

A blue and black logo

Description automatically generated with low confidence

AMI-GITLAB MODERNIZATION

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Prep-Work Documentation**

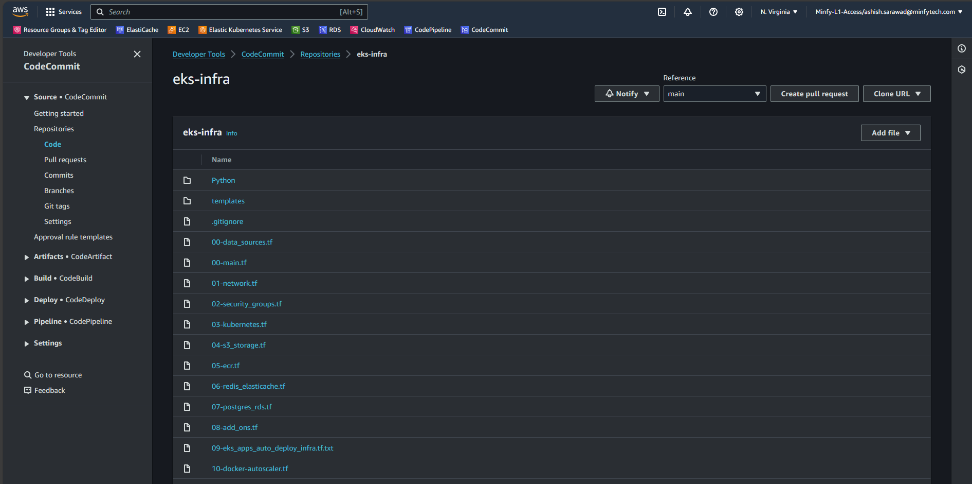
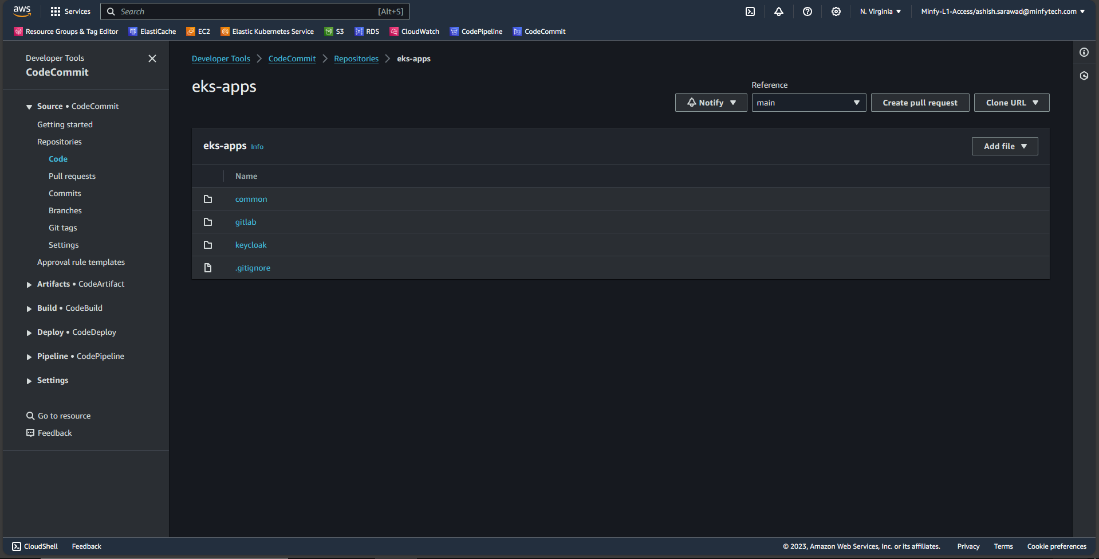
**INTRODUCTION**

The journey to a successful GitLab deployment in AWS begins with comprehensive preparation work, often referred to as "prep-work." This phase is characterized by a series of foundational tasks and essential groundwork that are pivotal to the overall success of the deployment. Let's delve into the key aspects of prep-work in more detail.

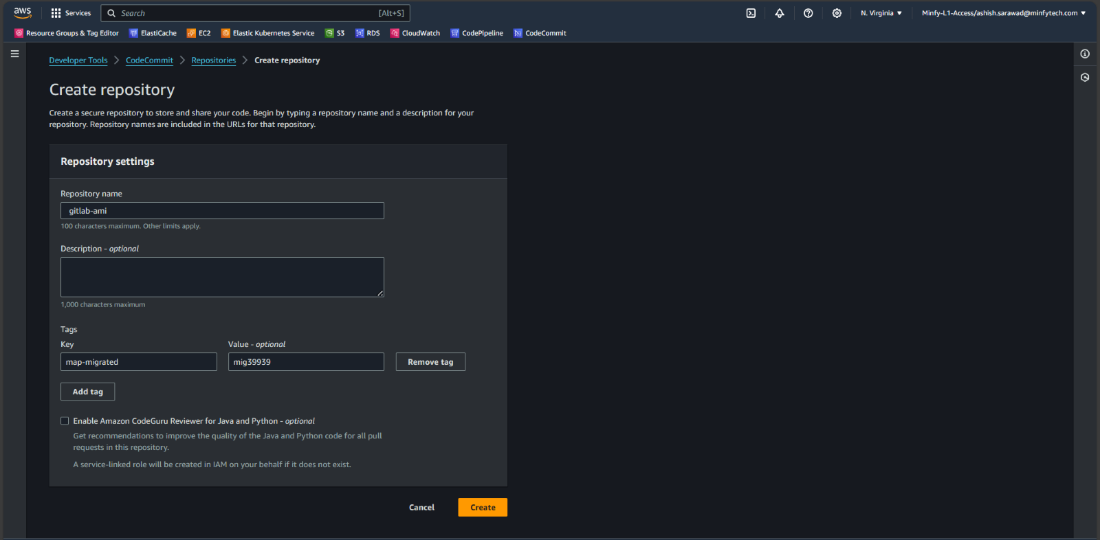
In starting of this phase, it is imperative to verify the sharing status of the **Transit Gateway** with this account. Terraform will automatically fetch the Transit Gateway ID using data block, requiring precise alignment with the configuration. In the network account, the VPC attachment request generated by Terraform should seamlessly undergo an automatic acceptance process. The coordination of these steps is crucial for the smooth integration of the designated VPC with the Transit Gateway. Post-deployment, checks must be performed to confirm successful attachment and establish communication between connected VPCs. Timely coordination, accurate configuration, and proactive monitoring are key elements for a seamless attachment process outlined in this document

**1. Code Commit Workflow Establishment:**

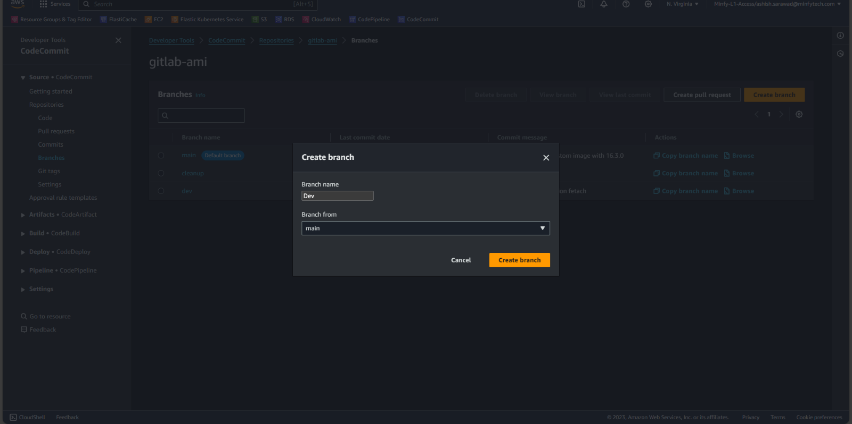
The foundational step involves creating a robust AWS CodeCommit workflow, a critical component of version control and code review. This workflow contains the systematic progression of code submissions: code changes are initially committed to the development (dev) branch, followed by a peer review. Upon successful review and approval, code changes are seamlessly merged into the main branch, serving as a trigger for subsequent pipeline stages. This well-structured workflow ensures that your codebase remains organized, secure, and keenly manage  
In automation setup, we're establishing two CodeCommit repositories for distinct roles: one for infrastructure provisioning and the other for application deployment. This clear separation ensures effective version control and collaboration. Notably, this approach allows seamless addition of future applications without disrupting the current environment, maintaining a scalable and adaptable development ecosystem.

  
 *Image: CodeCommit Repository for infrastructure management*  
   
   
 *Image: CodeCommit Repository for application deployment*

**Steps involved in establishing the code versioning workflow with CodeCommit**  
**I.** To initiate the creation of an AWS CodeCommit Repository, access the “**AWS CodeCommit**” Console "**Developers Tools**" in AWS, proceed to "**Create Repository**" specify the desired Repository Name along with relevant tags, and conclude the process by clicking "**Create**".

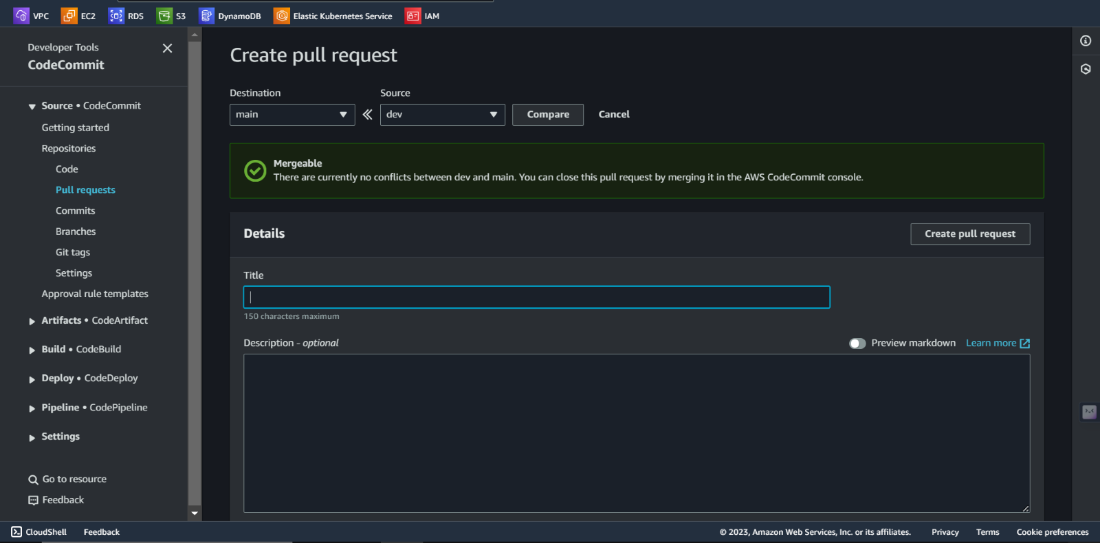
  
 *Image: Creating Code Commit Repository*

**II.** Navigate to the "**Branches**" section and initiate the creation of a "**dev**" branch from the "**main**" branch by selecting "**Create branch**" enabling a structured development environment.

  
  *Image: Creating dev branch from main*

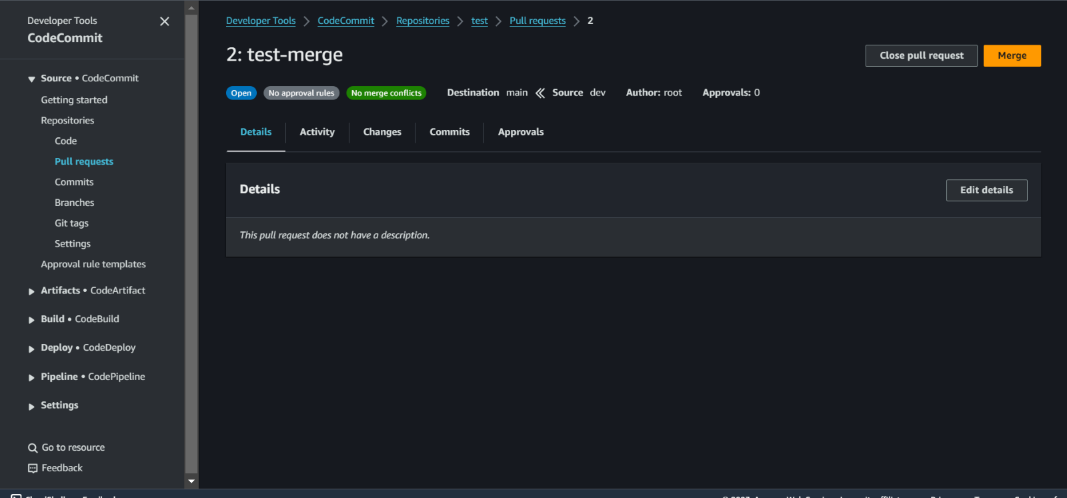
**III.** Now for hosting the constructed Terraform code in the CodeCommit repository, navigate to the repository console, click on "**Clone URL**" and select "**Clone HTTPS**" to obtain the clone URL. Subsequently, execute the provided commands in CLI to seamlessly push the code to the designated dev branch, ensuring a streamlined and organized version control process.   
“**git clone** [**https://repository\_url**](https://repository_url/)”  
“**cd eks-infra**"  
“**git checkout dev**”  
“**git add .**”  
“**git commit -m “commit message”**”  
“**git push origin dev**”

