AUTOMATING INFRASTRUTURE WITH TERRAFORM

CONTENTS

ABSTRACT

This project represents a comprehensive implementation of Infrastructure as Code (IaaC) using Terraform for orchestrating the deployment and management of cloud infrastructure on Amazon Web Services (AWS). The primary focus is on building a resilient and scalable environment, incorporating key AWS services such as EC2, RDS, VPC, subnets, and security groups. Embracing Terraform's advanced features like modules, workspaces, and a state backend using S3, the project establishes a foundation for flexible and modular infrastructure deployment.

In tandem with infrastructure provisioning, the project integrates Continuous Integration/Continuous Deployment (CI/CD) practices using Jenkins. A Python Flask application, encompassing user authentication functionalities (login and signup pages) and a dashboard, is developed. The application data is stored in an RDS MySQL database, seamlessly connected to the deployed infrastructure. Dockerization of the Flask application ensures consistency across various environments, and the Docker image is stored in Docker Hub.

The Jenkins pipeline orchestrates the end-to-end process, from incrementing version numbers to provisioning infrastructure using Terraform and deploying the Dockerized application on an EC2 instance. The Jenkinsfile includes stages for version management, artifact cleaning, Docker image building and pushing, Terraform infrastructure provisioning, and finally, the deployment of the application on the EC2 instance.

The project's key innovations lie in the integration of Terraform for infrastructure as well as CI/CD automation, ensuring a seamless and reproducible deployment process. The provided Jenkinsfile orchestrates the entire workflow, from initial infrastructure setup to application deployment, highlighting the power of automation in maintaining a consistent and scalable cloud environment. Overall, this project serves as a comprehensive guide for implementing IaaC and CI/CD practices using Terraform and Jenkins, emphasizing efficiency, reliability, and maintainability in cloud-based application deployment.

1. INTRODUCTION

1.1 Purpose and Goals of Implementing IaaC

1. Streamlining Infrastructure Deployment:

Purpose: The primary purpose of implementing Infrastructure as Code (IaaC) is to streamline and automate the deployment of infrastructure components. By codifying infrastructure in a script or configuration file, IaaC enables the automated provisioning of resources, reducing manual intervention and potential errors.

Goals:

- **Accelerate the deployment process:** IaaC aims to speed up the provisioning of infrastructure, allowing teams to respond rapidly to changing business needs.

- **Ensure consistency:** IaaC helps maintain a consistent and reproducible environment across various stages of development, testing, and production.

2. Enhancing Scalability and Flexibility:

- **Purpose:** IaaC supports the dynamic scaling of infrastructure resources based on demand. It allows for the easy addition or removal of resources, promoting adaptability to varying workloads.

- **Goals:**

  - **Scale resources efficiently:** IaaC enables the dynamic scaling of resources, ensuring that the infrastructure can handle varying levels of demand without manual intervention.

  - **Foster flexibility:** The ability to modify infrastructure configurations programmatically enhances the flexibility to accommodate evolving business requirements.

3. Improving Collaboration and Version Control:

- Purpose: IaaC encourages collaboration among development, operations, and other teams involved in the software delivery process. It promotes version control for infrastructure configurations, ensuring traceability and auditability.

- Goals:

- Collaborative development: IaaC allows teams to work collaboratively on infrastructure code, facilitating a DevOps culture and breaking down silos.

- Version-controlled infrastructure: Implementing IaaC ensures that changes to infrastructure configurations are versioned, enabling easy rollbacks and tracking of modifications over time.

4. Mitigating Risk and Enhancing Reproducibility:

- Purpose: IaaC mitigates the risks associated with manual configuration errors by providing a standardized and repeatable way to define and deploy infrastructure.

- Goals:

- Reduce human errors: Automation through IaaC reduces the likelihood of errors introduced by manual configuration, leading to more reliable and stable infrastructure.

- Reproducible deployments: IaaC ensures that infrastructure can be reproduced consistently across different environments, reducing the risk of deployment-related issues.

1.2 Importance of Infrastructure Automation:

1. Efficiency and Time Savings:

- Automated infrastructure provisioning reduces the time and effort required for deployment. This efficiency is crucial for meeting tight deadlines and maintaining a competitive edge.

2. Consistency and Standardization:

- Automation ensures that infrastructure configurations are consistent across different environments, reducing the risk of discrepancies between development, testing, and production.

3. Scalability and Resource Optimization:

- Automation allows for dynamic scaling of resources, optimizing infrastructure usage based on demand. This adaptability is essential for handling variable workloads.

4. Rapid Response to Changes:

- Automated infrastructure facilitates quick adaptation to changes in business requirements, enabling organizations to respond rapidly to evolving needs.

5. Reduction of Manual Errors:

- Automation minimizes the risk of human errors in configuration, resulting in more reliable and secure infrastructure.

6. Improved Collaboration in DevOps Practices:

- Infrastructure automation fosters collaboration between development and operations teams, promoting a DevOps culture and accelerating the software delivery lifecycle.

7. Enhanced Security and Compliance:

- Automation allows for the consistent application of security configurations and compliance policies, reducing vulnerabilities and ensuring a secure infrastructure environment.

