National College of Ireland

Configuration Manual

MSc Research Project
Cloud Computing

Paris Moore
Student ID: x14485758

School of Computing
National College of Ireland

Supervisor: Horacio Gonzalez-Velez

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Paris Moore
Student ID:	x14485758
Programme:	Cloud Computing
Year:	2021-22
Module:	MSc Research Project
Supervisor:	Horacio Gonzalez-Velez
Submission Due Date:	15th Aug 2022
Project Title:	'Continuous Benchmarking' in DevOps to support Quality of Deployments using Amazon Web Services
Word Count:	2042
Page Count:	18

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Paris Moore
Date:	15th September 2022

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

'Continuous Benchmarking' in DevOps to support Quality of Deployments using Amazon Web Services

Paris Moore
x14485758

1 Introduction

This configuration manual will help its readers to understand the system requirements, setup, software and install specifications that were used in this research. Also, this manual includes detailed explanation of the steps needed to follow when implementing this research project. The proof-of-concept pipeline is designed and built using Amazon Web Services' (AWS), which automatically deploys the system release, runs one or more benchmarks, collects and analyzes results, and decides whether the release fulfils predefined Quality of Deployment (QoD) goals.

Prerequisites

- AWS Cloud Knowledge
- Basic Command Line Knowledge
- AWS IAM access to create roles and add policies to users
- Have Python installed to be able to run the application locally.
- Have Git installed locally.

2 Before you begin

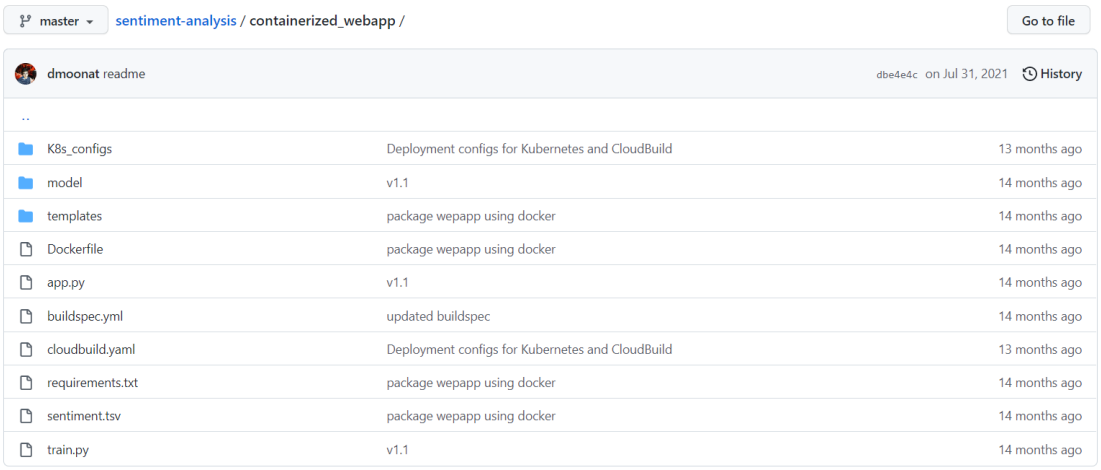
2.1 Software Installation

- Install AWS CLI locally following the steps for your Operating System
  https://docs.aws.amazon.com/cli/v1/userguide/install-windows.html
- Install Docker locally following the steps for your Operating System
  https://docs.docker.com/get-docker/

2.2 The Application

An application is required for analysis purposes and to test deployments on the pipeline. An open source Twitter Sentiment Analysis application is chosen. The app is built on the python flask framework with machine learning models developed to perform the predictions. The application will be containerized using docker and deployed using AWS Elastic

Container Service (ECS). Inside directory 'sentiment-analysis/containerized_webapp/' in the Application source file is a .Dockerfile and buildspec.yml files which will be very important when deploying the application on AWS Cloud.



GitHub Repository to download or clone application code:
https://github.com/dmoonat/sentiment-analysis/tree/master/containerized_webapp

Once the application has been downloaded and saved. The application can be ran locally by running the following commands in the 'sentiment-analysis/containerized_webapp/' directory.

- pip install -r requirements.txt
- python app.py

3 Deploying the Application on AWS

3.1 Docker

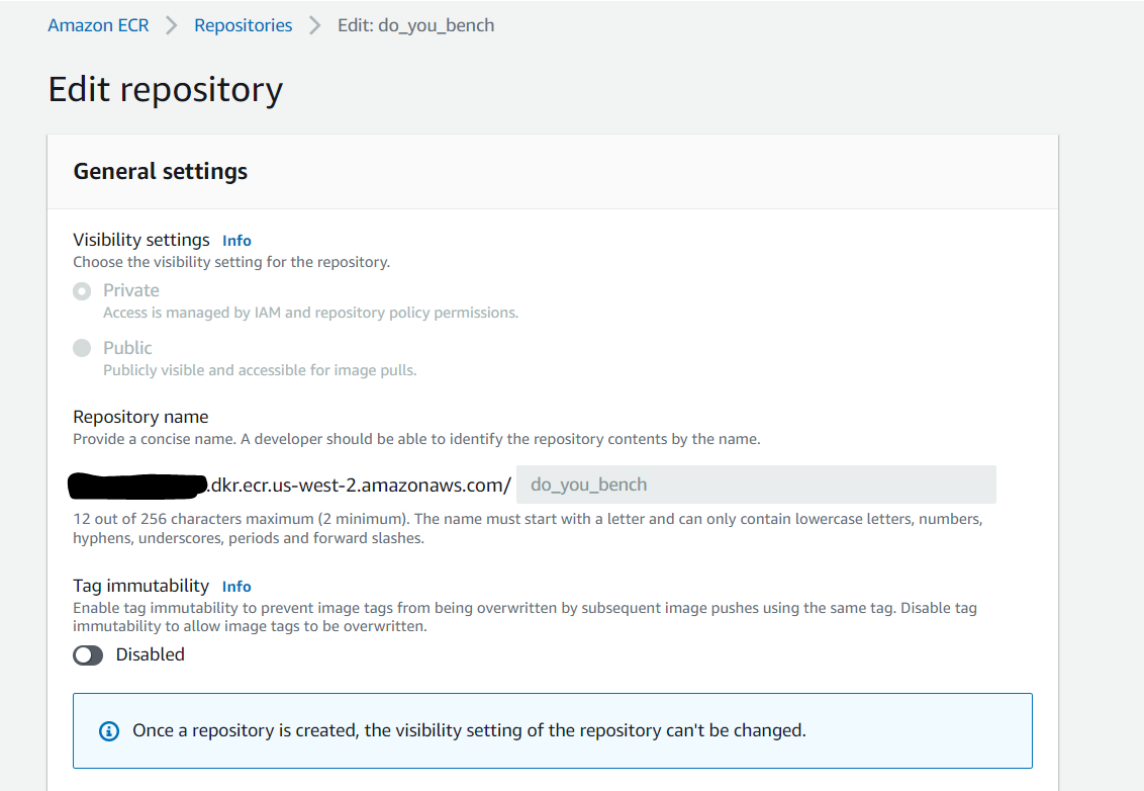
Once you have installed Docker, you can verify the installation is successful by running the 'docker' command on the terminal/command prompt. Its time to dockerize the Sentiment Analysis Application. The dockerfile already includes the necessary commands for docker to build an image.

- docker build -t final_project_2022:v1 -f Dockerfile .
- Check an image has been built using the 'docker images' command
- Test the image by running it using this command:
  docker run -p 5000:5000 -t -i final_project_2022:v1
- Go to the IP of the docker displayed to check that the app is running on a container.

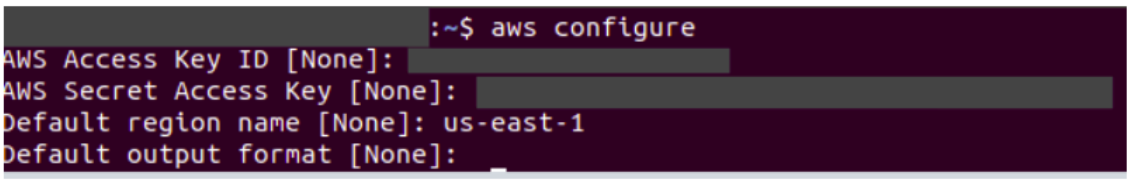
3.2 Elastic Container Registry

AWS ECR is a registry to store and manage container images, similar to DockerHub.

