CI/CD Configurations for Backend Applications using Cross account

Agenda:

CodeBuild:

AWS code build is a fully managed continuous integration service that compiles source code, runs tests, and produces software packages that are ready to deploy. With code build, you don't need to provision, manage, and scale your own build servers. code build scales continuously and processes multiple builds concurrently, so your builds are not left waiting in a queue. You can get started quickly by using prepackaged build environments, or you can create custom build environments that use your own build tools.

CodePipeline:

AWS code pipeline is a fully managed continuous delivery service that helps you automate your release pipelines for fast and reliable application and infrastructure updates. code pipeline automates the build, test, and deploy phases of your release process every time there is a code change, based on the release model you define. This enables you to rapidly and reliably deliver features and updates. You can easily integrate AWS code pipeline with third-party services such as git hub or with your own custom plugin.

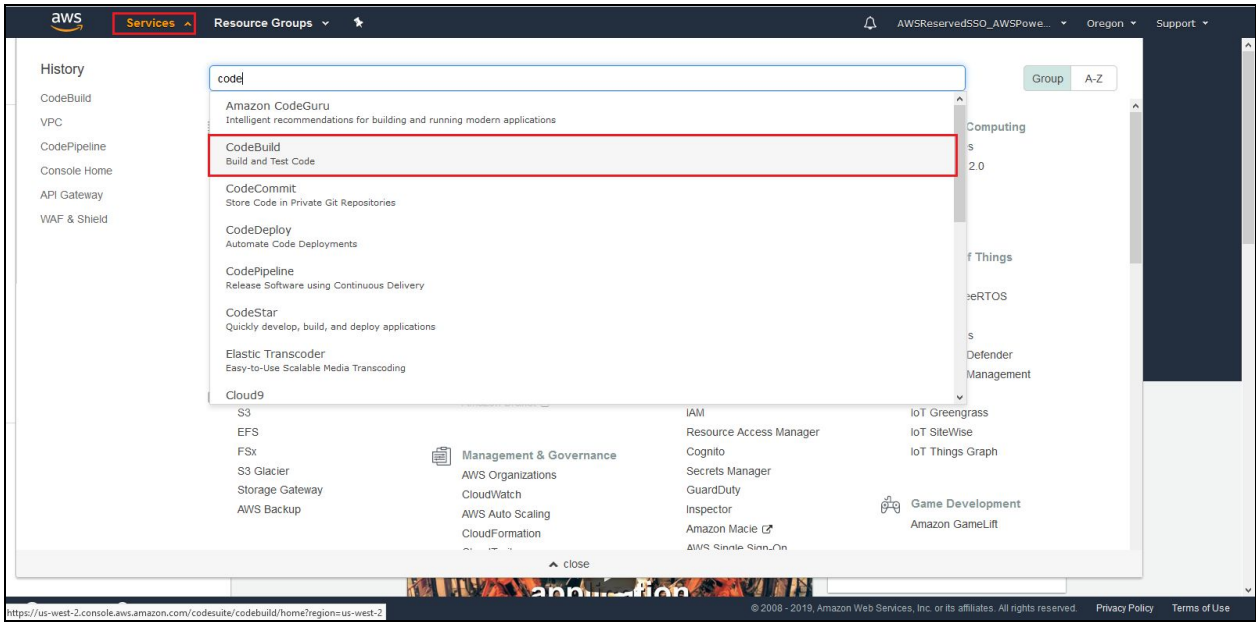
Serverless Deployment:

The serverless framework was designed to provision your AWS lambda functions, events and infrastructure resources safely and quickly.

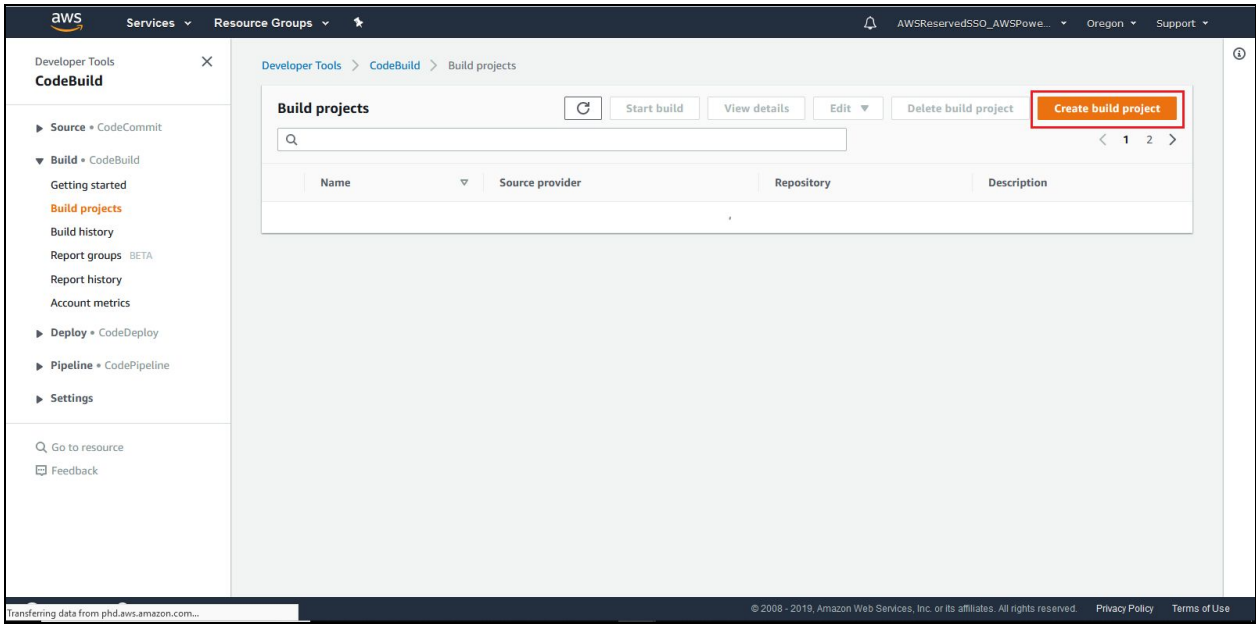
The serverless framework translates all syntax in **serverless.yml** to a single AWS cloudformation template. By depending on cloudformation for deployments, users of the serverless framework get the safety and reliability of cloudformation.

By the end of the document, we learn how to create a CI-CD pipeline for NodeJs and ASP dotnet core applications using cross account.

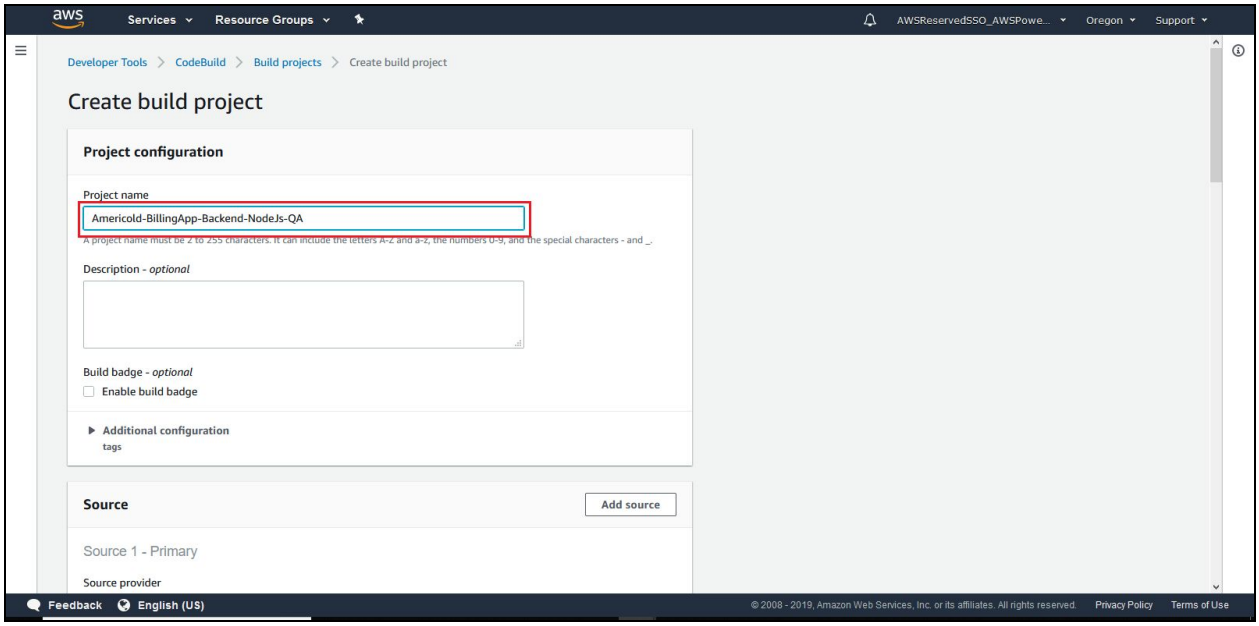
First we need to login into the AWS console and goto **code build** service.



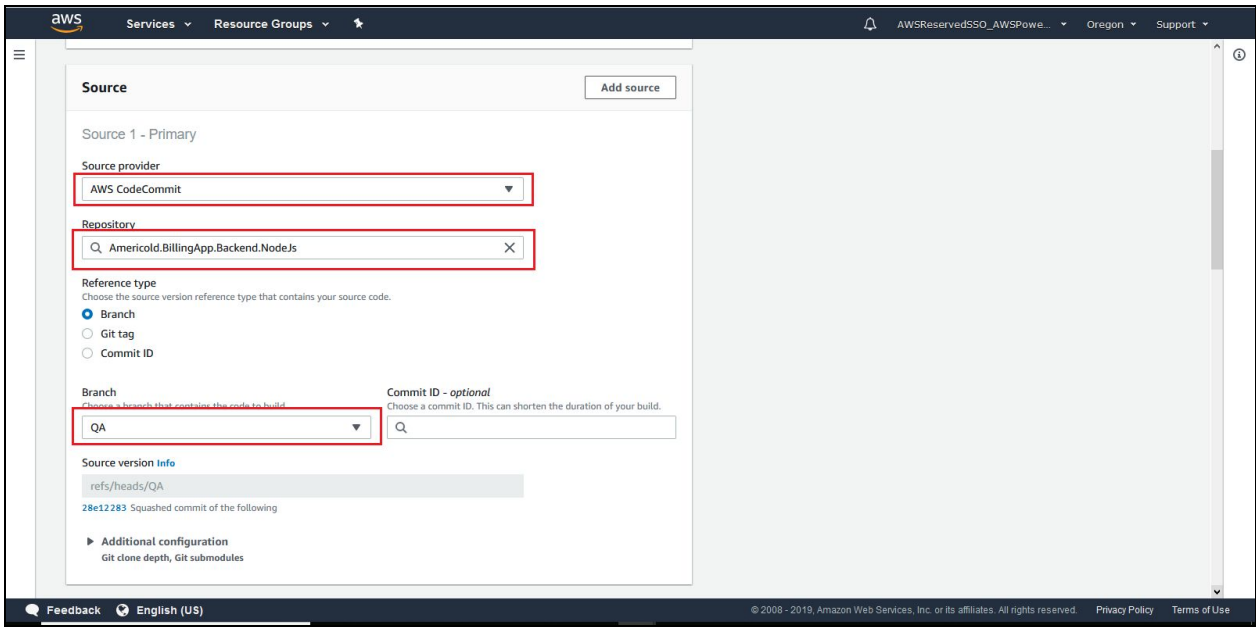
Goto build projects and click on **create build project** to create a new build project for integration**.**



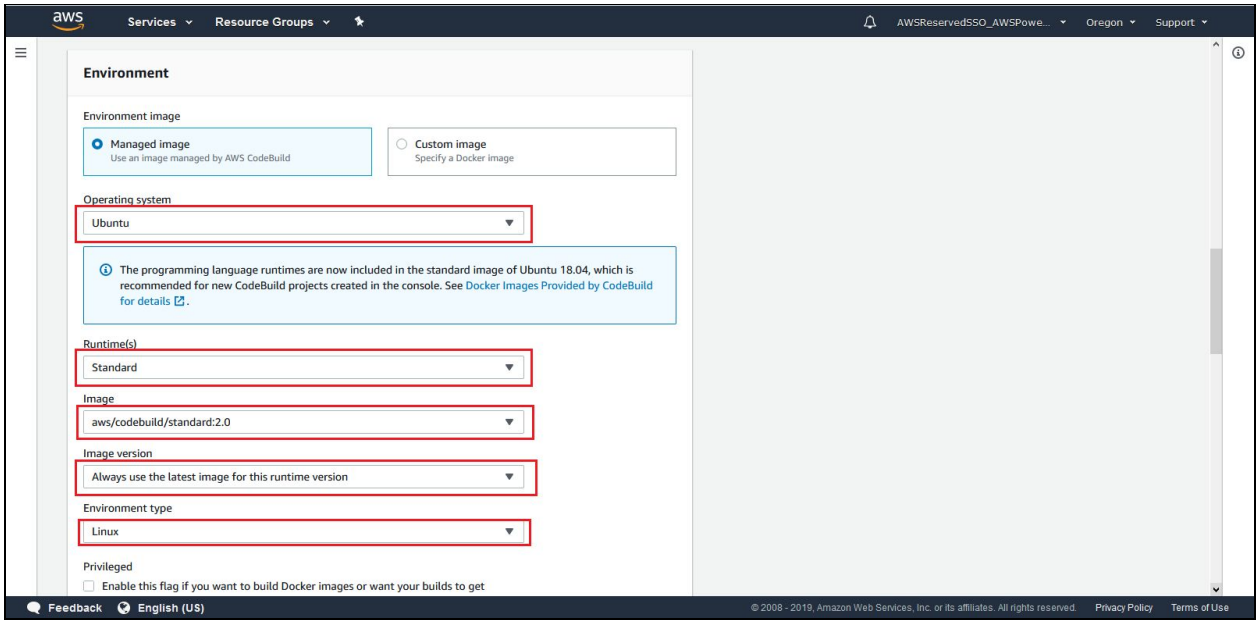
Enter the name of the project**.**

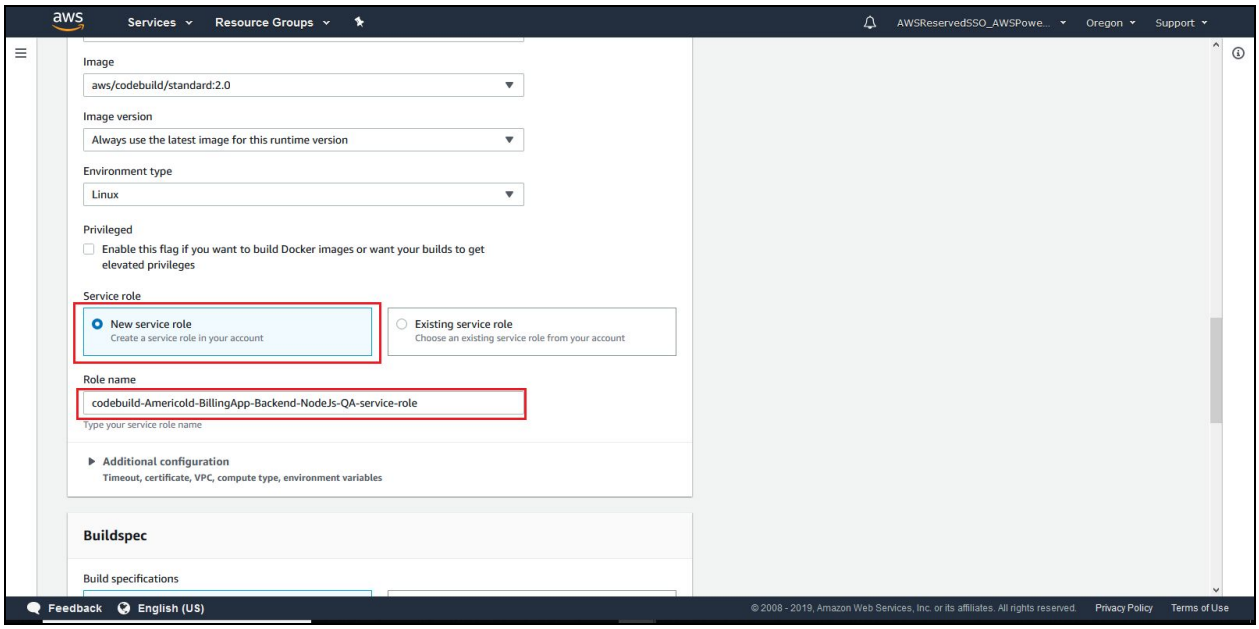


Under the source section, select the source provider as **AWS code commit**, and select the repository and branch as shown below.