2. CHOOSE A TOOL (AWS CLOUDFORMATION OR TERRAFORM) FOR MANAGING INFRASTRUCTURE AS CODE.

The choice between AWS CloudFormation and Terraform for managing infrastructure as code (IaaC) depends on various factors, including your specific requirements, preferences, and the existing environment. Here are some considerations to help you make an informed decision:

2.1 AWS CloudFormation:

Pros:

1. Native Integration: CloudFormation is native to AWS, offering seamless integration with AWS services and resources.

2. AWS-Specific Resource Types: It provides AWS-specific resource types, making it easy to define and provision AWS resources directly.

3. Stack Management: CloudFormation organizes resources into stacks, simplifying the management of related resources.

Cons:

1. AWS-Centric: It is designed specifically for AWS, which might limit flexibility in multi-cloud environments.

2. Learning Curve: While familiar for AWS users, there can be a learning curve, especially for those new to Infrastructure as Code concepts.

2.2 Terraform:

Pros:

1. Cloud Agnostic: Terraform is cloud-agnostic, supporting multiple cloud providers such as AWS, Azure, Google Cloud, and more.

2. Declarative Syntax: It uses a declarative language (HCL), offering a clear and concise way to define infrastructure.

3. Community and Ecosystem: Terraform has a large and active community, contributing to a rich ecosystem of modules and extensions.

4. Modularity: Terraform supports modularity, allowing users to create reusable components and share them across projects.

Cons:

1. Learning Curve: Users new to Infrastructure as Code might face a learning curve when getting started with Terraform.

2. Limited Abstraction: While providing a high level of abstraction, certain cloud-specific features might require knowledge of cloud provider nuances.

Decision:

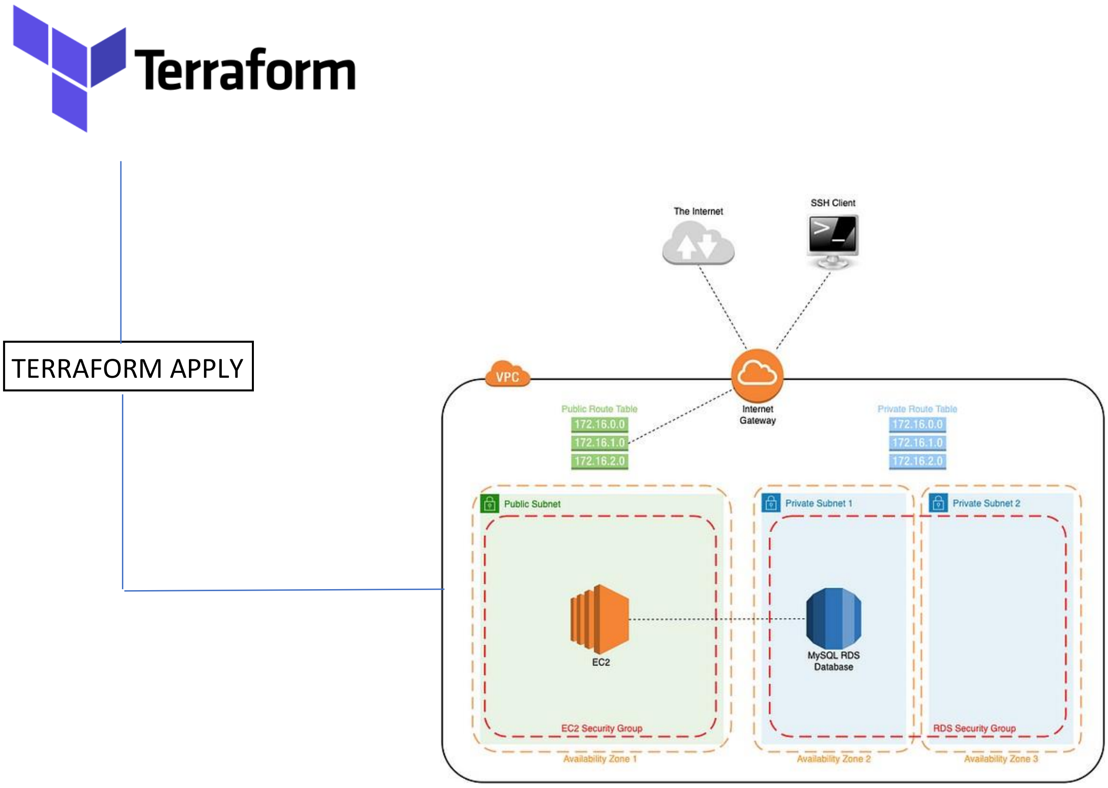
After careful consideration, the project has chosen “Terraform” as its preferred Infrastructure as Code (IaC) tool. This decision is based on Terraform's versatility, enabling seamless management of infrastructure across multiple cloud platforms. The robust community support of Terraform enhances its 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.

This decision positions the project to benefit from Terraform's cloud-agnostic nature, modularity, and continuous development, fostering a scalable and efficient infrastructure management process. The active Terraform community will provide valuable support as you navigate and implement your Infrastructure as Code strategy.

3. 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.

