Develop decoupled serverlesss backend architecture

Before going to develop the architecture, let's understand all the aws services and automation tool that we are using in this serverless architecture.

What is Serverless?

Serverless is a term that generally refers to serverless applications. Serverless applications are ones that don't need any server provision and do not require to manage servers.
A serverless infrastructure based on AWS Lambda has two key benefits:

1. We don't need to manage any virtual machines anymore.

2. Deploying new versions of our can be entirely controlled by API calls

How to process asynchronous tasks serverless:

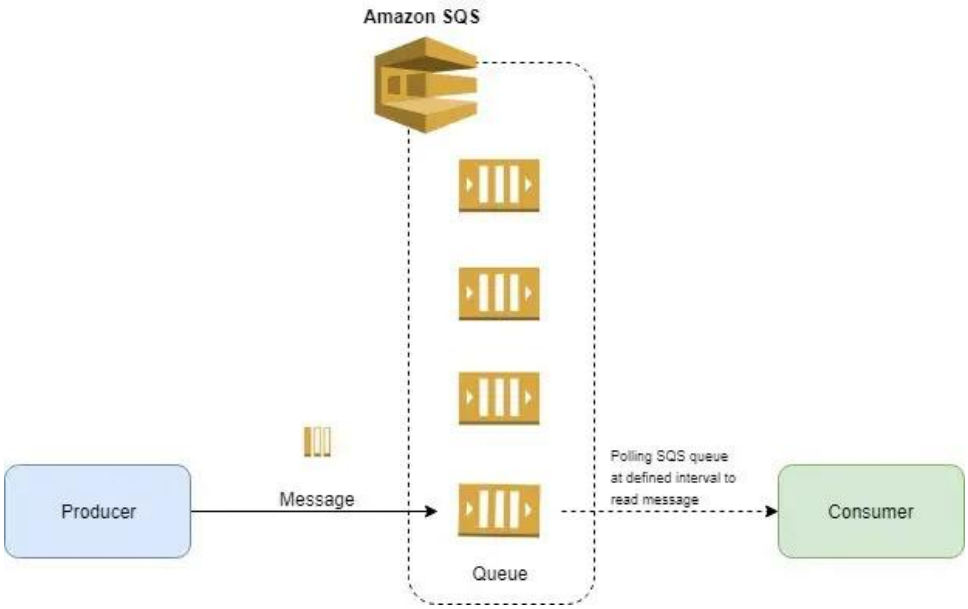
Possible use cases are: sending out massive amount of messages, emails, transcoding videofiles after uploaded, or analyzing user behavior. An SQS will be used to decouple our microservice from other parts of our system.

Firstly,

What is SQS?

Amazon Simple Queue Service (SQS) is a Fully managed message queuing for microservices, distributed systems, and serverless applications. It offers fault tolerant and scalable distributed message queues: simple to use but very powerful.

Asynchronous workflows have always been the primary use case for SQS. Using queues ensure one component can keep running smoothly without losing data when another component is unavailable or slow.



What is Lambda?
Lambda is an event-driven, serverless computing service provided by Amazon. Therefore we don't need to worry about Which AWS resources to launch, or how will you manage them. Instead, we need put the code on lambda, and it runs.

However, Amazon Lambda can only used to execute background tasks.

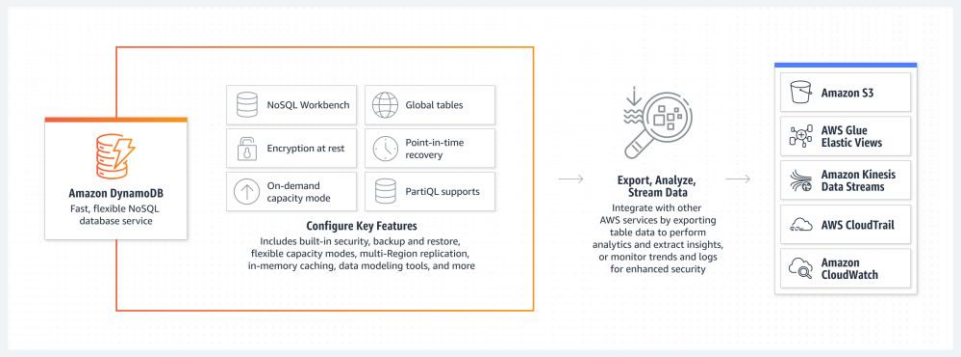
AWS Lambda fucntion helps you to focus on your core product and business logic instead of managing operating system(os) access control, provisioning, etc.

What is DynamoDB?

Fast, flexible NoSQL database service for single-digit millisecond performance at any scale

How it works

Amazon DynamoDB is a fully managed, serverless, key-value NoSQL database designed to run high-performance applications at any scale. DynamoDB offers built-in security, continuous backups, automated multi-Region replication, in-memory caching, and data import and export tools.

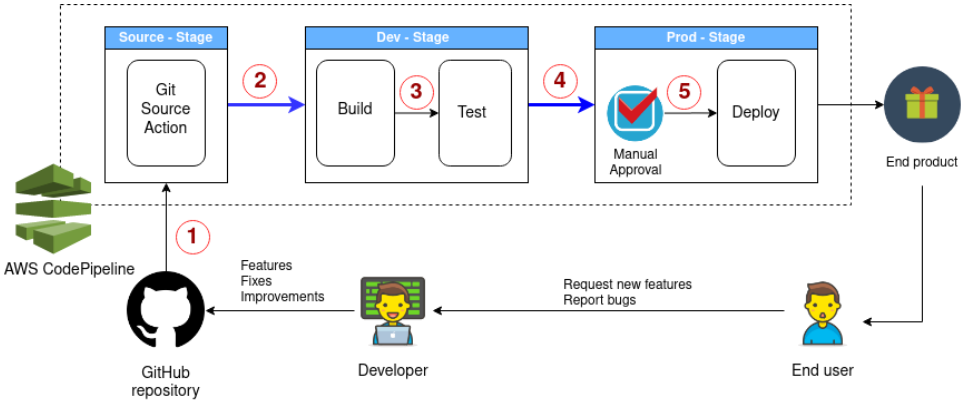


What is AWS CodePipeline?

AWS CodePipeline is an Amazon Web Services product that **automates the software deployment process,** allowing developers to quickly model, visualize and deliver code for new features and updates.

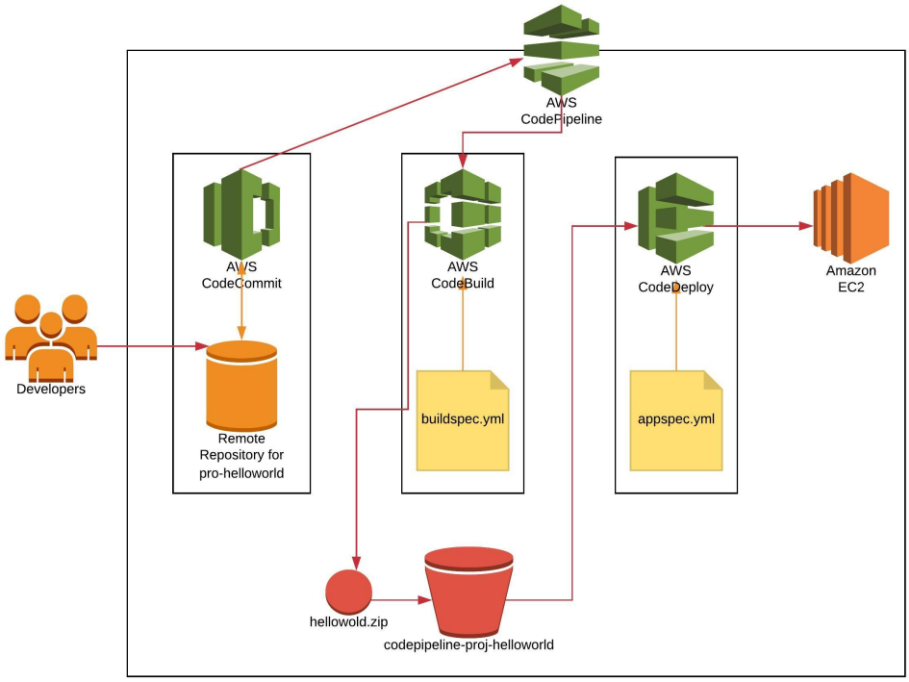
AWS CodePipeline automatically builds, tests and launches an application each time the code is changed.

AWS CodePipeline is a continuous delivery service that enables you to model, visualize, and automate the steps required to release your software.



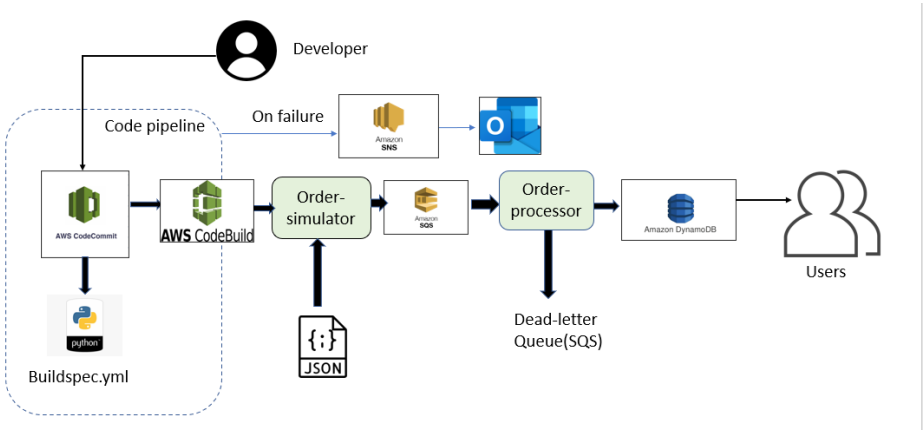
AWS Codepipeline providers

Setup and configuration



Finally After understanding all the basics of needed services to Develop decoupled serverlesss backend architecture We can start:

Inorder to start, lets understand the below architecture in steps -



Step 1 : Writing two Lambda functions, One Function(Order_simulator) which will create and send the messages to an SQS and second function(Order_processor) which will process all SQS messages and store in DynamoDB and SQS queue to store the badly or invalid data in dead letter queue - poll for the messages to view the exact input given

Dynamo DB to have item records for valid data.

Step 2 : Creating a Pipeline which can automate our Lambda Function

-> We need push our git code respository to code commit which contains buildspec.yaml/json and appspec.yaml/json.

Why buildspec.yaml/json need?

A buildspec is a collection of build commands and related settings, in YAML /JSON format, that codebuild uses to run a build. You can include a buildspec as part of the source code or you can define a buildspec when you create a build project.

Here is the code for buildspec.yaml file, this is for upadating our lambda function, if we creating lambda functions througn codebuild, then u need to add create-function commands, u can get these here by uncommenting the lines. We can also invoke the functions by sending the response file for lambda function.

Why appspec.yaml/json need?

The application specification file (AppSpec file) is a YAML -formatted or JSON-formatted file used by CodeDeploy to manage a deployment. The AppSpec file for an EC2/On-Premises deployment must be named appspec. yml or appspec. yaml , unless you are performing a local deployment.