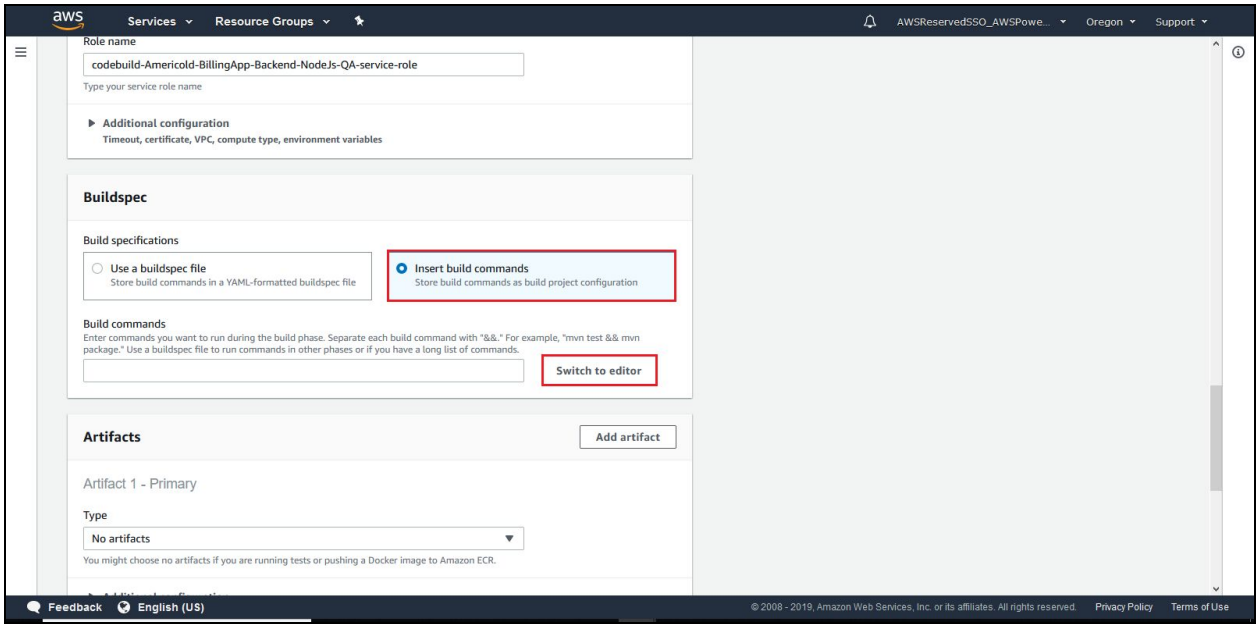


Under environment section we took the managed image that is provided by AWS, operating system as Ubuntu,runtime as standard, image as standard 2.0 version, image version as always latest, environment type as linux and created new service role for the code build .

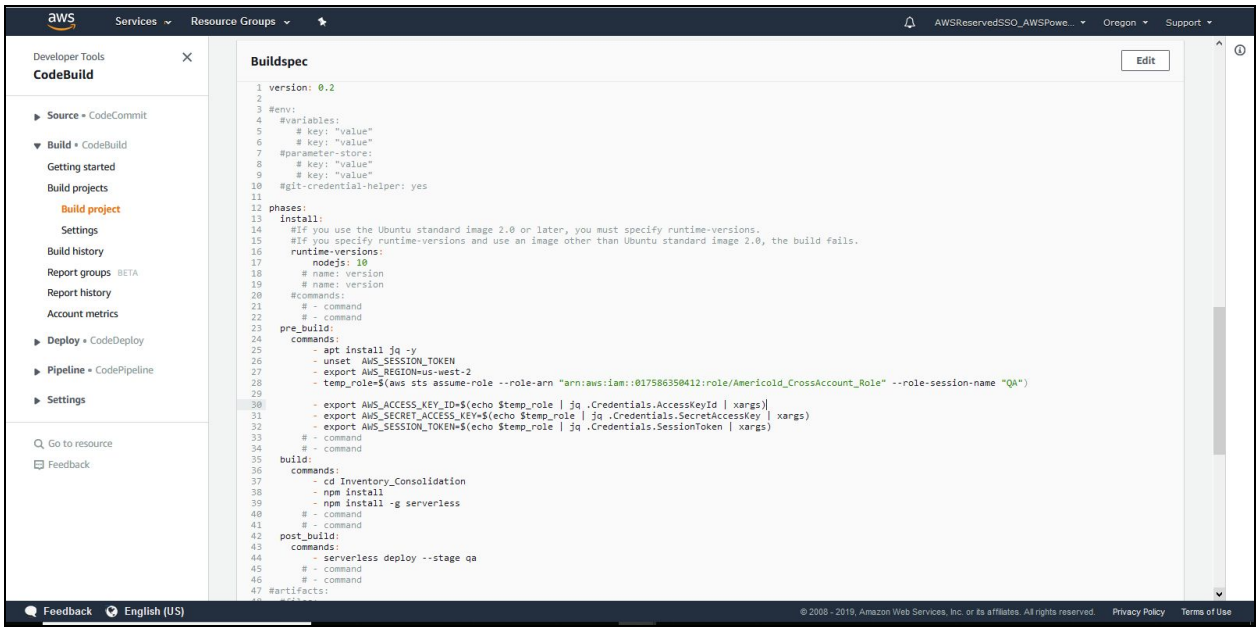




Under the build spec section, choose an option as **insert build commands** to edit the **buildspec.yml** file from console. buildspec.yml file has a set of commands for integration and deployment.



Configure the respective stages as per requirement in **buildspec.yml file** as shown below.



Below is the step by step explanation for the above script:

Install stage:

- **runtime-verions:**

  nodejs : 10: It defines the runtime environment and version for the application

Pre_build stage:

- **apt install jq –y:** Install node modules using the following command
- **unset AWS_SESSION_TOKEN:** Clear the AWS session token
- **export AWS_REGION=us-west-2:** Set the AWS region
- **temp_role=$(aws sts assume-role --role-arn "arn:aws:iam::017586350412:role/AmericoId_CrossAccount_Role" --role-session-name "QA") :** Create temporary AWS IAM role and store in the variable as shown above
- **export AWS_ACCESS_KEY_ID=$(echo $temp_role | jq .Credentials.AccessKeyId | xargs)**

  export AWS_SECRET_ACCESS_KEY=$(echo $temp_role | jq .Credentials.SecretAccessKey | xargs)

  export AWS_SESSION_TOKEN=$(echo $temp_role | jq .Credentials.SessionToken | xargs)

  Set the AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, AWS_SESSION_TOKEN variables with their respective values using jq tool as shown above

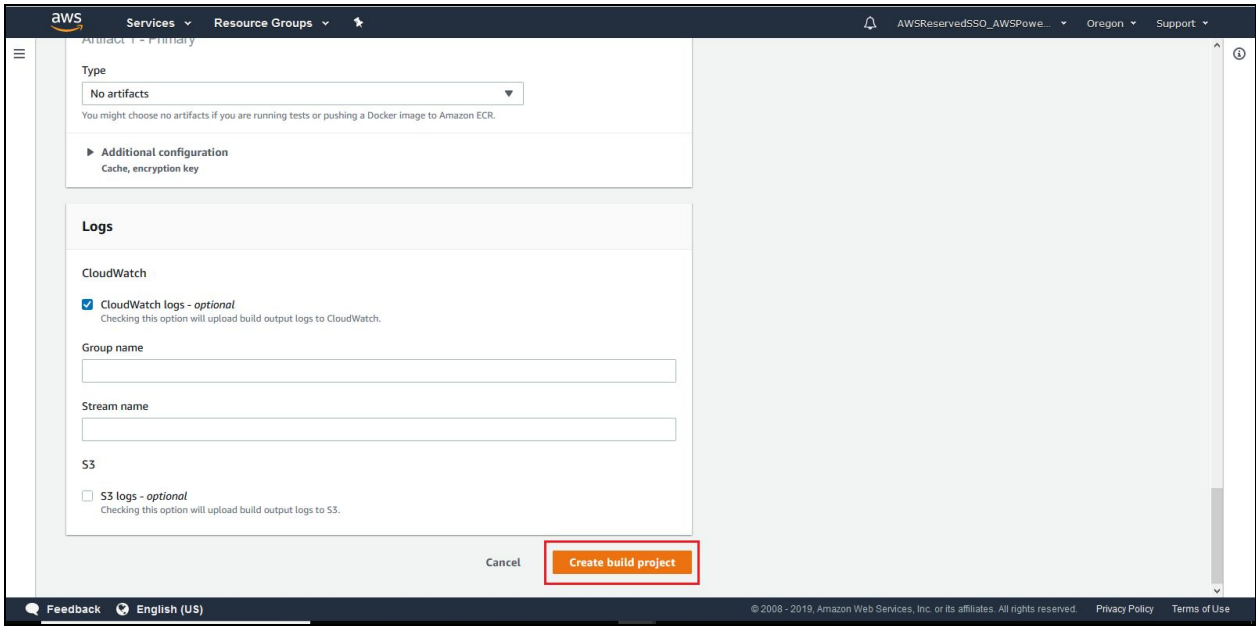
Build Stage:

- **cd Inventory_consolidation:** Go to the folder called Inventory_consolidation to install the necessary dependencies for integration.
- **npm install:** Install node modules using the following command
- **npm install -g serverless:** Install serverless module using below command

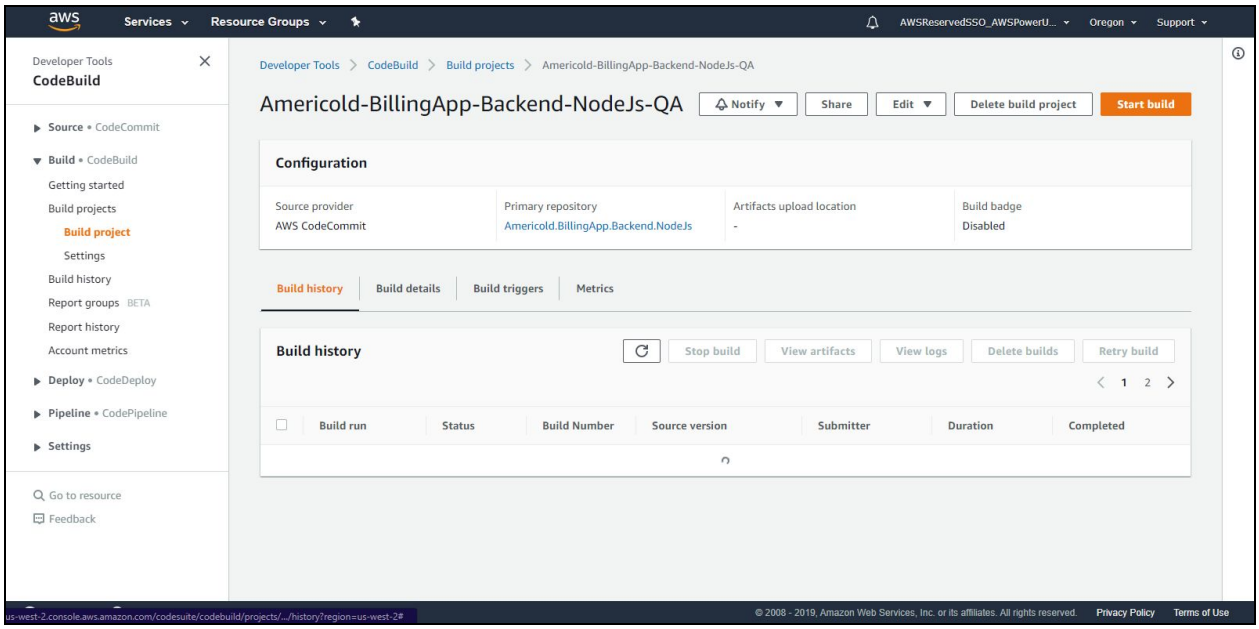
Post-Build stage:

- **serverless deploy --stage qa:** To deploy an application using serverless.yml template file we will run the following command

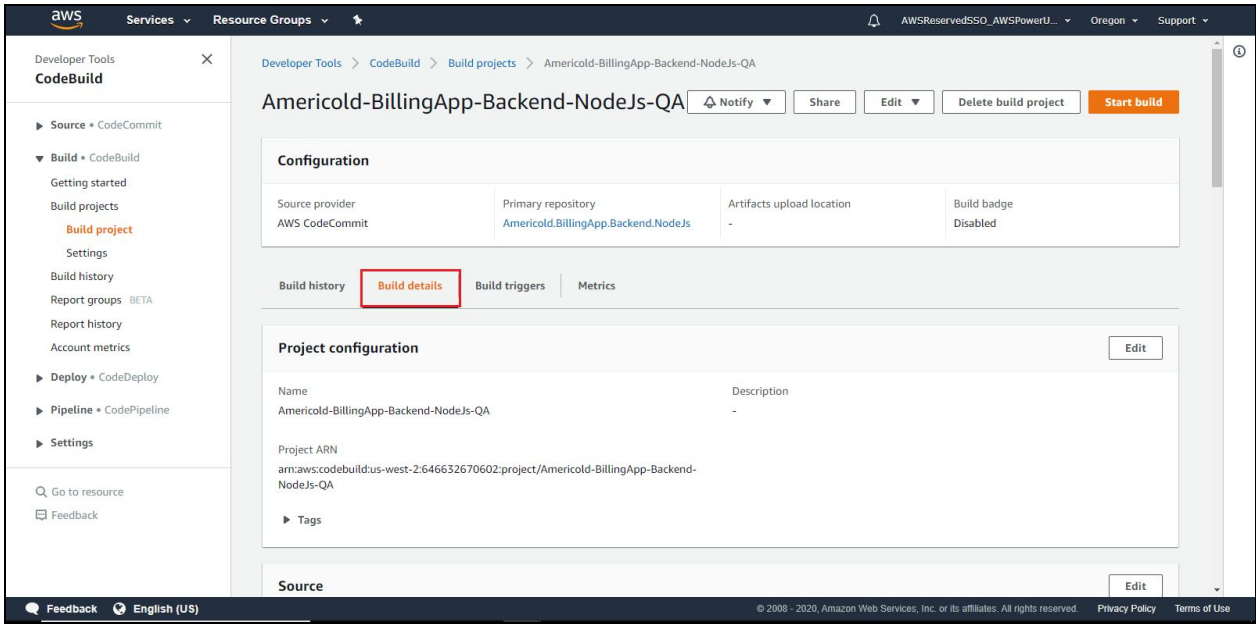
We are enabling cloud watch logs to check the build logs. Once all the configurations were done, click on **create build project.**



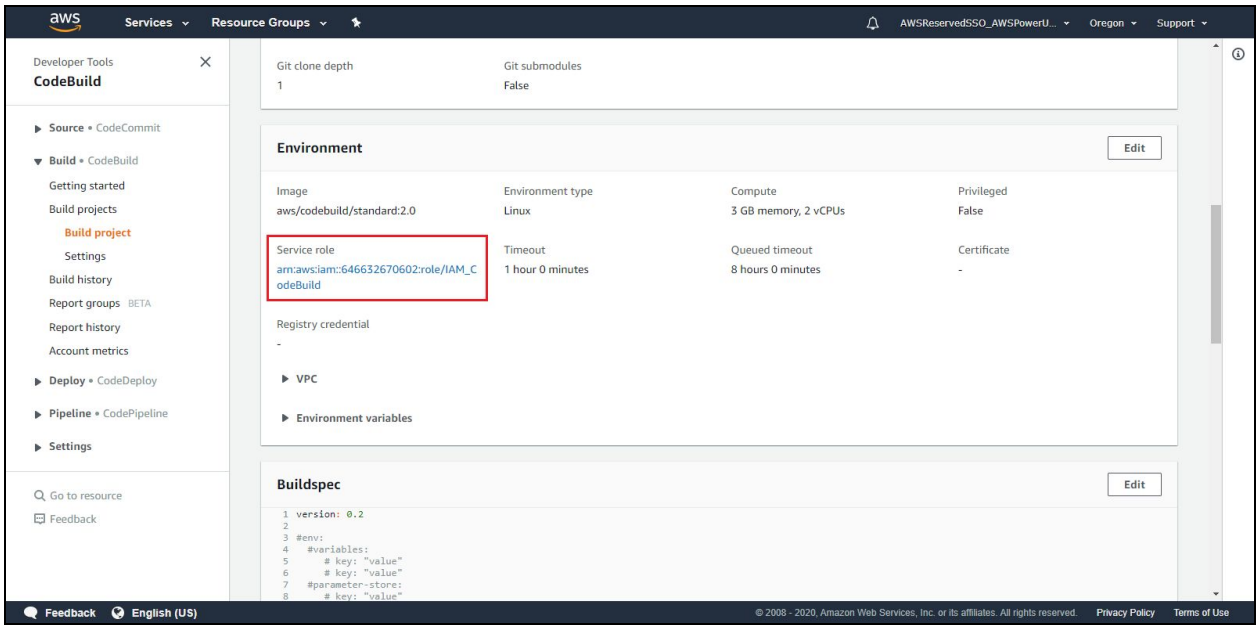
The new build project has been created and the dashboard is shown as below.