3.1 Project Architecture



3.2 Project Structure:

```
.
├── AuroPro_Project_3
│   ├── entry-script.sh
│   ├── main.tf
│   ├── modules
│   │   ├── database
│   │   │   ├── main.tf
│   │   │   ├── outputs.tf
│   │   │   └── variables.tf
│   │   ├── instances
│   │   │   ├── main.tf
│   │   │   ├── outputs.tf
│   │   │   └── variables.tf
│   │   └── network
│   │       ├── main.tf
│   │       ├── outputs.tf
│   │       └── variables.tf
│   ├── outputs.tf
│   ├── terraform.tfvars
│   └── variables.tf
├── Dockerfile
├── Jenkinsfile
├── app.py
├── docker-compose.yaml
├── requirements.txt
├── server-cmds.sh
├── setup.py
├── static
│   └── style.css
└── templates
    ├── dashboard.html
    ├── login.html
    └── signup.html
```

3.3 Terraform Configuration

1. Provider Configuration

The provider block specifies the AWS region dynamically using variables.

```
provider "aws" {
  region      = "${var.region}"
}
```

2. Terraform Backend Configuration

Terraform backend is configured to store the state file in an S3 bucket.

```
terraform {
  backend "s3" {
    bucket          = "auropro-terraform-project-3"
    key             = "Terraform/terraform.tfstate"
    region          = "ap-south-1"
    #dynamodb_table = "Terraform-state-lock"
  }
}
```

3. Network Infrastructure

3.1 VPC and Subnets

The network module is used to create the VPC and associated subnets.

```
module "my_vpc" {
  source                    = "./modules/network"
  vpc_cidr_block            = "${var.vpc_cidr_block}"
  env_prefix                = "${var.env_prefix}"
  public_subnet_cidr_block      = "${var.public_subnet_cidr_block}"
  private_subnet_cidr_block = "${var.private_subnet_cidr_block}"
  public_subnet_availability_zone = "${var.public_subnet_availability_zone}"
  private_subnet_availability_zone = "${var.private_subnet_availability_zone}"
}
```

4. EC2 Instances

4.1 EC2 Instance Module

A reusable module is used to create EC2 instances dynamically.

```
module "my_instance" {
    source = "./modules/instances"
    ami_id = "${var.ami_id}"
    instance_name = "${var.instance_name}"
    instance_type = "${var.instance_type}"
    subnet_id = module.my_vpc.public_subnet_id
    vpc_id = module.my_vpc.vpc_id
    env_prefix = "${var.env_prefix}"
}
```

5. RDS Databases

5.1 RDS Database Module

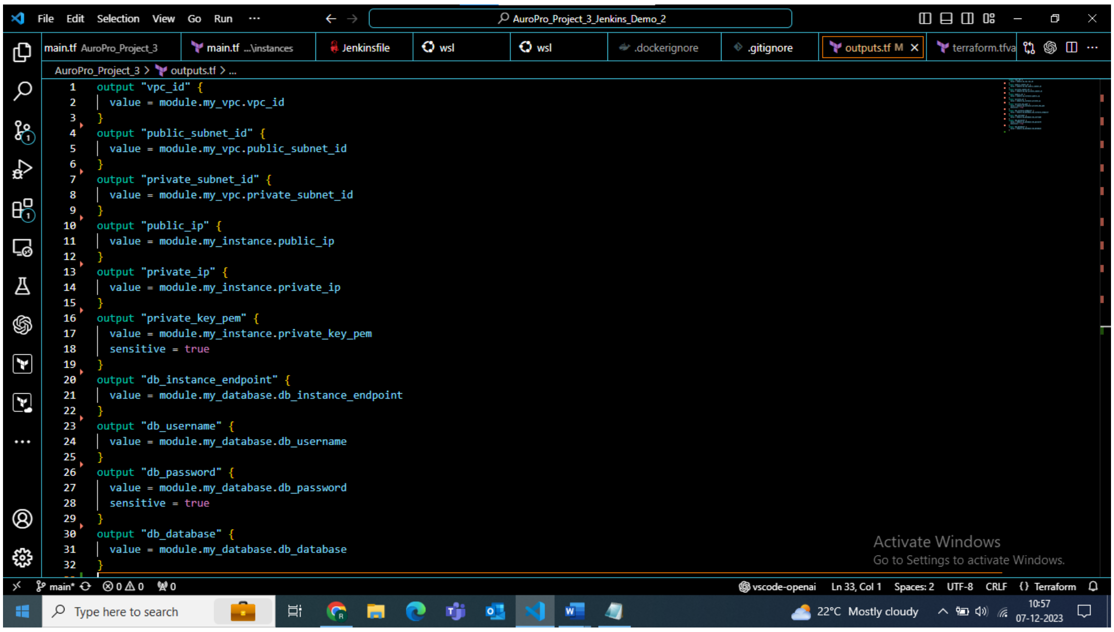
A modular approach is employed to create RDS databases.

```
module "my_database" {
  source                     = "./modules/database"
  vpc_id                     = module.my_vpc.vpc_id
  subnet_id_1                = module.my_vpc.public_subnet_id
  subnet_id_2                = module.my_vpc.private_subnet_id
  env_prefix                 = "${var.env_prefix}"
  db_instance_identifier     = "${var.db_instance_identifier}"
  db_allocated_storage       = "${var.db_allocated_storage}"
  db_engine                  = "${var.db_engine}"
  db_engine_version          = "${var.db_engine_version}"
  db_instance_class          = "${var.db_instance_class}"
  db_name                    = "${var.db_name}"
  db_username                = "${var.db_username}"
  db_password                = "${var.db_password}"
  db_multi_az                = "${var.db_multi_az}"
  db_backup_retention_period = "${var.db_backup_retention_period}"
  my_security_group_id       = module.my_instance.my_security_group_id
}
```

