

AUTOMATING INFRASTRUTURE WITH TERRAFORM

CONTENTS

S.NO	TITLE	PAGE NO
	COVERPAGE	i
	ABSTRACT	iv
1	INTRODUCTION	
	1.1. Purpose and Goals of Implementing IaaS	1
	1.2. Importance of Infrastructure Automation Objective	3
2	CHOOSE A TOOL	
	2.1. AWS CloudFormation pros and cons	4
	2.2. Terraform pros and cons	4
3	CREATE CODE TEMPLATES TO DEFINE YOUR AWS INFRASTRUCTURE	
	3.1. Project Architecture	6
	3.2. Project Structure	7
	3.3. Terraform Configuration	8
4	TEST THE TEMPLATES TO ENSURE THEY ACCURATELY REPRESENT YOUR DESIRED INFRASTRUCTURE	
	4.1. Test Infrastructure Templates	11

5	DEPLOY AND MANAGE YOUR INFRASTRUCTURE USING IAC	14
6	RESULTS	
	6.1. Code	17
	6.2. Screenshots	17
7	CONCLUSION	21
8	FUTURE ENHANCEMENT	22

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:** IaaS 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:** IaaS allows teams to work collaboratively on infrastructure code, facilitating a DevOps culture and breaking down silos.

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

4. Mitigating Risk and Enhancing Reproducibility:

- **Purpose:** IaaS 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 IaaS reduces the likelihood of errors introduced by manual configuration, leading to more reliable and stable infrastructure.

- **Reproducible deployments:** IaaS 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 (IaC) 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:

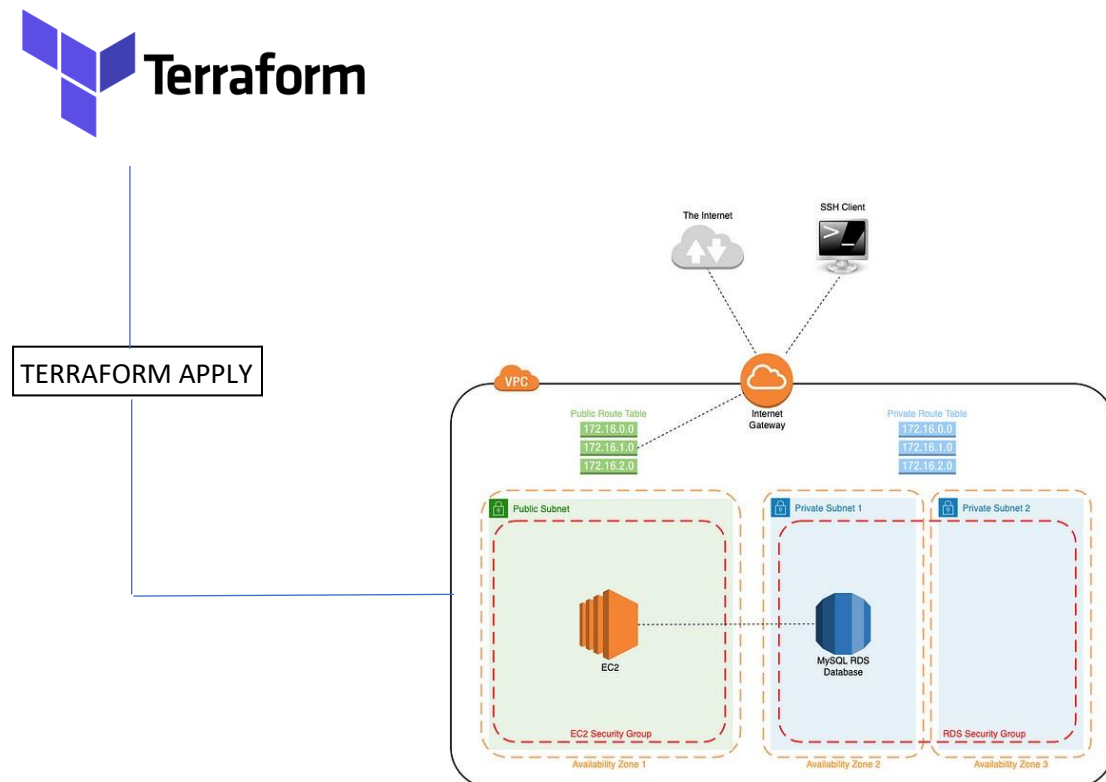
After careful consideration, the project has chosen “Terraform” as its preferred Infrastructure as Code (IaC) tool. This decision is based on Terraform's versatility, enabling seamless management of infrastructure across multiple cloud platforms. The robust community support of Terraform enhances its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