Now we can start the pipeline.

Step 3 : Creating an Simple Notification Service for failures occurs in codepipeline it will sent a notification to our outlook email

To create a notification rule (console)

1 . Sign in to the AWS Management Console and open the CodePipeline console at https://console.aws.amazon.com/codepipeline/.

2 . Choose **Pipelines**, and then choose a **pipeline** where you want to add notifications.

3 . On the pipeline page, choose **Notify**, and then choose **Create notification rule**. You can also go to the Settings page for the pipeline and choose Create notification rule.

4 . In Notification name, enter a name for the rule.

5 . In Targets, do one of the following:

- If you have not configured a resource to use with notifications, choose Create target, and then choose SNS topic. Provide a name for the topic after codestar-notifications-, and then choose Create.
- If you create the Amazon SNS topic as part of creating the notification rule, the policy that allows the notifications feature to publish events to the topic is applied for you. Using a topic created for notification rules helps ensure that you subscribe only those users that you want to receive notifications about this resource.
- If you want to use an existing Amazon SNS topic as a target, you must add the required policy for AWS CodeStar Notifications in addition to any other policies that might exist for that topic.
- Creating a Topic, Go to **SNS console** click on **Topic** -> click on **Create Topic** -> Give **Topic name** -> After click **Create Topic.** Click on **Create Subscription** in the Topic which we created. Then Give a name, select **Email as protocol. Click on Submit**.

6 . To finish creating the rule, choose Submit.

7 . You must subscribe users to the Amazon SNS topic for the rule before they can receive notifications.

Step 4 : Create a SQS queue and adding lambda trigger

1. Visit the SQS Console again and click on the queue created eariler:

2. Click on '**Lambda Triggers**' and click on '**Configure Lambda function trigger**' button.

3. Search for the lambda function created earlier by name in the drop down and select it. Click on the '**Save**' button. This would successfully add the lambda trigger to the SQS queue:

4. Now to test our setup, we shall send a message to the queue (as shown earlier in the article). The message must get passed on to the lambda function.

5. Go back to the lambda console and click on the '**Monitor**' tab. Then click on the '**View logs in CloudWatch**' button. This will open a new tab and take you to CloudWatch.

6. In the CloudWatch log group page, you would find a log stream pretaining to the latest execution of the lambda function. Click on that to see the logs :

7. The log of the message object is highlighted below. Expand it to view the entire object. The message we sent will appear inside the 'body' field of the message object.

So at this stage, we have a functional setup where a message sent into SQS gets pushed to a lambda function which in turn logs it into CloudWatch. Now for the final piece of the puzzle, we have to setup a DynamoDB table and write some code to insert data into it.

Step 5 : Create a DynamoDB that can store our SQS valid messages

lets understand the below architecuture,

Here after our order_simulator lambda function sent messages to the SQS , In the Next step we need add our order_processor lambda function as a trigger to the SQS .

Then Whenever The messeges added to our SQS Queue it trigger the order_processor function, then the function will trigger and push all the valid messages to the DynamoDB. Duplications or invalid messages added to the dead letter queue.

Here we can add Trigger in two ways -

1. By Going to SQS console

2. By adding triggers in Lambda function.

Here we are adding trigger by going to SQS console

Go to the DynamoDB console and click on '**Create Table**':

2. Provide a table name and a partition key as shown below. Leave the other settings as they are by default. Scroll to the bottom of the page and click on '**Create Table**' button :

3. The table would take some time to get created. In the meantime, go back to the Lambda console. Paste the following code inside the editor and click on '**Deploy**' (like we did previously) :

The above code parses the message received from SQS and writes it into the dynamoDB table we created. It also logs some additional information to CloudWatch in case of success as well as error.

Our main Goal is not creating the resources by going to every aws console manually, because we human can make errors and also spending lot of time by visiting every console, we need a configuration as a code which can automate everything regarding our serverless infrastructure.

So Let's get Start,

Terraform for Automating the AWS Serverless Infrastructure

So Far we creating everything mauallly, inorder to reduce the headache of creating iam roles, policies, sqs and dynamodb by visiting every aws console, its take a lot of time.

So for that, Now we are automating everything by using automation tools to reduce.Here I am using terraform.

Let's understand what is terraform and why we are so exicited to use.

Without any further , let's get started.

So let us start with creating the infrastructure required by the pipeline. We will make use of Terraform which is an open-source infrastructure as a code (IaaC)software tool.

Why infrastructure as a code?

It is fair to ask the question why to make use of infrastructure as code when we could make use of AWS services using the console? The reason is simple,

• Firstly, let say we develop the pipeline in development environment. Once happy with the pipeline we move the pipeline to test and then to prod environment. Imagine we make use of 100s of services, then we would have to recreate all those services manually first in test and then in development environment. This is a tedious and time consuming task.

• Secondly, with 100's of services, that needs to be created multiple time, there is a fair bit chance of errors that can happen and would be very difficult to debug when done manually.

• Finally, if we create services manually we can not use version control techniques i.e using Git. Infrastructure as a code (Iaac) is quite helpful in that aspect.

Note : We make use of terraform to implement infrastructure, however it is not the only tool out there in the market. Since we are using AWS we could have use AWS Cloudformation another IaaC service by AWS. I am use terraform due to its multi-cloud integration capabilities. So now let's understand briefly about terraform.

What is Terraform?
Terraform is an infrastructure as code tool that lets you build, change, and version cloud and on- prem resources safely and efficiently.

How does Terraform work?

Terraform creates and manages resources on cloud platforms and other services through their application programming interfaces (APIs). Providers enable Terraform to work with virtually any platform or service with an accessible API.

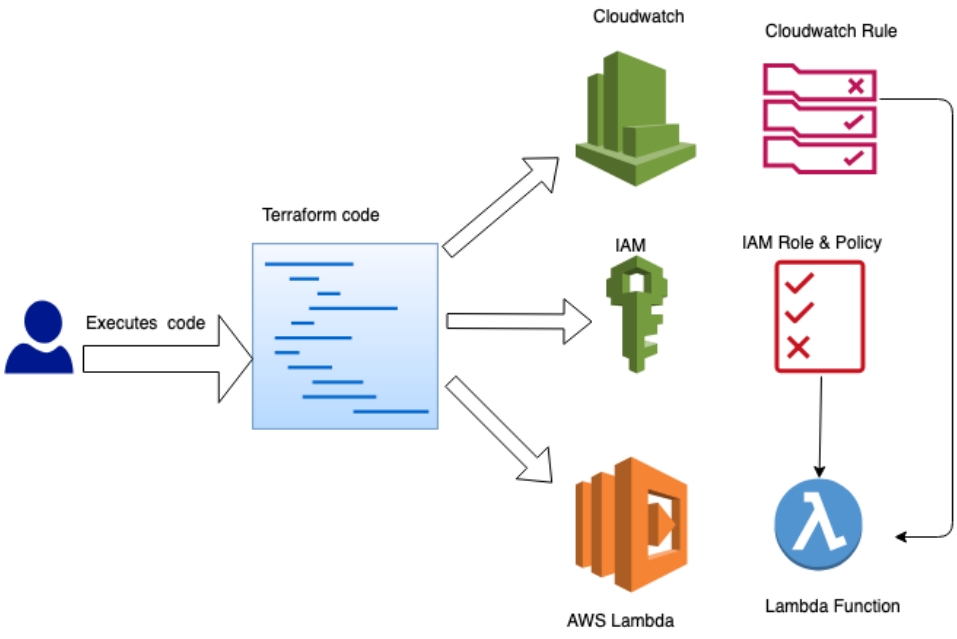
The focus is on automating as much as possible to reduce human errors and provide an efficient way. Although we would look at the pipeline in AWS, similar pipeline can be built any any cloud platform (including Google Cloud platform, Azure, etc)as they have very similar services.

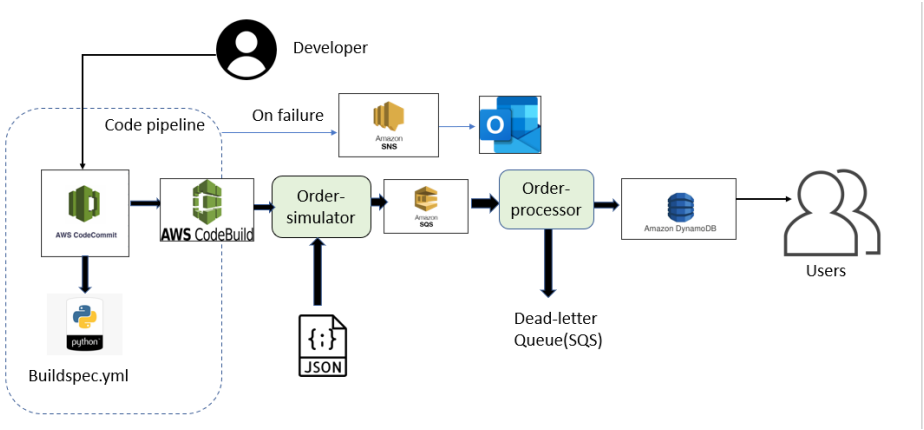
Why Terraform?

1. Manage any infrastructure 2. Track your infrastructure 3. Automate changes 4. Standardize configurations Collaborate

Furthermore, we would make use of Terraform to create the infrastructure as code. This helps in reducing the possibility of errors when moving the model from say development to production.

Let us start building the infrastructure

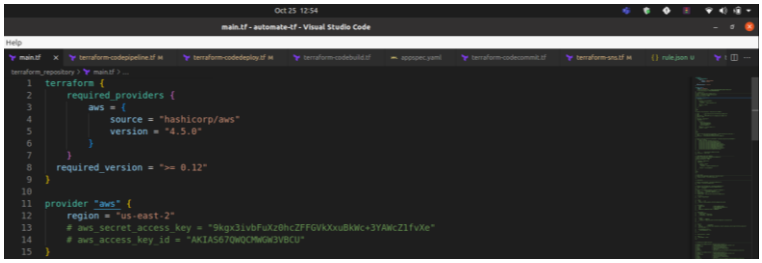




So Lets Start by installing terraform

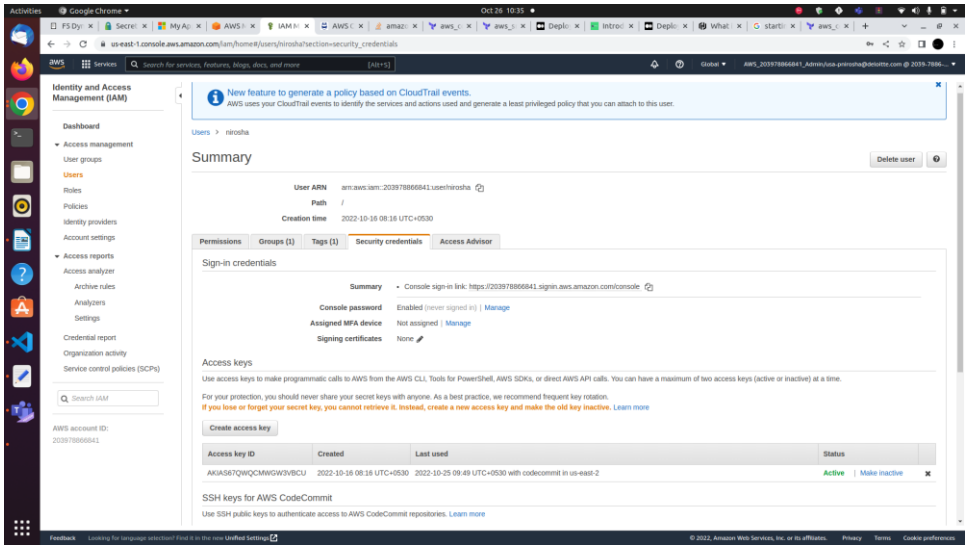
Step 1 : Install the terrafom.

