International Journal of Trend in Scientific Research and Development (IJTSRD)
Special Issue on Innovations in Computer Science and Applications
Available Online: www.ijtsrd.com e-ISSN: 2456 – 6470

Cost-Optimized and Secure Infrastructure Automation on AWS Using Terraform

Ashish Khatri

Department of Science and Technology,
G. H. Raisoni Skill Tech University, Nagpur, Maharashtra, India

Abstract

Cloud computing is widely used today because it offers flexible and scalable infrastructure. Amazon Web Services (AWS) provides various services such as EC2, VPC, IAM, and Auto Scaling to develop cloud environments. However, manually setting up and managing infrastructure in AWS can be time-consuming and may result in configuration errors, security vulnerabilities, and increased cloud costs. In this project, AWS infrastructure is automated using Terraform, an Infrastructure as Code (IaC) tool. Instead of creating resources manually through the AWS console, infrastructure is defined using code and deployed automatically. The main aim of this research is to reduce deployment time, enhance security, and optimize cloud costs. A modular Terraform structure is employed to create VPC, EC2 instances, security groups, and IAM roles. Cost optimization methods such as instance rightsizing and Auto Scaling are also implemented.

The findings indicate that automated deployment is faster and more consistent than manual setup.

It also helps to reduce unnecessary cloud expenses and enhance security configurations. This study demonstrates that integrating automation, cost optimization, and security practices results in a more efficient and reliable cloud infrastructure. The research centers around automating AWS infrastructure using Terraform, an Infrastructure as Code (IaC) tool. Rather than manually configuring resources, the entire infrastructure is described in code files and deployed automatically. In this project, a modular Terraform structure is used to provision VPC, public and private subnets, EC2 instances, security groups, IAM roles, and Auto Scaling configurations. The primary goal of this research is to build a secure and cost-effective cloud environment. To achieve cost efficiency, instance rightsizing and Auto Scaling techniques are applied to avoid over-provisioning of resources. For improved security, IAM least-privilege policies, private subnet deployment, and restricted security group rules are implemented. The performance of manual and automated deployment is evaluated based on deployment time and configuration consistency. Infrastructure automation significantly reduces deployment time and minimizes human errors. It also improves security posture and supports better cloud cost management. This research shows that combining automation, cost optimization strategies, and security best practices can result in a more efficient, reliable, and scalable cloud infrastructure for organizations.

KEYWORDS: *Infrastructure as Code; Terraform; Amazon Web Services (AWS); Cloud Automation; Cost Optimization; Cloud Security; DevOps; Auto Scaling; Virtual Private Cloud (VPC); IAM.*

1. Introduction

Cloud computing is widely adopted because it enables businesses to run applications without buying and maintaining physical servers. Amazon Web Services (AWS) is one of the most popular cloud platforms, offering a range of services like EC2 for virtual machines, VPC for network configuration, IAM for managing access, and Auto Scaling to handle varying workloads. Many organizations currently set up their cloud infrastructure manually through the AWS console, but this process can be slow and error-prone. For example, misconfigured security groups can lead to security vulnerabilities. Furthermore, poor management of cloud resources can result in increased costs due to the use of unnecessary or oversized instances. To overcome these issues, Infrastructure as Code (IaC) is used. IaC is the practice of defining and managing cloud resources using code rather than manual setup. Terraform is a widely used tool that allows users to write code to automatically create and manage AWS resources. This approach ensures that infrastructure can be consistently reproduced without mistakes. In this project, AWS infrastructure is automated using Terraform. The main objective is to reduce deployment time, improve security configurations, and manage cloud costs. The infrastructure includes components such as a VPC, subnets, EC2 instances, security groups, IAM roles, and Auto Scaling. This research illustrates how automation can enhance the security, consistency, and cost-efficiency of cloud infrastructure.

A major issue in cloud computing is managing costs.

AWS uses a pay-as-you-go model, where users are charged based on their resource usage. If instances are too large or if unused resources remain active, cloud expenses can increase significantly. Often, organizations neglect to shut down idle instances or fail to configure Auto Scaling properly, resulting in unnecessary charges. To tackle these problems, IaC has become increasingly popular. IaC involves writing code files that define cloud infrastructure instead of manually creating resources. Terraform is one of the most commonly used IaC tools. It allows users to create configuration files that describe the required infrastructure. Once the code is prepared, Terraform automatically sets up and manages AWS resources, reducing manual work and ensuring consistent configuration across multiple deployments.

In this research, AWS infrastructure is automated using Terraform.

The setup involves creating a VPC with both public and private subnets, launching EC2 instances, configuring security groups, assigning IAM roles, and setting up Auto Scaling. Alongside automation, this project focuses on security and cost efficiency. Security measures such as providing the least necessary IAM permissions and

implementing strict inbound traffic rules are applied. Cost optimization techniques like selecting appropriate instance types and enabling Auto Scaling help reduce expenses. The aim of this research is to demonstrate how integrating automation with strong security and cost control can improve cloud infrastructure management. By using Terraform, deployments become faster, more reliable, and less prone to errors. This structured approach helps organizations maintain secure and cost-effective cloud environments.

