**Runbook for Terraform**

1. **Installing Terraform**

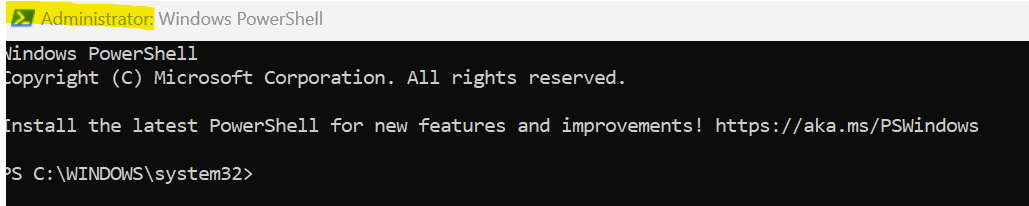
There exist two options to installing terraform. You can choose to install terraform using package managers such as **Chocolatey** for Windows, **Brew** for macOS or you can choose to install terraform using Binary files (manual process). Here’s the official link with the installation guide to terraform [Install Terraform | Terraform | HashiCorp Developer](https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli)

1. **Installation option with package Managers**

* **Windows users**

**Step1.** Install chocolatey ([Chocolatey Software | Installing Chocolatey](https://chocolatey.org/install))

* Open Powershell as an administrator (Navigate to your search menu on windows and search for **Powershell** >> Right click on **Windows Powershell** >> select **Run as administrator** >> when propted to allowthis app make changes to your computer, select **YES >>** This will open powershell as an adminas seen in the screenshot below)



* Run the following commands (Note the third command should be run as one)
* Get-ExecutionPolicy
* Set-ExecutionPolicy AllSigned
* Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))
* Wait a few seconds for the command to complete
* Type **choco** and confirm it is installed as seen in the screenshot below

Graphical user interface

Description automatically generated with medium confidence

**Step2. Install terraform by running the following commands:**

* **Choco install terraform**
* Type **terraform - -version** to confirm it is installed

**Graphical user interface, text

Description automatically generated**

* **Installation for macOS users**

**Step1. Install brew (**[Homebrew — The Missing Package Manager for macOS (or Linux)](https://brew.sh/)**)**

* Open Terminal and type the following commands:
* /bin/bash -c "$(curl -fsSL <https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh>)"
* Type your admin password for your mac laptop if prompted (note that you won’t see your keystrokes in the Terminal window — it’s a security measure)
* (echo; echo 'eval "$(/opt/homebrew/bin/brew shellenv)"') >> /Users/gen/.zprofile
* eval "$(/opt/homebrew/bin/brew shellenv)"
* Check if brew is installed by typing **brew**

**Step2. Install terraform**

* Run the following commands on your terminal
* brew tap hashicorp/tap
* brew install hashicorp/tap/terraform
* check if terraform is installed by typing **terraform -help** on your terminal

Text

Description automatically generated

1. **Manual Installation option** (Please only use this option if you did not succeed with the package manager option above)

* **Windows users**

**Step1.** Here is the link to download the binary files for the different OS [Install | Terraform | HashiCorp Developer](https://developer.hashicorp.com/terraform/downloads)

* Follow the link and choose the **Windows** option
* Under the **Binary download for Windows** click **Download** on the **386** option. This will download the binary file for terraform.
* Navigate to **file explorer** in your **Downloads** folder on your computer and you will see the zipped file you just download



* Right click on the **zipped file** and select **Extract all** and click on **extract,** copy the **terraform** file that displays after extract is complete

Graphical user interface, application

Description automatically generated

* Click This PC on your computer >> double click on drive C (c:) >> Create a folder and name it **tools >>** paste the terraform file copied earlier into this tools folder

Background pattern

Description automatically generated with low confidenceGraphical user interface, text, application

Description automatically generated Rectangle

Description automatically generated with medium confidence

* On your windows search menu, search for **Edit the System Environment Variables** and click on it when it pops up on the search menu
* Click on Environment variables at the next prompt>> double click on **Path >>** select **New >>** Paste **C:\tools** in path section (as seen in the fourth screenshot below) and click **Ok , Ok and OK**
* Then type **terraform** on your terminal to confirm terraform has been installed successfully

Graphical user interface, text, application, email

Description automatically generated Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated Graphical user interface, text, application

Description automatically generated

1. **Install AWS CLI**

**Step1. Open link below for installation guide**

[**https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html**](https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html)

**Step2. Select the OS of your personal computer**

Graphical user interface, application, Teams

Description automatically generated

1. **For macOS users**

**Step3. Select the macOS installing option and select the installation method and follow the guide**

**Graphical user interface, text, application, email

Description automatically generated**

1. **For Windows Users**

**Step3.** open the powershell terminal and run the following commands

**-** msiexec.exe /i <https://awscli.amazonaws.com/AWSCLIV2.msi>

Text

Description automatically generated

- You will see the AWS CLi installation wizard display and just follow the prompts with next until you install and finish

Graphical user interface, text, application, Word

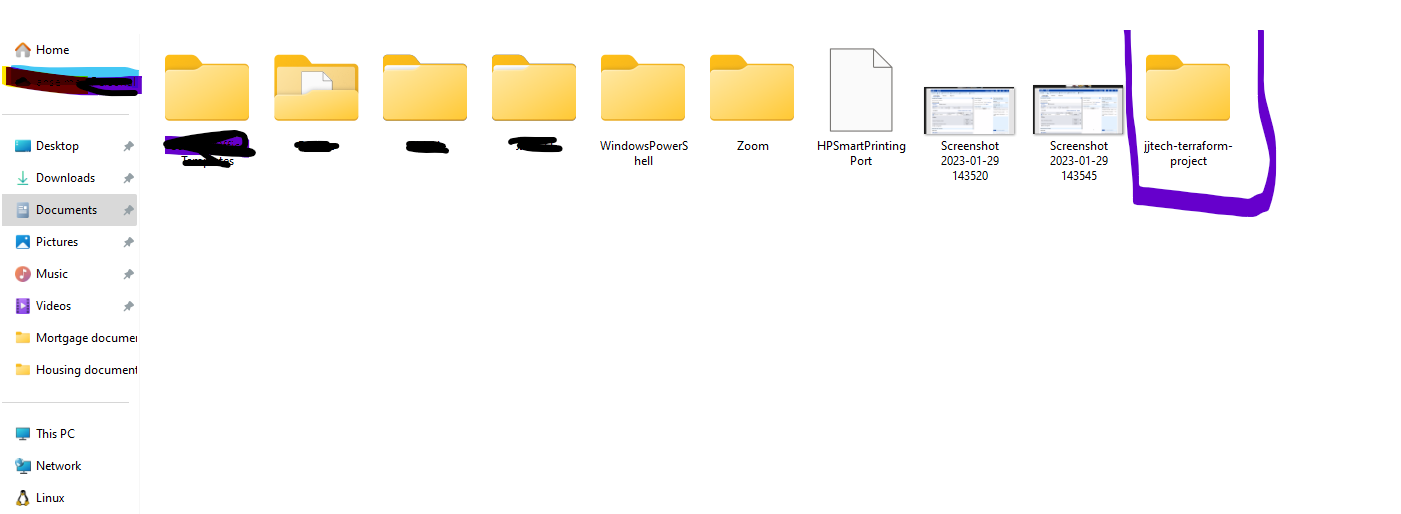
Description automatically generated

- To confirm that you have the aws cli installed, run **aws - -version** and you should see an output similar to the screenshot below.