Variables.tf

```
variable "region" {}
variable "vpc_cidr_block" {}
variable "env_prefix" {}
variable "public_subnet_cidr_block" {}
variable "private_subnet_cidr_block" {}
variable "public_subnet_availability_zone" {}
variable "private_subnet_availability_zone" {}
variable "instance_type" {}
variable "ami_id" {}
variable "instance_name" {}

variable "db_instance_identifier" {}
variable "db_allocated_storage" {}
variable "db_engine" {}
variable "db_engine_version" {}
variable "db_instance_class" {}
variable "db_name" {}
variable "db_username" {}
variable "db_password" {}
variable "db_multi_az" {}
variable "db_backup_retention_period" {}
```

output.tf



4. TEST THE TEMPLATES TO ENSURE THEY ACCURETELY REPRESENT YOUR DESIRED INFRASTRUCTURE

- Use validation tools specific to your chosen IaaC tool.
- Simulate stack creations and updates in a non-production environment.

4.1 Test Infrastructure Templates

Testing your infrastructure templates is a crucial step to ensure they accurately represent your desired infrastructure and to identify and address any potential issues. Below is a documentation guide for testing your Terraform templates.

1. Validation Tools:

1.1 Terraform Validation:

The terraform init command initializes your Terraform configuration, downloads providers, and sets up the backend.

terraform init

- This command initializes the working directory and downloads the necessary provider plugins.

- Terraform provides built-in validation tools to check the syntax and structure of your configuration files.

terraform validate

- This command checks your Terraform configuration files for syntax errors and correct resource references.

1.2 Terraform Plan:

- The `terraform plan` command provides an overview of the changes that Terraform plans to make to your infrastructure.

terraform plan

- Review the output to ensure that the planned changes align with your expectations.

2. Simulation in Non-Production Environment:

2.1 Create a Workspace:

- Use Terraform workspaces to create isolated environments for testing.
- Create a new workspace:

  terraform workspace new test

- Switch to the new workspace:

  terraform workspace select test

2.2 Simulate Stack Creations and Updates:

- Simulate the creation and updates of your infrastructure stack in a non-production environment.

  terraform apply

- Review the execution plan and confirm the changes.
- Inspect the created or modified resources in your AWS Management Console.

3.1 Validation and Plan Outputs:

- Document the outputs of the validation and plan commands for future reference.
- **Validation Output:**

  Success! The configuration is valid.

- **Plan Output:**

  Plan: 13 to add, 0 to change, 0 to destroy.

3.2 Simulation Results:

- Document the results of simulating stack creations and updates.

- **Apply Output:**

  Apply complete! Resources: 13 added, 0 changed, 0 destroyed.

Testing and validation are iterative processes. Regularly test your infrastructure templates with updated requirements to ensure they remain accurate and effective. The documentation provided will serve as a reference for understanding the testing procedures and interpreting the results.

5. DEPLOY AND MANAGE YOUR INFRASTRUCTURE USING IAC

- Automate the provisioning of resources for your application.
- Implement version control for IaaC code to track changes and updates.
- Automate the provisioning of resources for your application.
- Implement version control for IaaC code to track changes and update

This documentation provides a step-by-step guide on how to use Jenkins to automate the deployment of a Python Flask application using Docker and Terraform on an AWS EC2 instance.

1. **Setup**
    1. Prerequisites
    2. Installation
2. **Automate Resource Provisioning and Version Control**
3. **Jenkins Pipeline**
    1. Increment Version
    2. Clean Build Artifacts
    3. Build Image
    4. Provision Server
    5. Deploy with Docker Compose and Groovy
4. **Conclusion**

1. **Setup**
    1. **Prerequisites**
        1. Jenkins installed and configured
        2. Python 3 installed on the Jenkins server
        3. Docker installed on the Jenkins server
        4. AWS account with EC2 access
        5. Terraform installed on the Jenkins server

2. **Installation**
    1. Install required Jenkins plugins.
    2. Configure Jenkins credentials for Docker Hub, AWS, and Git.
    3. Install Python and Docker on the Jenkins server.
    4. Install Terraform on the Jenkins server.

2. **Automate Resource Provisioning and Version Control**
    1. Choose an IaC Tool:
        1. Select an IaC tool based on your preferences and requirements. Common choices include Terraform.
    2. Write IaC Code and Python Application:
        1. Create IaC code to define your infrastructure. This code should describe the resources, their configurations, and any dependencies.
    3. Execute IaC Code:
        1. Run the IaC code to provision the defined resources.
    4. Implement Version Control:
        1. Initialize a Git repository if not already done.
        2. Create a '.gitignore' file to exclude unnecessary files (e.g., Terraform state files) from version control.
        3. Commit your IaC code to the Git repository.

3. **Jenkins Pipeline**
    1. Increment Version
        1. This stage increments the version of the Flask application.
    2. Clean Build Artifacts
        1. This stage cleans the build artifacts.
    3. Build Image
        1. This stage builds the Docker image and pushes it to Docker Hub.
    4. Provision Server
        1. This stage provisions an EC2 instance using Terraform.
    5. Deploy
        1. This stage deploys the Flask application on the provisioned EC2 instance.