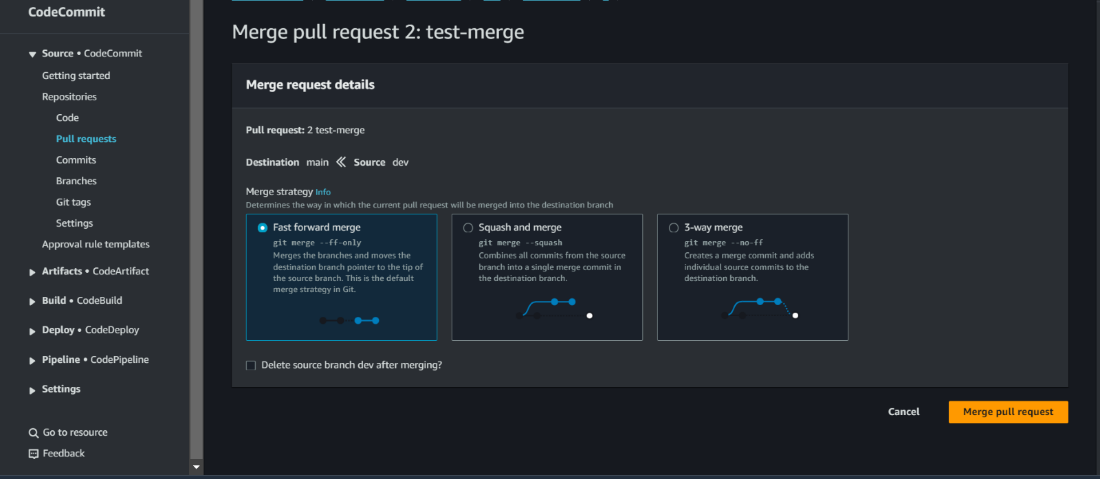
**IIII.** Creating pull request to merge the code from **dev** branch to **main** branch  
After Successfully pushing the code into **dev** branch click on “**Create pull request**” And select Source as **dev** and destination as **main.**  Give a suitable **Title** and **description** to the pull request and click on “**Create pull request”**.

  
  *Image: Creating pull request from dev branch to main*

Upon successfully pushing the code into the **dev** branch, initiate the pull request process by clicking on "**Create pull request**" Select the source as the **dev** branch and the destination as the **main** branch. Provide a pertinent title and description for the pull request, ensuring clarity and detail, and proceed by clicking on "**Create pull request**" to facilitate a structured and documented code merging process. The request will be received, and code reviews will be carried out and code will be successfully merged.

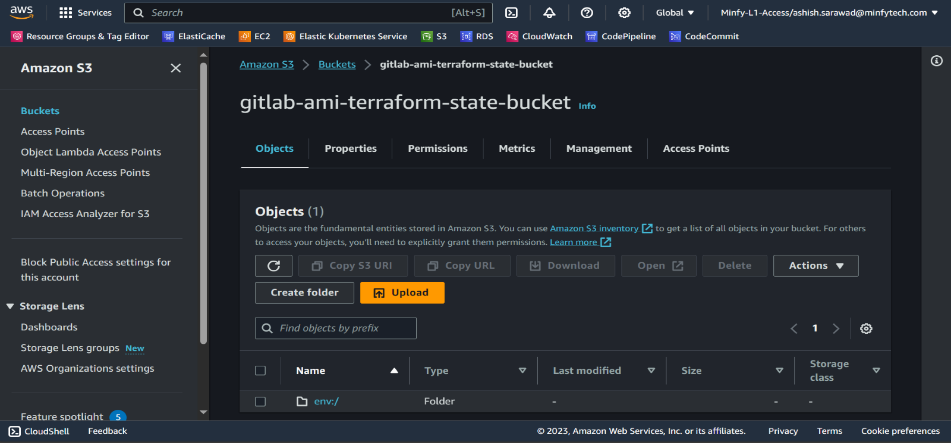
(Admin)Click on “**Pull requests**” to check the open pull requests and click on created PR, and further click on “**merge”** and select the **merge strategy** and click on merge pull request.

  
  *Image: Merging the pull request*

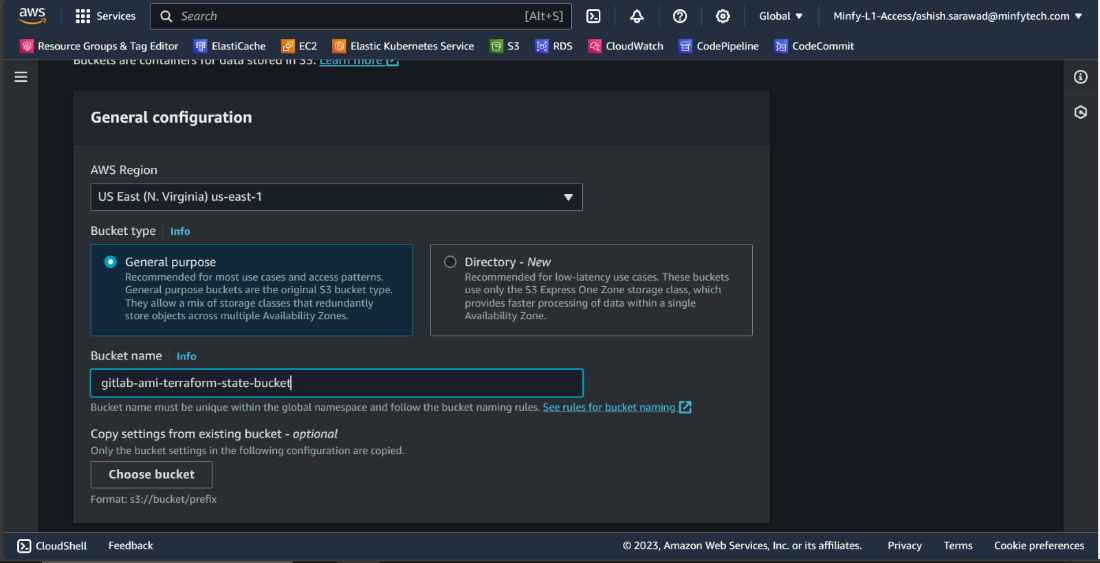
  
  *Image: Merge strategy*

Upon successful merging of code from the development (dev) to the main branch, the system is poised for further automation workflow configurations, ensuring a streamlined and efficient software development process.

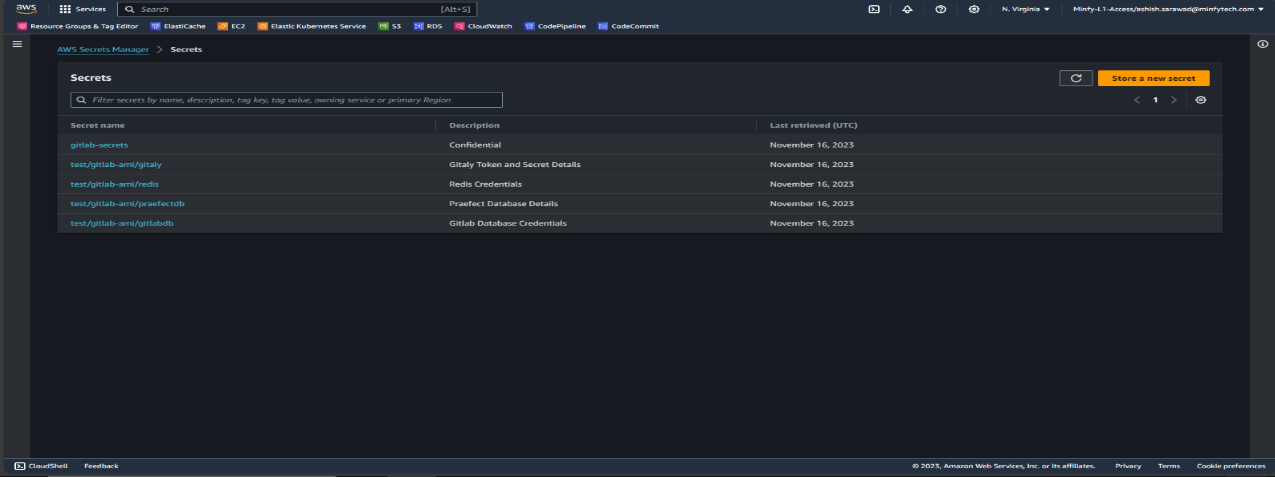
**2. S3 Backend for Terraform State Files:**  
As the deployment will heavily rely on Terraform for infrastructure provisioning, setting up S3 bucket to serve as the backend for Terraform state files is important as part of our prep-work operations. These S3 bucket must be configured with the appropriate permissions and security measures to ensure the safe and efficient storage of Terraform state files.

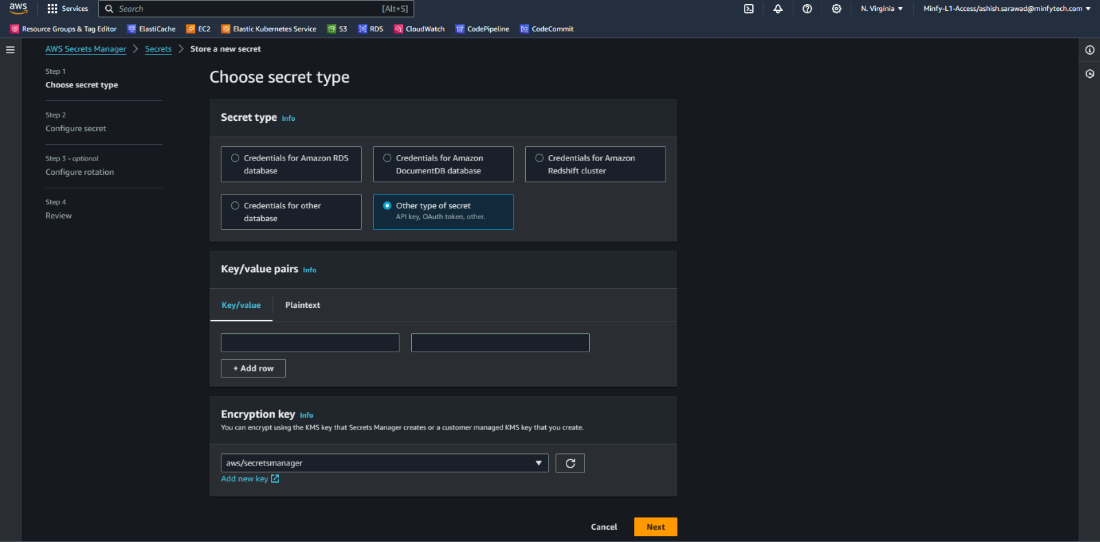
  *Image: s3 Bucket for Terraform remote state*

To establish a secure S3 bucket, access the S3 console, initiate it by clicking "**Create bucket**" process, choose the preferred region, provide a distinctive name for the bucket, maintain default settings for the remaining configurations, and conclude the setup by selecting "**Create bucket**" This ensures a methodical approach to creating a secure storage environment in Amazon S3.

  
  *Image: Creating S3 Bucket*

**3. Secrets Management:**  
Effective management of sensitive information is crucial for security and data protection. In our approach, we've developed a thorough strategy to securely store and handle critical details like Gitaly credentials, database access credentials, GitLab passwords, and various access tokens. AWS Secrets Manager is our preferred tool for this task.  
  
Here's how it works: we securely store these secrets in AWS Secrets Manager, and when needed, we access and retrieve them for use within Kubernetes Secrets. This ensures that sensitive information is handled with the utmost care.  
Moreover, our strategy includes a well-defined process for secret rotation, adding an extra layer of security and safeguarding your data effectively. It's all part of our commitment to maintaining a secure and resilient environment for your valuable information.

  *Image: Secret Manager for secret store*

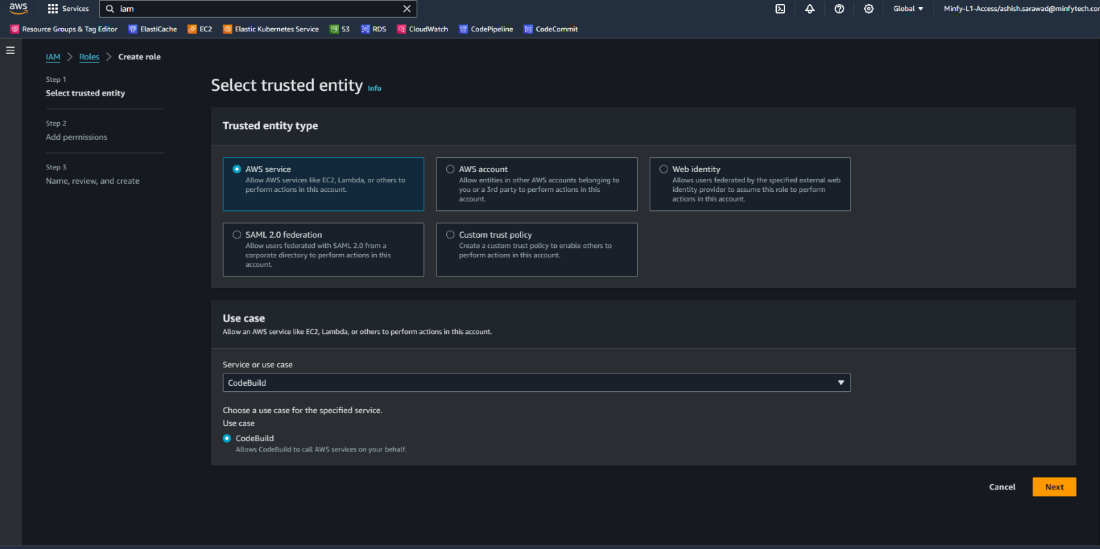
To securely store secrets in AWS Secret Manager, access the AWS Secret Manager console, initiate the process by selecting "**Store a new secret**" opt for the "**Other type of secret**" category, input the secrets in key/value pairs, proceed to the “**Next**” step. Assign a name for the secret, maintain default configurations, and finalize by clicking "**Store**" This ensures a secure and organized management of sensitive information within AWS Secret Manager.  
  