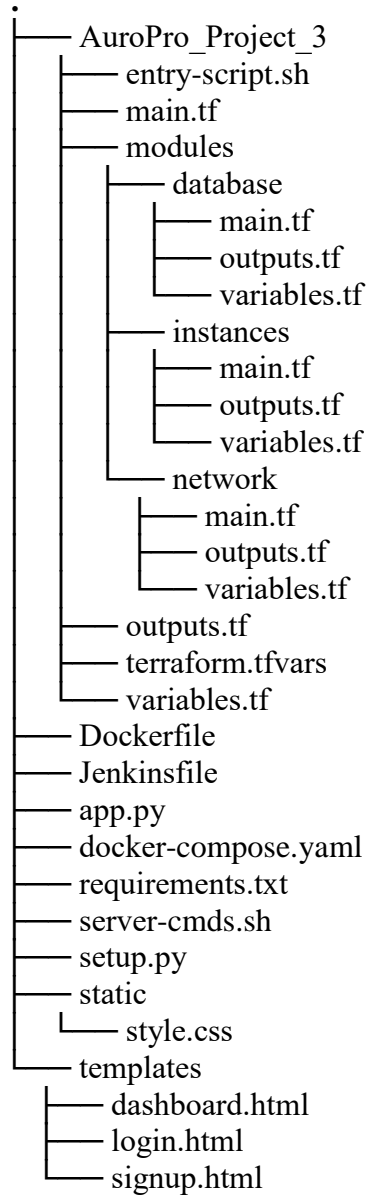
3. CREATE CODE TEMPLATES TO DEFINE YOUR AWS INFRASTRUCTURE

- 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:



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.

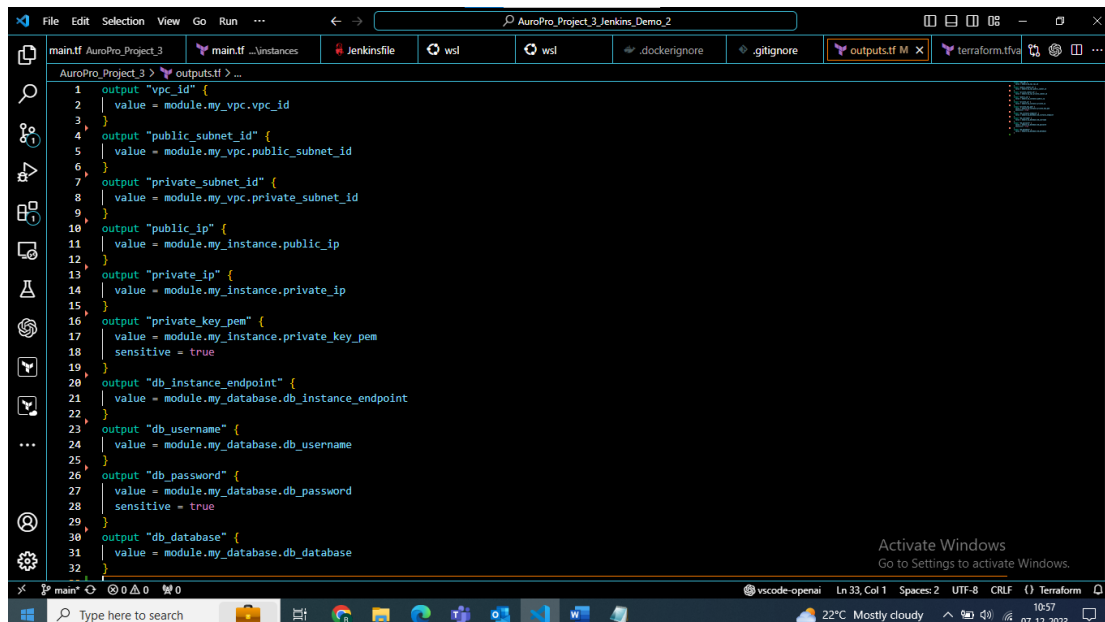
```
module "my_database" {  
    source = "../modules/database"  
    vpc_id = module.my_vpc.vpc_id  
    subnet_id_1 = module.my_vpc.public_subnet_id  
    subnet_id_2 = module.my_vpc.private_subnet_id  
    env_prefix = "${var.env_prefix}"  
    db_instance_identifier = "${var.db_instance_identifier}"  
    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



