



PROJECT 3

AUTOMATING INFRASTRUCTURE WITH TERRAFORM

SUBMITTED BY

J. SIREESHA

S.SRI NAVEEN

Y. MADEEP

P. RAHUL KUMAR

K. PAWAN KUMAR

Project-3

Infrastructure as Code (IaaC)

Sub Task 1: Choose a tool (AWS CloudFormation or Terraform) for managing infrastructure as code.

- Evaluate the pros and cons of each tool.
- Decide on the preferred tool for your project

AWS CloudFormation:

Pros:

- Native Integration: CloudFormation is AWS's native Infrastructure as Code (IaC) tool, providing seamless integration with other AWS services.
- Declarative Syntax: CloudFormation uses a declarative JSON or YAML syntax, making it easy to understand and write.
- AWS Resource Awareness: CloudFormation is aware of AWS resource dependencies and automatically manages the order of resource creation and deletion.
- AWS-Specific Features: Supports AWS-specific features and services, ensuring compatibility with the latest AWS offerings.

Cons:

- AWS-Specific: Limited support for non-AWS resources. If you're working in a multi-cloud environment, CloudFormation might not be the best choice.
- Learning Curve: It may have a steeper learning curve for beginners due to its AWS-specific nature.

Terraform:

Pros:

- Multi-Cloud Support: Terraform is a multi-cloud IaC tool, supporting various cloud providers, including AWS, Azure, Google Cloud, and more.
- Infrastructure Graph: Terraform creates a graph of resources and parallelizes the creation or modification of non-dependent resources, potentially speeding up the deployment process.
- Community Modules: A large and active community creates and shares Terraform modules, enabling easy reuse of infrastructure code.
- Immutable Infrastructure: Terraform follows an immutable infrastructure paradigm, allowing you to replace or recreate resources rather than modifying them in place.

Cons:

- Learning Curve: Terraform may have a learning curve, especially for those new to infrastructure as code. However, its versatility often compensates for this.
- State Management: Managing Terraform state across a team can be challenging, and proper state management practices are crucial.

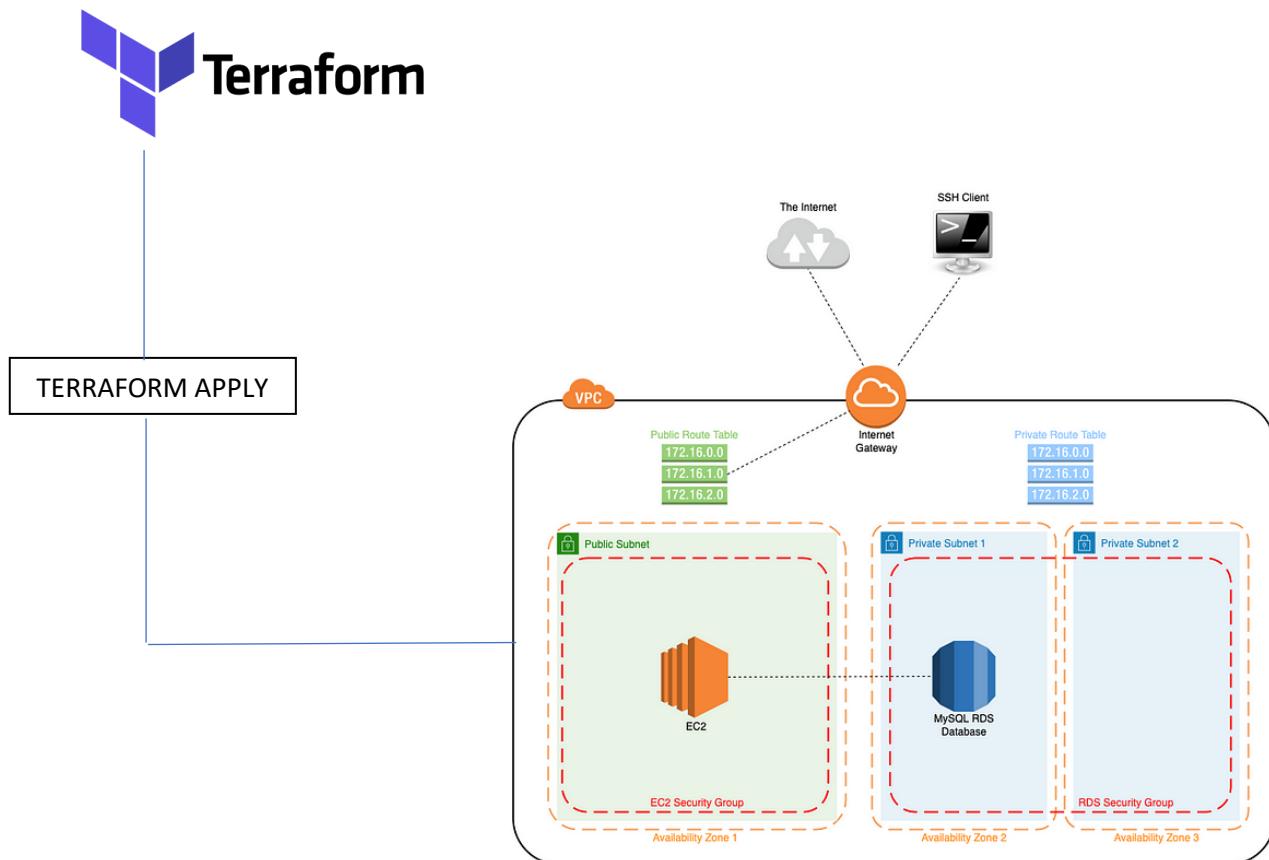
Decision:

After careful consideration, the project has chosen Terraform as its preferred infrastructure as code (IaC) tool. This decision is based on Terraform's versatility, allowing seamless management of infrastructure across multiple cloud platforms. Additionally, the robust community support of Terraform enhances its effectiveness, providing a wealth of shared knowledge and resources. While AWS CloudFormation offers native integration with AWS, the project's requirement for multi-cloud compatibility and a flexible infrastructure approach makes Terraform the optimal choice. This decision aligns with the goal of creating a scalable and adaptable infrastructure management strategy for the project.

Sub Task 2: Create code templates to define your AWS infrastructure:

- Define the network infrastructure (VPC, subnets).
- Specify resources such as EC2 instances, RDS databases, and security groups.
- Create reusable modules or stacks for common resource type

Architecture Diagram:



Project Structure:

```
project-root/
|
|   databases/
|   |   main.tf
|   |   variables.tf
|   |   outputs.tf
|   |
|   |   ...
|
|   network/
|   |   main.tf
|   |   variables.tf
|   |   outputs.tf
|   |
|   |   ...
|
|   security/
|   |   main.tf
|   |   variables.tf
|   |   outputs.tf
|   |
|   |   ...
|
|   instances/
|   |   main.tf
|   |   variables.tf
|   |   outputs.tf
|   |
|   |   ...
|
|   main.tf
```

Creating Networking Infrastructure:

- VPC: We define a Virtual Private Cloud (VPC) to isolate and organize our resources.
- Subnets: Both public and private subnets are created to segregate resources based on their access requirements:

Network module:

- Terraform configurations for defining network infrastructure (e.g., VPC, subnets, security groups).
- Resource definitions using Terraform blocks for networking components.

network.tf:

The screenshot shows the AWS Cloud9 IDE interface with the main.tf file open in the editor. The code defines a VPC with two public and two private subnets.

```
resource "aws_vpc" "main" {
    cidr_block      = var.vpc_cidr_block
    enable_dns_support = true
    enable_dns_hostnames = true

    tags = {
        Name = var.vpc_name
    }
}

resource "aws_subnet" "public" {
    count          = 2
    vpc_id         = aws_vpc.main.id
    cidr_block    = "10.0.${count.index + 1}.0/24"
    availability_zone = element(var.availability_zones, count.index)
    map_public_ip_on_launch = true

    tags = {
        Name = "public-subnet-${count.index + 1}"
    }
}

resource "aws_subnet" "private" {
    count          = 2
    vpc_id         = aws_vpc.main.id
    cidr_block    = "10.0.${count.index + 3}.0/24"
    availability_zone = element(var.availability_zones, count.index)
    map_public_ip_on_launch = false

    tags = {
        Name = "private-subnet-${count.index + 1}"
    }
}
```

The AWS Cloud9 interface includes an Explorer sidebar, a Timeline sidebar, and various AWS service icons. The status bar at the bottom shows the file is 12 lines long, 40 columns wide, in UTF-8 encoding, and is a Plain Text file. The date and time are also displayed.

The screenshot shows the AWS Cloud9 IDE interface with the main.tf file open in the editor. The code defines a VPC with an Internet Gateway and two Route Tables (Public and Private).

```
resource "aws_internet_gateway" "main" {
    depends_on = [aws_vpc.main]
}

resource "aws_route_table" "public" {
    vpc_id = aws_vpc.main.id

    route {
        cidr_block = "0.0.0.0/0"
        gateway_id = aws_internet_gateway.main.id
    }

    tags = {
        Name = var.public_route_table_name
    }
}

resource "aws_route_table" "private" {
    vpc_id = aws_vpc.main.id

    tags = {
        Name = var.private_route_table_name
    }
}
```

The AWS Cloud9 interface includes an Explorer sidebar, a Timeline sidebar, and various AWS service icons. The status bar at the bottom shows the file is 12 lines long, 40 columns wide, in UTF-8 encoding, and is a Plain Text file. The date and time are also displayed.

```
main.tf
66 resource "aws_route_table_association" "public_subnet_association" {
67   count          = 2
68   subnet_id     = aws_subnet.public[count.index].id
69   route_table_id = aws_route_table.public.id
70 }
71
72 resource "aws_route_table_association" "private_subnet_association" {
73   count          = 2
74   subnet_id     = aws_subnet.private[count.index].id
75   route_table_id = aws_route_table.private.id
76 }
77 # Add other security group rules as needed
78
```

Variables.tf:

- Variable declarations specific to the network module.
- Input variables for configurable parameters such as CIDR blocks, subnet configurations, etc.

```
variables.tf
1 variable "vpc_id" {}
2 variable "public_subnet_ids" {}
3 variable "private_subnet_ids" {}
4 variable "ec2_sg_id" {
5   description = "Security Group ID for RDS"
6 }
7
8
```

Outputs.tf

Output variables exposing important information from the created networking resources (e.g., VPC ID, subnet IDs).

```
variables.tf
1 variable "vpc_cidr_block" {}
2 variable "availability_zones" {}
3
4
5 variable "vpc_name" {
6   type    = string
7   default = ""
8 }
9
10 variable "public_subnet_names" {
11   type    = list(string)
12   default = []
13 }
14
15 variable "private_subnet_names" {
16   type    = list(string)
17   default = []
18 }
19
20 variable "igw_name" {
21   type    = string
22   default = ""
23 }
24
25 variable "public_route_table_name" {
26   type    = string
27   default = ""
28 }
29
30 variable "private_route_table_name" {
31   type    = string
32   default = ""
33 }
```

Creating ec2 instance:

- Terraform configurations for provisioning instances (e.g., EC2 instances, ECS clusters). Resource definitions using Terraform blocks for instance-related components.

main.tf

The screenshot shows the AWS Toolkit for Visual Studio Code interface. The left sidebar displays the project structure with files like `terra.yaml`, `main.tf`, `variables.tf`, and `main.tf`. The `main.tf` file is selected and its content is shown in the center editor:

```
tfproject-main > module > instances > main.tf
1 resource "aws_instance" "example" {
2   count           = 2
3   ami             = "ami-0fc5d935ebf8bc3bc"
4   instance_type   = "t2.micro"
5   subnet_id       = element(var.public_subnet_ids, count.index)
6   vpc_security_group_ids = [var.ec2_sg_id]
7   # Use vpc_security_group_ids instead
8
9   tags = [
10     Name = "ExampleInstance-${count.index + 1}"
11   ]
12 }
```

The bottom terminal window shows the command `PS C:\Users\User\Downloads\tfproject-main>`. A sidebar on the right lists Docker-related services: Timeline, Docker Images, Azure Container Registry, Docker Hub, and Suggested Docker Hub Images.

