**Project: Multi-Environment Scalable Web Application Infrastructure on AWS**

**Ensuring Scalability, Isolation, and Automation for Modern Web Applications**

**Submitted By – Heena**

**1. Introduction**

**1.1. Problem Statement**

Modern web applications require robust, scalable, and secure infrastructure that supports multiple isolated environments (development, staging, production). Manual provisioning leads to inconsistency, increased risk, and slow delivery cycles. There is a need for a fully automated, repeatable, and auditable approach to deploy, manage, and destroy AWS resources for each environment.

**1.2 Scope**

This project implements a multi-environment web application infrastructure on AWS. It uses Infrastructure as Code (Terraform) and DevOps automation (Jenkins) to provision, manage, and validate isolated environments for development, staging, and production. The solution ensures scalability, security, and operational efficiency.

**1.3 Objectives**

* Provision isolated AWS environments (dev, staging, prod) using reusable Terraform modules.
* Automate infrastructure deployment, update, and teardown using Jenkins.
* Store Terraform state securely in S3 with locking via DynamoDB.
* Enable seamless CI/CD for infrastructure changes.
* Host a static web application via EC2 and Application Load Balancer.
* Implement best practices for security, monitoring, and tagging.

**2. AWS Services Utilized: A Comprehensive Overview**

**2.1 Networking Services**

* **VPC (Virtual Private Cloud):**Provides isolated network environments for each environment.
* **Subnets:**Public and private subnets for resource segregation.
* **Internet Gateway & NAT Gateway:**Secure internet access for public and private resources.
* **Route Tables:**Controls traffic flow within the VPC.

**2.2 Compute Services**

* **EC2:** Hosts the static web application.
* **Auto Scaling Group:** Ensures scalability and high availability for EC2 instances.
* **Application Load Balancer (ALB):** Distributes traffic and performs health checks.

**2.3 Storage & Database Services**

* **RDS (MySQL):** Managed database service, deployed in private subnets.
* **S3:** Stores Terraform state files with versioning and encryption.

**2.4 Security and Identity Services**

* **IAM:** Manages user and service permissions, including roles for EC2 and RDS monitoring.
* **Security Groups:** Firewall rules for EC2, RDS, and ALB.

**2.5 Monitoring and Alert Services**

* **CloudWatch:** Collects logs and metrics, triggers alarms for resource health and scaling.
* **DynamoDB:** Used for Terraform state locking to prevent concurrent changes.

**3. DevOps Approach**

**3.1. Infrastructure as Code**

* All AWS resources are defined in modular Terraform scripts for repeatable, auditable deployments.
* Each environment (dev, staging, prod) has its own configuration and state.

**3.2. Continuous Integration & Delivery**

* **Jenkins pipeline** automates:
  + Pulling code from GitHub.
  + Running Terraform to plan, apply, or destroy infrastructure.
  + Parameterized builds for environment and action (plan, apply, destroy).
  + Manual approval for destructive actions.
  + Archiving outputs and exposing application URLs.
* **GitHub Actions** for code validation and CI.

**A diagram of a computer

AI-generated content may be incorrect.**

* + 1. Architectural Diagram of Multi-Environment Scalable Web Application Infrastructure on AWS.

**4. Detailed Services Implementation**

**4.1. Networking with VPC**

* Created a dedicated VPC for each environment with non-overlapping CIDR blocks.
* Set up public subnets (for ALB, NAT Gateway) and private subnets (for EC2, RDS).
* Configured route tables, internet gateways, and NAT gateways for secure connectivity.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. VPC, Subnets, Route tables and Internet & Network Gateway for Development environment.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Different VPCs with different CIDR blocks for each environment.

**4.2. Compute and Load Balancing**

* Deployed EC2 instances in private subnets using an Auto Scaling Group.
* Configured Application Load Balancer to distribute HTTP traffic and perform health checks.
* Used user data scripts to deploy a static HTML web app on each EC2 instance.

**A screenshot of a computer

AI-generated content may be incorrect.**

* + 1. One EC2 instance running for Development and two are running for Production environment simultaneously.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Two different Application Load Balancers for Development and Production environments.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. DNS, network settings, Listeners and rules for Application Load Balancer for Development environment.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Target Group running one healthy instance for Development Environment.

**4.3 Database and State Management**

* Deployed RDS MySQL in private subnets for database needs.
* Enabled backup retention and multi-AZ for production.
* Used S3 bucket for remote Terraform state storage, with versioning and encryption.
* Used DynamoDB for state locking.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. RDS deployed in private subnet of Development VPC to store Web-App data.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. S3 bucket storing “terraform.tfstate” file for every environment, allowing safe sharing and versioning of the infrastructure state for everyone working on the project.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. DynamoDB locking the Terraform state file, preventing multiple people or processes from making changes at the same time and ensuring the infrastructure state stays consistent

**4.4 Security and Monitoring**

* Created IAM roles for EC2 and RDS monitoring.
* Defined security groups to restrict access between ALB, EC2, and RDS.
* Enabled CloudWatch metrics and alarms for EC2, ALB, and RDS health.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. IAM roles created for EC2 and RDS monitoring in both development and production environments.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Security Groups for Production environment acting as virtual firewalls for controlling inbound and outbound traffic for EC2 instances and other AWS resources**.**

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. CloudWatch alarms monitoring CPU usage, database connections, and healthy hosts for different environments.