4. Conclusion

1. This documentation provides a comprehensive guide on setting up and using the Jenkins pipeline for automating the deployment of a Python Flask application using Docker and Terraform on AWS EC2.

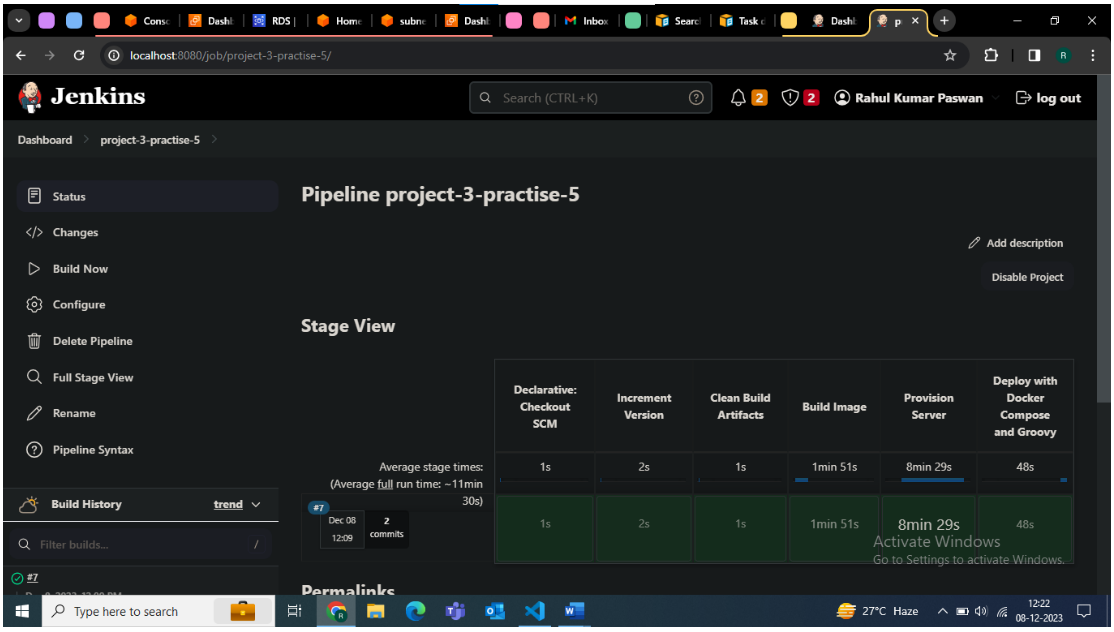
5. RESULTS

5.1 CODE

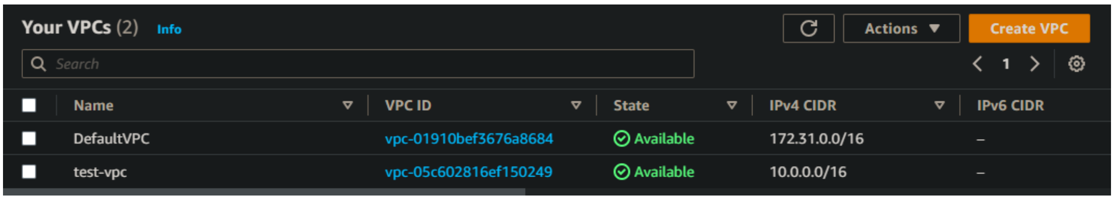
https://github.com/Rahul-Kumar-Paswan/AuroPro_Project_3_Jenkins_Demo_2.git

