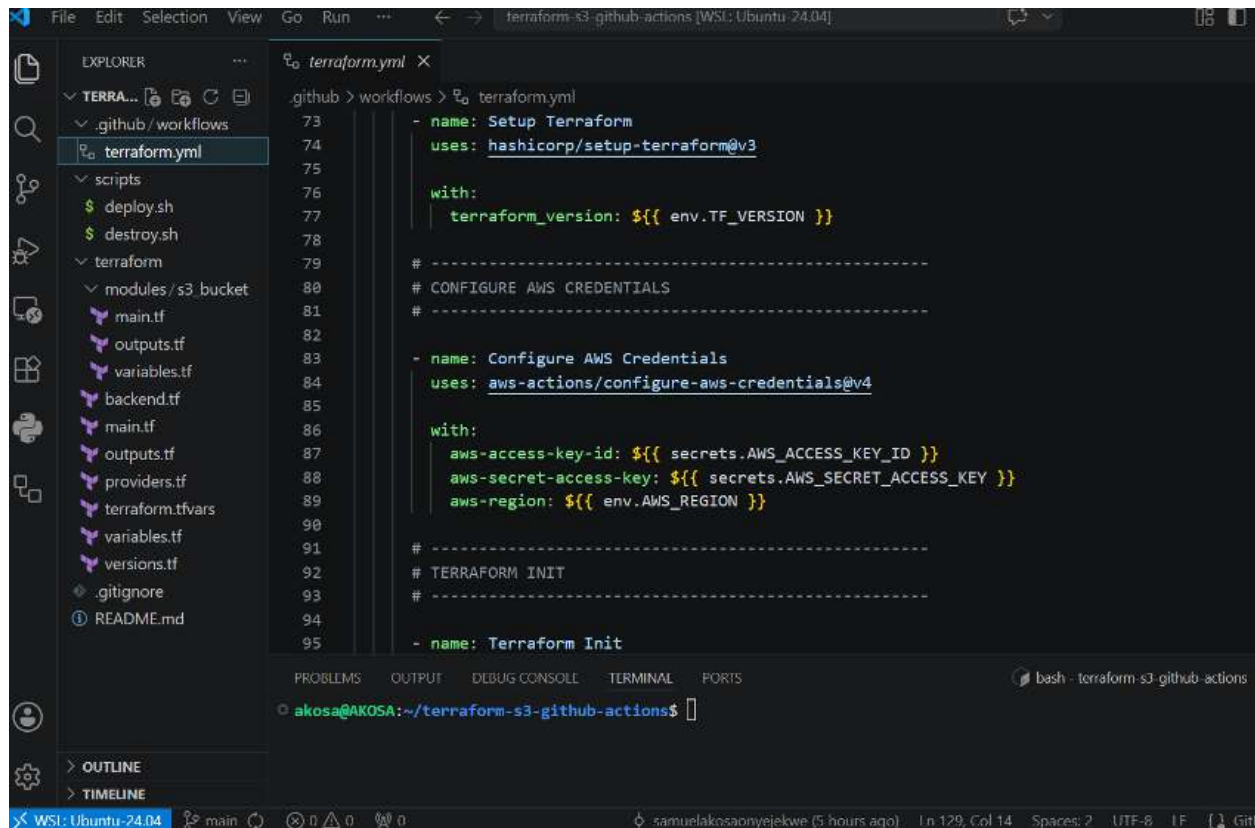
See the screenshot that follows.

The screenshot shows the GitHub Actions 'Secrets and variables' page. The left sidebar contains navigation links: General, Access, Collaborators, Moderation options, Code and automation (Branches, Tags, Rules, Actions, Models, Webhooks, Copilot, Environments, Codespaces, Pages), Security and quality (Advanced Security, Deploy keys, Secrets and variables, Actions, Agents, Codespaces, Dependabot), and Integrations (GitHub Apps, Email notifications). The main content area is titled 'Actions secrets and variables' and includes a description of secrets and variables, tabs for 'Secrets' and 'Variables', and sections for 'Environment secrets' (showing 'This environment has no secrets') and 'Repository secrets'.

Repository secrets

Name	Last updated
AMS_ACCESS_KEY_ID	3 minutes ago
AMS_REGION	now
AMS_SECRET_ACCESS_KEY	2 minutes ago

The opened terraform.yml workflow file, just below, is configured to automate Terraform operations and securely authenticate to AWS using GitHub repository secrets. (Please see lines 80 to 89).



Task 5: Connecting Local Repo, and Pushing to GitHub

I ran the following Linux commands on my Ubuntu terminal to connect local repo to GitHub.

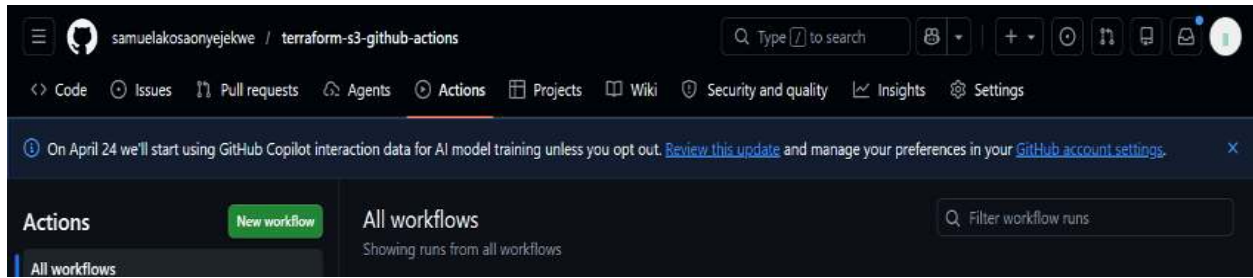
```
git remote add origin https://github.com/samuelakosaonyekwe/terraform-s3-github-actions.git
```

```
git branch -M main
```

```
git push -u origin main
```

5.1: Watching Workflow Run

- Went to my repository on GitHub
- Clicked the Actions tab



5.2: Fixing Terraform Backend Lock Table Error

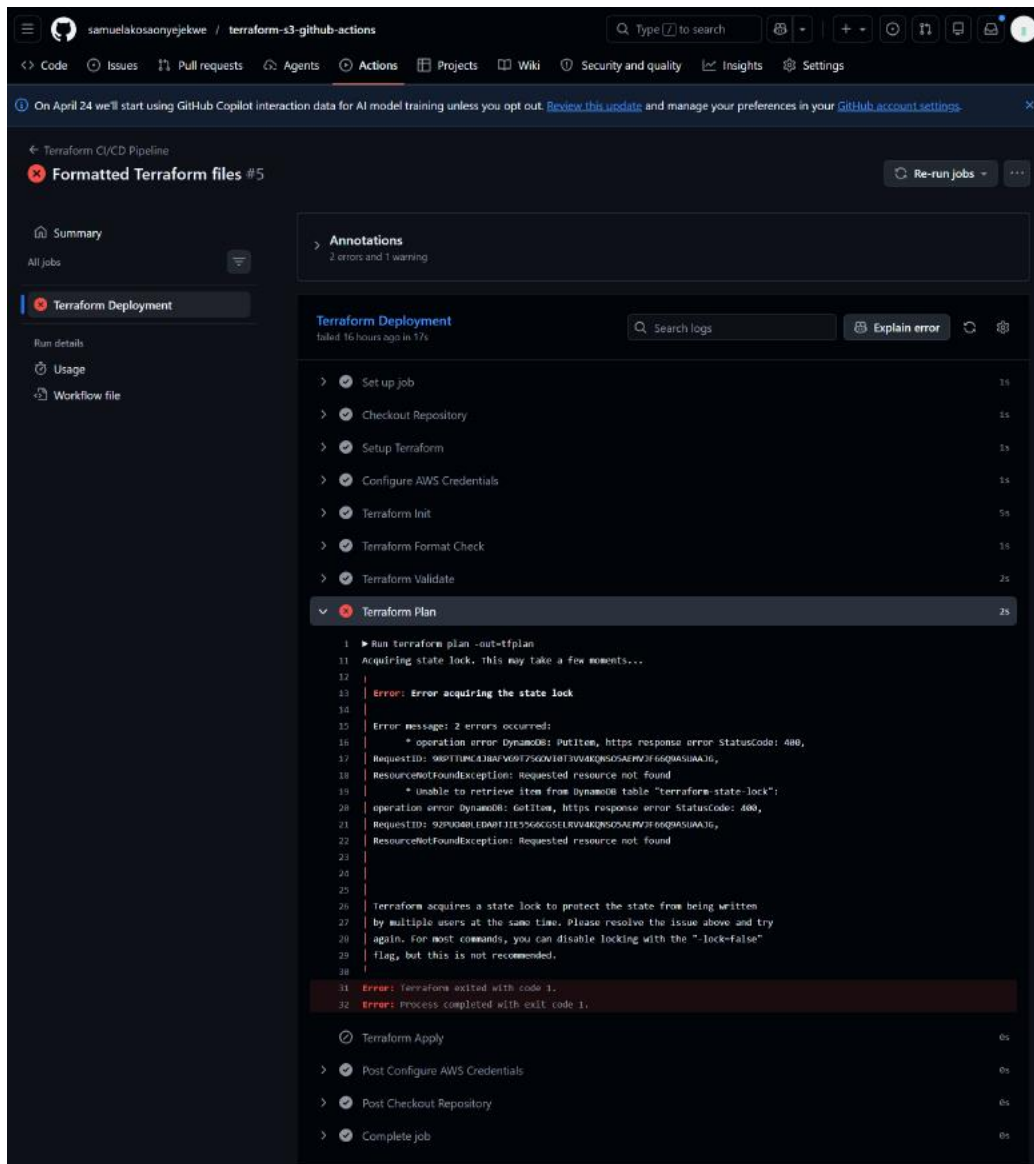
Earlier, my Formatted Terraform files workflow failed because my backend configuration was trying to use a DynamoDB table for Terraform state locking, but the table did not exist yet.

As can be seen in the screenshot that just follows, the key error is:

```
Unable to retrieve item from DynamoDB table "terraform-state-lock"
```

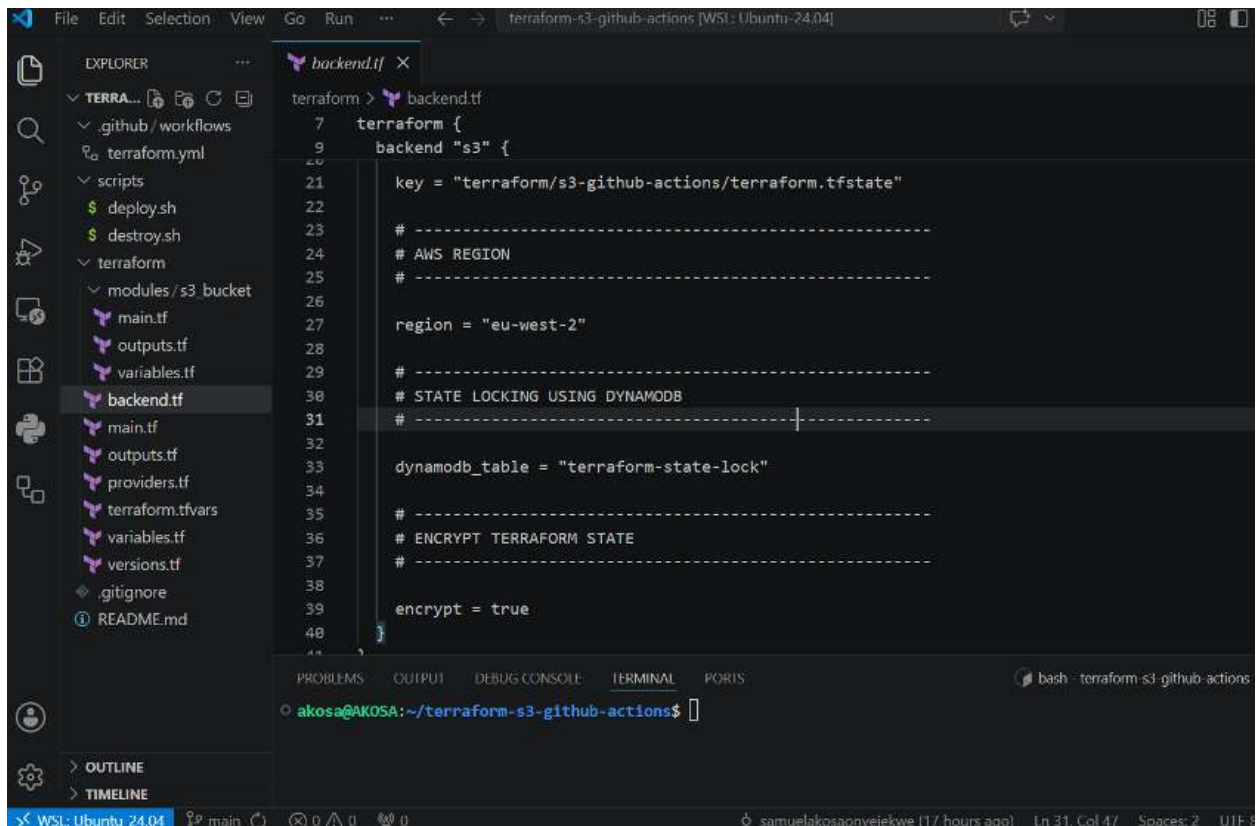
and:

```
ResourceNotFoundException
```



My backend.tf file, as seen below in line 33, contains this:

```
dynamodb_table = "terraform-state-lock"
```



Terraform uses this table to:

- prevent simultaneous state modifications
- avoid state corruption
- implement state locking

But the DynamoDB table must already exist before Terraform starts.

5.2.1: The Fix

Step 1: Create The DynamoDB Lock Table

I ran the following Linux command on Ubuntu terminal:

```
aws dynamodb create-table \  
  --table-name terraform-state-lock \  
  --attribute-definitions AttributeName=LockID,AttributeType=S \  
  --key-schema AttributeName=LockID,KeyType=HASH \  
  --billing-mode PAY_PER_REQUEST \  
  --region eu-west-2
```

```

akosa@AKOSA:~/terraform-s3-github-actions$ aws dynamodb create-table \
--table-name terraform-state-lock \
--attribute-definitions AttributeName=LockID,AttributeType=S \
--key-schema AttributeName=LockID,KeyType=HASH \
--billing-mode PAY_PER_REQUEST \
--region eu-west-2
{
  "TableDescription": {
    "AttributeDefinitions": [
      {
        "AttributeName": "LockID",
        "AttributeType": "S"
      }
    ],
    "TableName": "terraform-state-lock",
    "KeySchema": [
      {
        "AttributeName": "LockID",
        "KeyType": "HASH"
      }
    ],
    "TableStatus": "CREATING",
    "CreationDateTime": "2026-05-10T09:57:11.920000+03:00",
    "ProvisionedThroughput": {
      "NumberOfDecreasesToday": 0,
      "ReadCapacityUnits": 0,
      "WriteCapacityUnits": 0
    },
    "TableSizeBytes": 0,
    "ItemCount": 0,
    "TableArn": "arn:aws:dynamodb:eu-west-2:009850210027:table/terraform-state-lock",
    "TableId": "5c09a856-a7c0-4724-8b7d-e78228ed183d",
    "BillingModeSummary": {

```

Step 2: Verify Table Creation

I ran:

```

aws dynamodb describe-table \
--table-name terraform-state-lock \
--region eu-west-2

```

And then I saw:

```

"TableStatus": "ACTIVE"

```

```

akosa@AKOSA:~$ aws dynamodb describe-table \
--table-name terraform-state-lock \
--region eu-west-2
{
  "Table": {
    "AttributeDefinitions": [
      {
        "AttributeName": "LockID",
        "AttributeType": "S"
      }
    ],
    "TableName": "terraform-state-lock",
    "KeySchema": [
      {
        "AttributeName": "LockID",
        "KeyType": "HASH"
      }
    ],
    "TableStatus": "ACTIVE",
    "CreationDateTime": "2026-05-10T09:57:11.920000+03:00",
    "ProvisionedThroughput": {
      "NumberOfDecreasesToday": 0,
      "ReadCapacityUnits": 0,
      "WriteCapacityUnits": 0
    },
    "TableSizeBytes": 0,
    "ItemCount": 0,
    "TableArn": "arn:aws:dynamodb:eu-west-2:009850210027:table/terraform-state-lock",
    "TableId": "5c09a856-a7c0-4724-8b7d-e78228ed183d",
    "BillingModeSummary": {
      "BillingMode": "PAY_PER_REQUEST",
      "LastUpdateToPayPerRequestDateTime": "2026-05-10T09:57:11.920000+03:00"
    }
  },
}

```

Step 3: Rerun Workflow as Trigger Workflow rerun

I ran on Ubuntu terminal for workflow rerun:

```

cd ~/terraform-s3-github-actions
git commit --allow-empty -m "Trigger workflow rerun"
git push

```

See the following screenshot.

On April 24 we'll start using GitHub Copilot interaction data for AI model training unless you opt out. [Review this update](#) and manage your preferences in your [GitHub account settings](#).

← Terraform CI/CD Pipeline

Trigger workflow rerun #6 Re-run all jobs ...

Summary

All jobs

Terraform Deployment

Run details

Usage

Workflow file

Annotations

1 warning

Terraform Deployment

suspended 11 hours ago in 1m 42s

Search logs

- Set up job 2s
- Checkout Repository 1s
- Setup Terraform 1s
- Configure AWS Credentials 1s
- Terraform Init 7s
- Terraform Format Check 8s
- Terraform Validate 2s
- Terraform Plan 5s
- Terraform Apply 1m 19s
- Post Configure AWS Credentials 8s
- Post Checkout Repository 1s
- Complete job 0s

```

1 ▶ Run terraform apply -auto-approve tfplan
11 Acquiring state lock. This may take a few moments...
12 random_id.bucket_suffix: Creating...
13 random_id.bucket_suffix: Creation complete after 0s [id=bzudfq]
14 aws_s3_bucket.github_actions_bucket: Creating...
15 aws_s3_bucket.github_actions_bucket: Creation complete after 5s [id=github-actions-s3-dev-6f351d7d]
16 aws_s3_bucket_public_access_block.public_access_block: Creating...
17 aws_s3_bucket_server_side_encryption_configuration.encryption: Creating...
18 aws_s3_bucket_ownership_controls.ownership: Creating...
19 aws_s3_bucket_versioning.versioning: Creating...
20 aws_s3_bucket_logging.logging: Creating...
21 aws_s3_bucket_lifecycle_configuration.lifecycle: Creating...
22 aws_s3_bucket_public_access_block.public_access_block: Creation complete after 0s [id=github-actions-s3-dev-6f351d7d]
23 aws_s3_bucket_logging.logging: Creation complete after 1s [id=github-actions-s3-dev-6f351d7d]
24 aws_s3_bucket_server_side_encryption_configuration.encryption: Creation complete after 1s [id=github-actions-s3-dev-6f351d7d]
25 aws_s3_bucket_ownership_controls.ownership: Creation complete after 1s [id=github-actions-s3-dev-6f351d7d]
26 aws_s3_bucket_acl.bucket_acl: Creating...
27 aws_s3_bucket_acl.bucket_acl: Creation complete after 1s [id=github-actions-s3-dev-6f351d7d,private]
28 aws_s3_bucket_versioning.versioning: Creation complete after 3s [id=github-actions-s3-dev-6f351d7d]
29 aws_s3_bucket_lifecycle_configuration.lifecycle: Still creating... [30s elapsed]
30 aws_s3_bucket_lifecycle_configuration.lifecycle: Still creating... [20s elapsed]
31 aws_s3_bucket_lifecycle_configuration.lifecycle: Still creating... [30s elapsed]
32 aws_s3_bucket_lifecycle_configuration.lifecycle: Still creating... [40s elapsed]
33 aws_s3_bucket_lifecycle_configuration.lifecycle: Still creating... [50s elapsed]
34 aws_s3_bucket_lifecycle_configuration.lifecycle: Still creating... [100s elapsed]
35 aws_s3_bucket_lifecycle_configuration.lifecycle: Creation complete after 100s [id=github-actions-s3-dev-6f351d7d]
36
37
38 | Warning: Invalid Attribute Combination
39 |
40 |   with aws_s3_bucket_lifecycle_configuration.lifecycle,
41 |   on main.tf line 106, in resource "aws_s3_bucket_lifecycle_configuration" "lifecycle":
42 |   106: resource "aws_s3_bucket_lifecycle_configuration" "lifecycle" {
43 |
44 |       No attribute specified when one (and only one) of
45 |       [rule[0].filter,rule[0].prefix] is required
46 |
47 |   This will be an error in a future version of the provider
48 |
49 Apply complete! Resources: 9 added, 0 changed, 0 destroyed.
50
51 Outputs:
52
53 bucket_arn = "arn:aws:s3:::github-actions-s3-dev-6f351d7d"
54 bucket_domain_name = "github-actions-s3-dev-6f351d7d.s3.amazonaws.com"
55 bucket_hosted_zone_id = "Z36K2C52F8084"
56 bucket_name = "github-actions-s3-dev-6f351d7d"
57 bucket_region = "eu-west-2"
58 bucket_regional_domain_name = "github-actions-s3-dev-6f351d7d.s3.eu-west-2.amazonaws.com"