1.1. Motivation

In many organizations, cloud infrastructure is still created manually. This process requires repeating many steps and checking settings again and again. Sometimes mistakes happen in IAM permissions or network settings, which can create security risks. Another problem is high cloud cost. If instances are not properly sized or if Auto Scaling is not used, companies may pay more than required. Many times, unused resources remain active and increase the monthly bill.

Because of these problems, there is a need for a system that can automatically create infrastructure with proper security and cost control. In this project, Terraform is used to automate AWS infrastructure. Security rules and cost optimization techniques are included during the design itself. This helps in creating a better and safer cloud environment. This work demonstrates how automation, security best practices, and cost control strategies can be combined into a single framework for better cloud infrastructure management.

1.2. Contribution

The main contributions of this project are:

1. Automating AWS infrastructure using Terraform.
2. Creating a secure VPC with public and private subnets.
3. Applying IAM least-privilege access rules.
4. Using Auto Scaling and proper instance type to reduce cost.
5. Comparing manual setup with automated setup.
6. Developing a simple and repeatable cloud deployment model.

The rest of the paper explains related work, methodology, results, and conclusion of this research. This research contributes by presenting a simple and practical approach to automate AWS infrastructure using Terraform. Instead of manually configuring cloud resources, the entire infrastructure is defined using Infrastructure as Code, which makes deployment faster and more consistent. The study focuses not only on automation but also on integrating security and cost optimization during the design phase itself. A secure network architecture is created using VPC with public and private subnets, and IAM least-privilege policies are applied to control access. Security group rules are carefully configured to restrict unnecessary inbound traffic.

2. Related work

In this part, we look at previous work on cloud automation, infrastructure as code, cost optimization, and cloud security. As more and more companies use cloud computing, many researchers and organizations have worked on improving how infrastructure is managed through automation tools. AWS is widely used for deploying applications, but managing cloud resources manually often leads to problems with costs and security. Because of this, many studies have looked into using automation tools to reduce human errors and make processes more efficient.

Several research studies highlight the importance of Infrastructure as Code (IaC) in cloud environments.

IaC lets infrastructure be defined using configuration files instead of manual setup through a console. Tools like Terraform and AWS CloudFormation are commonly used for this. Research shows that using IaC improves consistency in deployments and reduces differences in configurations between environments. It also helps with version control and teamwork among groups.

Some studies focus on techniques for optimizing cloud costs.

Researchers have talked about the importance of instance rightsizing, Auto Scaling, resource tagging, and using monitoring tools to cut down on unnecessary expenses. It has been noted that many organizations end up spending more on cloud services because of idle instances and wrong scaling setups. Using automated scaling policies and continuous monitoring can help reduce cloud costs without affecting performance.

Cloud security is another important area of research.

Many security breaches in the cloud happen because of misconfigured IAM policies or open security group ports. Research emphasizes the shared responsibility model in cloud computing, where users are responsible for securing their own configurations. Studies suggest using least-privilege access control, network isolation through private subnets, and regular permission audits to improve security.

Some recent studies combine DevOps and cloud automation, often called DevSecOps.

This approach includes security checks and policy enforcement within the automation process. Rather than applying security measures after deployment, security is built in during infrastructure creation. This lowers risks and ensures a secure cloud setup from the start.

However, most existing studies look at automation, cost optimization, or security separately.

There's not much research that combines all three aspects into a single structured model. Because of this, this project focuses on integrating automation using Terraform with cost control methods and security best practices in AWS infrastructure. The aim is to create a practical and repeatable framework that speeds up deployment, lowers costs, and boosts security.

Cost optimization is a major area of research in cloud computing.

Many organizations end up with unexpected cloud bills due to poor resource planning. Researchers suggest techniques like instance rightsizing, Auto Scaling, Reserved Instances, and monitoring using AWS CloudWatch. Studies show that idle EC2 instances and unused storage services are mainly responsible for cost wastage. Automated scaling policies help adjust resources based on workload demand, which improves cost efficiency.

Security in AWS cloud environments has also been widely studied.

Research shows that misconfigured IAM roles and open security group ports are common reasons for cloud security breaches. The principle of least privilege is strongly recommended in many studies. Network segmentation using public and private subnets within a VPC is also seen as a best practice. Some research suggests implementing multi-layer security that includes IAM policies, network ACLs, encryption, and monitoring tools like AWS CloudTrail.

Recent studies introduce the idea of embedding security directly into development and deployment processes.