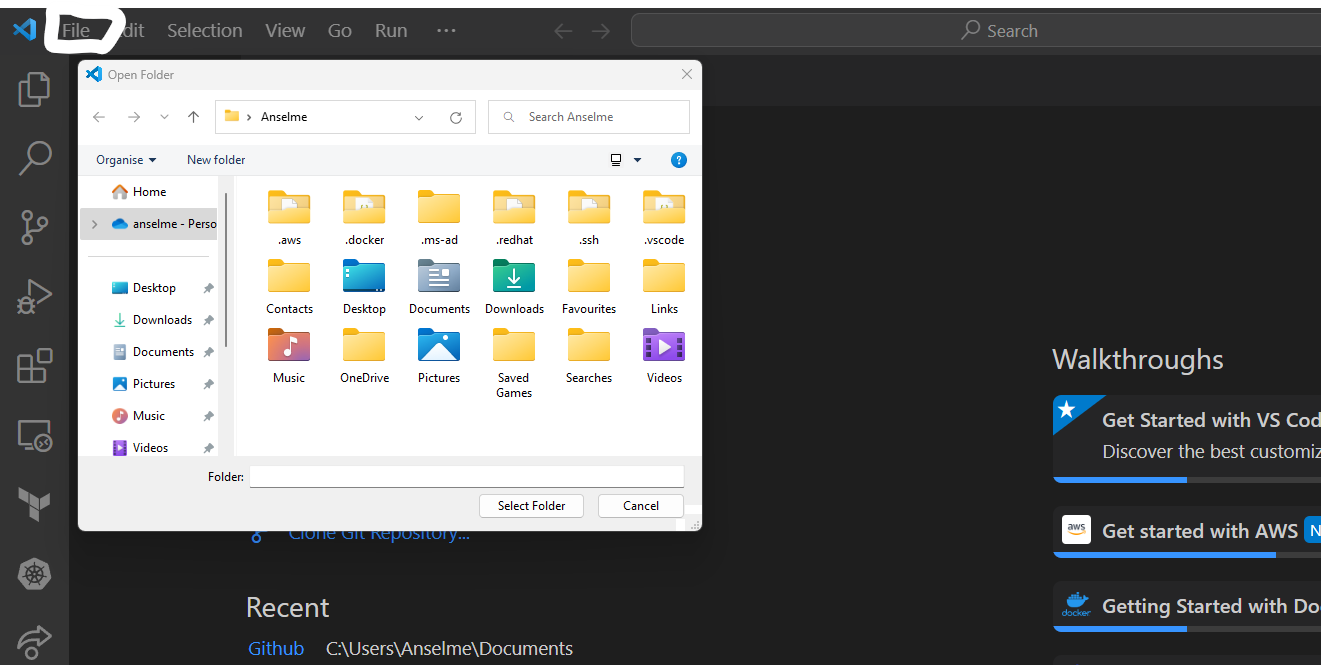
Text

Description automatically generated

1. **Integrating VS CODE with Github**
2. Navigate to windows explorer and in Documents directory, create a folder called **jjtech-terraform-projects**



1. Open VSCODE IDE, navigate to file section and click on open folder



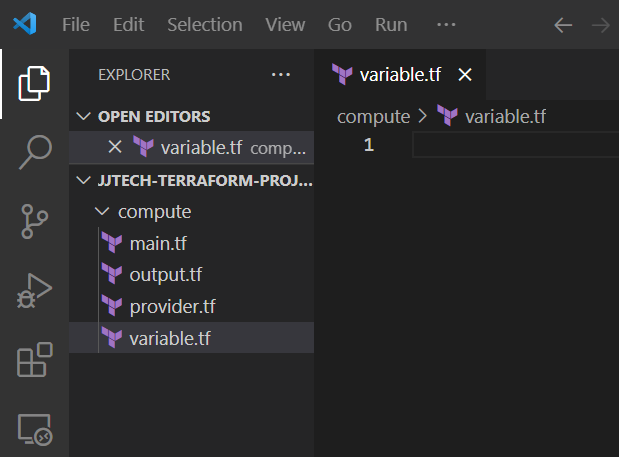
1. Open jjtech-terraform-projects created above
2. Create a Sub-folder called compute. Then create the following files:

- provider.tf

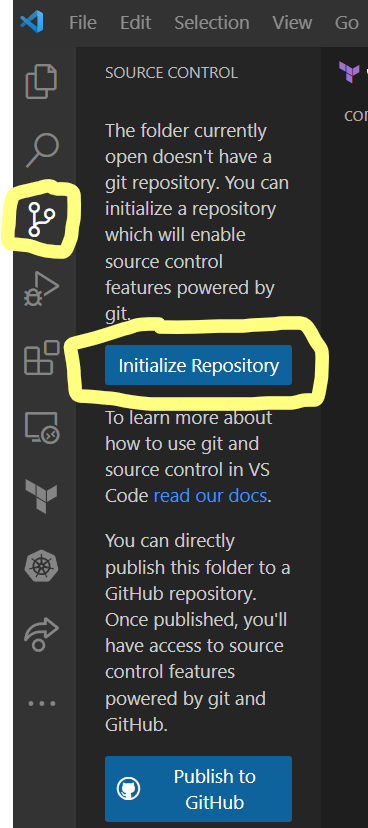
- main.tf

- variable.tf

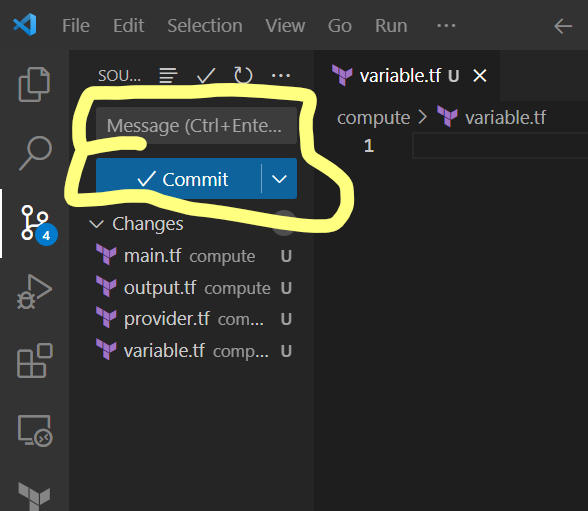
- output.tf



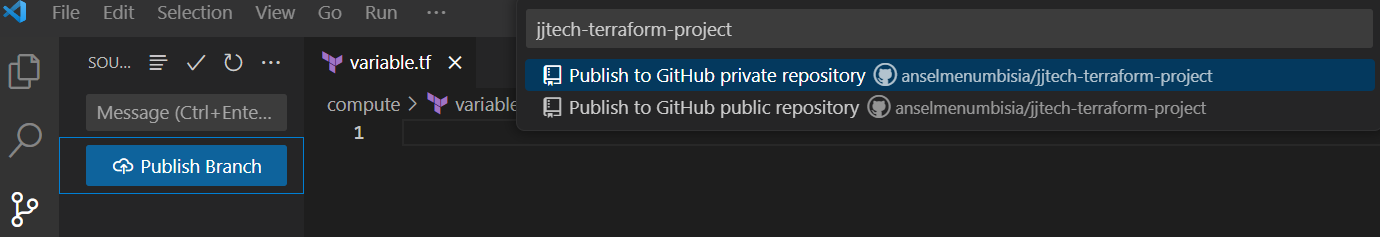
1. Click on the **source control** icon on the left section of VSCODE and click on **initialise Repository.** If prompted to login into Github**,** follow prompts for login and enter username and password.