- Login to AWS management console → ECR
- Create a repository for the application



- To communicate with Amazon services from your local machine, you must install the AWS CLI and configure your profile using the terminal. For NCI students, this must be done via SSO. For those with admin access, this can be achieved by generating secret access keys within your AWS account.



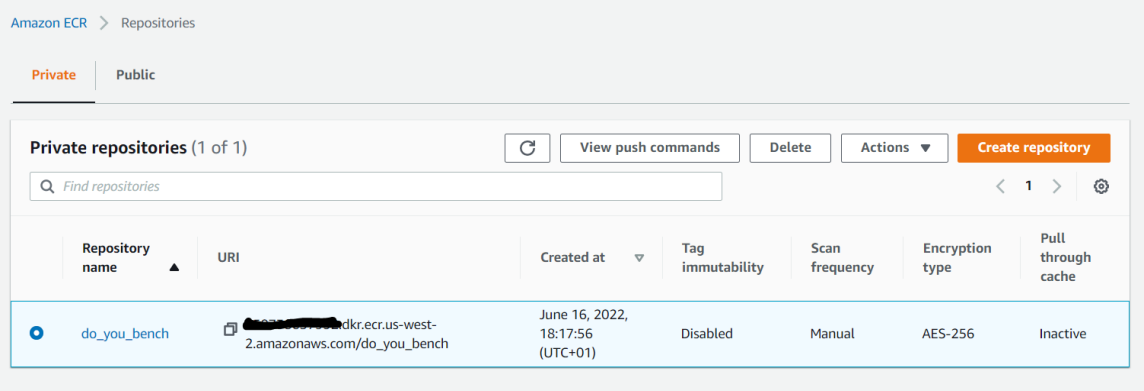
- Once AWS is configured, next you need to authenticate to your ECR repo using this command with your 'region' 'profile' and 'aws account ID' defined:

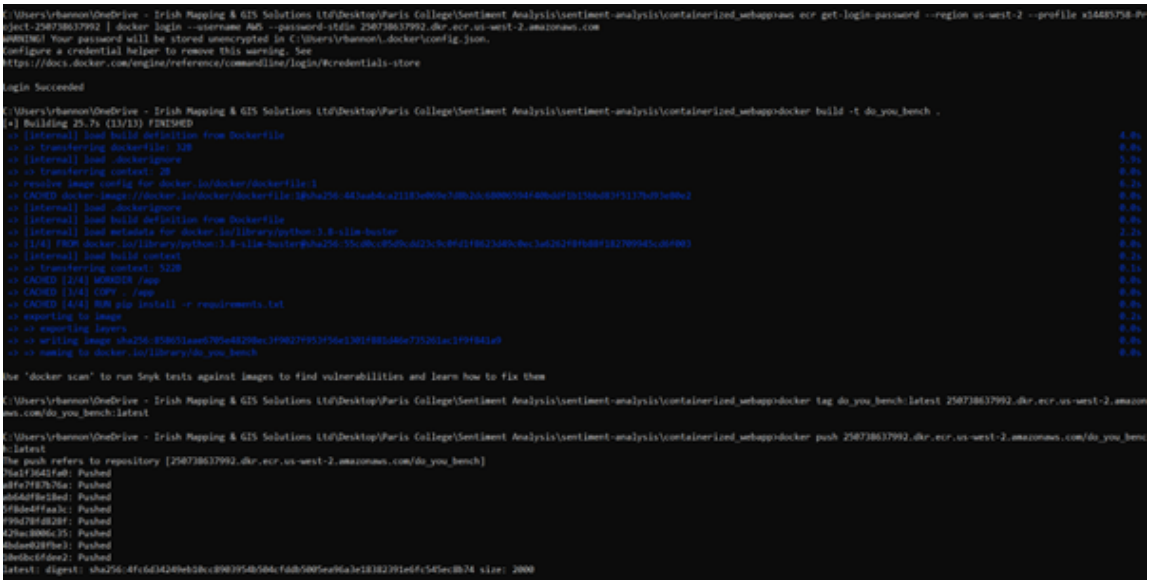
```
aws ecr get-login-password --region 'region' --profile 'profile'
| docker login --username AWS --password-stdin 'aws\_account\_id'.dkr.ecr.'region'.amazonaws.com
```

Further supporting documentation can be found here:
https://docs.aws.amazon.com/AmazonECR/latest/userguide/getting-started-cli.html

- Once Authenticated, follow the ECR push commands from your local terminal to upload a docker image to the repo.

- You can confirm that your image has been pushed by checking the ECR console.

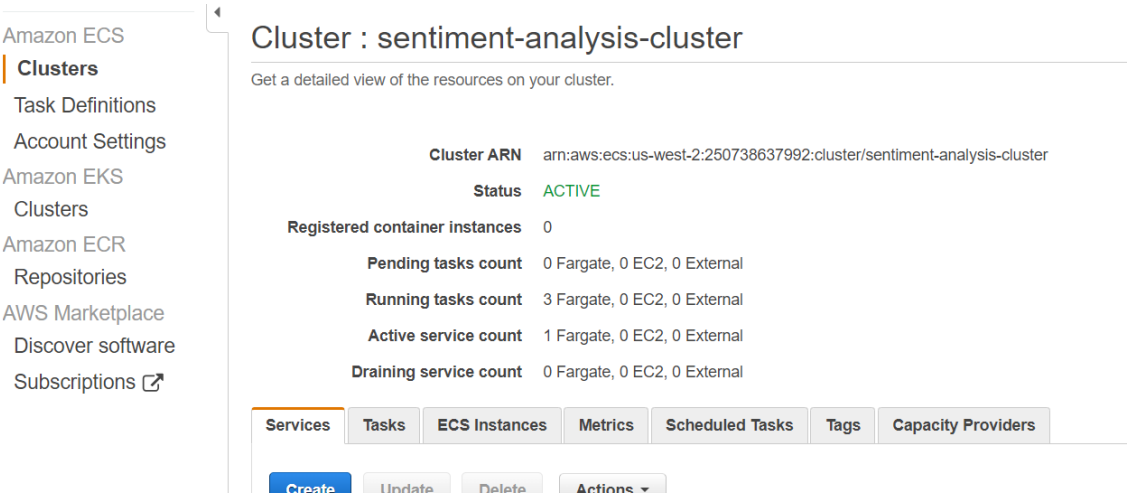




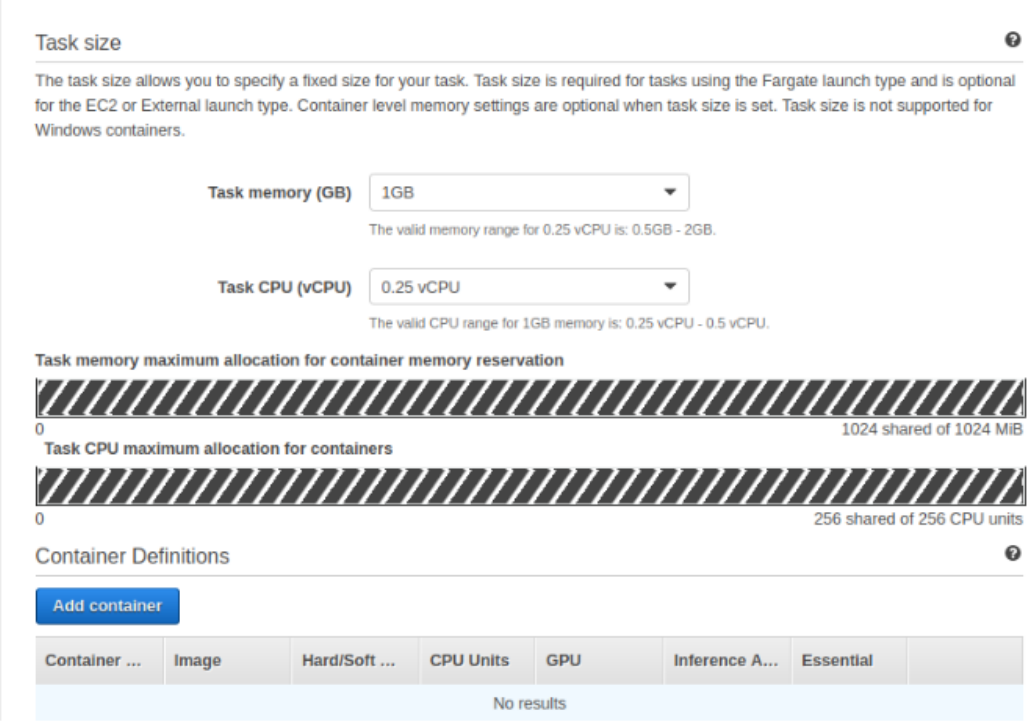
3.3 Elastic Container Service

Amazon ECS is a container orchestration platform developed by Amazon, that helps schedule and orchestrate containers across a group of servers. The two major ECS components are Tasks and Services.