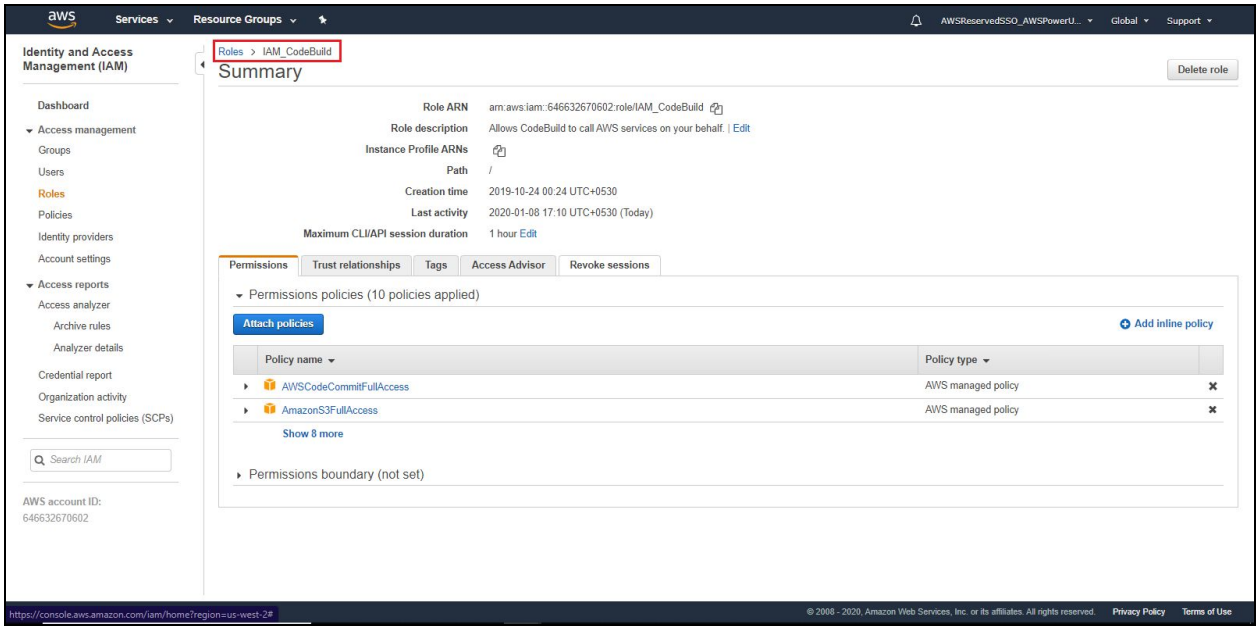
Goto details and goto **environment section** in build details.



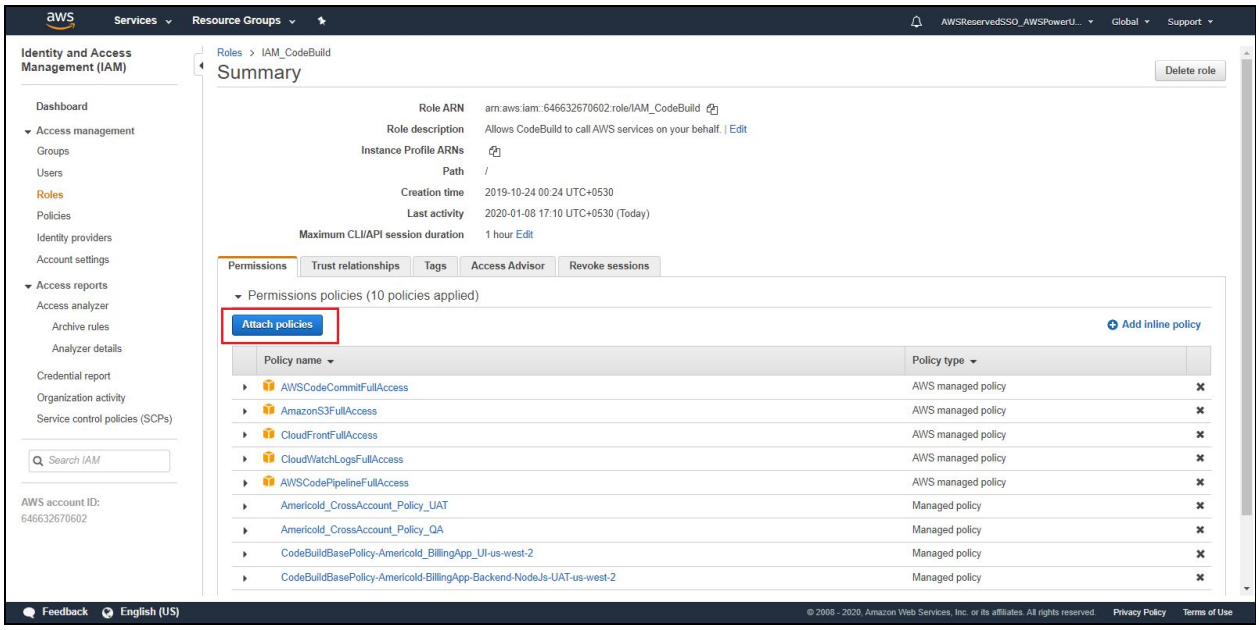
Click on service role which is new service role created while configuring code build. It will redirects us to **IAM console**.



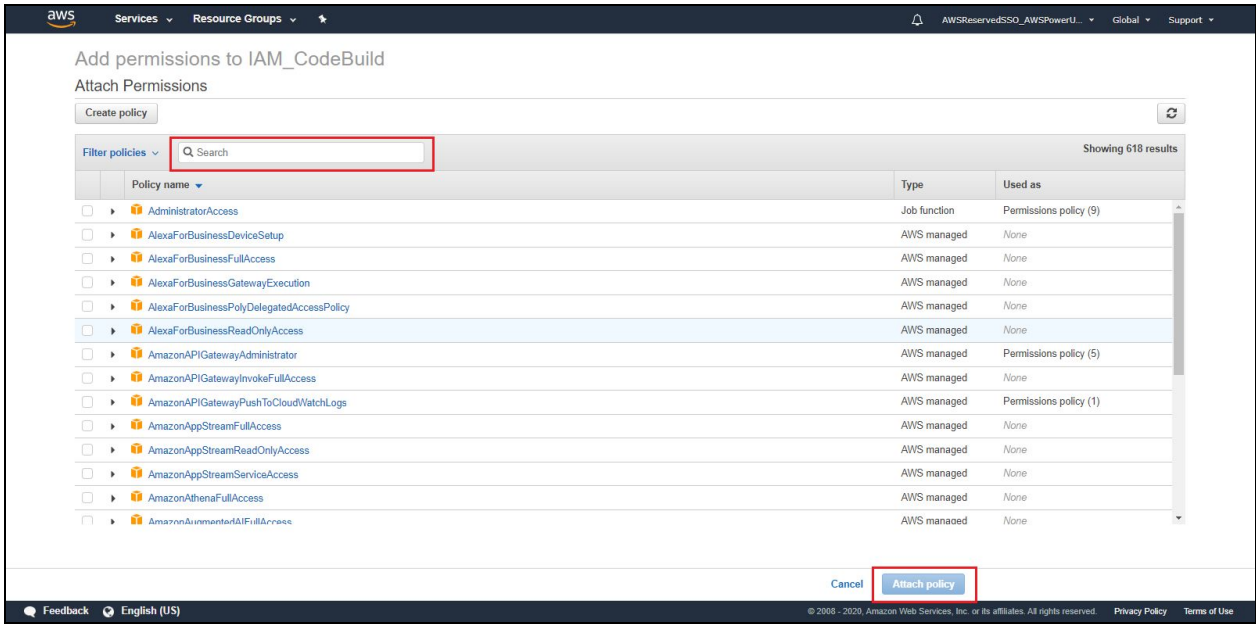
The IAM role dashboard will be as shown below and we need to add the required policies to access the AWS services for application deployment through code build.



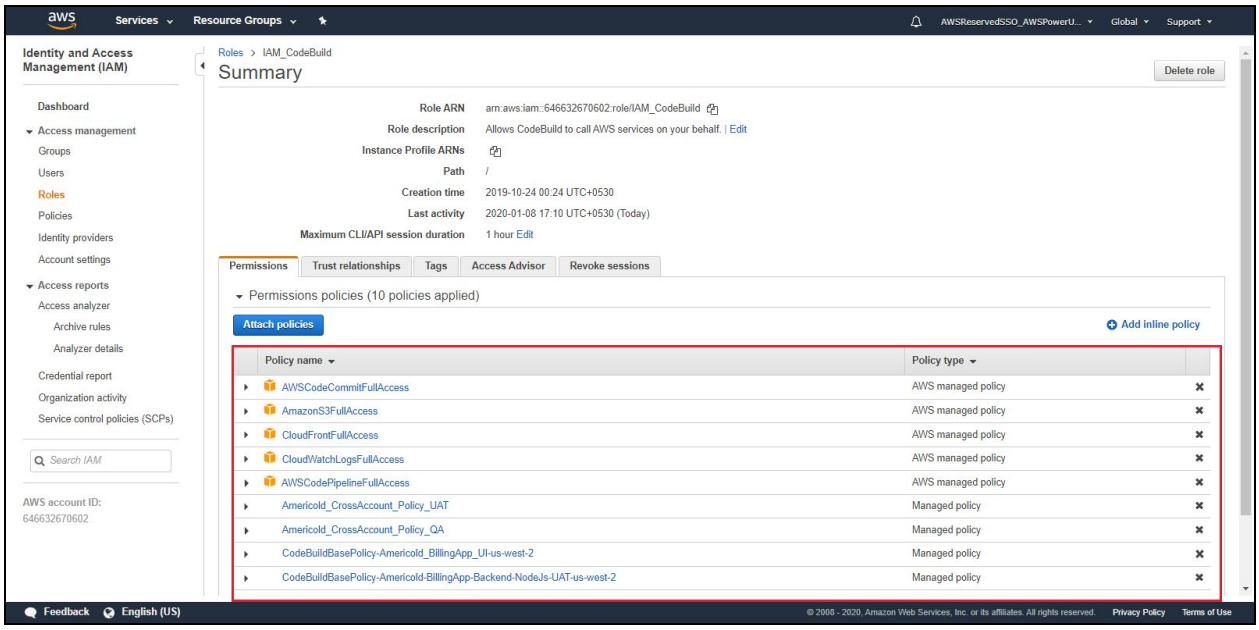
Click on **Attach policies** to add the policies for the IAM Role.



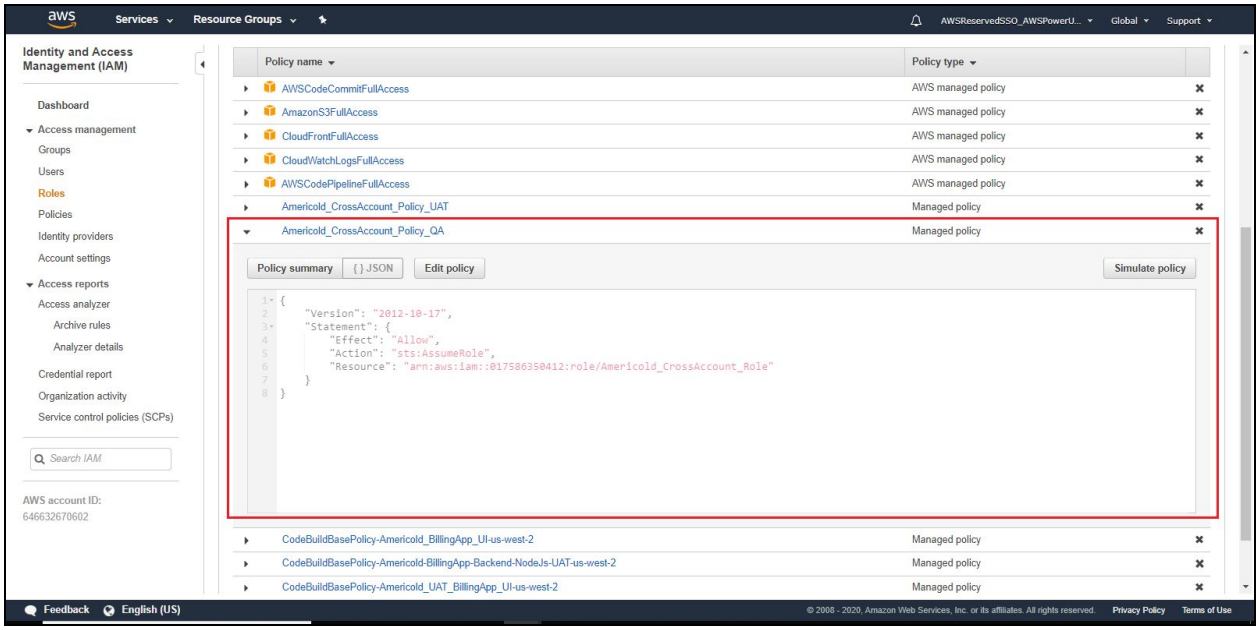
Search for the required policy and select the check boxes and click on Attach Policies.



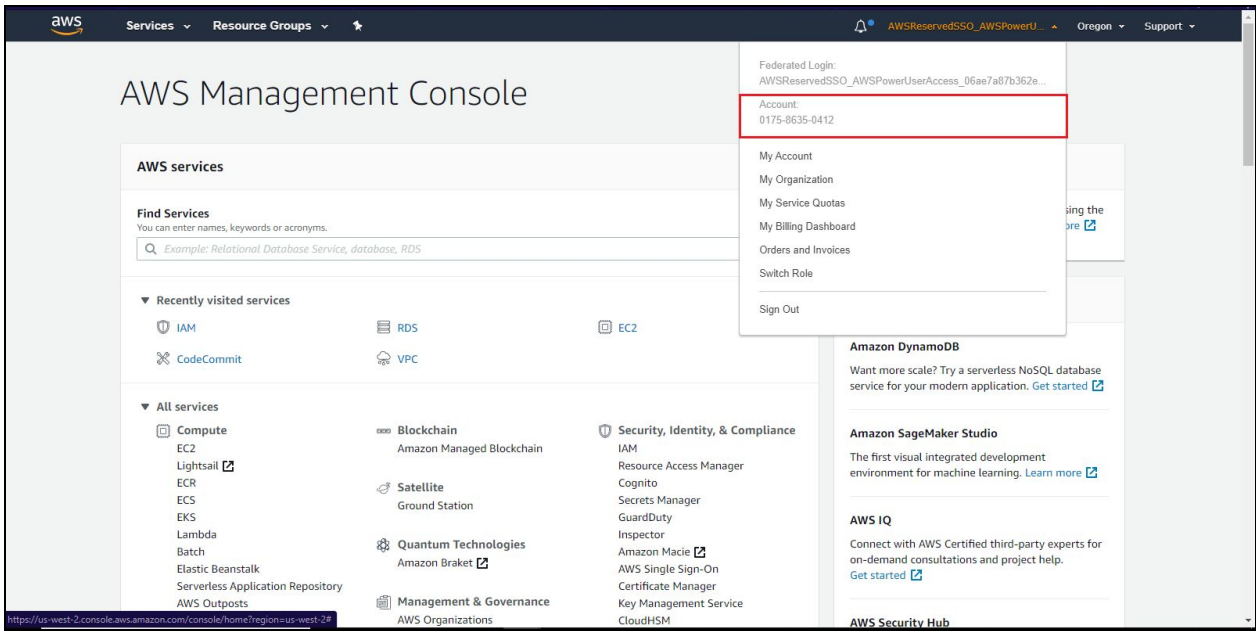
Here we have attached the policies for **Code commit, S3, cloud front, cloud watch logs, code pipeline and Cross account policy.**



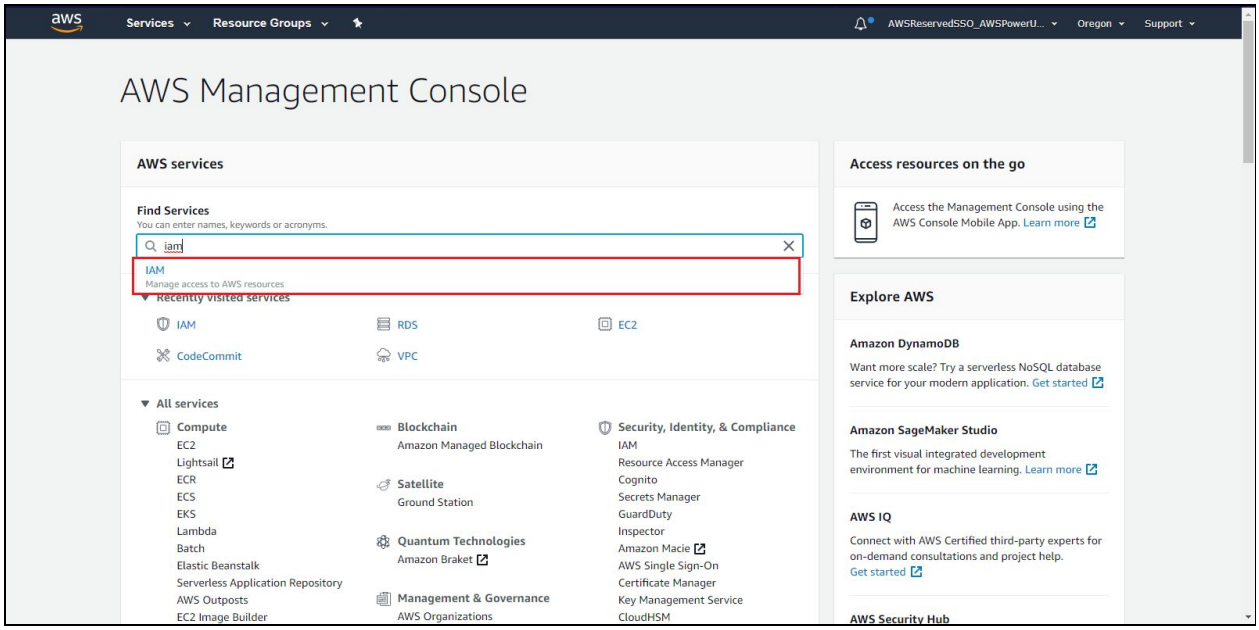
Here we have attached the QA environment account assume role policy to the code build role to provide the necessary permissions to deploy the code into QA environment.