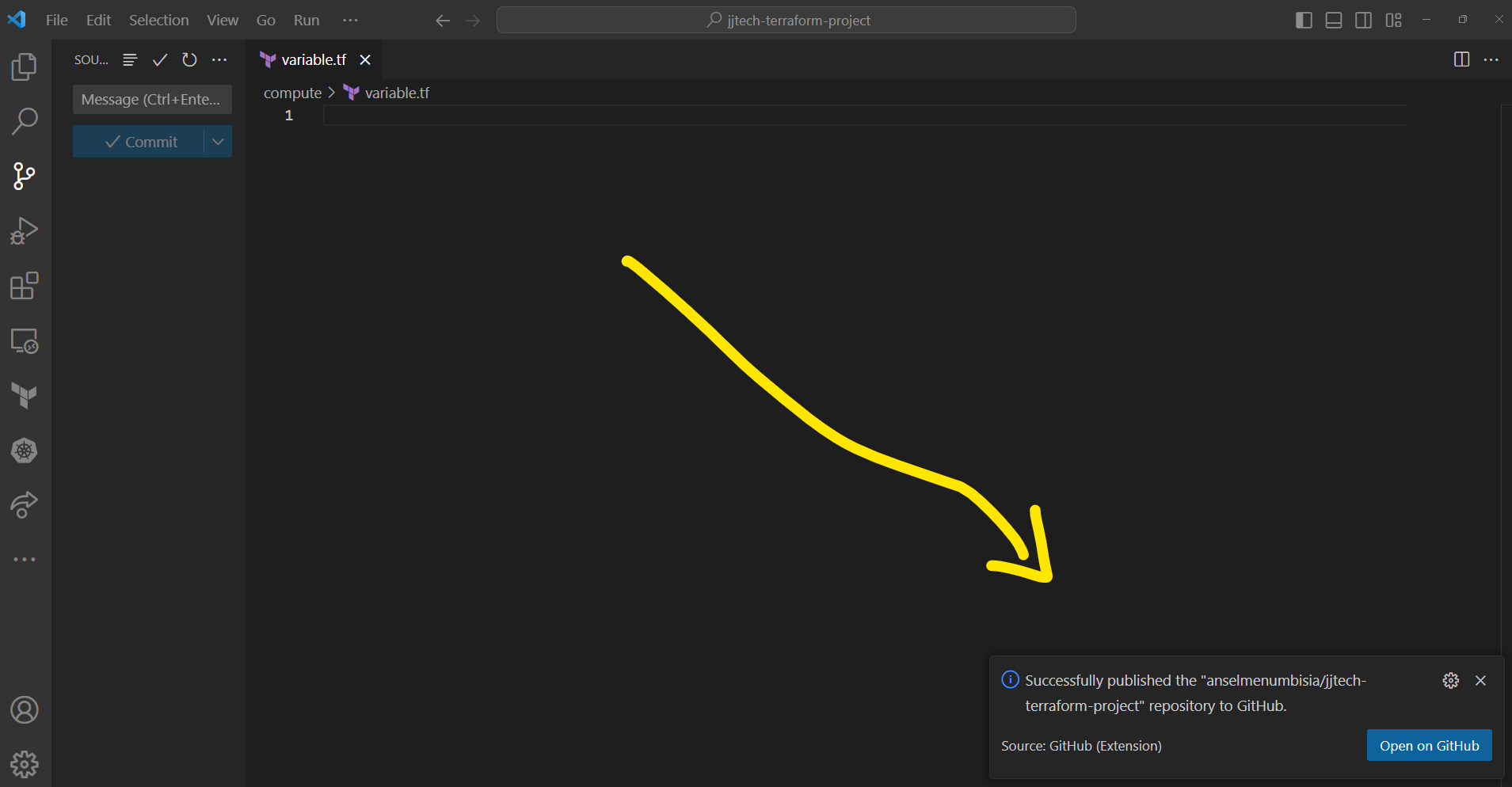
1. If repository was successfully initialised, you should see the following screen



1. Stage all changes by clicking on the + symbol next to **Changes**. (hover your cursor aroud **Changes** to view the + symbol). Once all changes are staged, fill in your first commit message and click on **Commit**.
2. Click **Publish Branch**. Pay attention to the repository name that will be created (You can modify the name). Click publish to github public repository and wait …



1. You will notice a message at the bottom right corner of your screen to confirm that the repository has been successfully published.



1. You can now open github to confirm that the repository was successfully created
2. **Runbook Terraform provisioners**

**Local-Exec Provisioners:**

In the example below, we create an EC2 instance in AWS. It makes use of a local-exec provisioner to save the private\_ip address of the instance which is created in a text file called **private\_ip.txt**, creates a folder called **Test** and moves the **private\_ip.txt** file into the folder **Test**. This provisioner executes in the same working directory where **terraform apply** is run once the provisioning is successful.

1. Create a file and name it local-exec.tf
2. Copy and paste the terraform resource block for ec2 instance below
3. Run terraform apply and check to confirm that the task was accomplished.
4. Run terraform destroy and also confirm that the second provisioner is run and that the **destruction.txt** file is created with the message **“Destruction successful”**

resource "aws\_instance" "web" {

  ami           =  var.ami

  instance\_type = var.instance\_type[2]

  key\_name = "jenkinskp"

provisioner "local-exec" {

    when = create

    command = "echo 'This is my private IP ${self.private\_ip}'>> private\_ip.txt && mkdir Test && mv private\_ip.txt Test"

}

provisioner "local-exec" {

    when = destroy

    command = "echo 'Destruction successful'>> destruction.txt"

}

}

**File Provisioners:**

In this example, we want to copy the provider.tf file existing in our terraform directory to the home directory of ec2 when the instance is created. For this, we need to configure elements for the connection block such as security group and ssh key

1. Create a file in your terraform directory and name it file-provisioner.tf
2. Copy the code below and paste
3. Navigate to your aws console and create a keypair and name it **httpkp**
4. Copy the private key file (**httpkp.pem**) that was downloaded into your terraform directory
5. Apply your code
6. Ssh into ec2 instance to confirm that the provider.tf was successfully copy in the home directory of ec2 user.

resource "aws\_security\_group" "http\_access" {

 name        = "http\_access"

 description = "Allow HTTP inbound traffic"

 ingress {

   description = "HTTP Access"

   from\_port   = 80

   to\_port     = 80

   protocol    = "tcp"

   cidr\_blocks = ["0.0.0.0/0"]

 }

 ingress {

   description = "SSH Access"

   from\_port   = 22

   to\_port     = 22

   protocol    = "tcp"

   cidr\_blocks = ["0.0.0.0/0"]

 }

 egress {

   from\_port   = 0

   to\_port     = 0

   protocol    = "-1"

   cidr\_blocks = ["0.0.0.0/0"]

 }

 tags = {

   Name = "http\_access"

 }

}

resource "aws\_instance" "web" {

  ami           =  var.ami

  instance\_type = var.instance\_type[2]

  vpc\_security\_group\_ids = [aws\_security\_group.http\_access.id]

  key\_name = "httpkp"

provisioner "file" {

    source = "./provider.tf"

    destination = "/home/ec2-user/provider.tf"

}

connection {

  host = self.public\_ip

  type = "ssh"

  user = "ec2-user"

  private\_key = file("./httpkp.pem")

}

}

**Remote-exec provisioners;**

The example below performs a simple task of installing and starting nginx on the EC2 instance that is created by Terraform. Once the EC2 instance creation is successful, Terraform’s remote-exec provisioner logs in to the instance via SSH using the connection block and executes the commands specified in the inline attribute array.