Variables.tf:

- Variable declarations specific to the instances module.
 - Input variables for configurable instance parameters such as instance types, AMIs, etc.

The screenshot shows the AWS Toolkit for Visual Studio Code interface. The left sidebar displays the project structure under 'EXPLORER' with files like 'terra.yaml', 'variables.tf', 'main.tf', and '.gitignore'. The main area shows the contents of 'variables.tf' with variables for VPC ID, public and private subnet IDs, and an EC2 security group ID for RDS. Below this is the 'TERMINAL' tab showing a PowerShell prompt in the directory 'C:\Users\User\Downloads\tfproject-main'. The bottom status bar indicates the file is at line 8, column 1, with 2 spaces, in UTF-8 encoding, and in Plain Text mode. The bottom navigation bar includes icons for File, Edit, Selection, View, Go, Run, and a search bar.

```
variable "vpc_id" {}
variable "public_subnet_ids" {}
variable "private_subnet_ids" {}
variable "ec2_sg_id" {
    description = "Security Group ID for RDS"
}
```

Creating RDS Database:

Terraform configurations for setting up databases (e.g., RDS instances, DynamoDB tables).

Resource definitions using Terraform blocks for database components.

main.tf:

The screenshot shows the AWS Cloud9 IDE interface with the following details:

- File Explorer (Left):** Shows the project structure with files: `terra.yml`, `main.tf`, `outputs.tf`, and `variables.tf`.
- Code Editor (Center):** Displays the `main.tf` Terraform configuration file. The code defines an AWS DB subnet group named "my-new-db-subnet-group-12" with private subnet IDs, and an AWS KMS key named "Example KMS Key". It then creates an AWS DB instance named "rds-db" with MySQL engine version 5.7, allocated storage of 10 GiB, and connects it to the subnet group.
- Bottom Status Bar:** Shows the current profile as "AWS: profile:default", along with other status indicators like network connection and battery level.

```
// databases/main.tf
resource "aws_db_subnet_group" "example" {
    name            = "my-new-db-subnet-group-12"
    description     = "my-new-db-subnet-group-12"
    subnet_ids      = var.private_subnet_ids

    tags = {
        Name = "MyNewDBSubnetGroup"
    }
}

resource "aws_kms_key" "example" {
    description = "Example KMS Key"
}

resource "aws_db_instance" "default" {
    allocated_storage      = 10
    identifier             = "rds-db"
    db_name                = "mydb"
    engine                 = "mysql"
    engine_version         = "5.7"
    instance_class          = "db.t3.micro"
    manage_master_user_password = true
    master_user_secret_kms_key_id = aws_kms_key.example.key_id
    username                = "foo"
    parameter_group_name   = "default.mysql5.7"
    vpc_security_group_ids = [var.rds_sg_id]
    db_subnet_group_name    = aws_db_subnet_group.example.name
}
```

variables.tf:

- Variable declarations specific to the database module. Input variables for configurable parameters such as instance size, database engine, etc.

The screenshot shows the AWS Cloud9 IDE interface. The top bar displays 'File Edit Selection View Go Run ...' and the title 'Untitled (Workspace)'. The left sidebar has sections for 'EXPLORER', 'SEARCH', 'UNTITLED (...', 'tfproject-main', 'environments', 'module', 'aws', 'outputs', 'variables', 'instances', 'main', 'network', 'security', '.gitignore', and 'main'. The main editor area shows the 'outputs.tf' file content:

```
# outputs.tf
output "db_instance_address" {
  value = aws_db_instance.default.endpoint
}
```

The status bar at the bottom shows 'Ln 1, Col 1' through 'Ln 7, Col 1', 'Spaces: 2', 'UTF-8', 'CRLF', 'Plain Text', and system information like '26°C', 'ENG', '19:41', and '13-11-2023'.

Outputs.tf:

- Output variables exposing important information from the created resources (e.g., database endpoint URLs, ARNs).

The screenshot shows the AWS Cloud9 IDE interface. The top bar displays 'File Edit Selection View Go Run ...' and the title 'Untitled (Workspace)'. The left sidebar has sections for 'EXPLORER', 'SEARCH', 'UNTITLED (WORKSPACE)', 'tfproject-main', 'environments', 'module', 'aws', 'outputs', 'variables', 'instances', 'main', 'network', 'security', '.gitignore', and 'main'. The main editor area shows the 'variables.tf' file content:

```
variable "private_subnet_ids" {}
variable "public_subnet_ids" {
  type = list(string)
  default = [] # Add a default value or replace it with your actual default values
}
// databases/variables.tf

variable "vpc_id" [
  description = "The ID of the VPC"
]

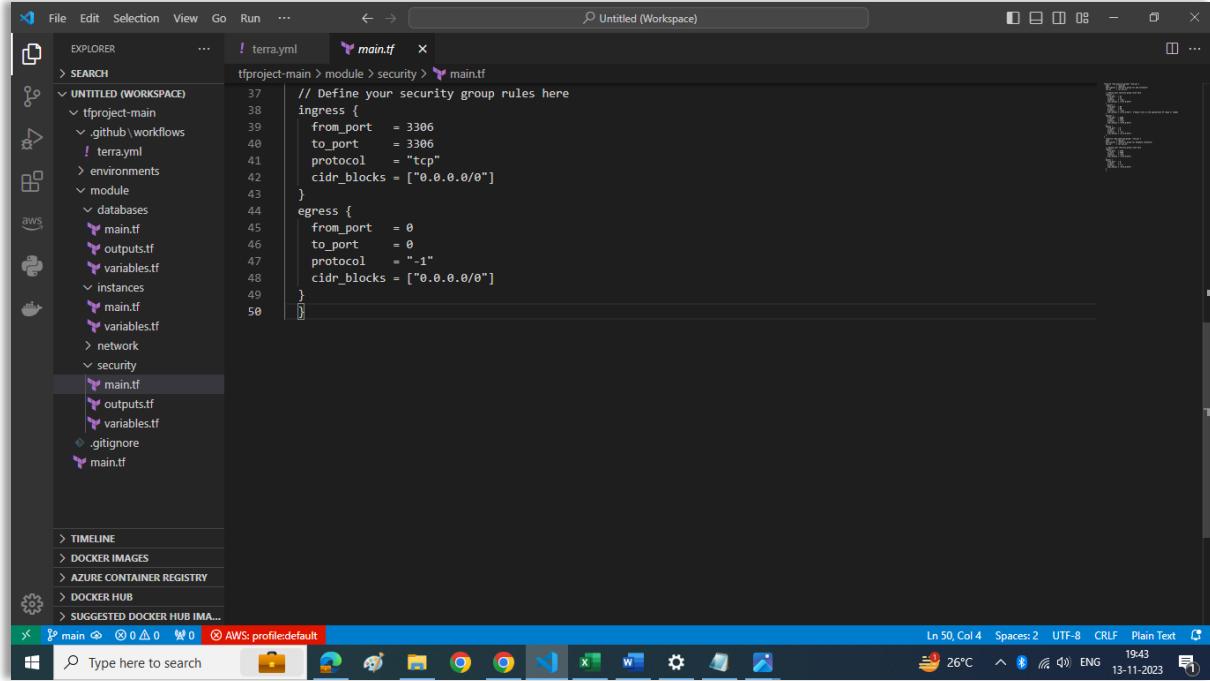
variable "rds_sg_id" [
  description = "Security Group ID for RDS"
]
# ... rest of the configuration
```

The status bar at the bottom shows 'Ln 10, Col 2' through 'Ln 17, Col 2', 'Spaces: 2', 'UTF-8', 'CRLF', 'Plain Text', and system information like '26°C', 'ENG', '19:41', and '13-11-2023'.

Creating Security Group:

- Terraform configurations for security-related resources (e.g., IAM roles, policies, security groups).
- Resource definitions using Terraform blocks for security components.

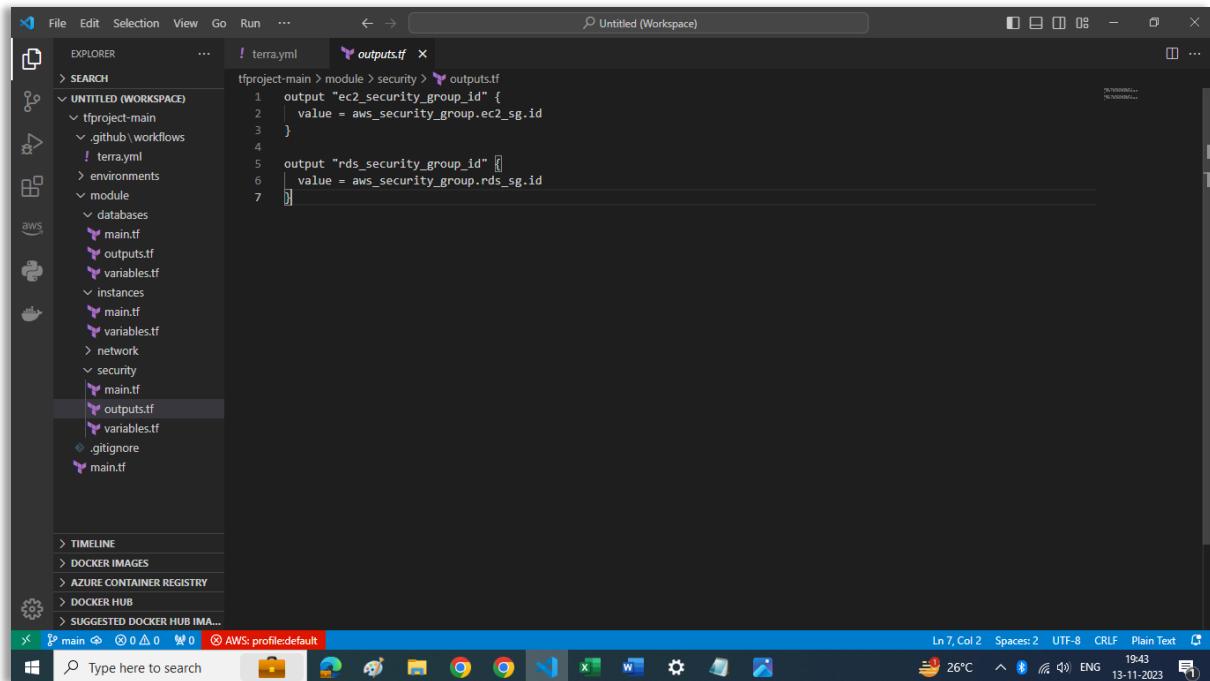
main.tf:



```
// Define your security group rules here
ingress {
  from_port  = 3306
  to_port    = 3306
  protocol   = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}
egress {
  from_port  = 0
  to_port    = 0
  protocol   = "-1"
  cidr_blocks = ["0.0.0.0/0"]
}
```

outputs.tf:

- Output variables exposing important information from the created security resources (e.g., IAM role ARNs, security group IDs).



```
output "ec2_security_group_id" {
  value = aws_security_group.ec2_sg.id
}
output "rds_security_group_id" {
  value = aws_security_group.rds_sg.id
}
```

Root main.tf :

- The main Terraform configuration file for orchestrating and organizing module usage.
- Calls and references to modules, sets up providers, and possibly root-level resources.
- This structure separates various aspects of infrastructure into modular pieces, allowing for easier management and better organization of Terraform code. Each module focuses on a specific aspect of infrastructure and can be developed, tested, and maintained independently. Adjust the content and descriptions within each module based on the specific resources and configurations used in your infrastructure setup.

```
File Edit Selection View Go Run ... ⏪ ⏴ Untitled (Workspace)
EXPLORER ! terrayml main.tf ...
tfproject-main > main.tf
1 module "network" {
2   source      = "./module/network"
3   vpc_cidr_block
4   availability_zones = ["us-east-1a", "us-east-1b"]
5   vpc_name    = "my-vpc1"
6   public_subnet_names = ["public-subnet-1"]
7   private_subnet_names = ["private-subnet-1"]
8   igw_name     = "my-igw"
9   public_route_table_name = "public-route-table"
10  private_route_table_name = "private-route-table"
11 }
12
13 module "instances" {
14   source      = "./module/instances"
15   vpc_id      = module.network.vpc_id
16   public_subnet_ids = module.network.public_subnet_ids
17   private_subnet_ids = module.network.private_subnet_ids
18   ec2_sg_id   = module.security.ec2_security_group_id # Reference the ec2_sg_id from the network module
19 }
20
21 module "databases" {
22   source      = "./module/databases"
23   vpc_id      = module.network.vpc_id
24   private_subnet_ids = module.network.private_subnet_ids
25   rds_sg_id   = module.security.rds_security_group_id
26   # Add other necessary arguments
27 }
28 module "security" {
29   source      = "./module/security"
30   vpc_id      = module.network.vpc_id
31 }
```

Sub Task 3: Testing and Validation

- Test the templates to ensure they accurately represent your desired infrastructure.
- Use validation tools specific to your chosen IaaC tool.
- Simulate stack creations and updates in a non-production environment.