   
  *Image: Storing a secret*

**4. IAM Roles and Permissions Definition:**

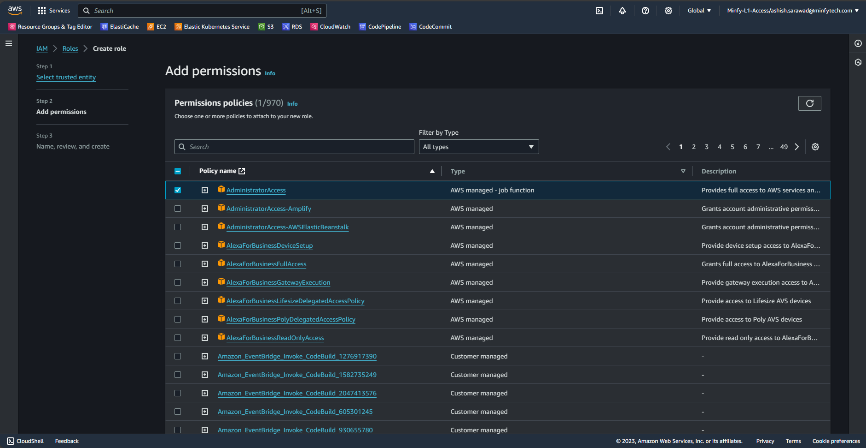
Creating IAM (Identity and Access Management) roles is essential to make sure that the right people and components have the proper permissions and access controls. We carefully define these roles to cater to different services and components within the GitLab environment. For instance, roles for CodeBuild, CodePipeline, and Terraform are precisely crafted to enable specific actions and responsibilities. It's all about ensuring that the right tasks are assigned to the right individuals and tools, maintaining a secure and well-organized environment.  
  
**Steps followed for creating IAM Roles in AWS IAM Console.**

**I.** Codebuild Service Role

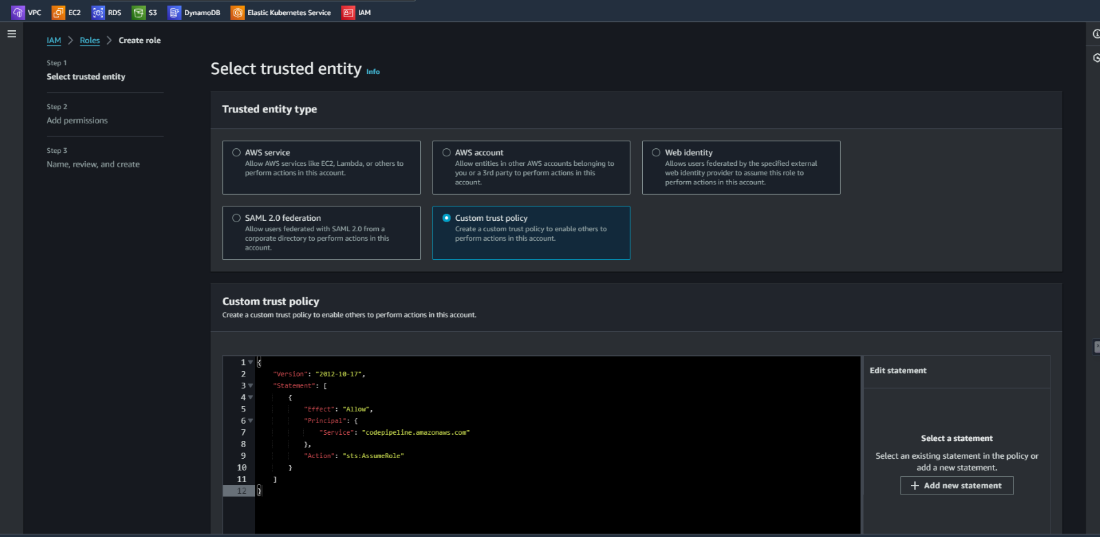
In IAM Console, click on “**Create role**” and select “**AWS Service**” and use “**Codebuild**” as Use case and click on “**Next**”. In “**Add permissions**” page click on “**Next**” and further specify a name and click on “**Create role**”.

  
  *Image: IAM Role Creation for Codebuild*

In Next step, for adding permissions to this Codebuild service Role, select “**AdministratorAccess**” for granting Administrator access for the codebuild service role as terraform will assume role on codebuild service role to provision the resources in the AWS Account

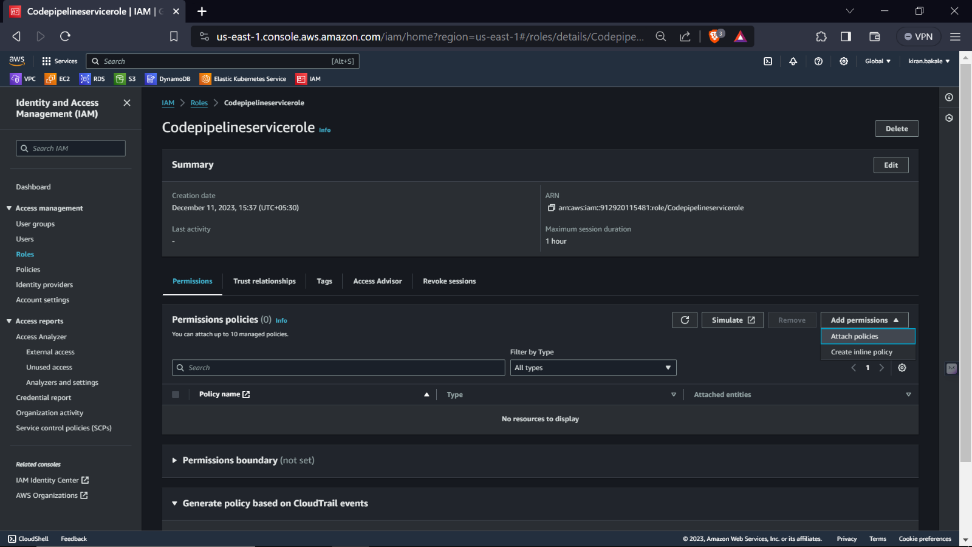
  
  *Image: Add permissions for Codebuild role*

**II.** CodePipeline Service role  
In IAM Console, click on “**Create role**” and select “**Custom trust policy**” and insert the below trust policy and click on “**Next**”. In “**Add permissions**” page click on “**Next**” and further specify a name and click on “**Create role**”.

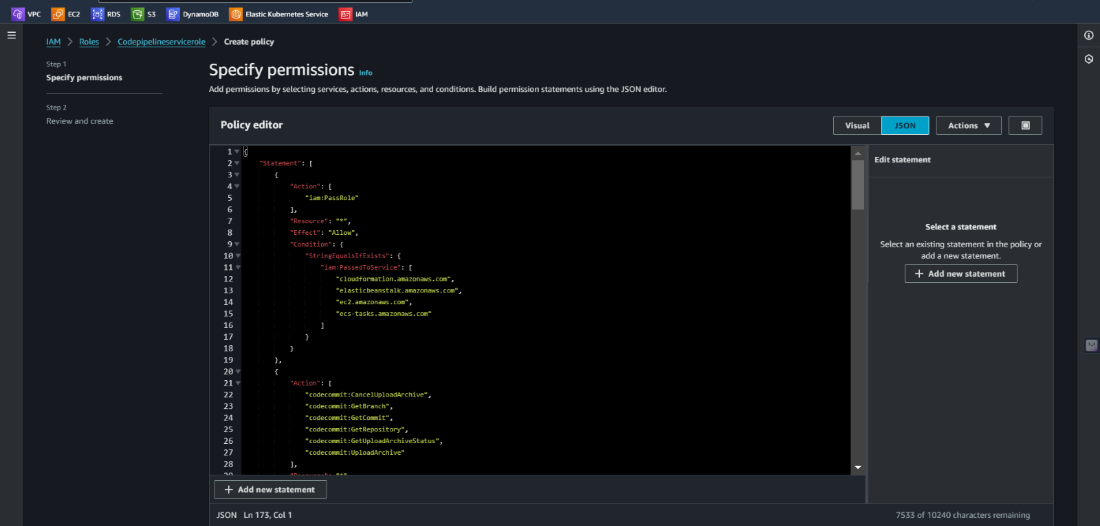
  
  *Image: Custom trust policy for codepipeline role*

|  |
| --- |
| ***{***  ***"Version": "2012-10-17",***  ***"Statement": [***  ***{***  ***"Effect": "Allow",***  ***"Principal": {***  ***"Service": "codepipeline.amazonaws.com"***  ***},***  ***"Action": "sts:AssumeRole"***  ***}***  ***]***  ***}*** |

*Code: trust policy for codepipeline service role*  
  
For adding permissions to this Codepipeline service Role,   
Click onto the “**codepipelineservicerole**”

  
  *Image: Add permissions for Codepipeline role*

Switch from “**visual**” to “**Json**” editor and insert the below given policy which will give set of permissions for the following resources and further click on “**Next**” and “**Create policy**”

  
  *Image: Inserting policy for Codepipeline role*

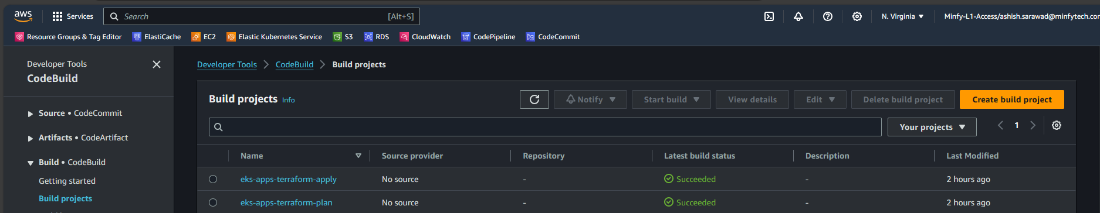
***Insert the below policy:***

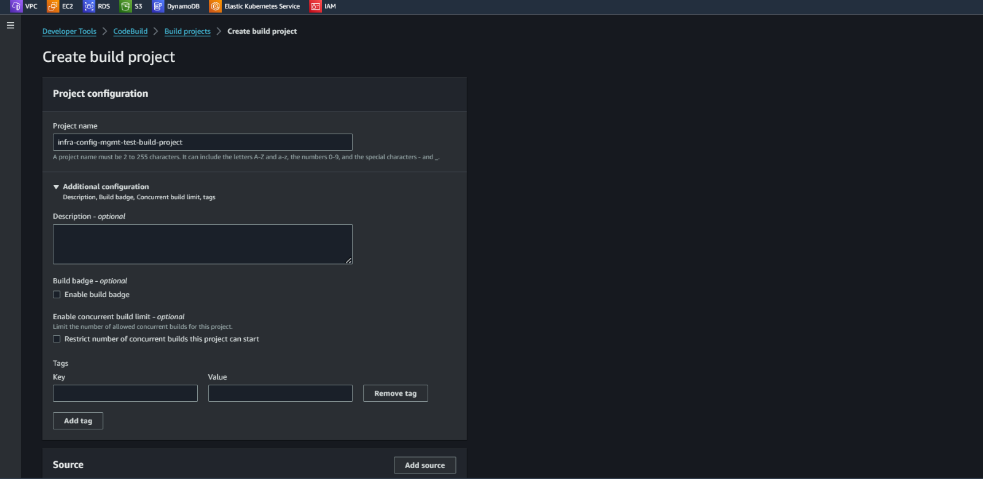
***Code : policy document for codepipeline service role***