1. Create a file and name it remote-exec-provisioner.tf
2. Copy the code below and paste it into the file
3. Create a a file called shell file in your terraform directory and name it nginx.sh and paste the shell script below:

#!/bin/bash

sudo yum update -y

sudo amazon-linux-extras install nginx1 -y

sudo systemctl enable nginx

sudo systemctl start nginx

1. Navigate to your aws console and create a keypair and name it **httpkp**
2. Copy the private key file (**httpkp.pem**) that was downloaded into your terraform directory
3. Apply your terraform script
4. Navigate to your AWS management console and get the public IP address of your instance
5. Paste the IP address copied on the browser and confirm you have a welcome to Nginx messgae

resource "aws\_security\_group" "http\_access" {

 name        = "http\_access"

 description = "Allow HTTP inbound traffic"

 ingress {

   description = "HTTP Access"

   from\_port   = 80

   to\_port     = 80

   protocol    = "tcp"

   cidr\_blocks = ["0.0.0.0/0"]

 }

 ingress {

   description = "SSH Access"

   from\_port   = 22

   to\_port     = 22

   protocol    = "tcp"

   cidr\_blocks = ["0.0.0.0/0"]

 }

 egress {

   from\_port   = 0

   to\_port     = 0

   protocol    = "-1"

   cidr\_blocks = ["0.0.0.0/0"]

 }

 tags = {

   Name = "http\_access"

 }

}

resource "aws\_instance" "web" {

  ami           =  var.ami

  vpc\_security\_group\_ids = [aws\_security\_group.http\_access.id]

  instance\_type = var.instance\_type[2]

  key\_name = "httpkp"

provisioner "file" {

    source = "./nginx.sh"

    destination = "/home/ec2-user/nginx.sh"

}

provisioner "remote-exec" {

    #script = "./nginx.sh"

    inline = [

      "chmod 777 ./nginx.sh",

      "./nginx.sh"

    ]

}

connection {

  host = self.public\_ip

  type = "ssh"

  user = "ec2-user"

  private\_key = file("./httpkp.pem")

}

}

1. **Deploying Terraform script using Gitlab CI/CD**

**Overview:**

In this tutorial, we will integrate **Terraform** with **GitLab CI/CD** and create various resources on **AWS.**

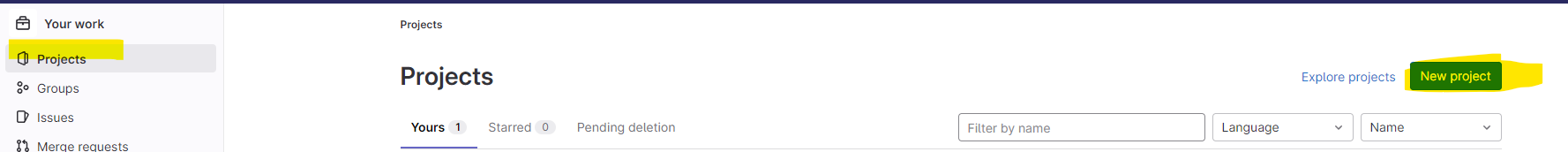
**Prerequisite:**

* **AWS & GitLab Account**
* Basic understanding of [**AWS**](https://docs.aws.amazon.com/), [**Terraform**](https://www.terraform.io/intro/index.html)**&**[**GitLab CI/CD**](https://docs.gitlab.com/ee/ci/)
* An **access key** & **secret key** created in the **AWS**

Lets, start with the configuration of the project:

Step1. Create a gitlab project

* Sign in to gitlab account. **Click on Create New Project, Create blank project** and fill out the information as required.



Graphical user interface, text, application, email

Description automatically generated

Step2. Create your terraform configuration file for resource creation. Get the sample code from [here](https://gitlab.com/anselmenumbisia/tf-pipeline-project)

- Download the source code from the repository

**Graphical user interface, text, application, email

Description automatically generated**

- Next add a remote repository pointing to your own gitlab repository by running the command

- git init

- git remote add <remote name> <url> of your repository

- git status

- git add .

- git commit -m “initial pipeline commit”

- git push <remote name> master

- When prompted for credentials, select password, then enter username and password

Step3. Create Environmental variables in Gitlab to store your AWS credentials

* In order to create the resources in the AWS account, we must need to have the **AWS Access Key** & **AWS Secret Key**
* Now, we need to store the **AWS Access Key** & **AWS Secret Key** in the secrets section of the repository
* Go to**settings**-> **CI/CD** -> **Variables** and click on **Expand** Under the variable section create the below variables and store your **AWS\_ACCESS\_KEY\_ID** & **AWS\_SECRET\_ACCESS\_KEY. To easily get these values from your cli, run the command notepad ~/.aws/credentials from your cli.**

**Table

Description automatically generated**

Table

Description automatically generated

**Step 4:-** Create a **workflow** file

* Now in order to create the terraform resources automatically, we need to create a workflow file
* Create **.gitlab-ci.yml** file and add the below code to it
* The below job will run on every **push** and **pull request** that happens on the **main** branch. In the build section, I have specified the image name and commands in the script section.

image:

  name: hashicorp/terraform

  entrypoint: [""]

variables:

  AWS\_DEFAULT\_REGION: ${AWS\_REGION}

  AWS\_ACCESS\_KEY\_ID: ${AWS\_ACCESS\_KEY\_ID}

  AWS\_SECRET\_ACCESS\_KEY : ${AWS\_SECRET\_ACCESS\_KEY}

before\_script:

  - rm -rf .terraform

  - terraform --version

  - terraform init -reconfigure

stages:

  - format

  - validate

  - plan

  - apply

  - destroy

format:

  stage: format

  script:

    - terraform fmt

validate:

  stage: validate

  script:

    - terraform validate

  dependencies:

    - format

plan:

  stage: plan

  script:

    - terraform plan -out "planfile"

  artifacts:

    paths:

      - planfile

  dependencies:

    - validate

apply:

  stage: apply

  allow\_failure: true

  script:

    - terraform apply -auto-approve -input=false "planfile"

  dependencies:

    - plan

  when: manual

destroy:

  stage: destroy

  script:

    - terraform destroy --auto-approve

  dependencies:

    - apply

  when: manual

Step5. Push your source code to gitlab and navigate to CI/CD >> Pipeline and you should see a running pipeline job with the different stages.

Step6. If prompted to validate account with credit card, proceed to clicking on **Validate Account**

**Step6.** The default pipeline runs from the main branch. To modify this

- click on **Build** >> **Pipeline** >> and click **Run Pipeline** on far right end

Graphical user interface, application

Description automatically generated with medium confidence

- on the **Run Pipeline page**, change the branch from **main** to **master** and click **Run pipeline.** This should trigger a new pipeline job

Graphical user interface, text, application, email

Description automatically generated

- You should now see the pipeline running

Graphical user interface, text, application, email

Description automatically generated

- You can manually validate the apply stage and after resources create, you can then run the destroy stage as well manually.

1. **Deploying AWS Resources using Terraform and Jenkins Pipeline**

**Overview:**

## Jenkins Pipeline

Jenkins is a self-contained, open source automation server used to automate tasks associated with building, testing, and delivering/deploying software. Jenkins Pipeline implements continuous deliver pipelines into Jenkins through use of plugins and a Jenkinsfile. The Jenkinsfile can be Declarative or Scripted and contains a list of steps for the pipeline to follow.

## **Prerequisites**

* Gitlab Account
* [AWS CLI](https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html)
* Install [Terraform](https://learn.hashicorp.com/tutorials/terraform/install-cli)
* [AWS Account](https://aws.amazon.com/premiumsupport/knowledge-center/create-and-activate-aws-account/)
* AWS user with Admin permissions
* Preferred IDE (I used VSCode)

**Getting started**

1. Install Jenkins

* Create an **Amazon Linux 2 VM** instance and call it "Jenkins"
* Instance type: t2.micro
* Security Group (Open): 8080 and 22 to 0.0.0.0/0
* Key pair: Select or create a new keypair
* **Attach Jenkins server with IAM role having "AdministratorAccess"**
* User data (Copy the following user data): <https://github.com/cvamsikrishna11/devops-fully-automated/blob/installations/jenkins-maven-ansible-setup.sh>
* Launch Instance
* After launching this Jenkins server, attach a tag as **Key=Application, value=Jenkins**
* **Copy the public IP of your Jenkins server and run with on a browser and add :8080 example x.x.x.x:8080**
* **When prompted for the password, Ssh into your Jenkins server and run the command sudo cat** /var/lib/jenkins/secrets/initialAdminPassword Get the password and paste in required box

Graphical user interface, text, application, email

Description automatically generated

* **Click on install suggested plugins**
* **Fill out the form to create first time admin user and follow the prompts at the bottom right corner to access Jenkins.**
* Graphical user interface, application

  Description automatically generated

Graphical user interface, application, Teams

Description automatically generated

1. Install/configure terraform and gitlab plugins in Jenkins
   1. Click **Manage Jenkins** from left hand navigation.

Graphical user interface, application

Description automatically generated

b. Select **Manage Plugins** from **System Configuration** section.