Step 2 : create a main.tf configuration file and add aws provider in it.

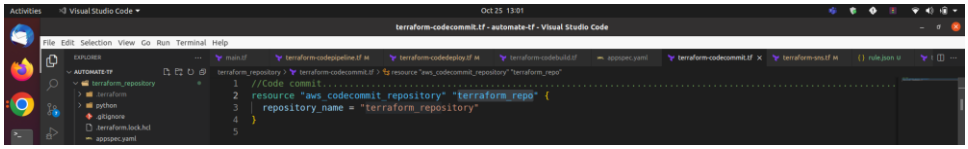


We should never pass the any secrets and access key, so for that configure the aws credentials in locally. Open the terminal, go to the folder and open the config file and paste the access key and secret key like show below.

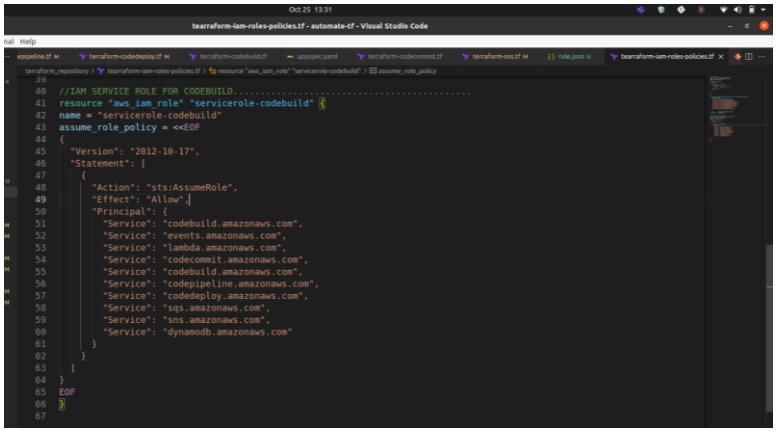
Open the aws IAM Console -> click on users -> create a user and after that open security credentials, in that we can see our accesskey, and for secret key it will show only once, download it, then copy the scret and access key paste in terminal.



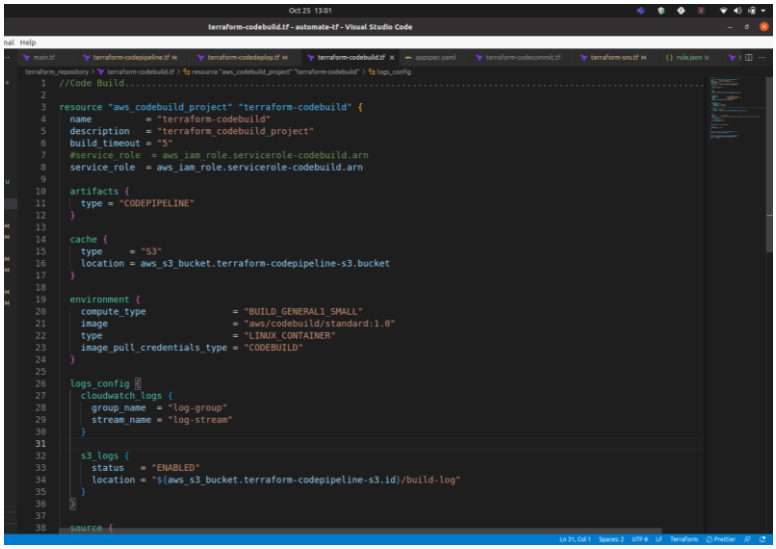
Step 3 : Create a file which contains all the configurations related to codecommit which should be .tf type. Using aws_codecommit_repository resource it can create a repository for us. Provide a repository name.



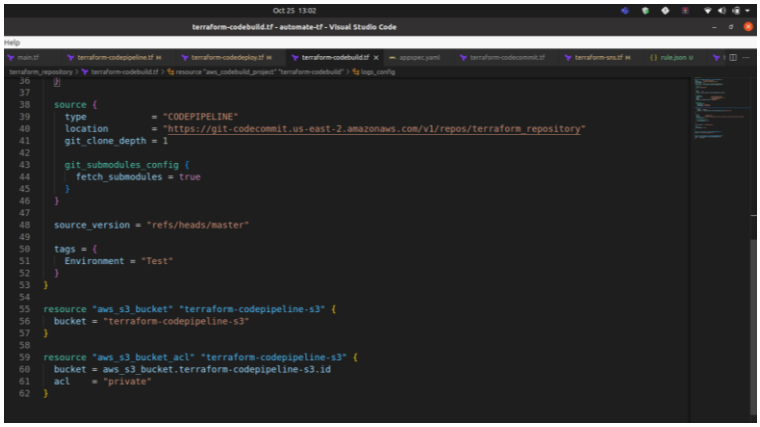
Step 6 : Before going to next step,we need create and setup a Service role that can allow the build to execute all services such as lambda,sqs, dynamobd,deploy, cloudwatch for logs and all the required iam permissions. We add json format policies and also we can direct add the services which allows all the permissions of the policies.



Step 5 : Next Create a file for Codebuild. Provide all the neccesary attributes such as build name, artifacts, source and service role that contain required iam permissions, make sure all the permissions are provided are not. Below is the code for my c odebuild.

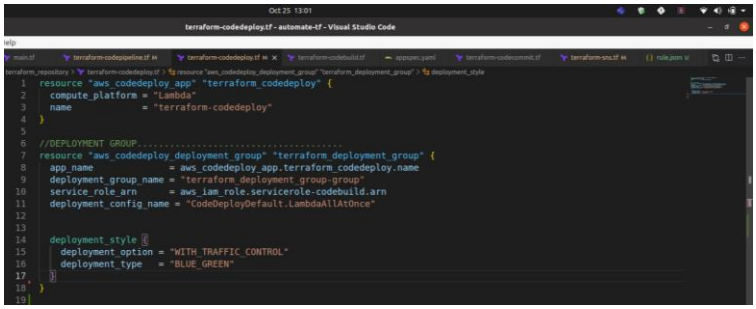


```
//Code Build.....

resource "aws_codebuild_project" "terraform-codebuild" {
  name          = "terraform-codebuild"
  description   = "terraform_codebuild_project"
  build_timeout = "5"
  #service_role  = aws_iam_role.servicerole-codebuild.arn
  service_role  = aws_iam_role.servicerole-codebuild.arn

  artifacts {
    type = "CODEPIPELINE"
  }

  cache {
    type     = "S3"
    location = aws_s3_bucket.terraform-codepipeline-s3.bucket
  }

  environment {
    compute_type                = "BUILD_GENERAL1_SMALL"
    image                       = "aws/codebuild/standard:1.0"
    type                        = "LINUX_CONTAINER"
    image_pull_credentials_type = "CODEBUILD"
  }

  logs_config {
    cloudwatch_logs {
      group_name  = "log-group"
      stream_name = "log-stream"
    }

    s3_logs {
      status   = "ENABLED"
      location = "${aws_s3_bucket.terraform-codepipeline-s3.id}/build-log"
    }
  }

  source {
```



```
  }

  source {
    type            = "CODEPIPELINE"
    location        = "https://git-codecommit.us-east-2.amazonaws.com/v1/repos/terraform_repository"
    git_clone_depth = 1

    git_submodules_config {
      fetch_submodules = true
    }
  }

  source_version = "refs/heads/master"

  tags = {
    Environment = "Test"
  }
}

resource "aws_s3_bucket" "terraform-codepipeline-s3" {
  bucket = "terraform-codepipeline-s3"
}

resource "aws_s3_bucket_acl" "terraform-codepipeline-s3" {
  bucket = aws_s3_bucket.terraform-codepipeline-s3.id
  acl    = "private"
}
```

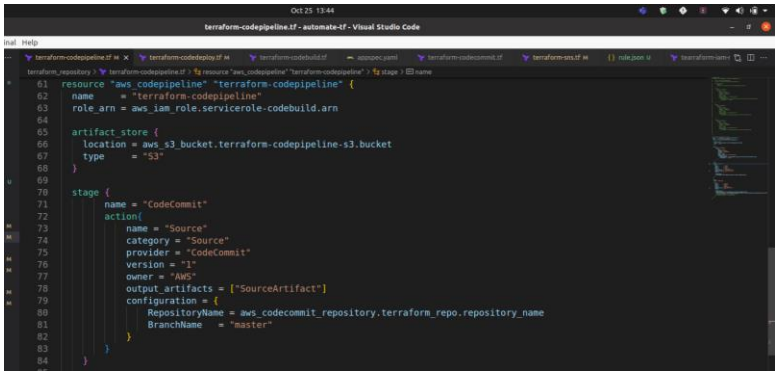
Step 6 : Create a another file which contains all the configurations related to code deploy. Provide all the required attribute fileds. Next For Code Deploy we need to create a Deployment_group also with deployment_style, make sure providing a correct lambda deployment_style which is blue/green by providing required permissions.



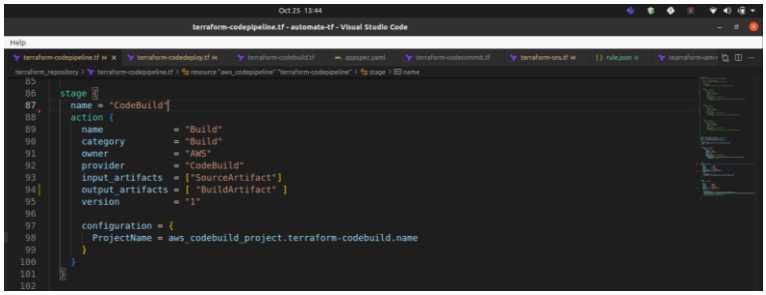
```
resource "aws_codedeploy_app" "terraform_codedeploy" {
  compute_platform = "Lambda"
  name             = "terraform-codedeploy"
}

//DEPLOYMENT_GROUP.....
resource "aws_codedeploy_deployment_group" "terraform_deployment_group" {
  app_name              = aws_codedeploy_app.terraform_codedeploy.name
  deployment_group_name = "terraform_deployment_group-group"
  service_role_arn      = aws_iam_role.servicerole-codebuild.arn
  deployment_config_name = "CodeDeployDefault.LambdaAllAtOnce"


  deployment_style {
    deployment_option = "WITH_TRAFFIC_CONTROL"
    deployment_type   = "BLUE_GREEN"
  }
}
```

Step 7 : Create a pipeline which contains all the three stages which source , build, deploy. For that, Create a another configuration file which contains all the confirguration data. Artifact_store is required to store our files in s3 bucket.