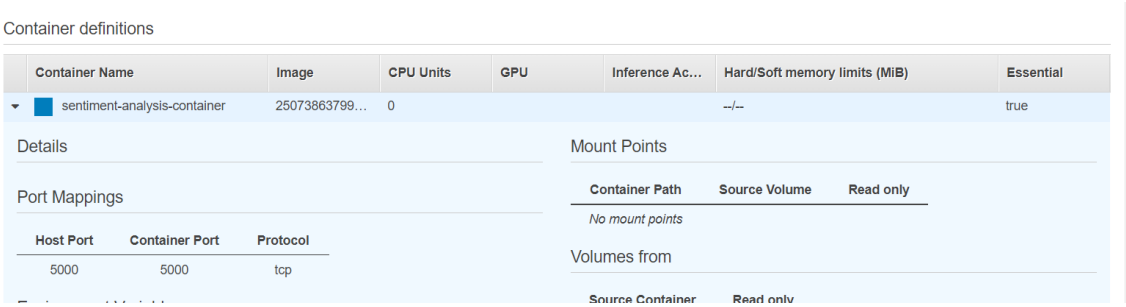
- First step is to create a cluster using ECS. Select 'Networking Only' and then create.



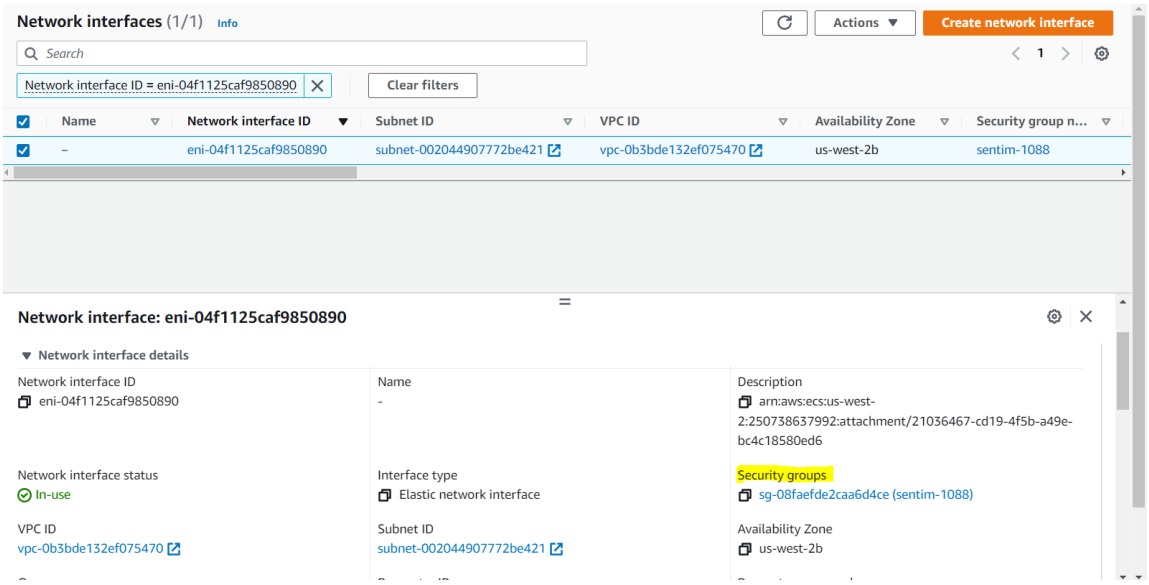
- Next you must create a 'Task Definition'and choose 'Fargate' as launch type. Give the task definition a name, specify task Memory and CPU needed to run the task.



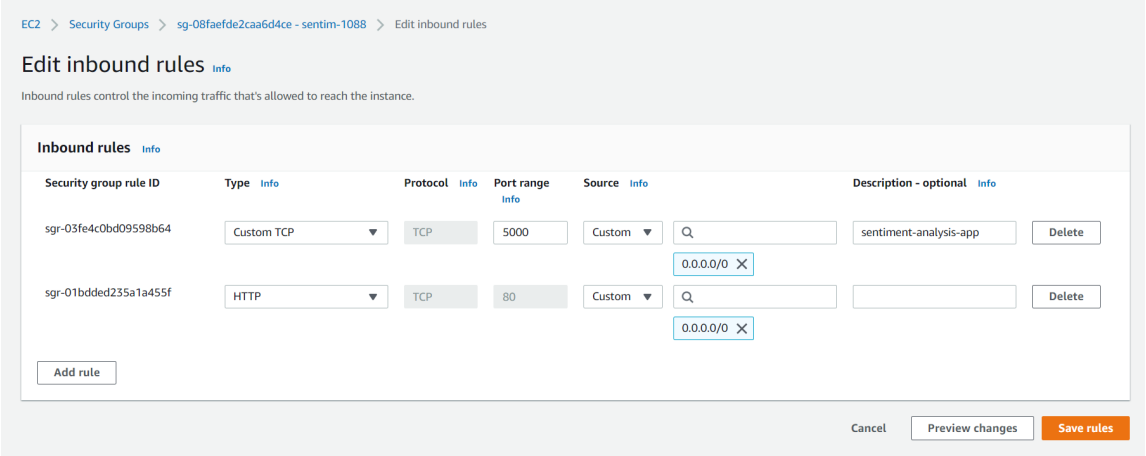
- On the same page, click 'Add container' and provide a name and the image URI (available from your ECR repo). As per how the application is developed, specify the port mappings to 5000 so containers can send or receive traffic.



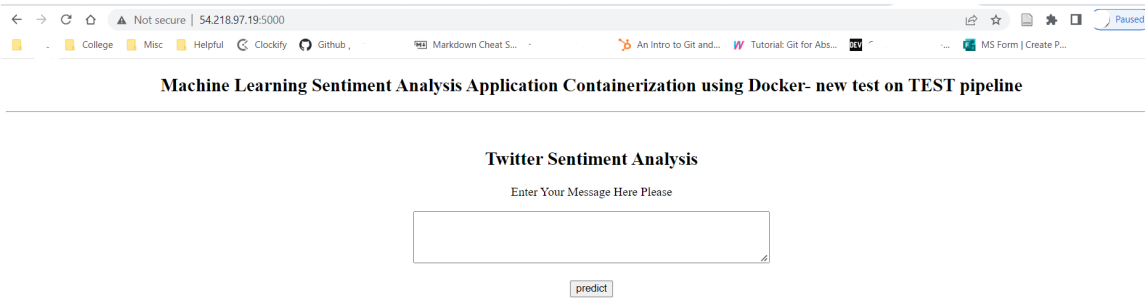
- Click on Actions and Run Task, select launch type as Fargate. Select the VPC and subnet from the dropdown and click Run Task.
- Our Task is created successfully, now we need to add an inbound rule to the security group to access our application on port 5000.
- Click on the created task → click 'ENI Id' → network interface page.
Click on the network checkbox and scroll down to 'Security groups'.



- Now click on Edit inbound rules and click Add rule. Add a custom TCP rule with port 5000 and source to be 0.0.0.0/0 (to make the application accessible through all IPs) and click Save rules.