```
1  output "vpc_id" {
2    | value = module.my_vpc.vpc_id
3  }
4  output "public_subnet_id" {
5    | value = module.my_vpc.public_subnet_id
6  }
7  output "private_subnet_id" {
8    | value = module.my_vpc.private_subnet_id
9  }
10 output "public_ip" {
11   | value = module.my_instance.public_ip
12 }
13 output "private_ip" {
14   | value = module.my_instance.private_ip
15 }
16 output "private_key_pem" {
17   | value = module.my_instance.private_key_pem
18   | sensitive = true
19 }
20 output "db_instance_endpoint" {
21   | value = module.my_database.db_instance_endpoint
22 }
23 output "db_username" {
24   | value = module.my_database.db_username
25 }
26 output "db_password" {
27   | value = module.my_database.db_password
28   | sensitive = true
29 }
30 output "db_database" {
31   | value = module.my_database.db_database
32 }
```

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

- Use validation tools specific to your chosen IaC 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

The screenshot shows the Jenkins web interface for a pipeline named 'project-3-practise-5'. The left sidebar contains navigation options: Status, Changes, Build Now, Configure, Delete Pipeline, Full Stage View, Rename, and Pipeline Syntax. The main area displays the 'Stage View' with a table of stages and their durations. Below the table, it shows the average stage times and the average full run time. The bottom of the interface includes a 'Build History' section and a 'Permalinks' section.

Declarative: Checkout SCM	Increment Version	Clean Build Artifacts	Build Image	Provision Server	Deploy with Docker Compose and Groovy
1s	2s	1s	1min 51s	8min 29s	48s

Average stage times:
(Average full run time: ~11min 30s)

Build History: Dec 08 12:09, 2 commits

VPC'S

Your VPCs (2) Info							Refresh	Actions	Create VPC
<input type="text" value="Search"/>							1		
<input type="checkbox"/>	Name	VPC ID	State	IPv4 CIDR	IPv6 CIDR				
<input type="checkbox"/>	DefaultVPC	vpc-01910bef3676a8684	Available	172.31.0.0/16	-				
<input type="checkbox"/>	test-vpc	vpc-05c602816ef150249	Available	10.0.0.0/16	-				

Subnets

Subnets (3) Info							Refresh	Actions	Create subnet
<input type="text" value="Find resources by attribute or tag"/>							1		
<input type="checkbox"/>	Name	Subnet ID	State	VPC	IPv4 CIDR				
<input type="checkbox"/>	Default-Subnet	subnet-0f7def8730790543f	Available	vpc-01910bef3676a8684 Defa...	172.31.0.0/16				
<input type="checkbox"/>	test-public-subnet	subnet-08e9eba42c50ef7ae	Available	vpc-05c602816ef150249 test...	10.0.1.0/24				
<input type="checkbox"/>	test-private-subnet	subnet-0d1f14a35fd89228d	Available	vpc-05c602816ef150249 test...	10.0.2.0/24				

Internet Gateways

Internet gateways (2) Info				
<input type="text" value="Search"/>				
<input type="checkbox"/>	Name	Internet gateway ID	State	VPC ID
<input type="checkbox"/>	test-internet-gateway	igw-03df1e45e4ee7febf	Attached	vpc-05c602816ef150249 test-vpc
<input type="checkbox"/>	demo-igw	igw-0b0c32e13ee262bda	Attached	vpc-01910bef3676a8684 DefaultVPC

Route Tables

Route tables (3) Info						
<input type="text" value="Find resources by attribute or tag"/>						
<input type="checkbox"/>	Name	Route table ID	Explicit subnet associati...	Edge associations	Main	
<input type="checkbox"/>	-	rtb-032c7f999da2c002d	-	-	Yes	
<input type="checkbox"/>	-	rtb-0a441aecfe6b75431	-	-	Yes	
<input type="checkbox"/>	test-public-route-table	rtb-023220cdb2c6eef1	subnet-08e9eba42c50ef7...	-	No	

RDS

RDS > Databases							
Databases (1) Group resources Modify Actions Restore from S3 Create database							
<input type="text" value="Filter by databases"/>							
<input type="checkbox"/>	DB identifier	Status	Role	Engine	Region & AZ	Size	Actions
<input type="checkbox"/>	my-db-instance	Available	Instance	MySQL Community	ap-south-1b	db.t2.micro	-

Security Groups

Security Groups (5) Info				
<input type="text" value="Find resources by attribute or tag"/>				
<input type="checkbox"/>	Name	Security group ID	Security group name	VPC ID
<input type="checkbox"/>	test-security-group	sg-0e34cb1ae5d62668c	test-security-group202312080641564...	vpc-05c602816ef150249
<input type="checkbox"/>	test-my-rds-sg	sg-083aee85a7704139e	terraform-202312080641582663000...	vpc-05c602816ef150249

Ec2

aws Services <input type="text" value="Search"/> [Alt+S]					
[ec2-user@ip-10-0-1-199 ~]\$ docker ps					
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
0085070feeff	rahulkumarpswan/auropo_project_3:1.0.2-7	"python -m app"	13 minutes ago	Up 13 minutes	0.0.0.0:5000->5000/tcp, :::5000->5000/tcp
jolly_poitras					