```

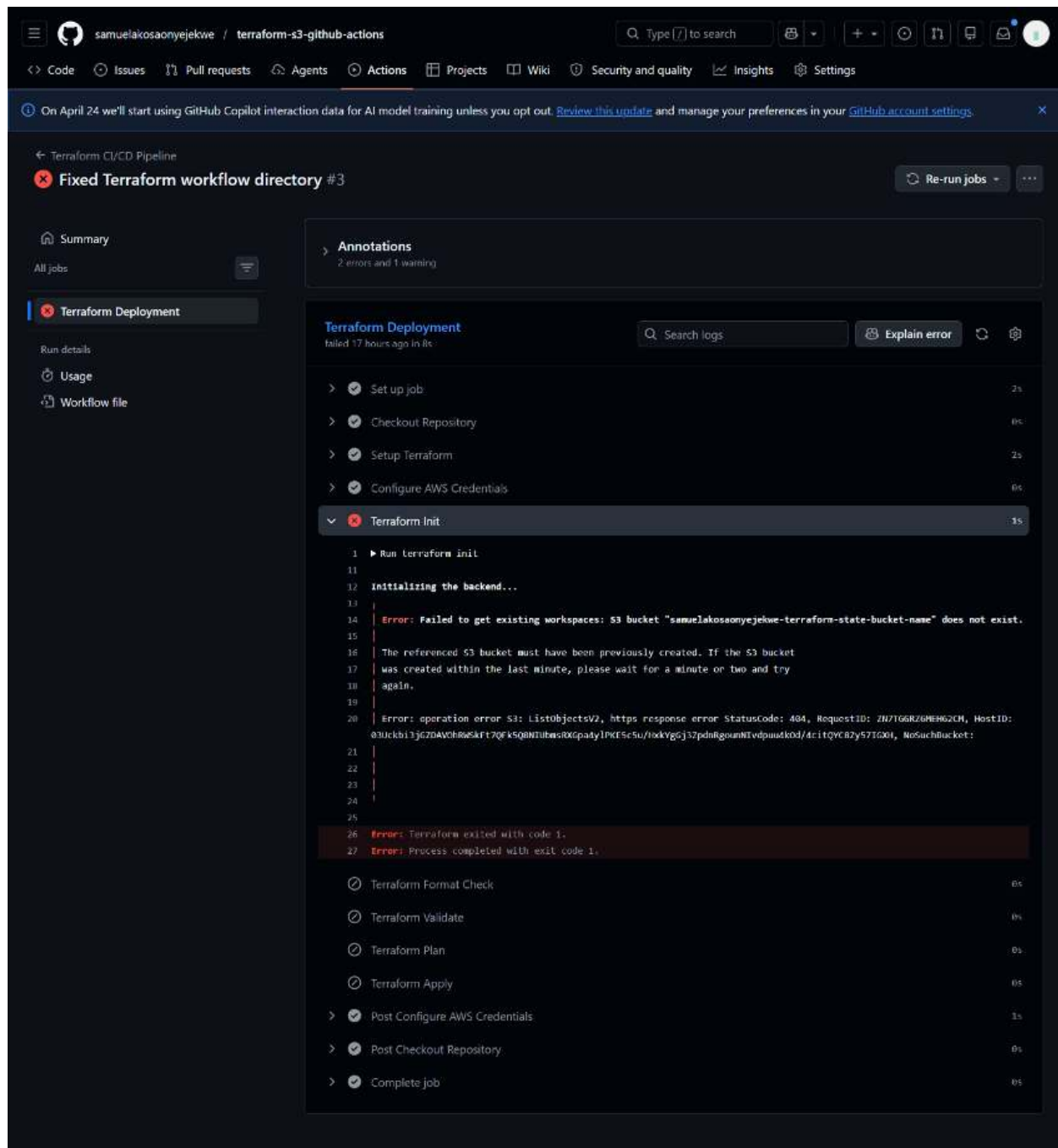
5.3: Fixing Error After Running Terraform workflow directory Workflow

This error occurred because backend S3 bucket with the name:

samuel-terraform-state-bucket

already exists or already taken by another user, but not in my AWS account.

See the screenshot that follows.



Step 1: Creating Backend Bucket

This error was fixed by creating backend bucket manually. I did this by replacing:

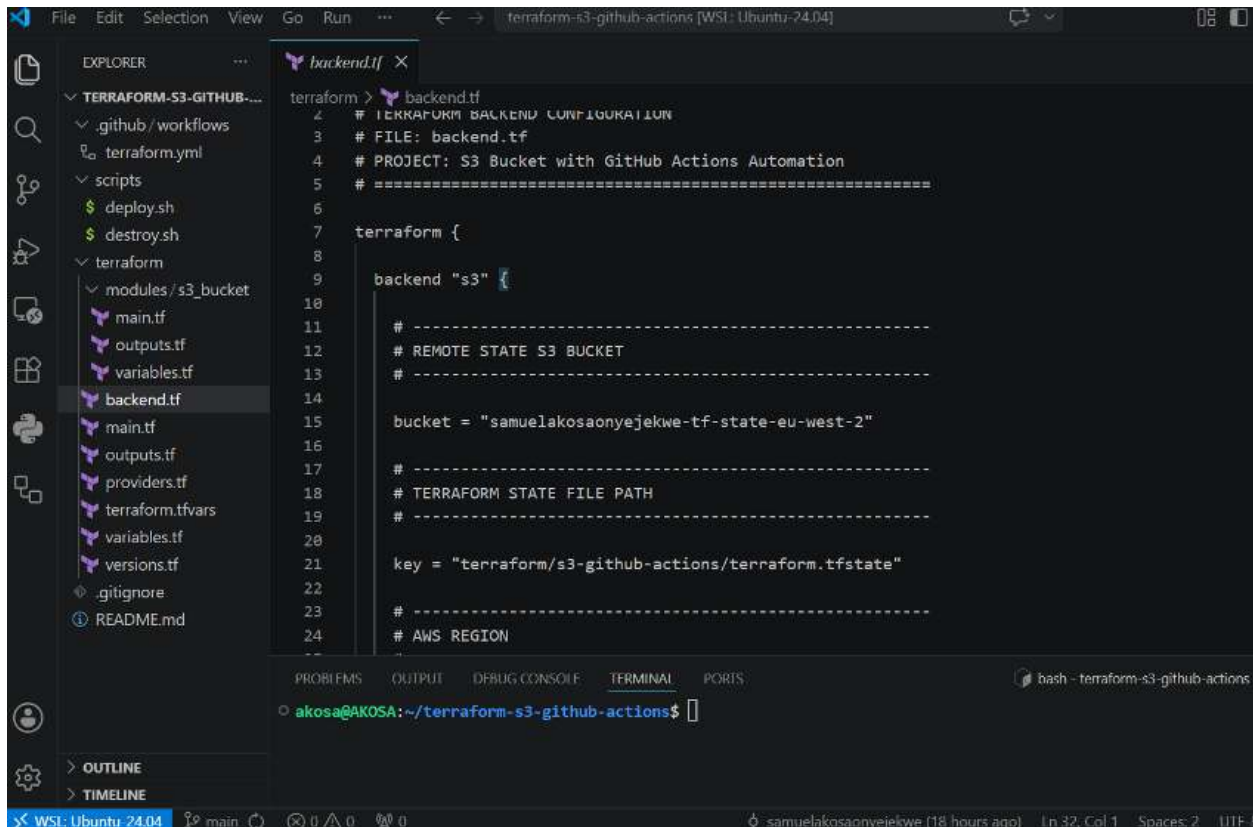
`samuel-terraform-state-bucket`

(which already existed)

by:

`samuelakosaonyekwe-tf-state-eu-west-2`

line 15 of the `backend.tf` file. See the screenshot that follows.



```
terraform > backend.tf
4  # TERRAFORM BACKEND CONFIGURATION
5  # FILE: backend.tf
6  # PROJECT: S3 Bucket with GitHub Actions Automation
7  # =====
8
9  terraform {
10
11      backend "s3" {
12
13          # -----
14          # REMOTE STATE S3 BUCKET
15          # -----
16
17          bucket = "samuelakosaonyekwe-tf-state-eu-west-2"
18
19          # -----
20          # TERRAFORM STATE FILE PATH
21          # -----
22
23          key = "terraform/s3-github-actions/terraform.tfstate"
24
25          # -----
26          # AWS REGION
27      }
28  }
```

This backend bucket is not created by Terraform automatically and is different from the application bucket.

It stores:

- `terraform.tfstate`
- Terraform locking/state data

Before Terraform can even start running, this backend bucket must already exist manually.

Step 2: Versioning of the Backend Bucket

After that, I enabled versioning of the backend bucket using the bash command:

```
akosa@AKOSA:~/terraform-s3-github-actions$ aws s3api put-bucket-versioning \
  --bucket samuelakosaonyekwe-tf-state-eu-west-2 \
  --versioning-configuration Status=Enabled
```

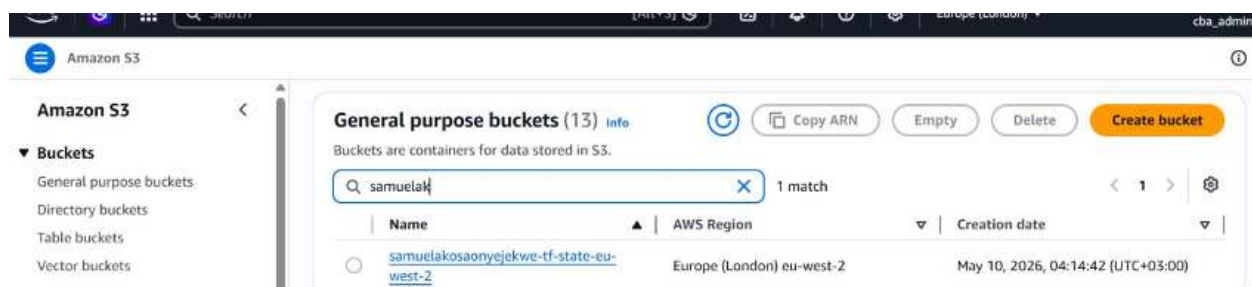
Step 3: Pushing fixed backend.tf file to GitHub

The following screenshot shows the run commands that push the fixed backend.tf file to GitHub.

```
akosa@AKOSA:~/terraform-s3-github-actions$ cd ~/terraform-s3-github-actions
git add .
git commit -m "Updated Terraform backend bucket"
git push
[main 863bba1] Updated Terraform backend bucket
 1 file changed, 1 insertion(+), 1 deletion(-)
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 403 bytes | 403.00 KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/samuelakosaonyekwe/terraform-s3-github-actions.git
 48a3a89..863bba1  main -> main
akosa@AKOSA:~/terraform-s3-github-actions$ |
```

Step 4: Verifying Backend Bucket in AWS Console

The following screenshot reveals the backend bucket provisioned by GitHub.



Task 6: Creating `deploy.sh` and `destroy.sh` files, and pushing these files to GitHub

The following bash commands were run to create the `deploy.sh` and `destroy.sh` files within the `scripts` folder.

```
akosa@AKOSA:~/terraform-s3-github-actions$ cd ~/terraform-s3-github-actions

mkdir -p scripts

touch scripts/deploy.sh
touch scripts/destroy.sh

chmod +x scripts/deploy.sh
chmod +x scripts/destroy.sh

ls -l scripts
total 0
-rwxr-xr-x 1 akosa akosa 0 May 10 11:42 deploy.sh
-rwxr-xr-x 1 akosa akosa 0 May 10 11:42 destroy.sh
akosa@AKOSA:~/terraform-s3-github-actions$ |
```

I ran the following Linux commands on Ubuntu terminal to push the `deploy.sh` and `destroy.sh` files to GitHub.

```
cd ~/terraform-s3-github-actions && \

chmod +x scripts/deploy.sh scripts/destroy.sh && \

git add . && \

git commit -m "Added deployment and destroy automation scripts" && \

git push origin main
```

```
akosa@AKOSA:~/terraform-s3-github-actions$ cd ~/terraform-s3-github-actions && \
chmod +x scripts/deploy.sh scripts/destroy.sh && \
git add . && \
git commit -m "Added deployment and destroy automation scripts" && \
git push origin main
[main 5673c35] Added deployment and destroy automation scripts
 2 files changed, 96 insertions(+)
  create mode 100755 scripts/deploy.sh
  create mode 100755 scripts/destroy.sh
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 4 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 1017 bytes | 339.00 KiB/s, done.
Total 5 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/samuelakosaonyekwe/terraform-s3-github-actions.git
 4577c1c..5673c35  main -> main
akosa@AKOSA:~/terraform-s3-github-actions$ |
```

The `deploy.sh` and `destroy.sh` files are created to automate and simplify Terraform infrastructure operations in a professional DevOps workflow.

They act as reusable operational scripts so we do not have to manually type long Terraform commands every time.

`deploy.sh`

Purpose:

- deploy infrastructure automatically

Typically handles:

```
terraform init
terraform fmt
terraform validate
terraform plan
terraform apply
```

So instead of typing the just above every time manually, we simply run:

```
./scripts/deploy.sh
```

`destroy.sh`

Purpose:

- destroy infrastructure safely and consistently

Typically runs:

```
terraform destroy -auto-approve
```

Instead of manually typing destroy commands repeatedly.

We simply run:

```
./scripts/destroy.sh
```

Adding deployment and destroy automation scripts

The screenshots below show the GitHub Actions page for the Terraform CI/CD Pipeline workflow. The workflow run titled:

```
Added deployment and destroy automation scripts
```

was successfully executed on the main branch.

The pipeline supports both:

- Terraform deployment automation
- Terraform destroy automation

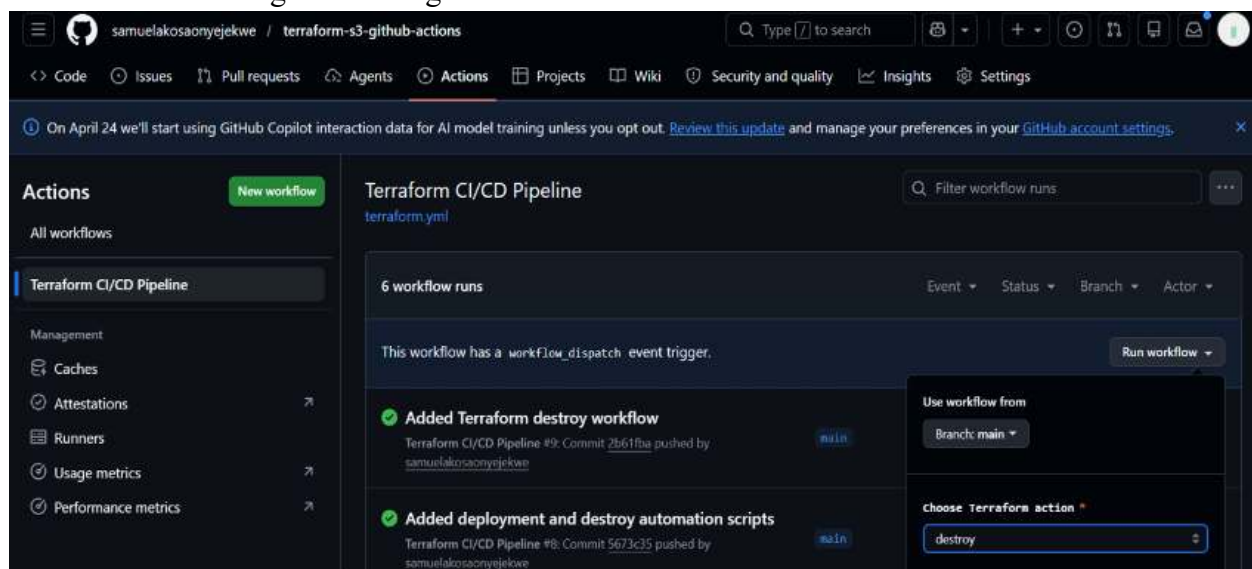
The workflow is configured with a manual trigger (`workflow_dispatch`) allowing users to select Terraform actions directly from GitHub Actions.

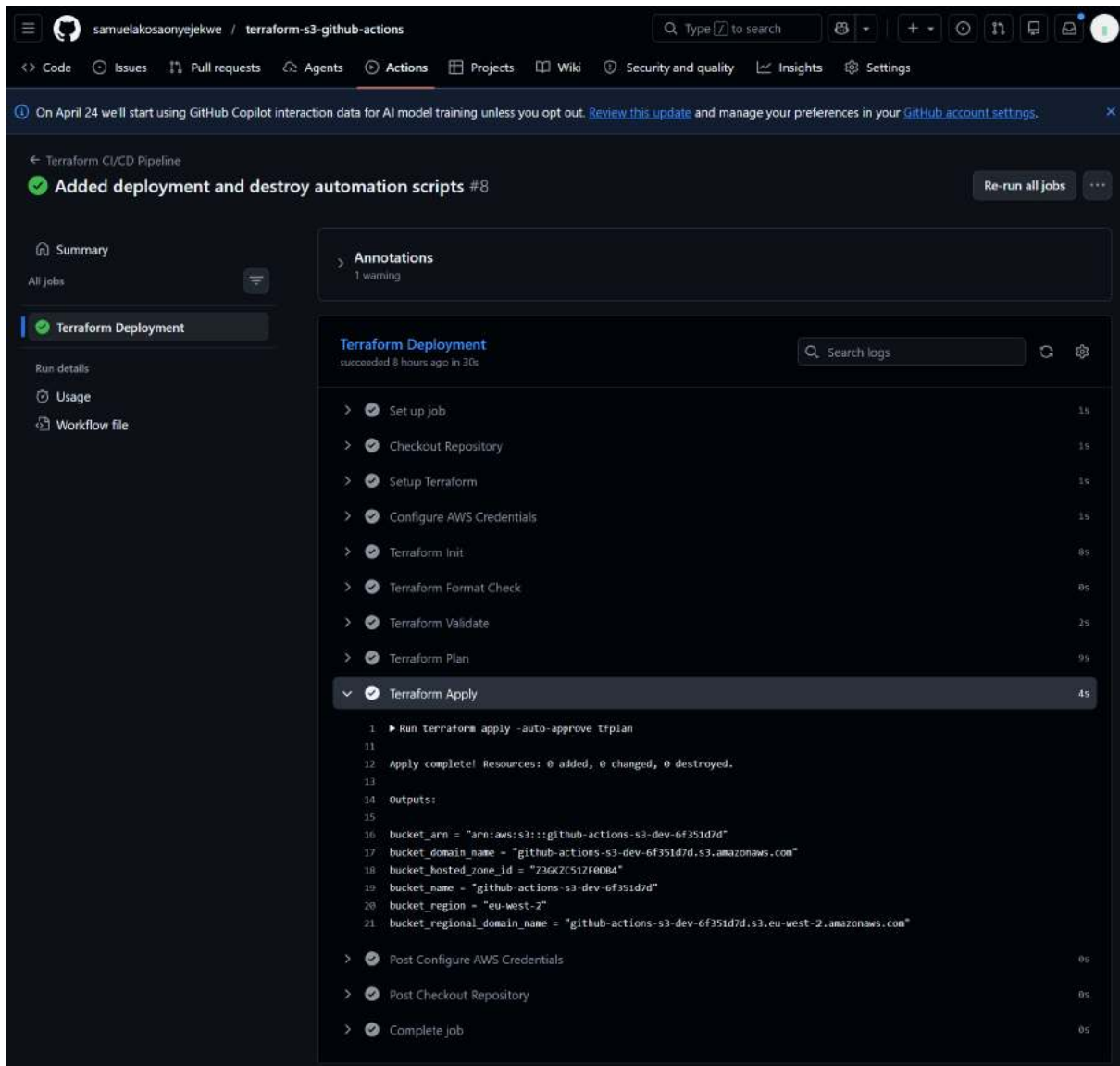
The dropdown menu shows the available Terraform action:

`destroy`

indicating that infrastructure destruction can now be executed automatically through the CI/CD pipeline.