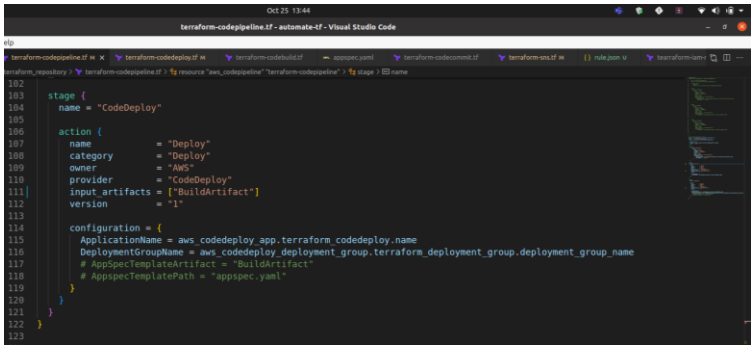
Adding Commit stage by specifiying all the configuration which we did in codecommit file. Make sure that use the Source catagory as Source and Branchname as u have, here i have master and provide a codecommit repository arn.



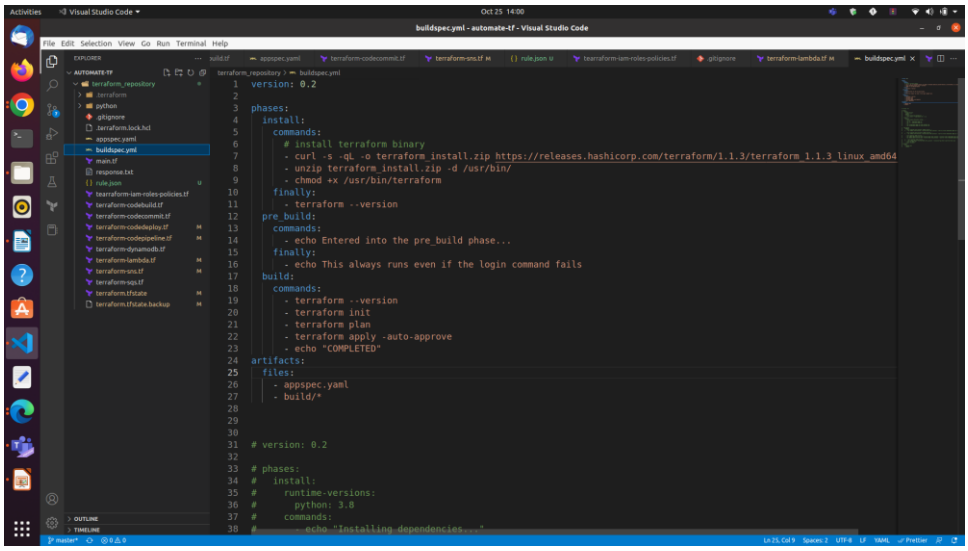
Adding Build Stage with the project name that we created previous in configuration section.



Adding the Deploy stage with the Aplication and deployment_group that we created previous in codedeploy configuration file.



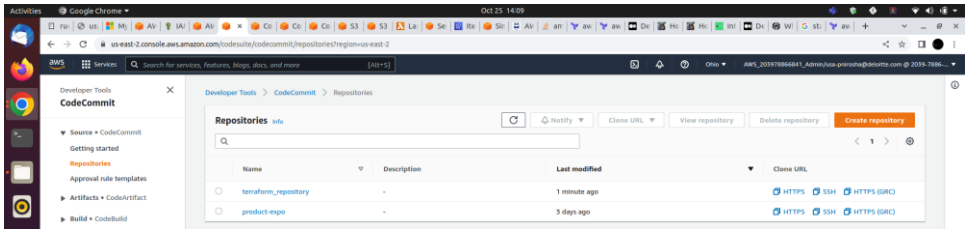
Before going to console and start the pipeline we need to create a **buildspec.yaml** file for codebuild. Here is my buildspec.yaml file.



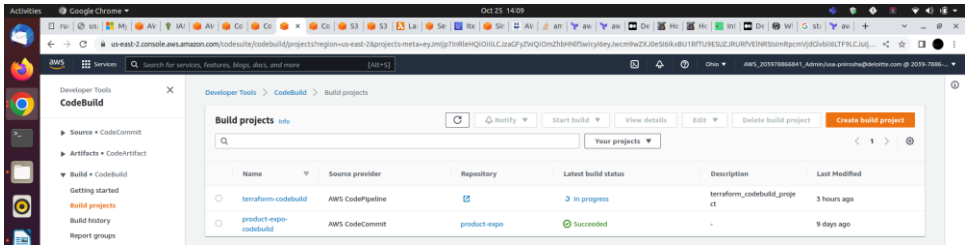
And also need to **appspec.yaml** file codedeploy is required. We will talk about this later, for now just add **2 stages** that are codecommit and codebuild.

So, Now go to the Console And check all the resources are created or not with all the configurations the we specified

Codecommit repository.

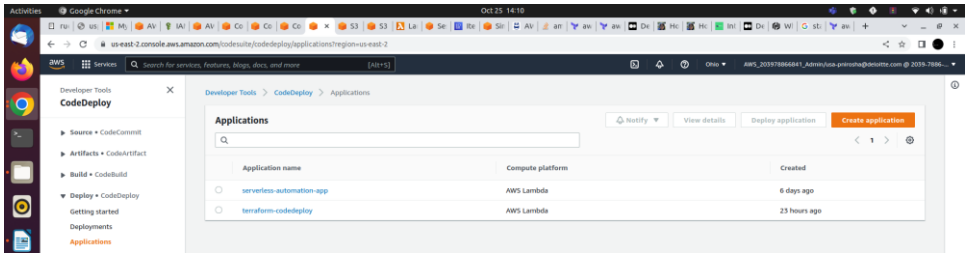


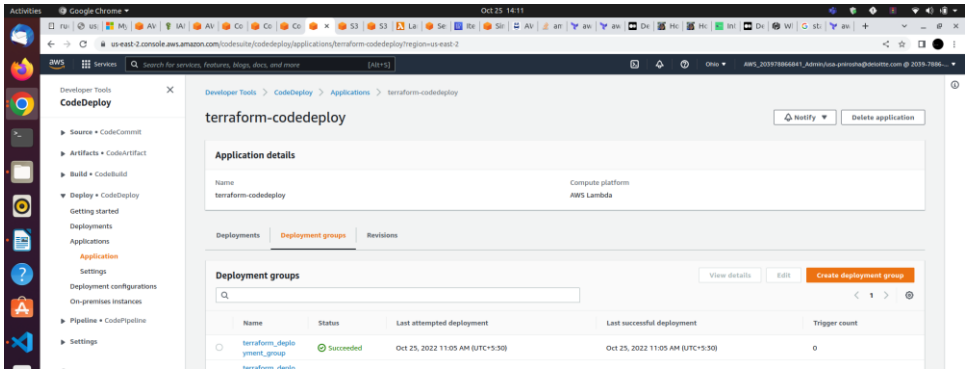
Codebuild project



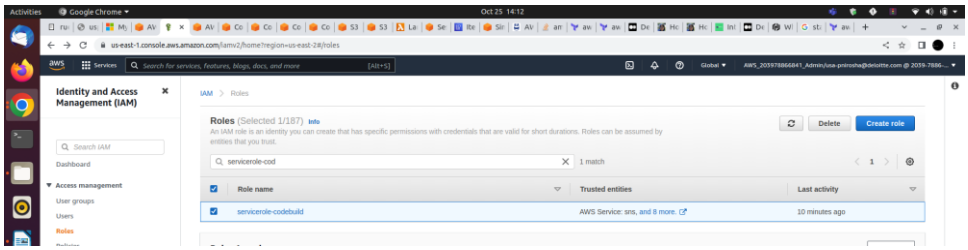
CodeDeploy Application,

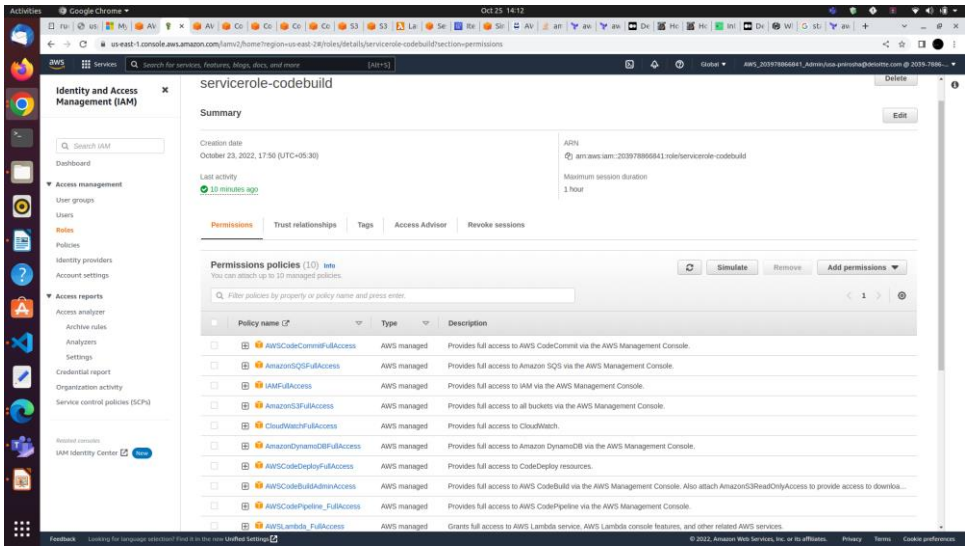
Note: Don’t add Codedeploy in the pipeline as a stage for now, we can configure this later, but we see now it will create a deployment group and application.



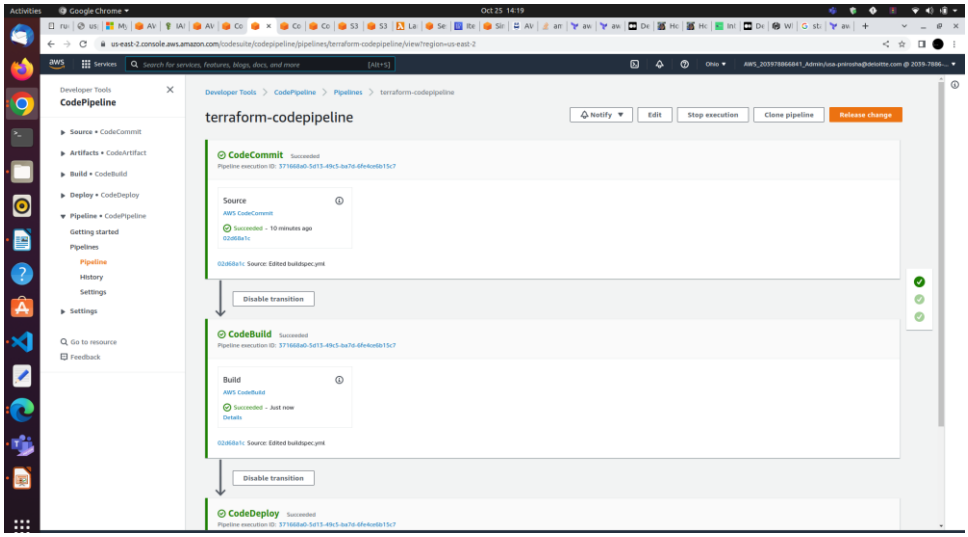


We can also check the IAM roles and policies that will created by terraform.

