Created by: Bernardo Rodrigues

Metrics APP

DevOps Pipeline Manual

Contents

1. Purpose

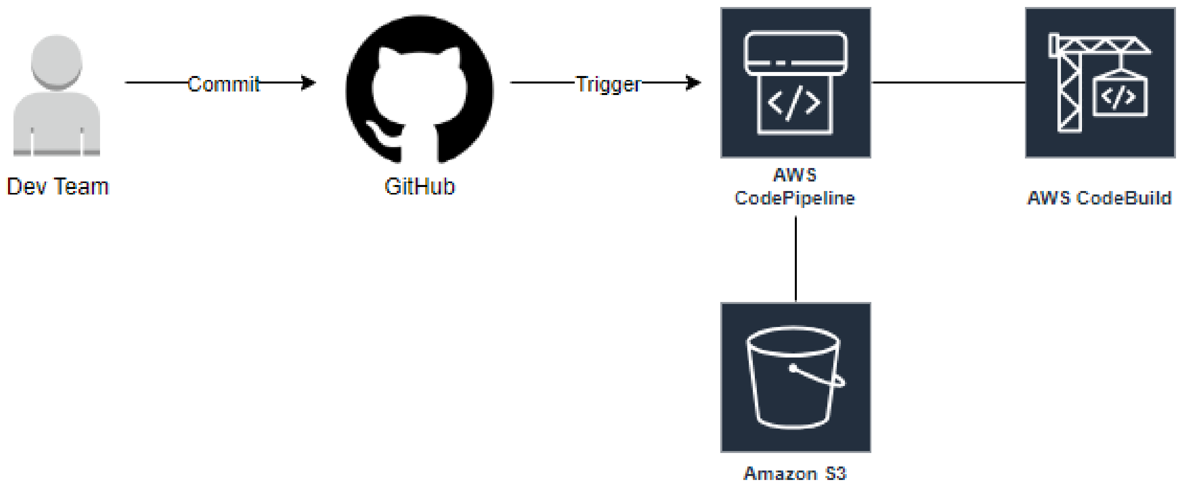
This application is designed to help the engineer track and record the activities performed for the different customer projects.

This document is focused on the development of a CI/CD pipeline to automate the implementation of the backend development and connectivity of the different services required to receive the information from an HTLM form, process the information and store the results.

2. Pipeline Design

The pipeline has, as a source, a GitHub repository where all the files to run the application, as well as the files to run CodeBuild correctly, are stored and modified with new updates from the Dev team.

CodePipeline is triggered when any commit and push actions are performed in the repository, allowing CodeBuild to fetch the files, prepared them and execute them using terraform to apply the necessary changes to the different AWS services in the application architecture.



In order to, proper integrate GitHub and CodePipeline, it is necessary to manually authenticate the connection between the two, within the AWS management console for the CodePipeline service.

Regarding CodeBuild specifically it is important to guarantee that its role has the necessary policies attached to it to execute the changes performed on the repository.

Finally, two Amazon S3 Buckets are used to store CodeBuild artifacts and the terraform state files for the application architecture.

3. Application Design

The main decision in the design phase of the application was deciding if it would be deployed in a traditional architecture based on EC2 VMs or a serverless architecture.