Graphical user interface, application

Description automatically generated

1. Click the**Available** tab and search **Terraform and then gitlab**

Graphical user interface, text

Description automatically generated

1. Select **Terraform** and then **Gitlab** and click **Install without restart**.
2. Restart Jenkins by running <Jenkinsurl:8080/restart>, when prompted to restart, click **YES**

Graphical user interface, text, application, email

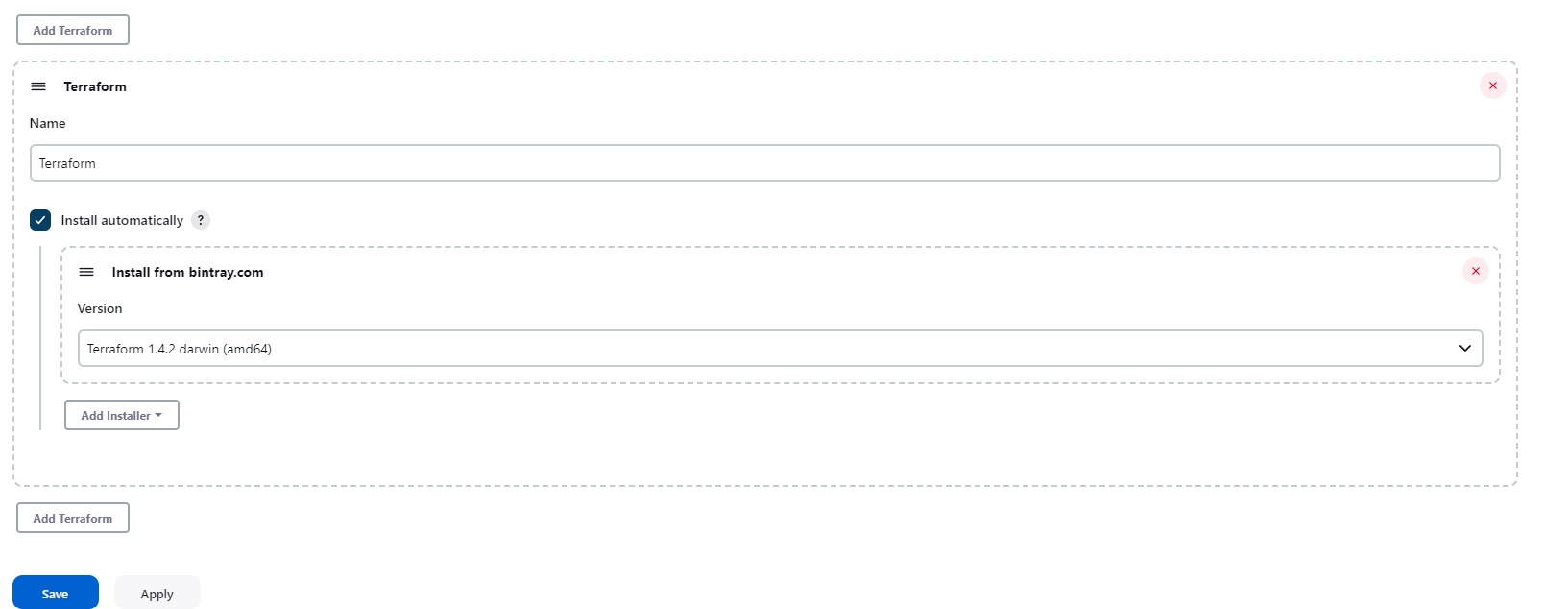
Description automatically generated

1. Click **Manage Jenkins** from left hand navigation.
2. Click  **Tool** from **System configuration** section

Graphical user interface, text, application

Description automatically generated

1. Scroll down to the **Terraform** section and click **Add Terraform**.
2. Enter a **Name** of your choice. I’m going to use “Terraform” to make things simple. Ensure **install automatically** is checked. Save and apply.



1. Create Gitlab token >> Navigate to Gitlab >> Click on profile logo >> the Access token >> fill name, expiration >> select a role (guest)>> select al scopes>>create project access token>>Copy token and save in secure location.

A screenshot of a computer

Description automatically generated

1. For gitlab configuration, Navigate to **Manage Jenkins** >> then **Systems** and lookout for **Gitlab >>** provide a name for the connection**,** for **Gitlab host URL** use[**https://gitlab.com**](https://gitlab.com/) **>>** for **Credentials**, click on **Add** and then **Jenkins**. For **kind** select **Gitlab API Token >>** provide the gitlabtoken **>>** for **ID** enter any name **e.g gitlab-creds >>** add the credential by clicking on the **add** section **>> Select credential from none to credential created >>** click on **test connection** and ensure to have a **success message** >> Once complete, **save** and then **apply**

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

1. Manage Credentials on Jenkins (Only add these AWS creds if you have not attached an IAM role with admin access to the Jenkins server)
2. Click **Manage Jenkins**.
3. Click **Manage Credentials** in the **Security** section.

Text

Description automatically generated

1. Click on **systems** >> [Global credentials (unrestricted)](http://18.118.155.107:8080/manage/credentials/store/system/domain/_) >> and add credentials on the top right corner

Graphical user interface, text, application

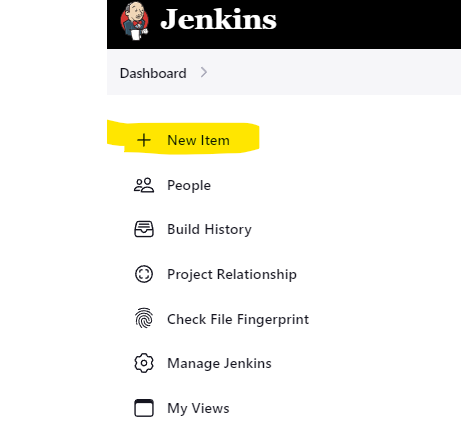
Description automatically generated

1. For **Kind** select **Secret text**. For **ID** type “**AWS\_ACCESS\_KEY\_ID**”. For **Secret** paste your **Access Key** for your user. Then click **OK**

Graphical user interface, text, application, email

Description automatically generated

1. Repeat previous step for your “**AWS\_SECRET\_ACCESS\_KEY**”
2. Create pipeline job
   1. Navigate back to the Jenkins dashboard and click on **New Item**



* 1. Enter an item name (name of the pipeline project you want to create) >> choose **Pipeline** in the job fields and click **OK.**

Graphical user interface, text, application, chat or text message

Description automatically generated

* 1. Navigate down to **Pipeline and** click on **Pipeline script** and select **Pipeline script from SCM**

Graphical user interface, text, application, email

Description automatically generated

* 1. Enter the following

- **SCM:** Git

- **Repository URL:** Your gitlab repo where you have the jenkinsfile (same as link from clone in gitlab)

- **Branch:** your primary branch e.g master

- **credentials:** select your gitlab credential created in the previous step. This is not required if your repo is public

- **Repository browser:** Auto

- **Script Path:** “jenkinsfile”

- click **save**

Text, application

Description automatically generated with medium confidence

Graphical user interface, text, application, email

Description automatically generated

1. **Run Jenkins Pipeline**
2. Select **Build with Parameters** from the left navigation.

Graphical user interface, application

Description automatically generated

1. For the **environment** parameter type the name you want to use for your Workspace. The default is “**development**”. For the **region**, fill in the name of the region where you want to deploy the resources. Click on **Build**

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, chat or text message

Description automatically generated

1. Now you should see the steps of the pipeline begin and the time it takes to complete each stage. The pipeline will pause on after the **Plan** step and prompt for a manual approval to proceed. Click on proceed to continue the pipeline.