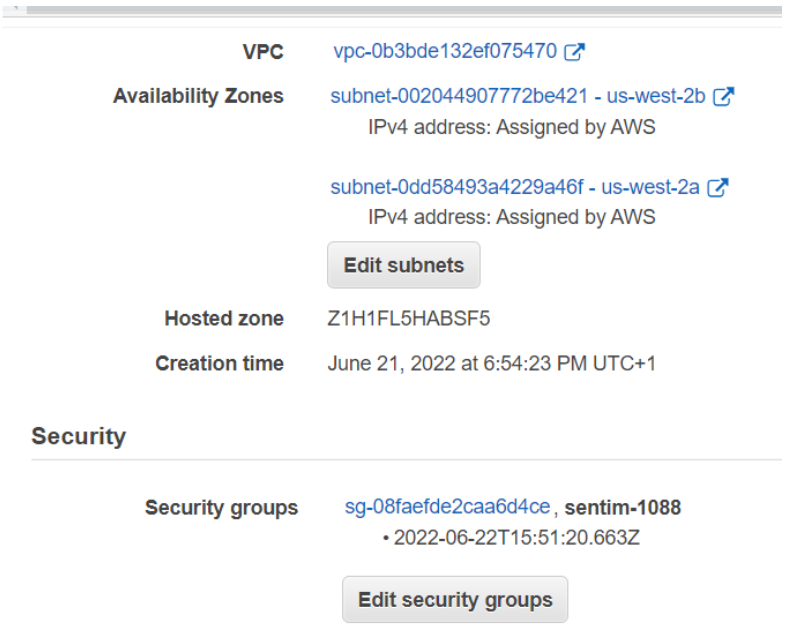
- Go back to the task page and using the public IP and port 5000, you can access the sentiment analysis application from the browser.



3.4 EC2 Load Balancer

Next, we create an Application load balancer (ALB) for our application. A load balancer serves as the single point of contact for clients and distributes incoming application traffic across multiple targets, such as EC2 instances, in multiple Availability Zones. This increases the availability and scalability of the application.

- Go to EC2 and click on Load Balancers. Choose Application load Balancer and Create.
- Add a name, VPC and availability zones.
- Create a new security group, add a name to Target group, for Target type select IP and then click Next.

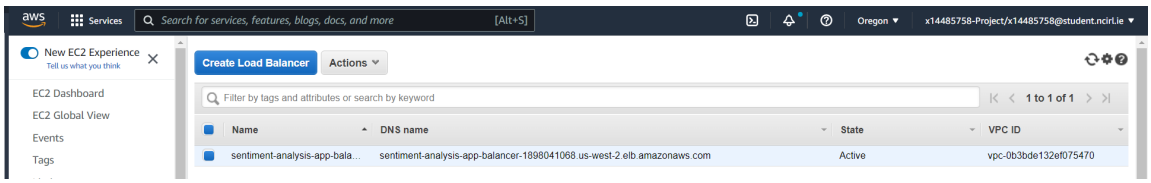


- Once created, note the DNS name, which is the public address where the application is accessed from the browser.

3.5 AWS Fargate

We will use the same task definition we already created in ECS to create a Fargate Service. AWS Fargate is a serverless compute engine for Amazon ECS that runs containers without requiring us to worry about the underlying infrastructure.

- Go to Task Definitions in Amazon ECS → tick the radio button corresponding to the existing task definition → click Actions → Create Service → Choose Fargate.
- Leave all other options as is, including rolling update as deployment type.
- Choose the same subnets as the one configured in the load balancer.
- Choose ALB as the load balancer type and add the already created ALB to it.
- Select the target group created in the ALB and click Create Service.
- Go back to the EC2 ALB and click on security groups; add a 'Custom TCP' rule with port '5000' under inbound rules. This is the internal port that the flask app is set to run on.
- Visiting the ALB DNS url previously noted, you can check the application is running on the fargate service.



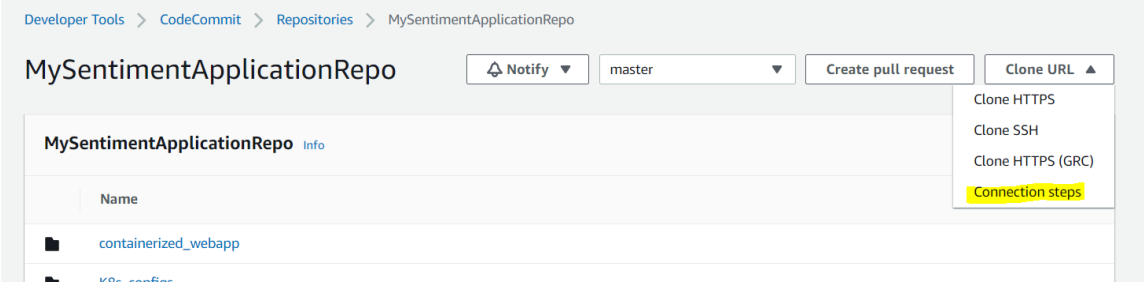
DNS URL to Running application: sentiment-analysis-app-balancer-1898041068.us-west-2.elb.amazonaws.com

4 Configuring the Pipeline

4.1 CodeCommit

AWS CodeCommit is used to store the applications artifacts. It is Amazon's version of Github.

- Click Create Repository and fill in the required details.
- Once created, click on the repo and from the drop-down choose 'Connection Steps'

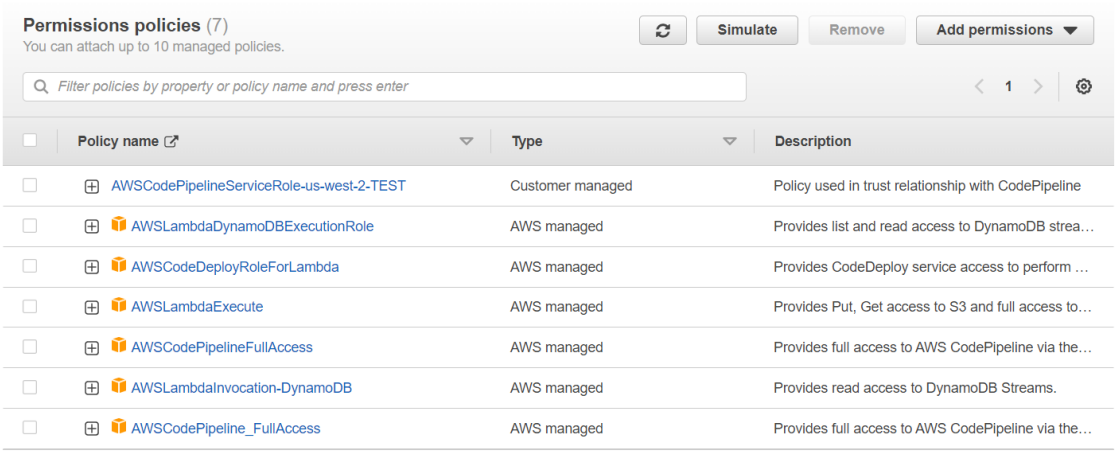


- You must still have an active session on the AWS CLI for the next steps. Otherwise, refresh the session by re-running 'aws configure'
- Next, follow the steps as outlined on the CodeCommit console to push the local application folder to the CodeCommit repository.

4.2 CodePipeline

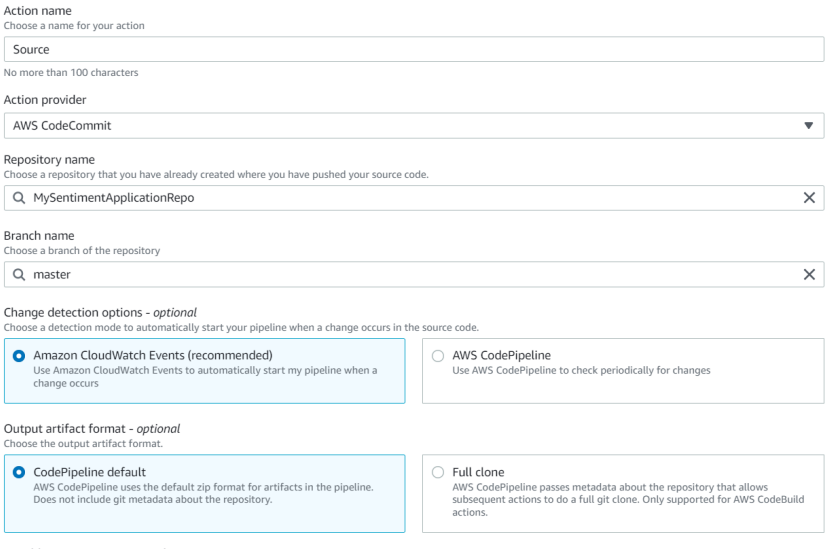
AWS CodePipeline is used to configure and build the CI/CD/CB pipeline.