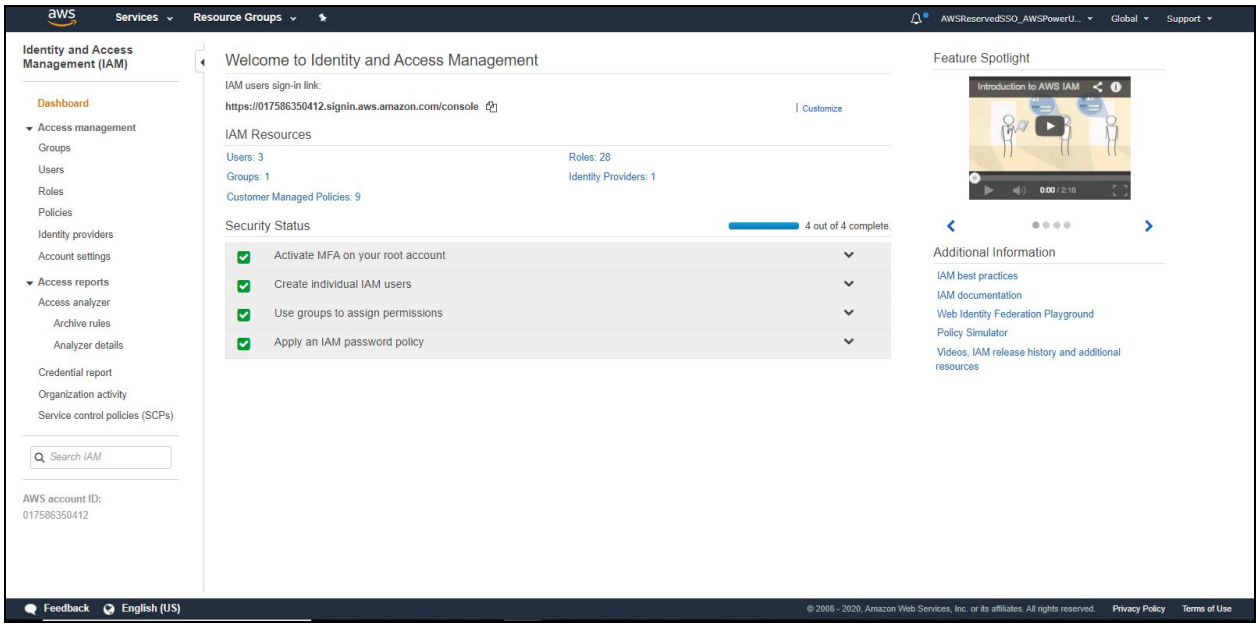
Login into the AWS QA environment account and we can verify the account number of QA environment with policy attached in dev environment.



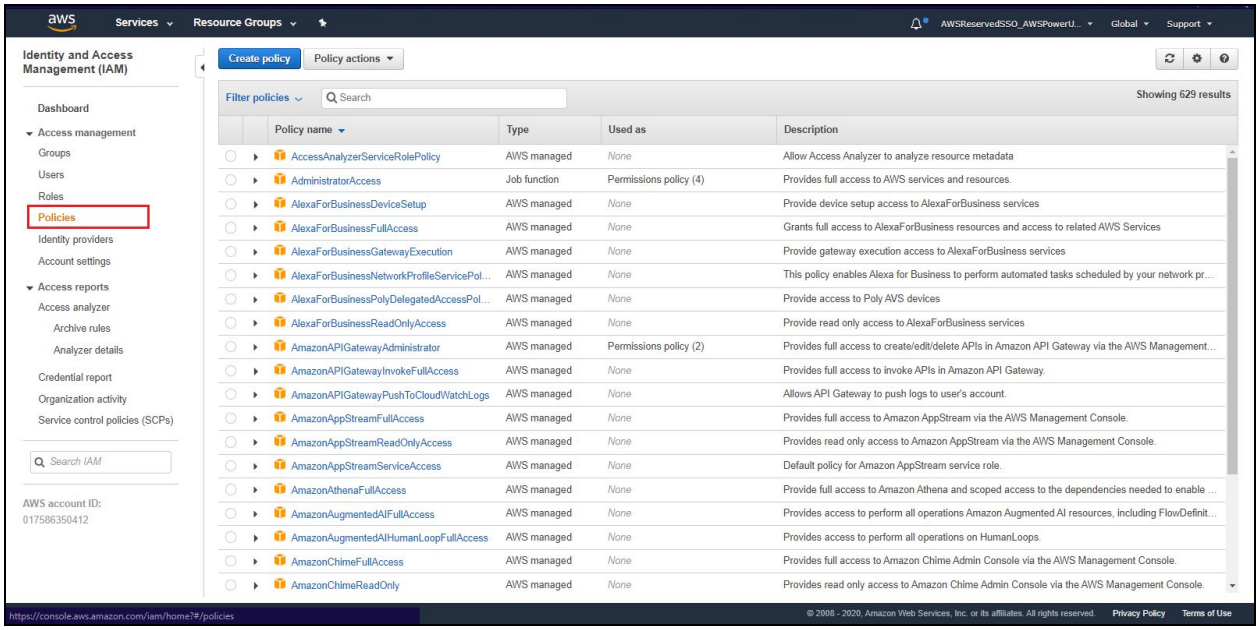
After login into the QA environment goto AWS IAM service



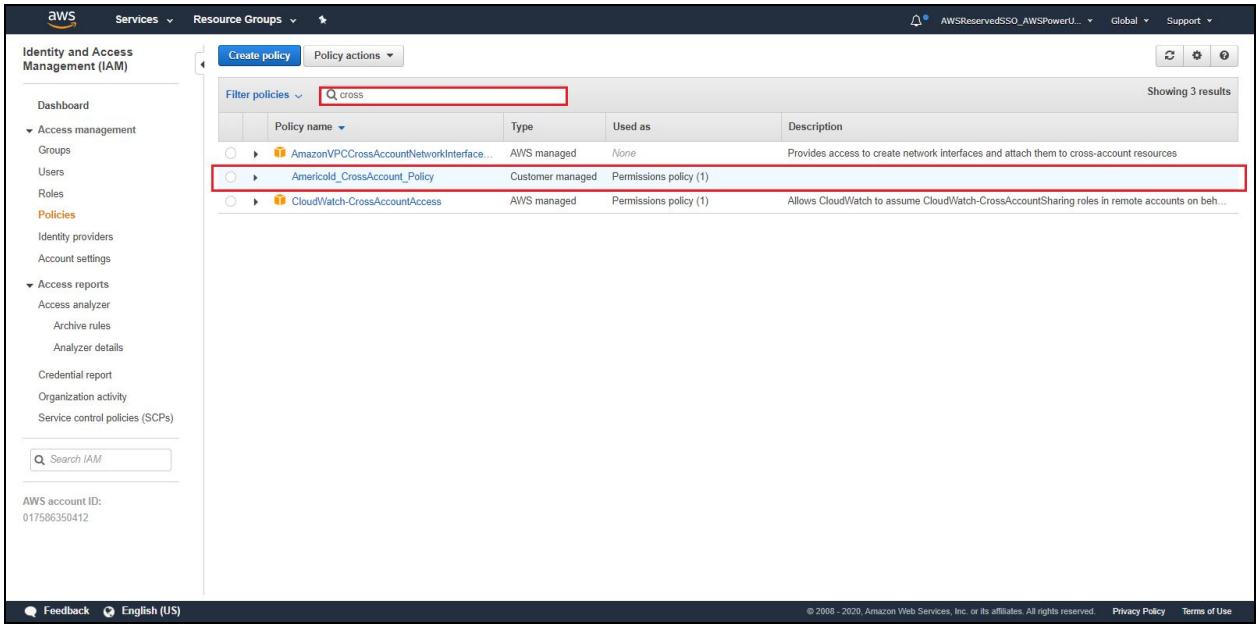
Dashboard of the AWS IAM service will be as shown below



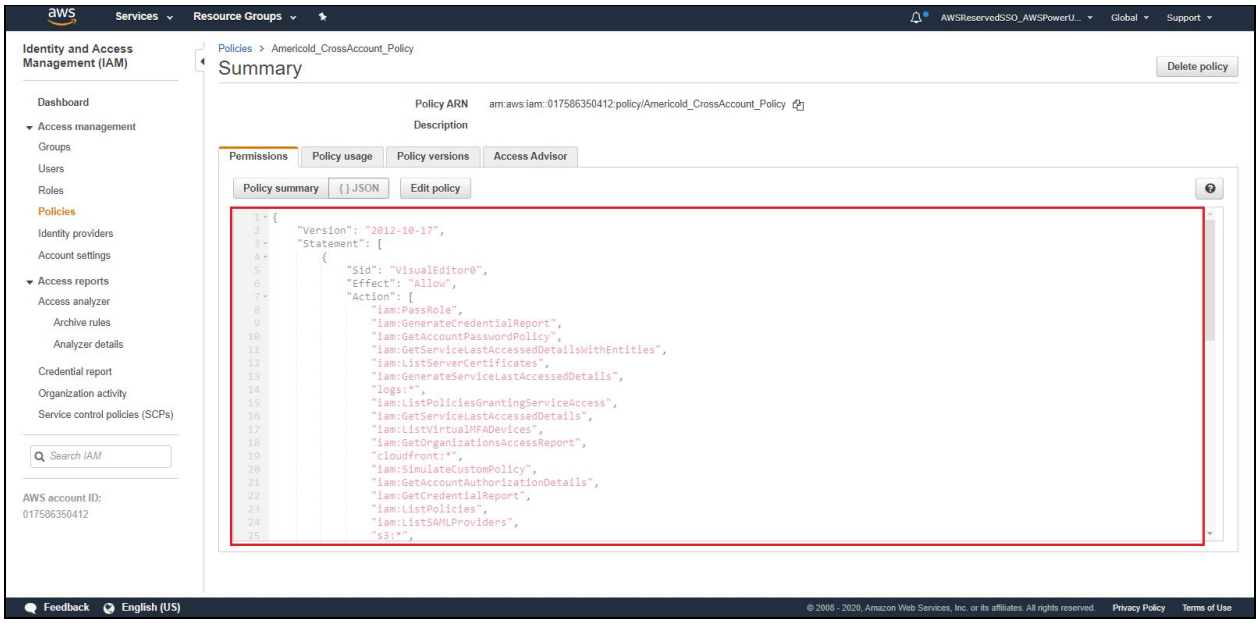
Goto policies section



Here we can see the cross account access policy which is created previously.

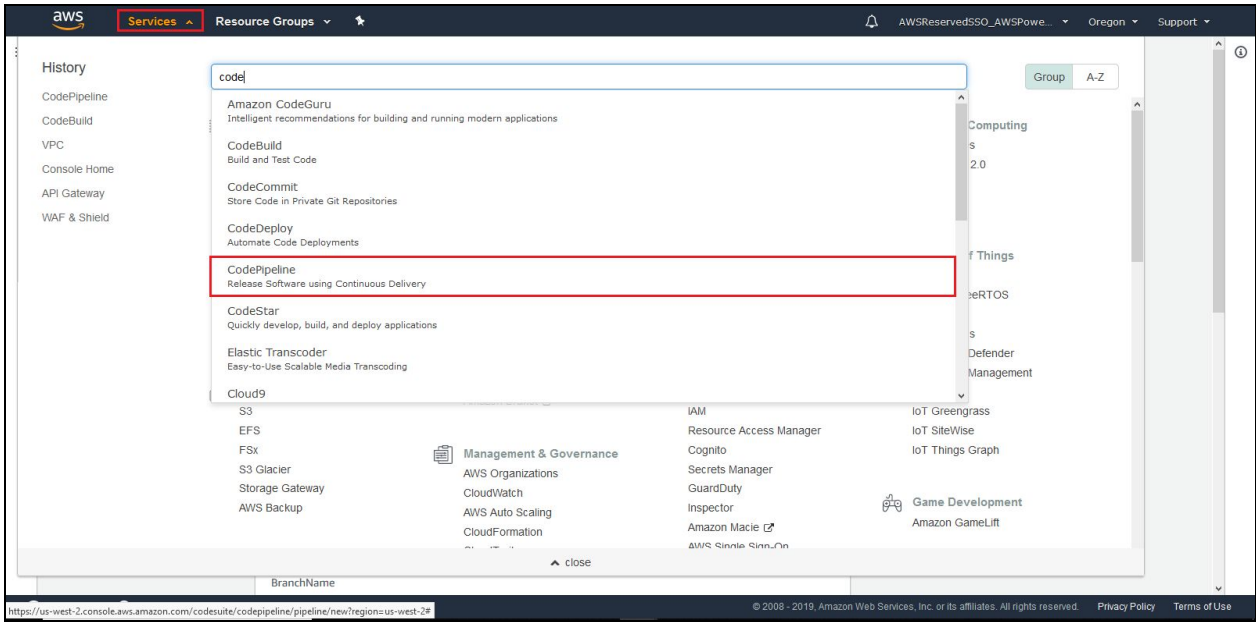


Here we can check the permissions given to the policy which are required for the deployment

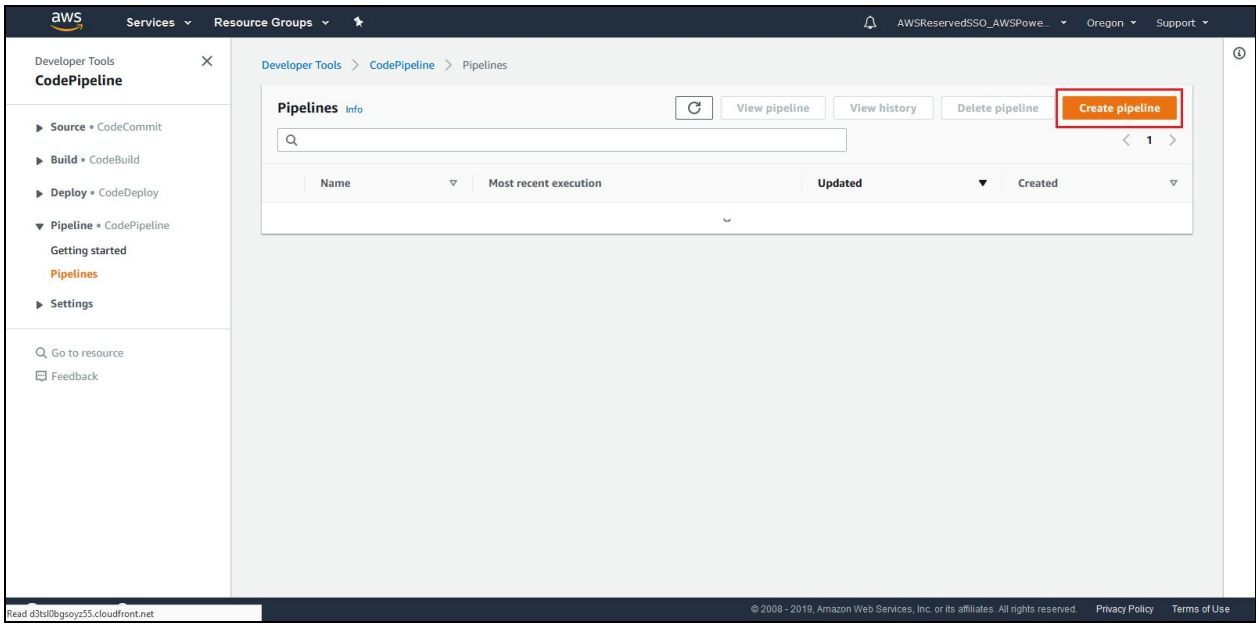


We are going to automate the process for integration and deployment using CodePipeline.