5.2 SCREENSHOTS

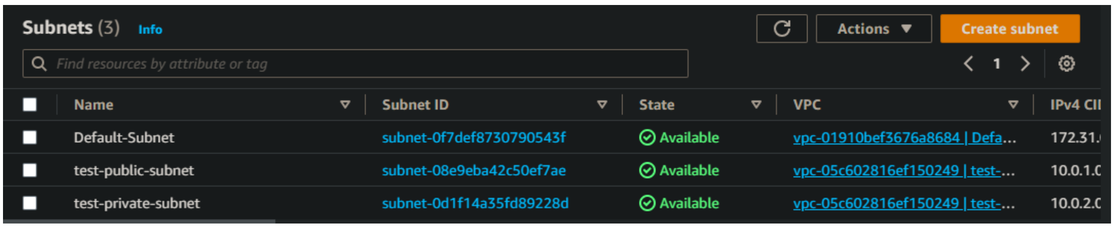
Jenkins Pipeline



VPC'S

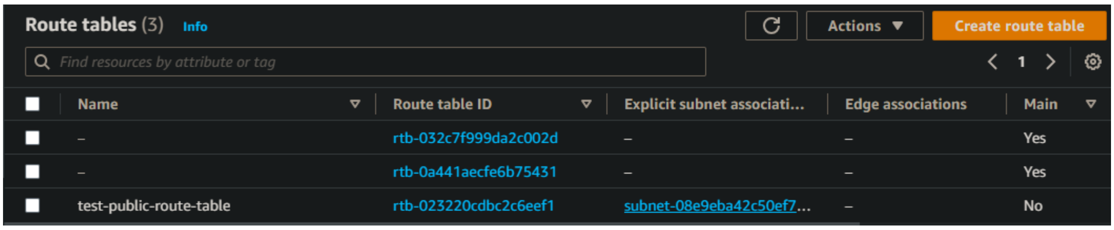


Subnets

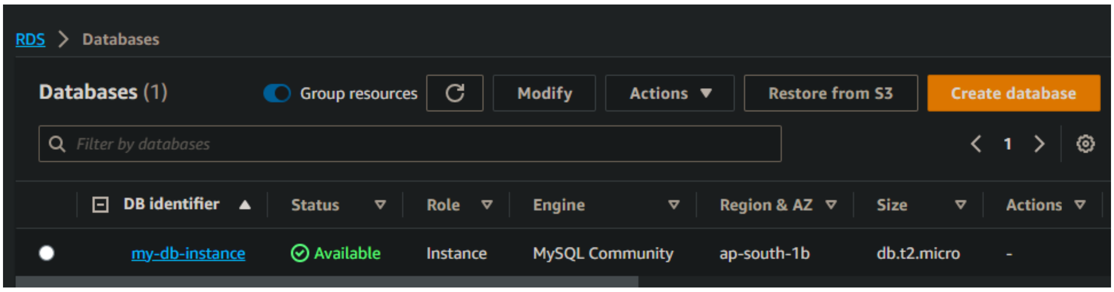


Internet Gateways

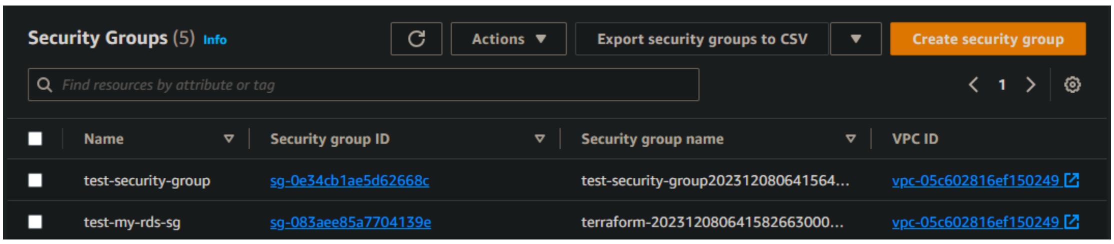
Route Tables



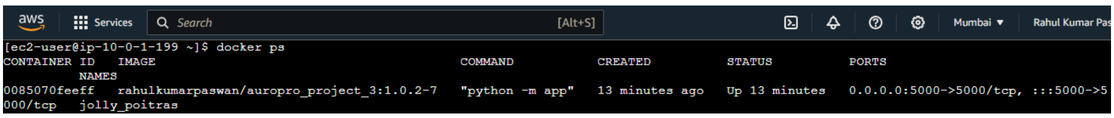
RDS

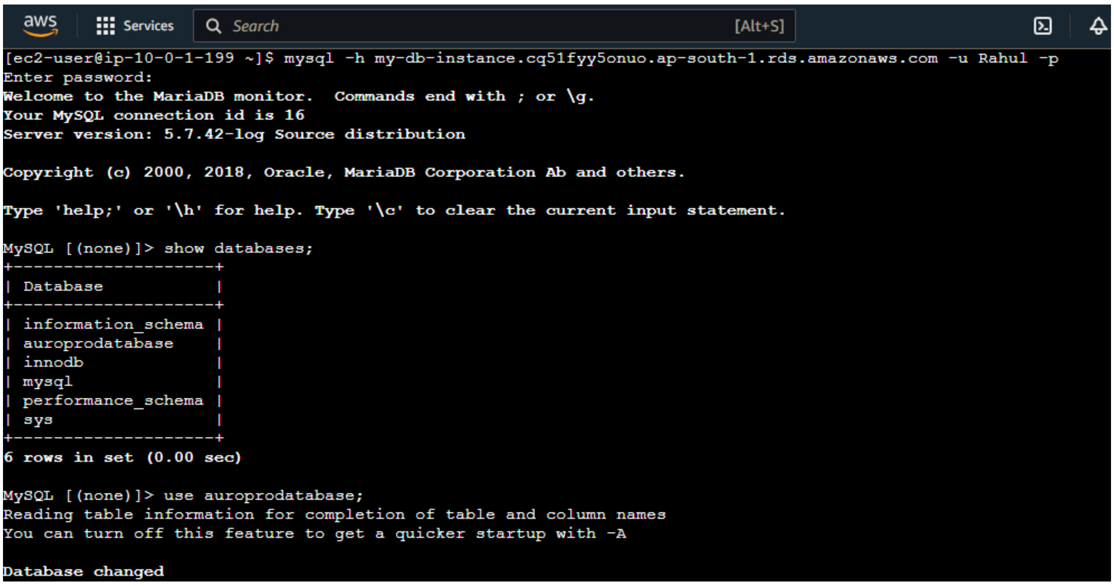


Security Groups

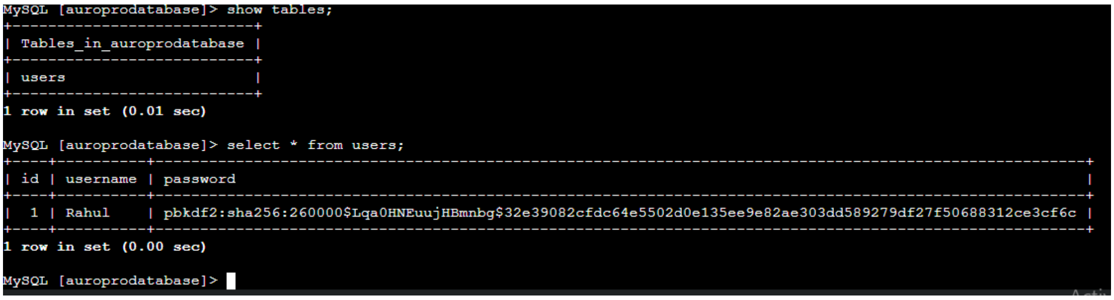


Ec2



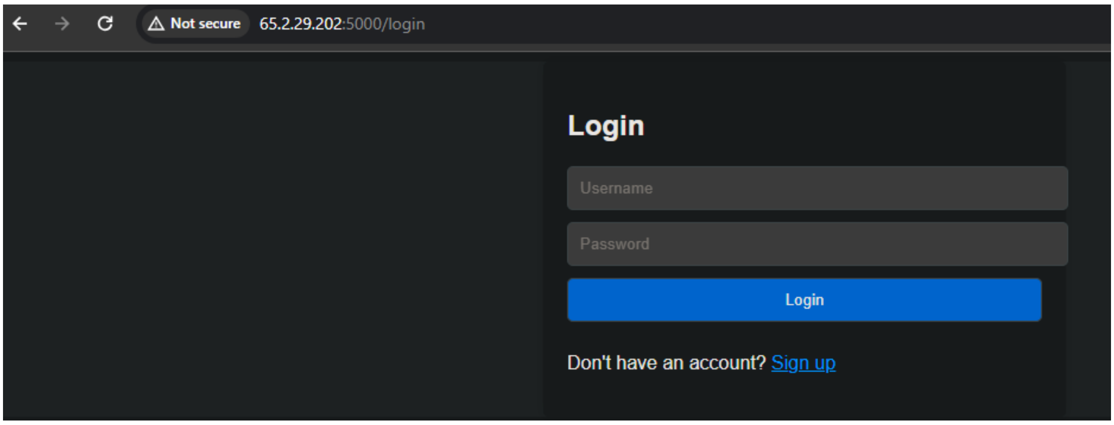


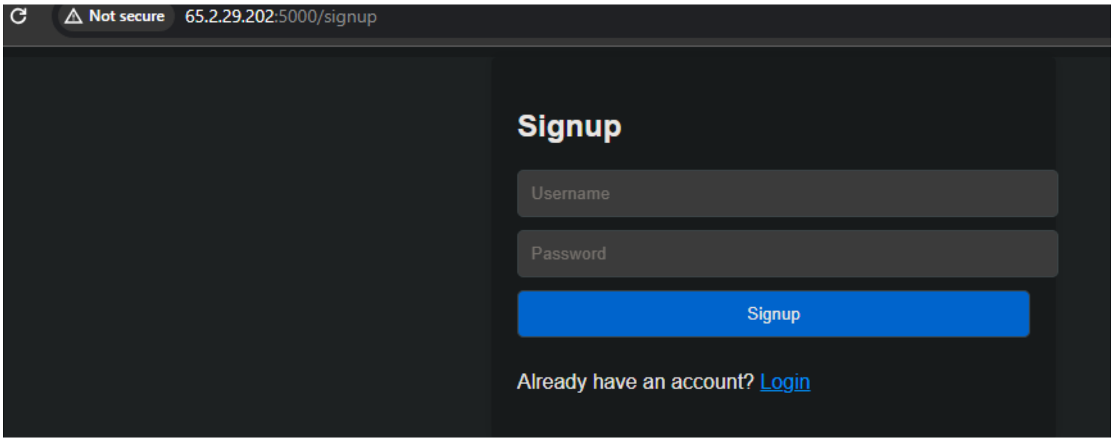

```