Instead of checking security after deployment, policies are enforced during infrastructure setup. Automated security checks, policy validation tools, and compliance scanning are becoming common in modern cloud infrastructure management. Continuous Integration and Continuous Deployment (CI/CD) pipelines are often used with Terraform to automate both infrastructure and application deployment. Studies show that automation reduces manual work and increases productivity. Automated scaling policies help adjust resources based on workload demand, which improves cost efficiency.

Even though many studies look at automation, cost management, or security on their own, there's limited research combining all three into one unified model.

Most research either focuses on deployment automation alone or cost analysis alone. Therefore, this project aims to integrate infrastructure automation using Terraform with built-in security best practices and cost optimization strategies in AWS. The goal is to create a structured and reusable cloud architecture that is secure, scalable, and cost-efficient.

3. Research Methodology

3.1 Problem statement

Today, many organizations are moving their applications to the AWS cloud due to its flexibility and scalability. However, managing cloud infrastructure manually presents significant challenges. When setting up resources such as EC2 instances, VPCs, subnets, security groups, load balancers, and IAM roles through the AWS console, the process is time-consuming. Even minor configuration errors can lead to serious issues within the system. One major challenge is inconsistency. When creating infrastructure manually, it is difficult to replicate the exact same setup in another environment. For instance, if a development environment is created and later a production environment is needed, subtle differences in configuration may occur. These can result in application failures or unexpected behavior. Another critical issue is security. Sometimes, users grant full access permissions in IAM roles or leave unnecessary ports open in security groups, which can expose the system to cyber threats. Many security breaches result from misconfigurations rather than weak systems.

Cost management is another significant challenge in cloud computing.

EC2 instances often remain active even when there is no traffic, leading to unnecessary expenses. If Auto Scaling is not properly configured, resources may not scale up during high demand or scale down during low usage, resulting in higher cloud costs and wasted resources. Manual effort is also a problem. Every time infrastructure needs to be updated, administrators have to log in and make changes manually, increasing workload and the likelihood of errors. This approach is not scalable for growing organizations. Therefore, there is a need for a solution that automates infrastructure creation, reduces manual mistakes, enhances security, and manages costs efficiently. In this project, we are using Terraform to implement Infrastructure as Code. By writing infrastructure in code, we can create, update, and manage AWS resources in a simple, repeatable, and secure manner.

Currently, many companies are using AWS cloud to deploy their applications.

However, manually creating and managing cloud resources via the AWS console is not straightforward. It is time-consuming and can lead to serious problems. For example, if an incorrect port is opened in a security group or full access is granted in an IAM role, it can introduce security vulnerabilities. Additionally, manually creating resources makes it difficult to replicate the same setup in another environment. Another problem is cloud costs. Many EC2 instances continue to run even when they are not needed, causing companies to pay more. Proper scaling is also challenging if not configured correctly. Therefore, there is a need for automation that can create infrastructure in a simple and repeatable way. In this project, we are addressing these issues by using Terraform. Terraform enables the creation of AWS infrastructure through code rather than manual steps, making the deployment process faster, reducing errors, improving security, and assisting in better cost management

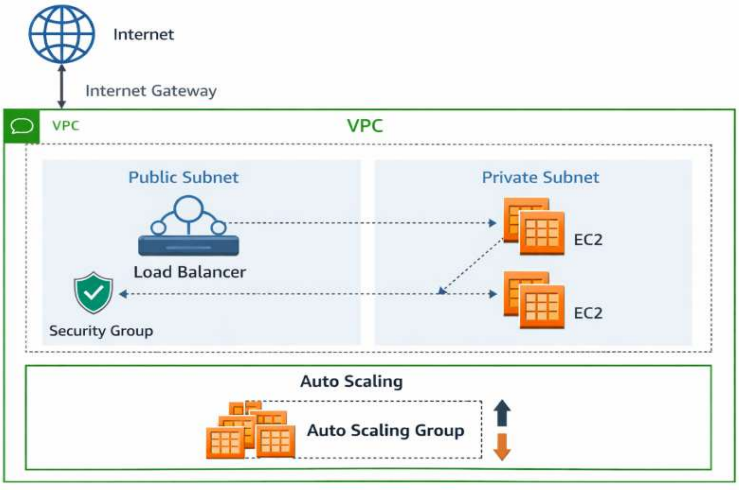


Fig 1. Diagram

3.2. Proposed System Overview

The proposed system automates AWS infrastructure using Terraform scripts. Instead of manually creating resources, all configurations are written in .tf files. These files define networking, compute resources, security settings, and scaling policies. Temporal Facial Feature Analysis.

The overall workflow of the system is:

1. Define infrastructure in Terraform configuration files.
2. Initialize Terraform environment.
3. Validate and plan infrastructure deployment.
4. Apply configuration to create AWS resources.
5. Monitor performance and scaling behavior.
6. Evaluate cost and security configuration.