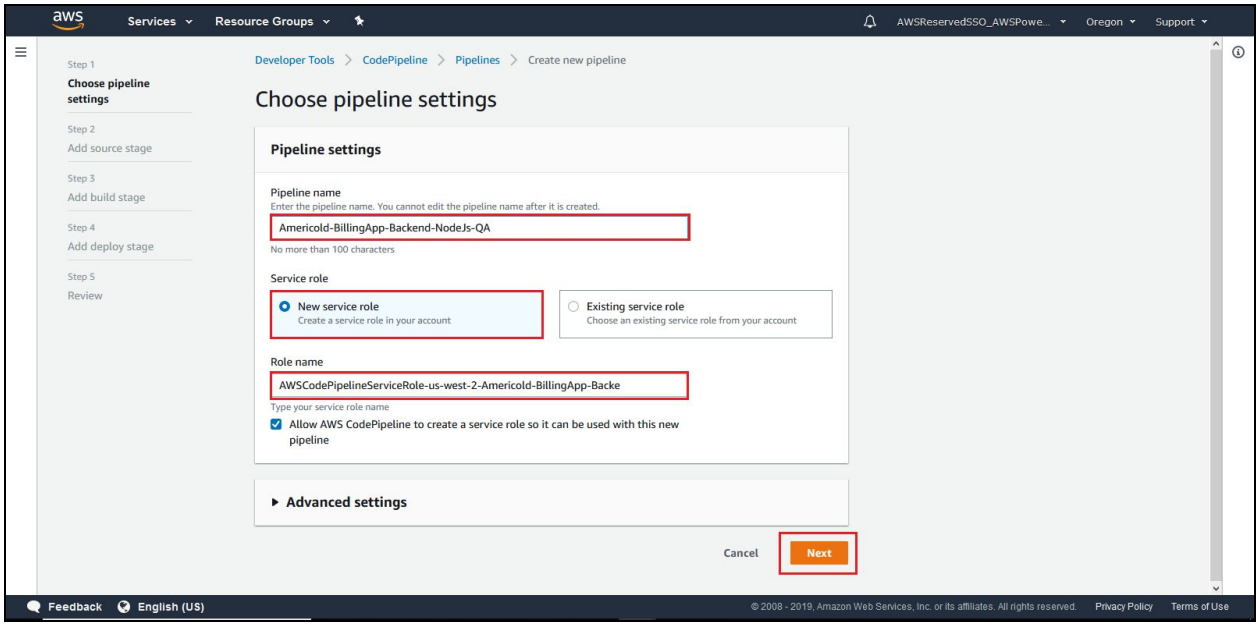
Goto **code pipeline service** in AWS console.



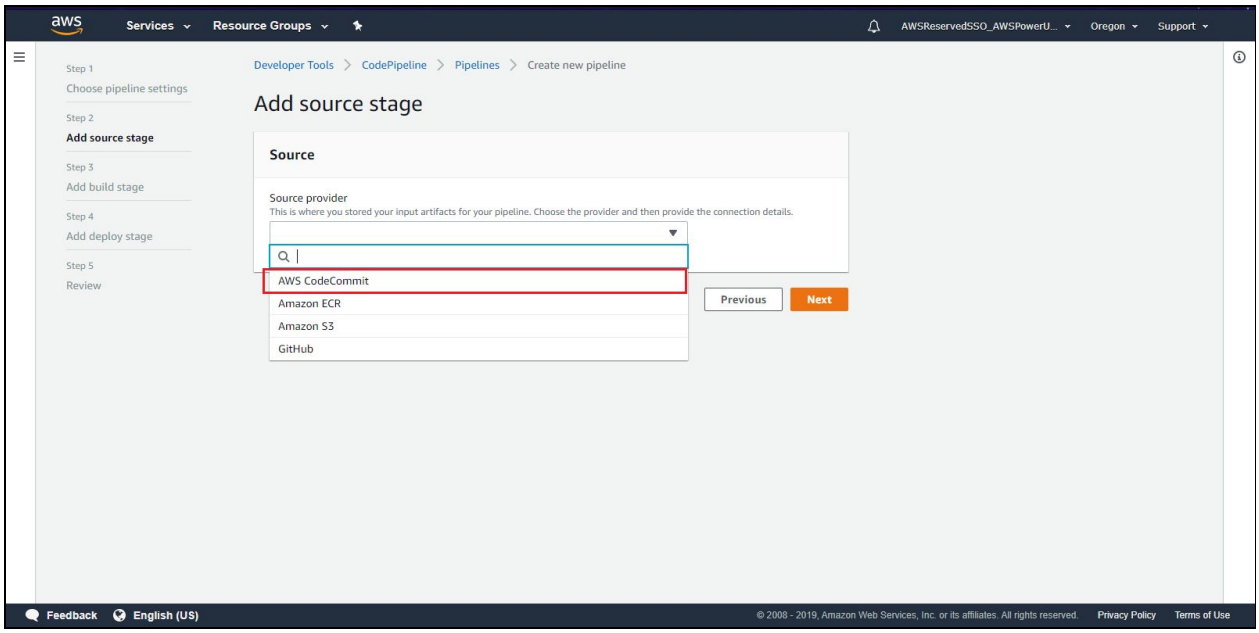
Click on **create pipeline** to create new pipeline for automated CI/CD.



Enter the name of the pipeline and choose the service role as new service role and click on **Next** as shown below. Here a new IAM service role will be created for code pipeline.

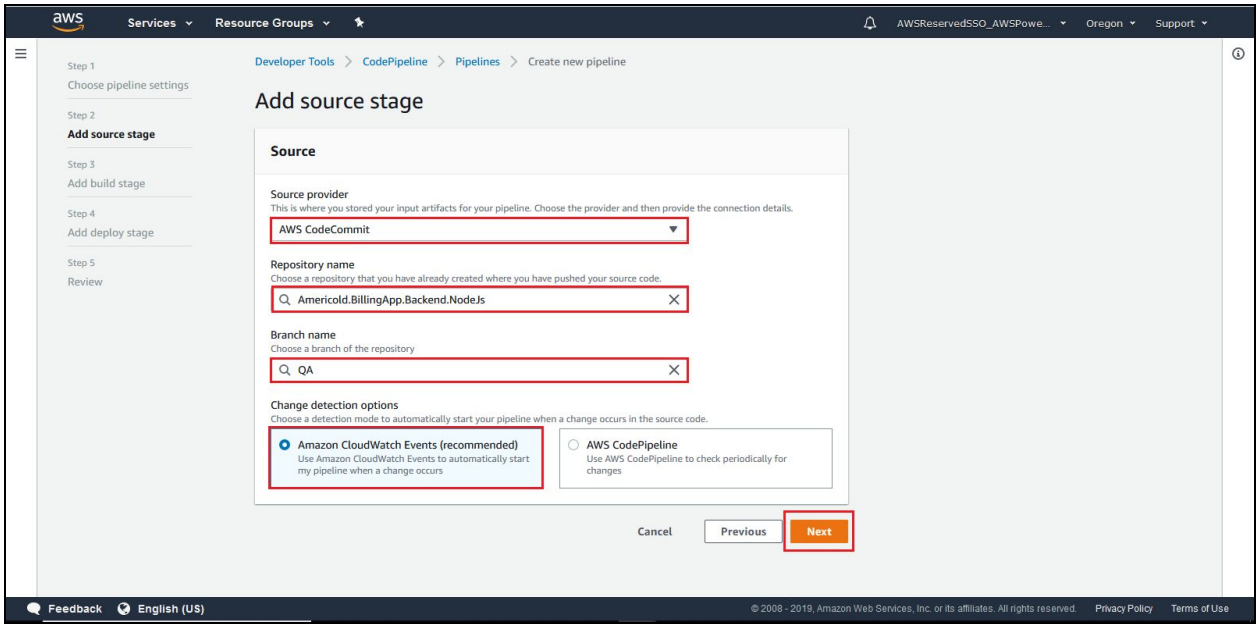


Under the Add source stage, select the source provider as **AWS code commit**.

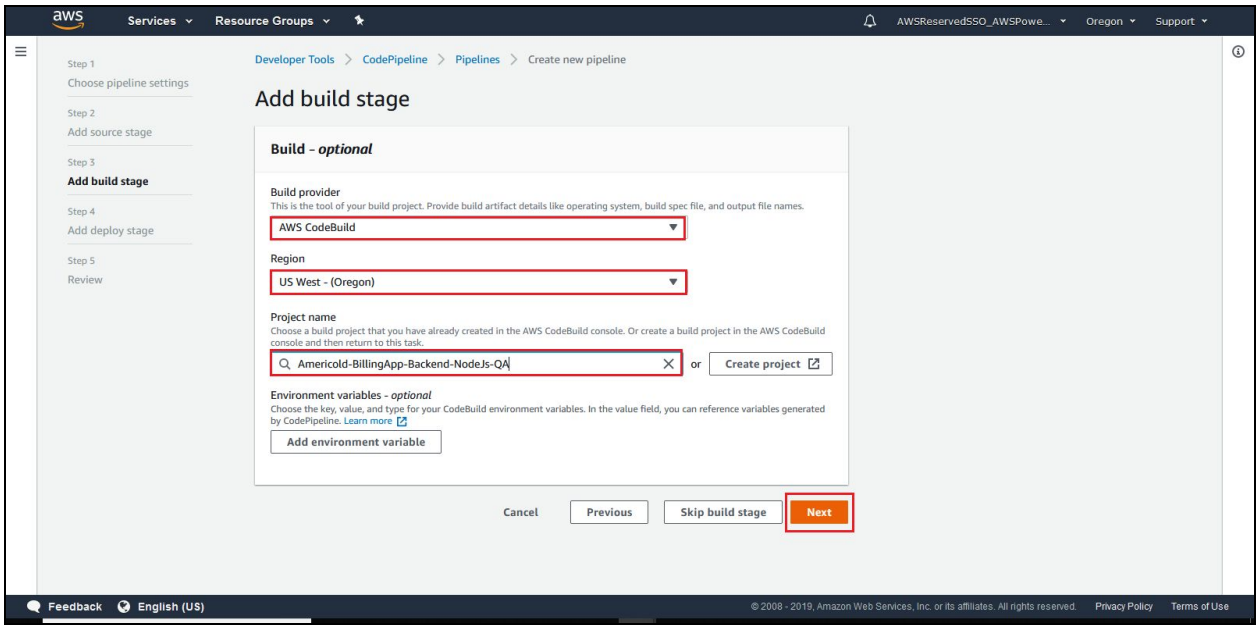


Select the name of the repository and branch name. choose the change detection options as

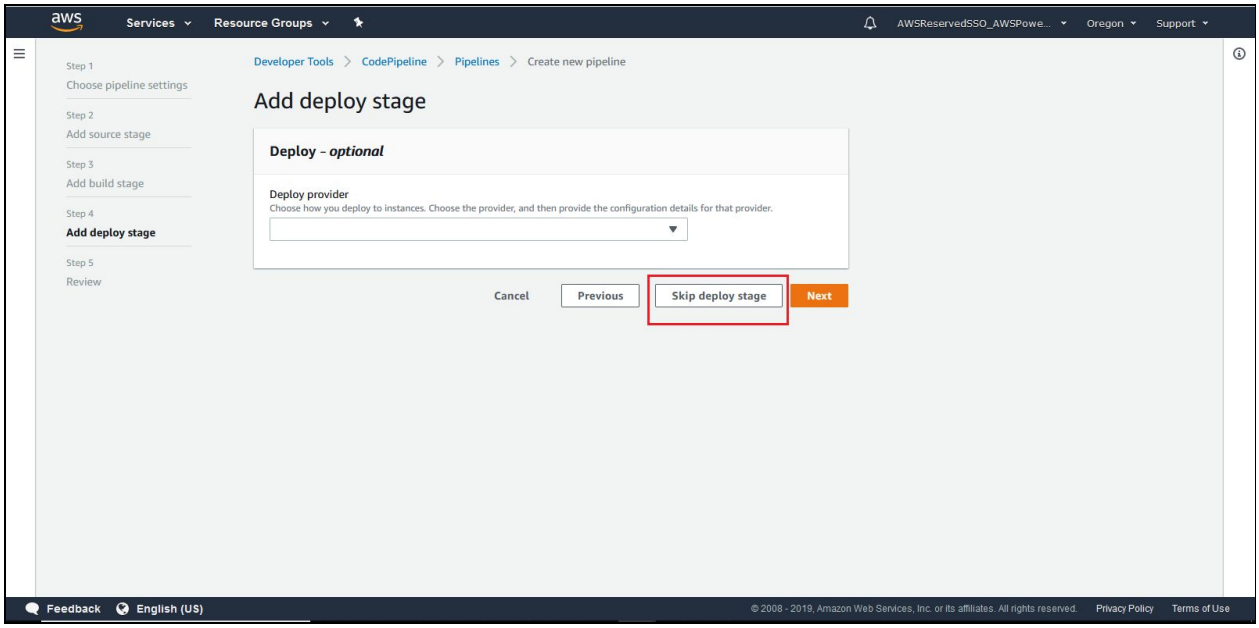
amazon cloud watch events to enable cloudwatch logs for the pipeline and click on **Next** to goto build configuration.



Under add build stage, select the build provider as **AWS code build**, region as **US West – (Oregon),** select the build project name that we created in the above steps and click on **Next** as shown below.

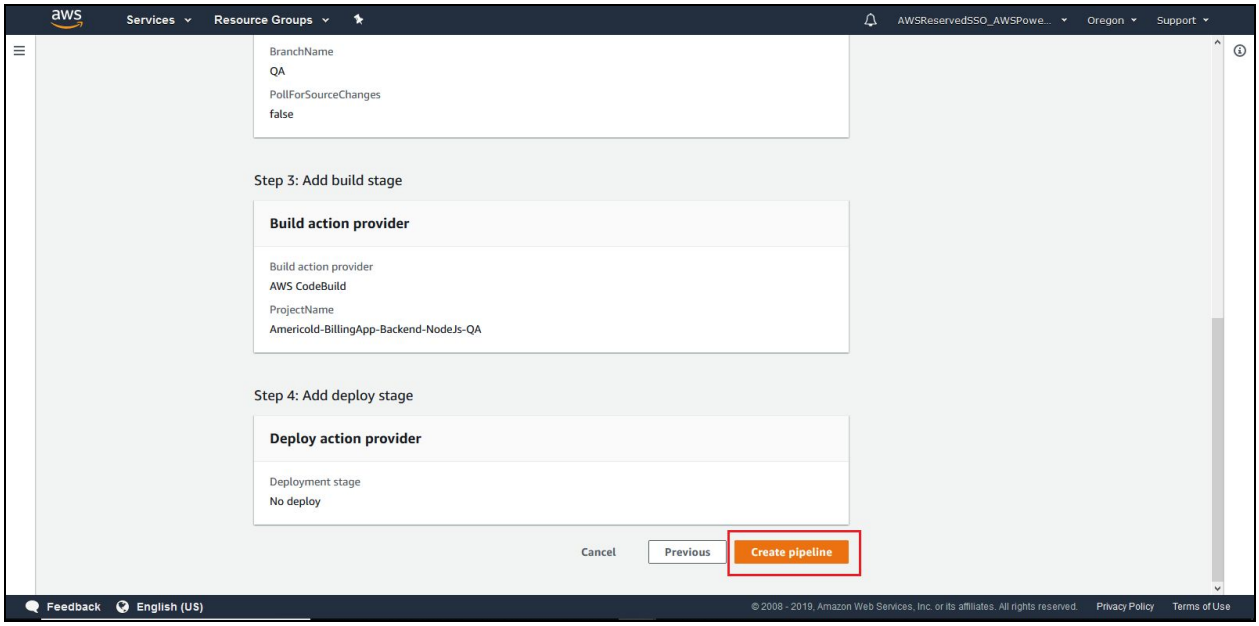


We are deploying through the commands. So, we can **skip** the deploy stage.

