CI/CD pipelines

Documentation

1. Introduction

The project employs GitHub Actions for Continuous Integration and Continuous Deployment (CI/CD) to streamline the development and deployment processes for the serverless football merchandise store. Each component of the stack like backend Lambda functions, Terraform configurations, and the Vue.js frontend is automated with dedicated workflows, ensuring efficient updates and consistency across environments.

2. Story

This semester, I wanted to learn CI/CD pipelines to expand my skills and complement the automation I was already doing with Terraform. Automating deployments felt like the logical next step—why manually zip Lambda functions or sync website files when GitHub Actions can handle it for me?

Now, every change pushed to the main branch triggers workflows that build and deploy the website, update Lambda functions, and apply Terraform configurations. For example, website updates are automatically uploaded to S3, and the CloudFront cache is invalidated, ensuring smooth and consistent deployments without manual effort.

These pipelines have streamlined my workflow, saved time, and taught me how crucial automation is in modern development.

3. Lambda CI/CD

The pipeline's purpose is to automate the packaging and deployment of Python-based AWS Lambda functions triggered by changes to the main branch. This workflow ensures quick updates and consistency in deploying serverless backend code.

- **Key steps:**

1. **Checkout code** - Retrieves the repository contents for processing.
2. **Install AWS CLI** - Set up the AWS CLI for deployment commands.
3. **Install Dependencies** - Installs Python libraries (boto3, simplejson) required for packaging or Lambda functionality.
4. **Package Lambda Functions** - Compresses each Lambda function into a .zip file for deployment.
5. **Deploy Lambda Functions** - Uploads the packaged files to AWS using aws lambda update-function-code.

This workflow ensures centralized and consistent deployment of multiple Lambda functions, with updates in the main branch quickly reflected in the backend.

```yaml
name: 'CI/CD for Lambda backend code deployment'

on:
  push:
    branches:
      - 'main' # Pipeline will run only when code is pushed to the main branch
    paths:
      - 'lambdas/**' # Pipeline will run only when code in the lambdas directory is changed

jobs:
  deploy:
    name: 'Deploy Lambda Functions'
    runs-on: ubuntu-latest # Use the latest version of Ubuntu

    steps:
    - name: Checkout repository # Checkout the repository
      uses: actions/checkout@v4

    - name: Install AWS CLI # Install the AWS CLI
      run: |
        sudo apt-get update
        sudo apt-get install -y awscli

    - name: Install Dependencies # Install boto3 and simplejson libraries
      run: pip install boto3 simplejson

    - name: Package Lambda Functions # Create a zip file for each Lambda function
      run: |
        mkdir -p lambdas/packages
        for FILE in lambdas/*.py; do
            FUNCTION_NAME=$(basename "${FILE%.*}")
            zip -r lambdas/packages/${FUNCTION_NAME}.zip ${FILE}
        done
```

The pipeline will be only activated on the main branch pushes when files in the lambdas folder change. Then it runs a job on Ubuntu, checkouts repository, and installs AWS CLI and necessary Python libraries. After that, it creates a new folder for the packaged lambdas, and for every Python file in the lambdas folder, it zips into a newly created folder with a specific name according to the lambda settings defined in Terraform.

```yaml
  - name: Deploy Lambda Functions # Update the code of each Lambda function
    env: # Set AWS credentials and region as environment variables
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      AWS_REGION: 'eu-central-1'
    run: |
      for FILE in lambdas/packages/*.zip; do
          FUNCTION_NAME=$(basename "${FILE%.*}")
          aws lambda update-function-code \
            --function-name ${FUNCTION_NAME} \
            --zip-file fileb://${FILE} \
            --region $AWS_REGION      alissento, 17 hours ago • ci/cd changes
      done
```

After that, it updates every lambda function in AWS with new zip files.

3. Terraform CI/CD

The pipeline's purpose is to automate infrastructure provisioning and data population, ensuring reliability and reducing manual effort.