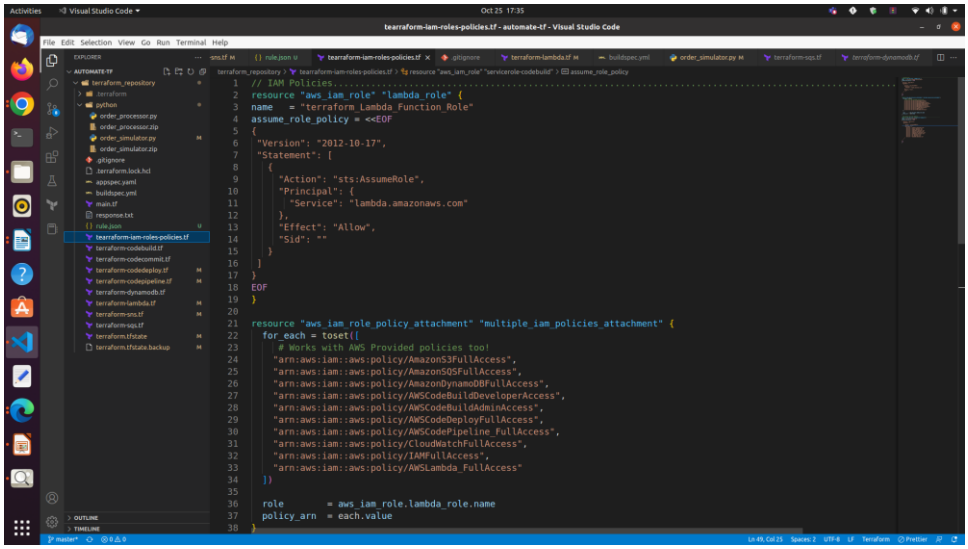




Now we can start the pipeline, Before that make sure the buildspec yaml file is there.

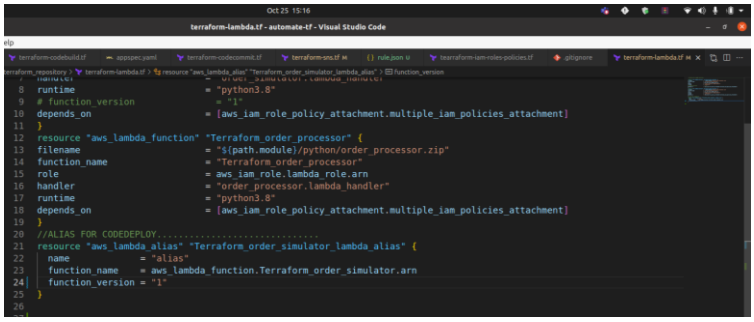


Step 8 : Before Creating a lamda functions, create a IAM role and add required policies to the lambda which will allows the lambda function to communicate other resources. So lets create a file, that it contain all the iam configuration for lamda function.



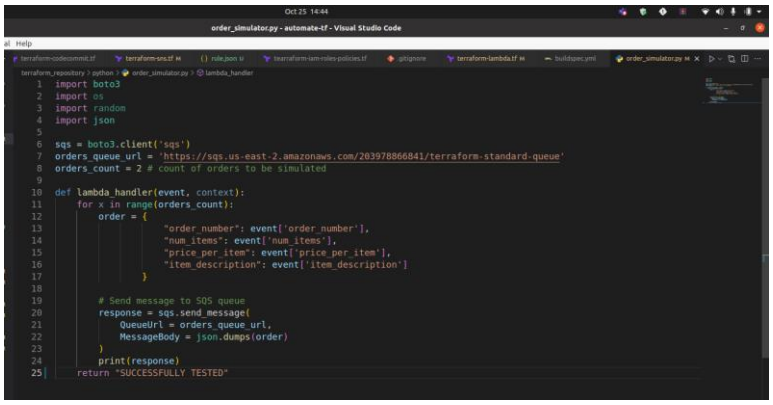
Step 9 : Create a configuration file that contains all the required configuration that can create a lambda function. Before that we need to zip our lambda function , for this i am using python3.8 for function code

Before that, here i have TWO lambda functions one is order_simulator.py and other one order_processor.py

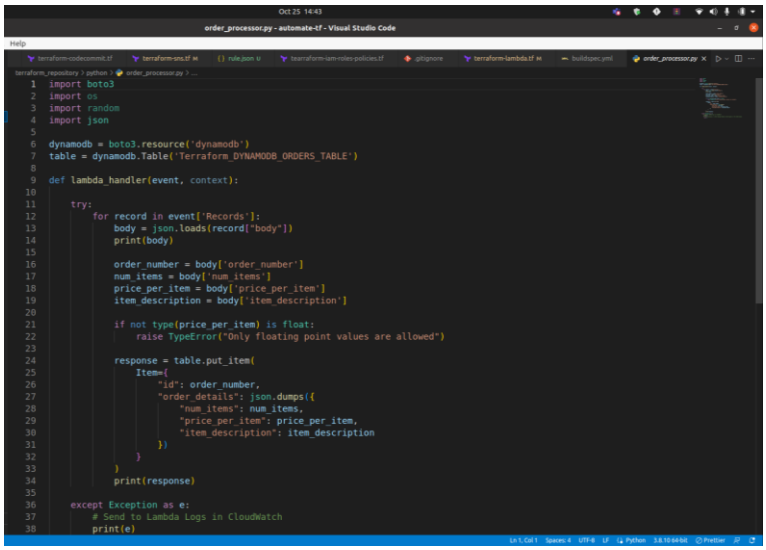


Let’s understand what all these function doing.

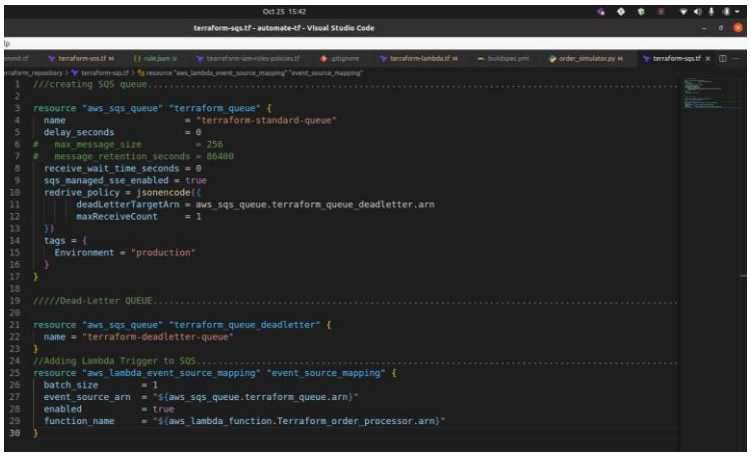
For order_simulator lambda functon – it will sends all our messages to the SQS with specified order. We need to mention QUEUE_URL...

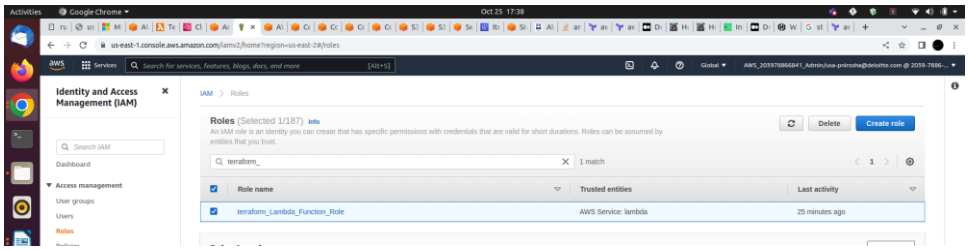


order_processor lamda function – it will process the messages that are stored in sqs and it sends to all the valid messages to the Dynamodb



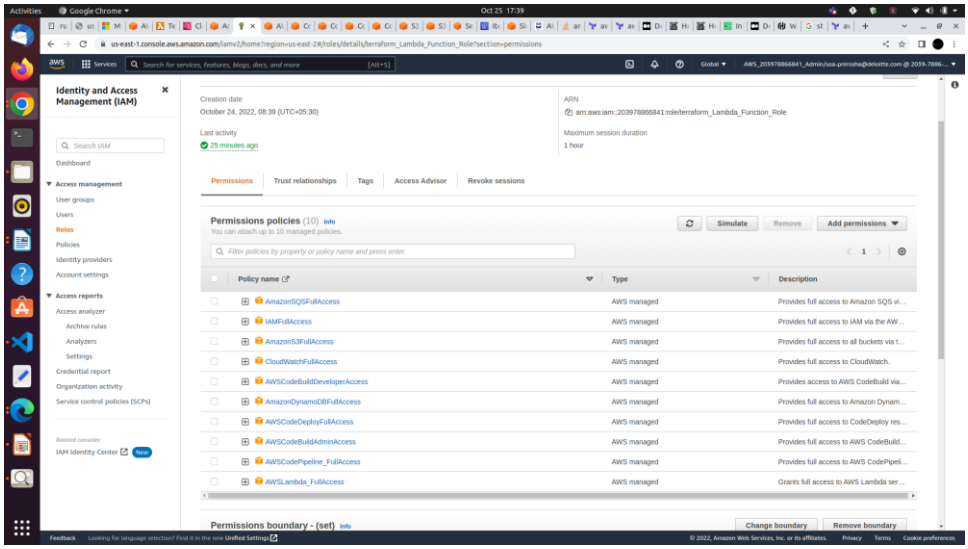
Step 10 : So Lets create an SQS queue and dead_letter_queue for storing invalid or bad data. For that create a configuration file which contians the sqs resource configuration.





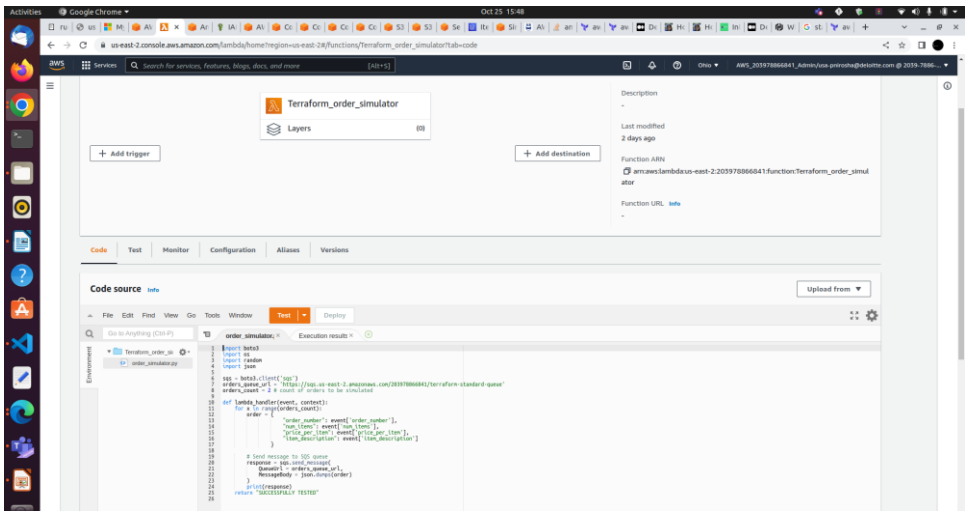
Now first go to IAM console, Click on roles and our terraform lambda role is created by terraform

Now click on the terraform_lambda_function_role, we can see all the policies are attached to the role which we specified in the configuration file.