|  |
| --- |
| ***{***  ***"Statement": [***  ***{***  ***"Action": [***  ***"iam:PassRole"***  ***],***  ***"Resource": "\*",***  ***"Effect": "Allow",***  ***"Condition": {***  ***"StringEqualsIfExists": {***  ***"iam:PassedToService": [***  ***"cloudformation.amazonaws.com",***  ***"elasticbeanstalk.amazonaws.com",***  ***"ec2.amazonaws.com",***  ***"ecs-tasks.amazonaws.com"***  ***]***  ***}***  ***}***  ***},***  ***{***  ***"Action": [***  ***"codecommit:CancelUploadArchive",***  ***"codecommit:GetBranch",***  ***"codecommit:GetCommit",***  ***"codecommit:GetRepository",***  ***"codecommit:GetUploadArchiveStatus",***  ***"codecommit:UploadArchive"***  ***],***  ***"Resource": "\*",***  ***"Effect": "Allow"***  ***},***  ***{***  ***"Action": [***  ***"codedeploy:CreateDeployment",***  ***"codedeploy:GetApplication",***  ***"codedeploy:GetApplicationRevision",***  ***"codedeploy:GetDeployment",***  ***"codedeploy:GetDeploymentConfig",***  ***"codedeploy:RegisterApplicationRevision"***  ***],***  ***"Resource": "\*",***  ***"Effect": "Allow"***  ***},***  ***{***  ***"Action": [***  ***"codestar-connections:UseConnection"***  ***],***  ***"Resource": "\*",***  ***"Effect": "Allow"***  ***},***  ***{***  ***"Action": [***  ***"elasticbeanstalk:\*",***  ***"ec2:\*",***  ***"elasticloadbalancing:\*",***  ***"autoscaling:\*",***  ***"cloudwatch:\*",***  ***"s3:\*",***  ***"sns:\*",***  ***"cloudformation:\*",***  ***"rds:\*",***  ***"sqs:\*",***  ***"ecs:\*"***  ***],***  ***"Resource": "\*",***  ***"Effect": "Allow"***  ***},***  ***{***  ***"Action": [***  ***"lambda:InvokeFunction",***  ***"lambda:ListFunctions"***  ***],***  ***"Resource": "\*",***  ***"Effect": "Allow"***  ***},***  ***{***  ***"Action": [***  ***"opsworks:CreateDeployment",***  ***"opsworks:DescribeApps",***  ***"opsworks:DescribeCommands",***  ***"opsworks:DescribeDeployments",***  ***"opsworks:DescribeInstances",***  ***"opsworks:DescribeStacks",***  ***"opsworks:UpdateApp",***  ***"opsworks:UpdateStack"***  ***],***  ***"Resource": "\*",***  ***"Effect": "Allow"***  ***},***  ***{***  ***"Action": [***  ***"cloudformation:CreateStack",***  ***"cloudformation:DeleteStack",***  ***"cloudformation:DescribeStacks",***  ***"cloudformation:UpdateStack",***  ***"cloudformation:CreateChangeSet",***  ***"cloudformation:DeleteChangeSet",***  ***"cloudformation:DescribeChangeSet",***  ***"cloudformation:ExecuteChangeSet",***  ***"cloudformation:SetStackPolicy",***  ***"cloudformation:ValidateTemplate"***  ***],***  ***"Resource": "\*",***  ***"Effect": "Allow"***  ***},***  ***{***  ***"Action": [***  ***"codebuild:BatchGetBuilds",***  ***"codebuild:StartBuild",***  ***"codebuild:BatchGetBuildBatches",***  ***"codebuild:StartBuildBatch"***  ***],***  ***"Resource": "\*",***  ***"Effect": "Allow"***  ***},***  ***{***  ***"Effect": "Allow",***  ***"Action": [***  ***"devicefarm:ListProjects",***  ***"devicefarm:ListDevicePools",***  ***"devicefarm:GetRun",***  ***"devicefarm:GetUpload",***  ***"devicefarm:CreateUpload",***  ***"devicefarm:ScheduleRun"***  ***],***  ***"Resource": "\*"***  ***},***  ***{***  ***"Effect": "Allow",***  ***"Action": [***  ***"servicecatalog:ListProvisioningArtifacts",***  ***"servicecatalog:CreateProvisioningArtifact",***  ***"servicecatalog:DescribeProvisioningArtifact",***  ***"servicecatalog:DeleteProvisioningArtifact",***  ***"servicecatalog:UpdateProduct"***  ***],***  ***"Resource": "\*"***  ***},***  ***{***  ***"Effect": "Allow",***  ***"Action": [***  ***"cloudformation:ValidateTemplate"***  ***],***  ***"Resource": "\*"***  ***},***  ***{***  ***"Effect": "Allow",***  ***"Action": [***  ***"ecr:DescribeImages"***  ***],***  ***"Resource": "\*"***  ***},***  ***{***  ***"Effect": "Allow",***  ***"Action": [***  ***"states:DescribeExecution",***  ***"states:DescribeStateMachine",***  ***"states:StartExecution"***  ***],***  ***"Resource": "\*"***  ***},***  ***{***  ***"Effect": "Allow",***  ***"Action": [***  ***"appconfig:StartDeployment",***  ***"appconfig:StopDeployment",***  ***"appconfig:GetDeployment"***  ***],***  ***"Resource": "\*"***  ***}***  ***],***  ***"Version": "2012-10-17"***  ***}*** |

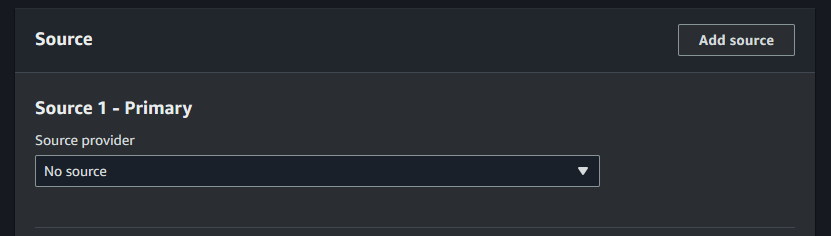
**5. CodeBuild Project Configuration:**

The CodeBuild project is technically configured to automate build and deployment processes. This includes specifying build environment parameters, setting up source code integration with Git repositories, configuring build specifications using buildspec commands and CodePipelines. Technical details encompass the choice of build runtime, build phases, and environment variables.  
  
In this automated workflow, we've implemented **two** CodeBuild projects to facilitate efficient infrastructure provisioning and application deployments. These Two Build projects will manage Terraform operations—one for planning and another for applying changes which are linked to the **Four** Codepipelines. A manual approval step follows the plan, ensuring a controlled deployment process.  
  
we have streamlined the process by consolidating the deployment into two CodeBuild projects, optimizing efficiency across four pipelines. The first pipeline handles infrastructure provisioning, while the second pipeline caters to both common EKS components and GitLab application provisioning. The third pipeline is dedicated to GitLab application provisioning, and the fourth pipeline is designed for Keycloak provisioning. This setup enhances simplicity and maintains flexibility for future additions, with the two CodeBuild projects supporting all four pipelines seamlessly.

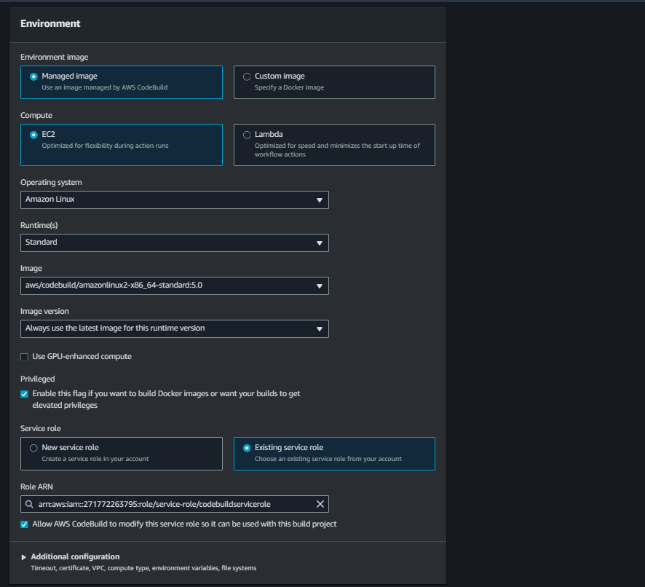
  *Image: Code Build Projects*

**Steps to configure a 2 CodeBuild projects with each use case follow the below steps**  
**I. CodeBuild Project Setup for Terraform Plan:** Access the CodeBuild Console, initiate the process by selecting "**Create build project**" and proceed by entering a name “**terraform-plan**” along with relevant tags based on specific project requirements.  
   
   
 *Image: Creating Code Build project – Project Configuration*

In Source section, choose "**No source**”

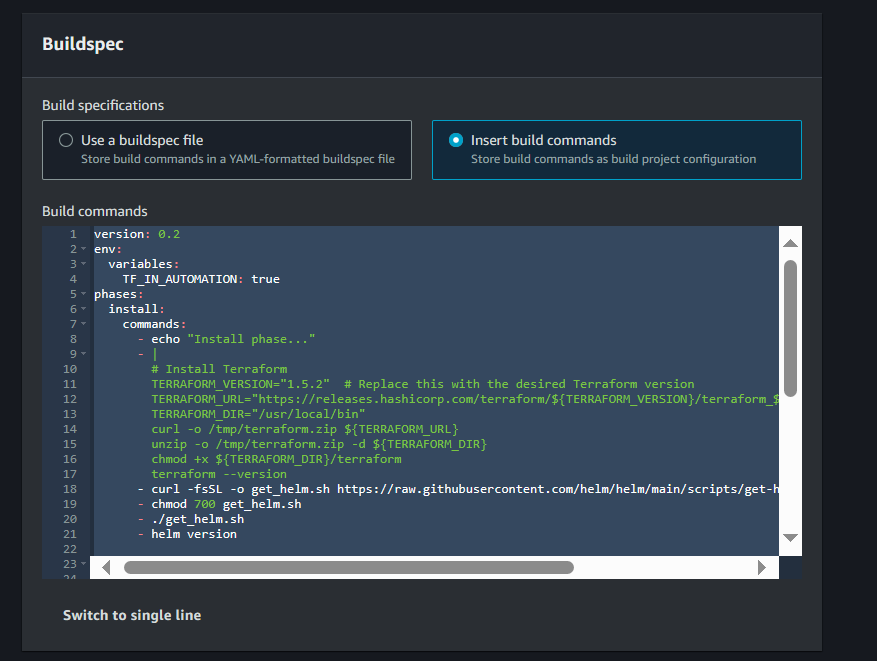
  
   *Image: Creating Code Build project – Source*

In Environment section, Choose Environment image as “**Managed image**”, Compute as “**EC2**”, Operating system as “**Amazon Linux**”, Runtime(s) as “**Standard**”, Image as “**aws/codebuild/amazonlinux2-x86\_64-standard5.0**”, image version as “**aws/codebuild/amazonlinux2-x86\_64-standard5.0-23.05.22”.** Enable the “**Privileged**” to allow codebuild to build docker images. For service role Choose “**Existing service role**” and in Role ARN choose the previously created “**codebuildservicerole**”

Note: The image versions will keep on updating make sure to always choose the latest version  
  
   
  *Image: Creating Code Build project – Environment*

keep Additional Configuration at its default settings. In Buildspec section, choose “**Insert build commands**” and insert the below buildspec code in it. This assignment ensures that the buildspec commands executes the necessary Terraform plan operations.