Infrastructure Testing:

In this phase, we focus on validating and testing the infrastructure templates to ensure they accurately represent the desired state of our environment.

Validation Tools:

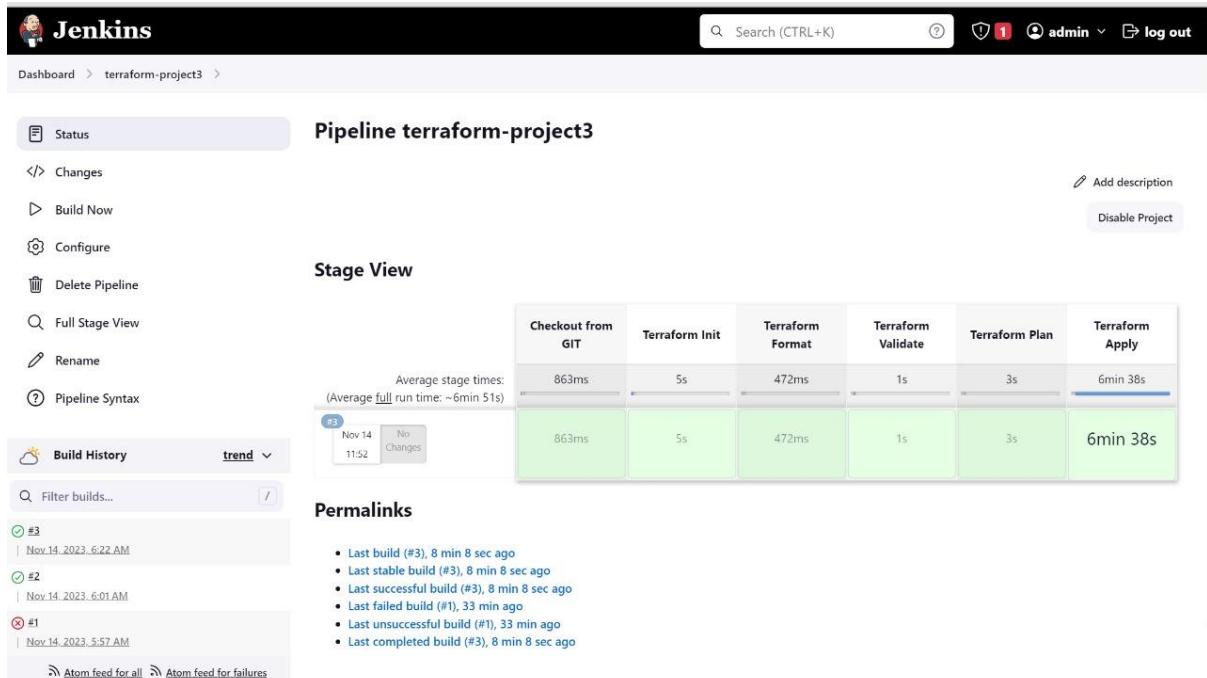
- We leverage specific validation tools provided by Terraform to ensure the correctness and consistency of our Infrastructure as Code (IaC).
- Run `terraform validate` and `tflint` to ensure your Terraform configuration is syntactically correct and follows best practices.

Some of the key validation steps include:

- `terraform fmt -recursive`
- `terraform validate`

Simulate Stack Changes:

- Run `terraform plan` to see the changes that Terraform will apply.
- Review the plan to verify that it aligns with your expectations.
- Apply Changes in a Non-Production Environment:
- If the plan looks good, apply the changes in a non-production environment
- `Terraform apply`



Sub Task 4: Deploying and Managing Infrastructure

- Deploy and manage your infrastructure using IaaS.
- Create and manage AWS CloudFormation stacks or Terraform workspaces.
- Automate the provisioning of resources for your application.
- Implement version control for IaaS code to track changes and update

Need of Terraform Workspaces:

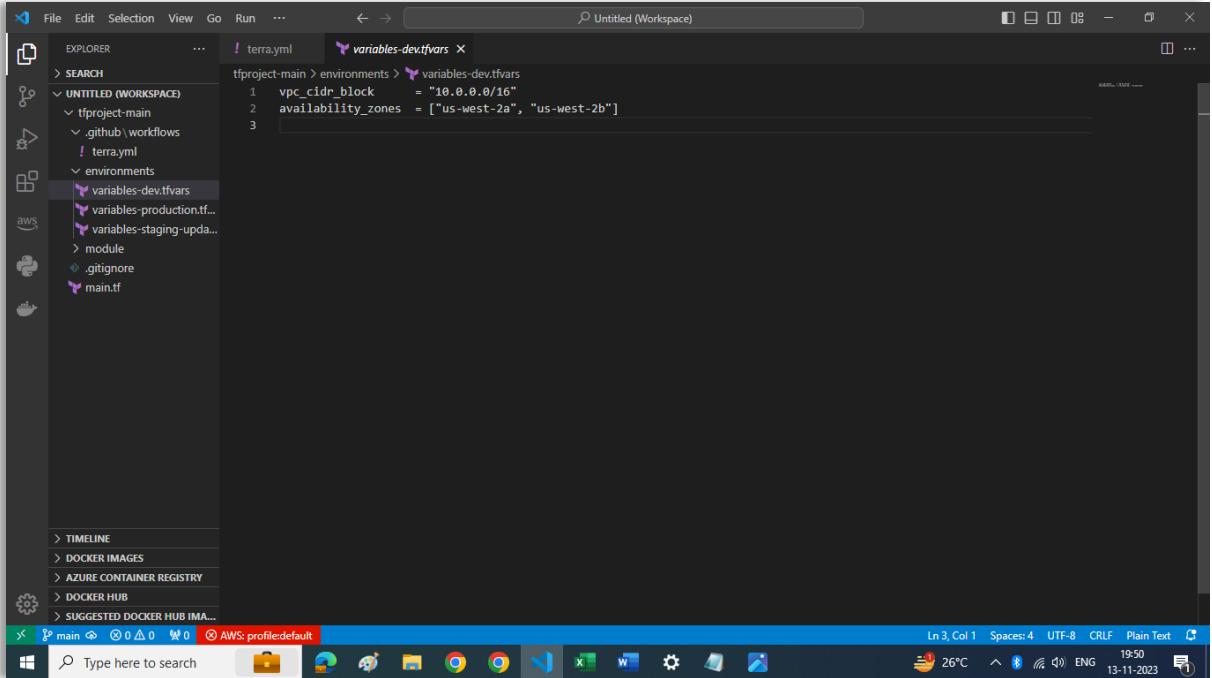
- a workspace is a way to manage multiple environments or configurations for the same set of infrastructure resources.
- Workspaces enable you to deploy and manage multiple instances of your infrastructure with different configurations or variables. Each workspace maintains its own state file, allowing you to isolate and manage the state for different environments.

Steps to create workspaces:

- `terraform workspace new (workspace name)`

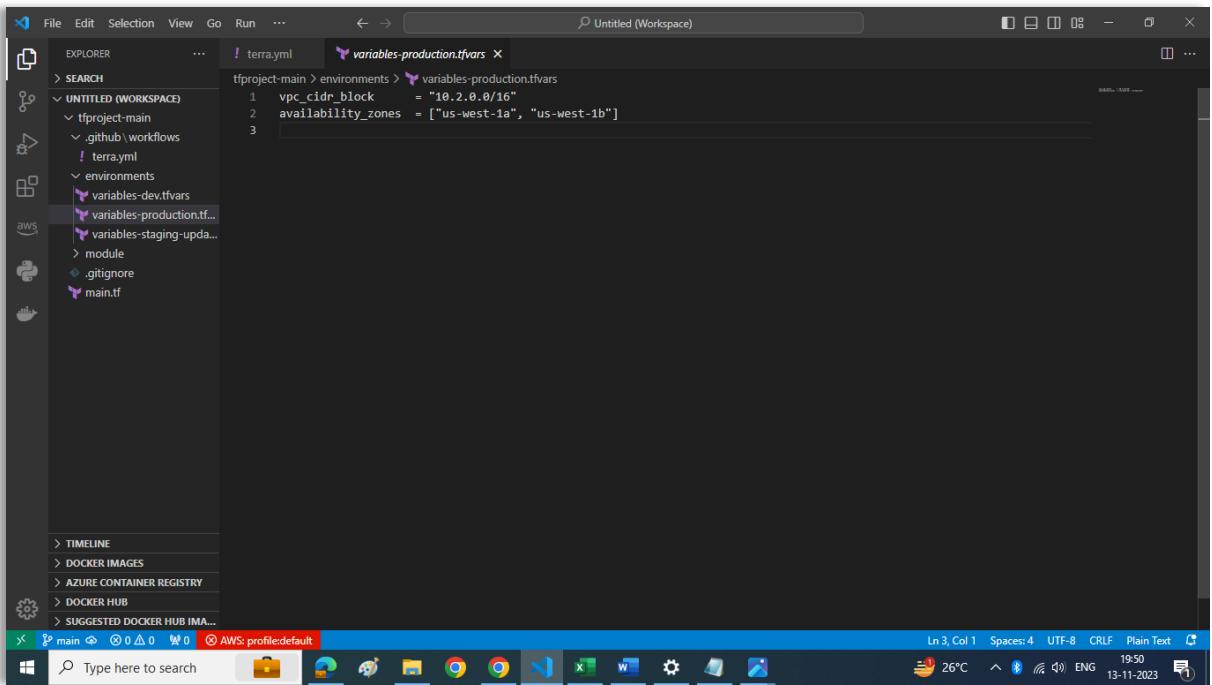
- terraform workspace list
- terraform workspace select dev(workspace name)
- Manage State:
- Each workspace has its own state file (.tfstate), allowing you to keep the state of different environments separate.
- This prevents conflicts and ensures that changes made in one workspace don't affect the resources in another.