- Go to CodePipeline using the AWS Console. Click Create Pipeline.
- Provide a name and take note of the pipeline role name created for you. Click next.
- Go to IAM and add the following policies to the CodePipeline role:



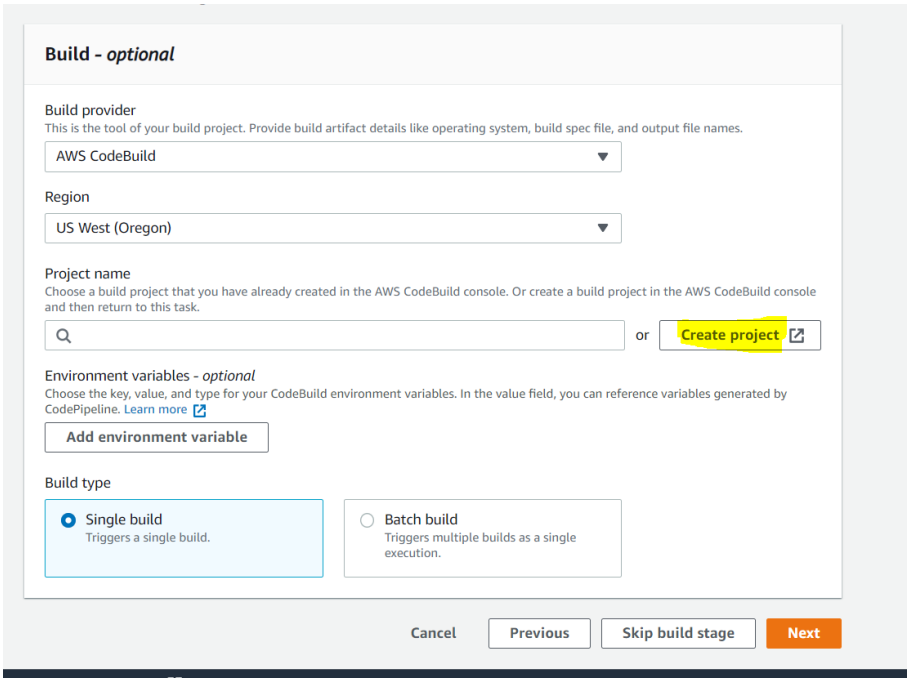
4.2.1 Source

- Next we configure the source stage. Choose AWS CodeCommit as the action provider.
- Choose your application repository in CodeCommit.
- Choose 'master' as branch name. This is what will become your release trigger.
- Keep CloudWatch logs enabled.

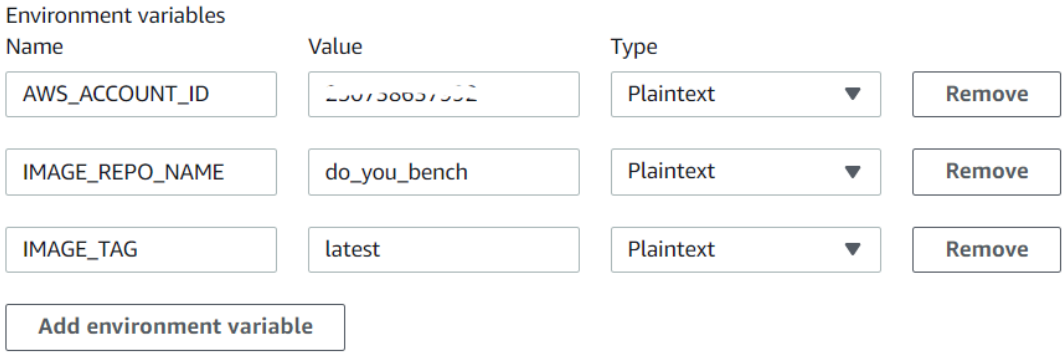


4.2.2 Build

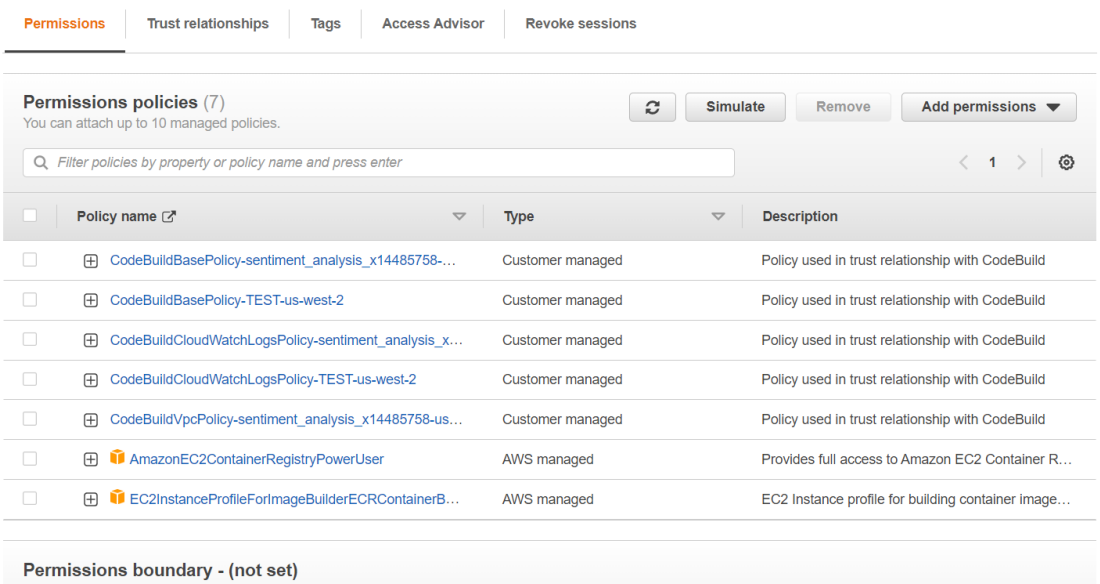
- The next stage is build. Choose CodeBuild as Build Provider and click 'Create Project'.



- Configure the build project by adding a name, choose 'custom image', environment type as 'Linux' and specify the Amazon ECR repo from the dropdown.
- Ensure you check the box that gives privileged access for CodeBuild to build docker images on your behalf.
- Add the following environment variables under Additional configuration:



- Under 'Logs', ensure both cloudwatch and S3 are checked. You will need to specify what bucket in S3 you wish to use.
- Leave everything else as default and click create.
- Go to IAM and add the following policies to the CodeBuild role that has just been created:



4.2.3 Deploy

- For deploy provider, chose Amazon ECS, cluster and service name. Include the name of the image definitions file as 'images.json' which is a command in the build.yml file that creates this file during the build process.

```
#Required sequence. Represents the commands CodeBuild runs during each phase of the build.
phases:
  pre_build:
    commands:
      - echo Logging in to Amazon ECR...
      - CODEBUILD_RESOLVED_SOURCE_VERSION="${CODEBUILD_RESOLVED_SOURCE_VERSION:-$IMAGE_TAG}"
      - IMAGE_TAG=$(echo $CODEBUILD_RESOLVED_SOURCE_VERSION | cut -c 1-7)
      - echo image_tag $IMAGE_TAG
      - REPO="$AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com"
      - IMAGE_URI="$AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com/$IMAGE_REPO_NAME:$IMAGE_TAG"
      - echo Repository $REPO
      - docker login -u AWS -p $(aws ecr get-login-password --region $AWS_DEFAULT_REGION) $REPO
  build:
    commands:
      - echo Build started on `date`
      - echo Building the Docker image...
      - docker build -t $IMAGE_URI .
  post_build:
    commands:
      - bash -c "if [ /"$CODEBUILD_BUILD_SUCCEEDING/" == /"0/" ]; then exit 1; fi"
      - echo Build stage successfully completed on `date`
      - echo Pushing the Docker image...
      - docker push $IMAGE_URI
      - printf '[{"name":"sentiment-analysis-container","imageUri":"%s"}]' "$IMAGE_URI" > images.json
artifacts:
  files: images.json
```

- Click next and Create Pipeline.