Graphical user interface, application, table

Description automatically generated

1. Pipeline apply phase is now complete . Navigate to aws console to confirm resource creation



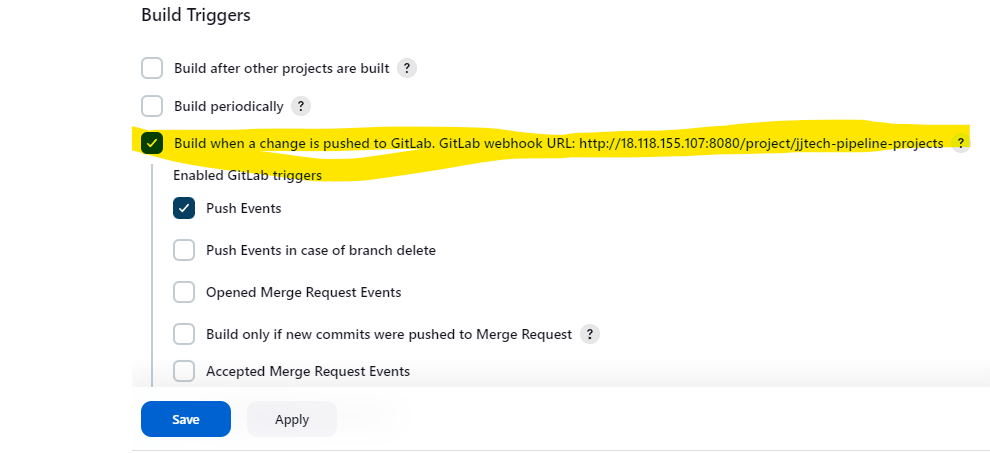
1. Proceed to validate the pipeline to destroy the resources provisioned

Graphical user interface, application, table

Description automatically generated

1. Verify that the resources have been destroyed.
2. **Trigger build based on push event on the gitlab repo. Integration tutorial** [**here**](https://hevodata.com/learn/gitlab-webhook-jenkins-integration/#:~:text=Step%201%3A%20Go%20the%20the,the%20%E2%80%9CSecret%20Token%E2%80%9D%20field.)
3. **In Jenkins**

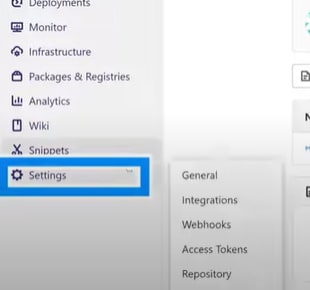
* **Step 1:**Go the the “Dashboard >> **Click on the Jenkins project >> click on configure**” of your Jenkins project.
* **Step 2:**Go to the “***Build Triggers***” section.
* **Step 3:**Under the “***Build when a change is pushed to Gitlab***” checkbox, click the “***advanced***” button.



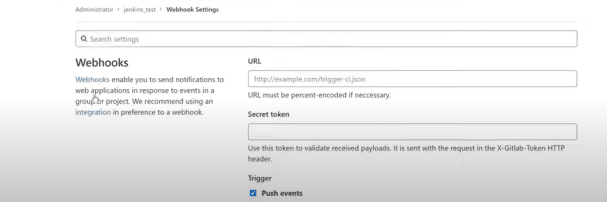
* **Step 4:**Scroll down andClick the “***Generate***” button under the “***Secret Token***” field.
* **Step 5:**Copy the resulting token, and **save** the job configuration.

#### Webhook Integration

* **Step 1: Navigate to gitlab.** In the left navigation pane, select the “***Settings***” option. Then click on the “***Webhooks***” option.



* **Step 2:**Now, in the Integration settings window, under the “***Integrations***” section, select the “***Webhook***” hyperlink.
* **Step 3:**In the “***Webhook Settings***” window, under the “***Webhooks***” section, paste the webhook URL (such as ***https://JENKINS\_URL/project/YOUR\_JOB***) copied from Jenkins server.



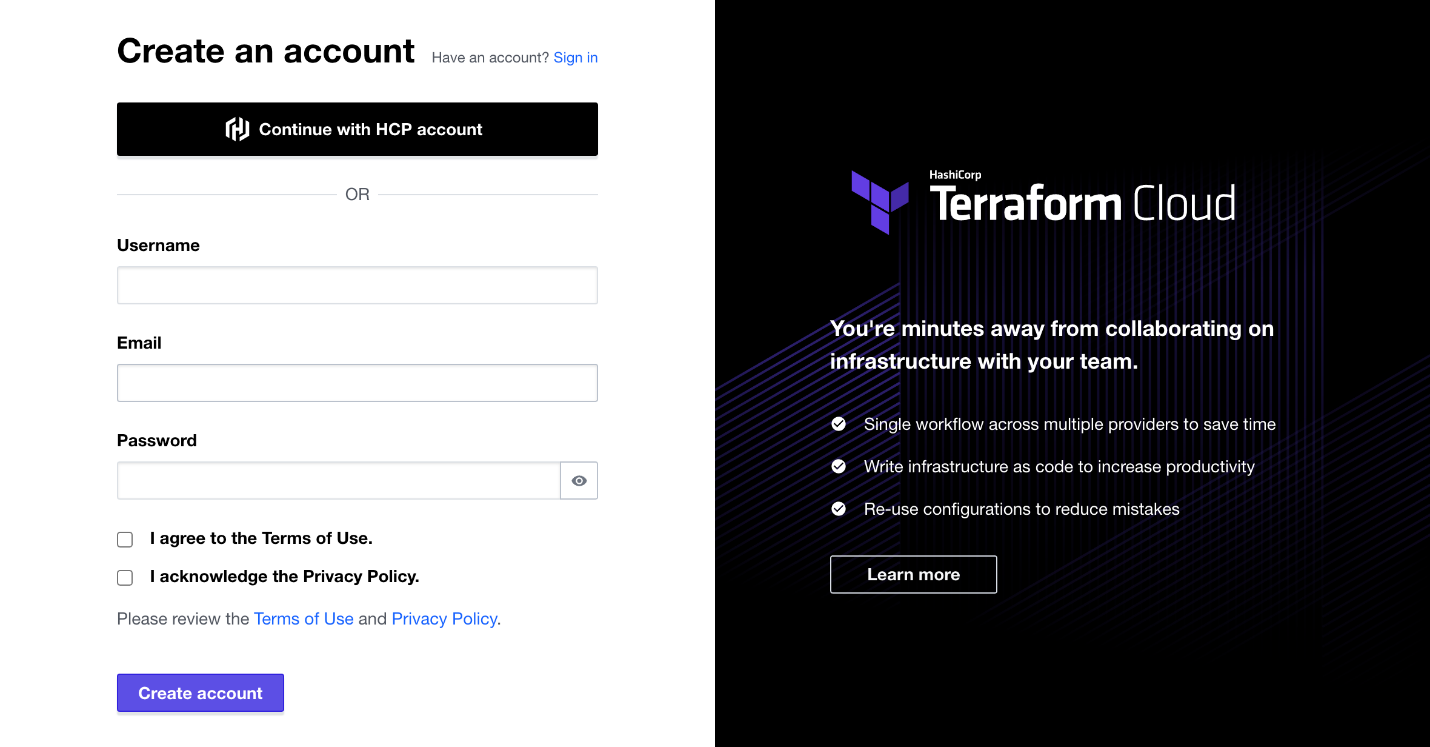
* **Step 4:** Paste the secret token which you have generated in the Jenkins server and check the **Push events** and click **Add webhook** at the bottom.
* **Step 5:** Scroll down to the webhook created and click on **Test** connection. Select **push events** and you should see **Hook executed successfully: HTTP 200**

1. Make changes to your source code, then push to the gitlab repo.
2. Navigate back to Jenkins and notice that a pipeline job is triggered.
3. **Terraform Cloud**