Variables-dev.tfvars:



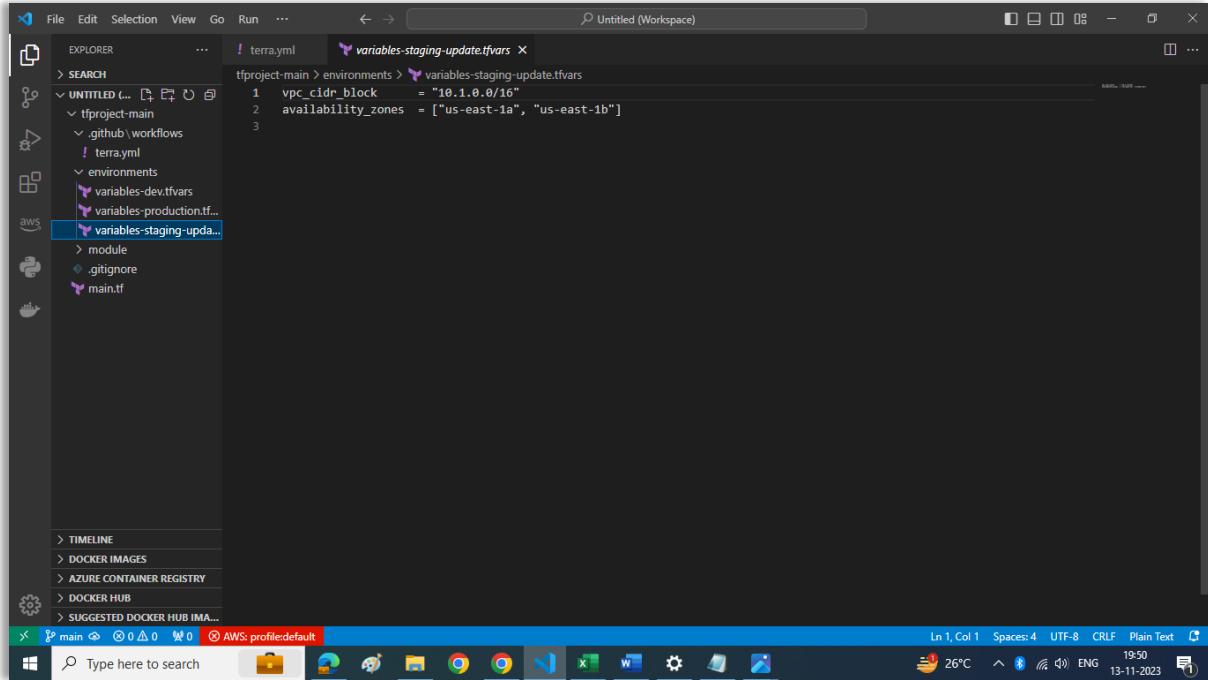
```
vpc_cidr_block = "10.0.0.0/16"
availability_zones = ["us-west-2a", "us-west-2b"]
```

Variables-staging-update.tfvars:



```
vpc_cidr_block = "10.2.0.0/16"
availability_zones = ["us-west-1a", "us-west-1b"]
```

Variables-production.tfvars:



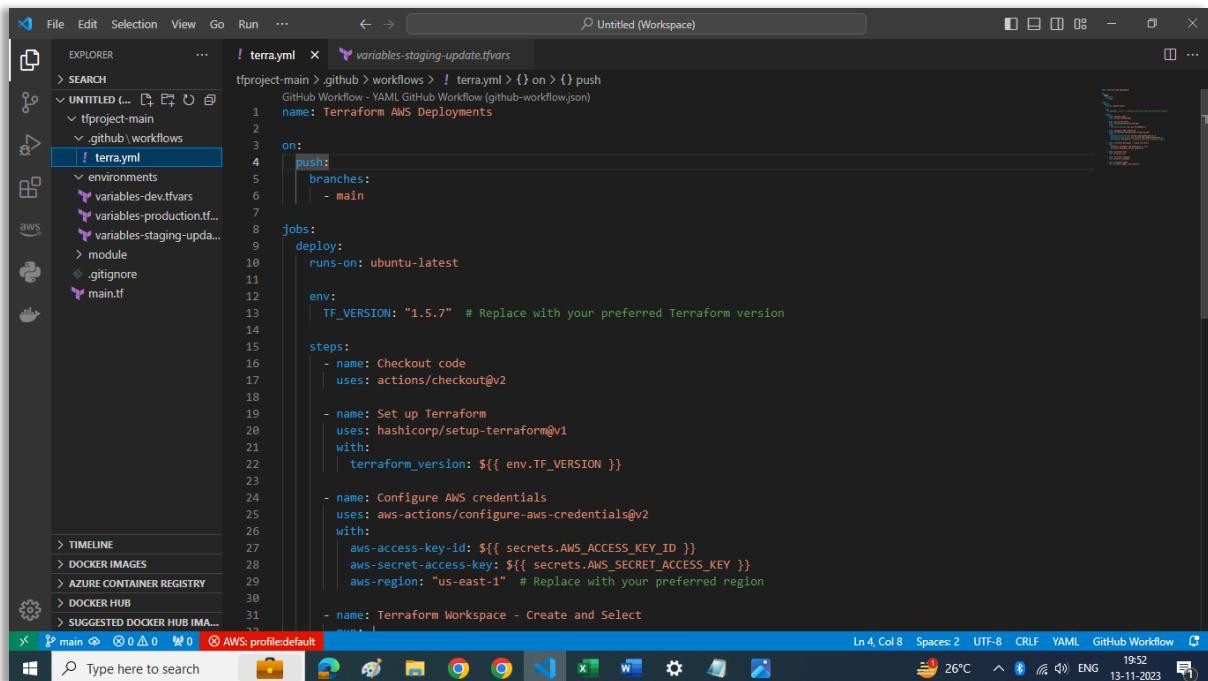
The screenshot shows the VS Code interface with the 'variables-production.tfvars' file open in the editor. The code defines two variables:

```
vpc_cidr_block = "10.1.0.0/16"
availability_zones = ["us-east-1a", "us-east-1b"]
```

Automating the infrastructure:

- Using GitHub Actions to automate the process of managing Terraform workspaces and applying changes for different environments.
- The yaml file for automating the infrastructure as follows

Github-workflow.yml:



The screenshot shows the VS Code interface with the 'Github-workflow.yml' file open in the editor. The workflow is defined as follows:

```
name: Terraform AWS Deployments
on:
  push:
    branches:
      - main
jobs:
  deploy:
    runs-on: ubuntu-latest
    env:
      TF_VERSION: "1.5.7" # Replace with your preferred Terraform version
    steps:
      - name: Checkout code
        uses: actions/checkout@v2
      - name: Set up Terraform
        uses: hashicorp/setup-terraform@v1
        with:
          terraform_version: ${{ env.TF_VERSION }}
      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: "us-east-1" # Replace with your preferred region
      - name: Terraform Workspace - Create and Select
        run: terraform workspace select -name main
```

```

name: Terraform Workspace - Create and Select
run: |
  terraform workspace new production || true
  terraform workspace select production

- name: Terraform Init
  run: terraform init

- name: Terraform Validate
  run: terraform validate

- name: Terraform Apply
  run: terraform apply --auto-approve

```

Implementing Version Control for IaC Code:

- Version control is a critical aspect of managing Infrastructure as Code (IaC) to track changes, collaborate effectively, and maintain a history of modifications. Here, we detail how we implement version control for our IaC codebase
- We use a distributed version control system (VCS) to manage our IaC codebase. Git is a popular choice due to its flexibility, distributed nature, and robust branching and merging capabilities.
- We use Git tags to mark specific releases or versions of our IaC code. Tagging allows us to easily reference and roll back to specific points in our code history.
- Clear and concise commit messages are essential for understanding the purpose of changes. Each commit should convey meaningful information about the modifications made.

Step 1: Choose a Version Control System

Choose a VCS that suits your needs. Git is widely used in the IaC community due to its popularity, distributed nature, and strong branching capabilities.

Step 2: Creating Repository

- Create a New Repository:
- Clone the Repository: `git clone <repository_url>`

- Navigate to Repository: cd <repository_directory>

Step 3: Initialize Git

- git init
- git add .
- git commit -m "Initial commit"

Step 4: Create Branches

- git branch -M main
- git checkout -b feature-branch

Step 5: Commit Changes

- git add .
- git commit -m "Meaningful commit message"
- Step 7: Push Changes
- git push origin mainStep

Step 6: Tagging Releases

Tag specific releases or versions with meaningful tags:

- git tag -a v1.0 -m "Release 1.0"
- git push origin v1.0

Step 7: GitHub Actions Workflow Execution:

- The workflow executes on a GitHub-hosted runner or a self-hosted runner.
- The workflow steps include checking out the code, configuring the environment, and running specified tasks, such as deploying infrastructure.

Step 8: Terraform Apply:

- One of the workflow steps involves running terraform apply.
- Terraform, based on the changes in your scripts, communicates with AWS APIs to create, update, or delete resources.

Step 9: Infrastructure Provisioning:

- Terraform communicates with the AWS provider, and AWS resources (VPCs, subnets, EC2 instances, etc.) are provisioned or updated based on your Terraform configurations. The created resources are,

VPC:

The screenshot shows the AWS VPC Details page for a VPC with ID `vpc-065050529f0f47b95`. The main pane displays the following details:

VPC ID	State	DNS hostnames	DNS resolution
<code>vpc-065050529f0f47b95</code>	Available	Enabled	Enabled
Tenancy	DHCP option set	Main route table	Main network ACL
Default	<code>dopt-0b86b43057e09cbce</code>	<code>rtb-0cd3544de8478f427</code>	<code>acl-06a2f1c805d1d0fe8</code>
Default VPC	IPv4 CIDR	IPv6 pool	IPv6 CIDR (Network border group)
No	10.0.0.0/16	-	-
Network Address Usage metrics	Route 53 Resolver DNS Firewall rule groups	Owner ID	
Disabled	-	<code>57188835380</code>	-

Below the details, there are tabs for **Resource map**, **CIDRs**, **Flow logs**, **Tags**, and **Integrations**. The **Resource map** tab is selected.

Subnets:

The screenshot shows the AWS Subnets page for a VPC. The main pane displays a list of subnets:

Name	Subnet ID	State	VPC	IPv4 CIDR
private-subnet-2	<code>subnet-0bb85832418306425</code>	Available	<code>vpc-065050529f0f47b95 my-vpc1</code>	10.0.4.0/24
private-subnet-1	<code>subnet-01ab57ed012054813</code>	Available	<code>vpc-065050529f0f47b95 my-vpc1</code>	10.0.3.0/24
public-subnet-2	<code>subnet-062ddc4747c726bf0</code>	Available	<code>vpc-065050529f0f47b95 my-vpc1</code>	10.0.2.0/24
public-subnet-1	<code>subnet-08c3c092b134f26a3</code>	Available	<code>vpc-065050529f0f47b95 my-vpc1</code>	10.0.1.0/24

Below the table, there is a section titled "Select a subnet" with three small icons.

Internet Gateway:

The screenshot shows the AWS VPC Internet Gateways console. The left sidebar is collapsed, and the main area displays the details of an Internet gateway named 'igw-060499e50283bbb1a' with the tag 'Name: my-igw'. The 'Details' tab is selected, showing the Internet gateway ID, state (Attached), VPC ID, and owner information.

Internet gateway ID	State	VPC ID	Owner
igw-060499e50283bbb1a	Attached	vpc-065050529f0f47b95 my-igw	571888835380

The 'Tags' section shows a single tag: Name = my-igw.

Route Tables:

The screenshot shows the AWS VPC Route Tables console. The left sidebar is collapsed, and the main area displays a list of route tables. There are three route tables listed: 'public-route-table' and 'private-route-table' (both associated with 2 subnets) and one unnamed route table (not associated with any subnets). The 'public-route-table' is marked as the 'Main' route table.