- The successful workflow confirms that:
 - Deployment automation scripts were integrated successfully
 - Destroy automation scripts were integrated successfully
 - The Terraform CI/CD pipeline is fully operational for infrastructure lifecycle management using Terraform and GitHub Actions.





Adding Terraform destroy workflow

Real Production Usage

In enterprise environments:

`deploy.sh`

may also include:

- environment selection
- logging

- security checks
- policy validation
- Terraform workspace switching
- notifications

destroy.sh

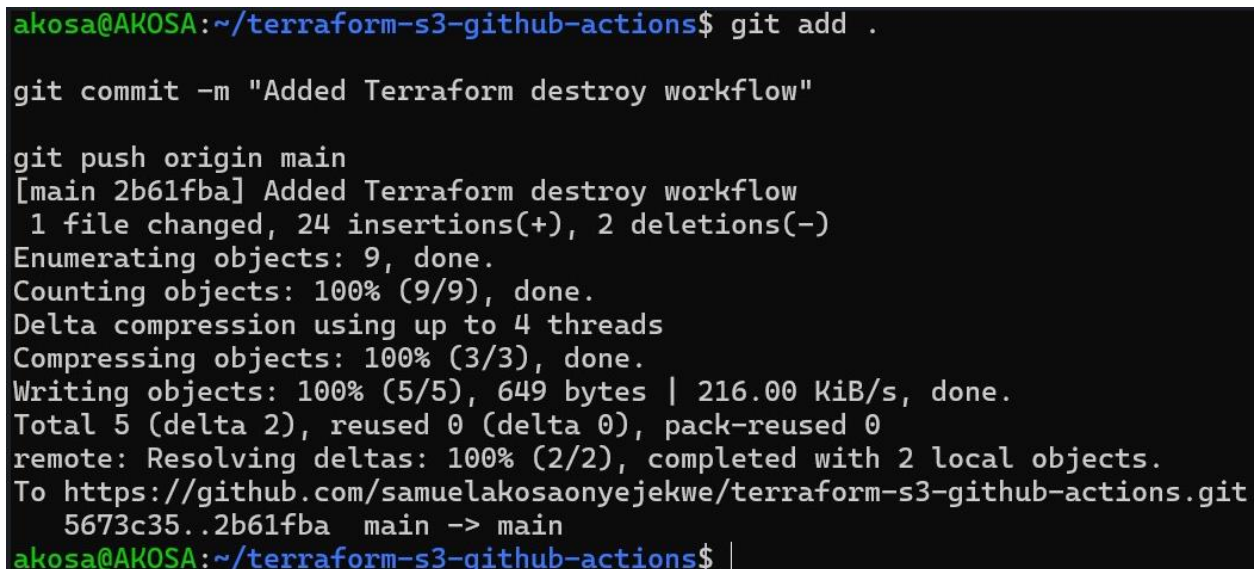
may include:

- confirmation prompts
- protection against production deletion
- approval checks
- backup creation before destroy

Added Terraform destroy workflow

I ran the following Linux commands to push the destroy.sh file to GitHub.

```
cd ~/terraform-s3-github-actions
git add .
git commit -m "Added Terraform destroy workflow"
git push origin main
```



```
akosa@AKOSA:~/terraform-s3-github-actions$ git add .
git commit -m "Added Terraform destroy workflow"
git push origin main
[main 2b61fba] Added Terraform destroy workflow
1 file changed, 24 insertions(+), 2 deletions(-)
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (5/5), 649 bytes | 216.00 KiB/s, done.
Total 5 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/samuelakosaonyejekwe/terraform-s3-github-actions.git
5673c35..2b61fba main -> main
akosa@AKOSA:~/terraform-s3-github-actions$
```

The screenshot below shows a successful GitHub Actions workflow execution titled:

Added Terraform destroy workflow

The Terraform CI/CD pipeline successfully executed all deployment stages including Terraform Apply and Terraform Destroy.

AWS authentication was securely configured using GitHub repository secrets.

The workflow confirmed successful Terraform infrastructure provisioning and destroy workflow integration.

The successful execution verifies that automated infrastructure destruction was successfully integrated into the production-grade Terraform IaC repository.

The screenshot displays the GitHub Actions interface for a workflow named 'Terraform CI/CD Pipeline'. The workflow is in a 'Completed' state, indicated by a green checkmark. The main section shows the 'Terraform Deployment' job, which succeeded 8 hours ago in 25s. The job details are expanded, showing a list of steps: 'Set up job', 'Checkout Repository', 'Setup Terraform', 'Configure AWS Credentials', 'Terraform Init', 'Terraform Format Check', 'Terraform Validate', 'Terraform Plan', 'Terraform Apply', 'Terraform Destroy', 'Post Configure AWS Credentials', 'Post Checkout Repository', and 'Complete job'. The 'Terraform Apply' step is highlighted, showing its output: 'Run Terraform apply -auto-approve tfplan'. The output text indicates that the apply was successful, with no resources added, changed, or destroyed. The 'Terraform Destroy' step is also shown as completed.

On April 24 we'll start using GitHub Copilot interaction data for AI model training unless you opt out. [Review this update](#) and manage your preferences in your [GitHub account settings](#).

← Terraform CI/CD Pipeline

✓ Added Terraform destroy workflow #9 Re-run all jobs ...

Summary

All jobs

✓ Terraform Deployment

Run details

Usage

Workflow file

Annotations

1 warning

Terraform Deployment

succeeded 8 hours ago in 25s Search logs ...

- Set up job 1s
- Checkout Repository 0s
- Setup Terraform 1s
- Configure AWS Credentials 1s
- Terraform Init 7s
- Terraform Format Check 0s
- Terraform Validate 2s
- Terraform Plan 7s
- ✓ Terraform Apply 3s
- Terraform Destroy 0s
- Post Configure AWS Credentials 0s
- Post Checkout Repository 1s
- Complete job 0s

```
1 ▶ Run Terraform apply -auto-approve tfplan
11
12 Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
13
14 Outputs:
15
16 bucket_arn = "arn:aws:s3:::github-actions-s3-dev-6f351d7d"
17 bucket_domain_name = "github-actions-s3-dev-6f351d7d.s3.amazonaws.com"
18 bucket_hosted_zone_id = "Z3GK2C5I2F00B4"
19 bucket_name = "github-actions-s3-dev-6f351d7d"
20 bucket_region = "eu-west-2"
21 bucket_regional_domain_name = "github-actions-s3-dev-6f351d7d.s3.eu-west-2.amazonaws.com"
```

Task 7: Provisioning and configuring a Jenkins Server

7.1: Create a Custom VPC and Subnets

Region: Europe, London (eu-west-2)

No of Availability Zones: 2 (eu-west-2a/b)

Subnets: 2 Public and 2 Private

Create VPC workflow

Success

Details

- ✓ Create VPC: [vpc-0f8d639b886e3f371](#)
- ✓ Enable DNS hostnames
- ✓ Enable DNS resolution
- ✓ Verifying VPC creation: [vpc-0f8d639b886e3f371](#)
- ✓ Create S3 endpoint: [vpce-04c760e14a6606124](#)
- ✓ Create subnet: [subnet-017f367933e3ab81d](#)
- ✓ Create subnet: [subnet-0038cd46b3241c98c](#)
- ✓ Create subnet: [subnet-01ca41cc26a0b3a35](#)
- ✓ Create subnet: [subnet-01ce9c4be5ba50741](#)
- ✓ Create internet gateway: [igw-0093a09babba3b099](#)
- ✓ Attach internet gateway to the VPC
- ✓ Create route table: [rtb-01975cc7e9e644679](#)
- ✓ Create route
- ✓ Associate route table
- ✓ Associate route table
- ✓ Create route table: [rtb-0ce43ba52dafb5521](#)
- ✓ Associate route table
- ✓ Create route table: [rtb-02f85c101a912e16a](#)
- ✓ Associate route table
- ✓ Verifying route table creation
- ✓ Associate S3 endpoint with private subnet route tables: [vpce-04c760e14a6606124](#)

Related VPC resources and services

Discover additional resources you can launch within your VPC to build your network architecture. From security components to connectivity services, explore the building blocks available for your infrastructure.



Client VPN

Networking

AWS Client VPN is a managed client-based VPN service that enables you to securely access AWS resources and resources in your on-premises network.

Create

Resources

Let us know what you would like to create in your VPC

e.g., Instances, Network ACL, Transit

100 characters available. Do not disclose any personal, commercially sensitive, or confidential information.

Submit feedback

View VPC

VPC > Your VPCs > vpc-0f8d639b886e3f371

vpc-0f8d639b886e3f371 / jenkins_sam-vpc

Details [Info](#)

VPC ID vpc-0f8d639b886e3f371	State Available	Block Public Access Off	DNS hostnames Enabled
DNS resolution Enabled	Tenancy default	DHCP option set dopt-01c8cc6efcc8126b4	Main route table rtb-00d4fb8d439a59ae9
Main network ACL acl-06644f245501fe8b1	Default VPC No	IPv4 CIDR 10.0.0.0/16	IPv6 pool -
IPv6 CIDR (Network border group) -	Network Address Usage metrics Disabled	Route 53 Resolver DNS Firewall rule groups -	Owner ID 009850210027
Encryption control ID -	Encryption control mode -		

VPC > Your VPCs > vpc-0f8d639b886e3f371

VPC

Your AWS virtual network

jenkins_sam-vpc

Subnets (4)

Subnets within this VPC

- eu-west-2a**
 - jenkins_sam-subnet-public1-eu-west-2a
 - jenkins_sam-subnet-private1-eu-west-2a
- eu-west-2b**
 - jenkins_sam-subnet-public2-eu-west-2b
 - jenkins_sam-subnet-private2-eu-west-2b

Route tables (4)

Route network traffic to:

- rtb-00d4fb8d439a59ae9
- jenkins_sam-rtb-public
- jenkins_sam-rtb-private2
- jenkins_sam-rtb-private1

VPC > Your VPCs

VPC dashboard

AWS Global View

Filter by VPC

Virtual private cloud

Your VPCs

Subnets

Route tables

Your VPCs

VPCs VPC encryption controls

Your VPCs (2) [Info](#)

Find VPCs by attribute or tag

Name	VPC ID	State	Encryption c...	Encryption contr
jenkins_sam-vpc	vpc-0f8d639b886e3f371	Available	-	-

VPC > Security Groups

Security Groups (2) [Info](#)

Find security groups by attribute or tag

Security group ID	Security group name	VPC ID
sg-0b176607af4a4a238	default	vpc-0f8d639b886e3f371

7.2: Launch an instance within the custom public subnet

I provisioned an EC2 instance with Jenkins pre-installed, using the following configuration details to set up the server.

Launch an instance within the custom public subnet

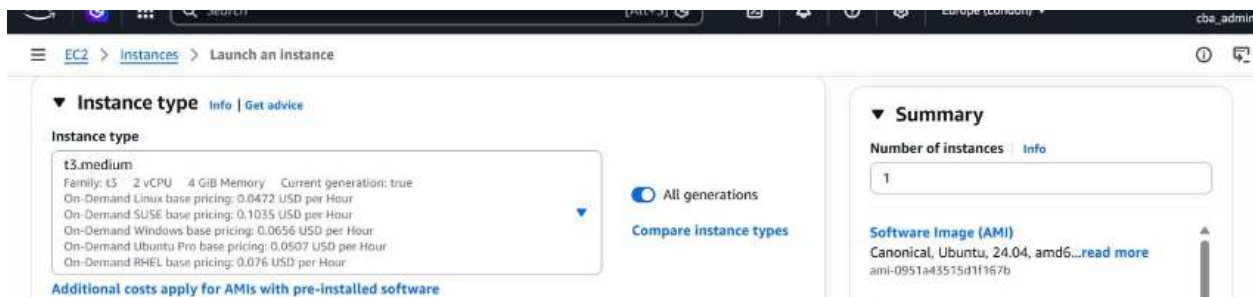
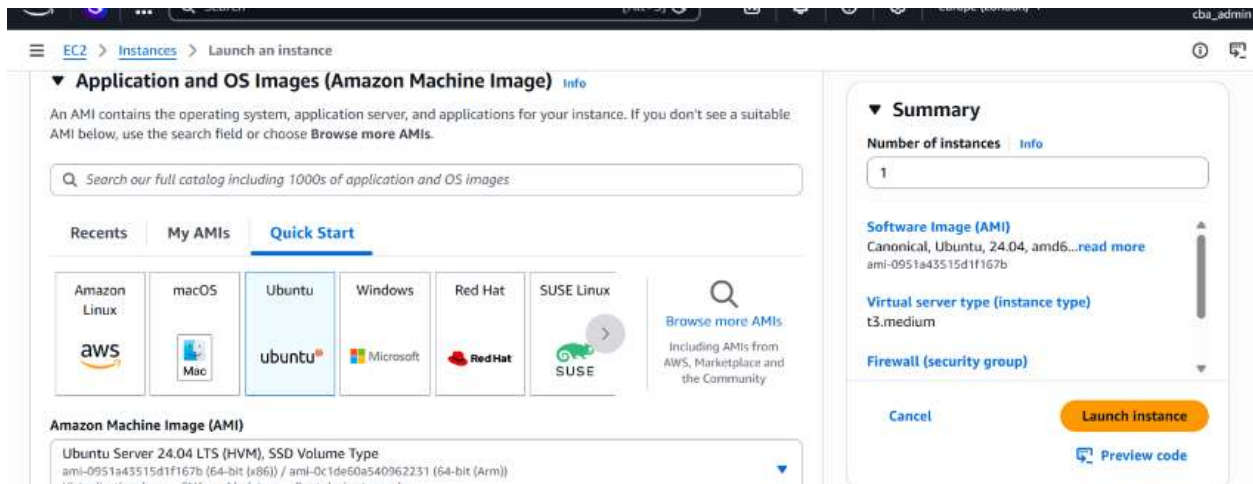
Instance type: t3.medium

AMI: Ubuntu 24.04

Security Group rule: Open port 22, 80 and 8080 (i.e. Custom TCP Protocol)

User data: can be found at:

<https://drive.google.com/file/d/1Hb3fVTGIHiBT90cJiH4bHrm6WVfWBhtM/view?usp=sharing>



▼ Security group rule 3 (TCP, 8080, 0.0.0.0/0)

Remove

Type | Info

Custom TCP ▼

Protocol | Info

TCP

Port range | Info

8080

Source type | Info

Anywhere ▼

Source | Info

Q Add CIDR, prefix list or security

0.0.0.0/0 X

Description - optional | Info

e.g. SSH for admin desktop

User data - optional | Info

Upload a file with your user data or enter it in the field.

↑ Choose file

```
#!/bin/bash
```

```
#
```

```
=====
```

```
==
```

```
# UPDATE SYSTEM
```

```
#
```

```
=====
```

```
==
```

```
apt update -y
```

```
apt upgrade -y
```

```
#
```

```
=====
```

- SSH into ready EC2 instance via terminal to get the Jenkins password

Instances (2) Info

Last updated
less than a minute ago



Connect

Instance state ▼

Actions ▼

Launch instances



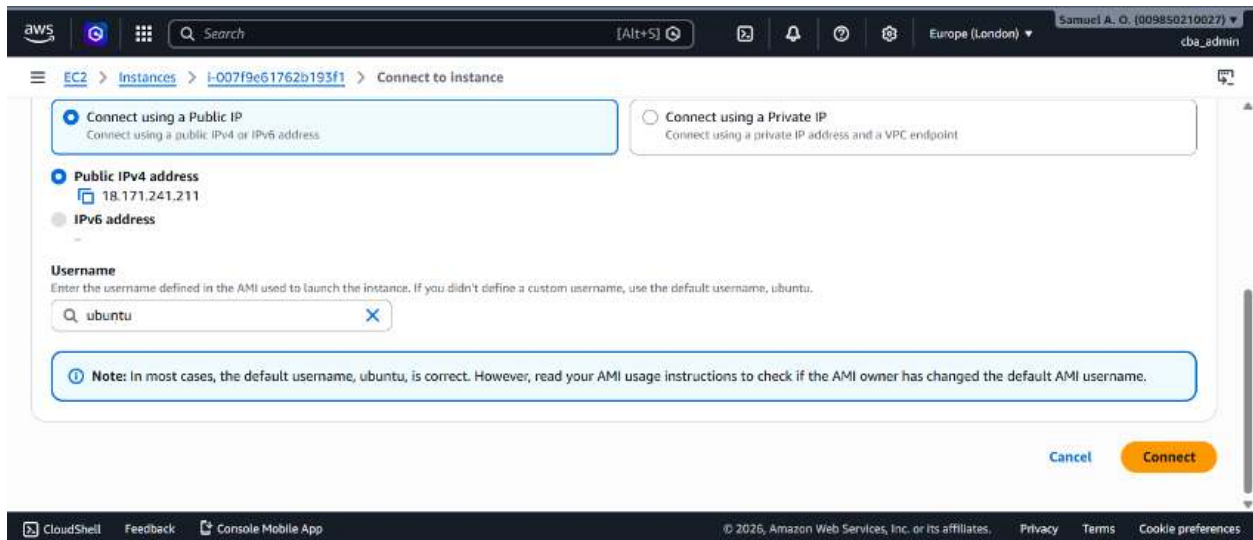
Saved filter sets

Choose filter set ▼

Q Find Instance by attribute or tag (case-sensitive)

< 1 > ⚙

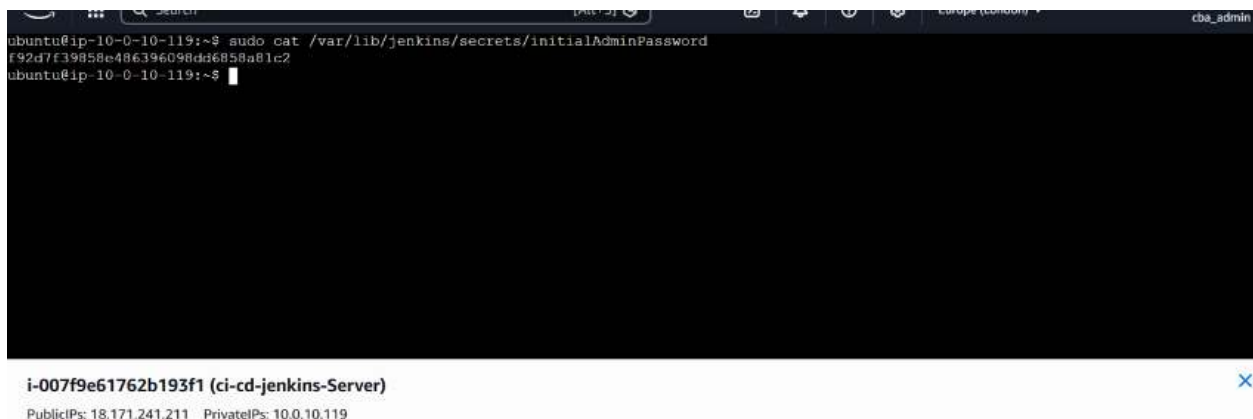
☐	Name	▲	Instance ID	Instance state ▼	Instance type ▼	Status check	Alarm status	Availability
☐	ci-cd-jenkins-Server		i-007f9e61762b193f1	Running	t3.medium	3/3 checks p:	View alarms +	eu-west-2a



Task 8: Jenkins Initial Configuration and Administrator Setup

- Printed out the Jenkins password using the command below:

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```



Server-generated password was:

f92d7f[REDACTED]81c2

- Logged in to Jenkins server via the web using my public IP address and Jenkins port number: 18.171.241.211:8080

i-007f9e61762b193f1 (ci-cd-jenkins-Server)

Auto-assigned IP address

 18.171.241.211 [Public IP]

VPC ID

 vpc-0f8d639b886e3f371 (jenkins_sam-vpc) 

- I inputted the password (f92d7f39858e486396098dd6858a81c2) and installed suggested plugin



Getting Started

Customize Jenkins

Plugins extend Jenkins with additional features to support many different needs.

Install suggested plugins

Install plugins the Jenkins community finds most useful.