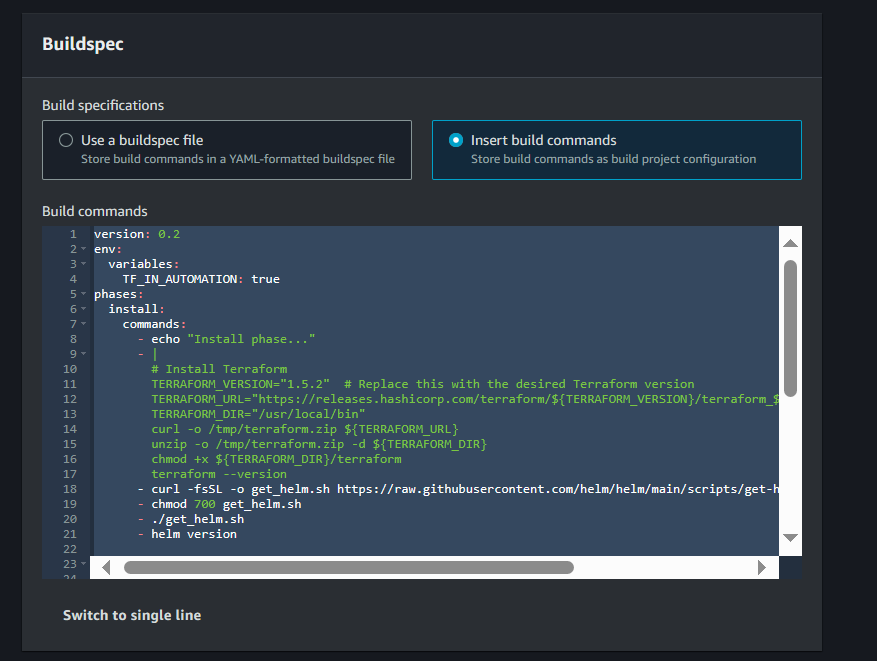
|  |
| --- |
| ***version: 0.2***  ***env:***  ***variables:***  ***TF\_IN\_AUTOMATION: true***  ***phases:***  ***install:***  ***commands:***  ***- echo "Install phase..."***  ***- |***  ***# Install Terraform***  ***TERRAFORM\_VERSION="1.5.2" # Replace this with the desired Terraform version***  ***TERRAFORM\_URL="https://releases.hashicorp.com/terraform/${TERRAFORM\_VERSION}/terraform\_${TERRAFORM\_VERSION}\_linux\_amd64.zip"***  ***TERRAFORM\_DIR="/usr/local/bin"***  ***curl -o /tmp/terraform.zip ${TERRAFORM\_URL}***  ***unzip -o /tmp/terraform.zip -d ${TERRAFORM\_DIR}***  ***chmod +x ${TERRAFORM\_DIR}/terraform***  ***terraform --version***  ***- curl -fsSL -o get\_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 #helm install***  ***- chmod 700 get\_helm.sh***  ***- ./get\_helm.sh***  ***- helm version***  ***pre\_build:***  ***on-failure: ABORT***  ***commands:***    ***- |***  ***if [ "$APP" == "gitlab" ]; then***  ***ls***  ***echo "Fetching the latest code..."***  ***git config --global credential.helper '!aws codecommit credential-helper $@'***  ***git config --global credential.UseHttpPath true***  ***git clone https://git-codecommit.us-east-1.amazonaws.com/v1/repos/eks-apps > /dev/null 2>&1***  ***cd eks-apps***  ***git diff HEAD HEAD~ --name-only***  ***CHANGED\_FILES=$(git diff HEAD HEAD~ --name-only)***  ***echo $CHANGED\_FILES***  ***### getting the gitlab version***  ***gitlab\_version=$(awk -v RS='}' '/variable\s+"gitlabversion"/{print $0}' $APP/variables.tf | grep -oP 'default\s+=\s+"\K[^"]+')***  ***echo $gitlab\_version***    ***if echo "$CHANGED\_FILES" | grep -q "$APP/webservice-custom-build/"; then***  ***echo "Building Docker image as changes detected in 'webservice-custom-build' folder."***  ***ecr\_repo\_url="271772263795.dkr.ecr.us-east-1.amazonaws.com/gitlab-webservice-ce"***  ***cd gitlab/webservice-custom-build***  ***pwd***  ***docker build -t $ecr\_repo\_url:v$gitlab\_version --build-arg gitlabversion=v$gitlab\_version .***  ***aws ecr get-login-password | docker login --username AWS --password-stdin $ecr\_repo\_url***  ***docker push $ecr\_repo\_url:v$gitlab\_version***  ***ls***  ***cd ..***  ***ls***  ***cd ../..***  ***pwd***  ***rm -rf eks-apps***  ***else***  ***echo "No changes detected in 'webservice-custom-build' folder. Skipping Docker image build."***  ***ls***  ***cd ..***  ***rm -rf eks-apps***  ***fi***    ***### Workhorse***    ***if echo "$CHANGED\_FILES" | grep -q "$APP/workhorse-custom-build/"; then***  ***echo "Building Docker image as changes detected in 'workhorse-custom-build' folder."***  ***workhorse\_ecr\_repo\_url="271772263795.dkr.ecr.us-east-1.amazonaws.com/gitlab-workhorse-ce"***  ***cd gitlab/workhorse-custom-build***  ***docker build -t $workhorse\_ecr\_repo\_url:v$gitlab\_version --build-arg gitlabversion=v$gitlab\_version .***  ***aws ecr get-login-password | docker login --username AWS --password-stdin $workhorse\_ecr\_repo\_url***  ***docker push $workhorse\_ecr\_repo\_url:v$gitlab\_version***  ***cd ../..***  ***pwd***  ***ls***  ***rm -rf eks-apps***  ***else***  ***echo "No changes detected in 'workhorse-custom-build' folder. Skipping Docker image build."***  ***rm -rf eks-apps***  ***ls***  ***fi***  ***else***  ***echo "Not a gitlab directory."***  ***fi***  ***- ls***    ***build:***  ***on-failure: ABORT***  ***commands:***  ***- echo $APP***  ***- echo $PWD***  ***- ls***  ***- cd $PWD/$APP***  ***- rm -rf .terraform/***  ***- ls***  ***- terraform init --input=false***  ***- echo "Terraform Plan for $TAG"***  ***- terraform plan --out=tfplan --input=false***    ***artifacts:***  ***files:***  ***- '\*\*/\*'***  ***name: TFPLAN*** |

  
  *Image: Creating Code Build project – Buildspec (Plan)*

Maintain default configurations for "**Artifacts**" and "**Logs**" allowing for seamless processes. Optionally, input a custom "**Group name**" if needed otherwise, the system will utilize the Build project's name, generating a corresponding log group in CloudWatch.

**II. CodeBuild Project Setup for Terraform Apply:** Proceed with creating another build project named " **terraform-apply**" to handle Terraform apply operations. In the buildspec section, choose “**insert build commands**” and insert the script given below. Once these steps are completed, the build projects for infrastructure provisioning and application deployment will be set up and ready to use.

|  |
| --- |
| ***version: 0.2***  ***env:***  ***variables:***  ***TF\_IN\_AUTOMATION: true***  ***phases:***  ***install:***  ***commands:***  ***- echo "Install phase..."***  ***- |***  ***# Install Terraform***  ***TERRAFORM\_VERSION="1.5.2" # Replace this with the desired Terraform version***  ***TERRAFORM\_URL="https://releases.hashicorp.com/terraform/${TERRAFORM\_VERSION}/terraform\_${TERRAFORM\_VERSION}\_linux\_amd64.zip"***  ***TERRAFORM\_DIR="/usr/local/bin"***  ***curl -o /tmp/terraform.zip ${TERRAFORM\_URL}***  ***unzip -o /tmp/terraform.zip -d ${TERRAFORM\_DIR}***  ***chmod +x ${TERRAFORM\_DIR}/terraform***  ***terraform --version***  ***- curl -fsSL -o get\_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 #helm install***  ***- chmod 700 get\_helm.sh***  ***- ./get\_helm.sh***  ***- helm version***  ***build:***  ***on-failure: ABORT***  ***commands:***  ***# - echo "Fetching the latest code..."***  ***# - git config --global credential.helper '!aws codecommit credential-helper $@'***  ***- echo "Fetching the latest code..."***  ***- echo $APP***  ***- echo $PWD***  ***- ls***  ***- cd $PWD/$APP***  ***- terraform apply tfplan*** |

  
  *Image: Creating Code Build project – Buildspec (apply)*

**6. CodePipeline Automation:**

CodePipeline is established with a technical focus on orchestrating the deployment workflow. The pipeline is created to define stages, manual approval actions, and transitions between stages. Technical details include specifying source, build, test, and deployment actions within the pipeline. Integration with Git repositories and AWS services is finely tuned to automate the deployment process.

These pipelines will be triggered through tag-based pushing, providing a systematic and version-controlled approach to initiating the deployment processes. By associating specific tags with commits, we ensure a deliberate and organized trigger mechanism for the pipelines, enhancing control and visibility over the deployment lifecycle.

We have implemented totally **Four** CodePipelines for infrastructure & Application deployment is designed with four distinct stages.

Firstly, in the **Source Stage**, the CodePipeline initiates by checking out the source code from the designated repository.  
Moving on to the second stage, the **Terraform Plan Stage**, a CodeBuild project is triggered to execute the **Terraform Plan.** This stage provides detailed insights in the build logs, outlining the resources that will be either deployed or updated.

The third stage introduces a critical **Manual Approval** Step. Here, an **SNS topic** is triggered to notify stakeholders, prompting them to review the **Terraform plan**. Upon their approval, the pipeline proceeds to the next stage.

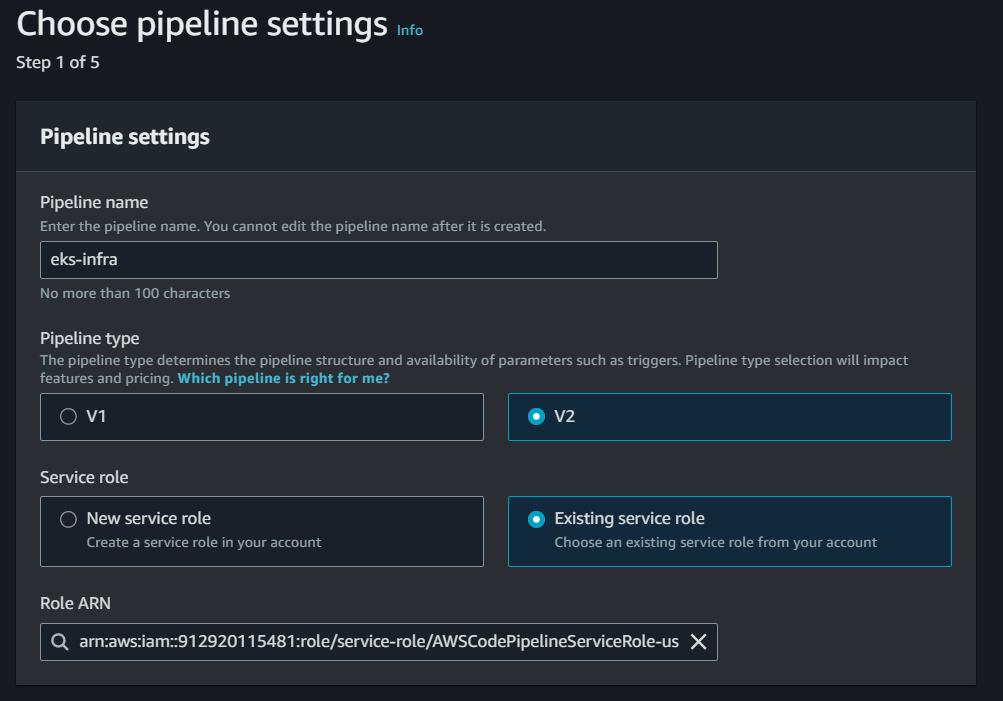
In the final stage, the **Terraform Apply Stage**, the pipeline executes the **Terraform Apply** automatically applying the Terraform configurations. This results in the creation of the specified infrastructure and Application deployment depending upon the tag pushed to the repository as per the approved plan.

This structured pipeline ensures a comprehensive and controlled approach to infrastructure provisioning.

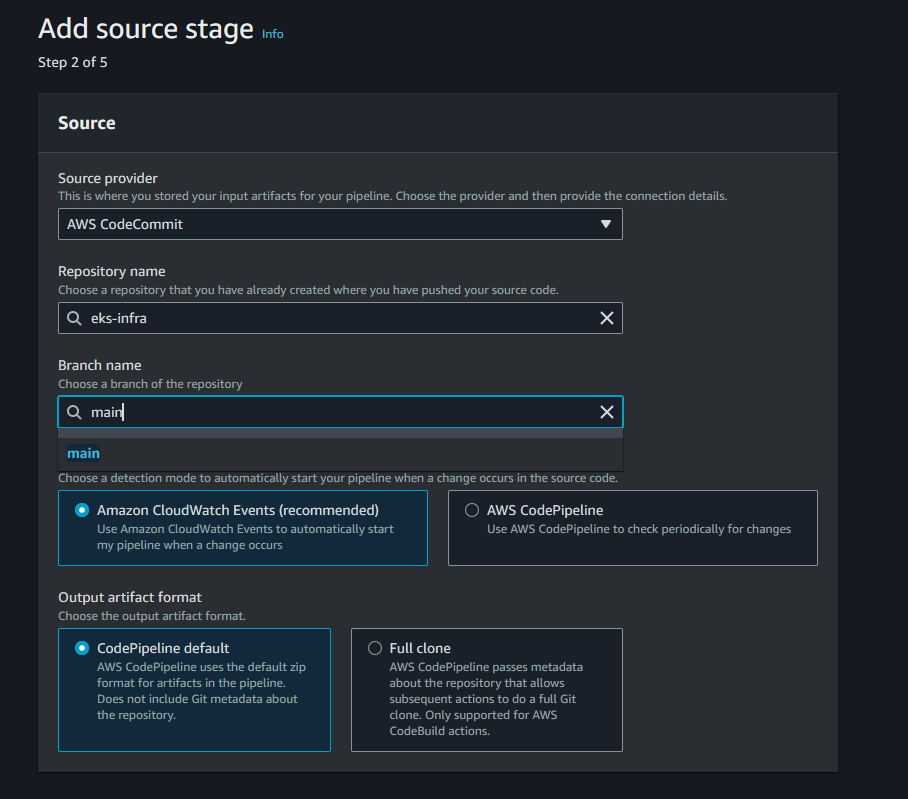
  *Image: Code Pipeline for infrastructure provisioning*

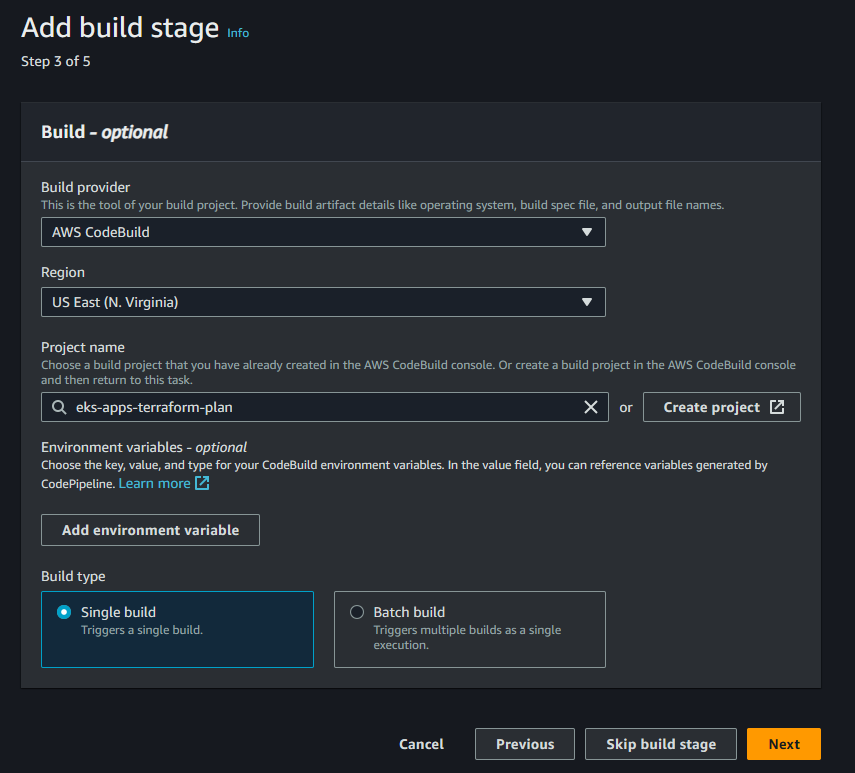
**I. Codepipeline Infrastructure Provisioning:**

For setting up a robust pipeline, access the CodePipline Console in Developers Tools. Click “**Create codepipeline**”. Enter a suitable name for the pipeline we have named it as “**eks-infra**” and choose pipeline type as “**V2**” and in Service Role, select “**Existing service role**” then choose previously created **codepipeline service role** and click on “**Next**”.