The system ensures that infrastructure can be recreated anytime using the same configuration files. we create a VPC to build a secure network. Inside that, we create one public subnet and one private subnet. The load balancer is placed in the public subnet so it can receive traffic from users. The EC2 instances are placed in the private subnet for better security. Security groups are configured to allow only required ports like HTTP or SSH. IAM roles are created with limited permissions to avoid security issues.

We also configure Auto Scaling so that when traffic increases, more EC2 instances are created automatically, and when traffic decreases, extra instances are removed. This helps to save cost and maintain performance.

By using Terraform, the whole setup becomes automatic, easy to manage, and can be created again anytime without doing manual steps.

3.3. Infrastructure Design

3.3.1. VPC and Network Configuration

A Virtual Private Cloud (VPC) is created to isolate resources. Inside the VPC:

- Public Subnet is created for Load Balancer.
- Private Subnet is created for EC2 instances.
- Internet Gateway is attached to allow external communication.
- Route tables are configured for proper traffic flow.

This setup improves security by placing application servers inside private subnet. Inside the VPC, we create two subnets: one public subnet and one private subnet. The public subnet is used to place the Load Balancer because it needs to receive traffic from users through the internet. The private subnet is used to launch EC2 instances where the actual application runs. Keeping EC2 instances inside the private subnet improves security because they are not directly accessible from the internet.

An Internet Gateway is attached to the VPC so that the public subnet can communicate with the outside world. A route table is configured to allow internet traffic to flow through the Internet Gateway for the public subnet. For the private subnet, route tables are configured in such a way that direct internet access is restricted. Security groups are also attached to control inbound and outbound traffic. Only required ports like HTTP (80) or SSH (22) are allowed. Unnecessary ports are blocked to avoid security risks. This network design ensures better security, controlled traffic flow, and proper separation between public-facing components and internal application servers.

3.3.2. EC2 and Auto Scaling Configuration

EC2 instances are launched inside the private subnet.

To avoid overuse of resources:

- Auto Scaling Group is configured.
- Minimum and maximum instance limits are defined.
- Scaling policy is based on CPU utilization.

EC2 instances are launched inside the private subnet so that they are not directly accessible from the internet. This improves the security of the application because users cannot access the servers directly. The Load Balancer in the public subnet handles the incoming traffic and forwards it to the EC2 instances running in the private subnet. To manage resources efficiently and avoid unnecessary cost, an Auto Scaling Group is configured. Auto Scaling helps to automatically increase or decrease the number of EC2 instances based on the workload.

In this setup, minimum and maximum instance limits are defined. The minimum value ensures that at least one instance is always running so that the application remains available. The maximum value limits the number of instances to control cost and avoid over-provisioning. The scaling policy is based on CPU utilization. When CPU usage increases above a defined threshold, new EC2 instances are automatically launched to handle the extra load. When CPU usage decreases, extra instances are terminated automatically.

This helps to maintain performance during high traffic and reduce cost during low traffic periods. Health checks are also configured to monitor the status of instances. If any instance becomes unhealthy, Auto Scaling replaces it automatically with a new one. This improves reliability and ensures continuous availability of the application. Overall, this configuration makes the system scalable, cost-effective, and highly available without requiring manual monitoring and intervention.

3.3.3 Security Implementation

Security is an important part of this project because cloud resources can be exposed to risks if they are not configured properly. In this system, security is implemented at different levels to protect the infrastructure. First, IAM roles and policies are created using the principle of least privilege. This means users and services are given only the permissions that are required for their tasks, not full access. This helps to reduce the chances of misuse or accidental changes.

Security groups are configured to control inbound and outbound traffic. Only necessary ports such as HTTP (80) and SSH (22) are allowed, and all other unnecessary ports are blocked. SSH access is restricted so that only authorized users can connect to the EC2 instances. The EC2 instances are placed inside a private subnet, which means they are not directly accessible from the internet. All external traffic first goes through the Load Balancer in the public subnet.

Network isolation using VPC also adds another layer of protection. Route tables are configured properly to control how traffic flows between subnets and the internet. Regular monitoring is done to check instance health and activity. By applying these security measures, the system becomes more

secure, controlled, and protected from common cloud misconfigurations.

We also placed EC2 instances inside the private subnet. Because of this, they are not directly connected to the internet. All incoming traffic first goes to the Load Balancer in the public subnet, and then it forwards traffic to EC2 instances. This adds an extra layer of protection. Route tables are configured properly to control traffic flow between subnets. Internet Gateway is attached only for public subnet access. Private subnet does not allow direct internet communication.

Health monitoring and logging are also enabled to track activity in the system. This helps to detect unusual behavior and take action if required. Overall, by using IAM control, security groups, private subnet placement, and proper routing configuration, we improved the security of the system and reduced the chances of cloud misconfiguration issues.

3.4. Data Pre-Processing/ Configuration Preparation

Before deploying infrastructure, the following preparation steps are performed:

1. Install Terraform and configure AWS CLI.
2. Set up AWS credentials securely.
3. Define variables in Terraform for reusable configuration.
4. Organize files into modules for better structure.