Select plugins to install

Select and install plugins most suitable for your needs.

Jenkins 2.555.1

Getting Started

Getting Started

✓ Folders	Formatter			STORYBOOK, ETC... API
✓ Timestamper	✓ Workspace Cleanup	✓ Ant	Gradle	** Jackson 3 API
⌚ Pipeline	⌚ GitHub Branch Source	⌚ Pipeline: GitHub Groovy Libraries	⌚ Pipeline Graph View	** JQuery3 API
⌚ Git	⌚ SSH Build Agents	⌚ Matrix Authorization Strategy	⌚ LDAP	** Echarts API
⌚ Email Extension	⌚ Mailer	⌚ Dark Theme		** Prism API

Workspace Cleanup

Ant

okhttp

JavaBeans Activation Framework (JAF) API

- required dependency

Jenkins 2.555.1

- I configured the login details for an admin user using my preferred details.

Getting Started

Create First Admin User

Username

phoenixsamuel

Password

Confirm password

Full name

Akosa Samuel Onyejekwe

E-mail address

samuelonyekwe852@gmail.com

Jenkins 2.555.1

Skip and continue as admin

Save and Continue

Instance Configuration

Jenkins URL:

http://18.171.241.211:8080/

The Jenkins URL is used to provide the root URL for absolute links to various Jenkins resources. That means this value is required for proper operation of many Jenkins features including email notifications, PR status updates, and the BUILD_URL environment variable provided to build steps.

The proposed default value shown is **not saved yet** and is generated from the current request, if possible. The best practice is to set this value to the URL that users are expected to use. This will avoid confusion when sharing or viewing links.

Jenkins 2.555.1

Not now

Save and Finish

Getting Started

Jenkins is ready!

Your Jenkins setup is complete.

Start using Jenkins

Jenkins 2.555.1

Task 9: GitHub Integration

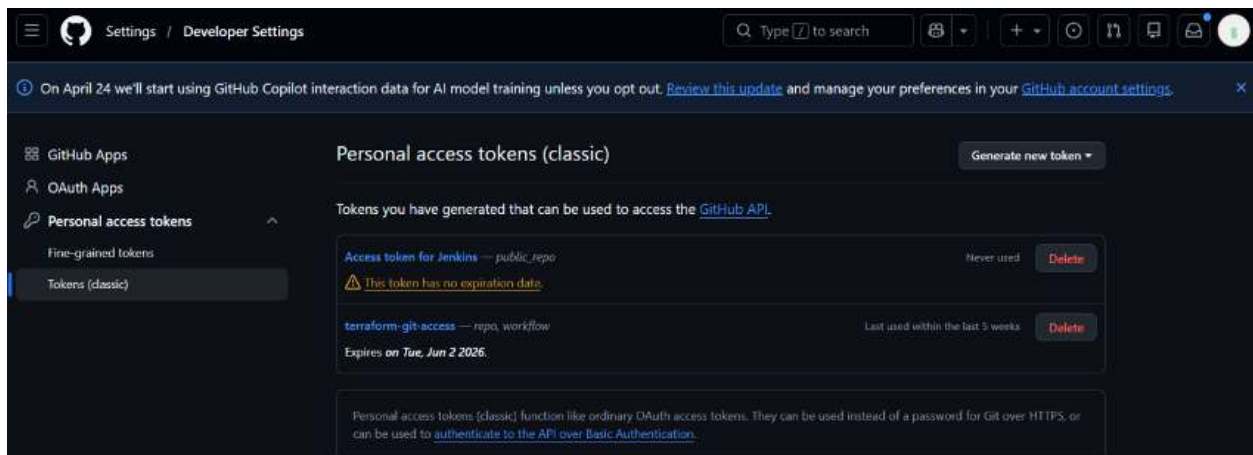
I integrated the Jenkins server with my GitHub account.

9.1: Generating Token from GitHub

I created a Personal Access Token (PAT) because GitHub and Jenkins require secure authentication and authorisation for Jenkins to access repositories, trigger pipelines, and integrate with my GitHub account.

Step 1:

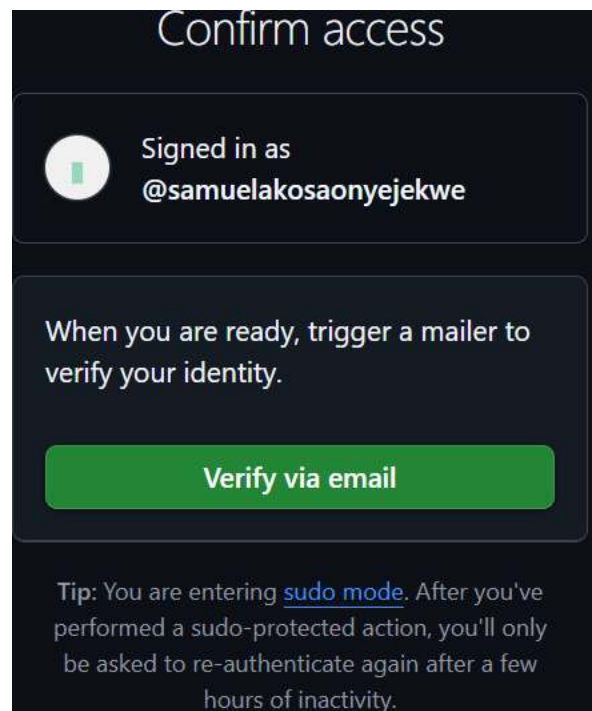
- I signed in to my GitHub account and navigated to the **Settings** option from the profile dropdown menu. From the left navigation panel, I scrolled down and selected **Developer settings**. I then accessed **Personal access tokens** and chose the **Tokens (classic)** option.

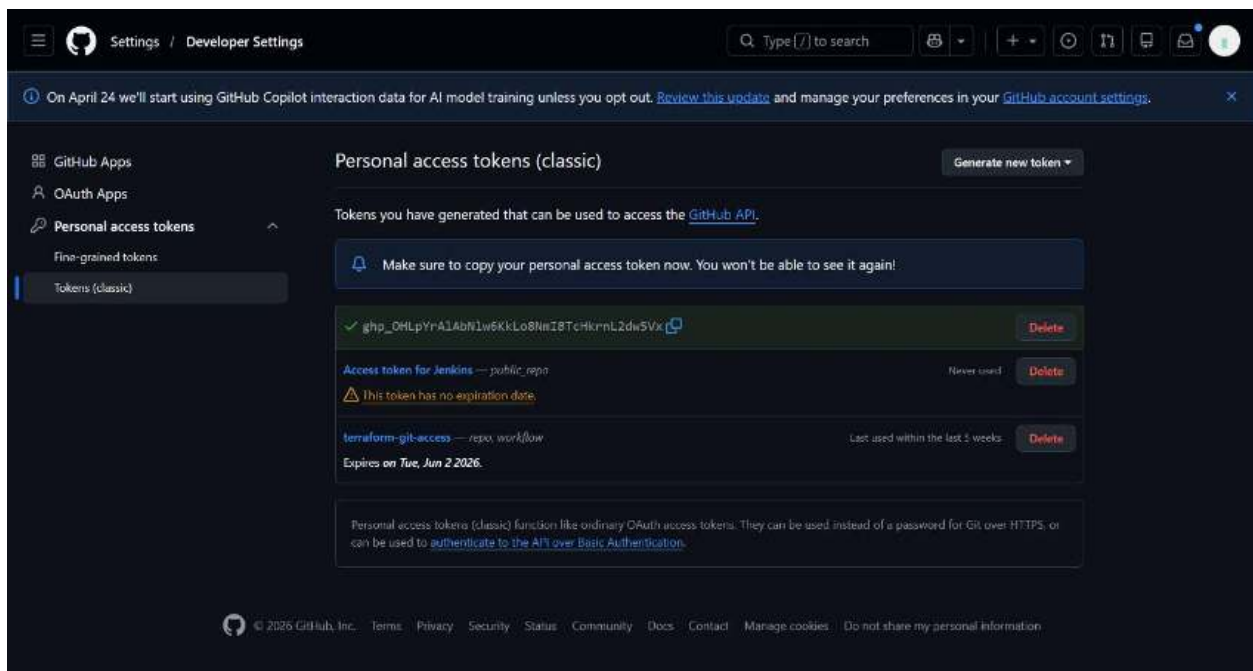
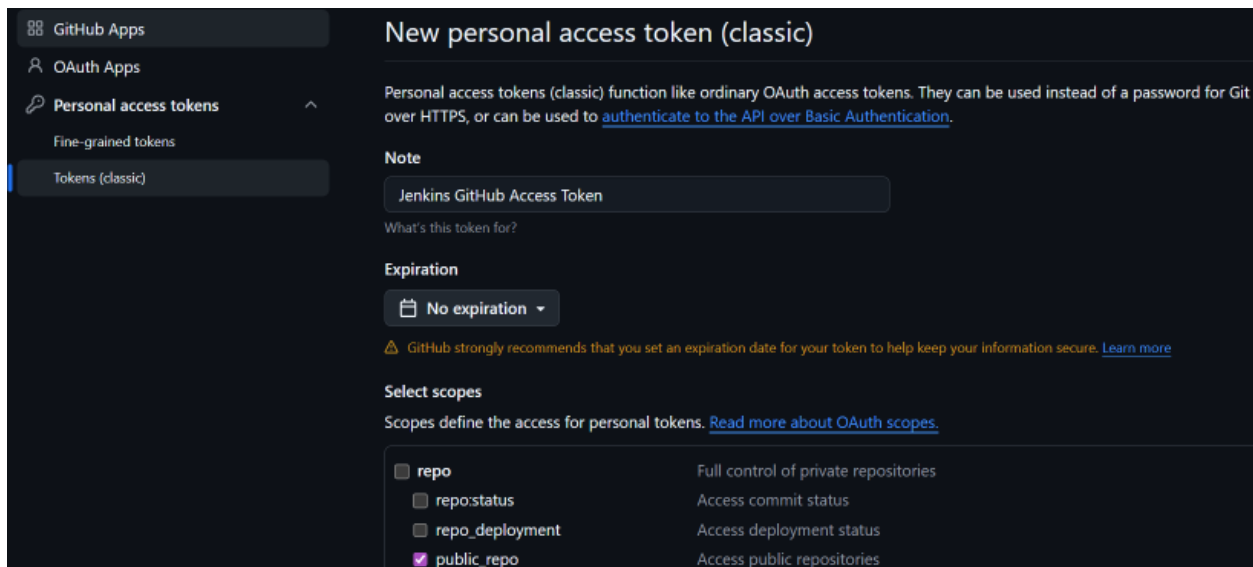


Step 2:

- I clicked on **Generate new token** and selected the **Generate new token (classic)** option. The token value generated was:

[REDACTED]





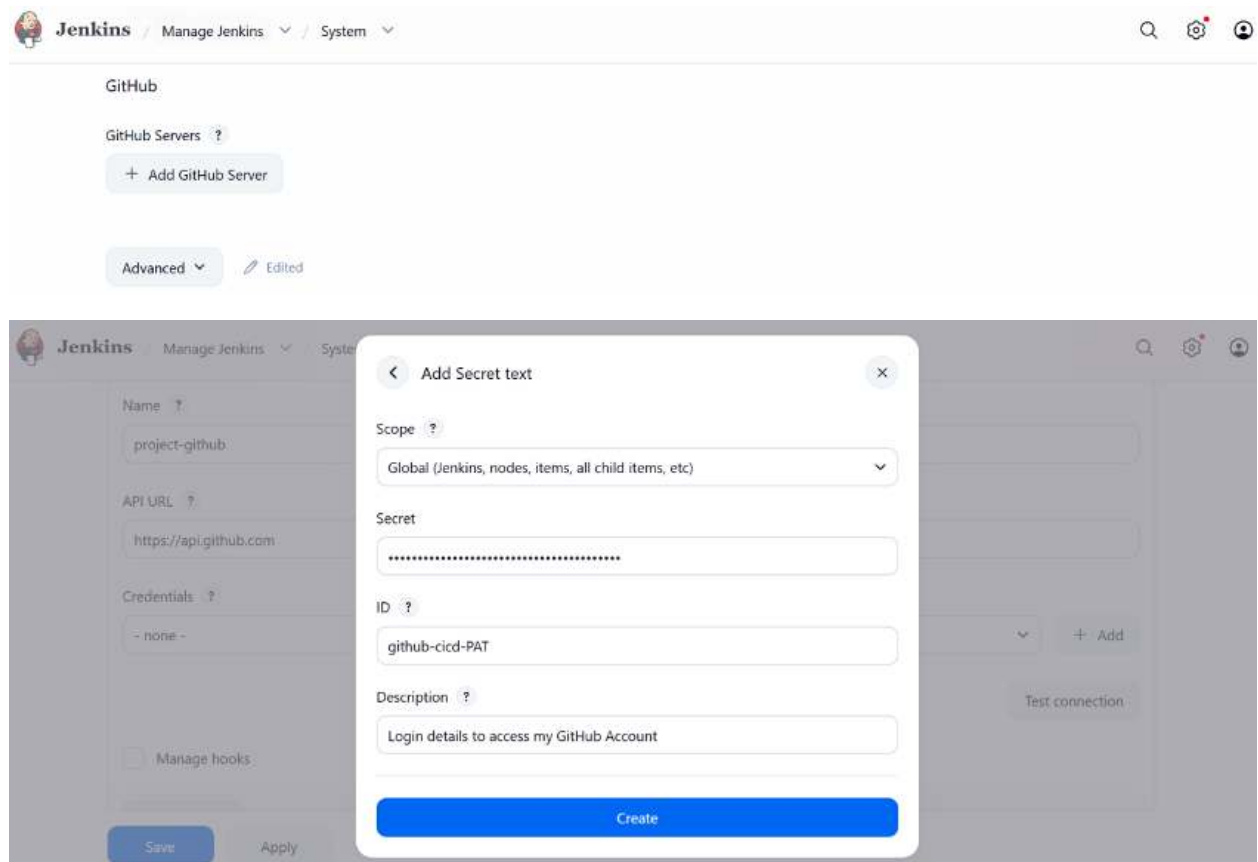
9.2: Creating a GitHub Credential in Jenkins

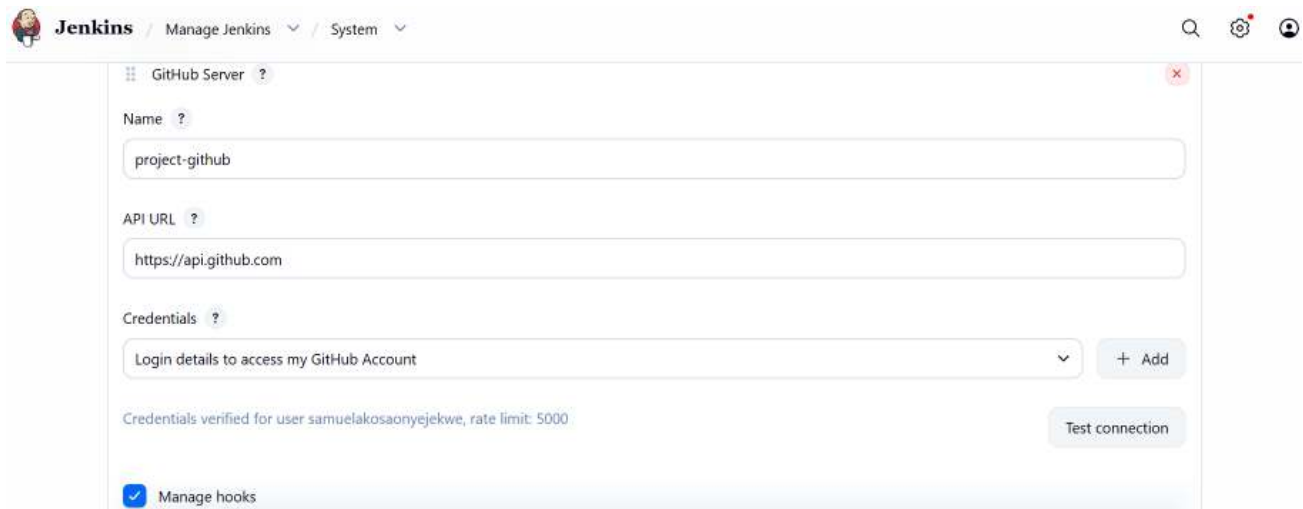
Action 1: Storing the token securely in Jenkins.

The following are steps taken to store the token securely in Jenkins.

- Returned to the Jenkins dashboard.
- Selected the settings (gear) icon located at the top-right section of the Jenkins dashboard.
- Under **System Configuration**, clicked on **System**.
- Scrolled down to the **GitHub** integration section.

- Clicked the **Add GitHub Server** drop-down menu and selected **GitHub Server**.
- Under **Credentials**, selected **Add** and chose **Global credentials**.
- On the **Add Credentials** page, selected **Secret text** under the **Kind** field and proceeded to the next step.
- Pasted the copied Personal Access Token (PAT) into the **Secret** field.
- Updated the credentials field by selecting the newly created GitHub credential.
- Clicked **Test Connection** to verify successful integration with GitHub.
- Ensured that the **Manage hooks** option was enabled.





9.3: Generating a Public Key from Jenkins Instance and adding it to GitHub

For secure code checkout, Jenkins needs SSH access to GitHub. For secure code checkout, Jenkins needs SSH access to GitHub.

- SSH into my Jenkins instance via terminal
- I ran the command:

```
ssh-keygen -t rsa
```

to generate SSH key

```
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ubuntu/.ssh/id_rsa
Your public key has been saved in /home/ubuntu/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:+CXKlNlKzfSkK2I3Fd6l+3Vxa49N66QhKNkuUomoZK4 ubuntu@ip-10-0-10-119
The key's randomart image is:
+---[RSA 3072]-----+
|
|
|  ...  .
| +.=o=. .
| . =oX.S.. ..
| +.O +ooo +
| ..=o+ +o.+ . o =.
| ...+. .o . o *+o|
| E.      . . .ooo|
+----[SHA256]-----+
ubuntu@ip-10-0-10-119:~$
```

i-007f9e61762b193f1 (ci-cd-jenkins-Server)

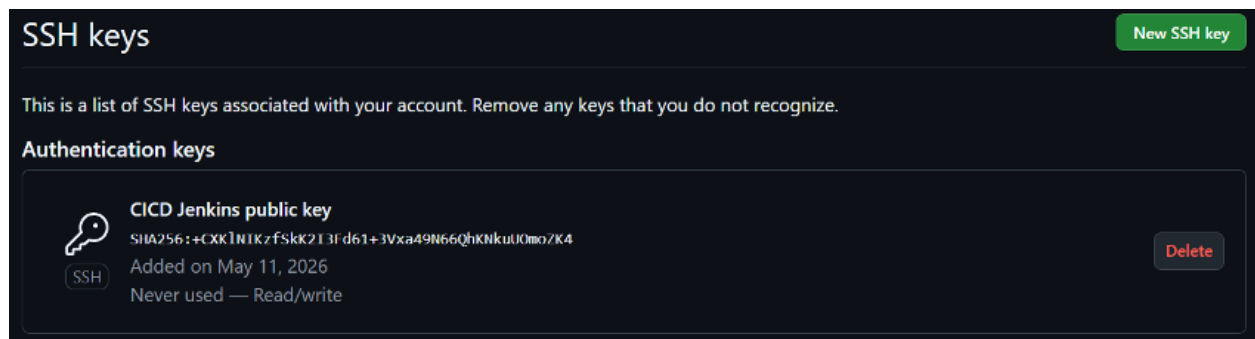
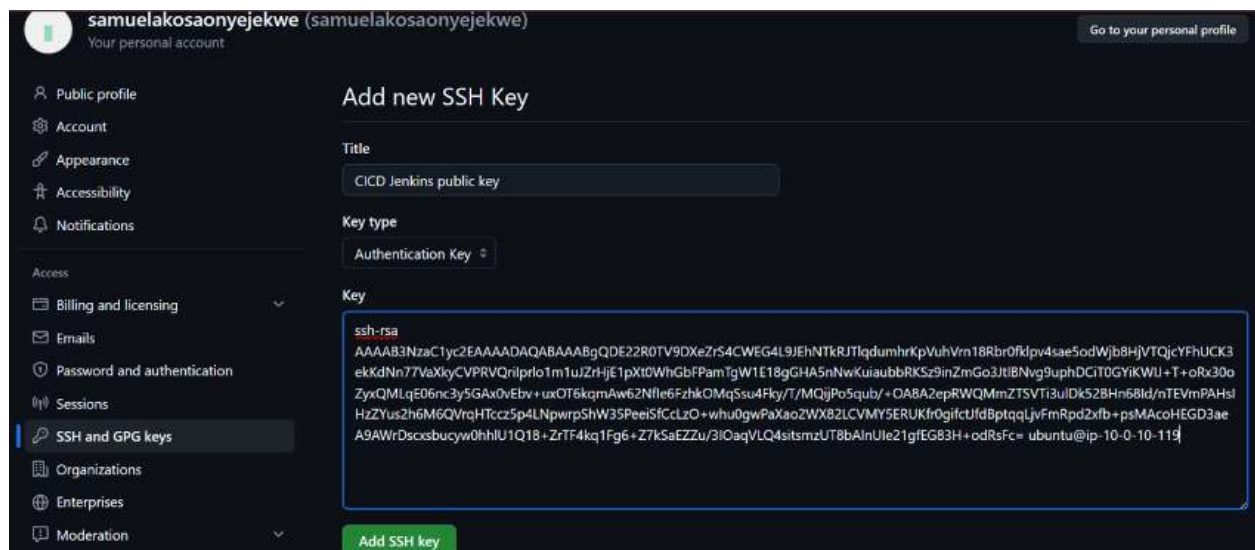
PublicIPs: 18.171.241.211 PrivateIPs: 10.0.10.119