```
aws Services Search [Alt+S]
[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

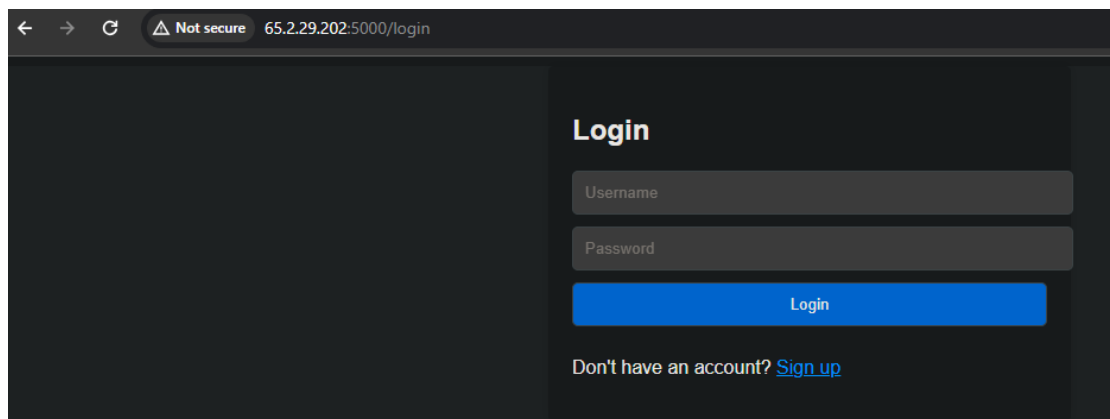
Database changed

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$32e39082cfd64e5502d0e135ee9e82ae303dd589279df27f50688312ce3cf6c |
+-----+
1 row in set (0.00 sec)

MySQL [auroprodatabase]>
```

Python Application



Not secure 65.2.29.202:5000/signup

Signup

Username

Password

Signup

Already have an account? [Login](#)

Not secure 65.2.29.202:5000/dashboard

Hello, Rahul

Logout

Terraform Destroy

localhost:8080/job/project-3-practise-5/

Jenkins

Search (CTRL+K)

Dashboard > project-3-practise-5

Pipeline project-3-practise-5

- Status
- Changes
- Build Now
- Configure
- Delete Pipeline
- Full Stage View
- Rename
- Pipeline Syntax

Stage View

Average stage times:
(Average full run time: ~5min 24s)

Declarative: Checkout SCM	Destroying Terraform Infrastructure	Destroying Everything
1s	514ms	5min 15s
1s	514ms	5min 15s

#8 Dec 08 15:11 1 commit

Build History trend

Filter builds...

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.