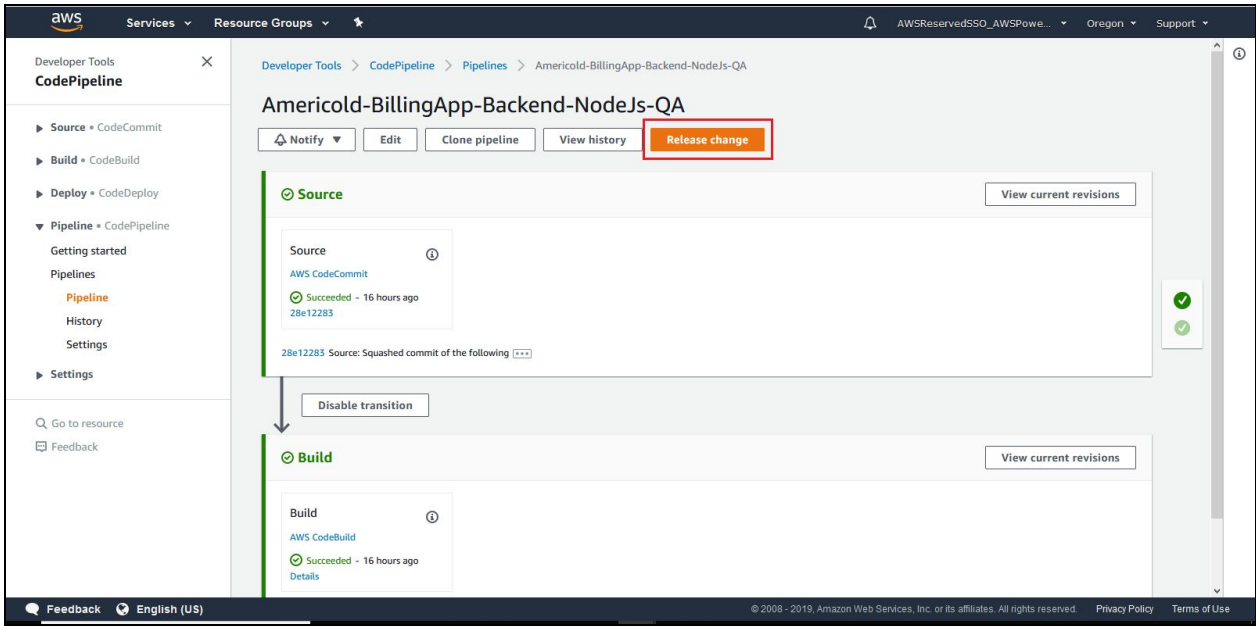


Click on **Skip.**

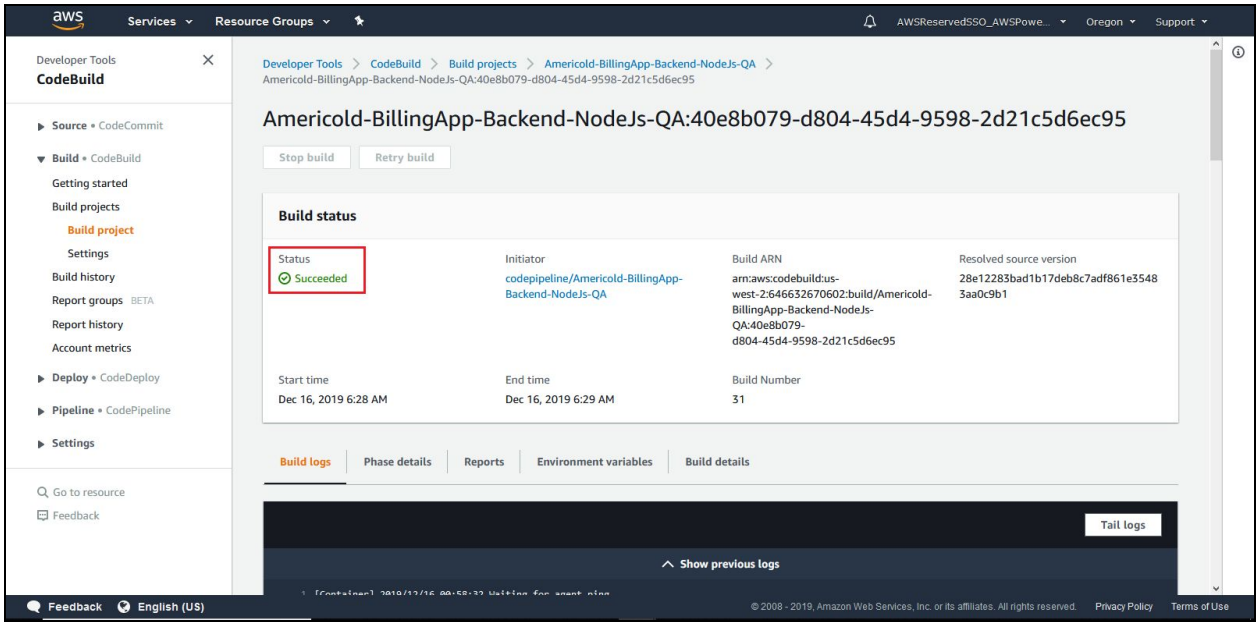
Review the pipeline configuration and click on **create pipeline**.



We can see the process is automated for CI/CD. We can check the stage logs by clicking on details. Pipeline will trigger automatically for the first time to release changes and later it will be triggered when the new changes are committed to repository.

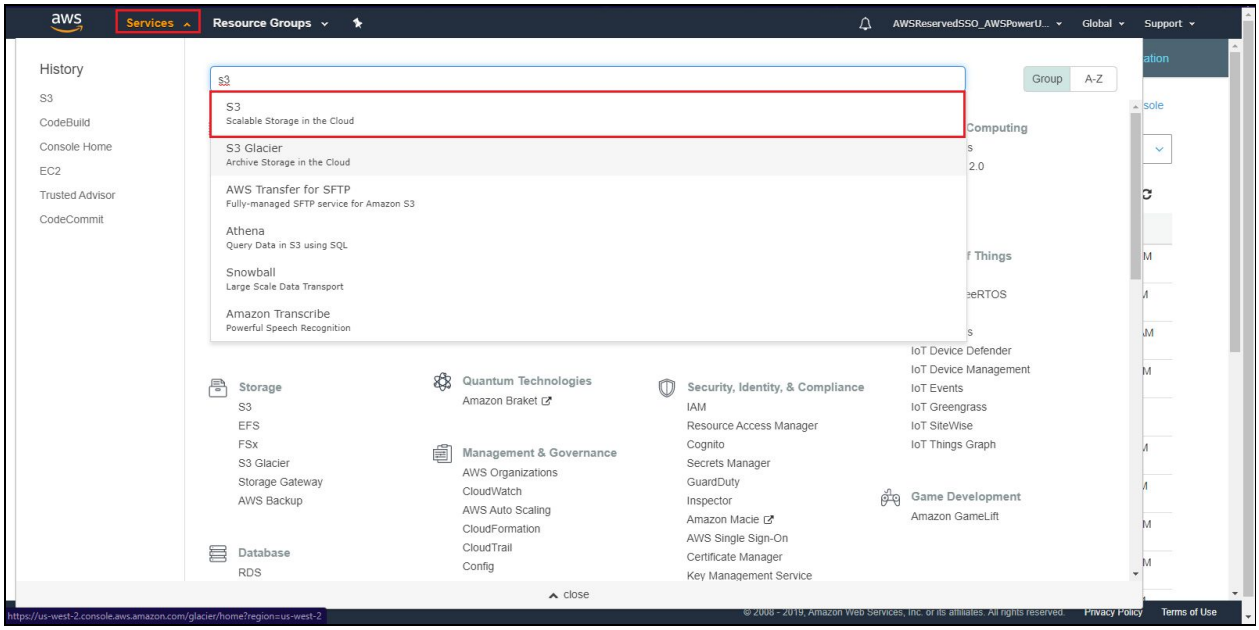


We can check the **build logs** here.

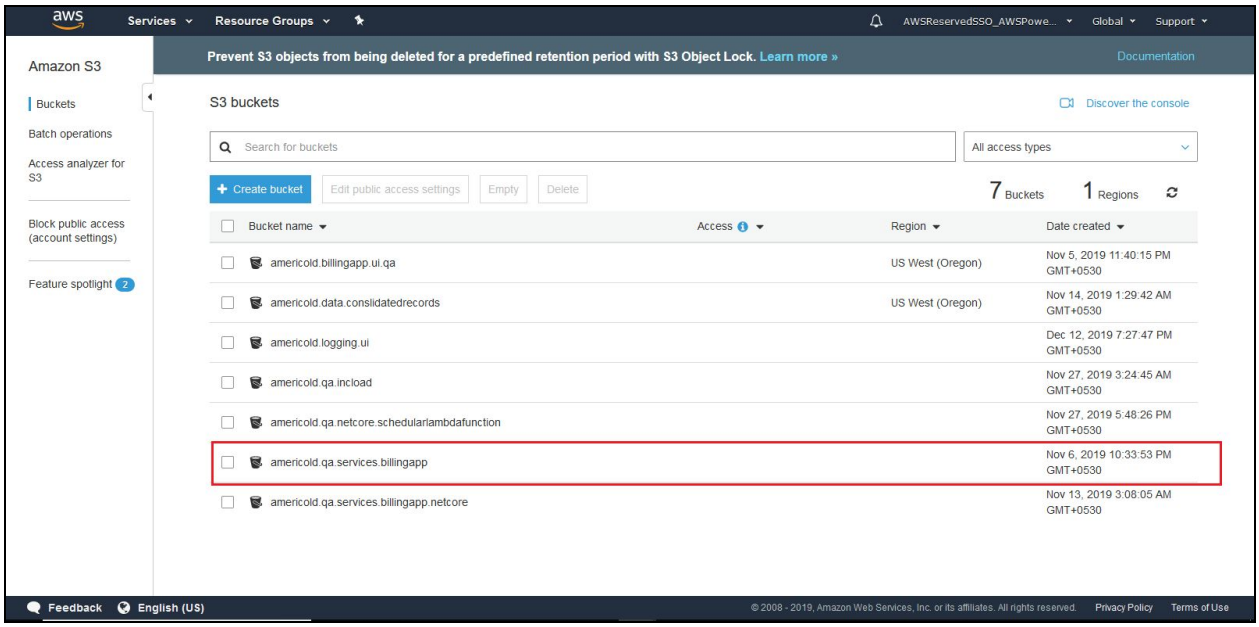


For backend applications, while running the **serverless deploy** command it will automatically create/updates the cloudformation stack using a serverless.yml template file.
Through the cloudformation template file, the artifacts will automatically deploy into **S3 bucket**, updates the lambda function and APIs

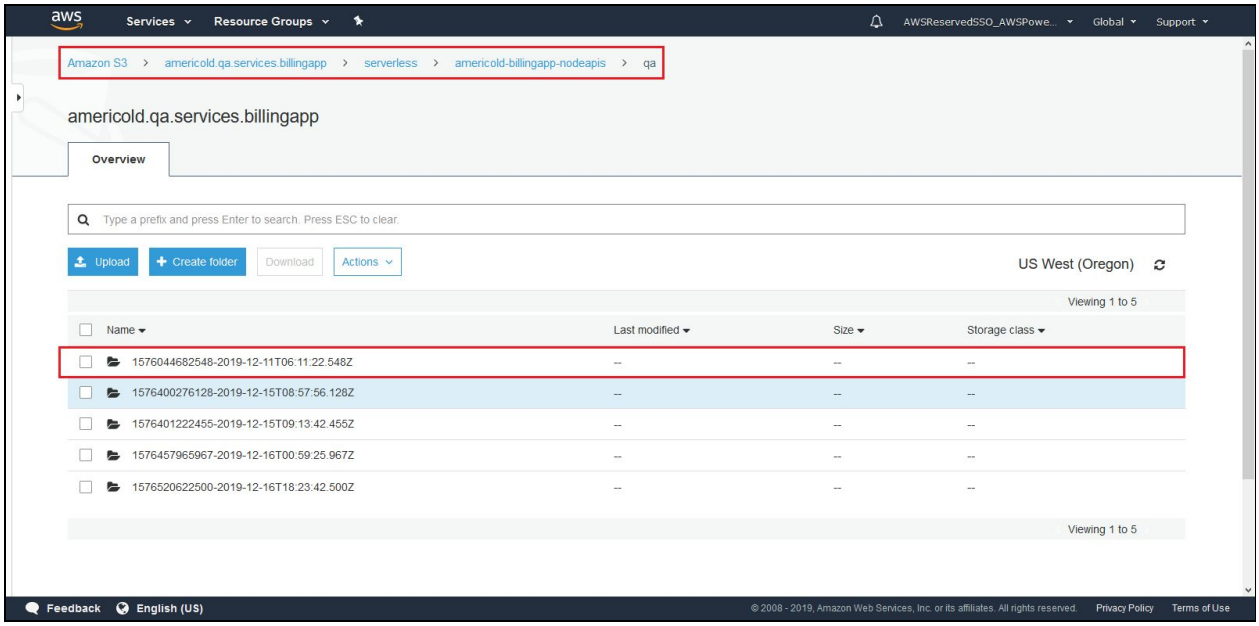
Go to S3 service to check whether the artifacts in the S3 bucket are updated or not.



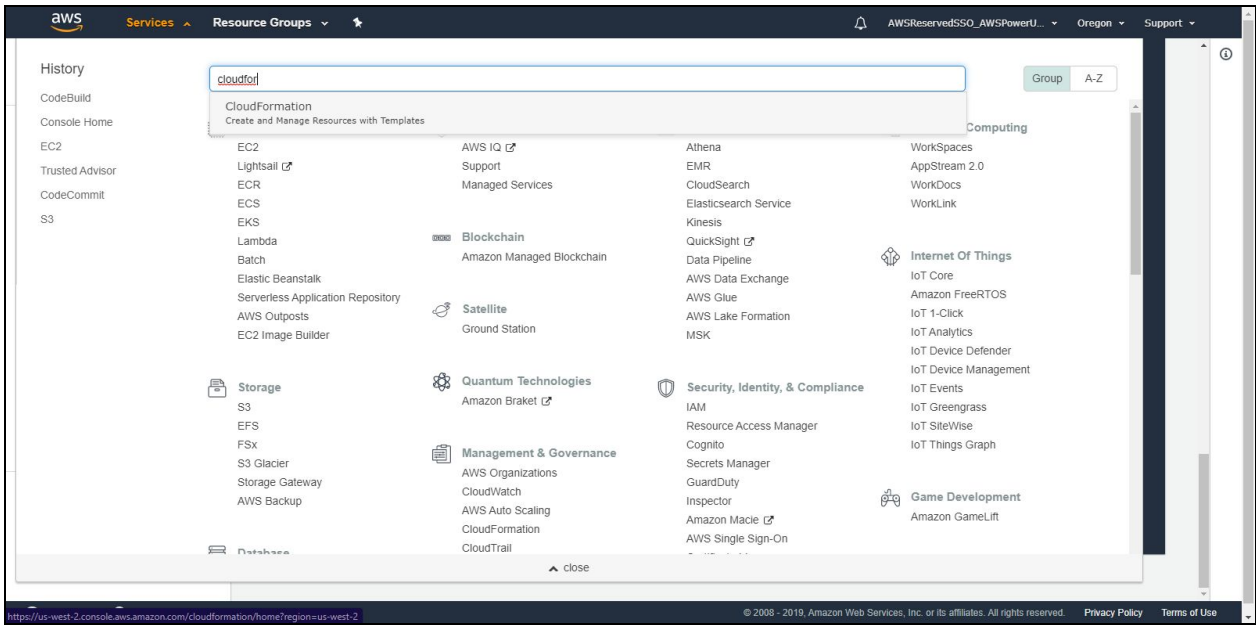
Following is the **S3 bucket** where we are storing our artifacts for backend Node.js application.



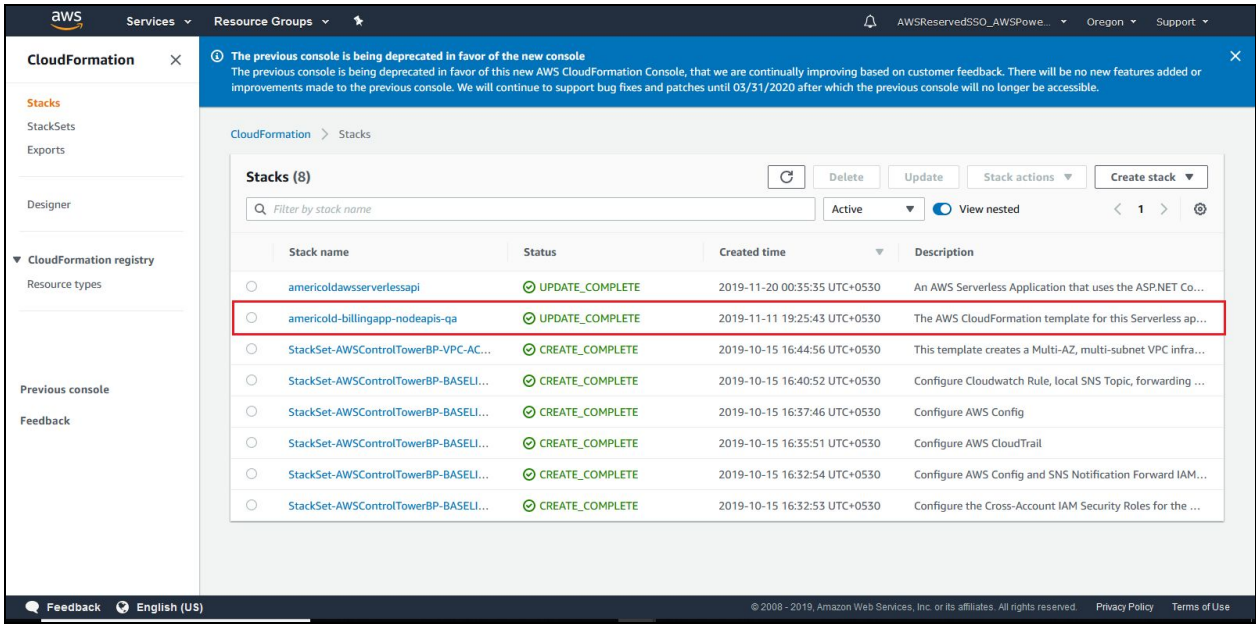
Here we can check our **latest artifact** in the S3 bucket.

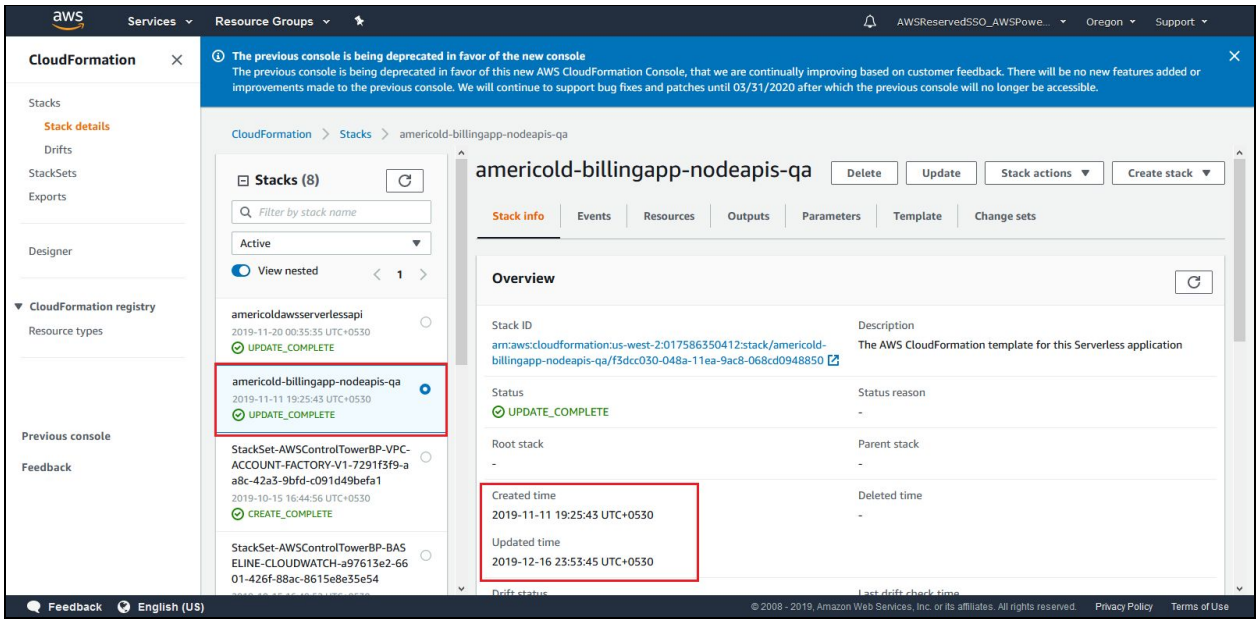


Go to **cloud formation** to check whether the events got triggered or not.