- To print out the generated public key, I ran the command:

```
cat ~/.ssh/id_rsa.pub
```

```
ubuntu@ip-10-0-10-119:~$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQgQDE22R0TV9DXeZrS4CWEG4L9JEhNTkRJTlqduhrKpVuhVrn18Rbr0fk1pv4sae5odWjb8HjVTQjcYfHUCK3ekKdNn77VaXkyCVPRVQ
riIprloIm1uJzrHjE1pXt0WhGbFPamTgW1E18gGHA5nNwKuiabBRKSz9inZmGo3Jt1BNvg9uphDCiT0GYiKWIJ+T+oRx30oZyxQMLqE06nc3y5GAx0vEbv+uxOT6kqmA62Nf1e6Fzh
kOMgSsu4Fky/T/MQijPo5qub/+OA8A2epRWQMmZTSVTi3u1Dk52BIn681d/nTEVmPAHsIHzzYus2h6M6QVrqHTccz5p4LNpwrpShW35PeeiSfCclzO+whu0gwPaXao2WX82LCVMY5ERU
Kfr0gIfctJfdBptqLjvFmRpd2xTb+psMAcoHEGD3aeA9AWrDscxsbcyW0hh1U1Q18+ZrTF4kq1Fg6+Z7kSaEZZu/3IOaqVLQ4sitsmzUT8bAlnUie21gFEg83H+odRsFc= ubuntu@
ip-10-0-10-119
ubuntu@ip-10-0-10-119:~$
```

- Navigated to my GitHub profile.
- Clicked on **Settings** within GitHub.
- Navigated down the left-hand panel to the **Access** section and selected **SSH and GPG keys**.
- Clicked on **New SSH Key** to create a new key entry.
- Pasted the copied public SSH key into the **Key** field.
- Selected **Add SSH Key** to save and register the SSH key with the GitHub account.



- Back to my Jenkins Instance, I ran the command:

```
cat ~/.ssh/id_rsa
```

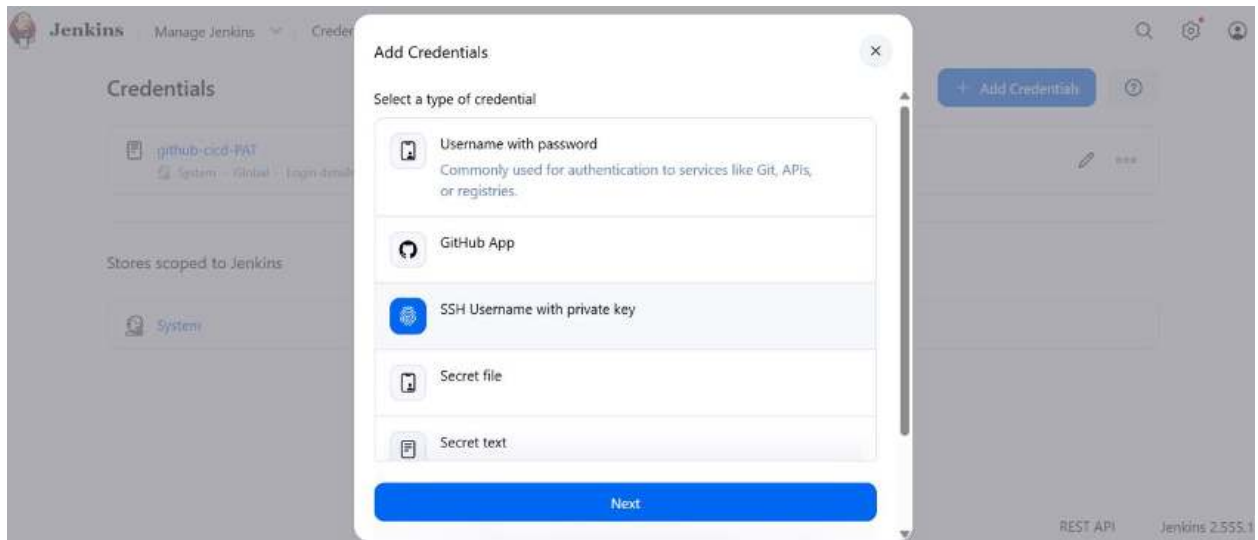
to print the private key

```
ubuntu@ip-10-0-10-119:~$ cat ~/.ssh/id_rsa
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktbjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAABlWAAAAdzc2gtcn
NhAAAAAwEAAQAAAEYEAxNtkdElfQ13ma0uAlhBuC/SRITU5ESU5anbpoayqVboVa59fEW69
H5Jab+LGnuaHVo2/B41U0I3GBYVAit3pCnTZ++1Wl5MglT0VUK4iKa5aNZtbiWax4xNaV7
dFoRmxT2pk4FtRNfIBhwOZzcCromrm20Sks/Yp2ZhqNybzQTb4PbqYQwok9BmIiliCfk/q
Ecd9KGcsUDC6hNOp3N8uRgMdLxG7/rsTk+pKpgMOTjXyHuhc4ZDjKkrLuBZMv0/zEIoz6O
arm//jgPANnqUVkDjM0U1U4t7pQ5Odgr5+vCHf50xFZjwB7CB82WLRNoejOkFa6h03HM+a
eCzacK6UoVt+T3noknwnC8zvsIbtIMD2l2qNl1/NiwlTGOREVCn69IIn3LSX3Qabaqi47x
ZkaXdsX2/qbDAHKBxBg92ngPQFqw7HMBG7nMsNIYZVNUNfPma0xeJKtRYOvme5EmhGWbv9
yDmq1S0OLIrBjs1E/GwJZ1CHttYHxBvNx/qHUBBXAAAFkItcul+LXLpfAAAAB3NzaC1yc2
EAAAGBAMTBZHRNXX0Nd5mtLgJYQbgv0kSElORElOWp26aGsqlW6FWufXxFuvR+SWm/ixp7m
h1aNvweNVNcNcgWfQIrd6Qp02fvtVpeTIJU9FVCuIimuWjWbW4lmseMTWle3RaEZsU9qZO
BbUTXyAYcDmc3Aq6Jq5ttEpLP2KdmYajcm2UE2+D26mEMKJPQZiIpYgn5P6hHHfShnLFAw
uoTTqdzfLkYDHS8Ru/67E5PqSqYDDrY18h7oXOGQ4ypKy7gWTL9P8xCKM+jmq5v/44DwDZ
6lFZAYz1NJVOLe6UOTnYEefrwh3+dMRWY8AewgfNli6zaHozpBWuodNxxPmngs2nCulKFb
fk956JJ8JwvM77CG7SDA9pdqjZZfzYsJUxjkRFQp+vSCJ9y0l90Gm2qouO8WZG13bF9v6m
wwBygcQYPdp4D0BasOxzGxu5zLDSGGVTVDXz5mtMXiSrUWDr5nuRJoRlm7/cg5qpUtDiyK
2ybNRPxsCwDqh7bWB8Qbzcf6h1GwVwAAAAMBAAEAAAGASX1l6Q0hbTLW3GNZY6xKiOgKbg
```

i-007f9e61762b193f1 (ci-cd-jenkins-Server)

PublicIPs: 18.171.241.211 PrivateIPs: 10.0.10.119

- Returned to the Jenkins dashboard.
- Clicked the settings (gear) icon located at the upper-right corner of the Jenkins dashboard.
- Under the **Security** section, selected **Credentials**.
- Clicked on **Add Credentials** to create a new credential entry.
- Under the **Kind** field, selected **SSH Username with Private Key**.
- Pasted the copied private SSH key into the appropriate field.
- Navigated to the top-right section and selected **Manage Jenkins**.
- Under the **Security** category, clicked on **Security** settings.
- Scrolled down to the **Git Host Key Verification Configuration** section.
- Under **Host Key Verification Strategy**, selected **No Verification**.



< Add SSH Username with private key >

Scope ?
Global (Jenkins, nodes, items, all child items, etc) ▼

ID ?
cicd-github-private-key

Description ?
my-jenkins-private-key

Username

Create

Private Key

☒ Enter directly

Key

Enter New Secret Below

```
-----BEGIN OPENSSH PRIVATE KEY-----
b3B1bnNzaC1rZXktdjEAAAABAG5vbmUAAAABbm9uZQAAAAAAAAABAAA
BlwAAAAadzcz2gtcn
NhAAAAAwEAAQAAAYEAXNtkdE1fQ13ma0uAlhBuC/SRITU5ESU5anbpo
ayqVboVa59fEW69
H5Jab+LGnuaHVo2/B41U0I3GBYVAit3pCnTZ++1W15Mg1T0VUK4iKa5
aNZtbiWax4xNaV7
dFoRmxT2pk4FtRNfIBhwOZzcCromrm20Sks/Yp2ZhqNybzQTb4PbqYQ
wok9BmIiliCfk/q
Ecd9KGcsUDC6hNOp3N8uRgMdLxG7/rsTk+pKpgM0tjXyHuhc4ZDjKkr
LwB7MwQ/75T076Q
```

Create

Jenkins / Manage Jenkins / Credentials

Credentials [+ Add Credentials](#) ⓘ

github-cicd-PAT	System - Global - Login details to access my GitHub Account	✎ ⋮
cicd-github-private-key jenkins	System - Global - my-jenkins-private-key	✎ ⋮

Stores scoped to Jenkins

System	Domains: Global
--------	-----------------

 Jenkins / Manage Jenkins / Security
Git Host Key Verification Configuration
Host Key Verification Strategy ?
No verification

⚠ This option is generally insecure. Host key will not be verified during SSH connection

Sandbox Configuration
☐ Force the use of the sandbox globally in the system ?

Save Apply

Task 10: Creating Jenkins Pipeline Job

10.1: Create New Jenkins Job

- Click **New Item**
- Under Item Name, type: **terraform-pipeline**
- Select: **Pipeline**
- Click OK.

 Jenkins / All / New Item

New Item

Enter an item name

terraform-pipeline

Select an item type

 Pipeline
Build, test, and deploy using pipelines. Supports stages, parallel work, and running on multiple agents.

 Freestyle project
Classic, general-purpose job type that checks out from up to one SCM, executes build steps serially, followed by post-build steps like archiving artifacts and sending email notifications.

 Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple

OK

Task 11: Connecting GitHub Repository to Jenkins Pipeline

Step 1: Configure GitHub Integration

- Under:
Pipeline
- Set:

Field	Value
Definition	Pipeline script from SCM
SCM	Git

Step 2: Configure Git Repository

- Under Repository URL:
`git@github.com:samuelakosaonyejekwe/terraform-s3-github-actions.git`

Step 3: Select SSH Credential

- Under Credentials:
select:
`github-private-key`

Step 4: Configure Branch and apply & save

- Under Branch Specifier:
`*/main`

 Jenkins

terraform-pipeline

Configure



Configure

General

Triggers

Pipeline

Advanced

Pipeline

Define your Pipeline using Groovy directly or pull it from source control.

Definition

Pipeline script from SCM

SCM

Git

Repositories

Repository URL

git@github.com:samuelakosaonyekwe/terraform-s3-github-actions.git

 Jenkins

terraform-pipeline

Configure



Configure

General

Triggers

Pipeline

Advanced

Credentials

jenkins (my-jenkins-private-key)

+ Add

Advanced

+ Add Repository

Branches to build

Branch Specifier (blank for 'any')

*/main

 Jenkins

terraform-pipeline

Configure



Configure

General

Triggers

Pipeline

Advanced

Script Path

Jenkinsfile

☒ Lightweight checkout

Pipeline Syntax

Advanced

Advanced

Save

Apply

Task 12: Creating Jenkinsfile for Terraform Pipeline

12.1: Create Jenkinsfile Locally

Inside my repository root:

```
terraform-s3-github-actions/
```

I created:

```
Jenkinsfile
```

This Jenkinsfile can be viewed at:

```
https://github.com/samuelakosaonyekwe/terraform-s3-github-actions/blob/main/Jenkinsfile
```

Task 13: Push Jenkinsfile to GitHub

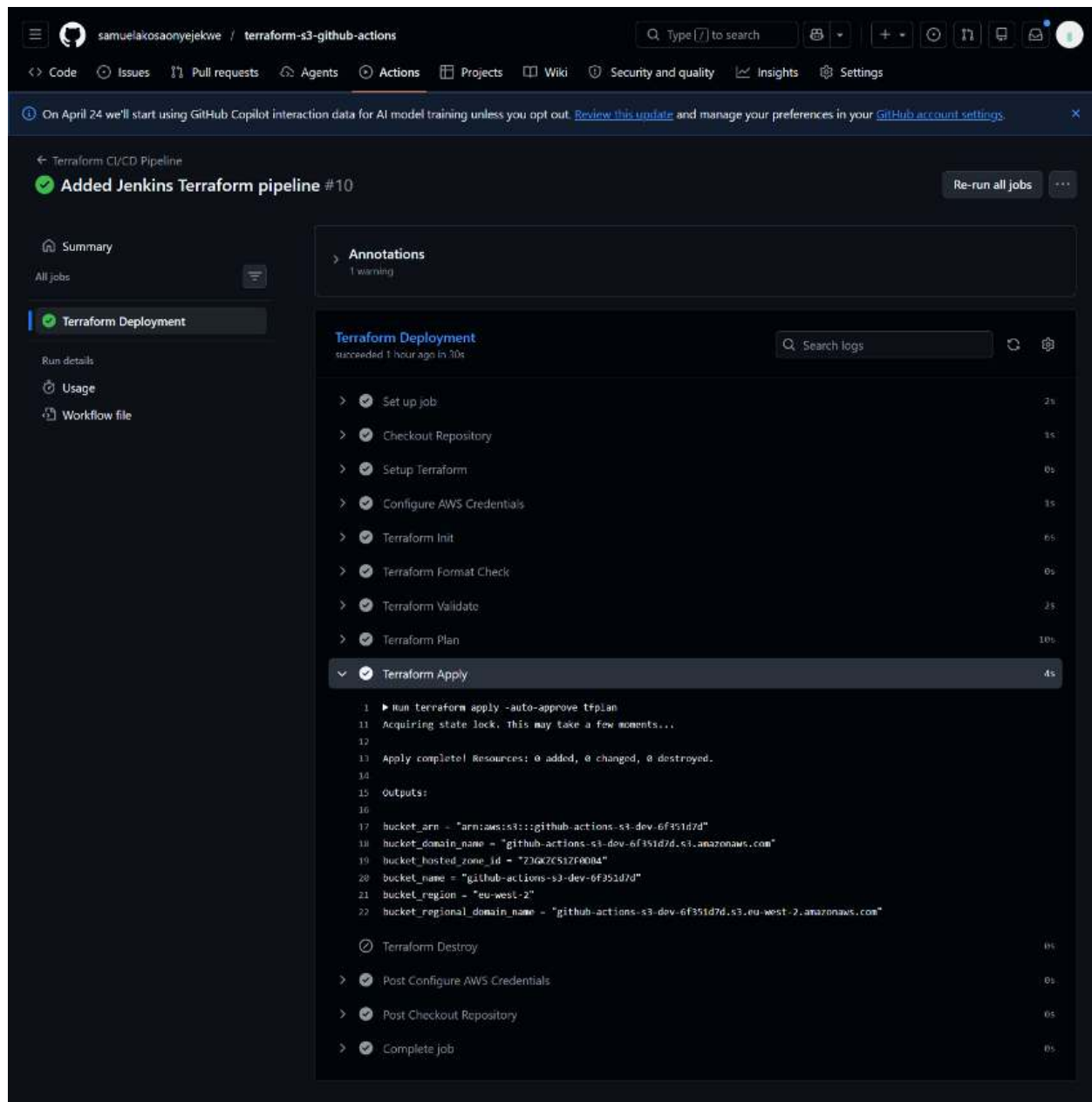
I ran locally:

```
cd ~/terraform-s3-github-actions
git add Jenkinsfile
git commit -m "Added Jenkins Terraform pipeline"
git push origin main
```

```
akosa@AKOSA:~$ cd ~/terraform-s3-github-actions
git add Jenkinsfile

git commit -m "Added Jenkins Terraform pipeline"

git push origin main
[main b42a1c5] Added Jenkins Terraform pipeline
 1 file changed, 56 insertions(+)
 create mode 100644 Jenkinsfile
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 609 bytes | 609.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/samuelakosaonyekwe/terraform-s3-github-actions.git
 2b61fba..b42a1c5  main -> main
akosa@AKOSA:~/terraform-s3-github-actions$ |
```



Task 14: Configuring AWS Credentials for Jenkins User

Jenkins executes Terraform as Jenkins NOT Ubuntu.

14.1: Install AWS Credentials Plugin

Inside Jenkins:

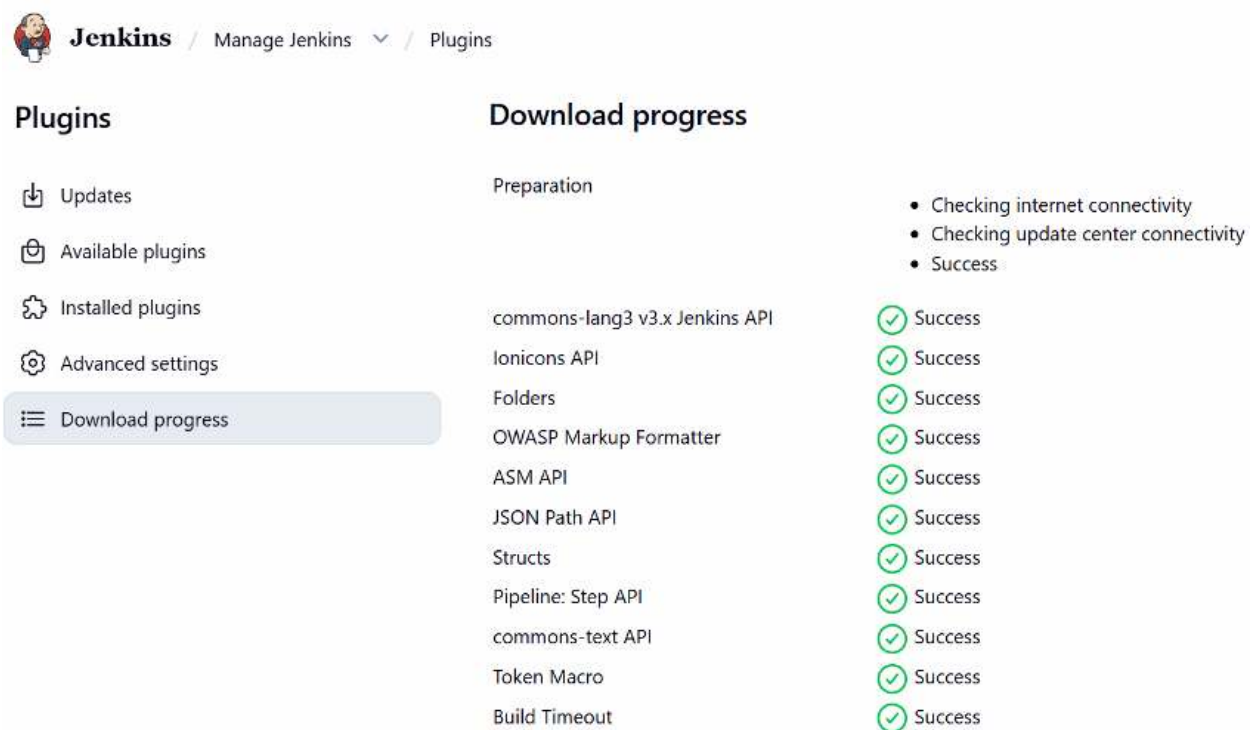
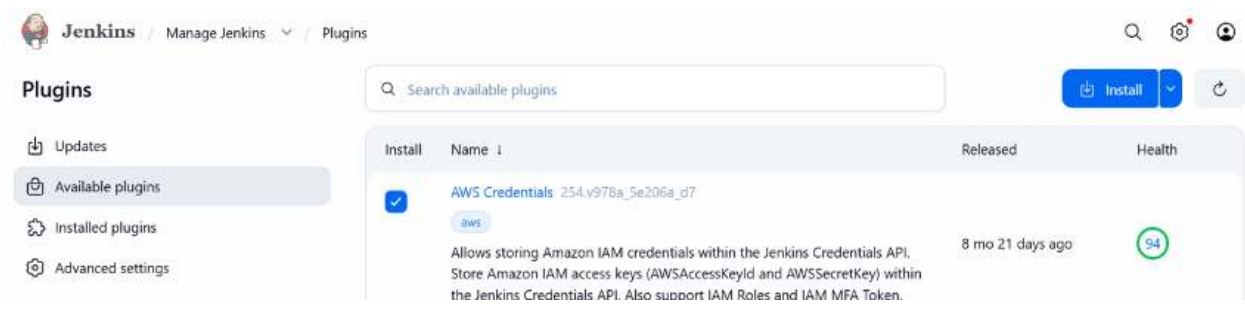
Dashboard → Manage Jenkins → Plugins

Search for:

AWS Credentials

Install it.