- **Key steps:**

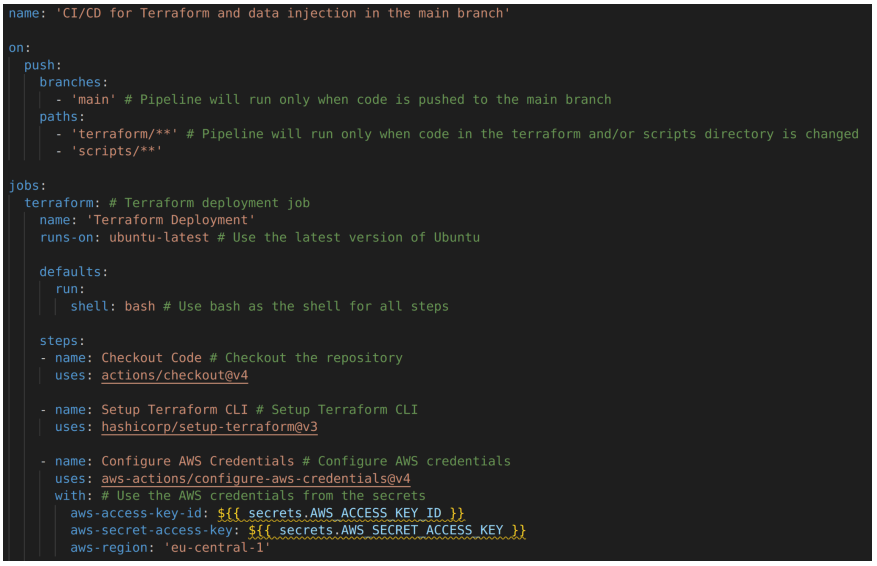
Terraform Deployment:

1. **Checkout code** - Retrieves Terraform files for processing.
2. **Setup Terraform** - Configures the Terraform CLI for use in the pipeline.
3. **AWS Credentials Configuration** - Authenticates with AWS to enable Terraform operations.
4. **Terraform Init** - Initializes Terraform, downloading required modules and providers.
5. **Terraform Format Check** - Ensures Terraform files are properly formatted.
6. **Terraform Plan & Apply** - Plans and applies infrastructure changes automatically.

Data injection:

1. **Setup Python** - Configures Python for running scripts.
2. **Install Dependencies** - Installs boto3 for AWS interactions.
3. **Run Data Injection Script** - Populates DynamoDB with data using a custom Python script.

The Terraform workflow automates both infrastructure provisioning and database initialization, providing consistent environment setups and reducing manual effort.



```yaml
name: 'CI/CD for Terraform and data injection in the main branch'

on:
  push:
    branches:
      - 'main' # Pipeline will run only when code is pushed to the main branch
    paths:
      - 'terraform/**' # Pipeline will run only when code in the terraform and/or scripts directory is changed
      - 'scripts/**'

jobs:
  terraform: # Terraform deployment job
    name: 'Terraform Deployment'
    runs-on: ubuntu-latest # Use the latest version of Ubuntu

    defaults:
      run:
        shell: bash # Use bash as the shell for all steps

    steps:
    - name: Checkout Code # Checkout the repository
      uses: actions/checkout@v4

    - name: Setup Terraform CLI # Setup Terraform CLI
      uses: hashicorp/setup-terraform@v3

    - name: Configure AWS Credentials # Configure AWS credentials
      uses: aws-actions/configure-aws-credentials@v4
      with: # Use the AWS credentials from the secrets
        aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
        aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        aws-region: 'eu-central-1'
```

This pipeline is only activated on the main branch pushes when files change in terraform and/or scripts folders. It uses Ubuntu and bash shell, checkouts repository and setups up terraforms and aws credentials.

```yaml
- name: Terraform Init # Initialize Terraform
  run: |
    cd terraform
    terraform init

- name: Terraform Format Check # Check if the Terraform code is formatted correctly
  run: |
    cd terraform
    terraform fmt -check -diff

- name: Terraform Plan # Create an execution plan
  run: |
    cd terraform
    terraform plan -input=false

- name: Terraform Apply # Apply the changes
  run: |
    cd terraform
    terraform apply -input=false -auto-approve
```

After that it initialises terraform, checks if terraform is formatted correctly, tests out with the terraform plan and finally applies infrastructure with terraform apply.

```yaml
data_injection: # Data Injection job
  name: 'Data Injection to DynamoDB'
  runs-on: ubuntu-latest # Use the latest version of Ubuntu
  needs: terraform # Wait for the Terraform deployment job to complete

  defaults:
    run:
      shell: bash # Use bash as the shell for all steps

  steps:
  - name: Checkout Code # Checkout the repository
    uses: actions/checkout@v4

  - name: Setup Python # Setup Python
    uses: actions/setup-python@v5
    with:
      python-version: '3.x' # Use Python 3.x

  - name: Install Dependencies # Install boto3 library
    run: pip install boto3

  - name: Configure AWS Credentials  # Configure AWS credentials
    uses: aws-actions/configure-aws-credentials@v4
    with: # Use the AWS credentials from the secrets
      aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
      aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      aws-region: 'eu-central-1'

  - name: Run Data Injection Script # Run the data injection script
    run: |
      cd scripts
      python3 data_injection.py
```

Second job in the pipeline. It is activating after the terraform job is done. It uses Ubuntu with shell as well, sets up Python environment, installs boto3, sets up AWS credentials and finally runs a data injection script that applies data to the DynamoDB.