  
  *Image: Creating Code Pipeline - Pipeline settings*

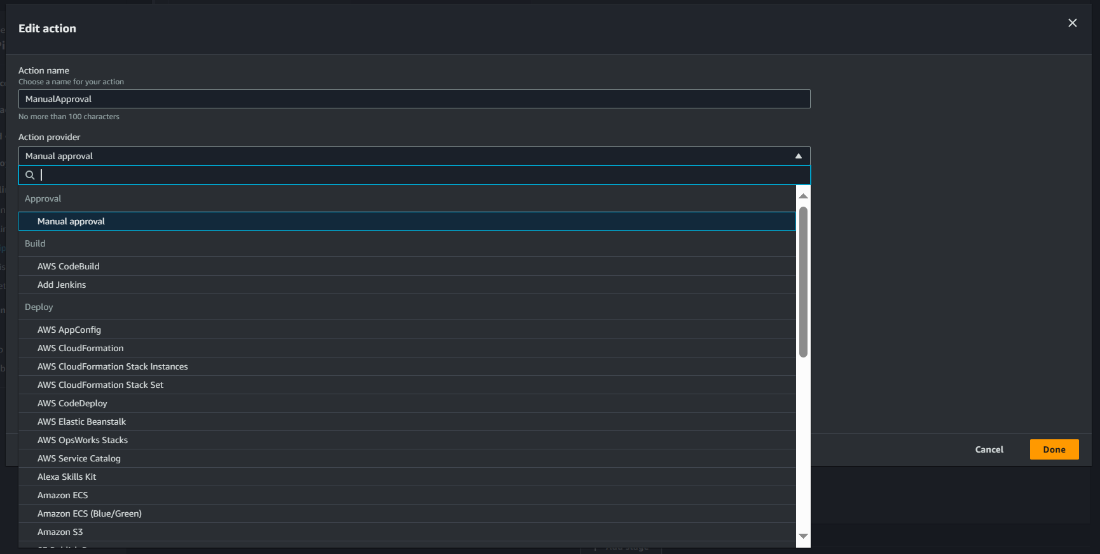
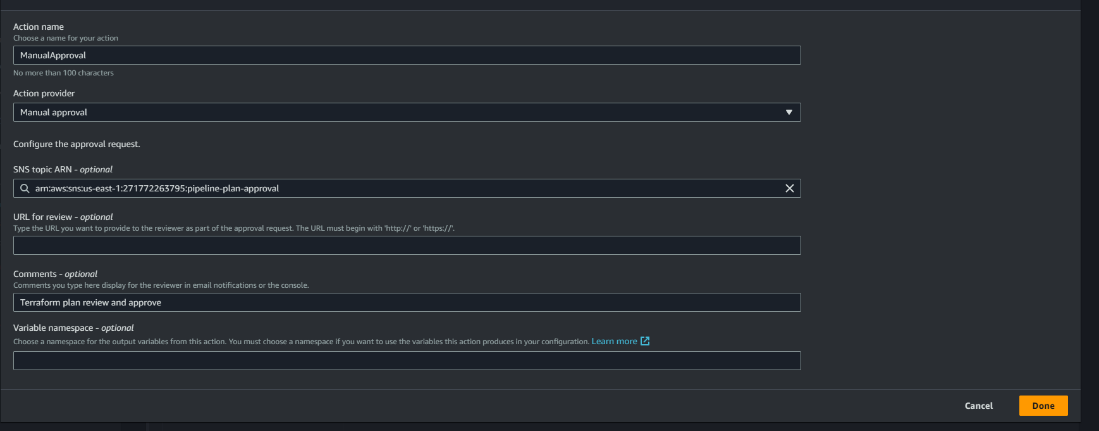
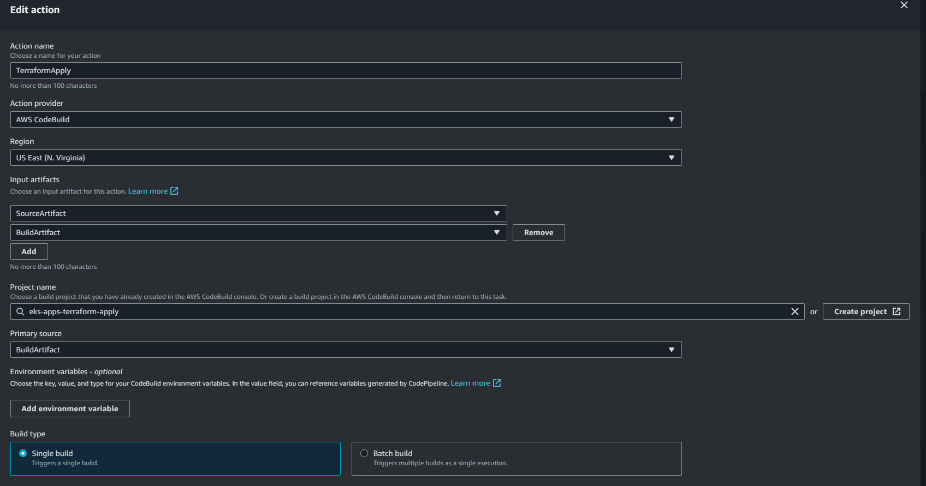
In Add Source stage, Choose **Source provider** as “**AWS CodeCommit**” in Repository name choose the previously created repository name i.e. “**eks-infra**” and Branch name as “**main**” then click on “**Next**”.

  
  *Image: Creating Code Pipeline - Source Stage*

In Build stage, Choose **Build provider** as “**AWS CodeBuild**” let the region be as default region, in Project name choose the project name created previously “**eks-apps-terraform-plan”** and click on “**Next**”  
   
  *Image: Creating Code Pipeline - Build Stage*

In the "**Deploy**" stage, opt to "**Skip deploy stage**" as it's unnecessary for our setup. Proceed to the **Review page** and select "**Create Pipeline**" Now Navigate inside the pipeline and click onto "**Edit**" then click on "**+ Add Stage**" following the **build stage** to introduce a new one stage.

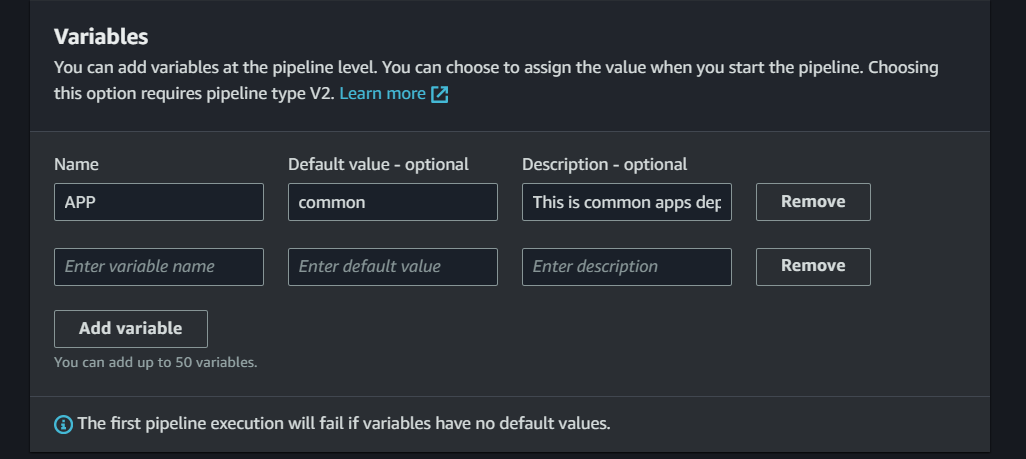
Name this stage as "**Approval Stage**" and add an action by clicking on "**Add action**" Label it "**ManualApproval**" and choose "**Manual approval**" as the action provider. Paste the ARN of the SNS topic for integration then Click on “**Done**”

***Follow this AWS Document for creating SNS topic and Subscribe to the topic***  
[Creating an Amazon SNS topic - Amazon Simple Notification Service](https://docs.aws.amazon.com/sns/latest/dg/sns-create-topic.html)  
*Image: CodePipeline Adding manual approval stage*  
  
   
  *Image: CodePipeline Adding manual approval stage*  
  
  
Next, add another stage by clicking on "**+ Add Stage**" after the manual approval stage. Name it "**Terraform Apply**" and include an action named "**TerraformApply**" For the **Action provider**, choose "**AWS CodeBuild**" and set the region as "**US-East (N.virginia)**" In Input artifacts, select "**SourceArtifact**" click on "**Add**" choose "**BuildArtifact**" and specify the project name as "**eks-apps-terraform-apply**" created earlier. Set "**Primary source**" as "**BuildArtifact**" and click on "**Done**" And lastly **save** the pipeline.  
  
   
 *Image: CodePipeline Adding Terraform Apply stage*

**II. Codepipeline Applications Provisioning:**

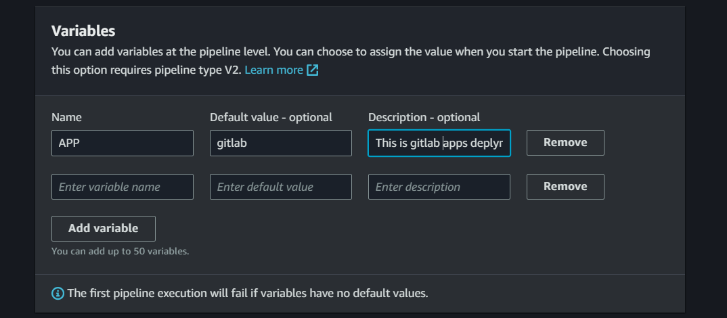
Expanding the automated deployment workflows to application-specific pipelines involves the creation of three dedicated CodePipelines, each tailored to handle distinct directories within the eks-apps repository. The goal is to achieve precise deployments by incorporating environment variables that specify the target application directory whenever there is a change in specific directory.  
**For Eks Common components pipeline**:

In **Choose pipeline settings** we have designated the pipeline as **eks-common-app.**  
In Variables, add a variable called **APP** and its value as **common.** This variable, set to "**common**" directs the pipeline to the common directory within the eks-apps repository. The pipeline orchestrates the deployment of fundamental EKS components, including kube Prometheus stack, fluentbit, cluster-autoscaler, AWS load balancer controller, metrics-server, and authentication mechanisms. The inclusion of the APP variable ensures a focused execution of Terraform plan and apply operations in **common** directory present in **eks-apps** repository. Further follow the same steps as followed in the infrastructure provisioning pipeline i.e.**eks-infra.**

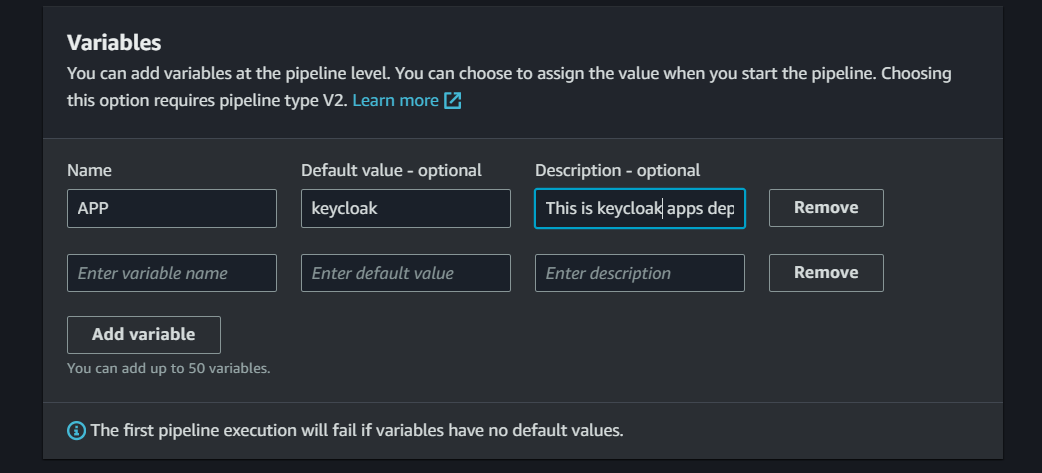
  
  *Image: Adding variables in eks-common pipeline*

**For Gitlab Application deployment pipeline**:

In **Choose pipeline settings** we have designated the pipeline as **eks-gitlab-app.**  
In Variables, add a variable called **APP** and its value as **gitlab.** This variable, set to "**gitlab**" directs the pipeline to the common directory within the eks-apps repository. This pipeline specializes in deploying the GitLab application onto the EKS cluster. By utilizing the APP variable, the pipeline navigates to the gitlab directory in the eks-apps repository, seamlessly executing the necessary Terraform operations in **gitlab** directory present in **eks-apps** repository for GitLab deployment. Further follow the same steps as followed in the infrastructure provisioning pipeline i.e.**eks-infra**

