**Cloud Resume Challenge – Project Report**

by Akil Riaz

Table of Contents

[1. Introduction 3](#_Toc202013594)

[2. Project Overview 3](#_Toc202013595)

[3. Architecture Design 3](#_Toc202013596)

[4. Backend Implementation 4](#_Toc202013597)

[5. Frontend Implementation 4](#_Toc202013598)

[6. Infrastructure as Code (IaC) with Terraform 4](#_Toc202013599)

[7. CI/CD and GitHub Actions 5](#_Toc202013600)

[8. Security and Access Management 5](#_Toc202013601)

[9. Challenges and Resolutions 5](#_Toc202013602)

[10. Key Learnings 6](#_Toc202013603)

[11. Conclusion 6](#_Toc202013604)

# 1. Introduction

The Cloud Resume Challenge is a multi-service cloud engineering project designed to demonstrate hands-on experience with AWS services, Infrastructure as Code (IaC), CI/CD pipelines, automation, and modern web deployment practices. In this project, I built and deployed two versions of a resume website: one manually using the AWS Management Console and the other through a fully automated Infrastructure as Code (IaC) setup using Terraform.

# 2. Project Overview

This project aimed to create a personal resume website that counts and displays the number of site visits using a serverless backend. The website is hosted on AWS, styled with HTML, CSS, and JavaScript, and integrates a dynamic visitor counter powered by AWS Lambda and DynamoDB. Two deployments were created:

* Manual Deployment: Hosted at <https://akilriaz.xyz>
* Terraform-based IaC Deployment: Hosted at <https://iac.akilriaz.xyz>

# 3. Architecture Design

The architecture consists of the following components:

* S3 for static frontend hosting.
* CloudFront as CDN for low latency and HTTPS.
* Route 53 for DNS management.
* ACM to issue SSL certificates.
* Lambda for backend logic.
* DynamoDB as a NoSQL data store for visitor counts.
* API Gateway to expose the backend as a REST API with CORS enabled.
* IAM for fine-grained access control between services.
* GitHub Actions for CI/CD of both frontend and Terraform infrastructure.

Both versions use the same architecture. The only difference lies in how they are provisioned; one manually, the other programmatically via Terraform.

# 4. Backend Implementation

The backend uses AWS Lambda written in Python with the Boto3 library to interact with DynamoDB. The function handles a simple logic: retrieve the existing view count, increment it, and update the table. API Gateway exposes this logic via a REST API and includes CORS configuration to allow frontend access.

Key elements:

* Lambda Function: Python-based, connected to DynamoDB.
* DynamoDB: Stores view counts with viewer\_id and viewer\_count.
* IAM Roles and Policies: Used to grant the Lambda function permission to read/write to DynamoDB.
* API Gateway: Enables secure HTTP access with CORS headers.

# 5. Frontend Implementation

The frontend is built using standard web technologies:

* HTML for structure
* CSS for styling
* JavaScript to dynamically fetch and display the visitor count from the backend

The files are uploaded to an S3 bucket and distributed globally via CloudFront. Every change to HTML, CSS, or JS triggers an automatic upload and invalidation via GitHub Actions to ensure up-to-date content delivery.

# 6. Infrastructure as Code (IaC) with Terraform

Terraform was used to automate the provisioning of all cloud infrastructure for the IaC deployment. This includes:

* Creating and configuring S3 buckets
* Setting up CloudFront distributions with OAC
* Provisioning and validating ACM SSL certificates
* Registering Route 53 DNS records
* Deploying and integrating Lambda, API Gateway, and IAM policies
* Populating DynamoDB with initial records

The entire infrastructure is managed from the infra/ directory in the project, allowing for modular and maintainable code.

# 7. CI/CD and GitHub Actions

Two separate GitHub Actions pipelines were created:

1. Frontend CI/CD: Watches for changes in frontend files and triggers:
   * Terraform apply to provision frontend infra
   * Uploads updated files to S3
   * Invalidates CloudFront cache to reflect changes
2. Infrastructure CI/CD: Watches changes to .tf files and triggers:
   * terraform fmt, validate, plan, and apply steps
   * Uses environment variables and Github secrets for authentication

CI/CD ensures that only changed files are updated, improving deployment efficiency.

# 8. Security and Access Management

Security is enforced using:

* IAM roles to isolate access between Lambda, DynamoDB, and other services
* Bucket policies and CloudFront Origin Access Control to secure S3 content
* Least privilege principle applied to Lambda execution roles
* HTTPS enforced via CloudFront with SSL certificates from ACM

# 9. Challenges and Resolutions

a. Large .terraform Folder

* Problem: Local .terraform directory became over 700 MB, causing GitHub push issues.
* Solution: Added a .gitignore to exclude .terraform/ and re-initialized backend.

b. Manual vs Terraform Drift

* Problem: Conflicts between manually created resources and Terraform state.
* Solution: Manually imported resources or deleted and recreated using Terraform to ensure consistency.

c. CORS Configuration

* Problem: The API Gateway initially blocked frontend requests.
* Solution: Added correct Access-Control-Allow-Origin headers using Terraform’s cors\_configuration.

d. CloudFront Caching

* Problem: CloudFront served outdated JS files after updates.
* Solution: GitHub Actions now includes an invalidation step after each deploy.

e. IAM Permissions

* Problem: Lambda failed due to missing DynamoDB permissions.
* Solution: Created custom IAM policy with GetItem, PutItem, UpdateItem permissions and attached it properly.

# 10. Key Learnings

* Deep understanding of AWS services and how they integrate with each other in real-world workflows.
* Terraform proficiency, particularly in using modules, state management, and resolving drift.
* CI/CD best practices using GitHub Actions, YAML workflows, and efficient cache invalidation.
* Security-first design, applying the least privilege model via IAM and protecting static content through CloudFront.
* Hands-on experience with REST APIs, CORS policies, and automated deployments for scalable cloud-native apps.

# 11. Conclusion

This project was an excellent exercise in end-to-end cloud solution design. It covered all major aspects of cloud development: frontend, backend, infrastructure, automation, and version control. The dual deployment approach (manual vs IaC) allowed me to contrast cloud console provisioning with infrastructure as code and reinforced the benefits of automation and reproducibility. I am now confident in deploying, managing, and scaling full-stack cloud solutions using modern DevOps practices.