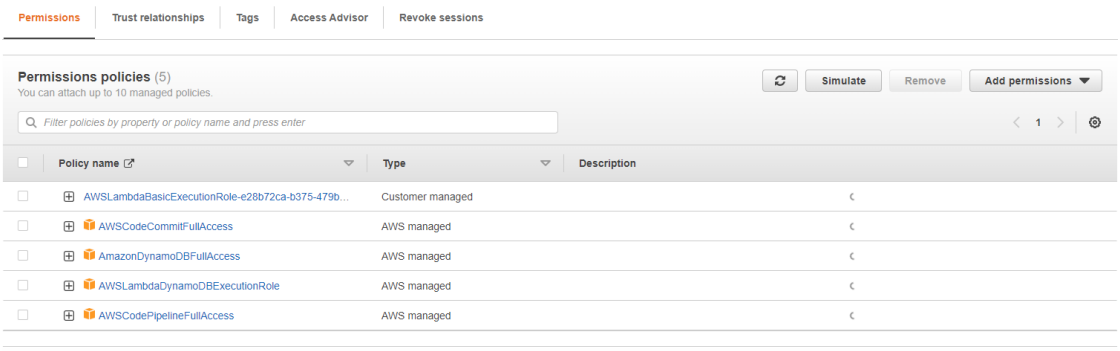
5 Implementing Benchmarking

Once the pipeline is built and the application has been deployed, its time to develop benchmarking using Lambda. AWS Lambda is an event-driven, serverless computing platform. Two additional stages will be created in the pipeline, both which will trigger lambda functions. The first benchmark will take place after the source stage and the second will be after the build stage.

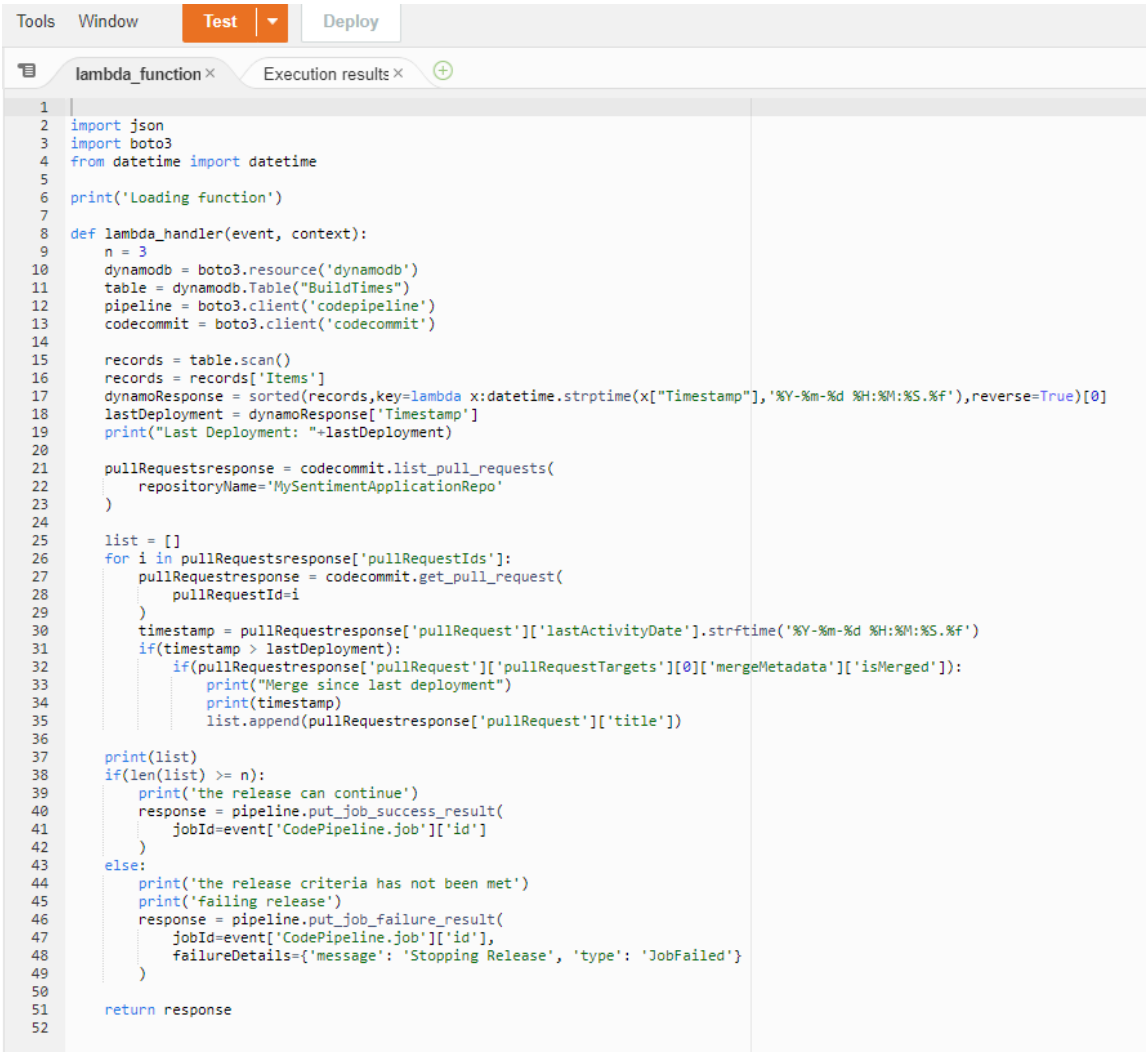
Benchmarking after the Source Stage

This lambda function is dependent on event data from CodeCommit in order to perform the benchmarks.

- Go to Lambda and create a new function. Choose 'Author from scratch'. Create a new role and add the following permissions:



- The python code for this function is below:

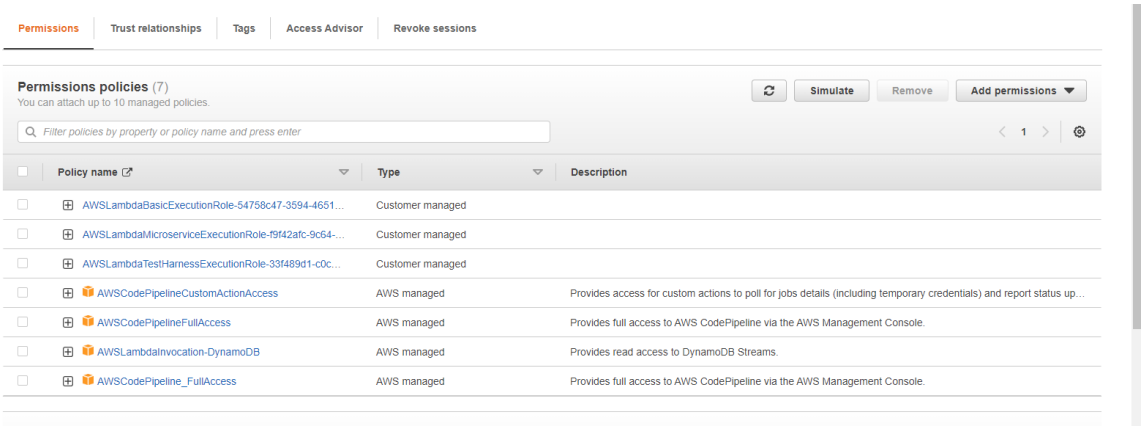


- Boto3 is an AWS SDK that provides a Python API for AWS infrastructure services. We use this SDK to query event data from CodeCommit. 16