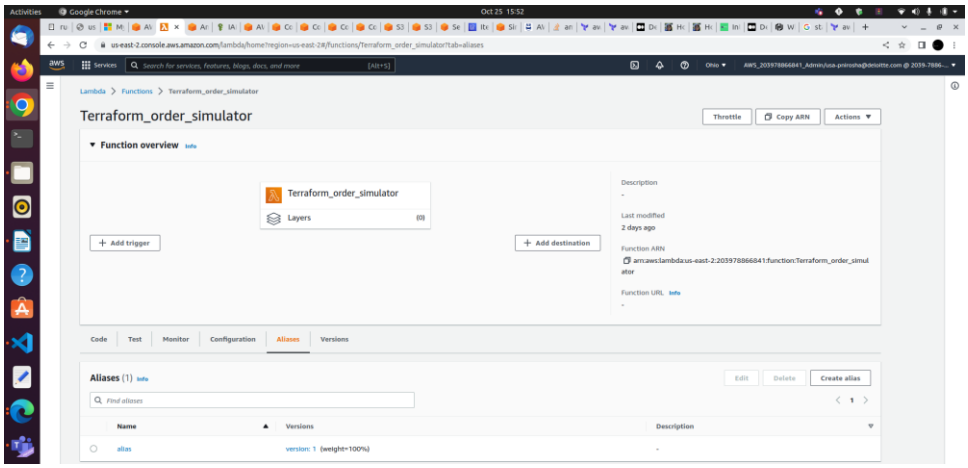


Now go to lambda console , terraform created two lambda function with s pecified configuration.

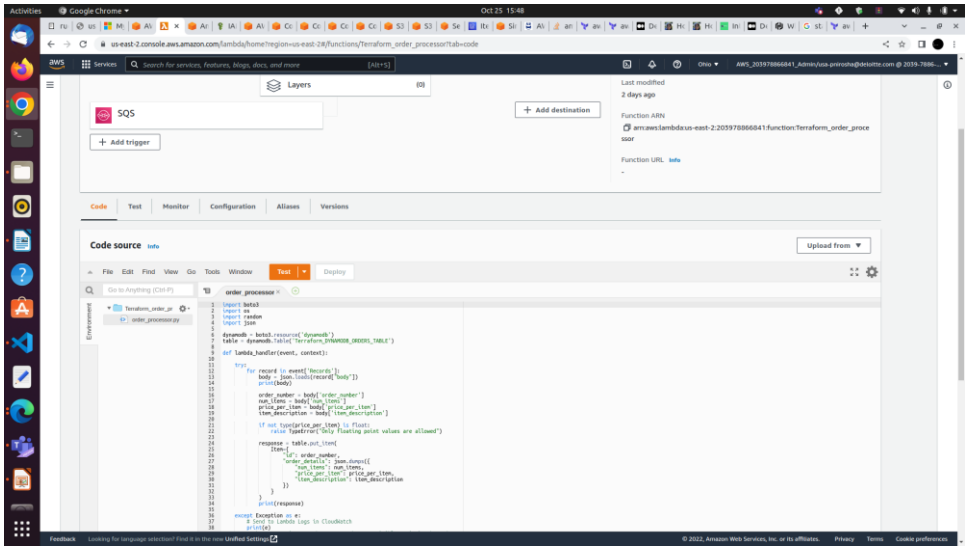
Now, Click the simulator function we can see our code.



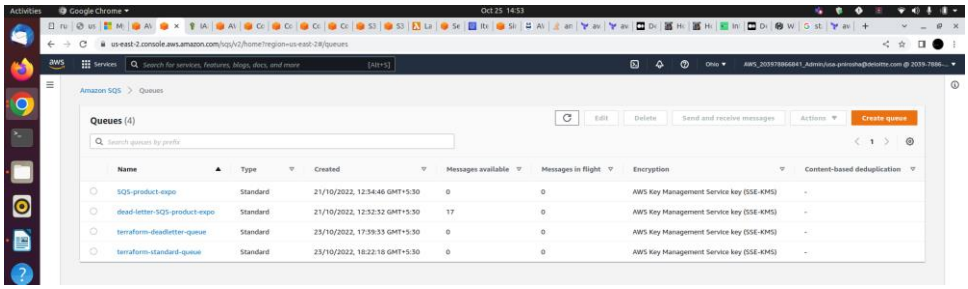
Now click alias , we can see our alias function also.



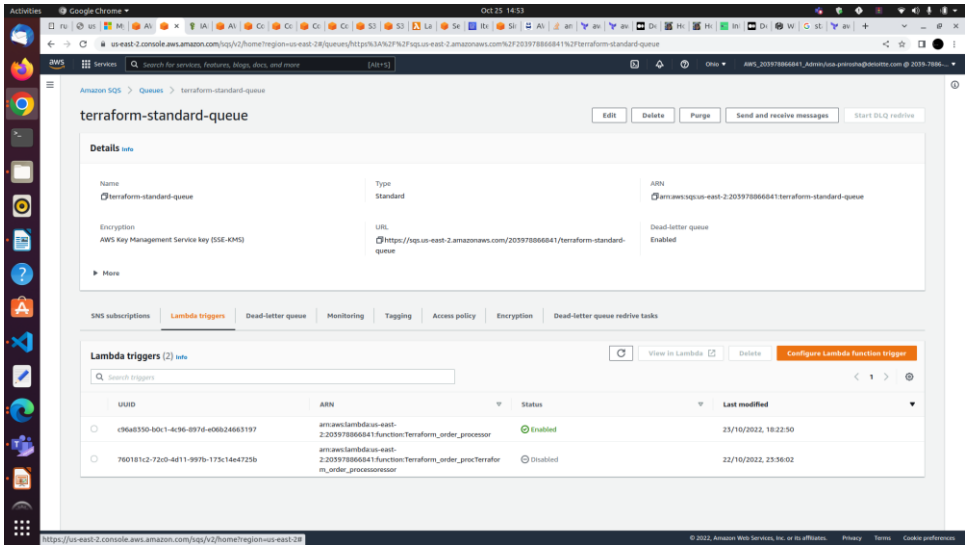
Now clicking our Terraform_order_processor function we can see that it added the sqs as trigger.



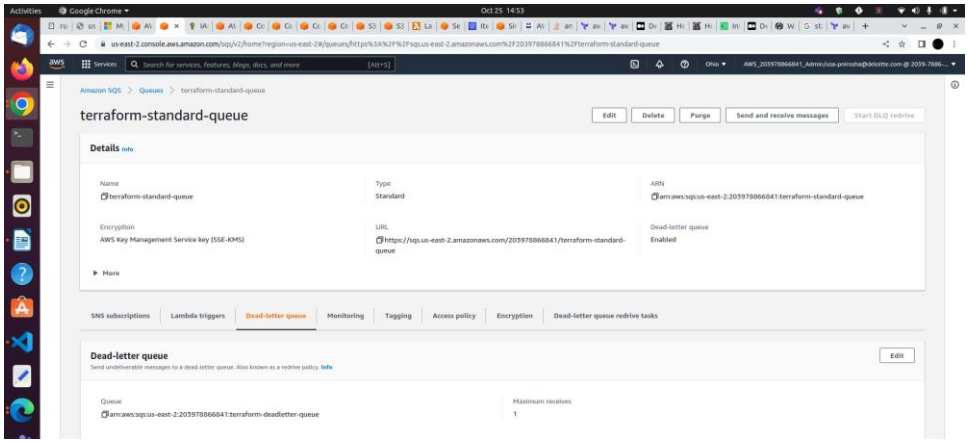
Now, Go to the sqs console, terraform is created two queues for us one for valid data.,



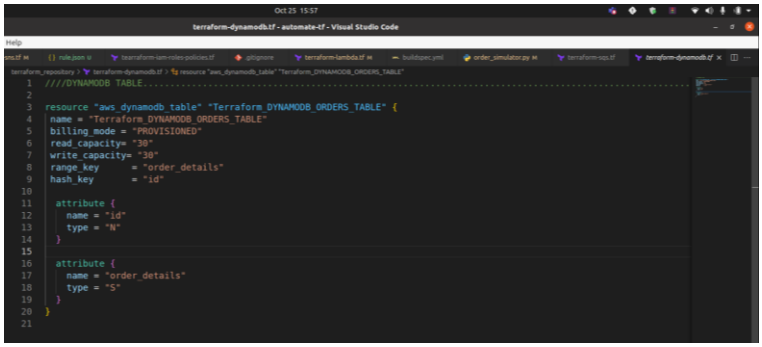
Now open the terraform_standard_queue, By clicking Lambda triggers we can s ee that terraform added the our order_processor lambda function



By clicking Dead letter queues, we can see that terraform added the our dead_lead_letter_queue

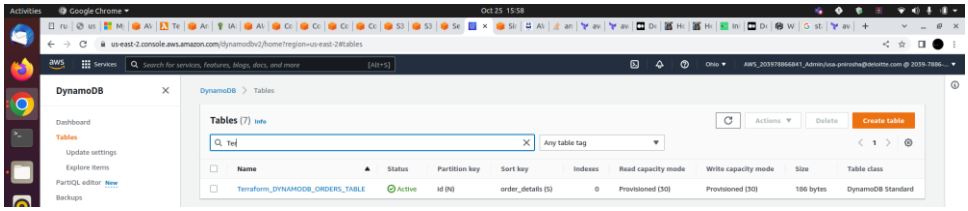


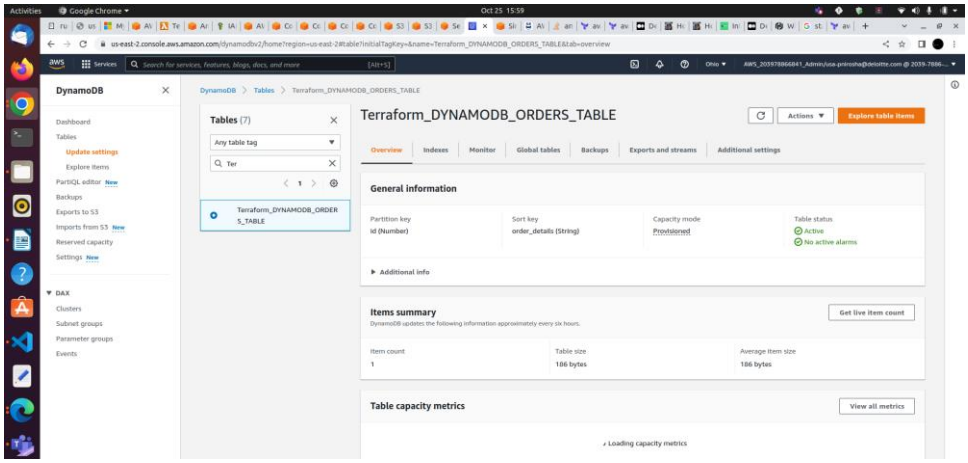
Step 11 : Now create a file for DynamoDB configuration and add the required attributes such as name attributes rangekey and hashkey..



```
////DYNAMODB TABLE
resource "aws_dynamodb_table" "Terraform_DYNAMODB_ORDERS_TABLE" {
 name = "Terraform_DYNAMODB_ORDERS_TABLE"
 billing_mode = "PROVISIONED"
 read_capacity= "30"
 write_capacity= "30"
 range_key      = "order_details"
 hash_key       = "id"

  attribute {
    name = "id"
    type = "N"
  }

  attribute {
    name = "order_details"
    type = "S"
  }
}
```