[ec2-user@ip-10-0-1-199 ~]$ mysql -h my-db-instance.cq51fyy5onuo.ap-south-1.rds.amazonaws.com -u Rahul -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 16
Server version: 5.7.42-log Source distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| auroprodatabase    |
| innodb             |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
6 rows in set (0.00 sec)

MySQL [(none)]> use auroprodatabase;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

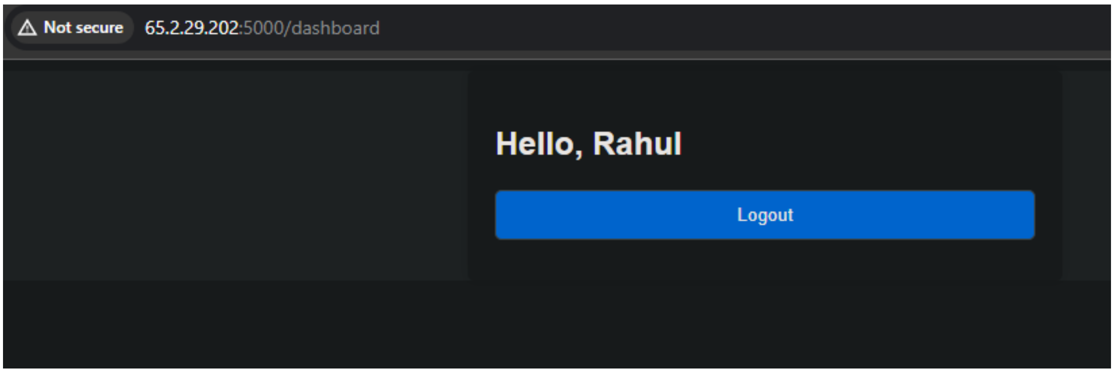