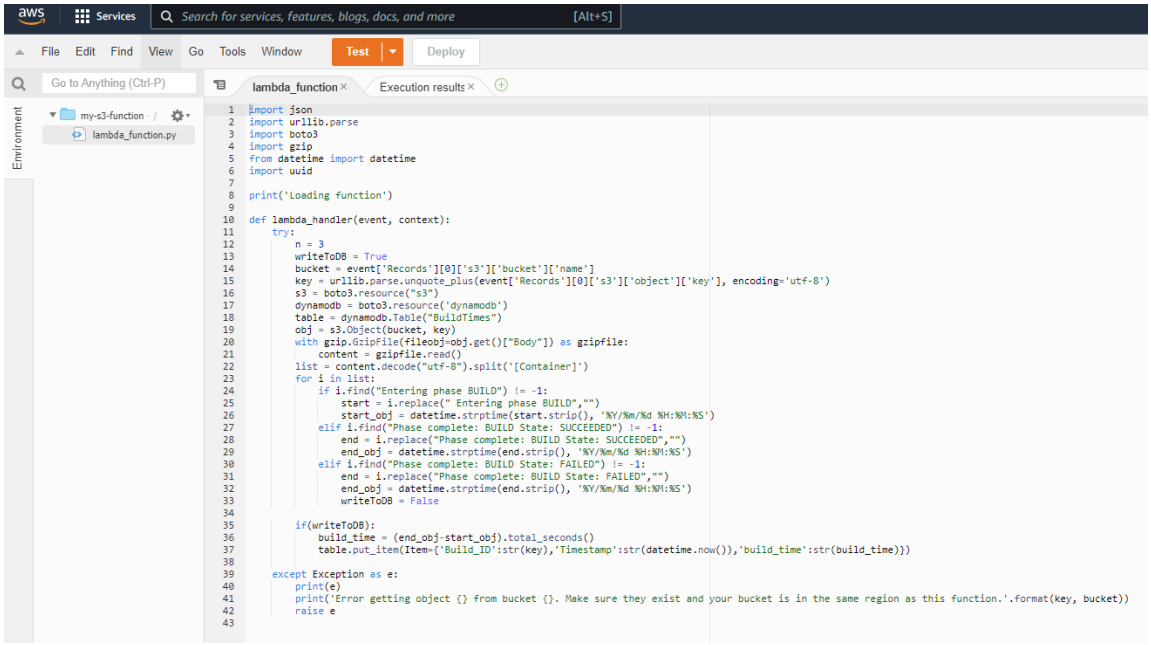
Benchmarking after the Build Stage

This lambda function is dependent on log data from CodeBuild in order to perform the benchmarks.

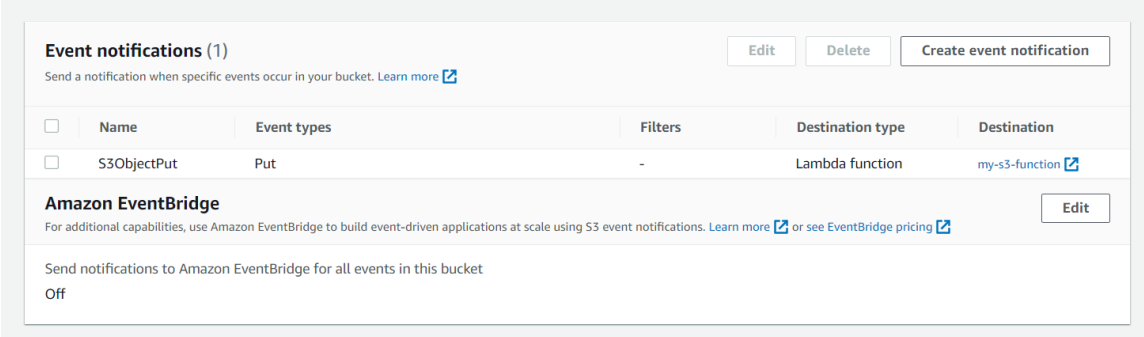
- The build project was setup to send the logs to an s3 bucket. By default, these appear as zipped files in s3 so in order to be able to review the logs and target certain key events, we will create a lambda function to unzip the log files from s3 and send our targeted attributes to a table in DynamoDB.
- Go to Lambda and create a new function. Choose 'Author from scratch'. Create a new role and add the following permissions:



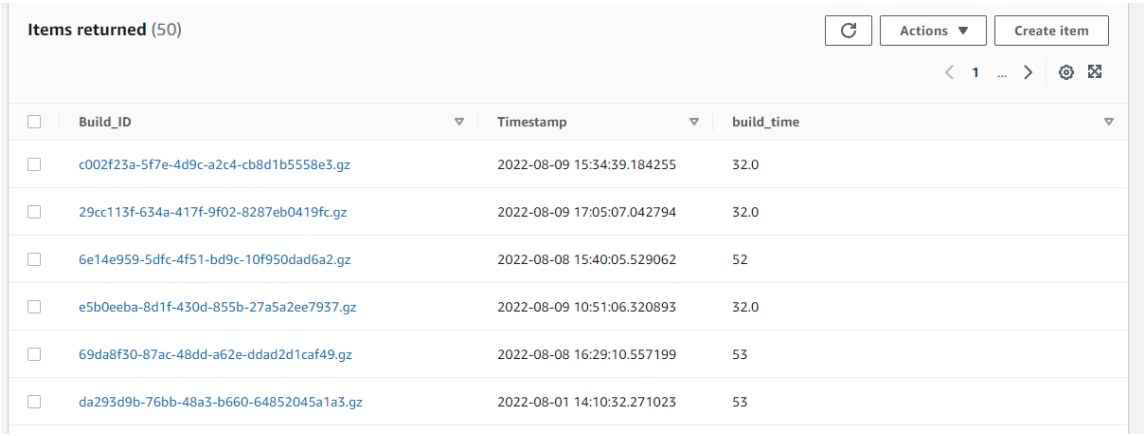
- The python code for this function is below:



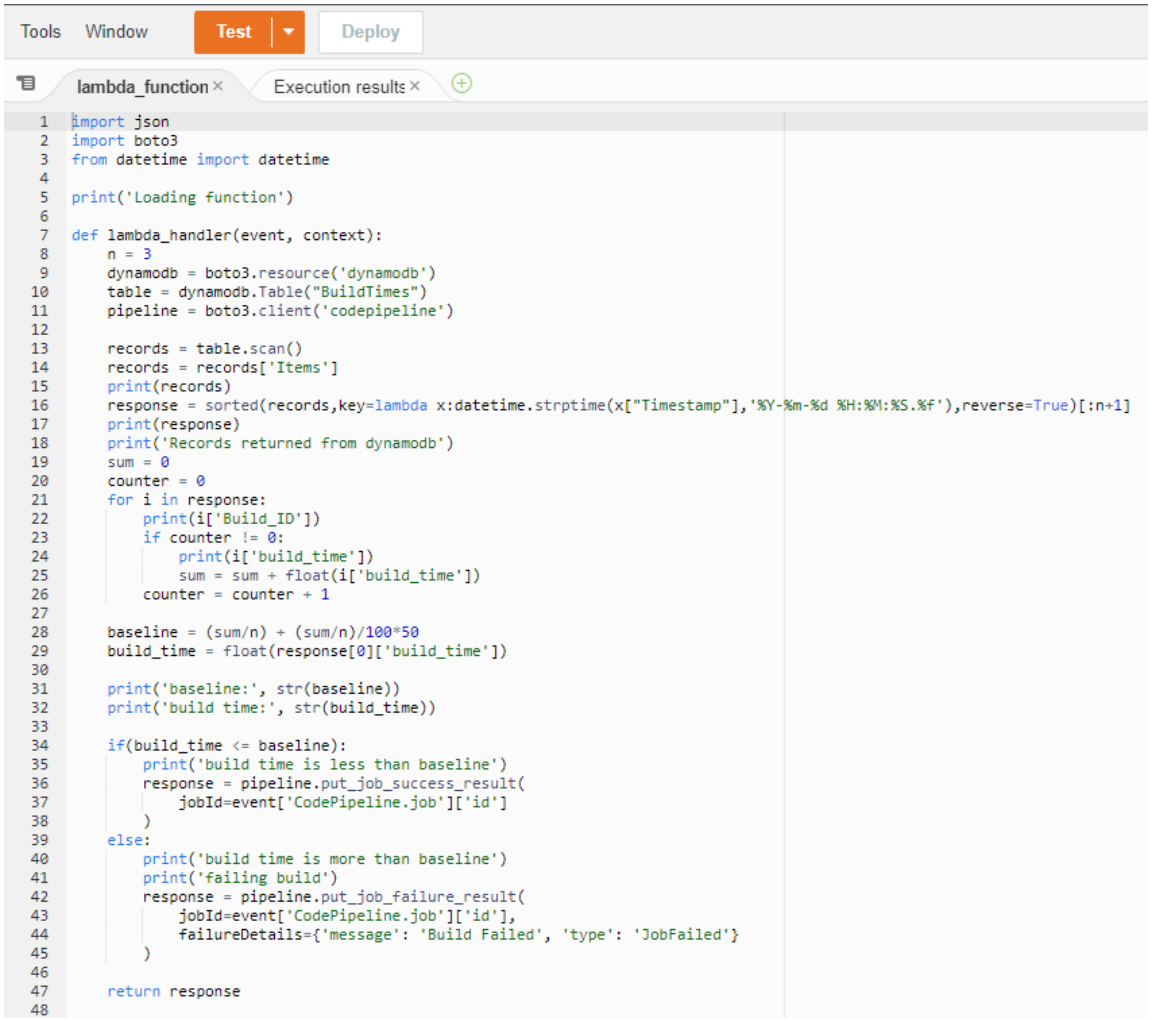
- To trigger this function, we add a lambda trigger to the S3 bucket where the build logs are being sent. Therefore, everytime a build completes, the log files are sent from cloudwatch to s3.