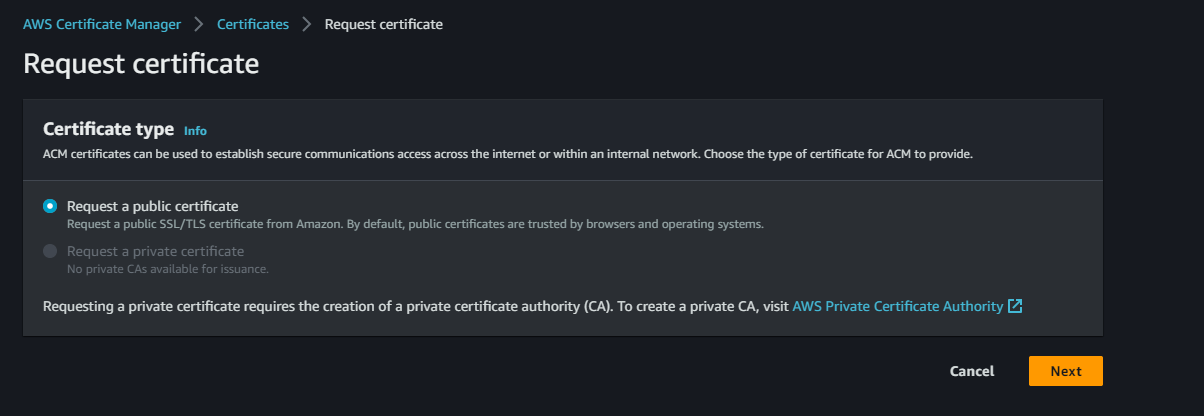
   
 *Image: Adding variables in eks-gitlab pipeline*

**For Keycloack Application deployment pipeline**:  
In **Choose pipeline settings** we have designated the pipeline as **eks-keycloack-app.**  
In Variables, add a variable called **APP** and its value as **keycloak.** This variable, set to "**keycloak**" directs the pipeline to the common directory within the eks-apps repository. This pipeline specializes in deploying the GitLab application onto the EKS cluster. By utilizing the APP variable, the pipeline navigates to the gitlab directory in the eks-apps repository, seamlessly executing the necessary Terraform operations in **keycloak** directory present in **eks-apps** repository for GitLab deployment. Further follow the same steps as followed in the infrastructure provisioning pipeline i.e.**eks-infra.**

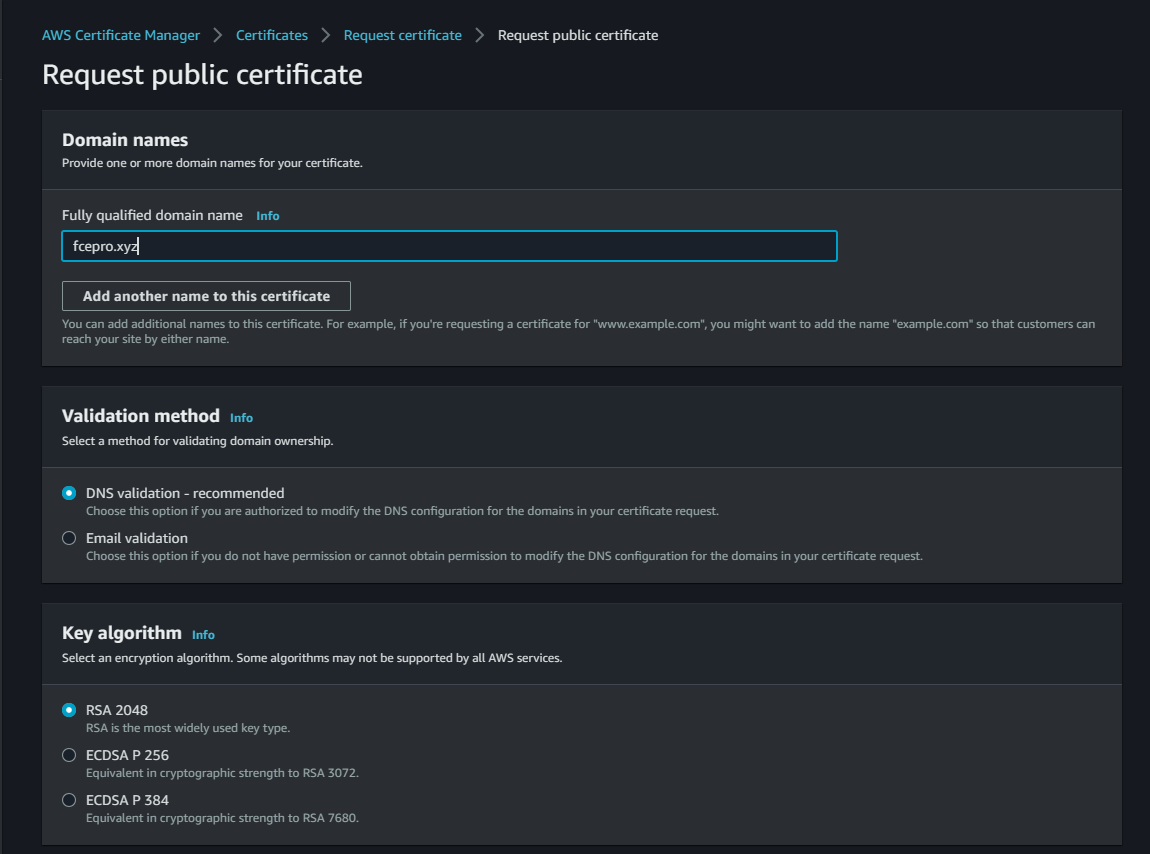
   
 *Image: Adding variables in eks-keycloak pipeline*

With these configurations, the pipeline is now set up, ensuring a seamless flow from source to manual approval and finally to the Terraform apply stage.

**7. Requesting an SSL Certificate in AWS Certificate Manager**:  
Acquiring an SSL certificate from AWS Certificate Manager is a pivotal step in our deployment process. This certificate serves as a digital credential that authenticates the identity of our domain, ensuring secure and encrypted communication between users and our infrastructure. By attaching this SSL certificate to our domain, we establish a foundation of trust and safeguard sensitive data exchanged over the network. This not only enhances the security posture of our applications but also instils confidence among users by signalling a commitment to the highest standards of data protection. The AWS Certificate Manager's seamless integration simplifies the certificate management process, allowing us to focus on delivering a secure and reliable experience to our users.  
To request a certificate from ACM, click on “**Request certificate”** in **AWS Certificate Manager** console. **Request a public certificate** will be default select as there’s no private CA created in aws account click on “**Next**”

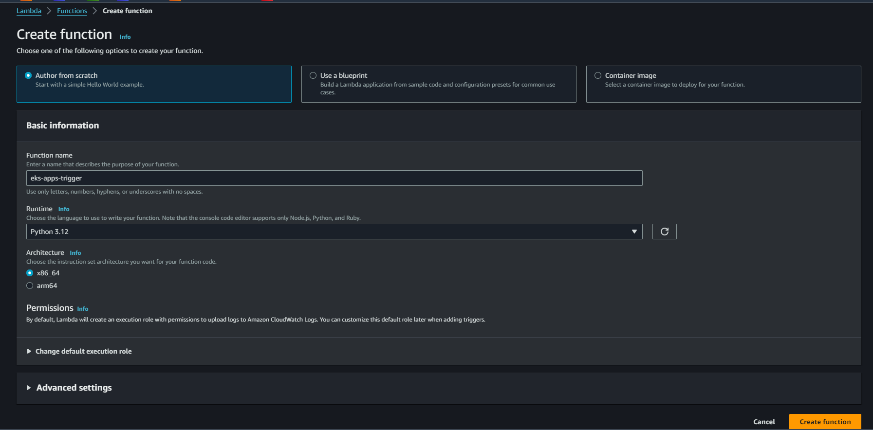
  
 *Image: Requesting SSL Certificate from ACM*

Add the fully qualified domain name for the validation and authorization for the domain name with the certificate and click on “**Request” (**it takes around 30mins to 1hour for AWS to provide the certificate)

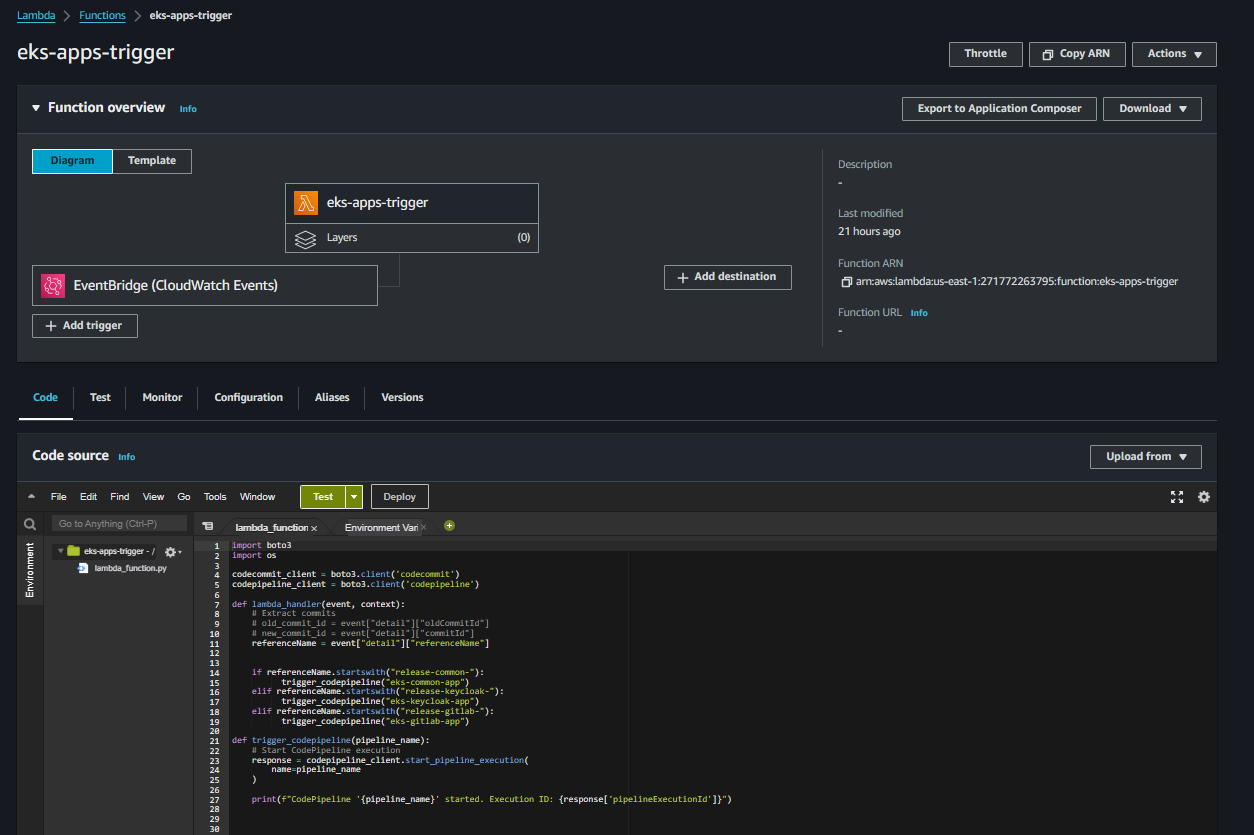
  
 *Image: Requesting SSL Certificate from ACM*

**8. Lambda Function for Application Deployment Trigger:**

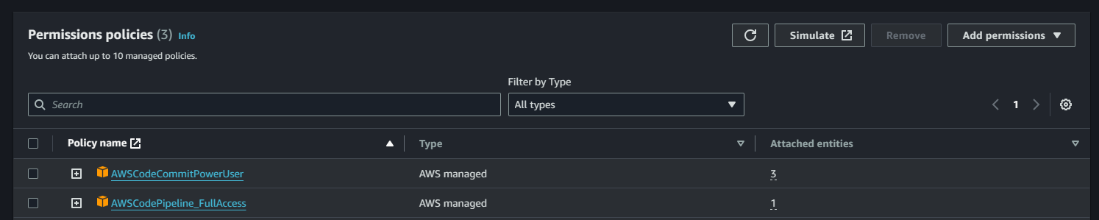
In our deployment architecture, a Lambda function plays a pivotal role in automating the initiation of Codepipeline tailored for application deployments. This Lambda function is designed to respond to tag pushes within distinct folders, such as "common," "gitlab," and "keycloak," within our source code repository i.e.**eks-apps**. When a commit occurs in any of these specific folders, the Lambda function is promptly invoked by event bridge rule which is configured to be triggered to respond on tag-based pushes. Subsequently, it orchestrates the seamless execution of corresponding Codepipelines, ensuring that updates in each application segment are efficiently and independently deployed. This streamlined approach optimizes our deployment pipeline, allowing for agile and targeted application updates triggered by specific code changes.