Name	Route table ID	Explicit subnet associations	Edge associations	Main
-	rtb-0cd3544de8478f427	-	-	No
public-route-table	rtb-0072a0530da567986	2 subnets	-	No
private-route-table	rtb-06b7b05cc970dda3f	2 subnets	-	No

RDS:

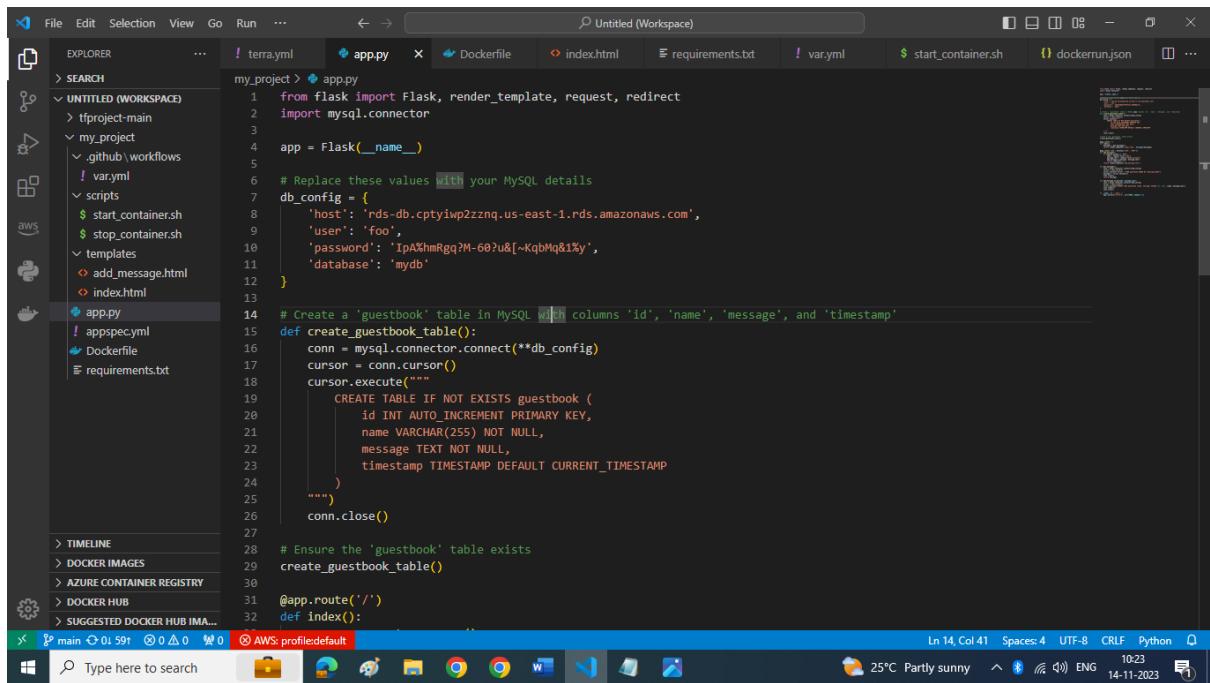
The screenshot shows the AWS RDS Databases console. At the top, there is a blue banner with an info icon and the text: "Consider creating a Blue/Green Deployment to minimize downtime during upgrades". Below the banner, the main interface has a title bar "Databases (1)" with various buttons: "Group resources", "Modify", "Actions", "Restore from S3", and "Create database". There is also a search bar "Filter by databases". The main table lists one database entry: "rds-db" (DB identifier), "Available" (Status), "Instance" (Role), "MySQL Community" (Engine), "us-east-1b" (Region & AZ), and "db.t3.micro" (Size). The "Actions" column shows "4 Actions".

The screenshot shows the "Connectivity & security" tab for the "rds-db" database. The left sidebar shows the "Amazon RDS" navigation menu with "Databases" selected. The main content area displays connectivity details in three columns: "Endpoint & port" (Endpoint: rds-db.cptyiwp2zznq.us-east-1.rds.amazonaws.com, Port: 3306), "Networking" (Availability Zone: us-east-1a, VPC: my-vpc1 (vpc-065050529f0f47b95), Subnet group: my-new-db-subnet-group-12, Subnets: subnet-0bb85832418306425, subnet-01ab57ed012054813), and "Security" (VPC security groups: db_sg (sg-0111851f81d4b1f2f) - Active, Publicly accessible: No, Certificate authority: rds-ca-2019, Certificate authority date: August 22, 2024, 22:38 (UTC+05:30), DB instance certificate).

Automated Deployment with GitHub Actions and AWS Services

- this process involves the following the step-by-step process of setting up an automated deployment pipeline for your Flask application using GitHub Actions and AWS services.
- The workflow includes building a Docker image, storing it in Amazon ECR, and deploying it to Amazon EC2 instances with AWS CodeDeploy.
- Additionally, we are connected the our application with Amazon RDS for a fully managed relational database.

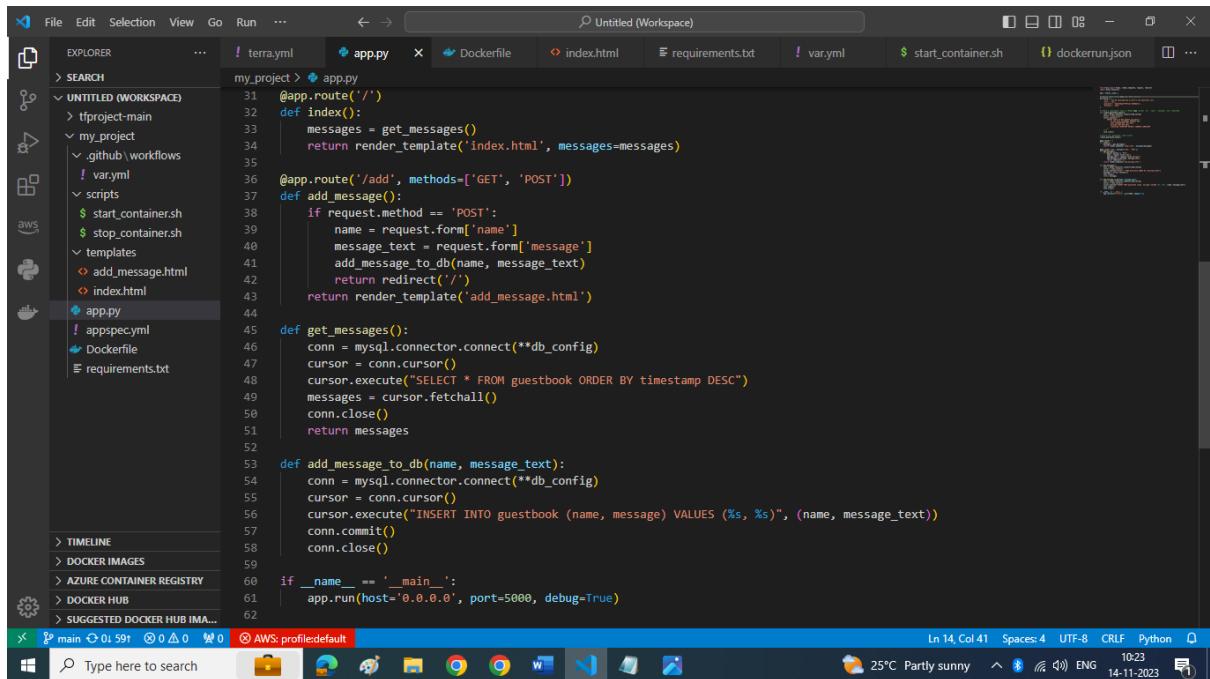
Application Source Code:



The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "UNTITLED (WORKSPACE)".
- Code Editor:** Displays the content of `app.py`. The code is a Flask application that connects to a MySQL database and creates a "guestbook" table if it doesn't exist. It includes routes for displaying messages and adding new messages.
- Bottom Status Bar:** Shows the file path as "main", line 14, column 41, and other status indicators like "AWS profile:default".

```
my_project > app.py
1  from flask import Flask, render_template, request, redirect
2  import mysql.connector
3
4  app = Flask(__name__)
5
6  # Replace these values with your MySQL details
7  db_config = {
8      'host': 'rds-db.cptyiwp2zznq.us-east-1.rds.amazonaws.com',
9      'user': 'foo',
10     'password': '1pA%hmRgq?M-60?u&[~KqbMq&1%y',
11     'database': 'mydb'
12 }
13
14 # Create a 'guestbook' table in MySQL with columns 'id', 'name', 'message', and 'timestamp'
15 def create_guestbook_table():
16     conn = mysql.connector.connect(**db_config)
17     cursor = conn.cursor()
18     cursor.execute("""
19         CREATE TABLE IF NOT EXISTS guestbook (
20             id INT AUTO_INCREMENT PRIMARY KEY,
21             name VARCHAR(255) NOT NULL,
22             message TEXT NOT NULL,
23             timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
24         )
25     """)
26     conn.close()
27
28 # Ensure the 'guestbook' table exists
29 create_guestbook_table()
30
31 @app.route('/')
32 def index():
33     messages = get_messages()
34     return render_template('index.html', messages=messages)
35
36 @app.route('/add', methods=['GET', 'POST'])
37 def add_message():
38     if request.method == 'POST':
39         name = request.form['name']
40         message_text = request.form['message']
41         add_message_to_db(name, message_text)
42         return redirect('/')
43     return render_template('add_message.html')
44
45 def get_messages():
46     conn = mysql.connector.connect(**db_config)
47     cursor = conn.cursor()
48     cursor.execute("SELECT * FROM guestbook ORDER BY timestamp DESC")
49     messages = cursor.fetchall()
50     conn.close()
51     return messages
52
53 def add_message_to_db(name, message_text):
54     conn = mysql.connector.connect(**db_config)
55     cursor = conn.cursor()
56     cursor.execute("INSERT INTO guestbook (name, message) VALUES (%s, %s)", (name, message_text))
57     conn.commit()
58     conn.close()
59
60 if __name__ == '__main__':
61     app.run(host='0.0.0.0', port=5000, debug=True)
```



The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "UNTITLED (WORKSPACE)".
- Code Editor:** Displays the content of `app.py`. This version of the code is identical to the one in the first screenshot, showing the same Flask application logic for a guestbook.
- Bottom Status Bar:** Shows the file path as "main", line 14, column 41, and other status indicators like "AWS profile:default".

```
my_project > app.py
1  from flask import Flask, render_template, request, redirect
2  import mysql.connector
3
4  app = Flask(__name__)
5
6  # Replace these values with your MySQL details
7  db_config = {
8      'host': 'rds-db.cptyiwp2zznq.us-east-1.rds.amazonaws.com',
9      'user': 'foo',
10     'password': '1pA%hmRgq?M-60?u&[~KqbMq&1%y',
11     'database': 'mydb'
12 }
13
14 # Create a 'guestbook' table in MySQL with columns 'id', 'name', 'message', and 'timestamp'
15 def create_guestbook_table():
16     conn = mysql.connector.connect(**db_config)
17     cursor = conn.cursor()
18     cursor.execute("""
19         CREATE TABLE IF NOT EXISTS guestbook (
20             id INT AUTO_INCREMENT PRIMARY KEY,
21             name VARCHAR(255) NOT NULL,
22             message TEXT NOT NULL,
23             timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
24         )
25     """)
26     conn.close()
27
28 # Ensure the 'guestbook' table exists
29 create_guestbook_table()
30
31 @app.route('/')
32 def index():
33     messages = get_messages()
34     return render_template('index.html', messages=messages)
35
36 @app.route('/add', methods=['GET', 'POST'])
37 def add_message():
38     if request.method == 'POST':
39         name = request.form['name']
40         message_text = request.form['message']
41         add_message_to_db(name, message_text)
42         return redirect('/')
43     return render_template('add_message.html')
44
45 def get_messages():
46     conn = mysql.connector.connect(**db_config)
47     cursor = conn.cursor()
48     cursor.execute("SELECT * FROM guestbook ORDER BY timestamp DESC")
49     messages = cursor.fetchall()
50     conn.close()
51     return messages
52
53 def add_message_to_db(name, message_text):
54     conn = mysql.connector.connect(**db_config)
55     cursor = conn.cursor()
56     cursor.execute("INSERT INTO guestbook (name, message) VALUES (%s, %s)", (name, message_text))
57     conn.commit()
58     conn.close()
59
60 if __name__ == '__main__':
61     app.run(host='0.0.0.0', port=5000, debug=True)
```

Index.html:

The screenshot shows the VS Code interface with the 'index.html' file open in the editor. The code is a Django template for a guestbook application. It includes a title, a heading, a list of messages, and a link to add a new message.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Guestbook</title>
</head>
<body>
    <h1>Guestbook</h1>
    <ul>
        {% for message in messages %}
            <li>
                <strong>{{ message[1] }}</strong>: {{ message[2] }}
            </li>
        {% endfor %}
    </ul>
    <a href="/add">Add Message</a>
</body>
</html>
```

Add-Massage.html

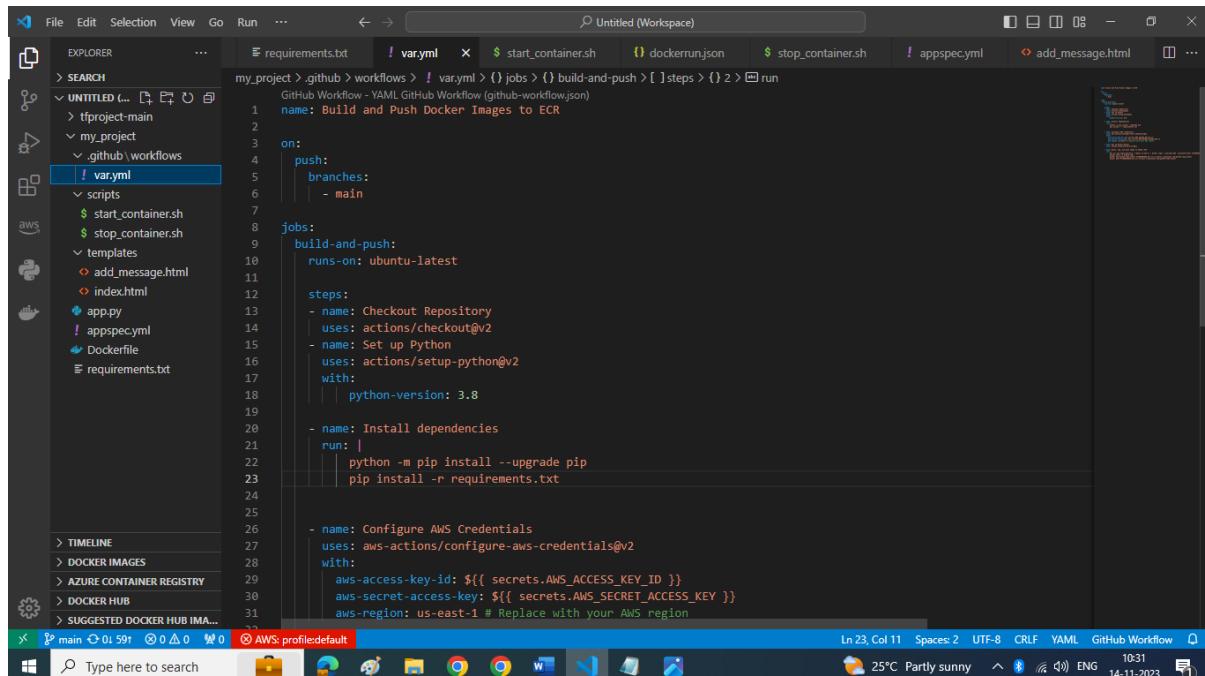
The screenshot shows the VS Code interface with the 'add_message.html' file open in the editor. The code is a Django template for adding a new message. It features a form with fields for name and message, and a submit button.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Add Message</title>
</head>
<body>
    <h2>Add Message:</h2>
    <form method="post" action="/add">
        <label for="name">Your Name:</label>
        <input type="text" id="name" name="name" required><br>
        <label for="message">Message:</label>
        <textarea id="message" name="message" rows="4" required></textarea><br>
        <button type="submit">Add Message</button>
    </form>
    <a href="/">Back to Guestbook</a>
</body>
</html>
```

GitHub Actions Workflow:

Using github action workflow it automatically build docker image from docker file and pushed to ecr whenever the code changes, github triggers the workflow

- Checkout Code
- Build Docker Image
- Deploy to ECR



A screenshot of the Visual Studio Code interface showing a GitHub Actions workflow configuration. The left sidebar shows a project structure with files like requirements.txt, var.yml, start_container.sh, dockerrun.json, stop_container.sh, appspec.yml, add_message.html, Dockerfile, and index.html. The main editor tab is open to a GitHub Workflow - YAML file named var.yml. The code defines a workflow named 'Build and Push Docker Images to ECR' with a single job named 'main'. The job runs on an Ubuntu-latest container and contains several steps: 1) Checkout Repository using actions/checkout@v2, 2) Set up Python using actions/setup-python@v2 with python-version: 3.8, 3) Install dependencies by running 'python -m pip install --upgrade pip' and 'pip install -r requirements.txt', 4) Configure AWS Credentials using aws-actions/configure-aws-credentials@v2 with AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables, and 5) Push the Docker image to the AWS ECR repository using aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 571888835380.dkr.ecr.us-east-1.amazonaws.com/python-repo:latest, docker build -t python-repo ., docker tag python-repo:latest 571888835380.dkr.ecr.us-east-1.amazonaws.com/python-repo:latest, and docker push 571888835380.dkr.ecr.us-east-1.amazonaws.com/python-repo:latest.

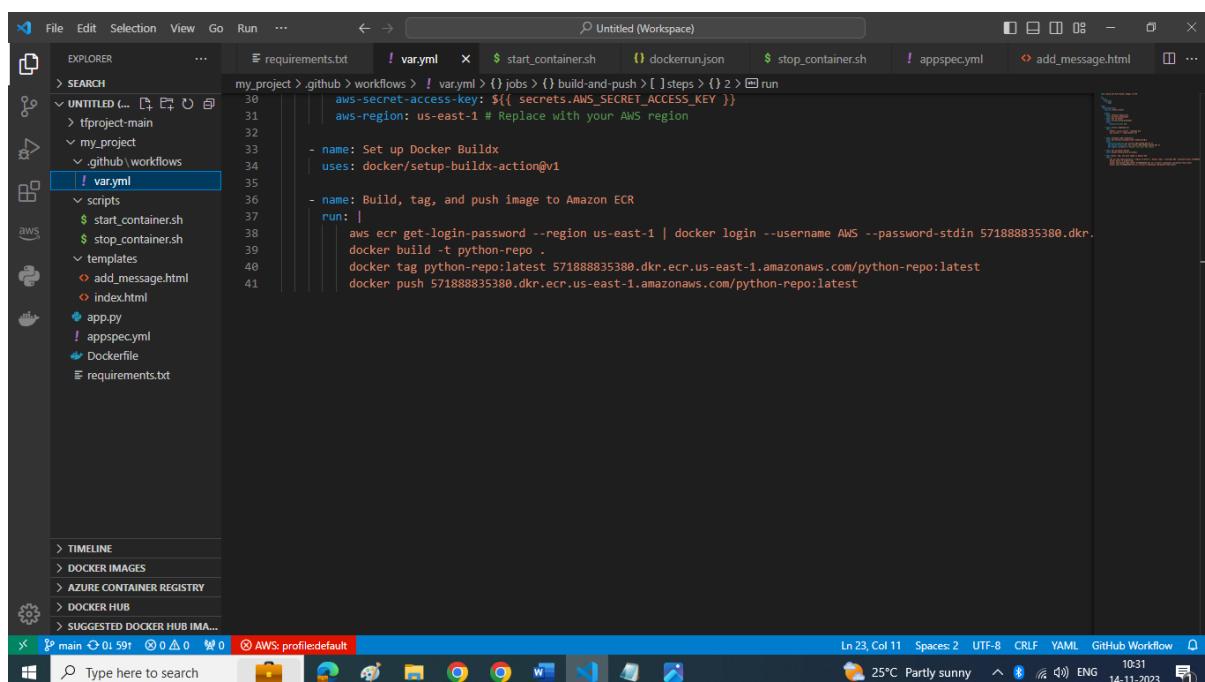
```
name: Build and Push Docker Images to ECR

on:
  push:
    branches:
      - main

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

    steps:
      - name: Checkout Repository
        uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: 3.8
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Configure AWS Credentials
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1 # Replace with your AWS region
```



A screenshot of the Visual Studio Code interface showing a GitHub Actions workflow configuration. The left sidebar shows a project structure with files like requirements.txt, var.yml, start_container.sh, dockerrun.json, stop_container.sh, appspec.yml, add_message.html, Dockerfile, and index.html. The main editor tab is open to a GitHub Workflow - YAML file named var.yml. The code is identical to the one in the previous screenshot, but the last step has been modified to use docker buildx instead of docker. The modified step is: 'aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 571888835380.dkr.ecr.us-east-1.amazonaws.com/python-repo:latest, docker buildx build --tag 571888835380.dkr.ecr.us-east-1.amazonaws.com/python-repo:latest --platform linux/amd64 .' The rest of the workflow remains the same, including the configuration of AWS credentials and the final push step.

```
name: Build and Push Docker Images to ECR

on:
  push:
    branches:
      - main

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

    steps:
      - name: Checkout Repository
        uses: actions/checkout@v2
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1
      - name: Build, tag, and push image to Amazon ECR
        run: |
          aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 571888835380.dkr.ecr.us-east-1.amazonaws.com/python-repo:latest
          docker buildx build --tag 571888835380.dkr.ecr.us-east-1.amazonaws.com/python-repo:latest --platform linux/amd64 .
```

AWS Configuration:

Creating code deploy application: Certainly! Here are the steps to create an AWS Code Deploy application and deployment group:

Step 1: Set Up AWS Code Deploy Application

- Open the AWS Management Console. In the "Find Services" search bar, type "Code Deploy" and select it.
- Create a New Application:
- Click on the "Create application" button.
- Enter a unique name for your application.
- Choose the compute platform that matches your application.

The screenshot shows the AWS Management Console interface. The top navigation bar includes links for AWS services like CloudShell, Feedback, and various AWS services (Flask, EC2, etc.). The main search bar is set to 'Search' and has a placeholder '[Alt+S]'. The breadcrumb navigation path is 'Developer Tools > CodeDeploy > Applications > Create application'. The main content area is titled 'Create application' and contains a form for 'Application configuration'. The 'Application name' field is filled with 'guest_notebook'. The 'Compute platform' dropdown is set to 'EC2/on-premises'. A 'Tags' section with an 'Add tag' button is present. At the bottom right of the configuration form are 'Cancel' and 'Create application' buttons. Below this, a success message box displays the text: 'Application created' followed by 'In order to create a new deployment, you must first create a deployment group.' The message also includes a 'Create a notification rule for this application' link. To the left of the main content is a sidebar titled 'CodeDeploy' which lists 'Source • CodeCommit', 'Artifacts • CodeArtifact', 'Build • CodeBuild', 'Deploy • CodeDeploy' (with 'Getting started', 'Deployments', and 'Applications' sub-options), 'Application' (with 'Settings', 'Deployment configurations', 'On-premises instances'), 'Pipeline • CodePipeline', and 'Settings'. The 'Deployment groups' tab is selected in the main content area, showing a table header for 'Name', 'Status', 'Last attempted de...', 'Last successful de...', and 'Trigger count'. The bottom of the screen shows a taskbar with icons for CloudShell, Feedback, and various applications, along with system status information including temperature (28°C Haze), battery level (ENG 14-11-2023), and network connectivity.

Step2: Creating Deployment Group

- After creating the application, click on the "Create deployment group" button.
- Enter a name for your deployment group.
- Choose the deployment type (In-Place or Blue/Green).
- Configure the deployment settings according to your requirements.
- Configure Deployment Settings:
- Choose the service role for AWS Code Deploy to use. If you don't have one, create a new service role.
- Configure the deployment settings like deployment configuration, load balancer, and alarms if needed.
- Click "Create deployment group" to save your settings

The screenshot shows the 'Deployment group name' step of the AWS CodeDeploy 'Create deployment group' wizard. The 'Deployment group name' input field contains 'demogroup'. Below it, the 'Service role' section shows a search bar with 'am:aws:iam::57188835380:role/codedeployrole' selected.

The screenshot shows the 'Deployment type' step of the wizard. The 'In place' radio button is selected, with a tooltip explaining it updates instances in the deployment group with the latest application revision. The 'Blue/green' radio button is also shown.

The screenshot shows the 'Deployment instances' step of the wizard. It lists 'Amazon EC2 instances' as selected, with a note that 1 unique matched instance was found. It also shows sections for 'Tag group 1' (with a key 'Name' and value 'ExampleInstance-1') and 'Matching instances' (1 unique matched instance). Other options like 'On-premises instances' are shown as unselected.