Restart Jenkins.



14.2: Add AWS Credentials to Jenkins

- Go to:

Manage Jenkins
↓
Credentials
↓
(System)
↓
Global credentials

- Click:

Add Credentials

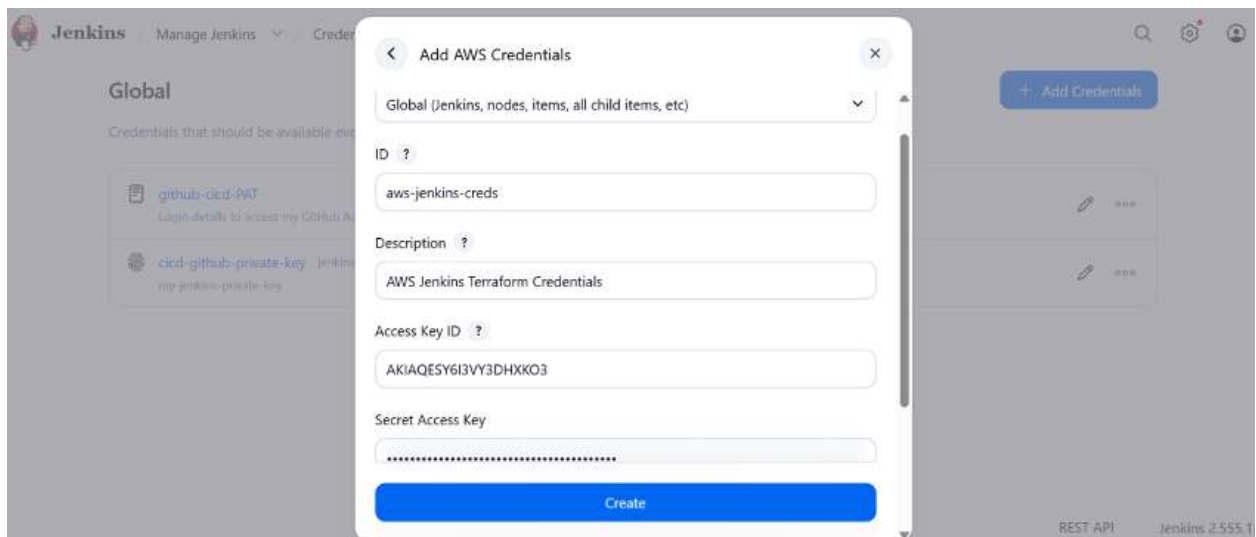
14.3: Enter AWS Keys

- Configure:

Field	Value
Kind	AWS Credentials
Scope	Global
ID	aws-jenkins-creds
Description	AWS Jenkins Terraform Credentials
Access Key ID	MY_ACCESS_KEY
Secret Access Key	MY_SECRET_KEY

- Click:

Create



Jenkins / Manage Jenkins / Credentials / System / Global

Global + Add Credentials

Credentials that should be available everywhere.

 github-cicd-PAT Login details to access my GitHub Account	 
 cicd-github-private-key jenkins my-jenkins-private-key	 
 aws-jenkins-creds AKIAQESY6I3VY3DHXKO3 AWS Jenkins Terraform Credentials	 

Task 15: Test Pipeline Manually

15.1: Build Now

Inside Jenkins:

- open
terraform-pipeline
- click:
Build Now

 **Jenkins** / terraform-pipeline

 Build Now

 Configure

 Delete Pipeline

 Stages

 Rename

 Pipeline Syntax

Builds >

Filter

Today

✓ #2 8:02 PM

 **Jenkins** / terraform-pipeline / #2

terraform.tfvars

```
bucket_arn = "arn:aws:s3:::github-actions-s3-dev-b7351d7d"
bucket_domain_name = "github-actions-s3-dev-6f351d7d.s3.amazonaws.com"
bucket_hosted_zone_id = "Z3GKZC51ZF0084"
bucket_name = "github-actions-s3-dev-6f351d7d"
bucket_region = "eu-west-2"
bucket_regional_domain_name = "github-actions-s3-dev-6f351d7d.s3.eu-west-2.amazonaws.com"

[Pipeline] }
[Pipeline] // withCredentials
[Pipeline] }
[Pipeline] // dir
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Task 16: Realistic Infrastructure Test

Since GitHub Actions already created the original S3 bucket, my Jenkins pipeline should now provision a **NEW** resource.

I created another S3 bucket through Terraform.

16.1: Modified Terraform

I modified the terraform/main.tf file, as can be seen at:

<https://github.com/samuelakosaonyejekwe/terraform-s3-github-actions/blob/main/terraform/main.tf>

Task 17: Pushing Terraform Change

I ran:

```
cd ~/terraform-s3-github-actions
git add .
git commit -m "Added Jenkins pipeline S3 bucket"
git push origin main
```

```
akosa@AKOSA:~/terraform-s3-github-actions$ cd ~/terraform-s3-github-actions
git add .

git commit -m "Added Jenkins pipeline S3 bucket"

git push origin main
[main 1efb9cd] Added Jenkins pipeline S3 bucket
 1 file changed, 150 insertions(+), 8 deletions(-)
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 1.42 KiB | 1.42 MiB/s, done.
Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/samuelakosaonyejekwe/terraform-s3-github-actions.git
 024d751..1efb9cd  main -> main
akosa@AKOSA:~/terraform-s3-github-actions$ |
```

What happens is this:

```
Push to GitHub
↓
Jenkins Pipeline Triggered
↓
Terraform Init
Terraform Validate
Terraform Plan
Terraform Apply
↓
New AWS S3 Bucket Created
```

A new S3 bucket was provisioned as can be verified in AWS console.

General purpose buckets (14) [Info](#)

Copy ARN

Empty

Delete

Create bucket

Buckets are containers for data stored in S3.

Q samuel-j

X

1 match

< 1 >

	Name	AWS Region	Creation date
	samuel-jenkins-pipeline-4cbab337	Europe (London) eu-west-2	May 11, 2026, 23:34:14 (UTC+03:00)

Page 65 of 97

Task 18: Configure GitHub Webhook

- Inside GitHub repository:

Settings → Webhooks

Payload URL:

`http://18.171.241.211:8080/github-webhook/`

Content Type:

`application/json`

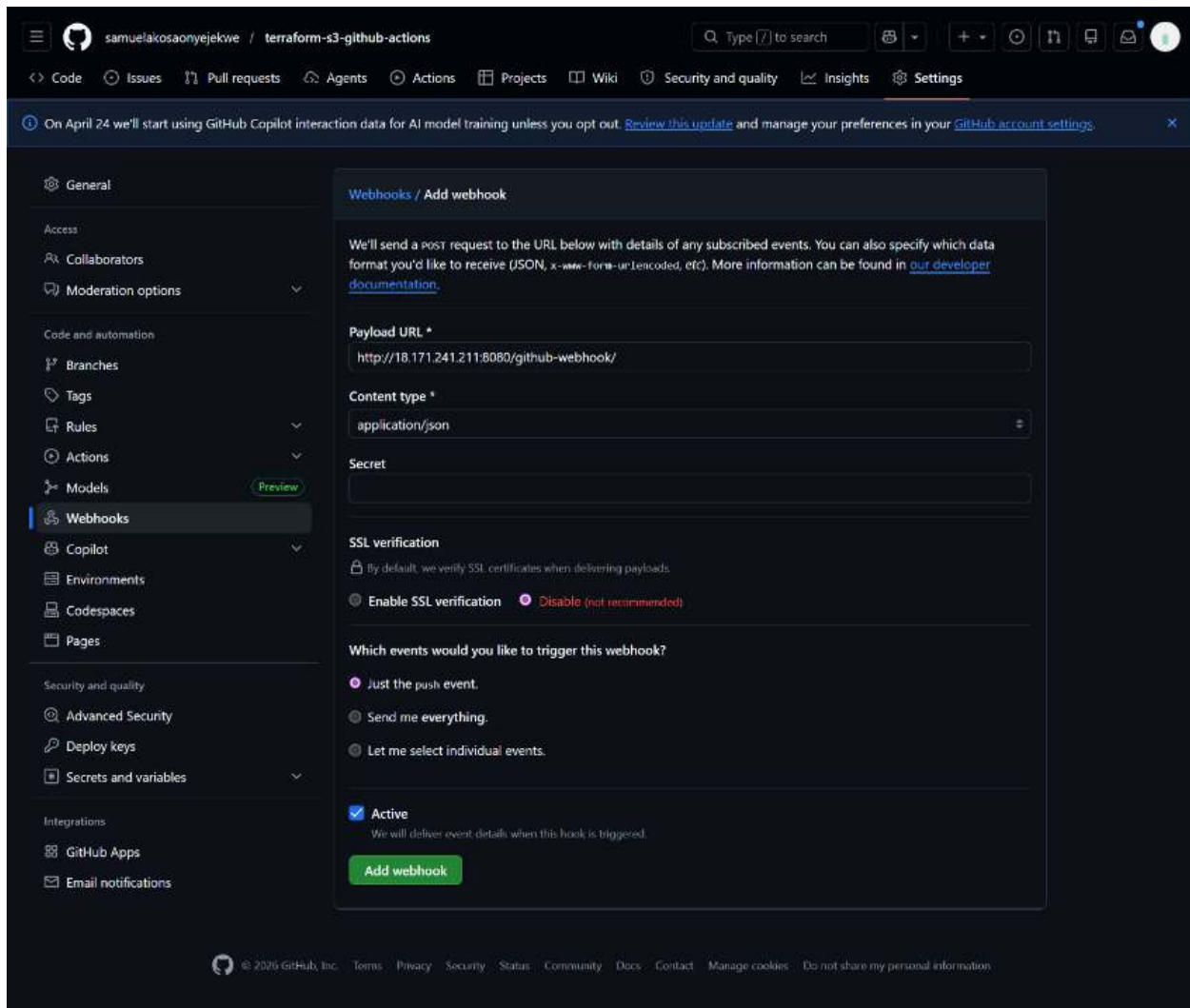
Event:

Just the push event

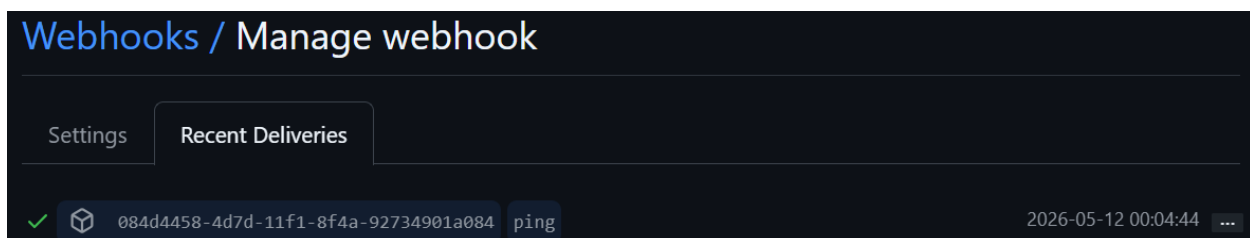
Active:

☒ checked

Save webhook.



The webhook was confirmed to be working after selecting the webhook and clicking on **Recent Deliveries**.



Task 19: Enabling Webhook in Jenkins Job Configuration

I enabled:

Build Triggers

☒ GitHub hook trigger for GITScm polling

Step 1: Open Pipeline Job

- Go to:
Jenkins Dashboard
→ terraform-pipeline

Step 2: Click Configure

- On the left menu:
Configure

Step 3: Open General Section

Scrolled upward until I saw:

→ Build Triggers

It is between:

- General
- Pipeline

sections.

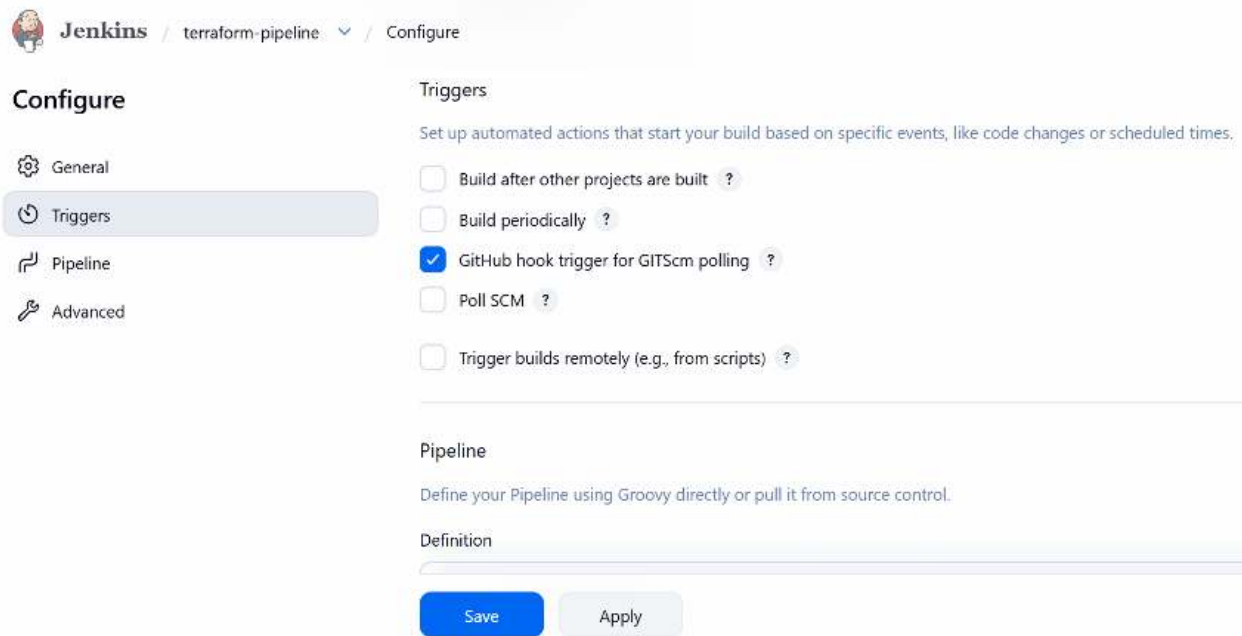
Step 4: Enable GitHub Trigger

- Find:
GitHub hook trigger for GITScm polling

and tick the checkbox.

Step 5: Save

- Scrolled down and clicked:
Save



This tells Jenkins to automatically start this pipeline whenever GitHub sends a webhook push event.

Without this checkbox:

- GitHub webhook works
- but Jenkins ignores the push

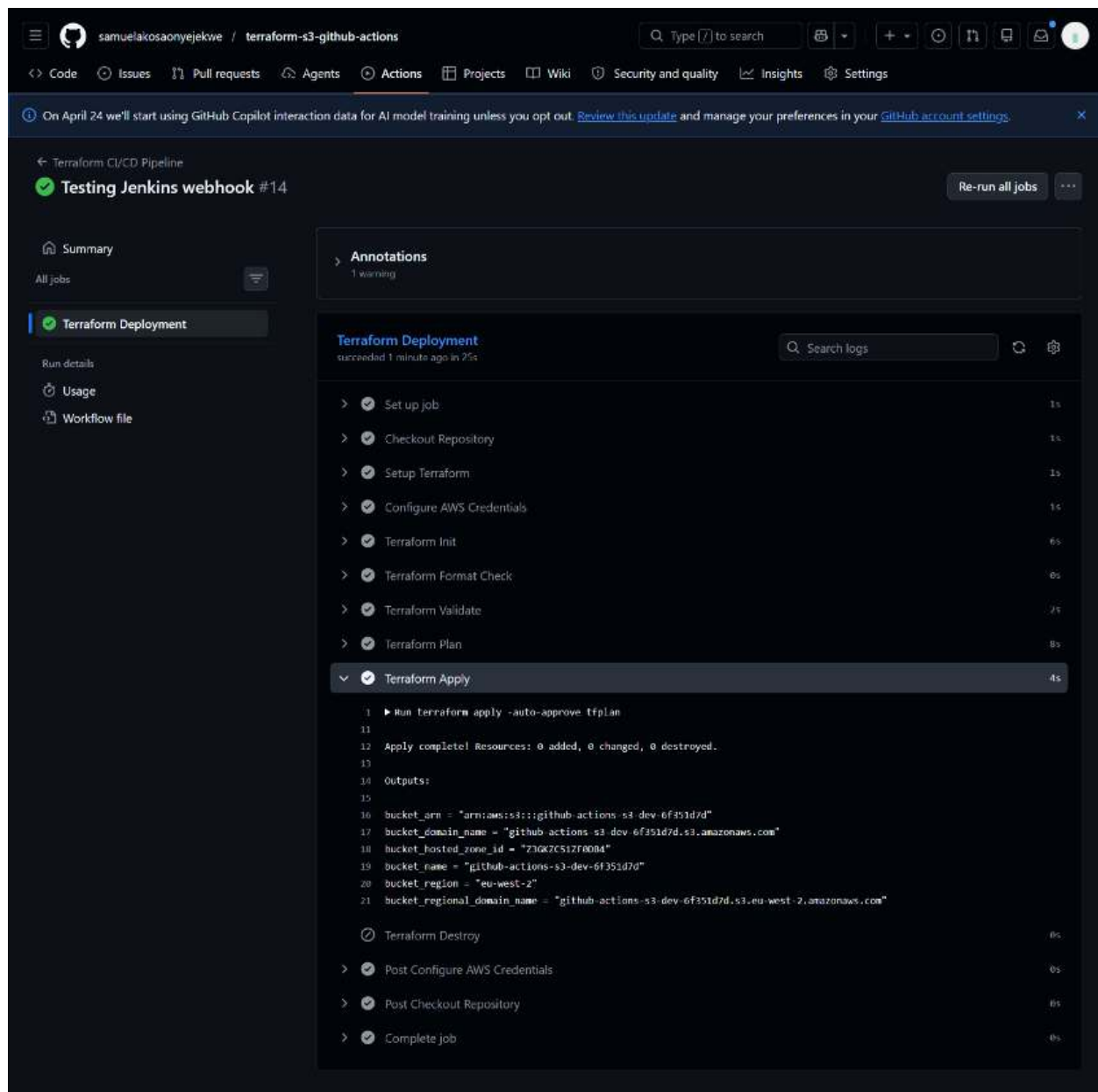
Task 20: Testing the Webhook

I triggered a real push. I made a tiny change on my README.md file.

From my Ubuntu/WSL terminal, I ran:

```
cd ~/terraform-s3-github-actions
echo "# webhook test" >> README.md
git add .
git commit -m "Testing Jenkins webhook"
git push origin main
```