After Going to DynamoDB console, Click Tables,Now we can see our table

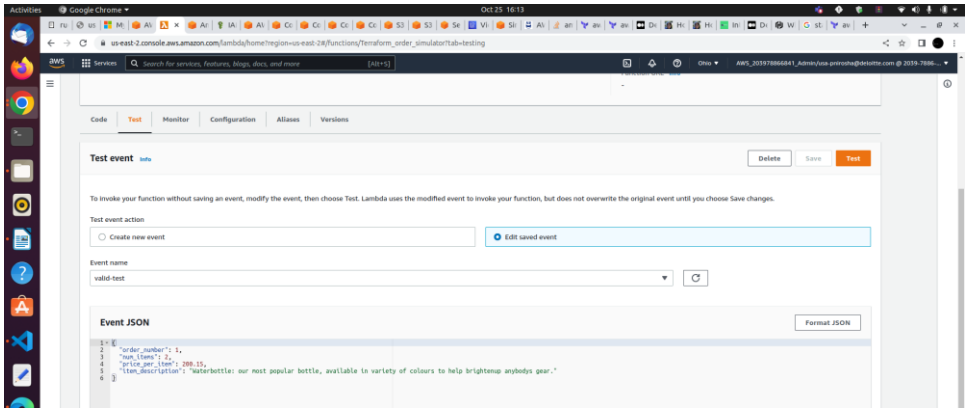




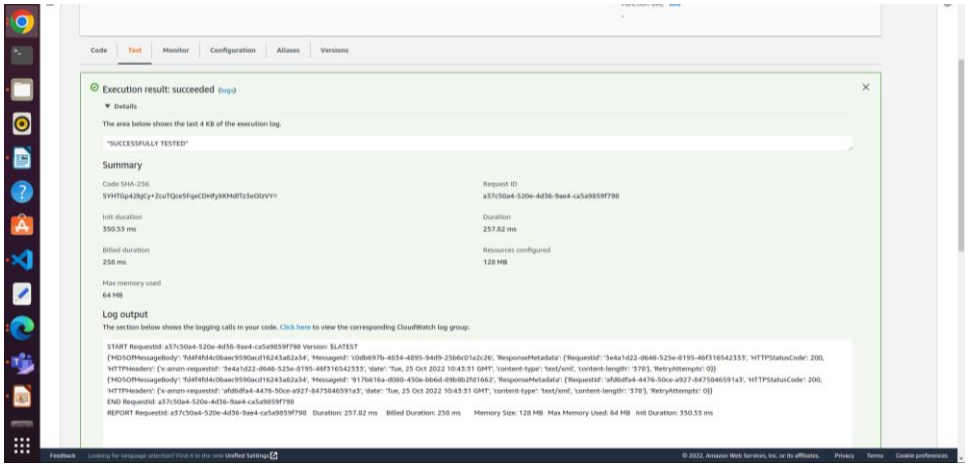
Step 12 : Now Test our lambda Function With valid and invalid input.

For this, Go to **Lambda Console**, -> click **Fucntions ->**

Open **Terraform_order_simulator** -> click **Test ->** Configure test event -> create a new event -> give a name and pass the valid json data -> click **Save** -> Click **Test**

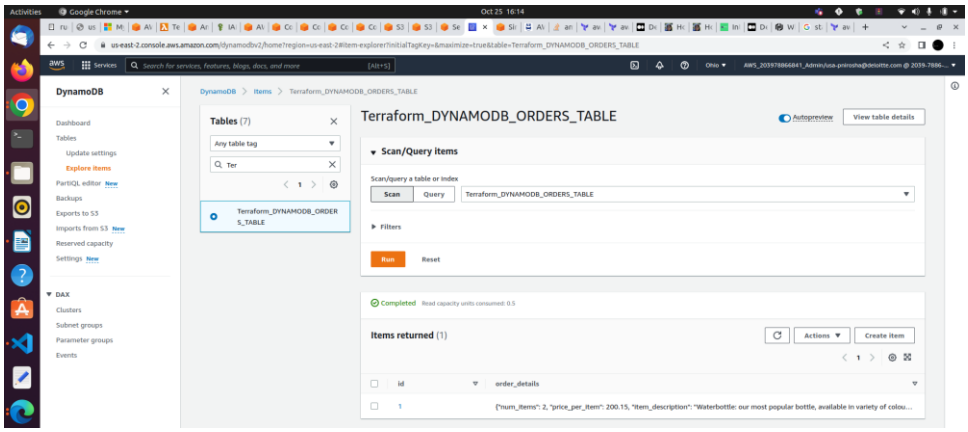


After Click on **Test**



If any Failures occurs while testing the function, make sure fuction has required iam policies to excute sqs and Dynamodb, make sure we are passing the correct SQS url and correct Dynamodb Table name in the function python code.

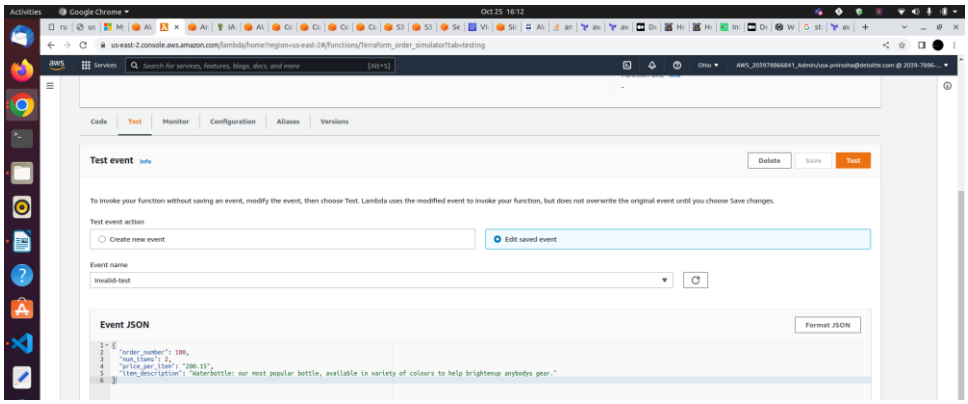
Now Go to **DynamoDB console** -> Click on **Tables** -> click on our table which is **Terraform_DYNAMODB_ORDERS_TABLE**



We can the our data is stored in dynamodb which is processed by our order_processor lambda function.

Now try to pass the duplicate or invalid data.

Here i am passing the **price as a string format**, which not allowed because we specified price as float value. So let’s try

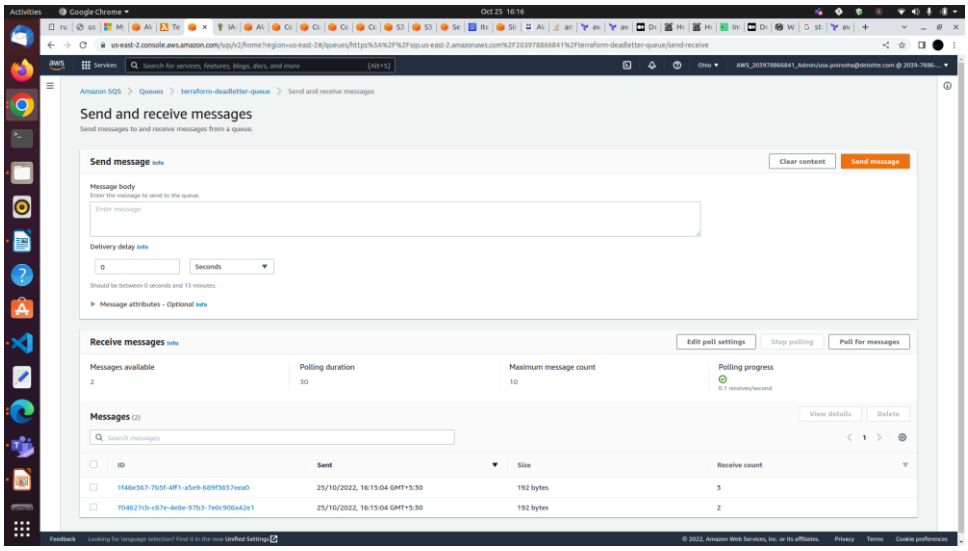


Click on save and Test.

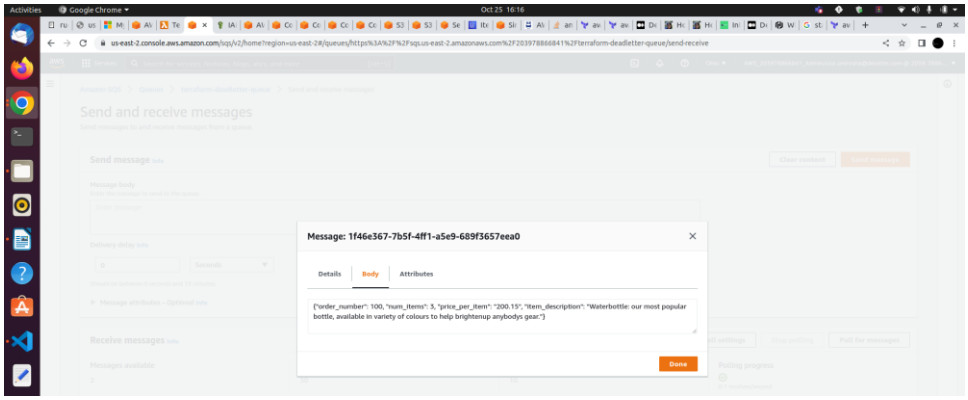
After Executing successfully,

Go to **SQS console** -> click on **Queues** -> click on our dead_letter_queue which is **Terraform-dead-letter-queue** -> Click on **Send and receive messages** -> Now at the bottom we can see poll for messages Click on **poll messages**

If any failures occcures, like not appearing the messages to the dead letter queue, try to increase the latency of standard queue. latency for Dead letter queue is minimum 10. maximum receives parameter had 10 value, So it was proccessing 10 times.. 11 time if not processed it puts to DLQ. Change the l atency to one , then you can see immadiate invalid data in dead-letter queue.