Terraform files include:

- main.tf
- variables.tf
- outputs.tf
- provider.tf

This structured format improves maintainability. Before deploying the infrastructure, some basic preparation steps are completed. First, Terraform is installed in the local system and AWS CLI is configured so that it can connect to the AWS account. After that, AWS credentials are set up securely to allow Terraform to create and manage resources in the cloud. Then, variables are defined in Terraform files so that the configuration can be reused easily without changing the full code every time. For better organization, the Terraform files are arranged properly and divided into different sections.

Important files such as main.tf, variables.tf, outputs.tf, and provider.tf are created to manage the configuration in a structured way. This organized setup makes the project easier to understand, update, and maintain in the future. We also organize the files in a proper folder structure so that the project looks clean and understandable. This structured approach makes the infrastructure easy to update, reuse, and maintain in the future. If any changes are required later, we can modify the code and apply it again without doing manual work in the AWS console.

3.5. Implementation Steps

Steps1: Initialize Terraform

terraform init

Steps2: Validate Configuration

terraform validate

Steps3: Plan Deployment

terraform plan

Steps4: Apply Configuration

terraform apply

Steps5: Monitor Infrastructure

- Check EC2 instance health.
- Verify Load Balancer routing.
- Monitor CPU usage for scaling.

3.6. Data Split (Testing Approach)

To evaluate performance:

- Manual Deployment approach is compared with Automated Deployment.
- Deployment time is measured.
- Error rate is observed.
- Cost estimation is compared.

To evaluate the performance of the proposed system, we compared the manual deployment method with the automated deployment using Terraform. In the manual method, resources were created step by step using the AWS console, which required more time and careful configuration. In the automated method, the same infrastructure was created using Terraform commands. We measured the total time taken in both approaches to see the difference.

We also observed the number of configuration mistakes that happened during manual setup, such as missing rules in security groups or wrong resource settings. In the automated approach, since everything was written in code, the chances of errors were less. Cost estimation was also compared in both cases. We checked how Auto Scaling and proper configuration helped in reducing unnecessary resource usage and saving money. Based on this comparison, we analyzed which approach is more efficient, reliable, and cost-effective.

3.7. Proposed Architecture Components

The system includes:

- VPC
- Public Subnet
- Private Subnet
- EC2 Instances
- Load Balancer
- Auto Scaling Group
- IAM Roles
- Security Groups

The system includes several important AWS components that work together to create a complete and secure cloud infrastructure. First, a VPC is created to provide a private network environment in the cloud. Inside the VPC, two subnets are configured: a public subnet and a private subnet. The public subnet is used to place the Load Balancer so that it can receive traffic from users through the internet. The

private subnet is used to launch EC2 instances where the application runs securely without direct internet access.

EC2 instances are the main servers that host the application. An Auto Scaling Group is configured to automatically increase or decrease the number of EC2 instances based on traffic demand. This helps to maintain performance and also reduce cost when traffic is low. IAM roles are created to manage permissions and provide secure access to AWS services without sharing credentials. Security groups are used to control inbound and outbound traffic by allowing only required ports and blocking unnecessary access. All these components work together to make the system secure, scalable, and easy to manage.

A Load Balancer is used to distribute incoming traffic evenly across multiple EC2 instances. This prevents one server from getting overloaded and improves availability. If one instance fails, the Load Balancer automatically sends traffic to other healthy instances. IAM roles are created to manage permissions securely. Instead of giving full access, only required permissions are assigned. This improves security and reduces the risk of misuse. Security groups are configured as virtual firewalls.

They control which type of traffic is allowed to enter or leave the instances. Only necessary ports like HTTP and SSH are allowed, and all other ports are blocked. All these components work together to create a system that is secure, scalable, reliable, and cost-effective. The design ensures proper separation of resources, controlled access, and automatic scaling based on demand.

3.8. Proposed Architecture Components

Input: AWS cloud environment requirements
Output: Automated, secure, and scalable infrastructure

Strategy:

Step 1. Define infrastructure requirements

Step 2. Create Terraform configuration files

Step 3. Configure networking (VPC, Subnets, Route Tables)

Step 4. Configure compute resources (EC2, Auto Scaling)

Step 5. Apply security policies (IAM, Security Groups)

Step 6. Initialize Terraform

Step 7. Validate and Plan configuration

Step 8. Apply infrastructure deployment

Step 9. Monitor performance and scaling

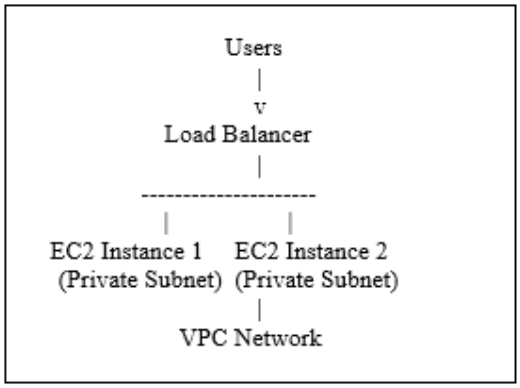
Step 10. Analyze cost and security improvements

4. Research Methodology

Terraform and AWS services are used to design and deploy cloud infrastructure. The main aim is to automate infrastructure creation and compare it with manual deployment. The system is tested based on deployment time, error reduction, scalability, and cost efficiency. The infrastructure is created using Terraform configuration files and deployed using Terraform commands. After deployment, performance is observed by monitoring instance behavior, scaling activity, and overall system availability.