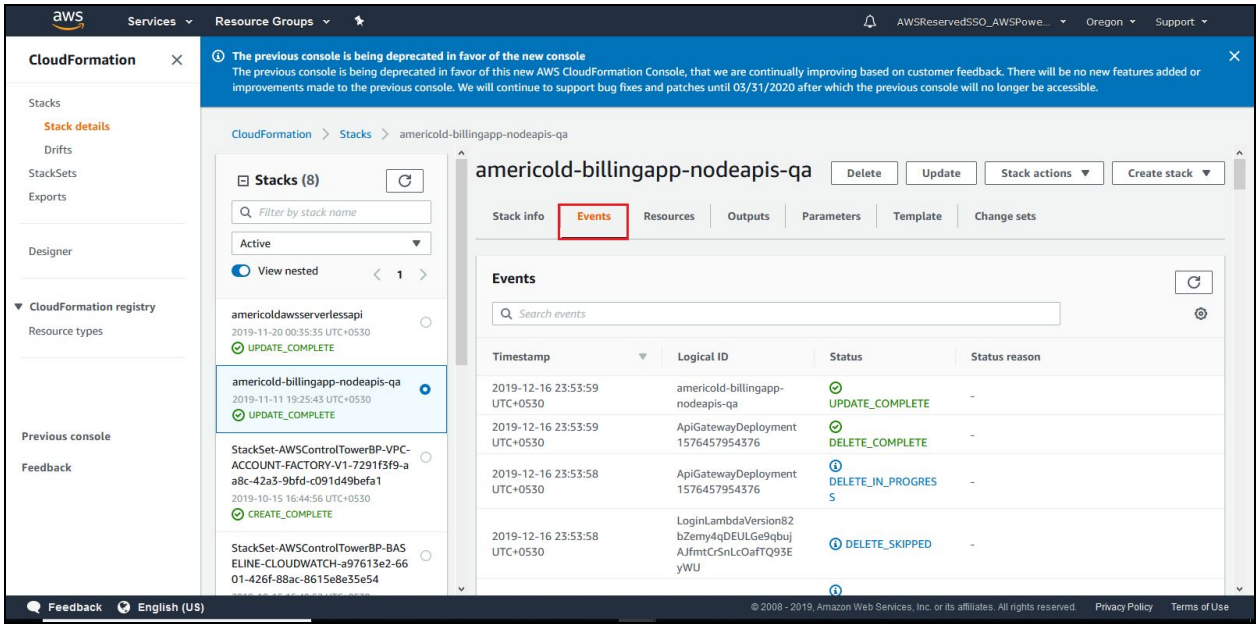


Here we can check our **cloud formation stack** updated time.

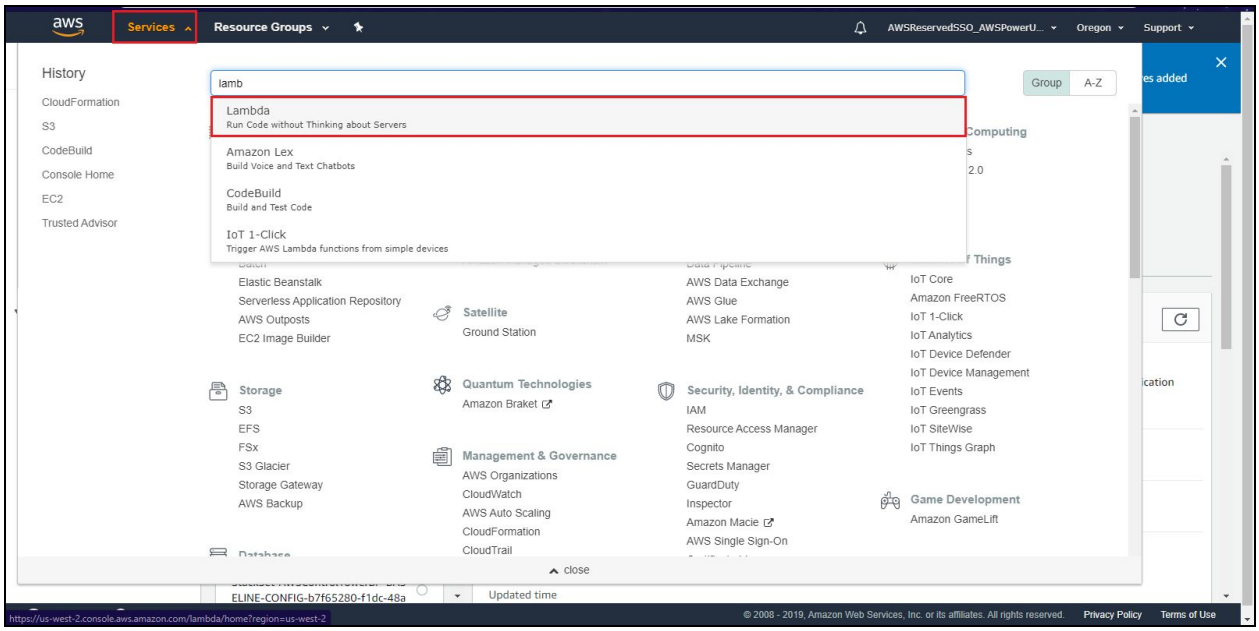




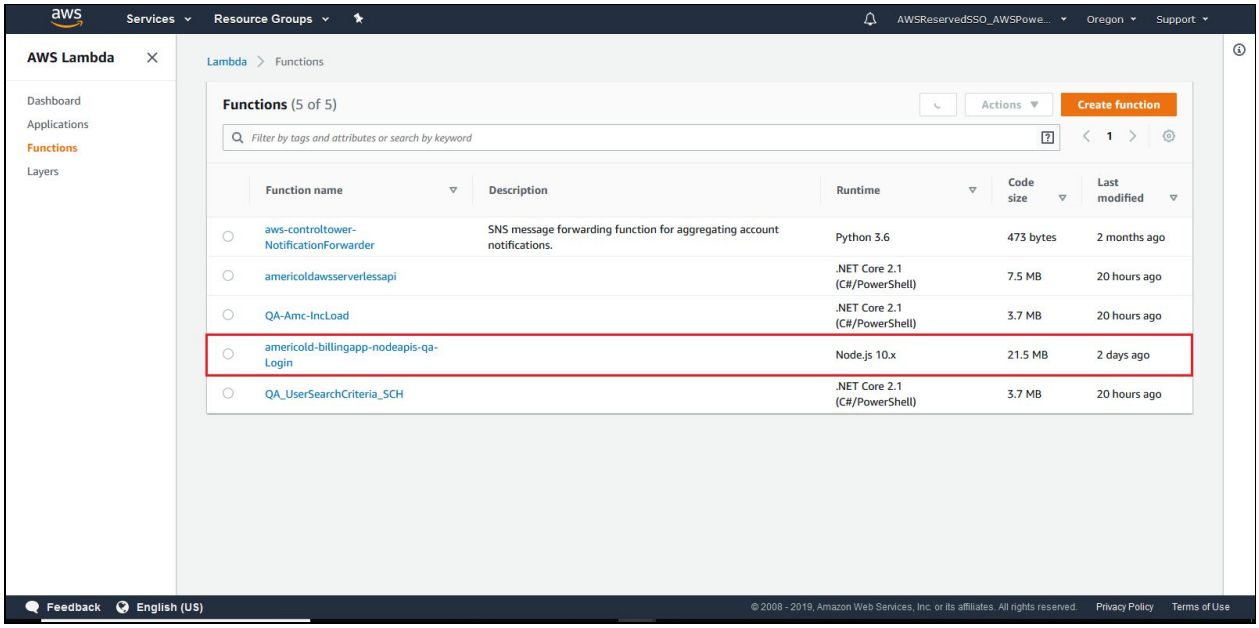
We can check our **events triggered** by our cloud formation stack.

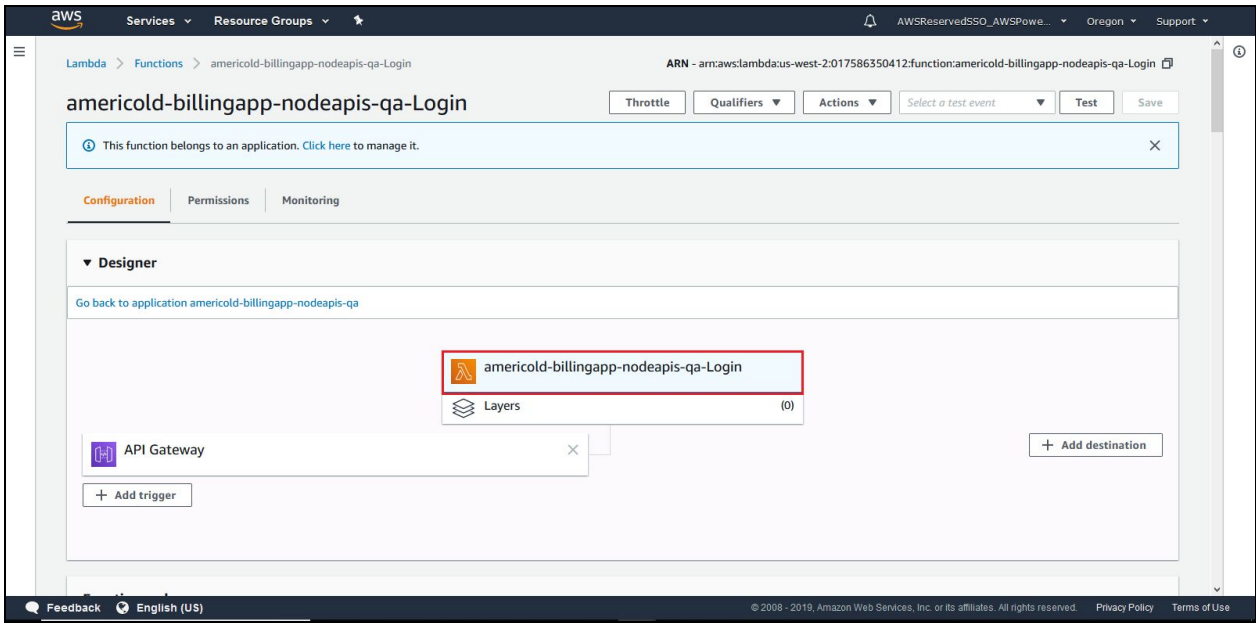


Go to the **Lambda service.**

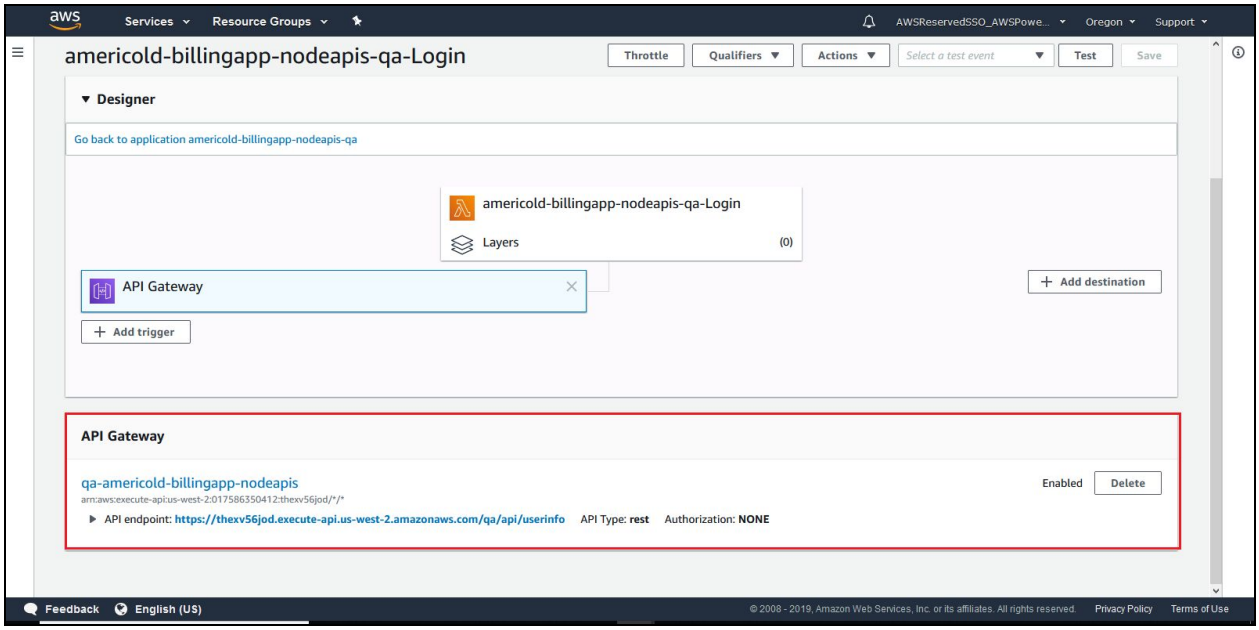


Following is the **lambda function** that got updated.

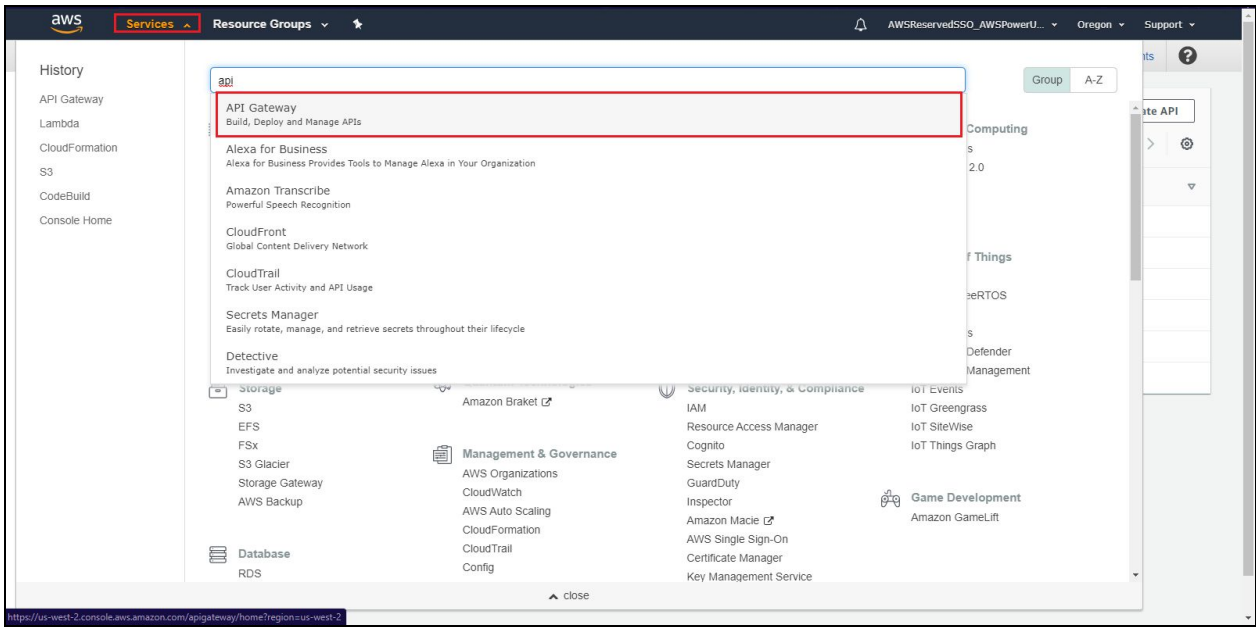




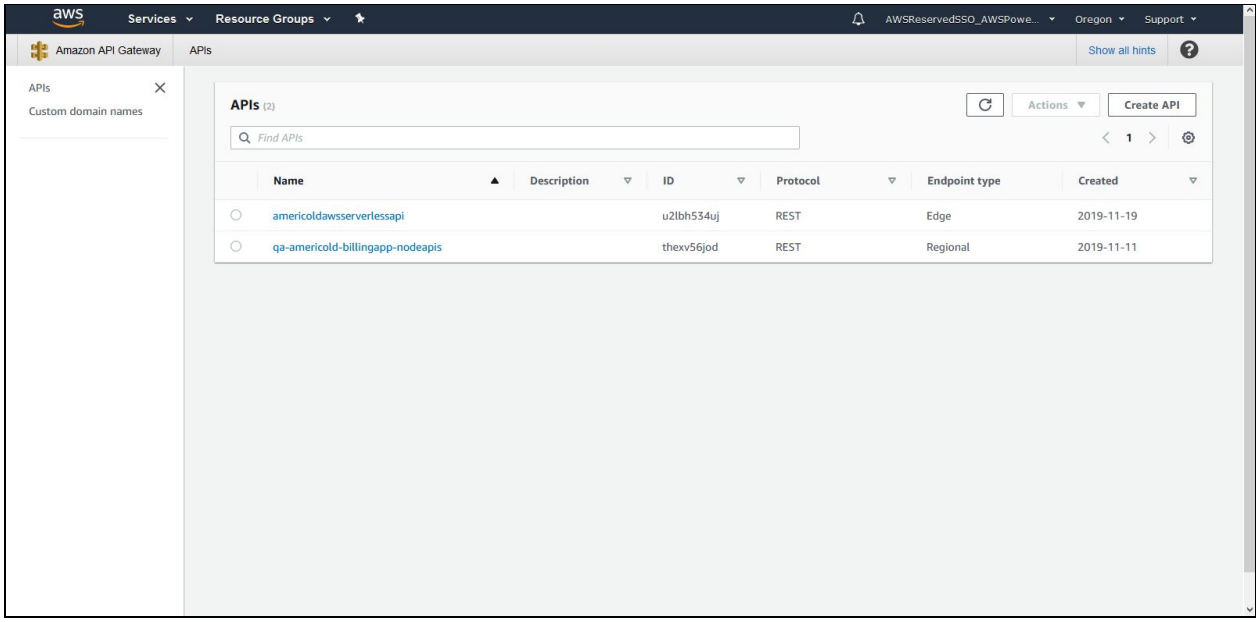
Here is the **endpoint** of API gateway.