The screenshot shows the AWS CodeDeploy console interface. A deployment group named "demogroup" has been created. The deployment group details table includes:

Deployment group name	Application name	Compute platform
demogroup	guest_notebook	EC2/on-premises
Deployment type	Service role ARN	Deployment configuration
In place	arn:aws:iam::571888835380:role/codedeployrole	CodeDeployDefault.AllAtOnce
Rollback enabled	Agent update scheduler	
False	Learn to schedule update in AWS Systems Manager	

Step 3: IAM Role for AWS Code Deploy

- Create an IAM Role:
- Open the IAM Console. In the left navigation pane, choose "Roles." Click on "Create role."
- Select "AWS service," then choose "Code Deploy."
- Attach Permissions:

- Attach policies that grant the necessary permissions for your deployment. At a minimum, you might attach the AWSCodeDeployRole policy.
- Review and Create Review your settings and give your role a meaningful name.
- Click "Create role."

The screenshot shows the AWS IAM Roles page. A role named 'codedeployrole' is selected. The 'Permissions' tab is active, showing one managed policy attached: 'AWSCodeDeployRole'. The policy details are as follows:

Policy name	Type	Attached entities
AWSCodeDeployRole	AWS managed	2

Step 4: Application Deployment Configuration

Configure Deployment Source:

Ensure that your GitHub repository is configured as the deployment source. This can be done during the creation of the deployment group or by updating the deployment group settings.

Create a Deployment:

- In the AWS Code Deploy console, select your application.

- Choose the deployment group you created.
- Click on "Create deployment."
- Specify the revision (e.g., the GitHub commit) to deploy.
- Click "Create deployment."

Monitor Deployment:

- Monitor the deployment progress in the AWS Code Deploy console.
- View deployment details, logs, and any errors that may occur.:

The screenshot shows the AWS Code Deploy console interface. On the left, there's a sidebar with 'Developer Tools' and 'CodeDeploy' selected. Under 'CodeDeploy', there are sections for Source, Artifacts, Build, Deploy, Pipeline, and Settings. The main area is titled 'd-HLNMY4AS2' and shows the 'Deployment status' section. It indicates that an application is being installed on instances, with 1 of 1 instances updated and a success rate of 100%. Below this is the 'Deployment details' section, which provides specific information about the deployment: Application (siri), Deployment ID (d-HLNMY4AS2), Deployment configuration (CodeDeployDefault.AllAtOnce), Deployment group (siri-1), and Status (Succeeded, Initiated by User action). At the bottom of the page, there are links for Go to resources, CloudShell, Feedback, and a search bar. The footer includes copyright information, privacy terms, and cookie preferences, along with weather and system status icons.

Step 5: Set Up AWS Code Pipeline

Navigate to AWS Code Pipeline:

- Open the AWS Management Console.
- In the "Find Services" search bar, type "Code Pipeline" and select it.
- Create a New Pipeline:
- Click on the "Create pipeline" button.
- Enter a name for your pipeline.
- Configure Source Stage:
- Choose your source provider (e.g., GitHub).
- Connect to your repository and select the branch to trigger the pipeline.

Screenshot of the AWS CodePipeline Pipeline settings page.

Pipeline settings

Pipeline name: terraform_pipeline

Pipeline type: V2 (selected)

Service role: New service role (selected)

Role name: AWSCodePipelineServiceRole-us-east-1-terraform_pipeline

Allow AWS CodePipeline to create a service role so it can be used with this new pipeline:

CloudShell Feedback

© 2023, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences 28°C Haze 11:46 ENG 14-11-2023

Screenshot of the AWS CodePipeline Pipeline settings page, showing the Variables section.

Variables

You can add variables at the pipeline level. You can choose to assign the value when you start the pipeline. Choosing this option requires pipeline type V2. [Learn more](#)

No variables defined at the pipeline level in this pipeline.

Add variable

You can add up to 50 variables.

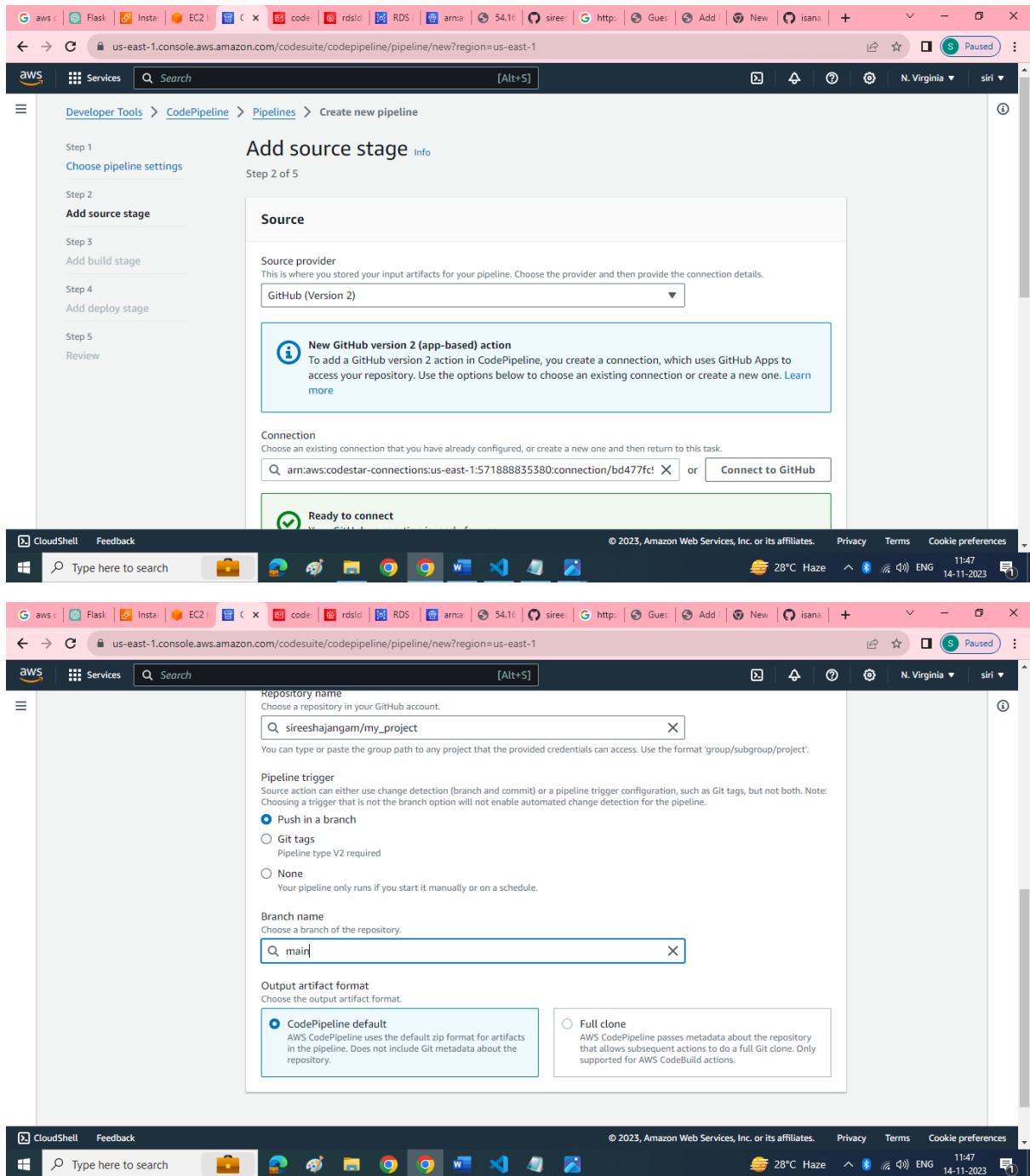
Note: The first pipeline execution will fail if variables have no default values.

Advanced settings

Cancel **Next**

CloudShell Feedback

© 2023, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences 28°C Haze 11:46 ENG 14-11-2023



Step 6: Configure Build Stage (Optional):

- If your application requires build steps, you can configure a build stage using AWS CodeBuild or any other build service.
- Deploy Stage with AWS Code Deploy:

Step 7: Add a deploy stage to your pipeline.

- Select "AWS Code Deploy" as the deployment provider.
- Choose the Code Deploy application and deployment group you created.

- Configure Approval Stage (Optional):
- Optionally, add an approval stage if you want manual approval before deployment.
- Review and Create Pipeline:
- Review your pipeline configuration.
- Click "Create pipeline" to save your pipeline configuration.

Step8: IAM Roles and Permissions

- Code Pipeline Service Role:
- Ensure that the Code Pipeline service role has the necessary permissions to interact with other AWS services (e.g., Code Deploy, Code Build).

Code Deploy Service Role:

- The IAM role used by Code Deploy (specified in the deployment group) should have the required permissions for interacting with EC2 instances, Lambda functions, and other AWS resources.

You cannot skip this stage
Pipelines must have at least two stages. Your second stage must be either a build or deployment stage. Choose a provider for either the build stage or deployment stage.

Deploy

Deploy provider
Choose how you deploy to instances. Choose the provider, and then provide the configuration details for that provider.
AWS CodeDeploy

Region
US East (N. Virginia)

Application name
Choose an application that you have already created in the AWS CodeDeploy console. Or create an application in the AWS CodeDeploy console and then return to this task.

Deployment group
Choose a deployment group that you have already created in the AWS CodeDeploy console. Or create a deployment group in the AWS CodeDeploy console and then return to this task.

Screenshot of the AWS CodePipeline console showing the "Add deploy stage" step. The pipeline is currently at Step 4.

Deploy

Deploy provider
Choose how you deploy to instances. Choose the provider, and then provide the configuration details for that provider.
AWS CodeDeploy

Region
US East (N. Virginia)

Application name
Choose an application that you have already created in the AWS CodeDeploy console. Or create an application in the AWS CodeDeploy console and then return to this task.
guest_notebook

Deployment group
Choose a deployment group that you have already created in the AWS CodeDeploy console. Or create a deployment group in the AWS CodeDeploy console and then return to this task.
demogroup

Cancel Previous Next

CloudShell Feedback © 2023, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences 28°C Haze 11:48 14-11-2023

Screenshot of the AWS CodePipeline console showing the "Step 1: Choose pipeline settings" step. The pipeline is currently at Step 1.

Step 1: Choose pipeline settings

Pipeline settings

Pipeline name: terraform_pipeline
Pipeline type: V2
Artifact location: codepipeline-us-east-1-238650183008
Service role name: AWSCodePipelineServiceRole-us-east-1-terraform_pipeline

Variables

No variables defined at the pipeline level in this pipeline.