4.1System Architecture

The architecture of the proposed system is shown below:



In this architecture:

- Users send requests through the internet.
- Load Balancer receives the traffic.
- Traffic is distributed to EC2 instances.
- EC2 instances are placed inside private subnet.
- All components are inside a VPC.

This design improves security and scalability. The system architecture of this project is designed to create a secure, scalable, and automated AWS infrastructure using Terraform. In this architecture, all resources are placed inside a Virtual Private Cloud (VPC), which acts as a private network in the cloud. Inside the VPC, two subnets are created: one public subnet and one private subnet. The Load Balancer is placed in the public subnet because it needs to receive traffic from users through the internet. The EC2 instances are launched inside the private subnet so that they are not directly exposed to the internet, which improves security. When a user sends a request, it first reaches the Load Balancer. The Load Balancer then distributes the traffic evenly to the available EC2 instances. An Auto Scaling Group is configured to automatically increase or decrease the number of EC2 instances based on CPU usage. This helps the system handle high traffic efficiently and reduce cost during low traffic. Security groups and IAM roles are also configured to control access and protect the infrastructure. Overall, the architecture ensures proper traffic flow, better security, automatic scaling, and easy management through Terraform automation.

4.2. Tools and Technologies

In this project, the following tools are used:

- AWS Cloud Platform
- Terraform
- AWS CLI
- EC2
- VPC
- Auto Scaling
- Load Balancer
- IAM
- Security Groups

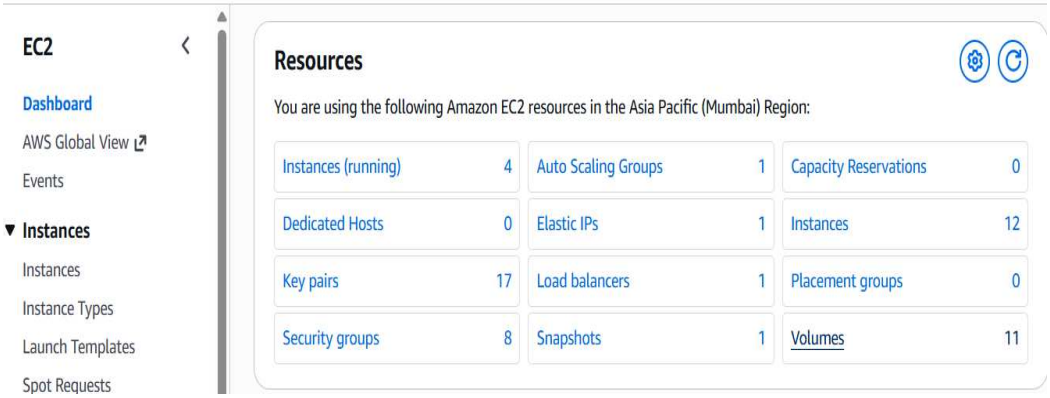
Terraform is used to write infrastructure as code. AWS CLI is used to connect local system with AWS account.

4.3Implementation Process

*Steps1:*Infrastructure Design

First, we planned the infrastructure components such as:

- VPC creation
- Public and Private subnets
- Internet Gateway
- Route tables
- EC2 instances
- Auto Scaling Group
- Load Balancer
- IAM roles



4.3.1 fig: Infrastructure Design

Steps2: Terraform Configuration

Terraform files are created:

- provider.tf → to define AWS provider and region
- main.tf → to define infrastructure resources
- variables.tf → to define reusable variables
- outputs.tf → to display output values

```
ubuntu@ip-172-31-6-180:~$ cd terr1
ubuntu@ip-172-31-6-180:~/terr1$ ls
ec2.tf  provider.tf  terraform.tfstate  terraform.tfstate.backup
ubuntu@ip-172-31-6-180:~/terr1$ cd
ubuntu@ip-172-31-6-180:~$ cd terraform-demo
ubuntu@ip-172-31-6-180:~/terraform-demo$ ls
main.tf  modules  output.tf  provider.tf  terraform.tfstate  terraform.tfstate.backup  variables.tf
ubuntu@ip-172-31-6-180:~/terraform-demo$ cd modules
ubuntu@ip-172-31-6-180:~/terraform-demo/modules$ ls
alb  asg  ec2  vpc
ubuntu@ip-172-31-6-180:~/terraform-demo/modules$ cat main.tf
cat: main.tf: No such file or directory
ubuntu@ip-172-31-6-180:~/terraform-demo/modules$ cd ..
ubuntu@ip-172-31-6-180:~/terraform-demo$ cat m
main.tf  modules/
ubuntu@ip-172-31-6-180:~/terraform-demo$ cat main.tf
module "ec2_instance" {
  source = "./modules/ec2"

  ami_id        = var.ami_id
  instance_type = var.instance_type
  instance_name = var.instance_name
}

module "vpc" {
  source = "./modules/vpc"
}

module "alb" {
  source     = "./modules/alb"
  vpc_id     = module.vpc.vpc_id
  subnet_ids = module.vpc.public_subnet_ids
}
```

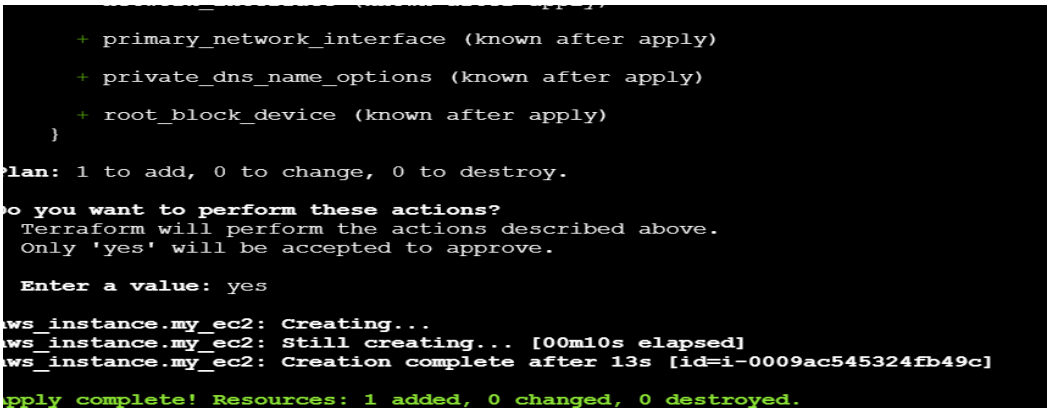
4.3.2 fig: Terraform Configuration

*Steps3:*Deployment Process

The following commands are executed:

1. terraform init
2. terraform validate
3. terraform plan
4. terraform apply

After applying, infrastructure is created automatically.



4.3.3 fig: Deployment Process

4.4. Performance Evaluation Method

To evaluate the system, we compared:

1. Manual deployment using AWS Console
2. Automated deployment using Terraform

We measured:

- Total time required for deployment
- Number of configuration errors
- Scalability performance
- Estimated monthly cost

To check the performance of our system, we compared manual deployment with automated deployment using Terraform. In manual deployment, we created all AWS resources step by step from the AWS console. In automated deployment, we used Terraform files and commands to create the same infrastructure automatically. We checked how much time was taken in both methods. We also observed if any configuration mistakes happened during manual setup. Then we tested Auto Scaling by increasing traffic and checking whether new EC2 instances were created automatically. When traffic was low, we checked if extra instances were removed .We also compared the estimated cost in both cases to see how automation helped in saving money. Based on time, errors, scaling, and cost, we analyzed which method works better.

Comparison Diagram

Manual Deployment

|

|-- More Time

|-- Higher Chance of Error

|-- Difficult to Repeat

|-- Manual Scaling

Automated Deployment (Terraform)

|

|-- Faster Deployment

|-- Less Errors

|-- Easy to Repeat

|-- Auto Scaling Enabled

4.5. Monitoring and Result Observation

After deployment, the system was monitored using:

- EC2 CPU utilization
- Auto Scaling activity
- Load Balancer health check
- Instance availability

When CPU usage increased above threshold, new instances were launched automatically. When traffic decreased, extra instances were removed. This confirmed that Auto Scaling was working properly.

4.6. Result Analysis

From testing and comparison, the following results were observed:

- Deployment time was reduced significantly using Terraform.
- Manual errors were minimized.
- Infrastructure could be recreated easily.
- Auto Scaling helped in cost optimization.
- Security configuration was more consistent.