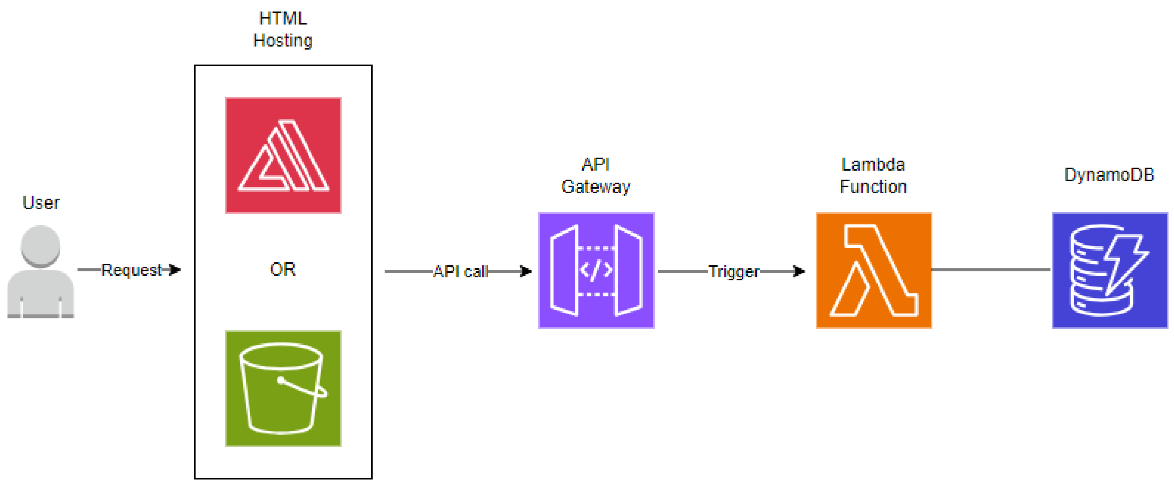
The EC2 design will give the maximum control and customization possible to develop the different layers of the application, with the downside of having several EC2 VMs always active to guarantee high availability and 24/7 service.

On the other hand, the serverless architecture, based on Lambda, would lose the control over the underlying infrastructure but gain in simplicity and cost effectiveness. Since it would only be necessary to add triggers trough an API to run the Lambda function when need it and only pay by the time that the function runs.

Due to the simplicity and use-case of the application the best option is the serverless architecture, showed below.

3.1 Architecture

The graph bellow shows the AWS services used and the flow of information between the different services



On the frontend both amplify service or S3 bucket, with static website hosting turn on, can be used to host the HTML form. For the ease of setup, the Amplify services will be used.
This HTML interface will make an API call to perform a POST request, to trigger the Lambda function to read the information HTML form, treat and save it to a database for record keeping.

3.2 API Gateway

In the design of the API Gateway, it is important to highlight that enabling CORS is a necessity to allow the frontend (Amplify service) to make the API call to the backend (lambda Function). Along side the permissions for the API to invoke the lambda function.

3.3 Lambda Function

The Lambda function is designed run a Python script that can read the information from the HTML form and prepare it to be stored inside of a NoSQL database. For this purpose, a runtime of python 3.12 was selected and the proper IAM permissions were created.

A specific IAM role is created, allowing the Lambda function itself to assume it, in order to run lambda functions and have full access to the DynamoDB database. These permissions are given by attaching the appropriate policies to the role.

3.4 DynamoDB

DynamoDB as a NoSQL database service is the best choice, over a SQL one, due to the structure of the data being easily mapped to key/value pairs and do not have significant relationships between them to justify the complexity of developing a schema and the cost involved with a full SQL service.

The DynamoDB table is provisioned with the default values of 5 for both write and read through put, since it is not expected to have a high volume of traffic/requests.

4. Pre-requisites

Before deploying the pipeline make sure that Terraform and AWS CLI is installed and correctly configured with the proper credentials.

Inside the variables.tf file there is 4 different variables to change to fit your specific information. Also modify the terraform backend configuration inside the DevOps_app_infra.tf file to your S3 bucket holding the terraform state files.

5. Application changes

	Files	Changes
V1	buildspec.yml DevOps_app_infra.tf metric_app.py	YAMAL file for CodeBuild instructions. Terraform file creating the Lambda function Python code for the Lambda function
V2	metric_app.py	Updated Lambda function code integrating Lambda with DynamoDB table
V3	DevOps_app_infra.tf	Terraform with the full application architecture

6. Deployment

1. Create your GitHub repository
2. Upload the V1 files to it
    - buildspec.yml
    - DevOps_app_infra.tf
    - metric_app.py
3. Update the variables.tf file
4. Initiate the Terraform plugin by running:

    terraform init
5. Simulate the pipeline deployment by:

    terraform plan
6. If the plan gives the green light, run:

    terraform apply
7. In the AWS management console authenticate the connection to GitHub

8. Push to the repository the changes for the different versions