To Create lambda function, click on “**Create function**” in **Lambda Console.** Choose a name, we named it as **“eks-apps-trigger"** and Runtime as **Python 3.12** and click on “**Create function”**  
  
  
*Image: Create lambda Function*

Post creating the function insert the python code residing in **Python** directoryin **eks-infa** repository.

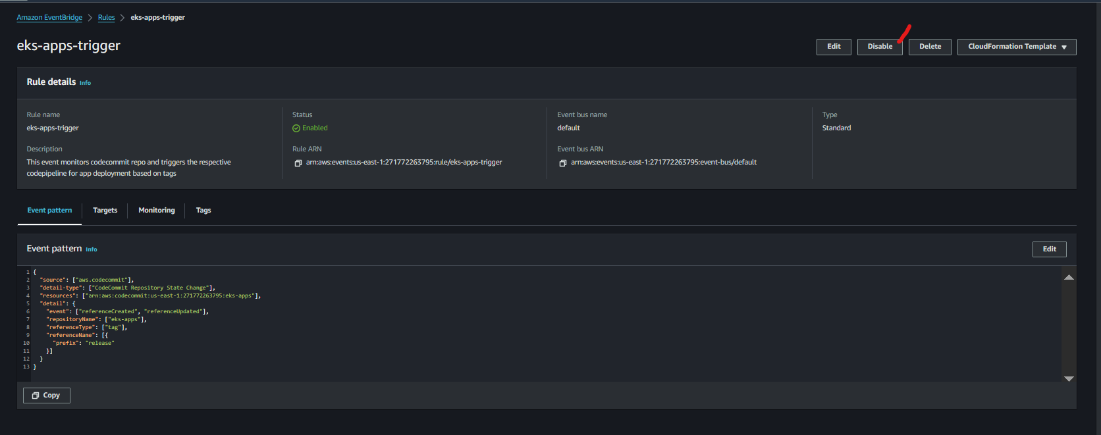
  
 *Image: Lambda Function*

And Attach navigate to **IAM Console** and find an IAM role created with the name of the lambda function created and attach two permissions to that role. One is of “**AWSCodePipelineFullAccess**” and “**AWSCodeCommitPowerUser**”. Now lambda function is ready to be set the event trigger.

   
 *Image: Permissions for Lambda Function*

**9. Creating EventBridge Rules for Pipeline Triggers:**

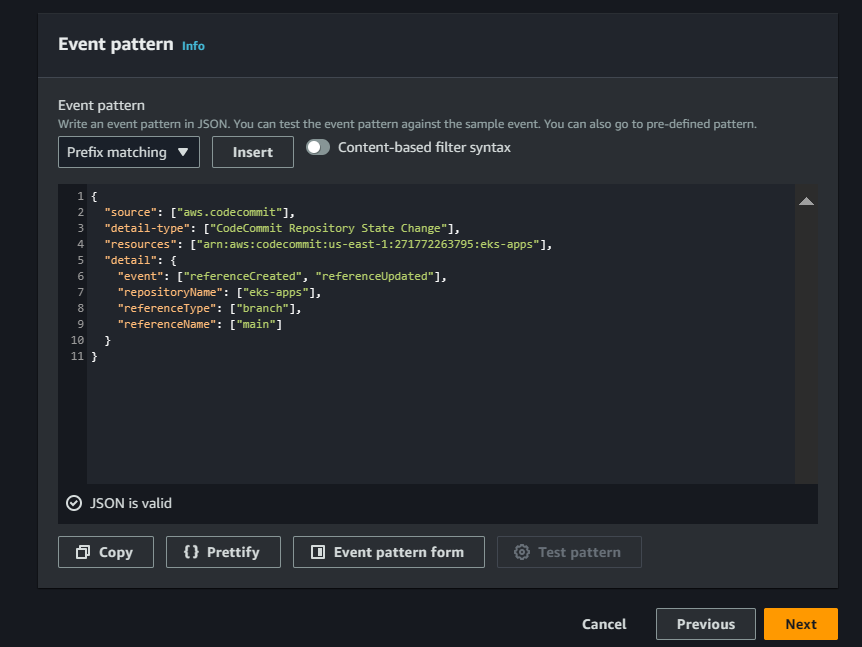
Our deployment pipeline incorporates an EventBridge rule, a key component that orchestrates automated actions based on specific events. This rule is configured to monitor commits within source code repositories. When a commit and tag push is detected, EventBridge seamlessly triggers a Lambda function and Infra pipeline. This Lambda function, acting as a catalyst, then initiates dedicated Codepipeline responsible for deploying the applications and for infrastructure it directly triggers the codepipeline. This automated event-driven mechanism ensures that any code changes are promptly and systematically translated into deployment actions, streamlining the continuous integration and delivery (CI/CD) process. By leveraging this cohesive integration of EventBridge, Lambda, and Codepipeline, we enhance the efficiency of our deployment workflow, promoting agility and responsiveness to code updates.  
  
**Prior to the creation** of new EventBridge rules, it is imperative to temporarily disable the existing rules automatically generated by CodePipelines. Navigate to the EventBridge rules section, and identify the rules associated with the pipeline names. Disable these rules to prevent them from triggering the pipelines upon each repository change. This precautionary step ensures a controlled setup for the upcoming EventBridge rules, allowing for seamless integration without interference from the pre-existing rules.

  
  *Image: Disable Existing Event Rules*   
   
**EventBridge Rule for Application Deployment:**  
To create eventbridge event rule click on “**Create rule**” in Amazon Eventbridge console  
specify a name to it, we have given “**eks-apps-trigger**" and click on “**Next**”. In **Build event pattern** scroll to last and click on **“Edit pattern”** insert the below event pattern and click on “**Next**”.

|  |
| --- |
| ***{***  ***"source": ["aws.codecommit"],***  ***"******detail-type": ["CodeCommit Repository State Change"],***  ***"resources": ["******arn:aws:codecommit:us-east-******1:271772263795:eks-apps"],***  ***"detail": {***  ***"event": ["referenceCreated", "referenceUpdated"],***  ***"repositoryName": ["eks-apps"],***  ***"referenceType": ["tag"],***  ***"referenceName": [{***  ***"prefix": "release"***  ***}]***  ***}***  ***}*** |

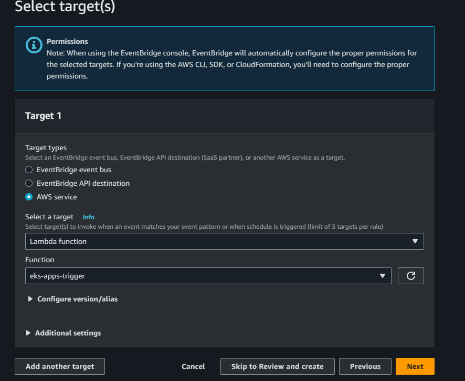
*Code: Tag-based Event pattern for lambda trigger*

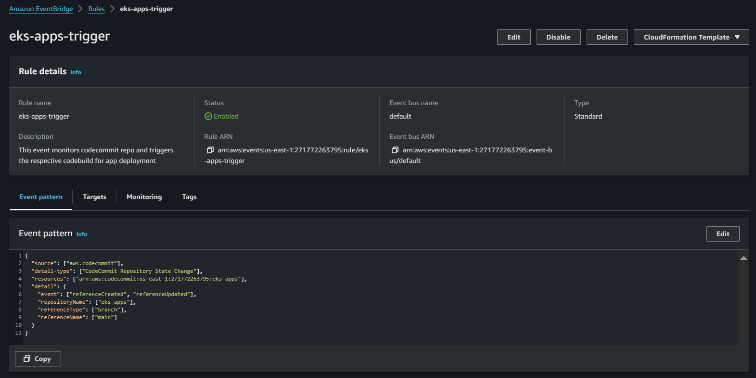
This EventBridge event pattern is designed to capture changes in the state of a CodeCommit repository, specifically targeting the "eks-apps" repository in the US East (N. Virginia) region. The pattern focuses on events related to the creation or update of references, specifically when dealing with tags in the repository. The events of interest include "referenceCreated" and "referenceUpdated." Furthermore, it filters for references with the name prefix "release" within the "eks-apps" repository. In essence, this event pattern is finely tuned to detect changes to tagged references with a "release" prefix in the specified CodeCommit repository, enabling streamlined triggering of subsequent actions in response to these events.

  
*Image: Event Pattern for lambda trigger*

In **Select targets,**

Select **AWS service** and in **Select a target** choose lambda function and in Function choose the function which is created previously **“eks-apps-trigger"** and click on **“Next”.**  
Click on “**Next**” in **tags** section and click on “**Create rule**” to create the event rule.

  
*Image: Targets for Event rule*

  
 *Image: Event rule*

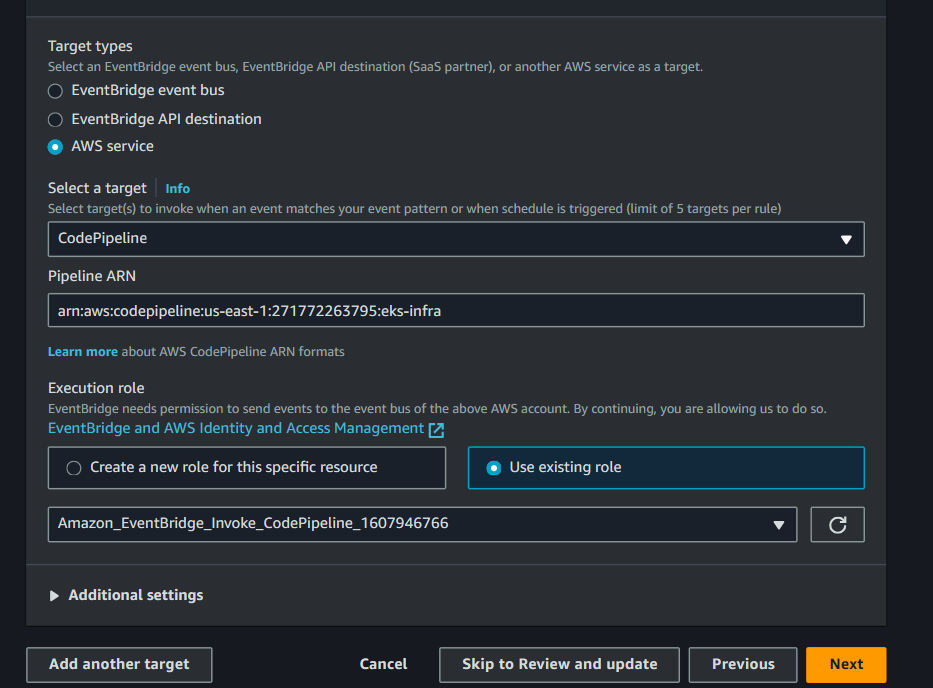
Now the event rule is ready to trigger the lambda function based on the tag pushes occurring in the “**eks-apps**” repository  
**EventBridge Rule for Infrastructure Provisioning:**   
To create eventbridge event rule click on “**Create rule**” in Amazon Eventbridge console   
specify a name to it, we have given “**eks-infra-trigger**" and click on “**Next**”. In Build event pattern scroll to last and click on “**Edit pattern**” insert the below event pattern and click on “**Next**”.

|  |
| --- |
| ***{***  ***"source": ["aws.codecommit"],***  ***"******detail-type": ["CodeCommit Repository State Change"],***  ***"resources": ["******arn:aws:codecommit:us-east-******1:271772263795:eks-infra"],***  ***"detail": {***  ***"event": ["referenceCreated", "referenceUpdated"],***  ***"repositoryName": ["eks-infra"],***  ***"referenceType": ["tag"],***  ***"referenceName": [{***  ***"prefix": "release"***  ***}]***  ***}***  ***}*** |

*Code: Tag-based Event pattern for Codepipeline trigger*

This EventBridge event pattern is tailored for monitoring changes in the state of a CodeCommit repository, specifically focusing on the "**eks-infra**" repository in the US East (N. Virginia) region. The pattern is designed to capture events related to the creation or update of references, with a specific emphasis on tags within the repository. The events of interest include "**referenceCreated**" and "**referenceUpdated**" The pattern further filters events to specifically target references with the name prefix "**release**" within the "**eks-infra**" repository. In essence, this event pattern is finely tuned to detect changes to tagged references with a "**release**" prefix in the specified CodeCommit repository, facilitating precise triggering of subsequent actions in response to these events.

In **Select targets,**

Select **AWS service** and in **Target types,** In **Select a target** choose CodePipeline and in **pipeline ARN** insert the pipeline arn of **eks-infra pipeline.** Choose **Use existing role** in **Execution role** and select **Invoke Codepipeline role** which is created previously and click on **“Next”.**  
Click on “**Next**” in **tags** section and click on “**Create rule**” to create the event rule.  
  
   
 *Image: Target for Event rule*

Now the event rule is ready to trigger the CodePipeline based on the tag pushes occurring in the “**eks-infra**” repository

This detailed technical prep-work serves as the bedrock for the subsequent implementation phases, ensuring that every technical aspect is configured and secured to support the GitLab deployment in the AWS cloud.

--------------------------------------------------------------------------------------------------------------------------------