4. Website Deployment

The pipeline's purpose is to automate the build and deployment process for the website hosted on AWS S3, with integrated cache invalidation for CloudFront.

- **Key steps:**

1. **Checkout Repository -** Retrieves the latest code from the main branch.
2. **Setup Node.js** - Configures the Node.js environment for building the Vue application.
3. **Set up Terraform CLI** - Prepares Terraform CLI for fetching CloudFront distribution details during deployment.
4. **Terraform Init** - Initializes Terraform to access outputs such as the CloudFront distribution ID.
5. **Install Dependencies** - Installs required Node.js packages like Vite or Tailwindcss.
6. **Build Website** - Executes both the Tailwind CSS build (npm run tailwindbuild) and Vue project build (npm run build) to prepare the website for deployment.
7. **Configure AWS Credentials** - Sets up AWS credentials for access to S3 and CloudFront.
8. **Deploy to S3** - Syncs the built website files to the S3 bucket, replacing outdated files.
9. **Fetch CloudFront Distribution ID** - Uses Terraform to extract the CloudFront distribution ID for cache invalidation.
10. **Invalidate CloudFront Cache** - Clears CloudFront cache to ensure the latest website version is served globally.

This workflow ensures that every update to the main branch on vue-website repository is automatically built, deployed, and propagated globally with minimal latency, maintaining a high-quality user experience.

```yaml
name: 'CD for website files in main branch'

on:
  push:
    branches:
      - 'main' # Pipeline will run only when code is pushed to the main branch
    paths:
      - 'vue-website/**' # Pipeline will run only when code in the vue-website directory is changed

jobs:
  website:
    name: 'Website deployment'
    runs-on: ubuntu-latest # Use the latest version of Ubuntu

    defaults:
      run:
        shell: bash # Use bash as the shell for all steps

    steps:
    - name: Checkout repository # Checkout the repository
      uses: actions/checkout@v4

    - name: Setup Node.js # Setup Node.js version 22
      uses: actions/setup-node@v4
      with:
        node-version: '22'
```

This pipeline activates only on the main branch pushes when files in vue-website change. It uses Ubuntu with shell, checkouts the repository and set up the Node.js environment.

```yaml
  - name: Set up Terraform CLI # Set up Terraform CLI
    uses: hashicorp/setup-terraform@v3

  - name: Terraform init # Initialize Terraform
    run: |
      cd terraform
      terraform init

  - name: Install dependencies # Install dependencies for the website
    run: |
     cd vue-website
     npm install

  - name: Build website # Build the website using npm run build command
    run: |
      cd vue-website
      npm run tailwindbuild
      npm run build

  - name: Configure AWS credentials # Configure AWS credentials for proper deployment
    uses: aws-actions/configure-aws-credentials@v4
    with:      You, 2 months ago • -created a python script to inject data to dyna…
      aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
      aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      aws-region: 'eu-central-1'
```

Then set up the terraform, initialise it and the goes to the website folder and installs all modules. After that, it builds tailwindcss file and then the whole website. Finally, it is setting up the AWS credentials necessary for proper deployment.

```yaml
- name: Deploy to S3 bucket # Deploy the website to the S3 bucket
  run: |
    cd vue-website
    aws s3 sync dist/ s3://nknez.tech/ --delete --region eu-central-1

- name: Fetch CloudFront distribution ID from Terraform output # Get the CloudFront distribution ID from the Terraform output
  run: echo "cloudfront_id=$(terraform output -raw cloudfront_distribution_id)" >> $GITHUB_ENV

- name: Invalidate CloudFront cache  # Invalidate the CloudFront cache to reflect the changes
  run: |
    cd terraform
    aws cloudfront create-invalidation --distribution-id ${{ env.cloudfront_id }} --paths "/*"
```

Then it syncs the dist folder files with the s3 bucket and deletes any outdated files. Then it gets CloudFront id form terraform outputs and finally, clears the cache in CloudFront distribution to reflect website changes on a live environment.

5. Conclusion

Building CI/CD pipelines with GitHub Actions has been a practical way to streamline deployments for my serverless football merchandise store. These workflows automate tasks like syncing website files, updating Lambda functions, and applying Terraform configurations, ensuring deployments are consistent and efficient. By removing repetitive manual steps, the pipelines make the development process smoother and more reliable, setting a strong foundation for scaling and maintaining the project in the future.