- Once they land in s3, our lambda function is triggered to get the data we need and store them in our dynamo table:



- Next we develop the logic to perform the benchmarking, the python code for this is below:

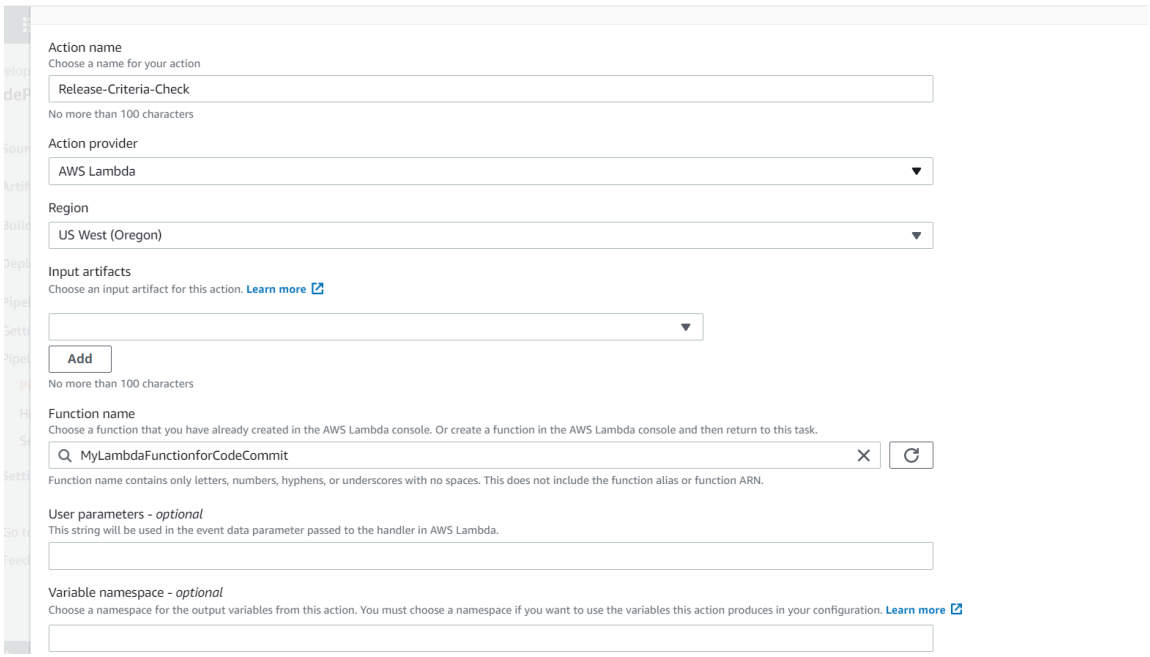


- Boto3 is an AWS SDK that provides a Python API for AWS infrastructure services. We use this SDK to communicate with other services such as S3 and dynamo.

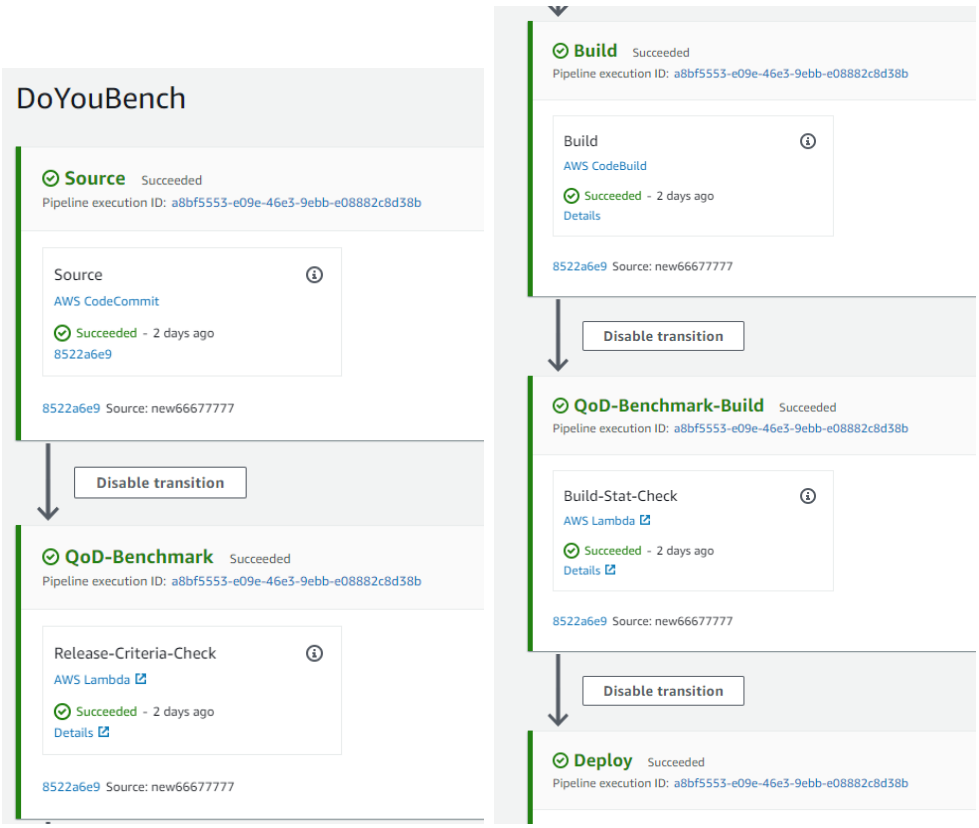
Add Benchmarking Stages to the Pipeline

Once the lambda functions have been developed, its time to add two new benchmarking stages to our pipeline.

- Go to CodePipeline → Click into your pipeline → Click Edit → Add Stage.
- Name your action and choose AWS Lambda as Action provider and choose the relevant function name from lambda:



- Repeat the above for the second benchmarking stage. Once you have done so, your pipeline should look something like this:

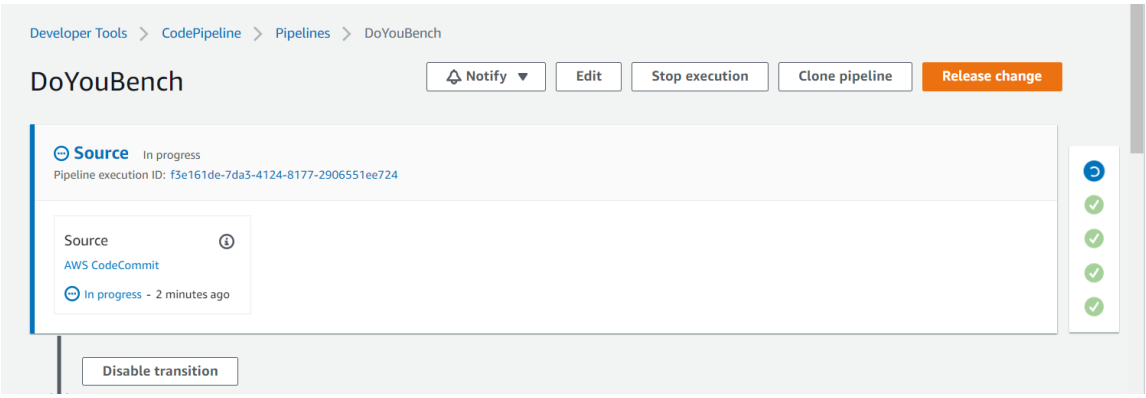


NB: The lambda functions code should be updated with suitable endpoints for your project such as CodeCommit repo, dynamo table, S3 bucket etc.

6 Running Deployments

Now its time to use the pipeline to deploy application changes.

- Make a code change to a html file and push the changes to the master branch in your CodeCommit repo.
- git add .
- git commit -m ” ”
- git push
- The pipeline release should start instantly.



- Both benchmarking functions include logic that will send a response back to the pipeline to say the benchmarks has pass or not. This will determine whether the application release can proceed to the next stage of the pipeline. If the criteria was not met, a response is sent to 'fail' that stage of the pipeline, preventing the cycle from proceeding to the next stage.
- The criteria for when each benchmark should pass and should fail is discussed in detail in the research paper.