The automated system performed better compared to manual setup

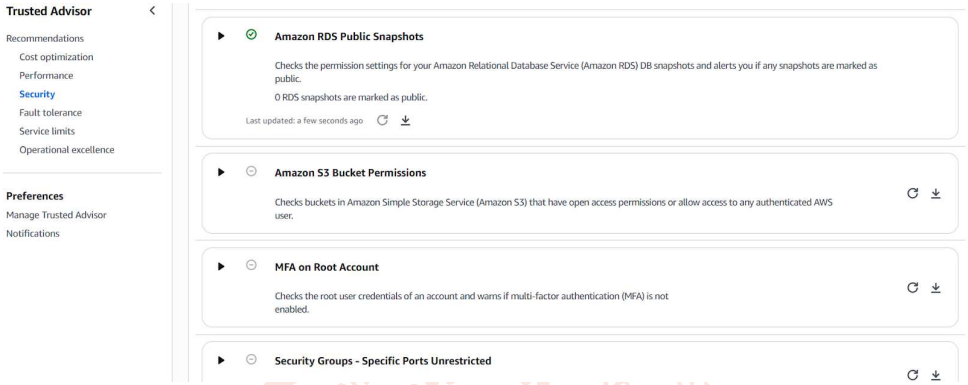


Fig 4.6.1 Security purpose

5. Conclusion and Future work

In this project, we implemented infrastructure automation on AWS using Terraform. The main aim was to reduce manual work and make the deployment process faster and more reliable. We created a structured cloud architecture including VPC, public and private subnets, EC2 instances, Load Balancer, Auto Scaling Group, IAM roles, and Security Groups. By using Terraform scripts, the complete infrastructure was deployed automatically with proper configuration.

After comparing manual deployment with automated deployment, we observed that automation reduces deployment time and minimizes configuration errors. It also helps in maintaining consistency because the same code can be reused multiple times. Auto Scaling improved system performance during high traffic and helped reduce cost during low usage. Overall, automation made the system more efficient, scalable, and secure.

In future work, this project can be extended by integrating CI/CD pipelines for fully automated application deployment. More monitoring tools like CloudWatch can be added for better performance tracking. Security can also be improved by implementing advanced IAM policies and encryption techniques. Multi-region deployment can be explored to increase availability and fault tolerance.

We designed a proper cloud architecture that includes VPC, public and private subnets, EC2 instances, Load Balancer, Auto Scaling Group, IAM roles, and Security Groups. This architecture provides better security by keeping application servers inside the private subnet and controlling access using security rules. The Load Balancer helps in distributing traffic evenly, and Auto Scaling ensures that new instances are created automatically during high traffic and removed during low traffic.

6. References

[1] HashiCorp, "Terraform Documentation," Available: https://developer.hashicorp.com/terraform/docs

[2] Amazon Web Services, "Amazon VPC User Guide," Available: https://docs.aws.amazon.com/vpc/

[3] Amazon Web Services, "Amazon EC2 Documentation," Available: https://docs.aws.amazon.com/ec2/

[4] Amazon Web Services, "Auto Scaling Documentation," Available: https://docs.aws.amazon.com/autoscaling/

[5] B. Wittig and R. Wittig, *Amazon Web Services in Action*, Manning Publications, 2016.

[6] Y. Brikman, *Terraform: Up & Running*, O'Reilly Media, 2019.

[7] B. Burns, *Designing Distributed Systems*, O'Reilly Media, 2018.

[8] A. Gourav, "Infrastructure as Code using Terraform and AWS," *International Journal of Computer Applications*, 2021.

[9] HashiCorp, "Terraform AWS Provider Documentation," 2023.

[10] Amazon Web Services, "AWS Well-Architected Framework," 2023.

[11] Amazon Web Services, "AWS Cost Management Documentation," Available: https://docs.aws.amazon.com/awsaccountbilling/

[12] Amazon Web Services, "AWS Security Best Practices," Available: https://aws.amazon.com/architecture/security-identity-compliance/

[13] Amazon Web Services, "AWS Identity and Access Management (IAM) User Guide," Available: https://docs.aws.amazon.com/iam/

[14] Amazon Web Services, "AWS Trusted Advisor," Available: https://docs.aws.amazon.com/awssupport/latest/user/trusted-advisor.html

[15] Amazon Web Services, "AWS CloudTrail Documentation," Available: https://docs.aws.amazon.com/cloudtrail/

[16] Amazon Web Services, "AWS Config Documentation," Available: https://docs.aws.amazon.com/config/

[17] HashiCorp, "Terraform Security Best Practices," Available: https://developer.hashicorp.com/terraform/docs/security

[18] HashiCorp, "Terraform State Management," Available: https://developer.hashicorp.com/terraform/docs/language/state

[19] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud*, O'Reilly Media, 2016.

[20] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010.

[21] M. Rahman, A. Williams, and T. Sharma, "Characterizing Infrastructure as Code Scripts in DevOps," IEEE/ACM International Conference on Automated Software Engineering, 2018.

[22] A. Y. Ding et al., "Cloud Cost Optimization Techniques: A Survey," IEEE Access, 2020.

[23] Amazon Web Services, "AWS Well-Architected Framework – Cost Optimization Pillar," 2023.

[24] Amazon Web Services, "AWS Well-Architected Framework – Security Pillar," 2023.

[25] T. Sharma et al., "DevOps: Concepts and Challenges," IEEE Software, 2017.