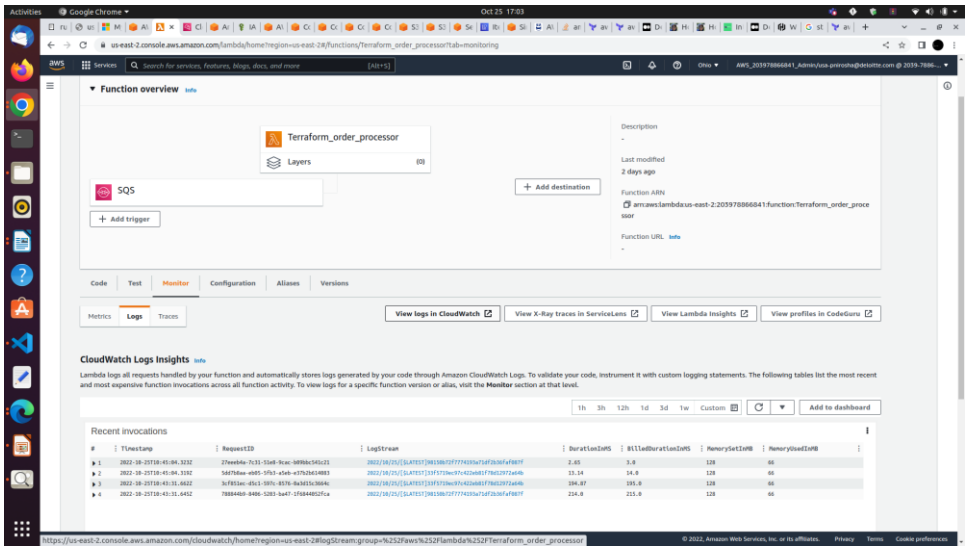


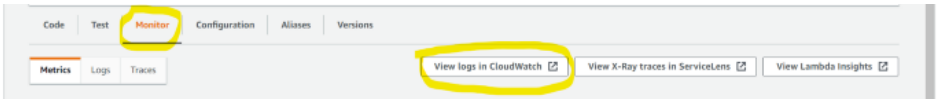
We can see our invalid values are stored. Now click on one messages to see our v alues. After click on Done.

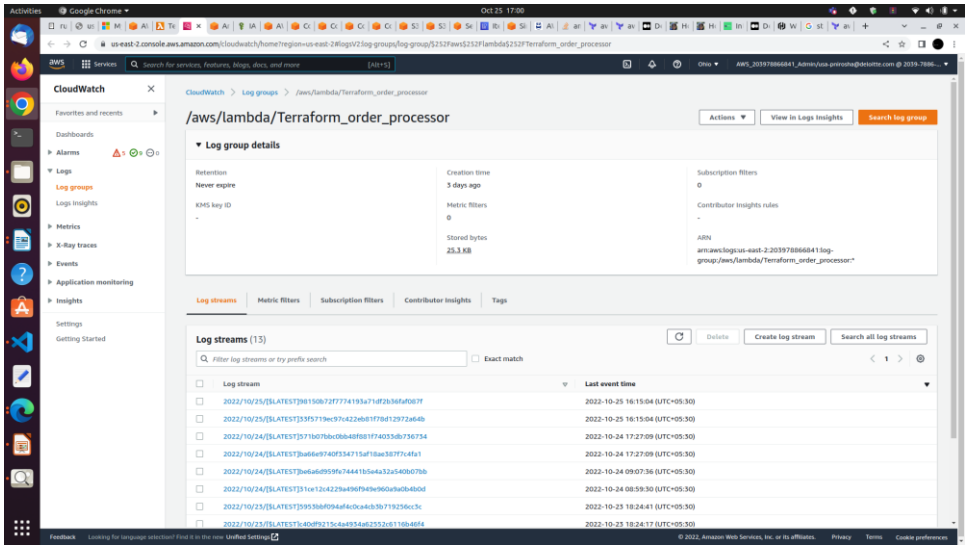


Also, we could check Cloudwatch logs for the information being logged there:

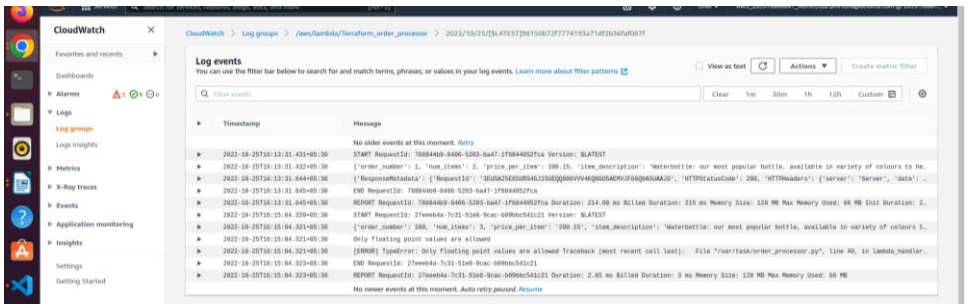
We can check by directly going to cloudwatch console or By Going back to the lambda console and click on the '**Monitor**' tab. Then click on the '**View logs in CloudWatch**' button. This will open a new tab and take you to CloudWatch.



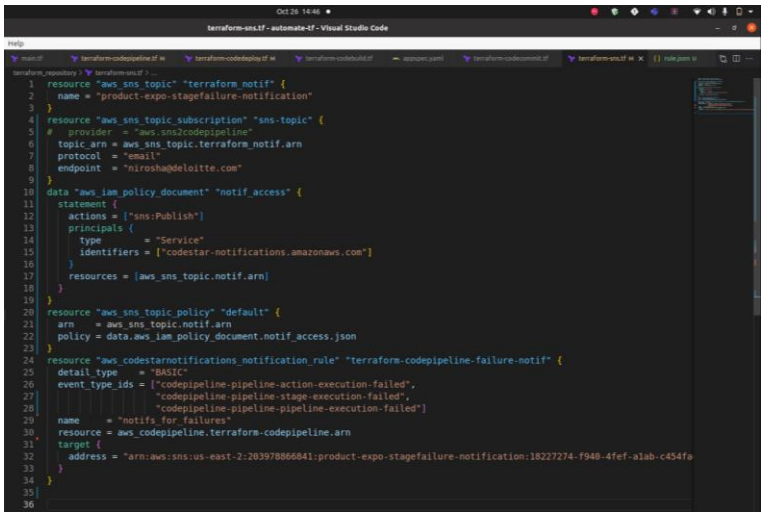




By clicking latest event logs we can see all the things that are proccessed by a order_procesor lambda function.

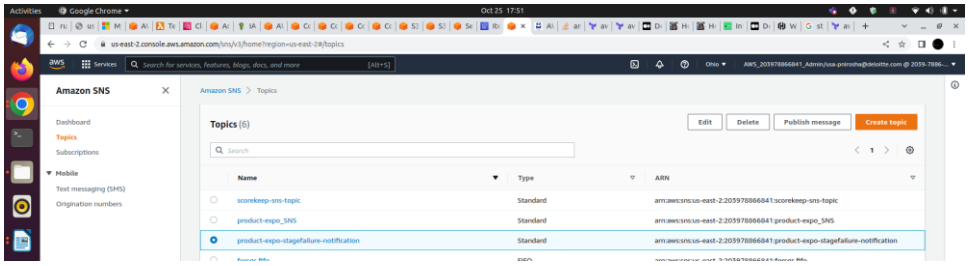


Step 13 : Create a configuration file which will create an SNS Topic and Subscription to the pipeline.

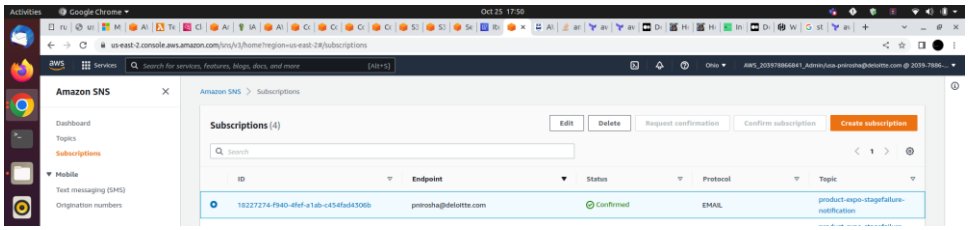


Now Go to **Simple Notification Service(SNS) console** -> Click on **Topics**

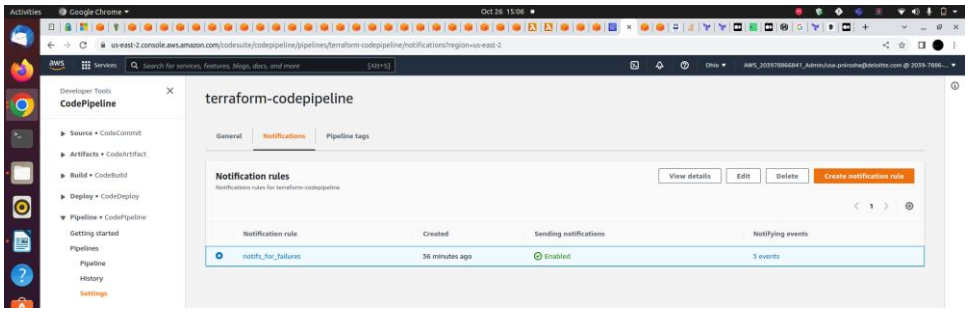
We can see that terrform created an sns topic for us.



Now For subscription, it will send a subscription mail to us, by clicking subscribe in the mail, it will add to the subscription status as confirmed.



Now By visiting our codepipeline, Click on **Notify** -> click on **manage notification rule** -> click on **Notifications** we can a rule that is created. It will send a notifications based on the events that we cofigure in the terraform file, for this it will sends a notifications to our mail whenever the pipeline execution fails.



Developers can be notified via email if their build fails It will be usefull whenever we are working on teams.

So Finally, We completed the all the things that are in the below serverless architeture.

So what are the usecases of it,

Use cases of Decoupled serverless architecture

Use case 1:

If a team has been given a mandate to recommend an upgrade for the backend architecture to prepare for the upcoming Thanksgiving sale. Historically, our company has seen a 5X traffic spike for e-commerce for the duration of this sale. In the past, there have been cases where the system went down and customer orders were lost, leading to significant revenue loss for the company. We need you to devise a decoupled serverless backend architecture that is able to handle the spike in traffic for the duration of the sale.

This solution, we will leverage an SQS standard queue to send, store, and receive order messages and then configure the SQS Event Source to trigger the Lambda function which in turn processes the orders into a DynamoDB table.

use case 2:

The Engineering team at a company has raised an issue. Every time a developer commits their code to the repository, they have to manually trigger a single EC2 instance-based Jenkins pipeline to build and test the code. Often, the Jenkins server slows down and stops responding, resulting in a lot of time wastage. This approach is a time-consuming task for the developers. They would just like to focus on pushing their application code to the Git repository, and the rest should be taken care of automatically by a more stable pipeline.

By creating an automated pipeline using AWS services that can achieve this. In addition, developers should be notified via email if their build fails.