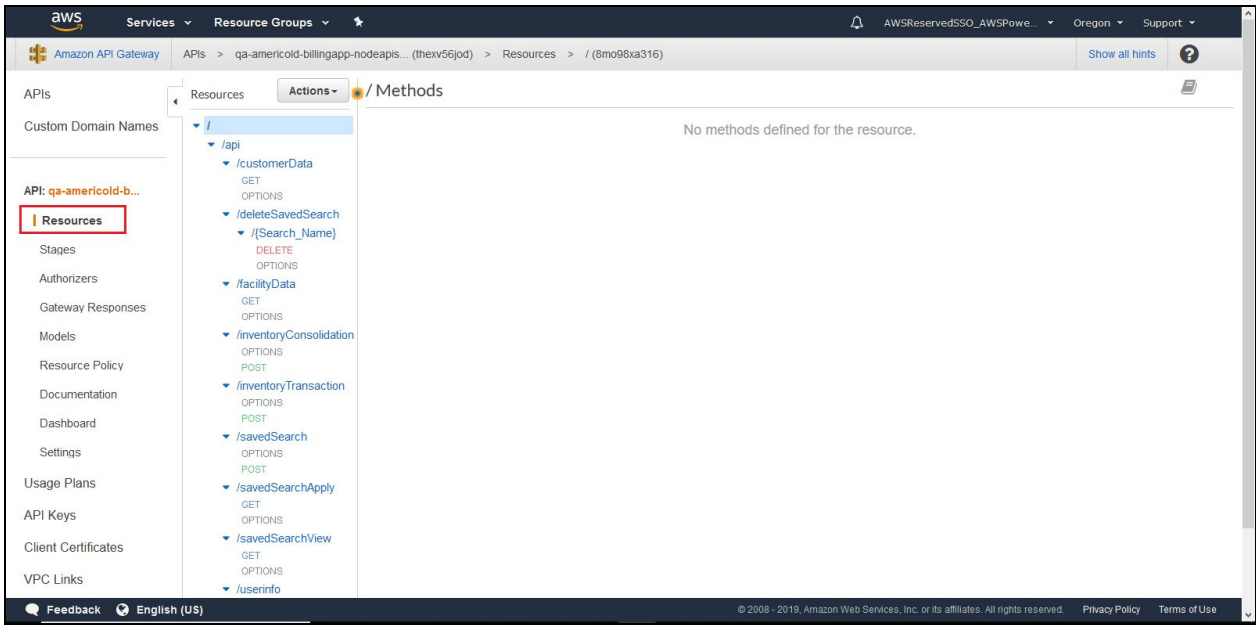
Goto AWS console and select **API gateway** service.



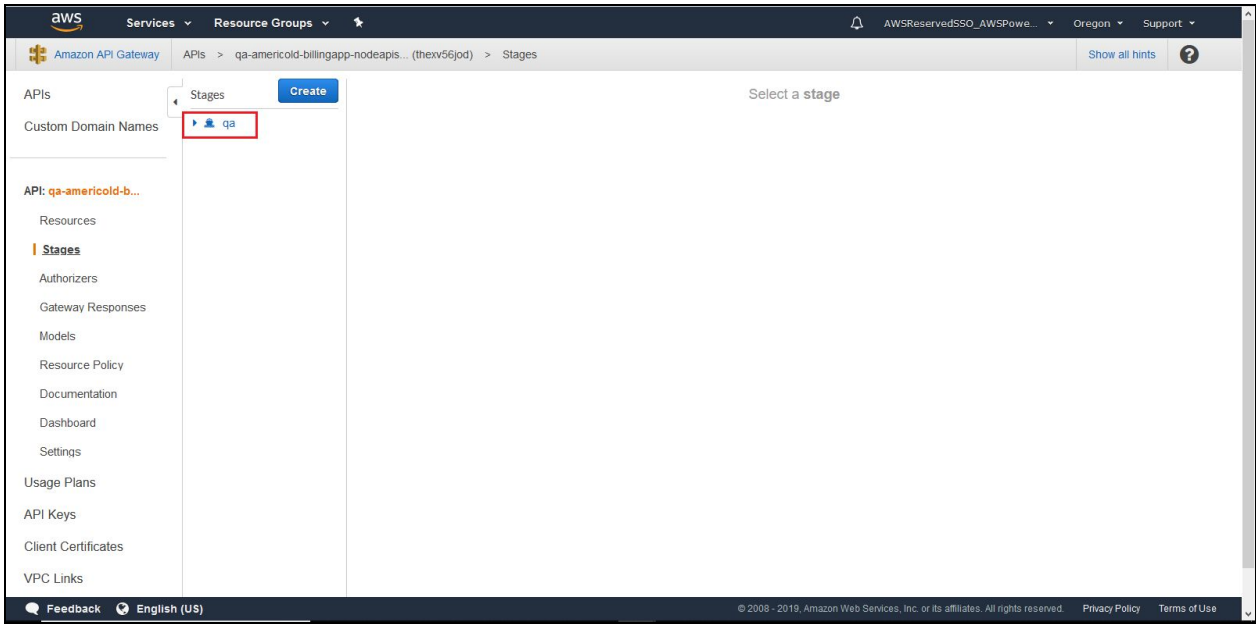
select the **API gateway** which is created/updated.



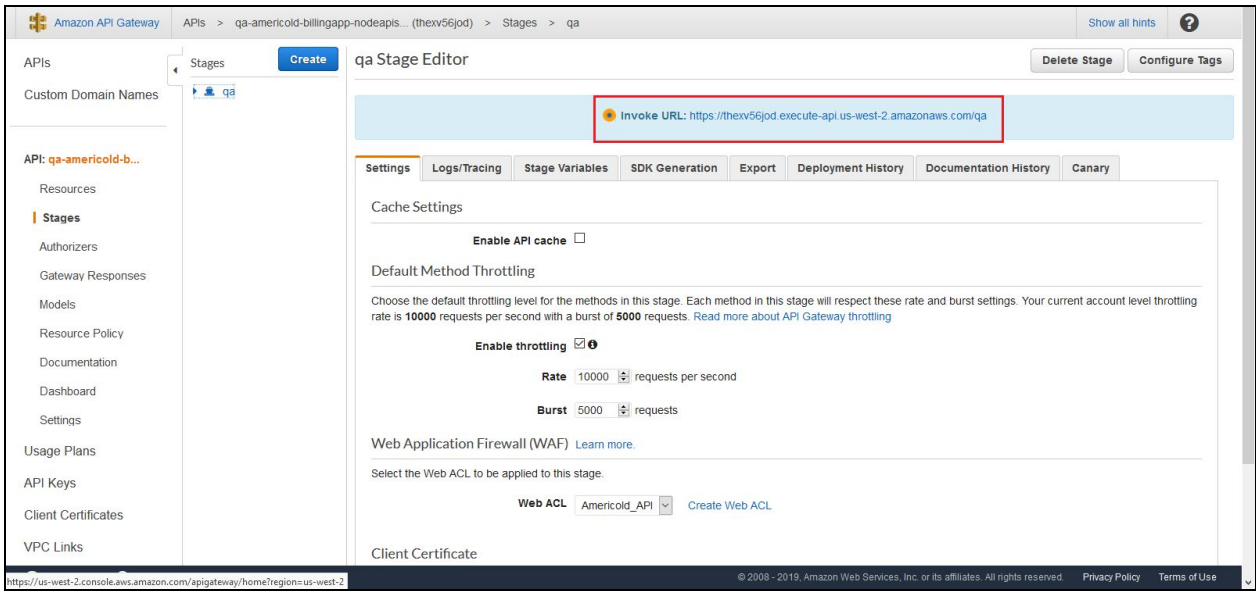
Here is the dashboard of the **API gateway**.



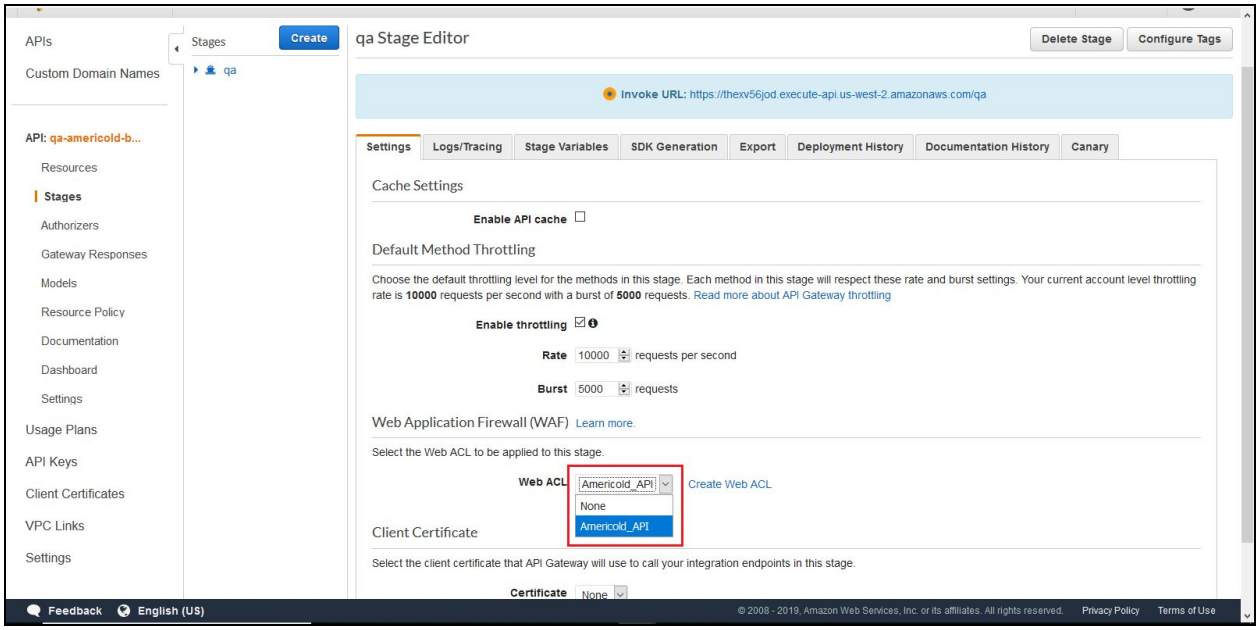
For API stage URL click on the stages and click on qa



Here we can seed the Node.js API URL

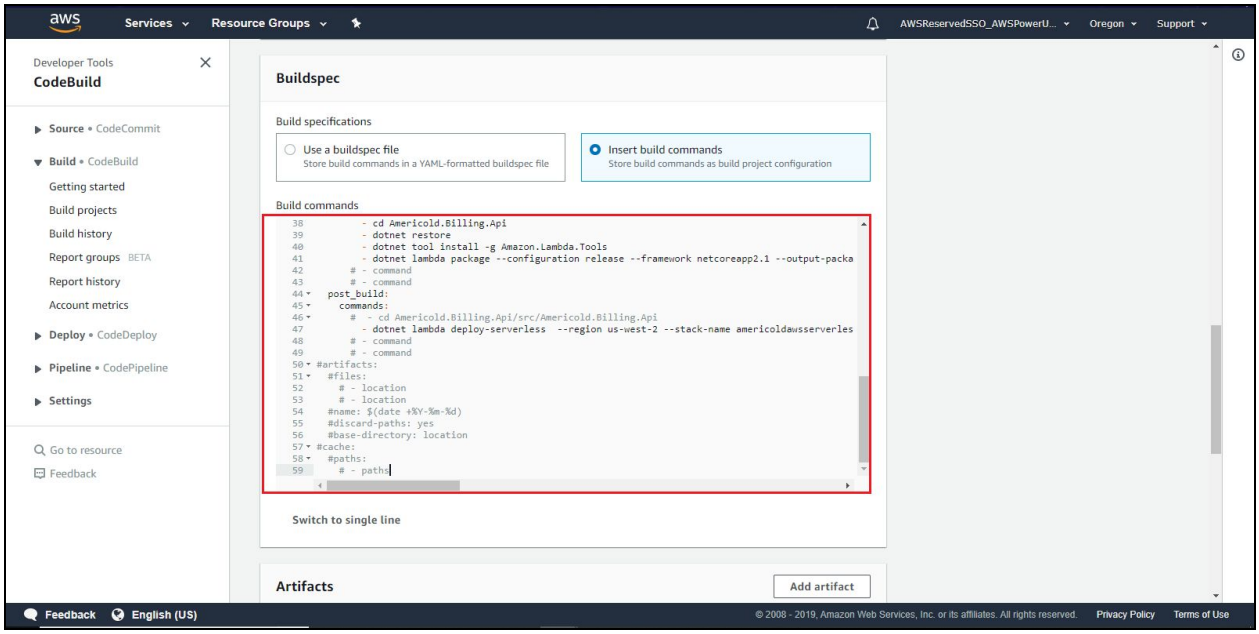


For backend API's we are configuring AWS WAF to provide security.



Similarly, we are creating the CI/CD pipeline for the ASP dotnet core application. Following is the buildspec.yml file that we are using for integration and deployment for the ASP dotnrtet core application.

Following is the **buildspec.yml file** for ASP dotnet core application.



Below is the step by step explanation for the above script:

Install stage:

- **runtime-verions:**

  dotnet : 2.2: It defines the runtime environment and version for the application

Pre_build stage:

- **apt install jq –y:** Install node modules using the following command
- **unset AWS_SESSION_TOKEN:** Clear the AWS session token
- **export AWS_REGION=us-west-2:** Set the AWS region
- **temp_role=$(aws sts assume-role --role-arn "arn:aws:iam::017586350412:role/Americold_CrossAccount_Role" --role-session-name "QA"):** Create temporary AWS IAM role and store in the variable as shown above.
- **export AWS_ACCESS_KEY_ID=$(echo $temp_role | jq .Credentials.AccessKeyId | xargs)**

  export AWS_SECRET_ACCESS_KEY=$(echo $temp_role | jq .Credentials.SecretAccessKey | xargs)

  export AWS_SESSION_TOKEN=$(echo $temp_role | jq .Credentials.SessionToken | xargs): Set the AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, AWS_SESSION_TOKEN variables with their respective values using jq tool as shown above

Build Stage:

- **cd Inventory_consolidation:** Go to the folder called Inventory_consolidation
- **dotnet restore:** Following command uses NuGet to **restore** dependencies
- **dotnet tool install -g Amazon.Lambda.Tools:** Following commands used for Dotnet CLI

to deploy AWS Lambda functions

- **dotnet lambda package --configuration release --framework netcoreapp2.1 --output-package bin/release/netcoreapp2.1/Americold.Billing.Api.zip:** To create a package to deploy we are using the following command

Post-Build stage:

- **dotnet lambda deploy-serverless --region us-west-2 --stack-name americoldawsserverlessapi --s3-bucket americold.qa.services.billingapp.netcore --s3-prefix Americold.Billing.Api/ --template-parameters "EnvironmentName=QA;LambdaSecurityGroups=\"sg-0a0182ed805d10d2f\";LambdaSubnetID=\"subnet-0b68418e0d291027c,subnet-05105151bc2487a14,subnet-085bab28792c2831f\""**

  To deploy ASP Net Core application using serverless. template file we will run the following command. Here we have given some extra configuration for S3 bucket, VPC, and Cloudformation stack