**4.5 CI/CD with Jenkins**

* Installed Jenkins locally and configured required plugins (AWS, Git, Pipeline).
* Added AWS credentials and DB password as Jenkins secrets.
* Created a parameterized Jenkins pipeline using a Jenkinsfile from the repo.
* Pipeline stages: checkout, setup, init, validate, plan, apply/destroy, health check, cleanup.
* Manual approval required for destructive actions.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Jenkins storing GitHub, AWS credentials as Jenkins secrets.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Build with Parameters option allows to build infrastructure in Development, Staging or Production environments.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Successful Jenkins runs.

**5. Testing & Validation**

**5.1 Infrastructure Validation**

* Verified all AWS resources were created via the AWS Console and Terraform outputs.
* Checked ALB DNS to confirm the web application was accessible.

**5.2 Health Check Validation**

* Ensured ALB health checks passed and EC2 targets were healthy.
* Simulated instance failure to confirm auto-scaling and recovery.

**5.3 CI/CD Validation**

* Triggered Jenkins builds with different parameters (dev, staging, prod; plan, apply, destroy).
* Confirmed correct deployment, update, and teardown of resources.

**6. Challenges & Solutions**

**6.1. Subnet CIDR Allocation Errors**

* **Error:**  
  InvalidSubnet.Range: The CIDR '10.0.1.0/24' is invalid.
* **Root Cause:**  
  Subnet CIDRs did not match the VPC CIDR block for each environment.
* **Solution:**  
  Updated subnet CIDRs to match each environment’s VPC (e.g., 10.1.x.0/24 for staging, 10.2.x.0/24 for prod) and used Terraform’s cidr subnet for dynamic allocation.

**6.2. Destroy Operation Did Not Remove All Resources**

* **Issue:**  
  Resources remained in AWS Console after destroy.
* **Diagnosis:**  
  Only the destroy plan was run, or dependencies/protection prevented deletion.
* **Solution:**  
  Ensured terraform apply destroy-plan was executed, checked for errors, and manually deleted stuck resources if needed.

**6.3. ALB Target Group Unhealthy**

* **Issue:**  
  ALB target group showed unhealthy targets.
* **Diagnosis:**  
  Health check path/port mismatch or web server not running.
* **Solution:**  
  Verified health check settings, security group rules, and ensured the web server was running and accessible.

**6.4. Jenkins Pipeline Directory Issues**

* **Issue:**  
  Jenkins pipeline failed due to missing Terraform files.
* **Diagnosis:**  
  Incorrect working directory in Jenkinsfile.
* **Solution:**  
  Added debug steps, confirmed directory structure, and ensured all files were present.

**7. Lessons Learned**

* Gained practical experience with AWS multi-environment architecture and DevOps automation.
* Learned the importance of remote state management and state locking for team safety.
* Understood the value of modular Terraform code for reusability and scalability.
* Improved troubleshooting skills by resolving real AWS and CI/CD pipeline errors.

**8. Future Scope**

* Integrate more advanced monitoring (e.g., AWS X-Ray, CloudTrail).
* Add automated rollback and blue/green deployment strategies.
* Implement cost optimization (e.g., spot instances, reserved RDS).
* Use private subnets and bastion hosts for enhanced security.
* Expand to multi-region or disaster recovery architectures.

**9. Conclusion**

This project demonstrates how to build a robust, automated, and scalable cloud infrastructure using AWS, Terraform, and Jenkins. By supporting multiple isolated environments and leveraging CI/CD automation, the solution enables rapid, safe, and repeatable deployments—meeting the needs of modern development and operations teams.

**10. Demo**

* **Application URL:** Application running at EC2 instances for each environment was accessible through its Application Load Balancer’s DNS name.

A computer screen shot of a code

AI-generated content may be incorrect.

* + 1. Outputs for Development environment with app accessible at URL: <http://multi-env-web-app-dev-alb-776371601.us-east-1.elb.amazonaws.com>

**A screenshot of a computer

AI-generated content may be incorrect.**

18. Application accessible through respective ALB’s DNS name.

**11. Appendix**

**11.1. Terraform Scripts**

* [Repository Link or File Paths]

**11.2. Jenkinsfile**

* [Insert or Link to Jenkinsfile]

**11.3. Screenshots and Diagrams**

**[Leave space here for:**

* AWS Architecture Diagram
* Console screenshots of VPC, EC2, ALB, RDS, S3, DynamoDB, CloudWatch, Jenkins pipeline, and application running  
  **]**