```
akosa@AKOSA:~/terraform-s3-github-actions$ cd ~/terraform-s3-github-actions
echo "# webhook test" >> README.md
git add .
git commit -m "Testing Jenkins webhook"
git push origin main
[main b08894b] Testing Jenkins webhook
 1 file changed, 1 insertion(+), 1 deletion(-)
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 321 bytes | 107.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/samuelakosaonyekwe/terraform-s3-github-actions.git
 1efb9cd..b08894b  main -> main
akosa@AKOSA:~/terraform-s3-github-actions$ |
```



20.1: In GitHub

Webhook sends push event to Jenkins.

Under:

Recent Deliveries

I now see:

push

instead of only:

ping

20.2 In Jenkins

My pipeline automatically started.

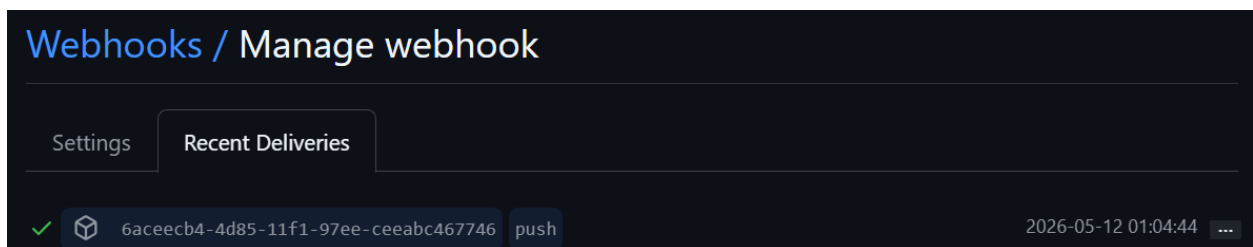
I now see:

Started by GitHub push

instead of:

Started by user

A new build appears automatically.



Task 21: Explaining Error that Occurred after Testing Webhook

My pipeline is working well overall.

From the console output:

- GitHub webhook triggered Jenkins successfully
- Jenkins cloned the repo successfully
- Jenkins read the Jenkinsfile
- Terraform was found and executed
- terraform init succeeded
- terraform validate succeeded

So, my CI/CD pipeline is already functioning correctly up to the Terraform planning stage.

The failure (#3 build) occurred during:

terraform plan

Specifically:

Error acquiring the state lock

and:

ConditionalCheckFailedException

```
[33m|[0m |[0m[1m[33mWarning: |[0m|[0m[1mDeprecated Parameter|[0m
[33m|[0m |[0m
[33m|[0m |[0m[0mThe parameter "dynamodb_table" is deprecated. Use parameter "use_lockfile"
[33m|[0m |[0minstead.
[33m'|[0m|[0m
[31m|[0m|[0m
[31m|[0m |[0m[1m[31mError: |[0m|[0m[1mError acquiring the state lock|[0m
[31m|[0m |[0m
[31m|[0m |[0m[0mError message: operation error DynamoDB: PutItem, https response error
[31m|[0m |[0m[0mStatusCode: 400, RequestID:
[31m|[0m |[0m[0m7ALSQNDN1I1M406509ITTCEA6FVV4KQNS05AEMVJF66Q9ASUAAJG,
[31m|[0m |[0m[0mConditionalCheckFailedException: The conditional request failed
[31m|[0m |[0m[0mLock Info:
[31m|[0m |[0m |[0m ID:          66685db4-bca7-fa1b-886c-2cc79c69a01f
[31m|[0m |[0m |[0m Path:          samuelakosaonyejekwe-tf-state-eu-west-2/terraform/s3-github-
actions/terraform.tfstate
[31m|[0m |[0m |[0m Operation: OperationTypePlan
[31m|[0m |[0m |[0m Who:          runner@runnervmorf1
[31m|[0m |[0m |[0m Version:      1.8.5
[31m|[0m |[0m |[0m Created:      2026-05-11 22:05:10.239434427 +0000 UTC
```

Terraform uses my DynamoDB table to lock the Terraform state so that:

- two users
- or two pipelines

cannot modify infrastructure simultaneously.

Right now:

- a previous lock still exists
- or Terraform thinks another operation is running.

The issue is only stale Terraform state lock.

- Then enable:

Do not allow concurrent builds

This is extremely important for Terraform pipelines.

Terraform state must NOT be modified simultaneously.

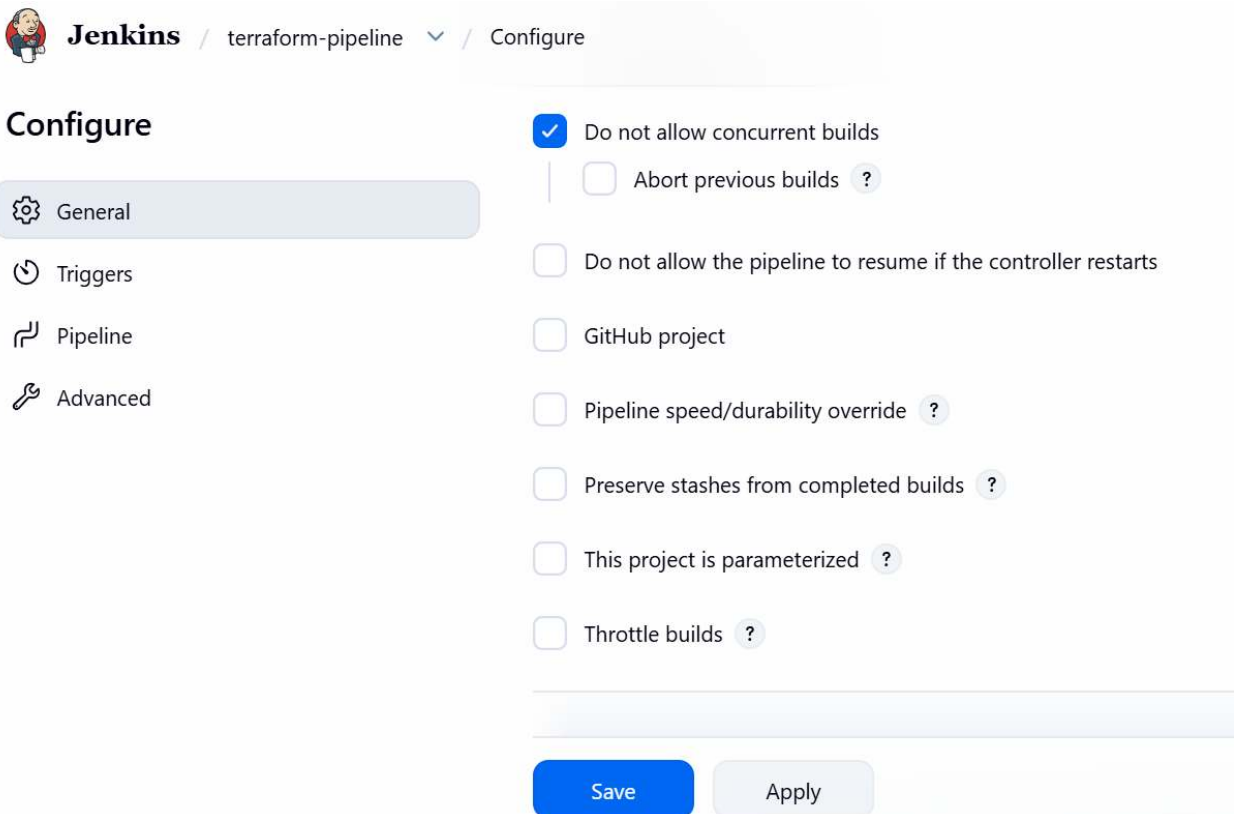
Without this setting:

- Jenkins may trigger another build
- while previous state operation is still active
- causing lock conflicts again.

- After Enabling

Click:

Save



The screenshot shows the Jenkins web interface. At the top, there's a breadcrumb trail: 'Jenkins / terraform-pipeline / Configure'. Below this, the 'Configure' section is active, showing a sidebar with 'General', 'Triggers', 'Pipeline', and 'Advanced' tabs. The 'General' tab is selected. In the main content area, there's a list of configuration options. The first option, 'Do not allow concurrent builds', is checked with a blue checkbox. Below it, 'Abort previous builds' is unchecked. Other options like 'Do not allow the pipeline to resume if the controller restarts', 'GitHub project', 'Pipeline speed/durability override', 'Preserve stashes from completed builds', 'This project is parameterized', and 'Throttle builds' are all unchecked. At the bottom of the configuration area, there are two buttons: 'Save' (in blue) and 'Apply' (in light blue).

Jenkins / terraform-pipeline / Configure

Configure

- General
- Triggers
- Pipeline
- Advanced

- ☒ Do not allow concurrent builds
- ☐ Abort previous builds ?
- ☐ Do not allow the pipeline to resume if the controller restarts
- ☐ GitHub project
- ☐ Pipeline speed/durability override ?
- ☐ Preserve stashes from completed builds ?
- ☐ This project is parameterized ?
- ☐ Throttle builds ?

Save Apply

Step 2: Clearing DynamoDB Lock

- Used the AWS CLI to list available DynamoDB tables in the eu-west-2 region
- Confirmed that the Terraform lock table being used was:
`terraform-state-lock`
- Scanned the DynamoDB lock table using:
`aws dynamodb scan`
- Verified that a Terraform state lock entry existed inside the table
- Identified the active LockID associated with the Terraform remote state file
- Confirmed that Terraform state locking was functioning correctly through DynamoDB
- Navigated into the Terraform working directory:
`~/terraform-s3-github-actions/terraform`
- Attempted to execute:
`terraform force-unlock`
- Encountered a backend initialisation error because the Terraform S3 backend had not yet been initialised locally
- Terraform requested execution of:
`terraform init`
- Executed:
`terraform init`
- Successfully initialised the Terraform S3 backend configuration
- Downloaded and initialised required Terraform providers:
 - hashicorp/aws
 - hashicorp/random
- Generated the provider lock file:
`.terraform.lock.hcl`
- Confirmed Terraform backend and provider configuration were operational
- Manually removed the stale Terraform state lock from DynamoDB using:
`aws dynamodb delete-item`
- Deleted the specific LockID associated with the failed Terraform operation
- Cleared the stale Terraform lock entry from the DynamoDB table
- Re-scanned the DynamoDB lock table after deletion

- Confirmed successful lock removal when the scan returned:
"Items": []
- Verified that no active Terraform state lock remained in the backend table
- Overall, the screenshots demonstrate:
 - successful Terraform backend initialisation
 - successful DynamoDB state-lock inspection
 - successful stale lock troubleshooting
 - successful manual lock removal
 - operational Terraform remote backend configuration using:
 - S3 state storage
 - DynamoDB state locking
 - AWS CLI administration
 - Jenkins/GitHub-integrated CI/CD workflow management

The following screenshots show bash commands run:

```
akosa@AKOSA:~/terraform-s3-github-actions$ aws dynamodb list-tables --region eu-west-2
{
  "TableNames": [
    "terraform-state-lock"
  ]
}
akosa@AKOSA:~/terraform-s3-github-actions$ |
```

```
akosa@AKOSA:~/terraform-s3-github-actions$ aws dynamodb scan \
--table-name terraform-state-lock \
--region eu-west-2
{
  "Items": [
    {
      "LockID": {
        "S": "samuelakosaonyekwe-tf-state-eu-west-2/terraform/s3-github-actions/terraform.tfstate-md5"
      },
      "Digest": {
        "S": "f127bc6b26535b26e764fe7a0f5f3faf"
      }
    }
  ],
  "Count": 1,
  "ScannedCount": 1,
  "ConsumedCapacity": null
}
akosa@AKOSA:~/terraform-s3-github-actions$ |
```

```

akosa@AKOSA:~/terraform-s3-github-actions$ cd ~/terraform-s3-github-actions/terraform
akosa@AKOSA:~/terraform-s3-github-actions/terraform$ terraform force-unlock 66850b4-bca7-falb-886c-2cc79c69a01f

Error: Backend initialization required, please run "terraform init"

Reason: Initial configuration of the requested backend "s3"

The "backend" is the interface that Terraform uses to store state,
perform operations, etc. If this message is showing up, it means that the
Terraform configuration you're using is using a custom configuration for
the Terraform backend.

Changes to backend configurations require reinitialization. This allows
Terraform to set up the new configuration, copy existing state, etc. Please run
"terraform init" with either the "-reconfigure" or "-migrate-state" flags to
use the current configuration.

If the change reason above is incorrect, please verify your configuration
hasn't changed and try again. At this point, no changes to your existing
configuration or state have been made.

```

```

akosa@AKOSA:~/terraform-s3-github-actions/terraform$ terraform init
Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
Initializing provider plugins...
- Finding hashicorp/random versions matching "~> 3.6"...
- Finding hashicorp/aws versions matching "~> 5.0"...
- Installing hashicorp/random v3.8.1...
- Installed hashicorp/random v3.8.1 (signed by HashiCorp)
- Installing hashicorp/aws v5.100.0...
- Installed hashicorp/aws v5.100.0 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
akosa@AKOSA:~/terraform-s3-github-actions/terraform$ |

```

```

akosa@AKOSA:~/terraform-s3-github-actions/terraform$ aws dynamodb delete-item \
--table-name terraform-state-lock \
--key '{"LockID":{"S":"samuelakosaonyejekwe-tf-state-eu-west-2/terraform/s3-github-actions/terraform.tfstate-md5"}}' \
--region eu-west-2
akosa@AKOSA:~/terraform-s3-github-actions/terraform$ aws dynamodb scan \
--table-name terraform-state-lock \
--region eu-west-2
{
  "Items": [],
  "Count": 0,
  "ScannedCount": 0,
  "ConsumedCapacity": null
}
akosa@AKOSA:~/terraform-s3-github-actions/terraform$

```

21.2: Triggering Jenkins Again

I ran on Ubuntu terminal:

```
cd ~/terraform-s3-github-actions

echo "# retry after disabling concurrent builds" >> README.md

git add .

git commit -m "Retry after disabling concurrent builds"