```
MySQL [auroprodatabase]> show tables;
+---------------------------+
| Tables_in_auroprodatabase |
+---------------------------+
| users                     |
+---------------------------+
1 row in set (0.01 sec)

MySQL [auroprodatabase]> select * from users;
+----+----------+-----------------------------------------------------------------------------------------------------+
| id | username | password                                                                                            |
+----+----------+-----------------------------------------------------------------------------------------------------+
|  1 | Rahul    | pbkdf2:sha256:260000$Lqa0HNEuujHBmnbg$32e39082cfdc64e5502d0e135ee9e82ae303dd589279df27f50688312ce3cf6c |
+----+----------+-----------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)

MySQL [auroprodatabase]>
```

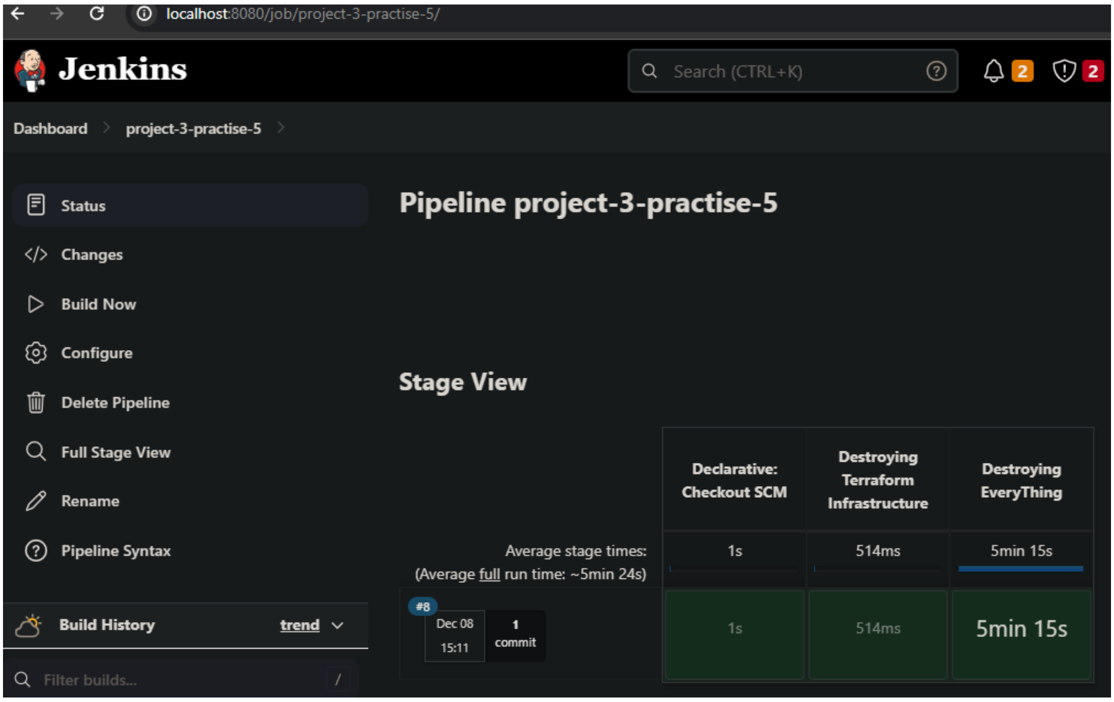
Python Application







Terraform Destroy



6. CONCLUSION

- **Efficiency:** Automation has significantly improved the efficiency of infrastructure provisioning and management.
- **Consistency:** IaC ensures that infrastructure configurations remain consistent across different environments.
- **Traceability:** Version control implementation provides a detailed history of changes, facilitating easy tracking and auditing.

7. FUTURE ENHANCEMENT

While the current implementation is robust, there are opportunities for future enhancements to further improve the IaC process and overall infrastructure management:

Enhancements

Continuous Integration/Continuous Deployment (CI/CD): Implement CI/CD pipelines to automate the testing and deployment of IaC changes, ensuring faster and more reliable updates to the infrastructure.

Advanced Monitoring and Logging: Integrate advanced monitoring and logging solutions to gain deeper insights into the performance and health of the deployed infrastructure.

Cost Optimization: Explore strategies for cost optimization, such as leveraging reserved instances, right-sizing resources, and implementing auto-scaling based on demand.

Security Improvements: Enhance security measures by incorporating best practices for IAM roles, encryption, and regular security audits.

Documentation Updates: Keep documentation up-to-date with any changes to the infrastructure, making it easier for team members to understand and contribute.

Multi-Cloud Support: If applicable, consider extending IaC support to multiple cloud providers for increased flexibility and resilience.