**Project Report**

**Title:**

**Automated Web Application Deployment Using Terraform and AWS CI/CD Stack**

**1. Introduction**

In the current landscape of cloud-native applications and DevOps practices, automation of infrastructure and application deployment has become essential. This project focuses on creating a complete and automated CI/CD pipeline for deploying a Dockerized web application using AWS services. It leverages Infrastructure as Code (IaC) principles through Terraform, and implements automated build, test, and deployment workflows using AWS CodePipeline, CodeBuild, and CodeDeploy. The solution adheres to Site Reliability Engineering (SRE) principles, such as idempotency, rollback support, and observability, to ensure that deployments are reliable, repeatable, and resilient.

**2. Objectives**

The main objectives of the project are to automate infrastructure provisioning using Terraform, containerize a sample web application using Docker, implement a robust CI/CD pipeline using AWS native tools, and embed SRE best practices. The project aims to deliver a scalable, high-availability system that can be deployed with minimal manual intervention and rolled back in case of failures.

**3. Tools & Technologies Used**

This project uses Terraform for provisioning infrastructure, Docker for containerizing the application, and a combination of AWS CodePipeline, CodeBuild, and CodeDeploy to implement CI/CD. Monitoring and logging are handled by AWS CloudWatch, and other supporting AWS services such as EC2, ALB, S3, IAM, VPC, ECR, and NAT Gateway are also utilized to build a complete deployment environment.

**4. Application Overview**

The application deployed in this project is a basic Python Flask-based web service. When accessed, it returns a static message saying "Hello from DevOps!". The application is containerized using Docker, allowing it to be portable and easily deployable on any compatible infrastructure. The container is built as part of the CI/CD pipeline and stored in Amazon Elastic Container Registry (ECR).

**5. Infrastructure Architecture**

The infrastructure is fully provisioned through Terraform. It includes a VPC with both public and private subnets across two availability zones to ensure high availability. An Internet Gateway and NAT Gateway are configured for routing internet-bound traffic. Routing tables are associated with each subnet appropriately.

For compute resources, an Auto Scaling Group (ASG) launches EC2 instances behind an Application Load Balancer (ALB). The ALB manages incoming traffic and performs health checks to route traffic to only healthy instances. IAM roles are created for EC2, CodeBuild, and CodeDeploy with minimal privileges to follow security best practices.

Artifacts generated during the build process are stored in an S3 bucket, while Docker images are pushed to an ECR repository. Logs from EC2 and the CI/CD process are routed to CloudWatch, and alarms are configured to monitor EC2 health and CPU usage.

**6. CI/CD Pipeline Design**

The CI/CD pipeline consists of three stages: Source, Build, and Deploy. In the Source stage, CodePipeline is triggered by a GitHub push or an S3 upload. It retrieves the application source code and Terraform configurations.

In the Build stage, CodeBuild authenticates with ECR, builds the Docker image, pushes the image to ECR, and then executes Terraform commands such as init, plan, and apply to provision or update infrastructure. A buildspec.yml file defines these steps and ensures logs are pushed to CloudWatch.

In the Deploy stage, CodeDeploy uses an AppSpec file and associated shell scripts to pull the latest Docker image and run it on EC2 instances. This process is entirely automated and monitored using the CodeDeploy agent installed on each instance.

**7. Rollback Strategy**

The deployment strategy includes automatic rollback support. If CodeDeploy detects that an updated application causes unhealthy states (e.g., failed ALB health checks), it automatically reverts to the last known good version. Manual rollback is also possible via the AWS Management Console. This ensures continuous service availability and minimal downtime during failures.

**8. Monitoring & Observability**

Monitoring and observability are handled through AWS CloudWatch. Application logs and build logs are routed to CloudWatch Logs for real-time analysis. CloudWatch Alarms monitor key performance metrics like CPU utilization and the health of ALB targets. These alerts can trigger SNS notifications or Lambda functions to take corrective actions.

**9. Deployment Triggering**

The deployment pipeline can be triggered automatically upon code push to GitHub, ensuring immediate feedback and delivery. Manual deployment can be initiated through the AWS CodePipeline console. Additionally, scheduled deployments can be configured using Amazon EventBridge to run deployments at specific times or intervals.

**10. Cleanup Instructions**

To remove all resources and avoid unnecessary costs, the Terraform destroy command can be executed. Additionally, any Docker images stored in ECR, CloudWatch log groups, and artifacts in S3 should be manually deleted if not required.

**12. Conclusion**

This project successfully demonstrates the implementation of an automated CI/CD pipeline using AWS native services and Terraform. It highlights best practices in DevOps and SRE by ensuring deployments are automated, observable, and resilient. The approach taken here is modular and scalable, and can be reused for more complex application deployments in production environments.