git push origin main
```

```
akosa@AKOSA:~/terraform-s3-github-actions$ cd ~/terraform-s3-github-actions
echo "# retry after disabling concurrent builds" >> README.md
git add .
git commit -m "Retry after disabling concurrent builds"
git push origin main
[main cdd4ff9] Retry after disabling concurrent builds
 1 file changed, 1 insertion(+)
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 355 bytes | 177.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/samuelakosaonyekwe/terraform-s3-github-actions.git
 cbeea7c..cdd4ff9  main -> main
akosa@AKOSA:~/terraform-s3-github-actions$ |
```

Once I pushed the changes to GitHub, the configured webhook notified Jenkins automatically, causing the Jenkins pipeline to start and execute the workflow again.

GitHub repository: samuelakosaonyejekwe / terraform-s3-github-actions

On April 24 we'll start using GitHub Copilot interaction data for AI model training unless you opt out. [Review this update](#) and manage your preferences in your [GitHub account settings](#).

Terraform CI/CD Pipeline

Retry Jenkins pipeline after lock removal #15

Re-run all jobs

Summary

All jobs

Terraform Deployment

Run details

Usage

Workflow file

Annotations

1 warning

Terraform Deployment

succeeded 8 minutes ago in 28s

Search logs

- Set up job 1s
- Checkout Repository 0s
- Setup Terraform 2s
- Configure AWS Credentials 1s
- Terraform Init 0s
- Terraform Format Check 1s
- Terraform Validate 2s
- Terraform Plan 7s
- Terraform Apply 4s
 - Run terraform apply -auto-approve tfplan
 - Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
 - Outputs:
 - bucket_arn = "arn:aws:s3:::github-actions-s3-dev-6f351d7d"
 - bucket_domain_name = "github-actions-s3-dev-6f351d7d.s3.amazonaws.com"
 - bucket_hosted_zone_id = "Z3QK7C51ZF80M4"
 - bucket_name = "github-actions-s3-dev-6f351d7d"
 - bucket_region = "eu-west-2"
 - bucket_regional_domain_name = "github-actions-s3-dev-6f351d7d.s3.eu-west-2.amazonaws.com"
- Terraform Destroy 0s
- Post Configure AWS Credentials 0s
- Post Checkout Repository 0s
- Complete job 0s

Jenkins

+ New Item

Build History

Project Relationship

Check File Fingerprint

Build Queue

No builds in the queue.

Build Executor Status

0/2

All

S	W	Name	Last Success	Last Failure	Last Duration
✓	☁	terraform-pipeline	11 min #5	42 min #4	34 sec

Icon: S M L

The Terraform state lock conflict occurred because both Jenkins and GitHub Actions were simultaneously attempting to execute Terraform operations against the same remote backend infrastructure state. Both systems were configured to use the same:

- S3 Terraform state file
- DynamoDB lock table
- Terraform backend configuration

As a result, when one CI/CD system acquired the Terraform state lock, the other system was denied access and failed with a DynamoDB conditional lock error. This demonstrated how concurrent infrastructure deployments from multiple automation platforms can create Terraform state-lock conflicts when they target the same backend resources.

Jenkins Build History Explained

- Jenkins pipeline was successfully integrated with GitHub using webhook automation.
- Build #2 completed successfully after fixing the Jenkins SSH credential configuration.
- Build #3 confirmed that GitHub webhook triggering was working correctly because Jenkins automatically started after a GitHub push.
- Build #3 failed during Terraform execution due to Terraform state lock conflict in the remote backend.
- The failure demonstrated that Terraform DynamoDB locking protection was functioning correctly to prevent simultaneous infrastructure modification.
- Build #4 was triggered after manual Terraform lock removal but still failed due to concurrent Terraform execution attempts.
- The failure occurred because both Jenkins and GitHub Actions were running Terraform operations simultaneously against the same backend state.
- Build #5 completed successfully after disabling concurrent Jenkins builds and mitigating Terraform state-lock conflicts.
- The successful build confirmed stable Jenkins webhook automation and successful Terraform pipeline execution.
- The screenshots demonstrate real-world DevOps troubleshooting involving:
 - Terraform remote state management
 - DynamoDB state locking
 - GitHub webhook automation

- Jenkins CI/CD orchestration
- Infrastructure concurrency conflicts
- Multi-pipeline Terraform execution
- The ultimate production-grade solution is to:
 - Use only one CI/CD platform for Terraform deployments, or
 - Separate environments using different Terraform state files and lock tables.


#5 (May 11, 2026, 11:14:28 PM)

 Add description

Keep this build forever


 Started by [GitHub push by samuelakosaonyejekwe](#)

Started 5 hr 7 min ago
Took [34 sec](#)


 This run spent:

- 9.1 sec waiting;
- 34 sec build duration;
- 43 sec total from scheduled to completion.


Revision: [cdd4ff95ba0f50c2fe1e5a211dff5645b6902b3b](#)
Repository: [git@github.com:samuelakosaonyejekwe/terraform-s3-github-actions.git](#)

- [refs/remotes/origin/main](#)




 Retry after disabling concurrent builds [cdd4ff9](#)
 |  [samuelonyejekwe852](#) at 11:14 PM 5/11/26


#4 (May 11, 2026, 10:44:13 PM)

 Add description


 Started by [GitHub push by samuelakosaonyejekwe](#)


 This run spent:

- 5.8 sec waiting;
- 21 sec build duration;
- 27 sec total from scheduled to completion.


Revision: [cb6ea7cb1232395697a5b933a57ef898cee10883](#)
Repository: [git@github.com:samuelakosaonyejekwe/terraform-s3-github-actions.git](#)

- [refs/remotes/origin/main](#)




 Retry Jenkins pipeline after lock removal [cb6ea7c](#)
 |  [samuelonyejekwe852](#) at 10:44 PM 5/11/26

 #3 (May 11, 2026, 10:04:53 PM)

Add description

 Started by GitHub push by samuelakosaonyejekwe

 This run spent:

- 6.4 sec waiting;
- 22 sec build duration;
- 29 sec total from scheduled to completion.

 git

Revision: b08894b7c6f1ff8b735c684b984a35370ed7b0cf

Repository: git@github.com:samuelakosaonyejekwe/terraform-s3-github-actions.git

- refs/remotes/origin/main



Added Jenkins pipeline S3 bucket 1efb9cd

samuelonyejekwe852 at 8:33 PM 5/11/26

Testing Jenkins webhook b08894b

samuelonyejekwe852 at 10:04 PM 5/11/26

 #2 (May 11, 2026, 8:02:19 PM)

Add description

 Started by user Akosa Samuel Onyejekwe

 This run spent:

- 13 ms waiting;
- 39 sec build duration;
- 39 sec total from scheduled to completion.

 git

Revision: 024d7515a30e3ed9328838fad0919f7cd12e4306

Repository: git@github.com:samuelakosaonyejekwe/terraform-s3-github-actions.git

- refs/remotes/origin/main



Fixed Jenkins SSH credential ID 024d751

samuelonyejekwe852 at 8:00 PM 5/11/26

When the Terraform state lock was removed, the Jenkins pipeline executed successfully after the GitHub webhook trigger, as shown in Builds #5, #4, and #3. However, the GitHub Actions workflow failed because both Jenkins and GitHub Actions were configured to run Terraform operations simultaneously against the same remote backend infrastructure state. As a result, both CI/CD systems attempted to acquire the Terraform state lock on the shared S3 backend and DynamoDB lock table at nearly the same time. Although one pipeline successfully obtained the lock and completed execution, the other pipeline failed due to Terraform state concurrency protection mechanisms designed to prevent simultaneous infrastructure modifications and potential state corruption.

The ultimate production-grade solution is to avoid running multiple Terraform deployment pipelines against the same backend state concurrently. In enterprise DevOps environments, a single CI/CD platform should be designated as the primary infrastructure deployment orchestrator. Therefore, either Jenkins or GitHub Actions should exclusively handle Terraform apply and destroy operations for the shared infrastructure environment.

Additional production-grade mitigation strategies include:

- Enabling concurrency control within GitHub Actions using:
 - concurrency:
 - group: terraform-production
 - cancel-in-progress: false
- Enabling:
 - Do not allow concurrent builds within Jenkins pipeline configuration.
- Separating environments and Terraform states:
 - Jenkins → development environment
 - GitHub Actions → production environment
- Using separate backend state files and DynamoDB lock tables per environment.
- Introducing pipeline approval gates and deployment sequencing.
- Implementing Terraform Cloud or enterprise remote execution platforms for centralised state management and locking orchestration.

This issue demonstrates a real-world DevOps challenge involving distributed CI/CD orchestration, Terraform remote state management, and infrastructure concurrency control in multi-pipeline environments.

See screenshots that follow.

 **#5 (May 11, 2026, 11:14:28 PM)**

 Add description

 Keep this build forever

 Started by [GitHub push by samuelakosaonyejekwe](#)

Started 5 hr 7 min ago
Took [34 sec](#)

 This run spent:

- 9.1 sec waiting;
- 34 sec build duration;
- 43 sec total from scheduled to completion.

 **Revision:** [cdd4ff95ba0f50c2fe1e5a211dff5645b6902b3b](#)

Repository: [git@github.com:samuelakosaonyejekwe/terraform-s3-github-actions.git](#)

- [refs/remotes/origin/main](#)

  **Retry after disabling concurrent builds** [cdd4ff9](#)

|  [samuelonyejekwe852](#) at 11:14 PM 5/11/26

Page 85 of 97

 #4 (May 11, 2026, 10:44:13 PM)

 Add description



Started by GitHub push by samuelakosaonyekwe



This run spent:

- 5.8 sec waiting;
- 21 sec build duration;
- 27 sec total from scheduled to completion.



Revision: cbeea7cb1232395697a5b933a57ef898cee10883

Repository: git@github.com:samuelakosaonyekwe/terraform-s3-github-actions.git

- refs/remotes/origin/main



 Retry Jenkins pipeline after lock removal cbeea7c
|  samuelonyekwe852 at 10:44 PM 5/11/26

 #3 (May 11, 2026, 10:04:53 PM)

 Add description



Started by GitHub push by samuelakosaonyekwe



This run spent:

- 6.4 sec waiting;
- 22 sec build duration;
- 29 sec total from scheduled to completion.



git

Revision: b08894b7c6f1ff8b735c684b984a35370ed7b0cf

Repository: git@github.com:samuelakosaonyekwe/terraform-s3-github-actions.git

- refs/remotes/origin/main

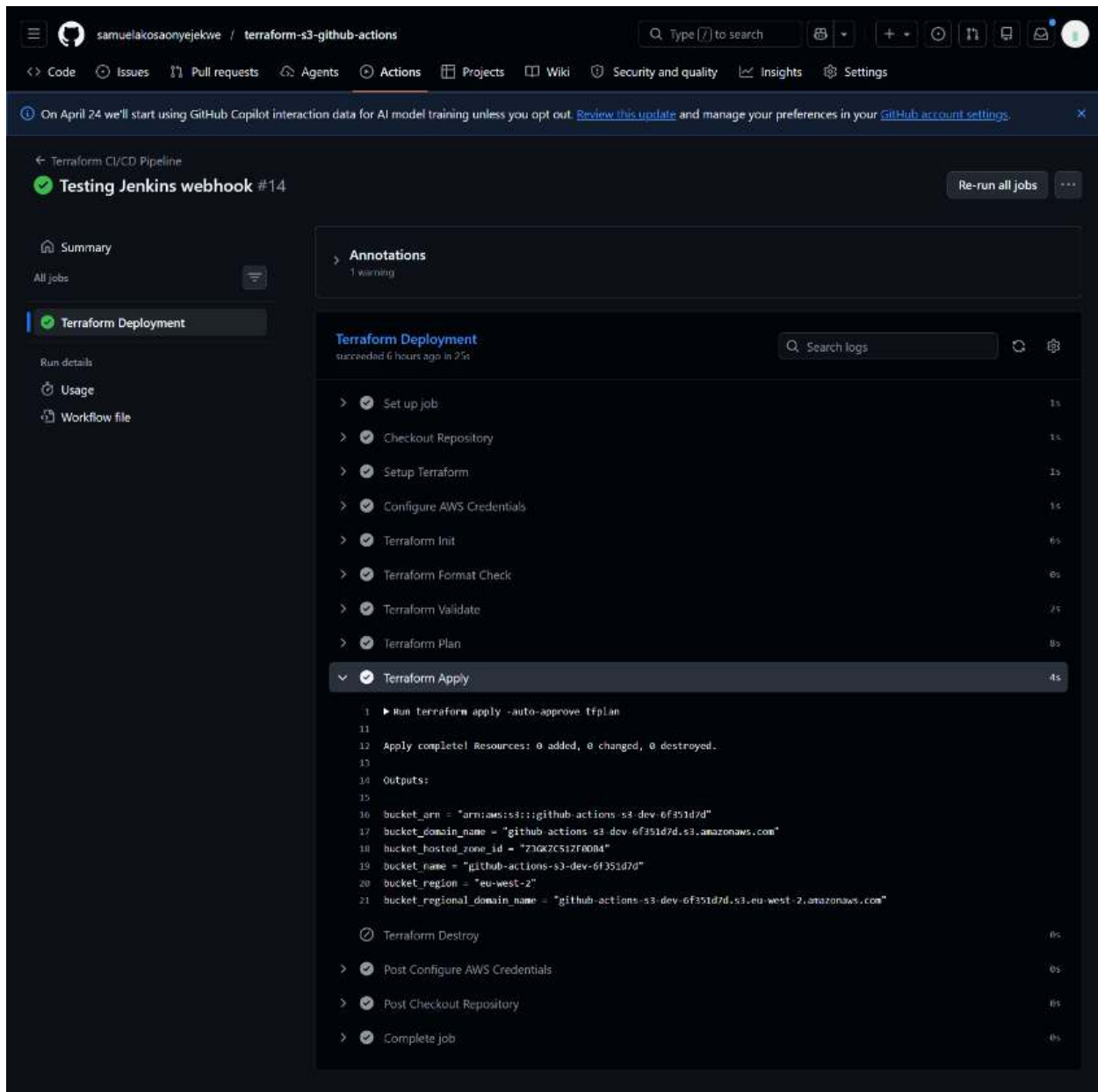


Added Jenkins pipeline S3 bucket 1efb9cd

|  samuelonyekwe852 at 8:33 PM 5/11/26

Testing Jenkins webhook b08894b

|  samuelonyekwe852 at 10:04 PM 5/11/26



Jenkinsfile Integration and CI/CD Automation

The terraform-s3-github-actions project was enhanced through the introduction of a root-level Jenkinsfile, which enabled integration with Jenkins for automated CI/CD execution. The Jenkinsfile transformed the project from a manually executed Terraform deployment into a fully automated Infrastructure-as-Code pipeline capable of performing source code checkout, Terraform initialisation, validation, planning, and deployment automatically whenever code was pushed to GitHub.

Written using Jenkins Declarative Pipeline syntax, the Jenkinsfile introduced a structured, stage-based workflow that included stages such as Checkout Code, Terraform Version, Terraform Init, Terraform Validate, Terraform Plan, and Terraform Apply. This improved deployment transparency, debugging, monitoring, and operational control by separating the CI/CD process into clearly defined stages.

The Jenkinsfile also implemented the “Pipeline as Code” concept, allowing deployment workflows to be version-controlled directly alongside the infrastructure source code. This made the CI/CD process reproducible, auditable, maintainable, and scalable across different environments. Through GitHub webhook integration, Jenkins could automatically trigger infrastructure deployments on every repository push event, reducing manual intervention and minimising human error.

A key justification for the Jenkinsfile was its role in centralising Terraform automation and safely coordinating remote backend operations involving Amazon S3 state storage and DynamoDB state locking. The pipeline therefore supported not only automation but also deployment governance, Terraform state management, operational consistency, and enterprise-style DevOps practices aligned with modern cloud infrastructure deployment standards.

```
terraform-s3-github-actions/  
├── Jenkinsfile  
├── .github/  
│   └── workflows/  
│       └── terraform.yml  
├── scripts/  
│   ├── deploy.sh  
│   └── destroy.sh  
├── terraform/  
│   ├── backend.tf  
│   ├── main.tf  
│   ├── outputs.tf  
│   ├── providers.tf  
│   ├── terraform.tfvars  
│   ├── variables.tf  
│   └── versions.tf  
│   └── modules/  
│       └── s3_bucket/  
│           ├── main.tf  
│           ├── outputs.tf  
│           └── variables.tf  
├── .gitignore  
└── README.md
```

AKOSA > Cloud class > EXTRA PROJECTS > terraform-s3-github-actions >			
Sort ▾ View ▾ ...			
Name	Date modified	Type	Size
📁 .git	11/05/2026 11:57 pm	File folder	
📁 .github	11/05/2026 11:57 pm	File folder	
📁 scripts	11/05/2026 11:57 pm	File folder	
📁 terraform	11/05/2026 11:57 pm	File folder	
⚙️ .gitignore	10/05/2026 12:21 am	Git Ignore Source File	3 KB
📄 Jenkinsfile	11/05/2026 11:00 pm	File	2 KB
📄 README	10/05/2026 12:00 am	MD File	4 KB

The screenshot shows the VS Code editor interface. On the left, the Explorer pane shows the file structure of the 'terraform-s3-github-actions' project, including folders like '.github', 'scripts', and 'terraform', and files like '.gitignore', 'Jenkinsfile', and 'README.md'. The main editor area displays the 'Jenkinsfile' with the following content:

```

1 pipeline {
2     agent any
3
4     environment {
5         AWS_DEFAULT_REGION = 'eu-west-2'
6     }
7
8     stages {
9
10        stage('Checkout Code') {
11            steps {
12                git branch: 'main',
13                  credentialsId: 'cicd-github-private-key',
14                  url: 'git@github.com:samuelakosaonyekwe/terraform-s3-github-actions.git'
15            }
16        }
17
18        stage('Terraform Version') {
19            steps {
20                sh 'terraform version'
21            }
22        }
23
24        stage('Terraform Init') {
25
26        }
27    }
28 }

```

At the bottom, the Terminal pane shows a bash shell prompt: `akosa@AKOSA:~/terraform-s3-github-actions$`. The status bar at the very bottom indicates the current file is 'main', the editor is at line 19, column 20, and the encoding is UTF-8.

The GitHub Actions Flow

Terraform Layer

Defines:

- S3 bucket
- Encryption
- Versioning
- Security
- Lifecycle policies

GitHub Actions Layer

Automates:

- terraform init
- terraform validate
- terraform plan
- terraform apply

whenever code is pushed to GitHub.

Production Workflow

The production flow for this DevOps project begins with me making infrastructure changes locally within the Terraform IaC repository using Visual Studio Code running on Ubuntu WSL. The repository contains modular Terraform configuration files, reusable modules, shell automation scripts, GitHub Actions workflow files, and a Jenkinsfile for pipeline automation.

The workflow starts when I update Terraform configuration files such as:

- main.tf
- variables.tf
- backend.tf
- terraform.tfvars
- Jenkinsfile
- GitHub workflow files

After modifications are completed locally, Git commands are executed:

- `git add .`
- `git commit`
- `git push origin main`

This push operation sends the updated Infrastructure-as-Code repository to the GitHub remote repository.

Once the code reaches GitHub, two automation systems become involved:

- GitHub Actions
- Jenkins Pipeline

GitHub Actions automatically detects the push event through the configured workflow file: `.github/workflows/terraform.yml` which initiates the CI/CD pipeline.

The GitHub Actions workflow then performs automated Terraform operations including:

- `terraform init`
- `terraform validate`
- `terraform plan`
- `terraform apply`
- `terraform destroy` (when manually selected)

AWS authentication is securely handled using encrypted GitHub repository secrets:

- `AWS_ACCESS_KEY_ID`
- `AWS_SECRET_ACCESS_KEY`
- `AWS_REGION`

Terraform then connects to AWS using the configured provider and remote backend. The backend uses:

- an S3 bucket for Terraform state storage
- a DynamoDB table for Terraform state locking

This prevents simultaneous infrastructure modifications and protects the Terraform state from corruption.

The Terraform layer provisions and manages AWS infrastructure resources such as:

- S3 buckets

- versioning
- encryption
- lifecycle policies
- backend state management resources

Simultaneously, Jenkins is integrated into the production workflow through GitHub webhooks. A webhook configured in GitHub sends push-event notifications directly to the Jenkins server endpoint:

`/github-webhook/`

The Jenkins pipeline job:
`terraform-pipeline`
 is configured to:

- pull source code from GitHub
- authenticate using SSH credentials
- read the Jenkinsfile
- execute Terraform commands automatically

Once triggered, Jenkins performs:

- Terraform initialisation
- Terraform validation
- Terraform planning
- Terraform apply operations

using AWS credentials stored securely within Jenkins Credentials Manager.

The complete production CI/CD workflow therefore becomes:

```

I Push
↓
GitHub Repository Updated
↓
GitHub Actions Triggered
↓
Terraform Validation and Deployment
↓
GitHub Webhook Sends Notification
↓
Jenkins Pipeline Triggered
↓
Terraform Executes via Jenkins
  
```

↓
AWS Infrastructure Provisioned Automatically

This production-style workflow demonstrates:

- Infrastructure-as-Code automation
- Continuous Integration
- Continuous Deployment
- GitOps practices
- Terraform remote state management
- Multi-platform CI/CD orchestration
- Automated infrastructure provisioning on AWS

Terraform GitHub CI/CD Pipeline Workflow Runs

The following screenshot shows the GitHub Actions dashboard for the repository:
`terraform-s3-github-actions`

The repository is using a GitHub Actions workflow named:
`Terraform CI/CD Pipeline`

to automate Terraform infrastructure deployment and management.

The dashboard displays the workflow execution history containing multiple CI/CD pipeline runs.

Successful workflow runs are indicated with green check icons, while failed runs are indicated with red cross icons.

The successful workflow runs shown include:

- Added Terraform destroy workflow
- Added deployment and destroy automation scripts
- Fixed Terraform lifecycle configuration warning
- Trigger workflow rerun

The failed workflow runs shown include:

- Formatted Terraform files
- Fixed Terraform workflow directory

The workflow history demonstrates progressive troubleshooting and improvement of the Terraform CI/CD pipeline configuration.

The pipeline executions were performed on the:

Main

branch.

The displayed durations indicate the execution time for each workflow run, ranging from a few seconds to over one minute depending on the Terraform operations performed.

The Actions dashboard confirms that:

- GitHub Actions CI/CD automation is fully configured
- Terraform workflows are actively being tested and improved
- Infrastructure deployment and destroy operations are integrated into the DevOps pipeline
- The repository follows a production-style IaC workflow using Terraform and GitHub Actions.

The screenshot shows the GitHub Actions dashboard for the repository 'samuelakosaonyejekwe / terraform-s3-github-actions'. The left sidebar contains the 'Actions' section with a 'New workflow' button and a list of workflow runs. The main area displays 'All workflows' with a search bar and a table of workflow runs. The table lists six workflow runs, each with a status icon, title, description, workflow name, branch, and duration. The runs are: 'Added Terraform destroy workflow' (green checkmark, 34s), 'Added deployment and destroy automation scripts' (green checkmark, 34s), 'Fixed Terraform lifecycle configuration warning' (green checkmark, 1m 35s), 'Trigger workflow rerun' (green checkmark, 1m 47s), 'Formatted Terraform files' (red X, 19s), and 'Fixed Terraform workflow directory' (red X, 13s).

Workflow	Event	Status	Branch	Actor
Added Terraform destroy workflow	Terraform CI/CD Pipeline #9: Commit 2b511ba pushed by samuelakosaonyejekwe	Success	main	Today at 12:25 PM
Added deployment and destroy automation scripts	Terraform CI/CD Pipeline #8: Commit 5673c35 pushed by samuelakosaonyejekwe	Success	main	Today at 11:47 AM
Fixed Terraform lifecycle configuration warning	Terraform CI/CD Pipeline #7: Commit 4577c1c pushed by samuelakosaonyejekwe	Success	main	Today at 10:27 AM
Trigger workflow rerun	Terraform CI/CD Pipeline #6: Commit 5d430d9 pushed by samuelakosaonyejekwe	Success	main	Today at 10:02 AM
Formatted Terraform files	Terraform CI/CD Pipeline #5: Commit 2ebc474 pushed by samuelakosaonyejekwe	Failure	main	Today at 4:46 AM
Fixed Terraform workflow directory	Terraform CI/CD Pipeline #3: Commit 5d376ab pushed by samuelakosaonyejekwe	Failure	main	Today at 4:27 AM

Conclusion

In conclusion, this project successfully demonstrated the implementation of a production-style DevOps workflow using Terraform, GitHub Actions, and Jenkins for Infrastructure-as-Code (IaC) automation on AWS. The project established a fully functional CI/CD pipeline capable of provisioning, validating, deploying, and destroying cloud infrastructure resources in an automated and repeatable manner.

The implementation began with the creation of a modular and enterprise-aligned Terraform repository structure that separated infrastructure configuration, reusable modules, automation scripts, workflow definitions, and documentation into organised directories. This modular design improved maintainability, scalability, and reusability while aligning with real-world DevOps engineering practices.

The project further demonstrated the successful integration of GitHub Actions as a CI/CD automation platform. Through the `terraform.yml` workflow, Terraform operations such as `init`, `validate`, `plan`, `apply`, and `destroy` were automated securely using GitHub repository secrets. The workflow executions validated that infrastructure provisioning and lifecycle management can be performed efficiently through automated pipelines without manual intervention.

An important aspect of the assignment involved troubleshooting and resolving real-world DevOps issues. Errors relating to missing Terraform backend resources, DynamoDB state locking conflicts, and backend S3 bucket configuration were successfully identified, analysed, and corrected. This provided practical exposure to Terraform remote state management, state locking mechanisms, and concurrency protection within CI/CD environments.

The deployment and integration of Jenkins introduced an additional enterprise-grade CI/CD automation layer into the solution. Jenkins was successfully configured with GitHub integration, SSH authentication, AWS credentials management, webhook triggering, and Terraform pipeline execution. The webhook-based automation proved that infrastructure deployment pipelines can be triggered automatically whenever changes are pushed to GitHub repositories.

The project also highlighted the importance of secure DevOps practices. Sensitive credentials were securely managed using GitHub Secrets and Jenkins credential management systems. In addition, Terraform remote backend storage using S3 and DynamoDB demonstrated how infrastructure state can be centrally managed while preventing concurrent modifications and infrastructure corruption.

Overall, the assignment provided practical experience in modern cloud automation, Infrastructure-as-Code implementation, CI/CD pipeline engineering, Terraform state management, Jenkins automation, GitHub integration, and AWS infrastructure provisioning. The completed solution reflects real production DevOps workflows commonly used in enterprise cloud environments and demonstrates a strong understanding of automated infrastructure deployment and continuous delivery principles.