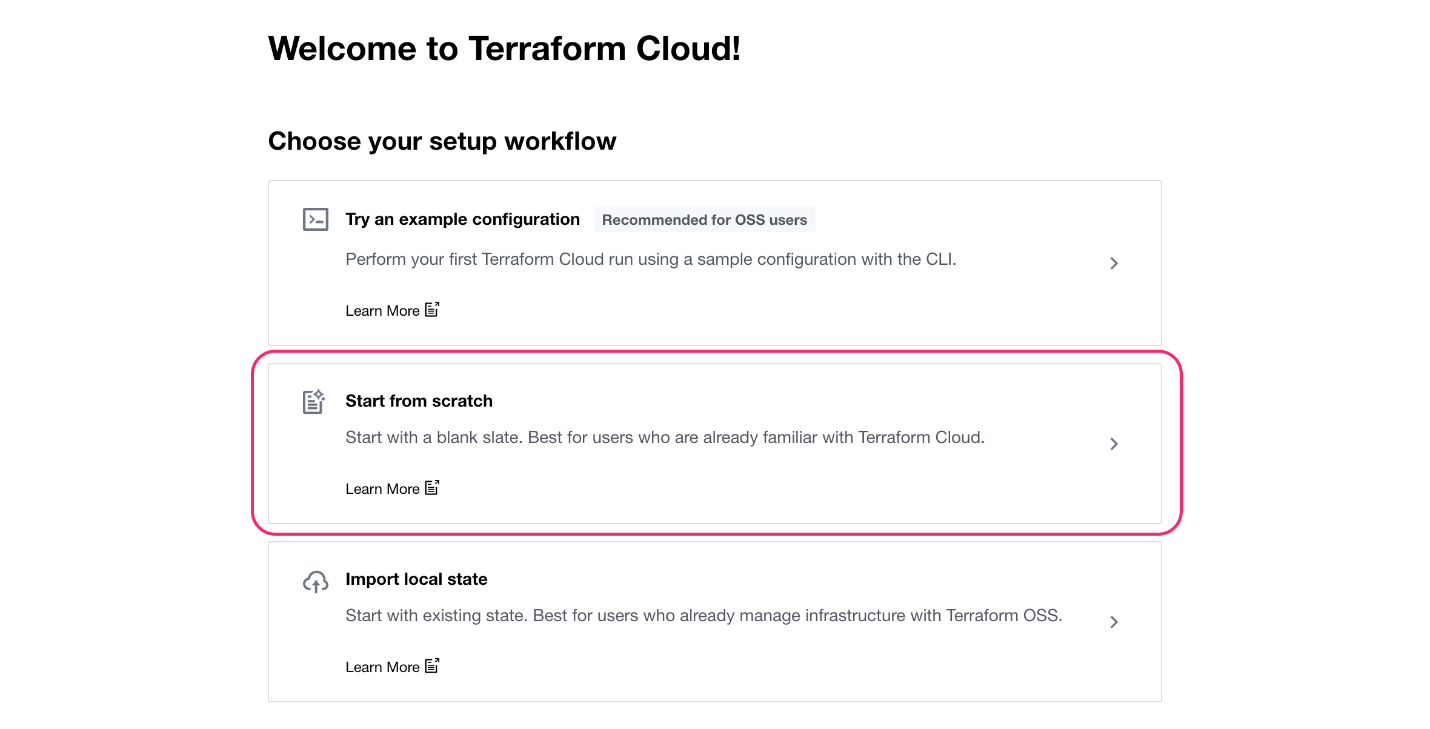
**CLI-Driven Workflow**

1. **Create Terraform Cloud account**

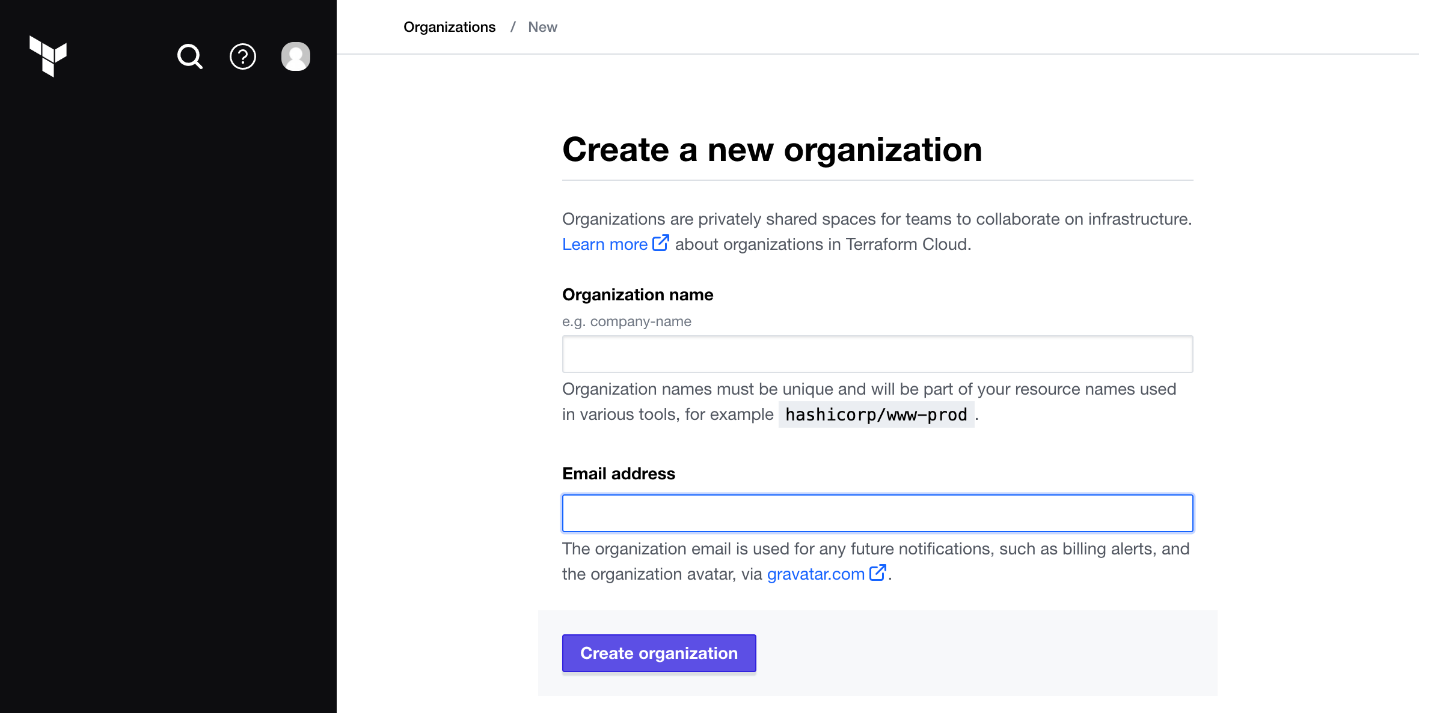
**-** Visit [https://app.terraform.io/signup/account](https://app.terraform.io/public/signup/account?utm_source=learn) and follow the prompts to create a free Terraform Cloud account.



- **step2**. When you sign up, you will receive an email asking you to confirm your email address. Confirm your email address before moving on. When you click the link to confirm your email address, the Terraform Cloud UI will ask which setup workflow you would like to use. Select **Start from scratch**.



**Step3.** create a new organization. Creating organizations of up to 5 users is free, and the members you add to the organization will be able to collaborate on your workspaces and share private modules and providers.



1. **Log in to Terraform Cloud from the CLI**

Terraform Cloud runs Terraform operations and stores state remotely, so you can use Terraform without worrying about the stability of your local machine, or the security of your state file.

To use Terraform Cloud from the command line, you must log in. Logging in allows you to trigger remote plans and runs, migrate state to the cloud, and perform other remote operations on configurations with Terraform Cloud

**Step1.** run the **terraform login** subcommand. Respond **yes** to the prompt to confirm that you want to authenticate. A browser window will automatically open to the Terraform Cloud login screen. Enter a token name in the web UI, or leave the default name, terraform login.