CloudShell Feedback © 2023, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences 28°C Haze 11:48 14-11-2023

Step 3: Add build stage

Build action provider

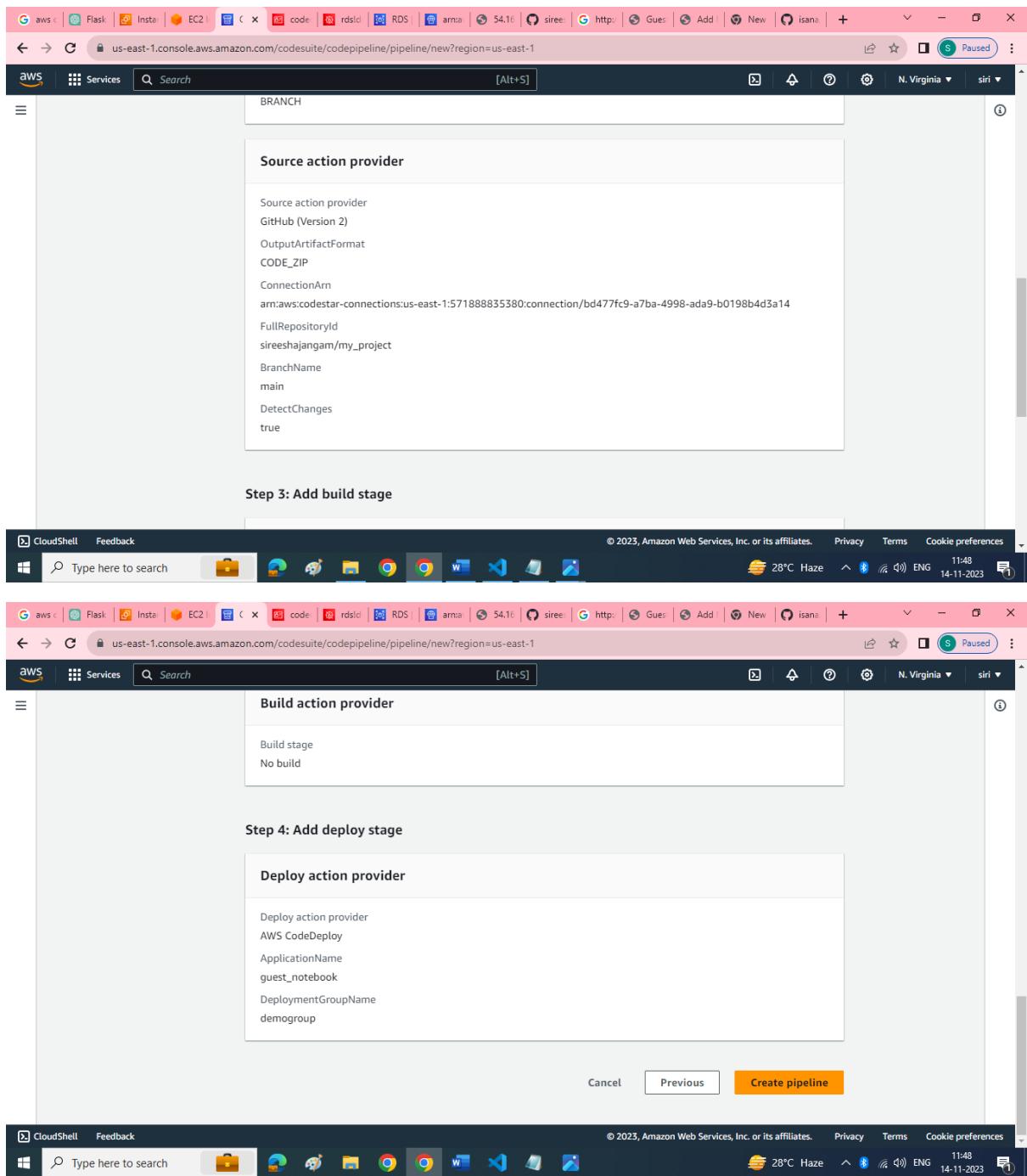
Build stage
No build

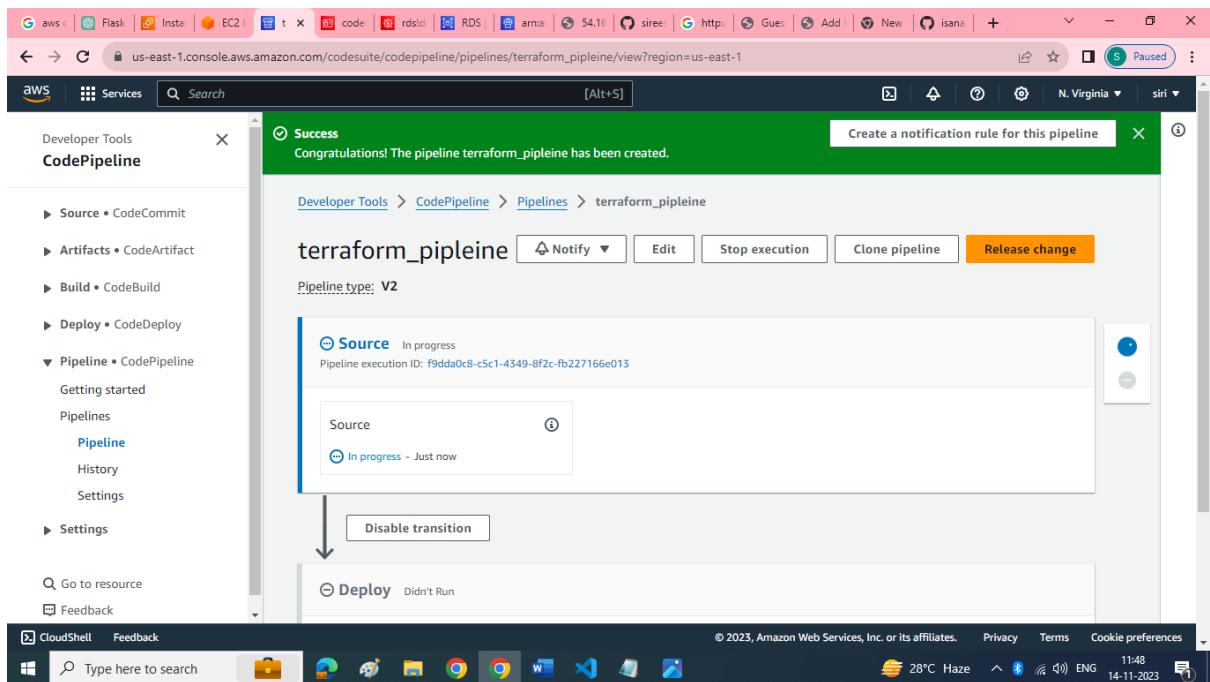
Step 4: Add deploy stage

Deploy action provider

Deploy action provider
AWS CodeDeploy
ApplicationName
guest_notebook
DeploymentGroupName
demogroup

Cancel Previous **Create pipeline**

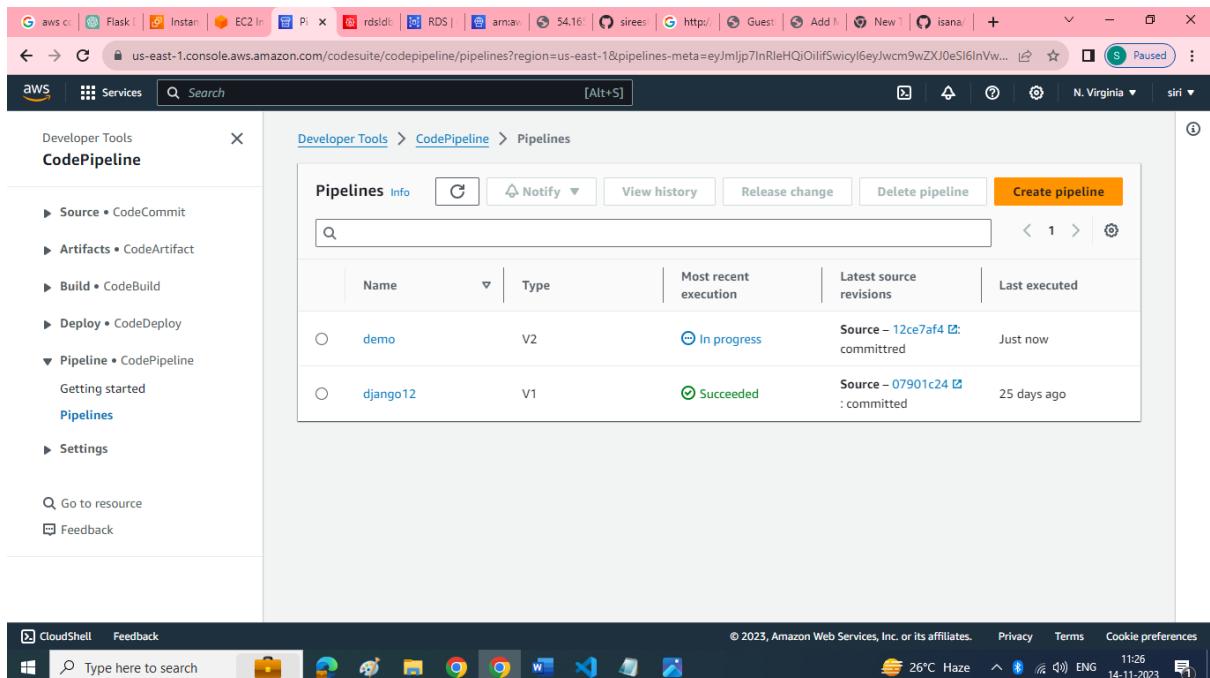




Step 4: Test the Pipeline

- Trigger a Code Pipeline Run:
- Make a change to your source code repository (e.g., push a new commit to the specified branch).
- Monitor Pipeline Execution:

Open the CodePipeline console and monitor the execution of your pipeline. Observe the stages, and check for any errors in the pipeline execution. Approve Manual Approval Stage (if applicable): If you added a manual approval stage, approve the deployment. Verify Deployment:



Screenshot of the AWS CodePipeline console showing a pipeline named "demo".

The pipeline consists of two stages:

- Source**: GitHub (Version 2) - Succeeded, Pipeline execution ID: c684e19f-f80a-4ee0-8b51-d51f6d25d155. Status: Succeeded - 2 minutes ago. Details: 49b5f4a4. A green checkmark icon is present.
- Deploy**: AWS CodeDeploy - Succeeded, Pipeline execution ID: c684e19f-f80a-4ee0-8b51-d51f6d25d155. Status: Succeeded - 1 minute ago. Details: 49b5f4a4. A green checkmark icon is present.

Buttons at the top right include: Notify (dropdown), Edit, Stop execution, Clone pipeline, and Release change (highlighted in orange).

The left sidebar shows the navigation path: Developer Tools > CodePipeline > Pipelines > demo.

Bottom navigation bar includes CloudShell, Feedback, and search bar: Type here to search. Status bar shows: © 2023, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences. Earnings upcoming 11:31 14-11-2023.

Screenshot of the AWS CodePipeline console showing a pipeline named "demo".

The pipeline consists of two stages:

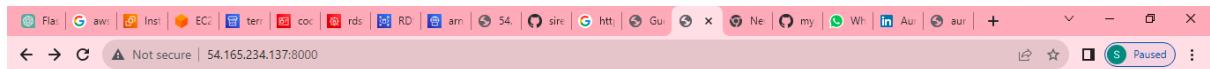
- Source**: GitHub (Version 2) - Succeeded, Pipeline execution ID: c684e19f-f80a-4ee0-8b51-d51f6d25d155. Status: Succeeded - 2 minutes ago. Details: 49b5f4a4. A green checkmark icon is present.
- Deploy**: AWS CodeDeploy - Succeeded, Pipeline execution ID: c684e19f-f80a-4ee0-8b51-d51f6d25d155. Status: Succeeded - 1 minute ago. Details: 49b5f4a4. A green checkmark icon is present.

Buttons at the top right include: Notify (dropdown), Edit, Stop execution, Clone pipeline, and Release change (highlighted in orange).

The left sidebar shows the navigation path: Developer Tools > CodePipeline > Pipelines > demo.

Bottom navigation bar includes CloudShell, Feedback, and search bar: Type here to search. Status bar shows: © 2023, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences. Earnings upcoming 11:31 14-11-2023.

Deployment Output:



Guestbook

- sirshaa: aws 123
- siri: my password is siri123 for gmail

[Add Message](#)



Add Message

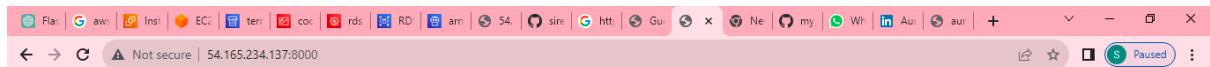
Your Name:
project-3

Message:

[Add Message](#)

[Back to Guestbook](#)





Guestbook

- auopro: project-3
- sirshaa: aws 123
- siri: my password is siri123 for gmail

[Add Message](#)



CONCLUSION

"Through Terraform, we automated the orchestration of AWS resources—VPC, EC2, and RDS—establishing a robust infrastructure. Leveraging GitHub Actions and CodePipeline, I streamlined the application deployment process, demonstrating seamless integration between version control and continuous deployment, enhancing efficiency and reliability in managing infrastructure and deploying applications on AWS."

THANK YOU