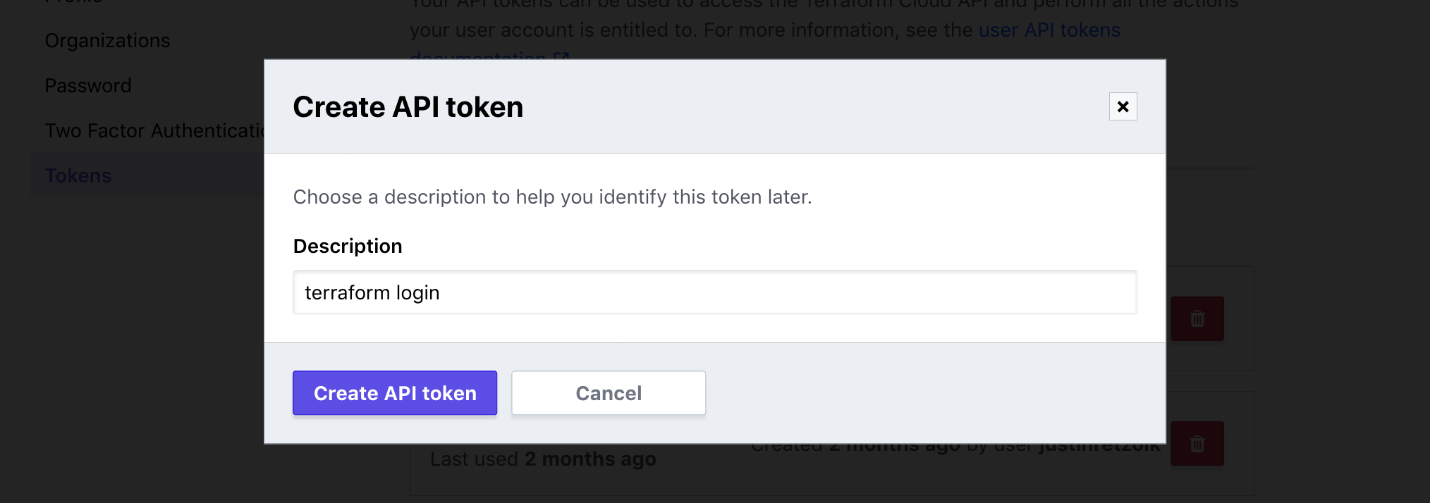
Text

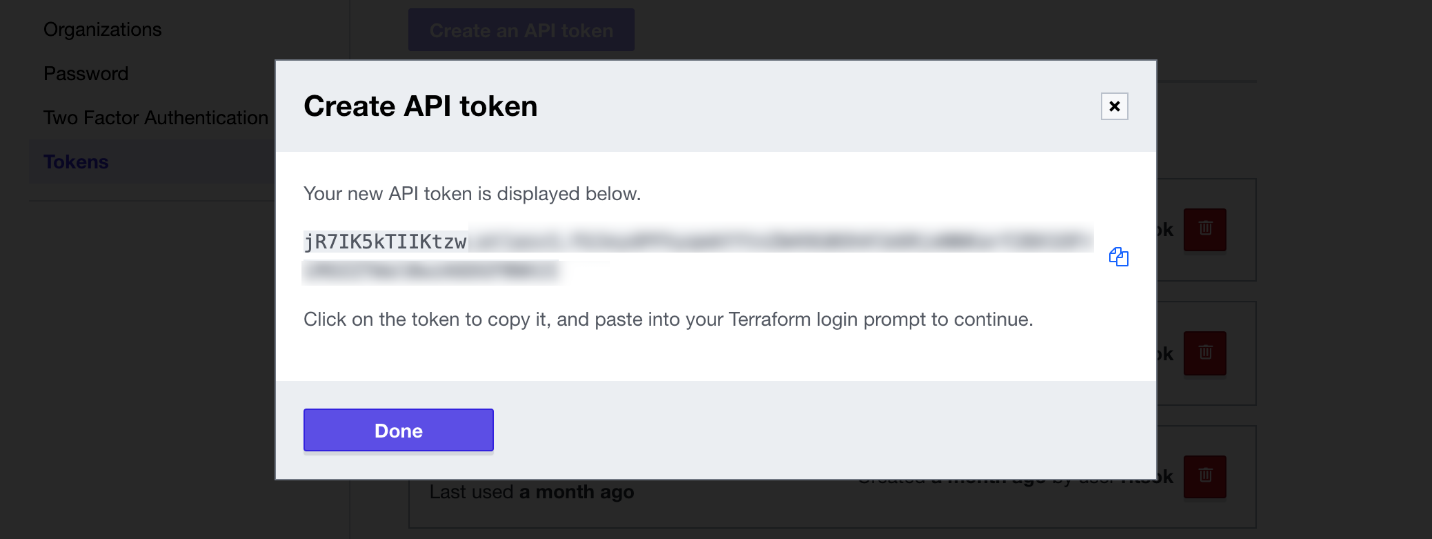
Description automatically generated

Graphical user interface, application

Description automatically generated

**Step2**. Click **Create API token** to generate the authentication token. Save a copy of the token in a secure location. It provides access to your Terraform Cloud organization. Terraform will also store your token locally at the file path specified in the command output





**Step3.** When the Terraform CLI prompts you, paste the user token exactly once into your terminal. Terraform will hide the token for security when you paste it into your terminal. Press **Enter** to complete the authentication process.

Text

Description automatically generated

1. **Creating variable sets**

Variable sets allow you to avoid redefining the same variables across workspaces, so you can standardize common configurations throughout your organization. One common use case for variable sets is for provider credentials. By defining a variable set for your credentials, you can easily reuse the same variables across multiple workspaces and efficiently and securely rotate your credentials. We will create a variable set for our AWS credentials.

**Step1**. Navigate to terraform cloud >> click on settings >> Variable sets

Graphical user interface, text, application, email

Description automatically generated

**Step2.** Click **Create Variable sets**

**- Name: (provide name)**

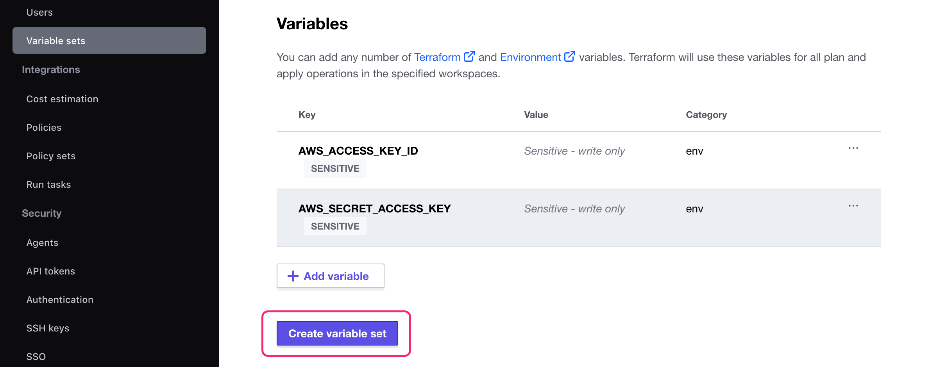
**- Description:**

**- Workspaces: Apply to all workspaces in this organisation**

**Step3.** Click **+Add Variable**. Select the **Environment variable** option. Set the key to AWS\_ACCESS\_KEY\_ID and the value to your AWS **Access Key ID**. Mark it as **Sensitive** and click **Add variable**.

Step4. Click **+ Add Variable** again. Define another environment variable named AWS\_SECRET\_ACCESS\_KEY and set it to your AWS **Secret access key**. Mark it as **Sensitive** and click **Add variable**.

**Step4.** Click **create Variable sets**



1. **Create Workspace**

**Step1.** Navigate to vscode where you have your configuration file andconfigure **provider.tf** file to add configuration for **terraform cloud.** This cloud block specifies which Terraform Cloud organization and workspace to use for the operations in this working directory. When using the CLI-driven Terraform Cloud workflow, running terraform init on configuration with a cloud block creates the Terraform Cloud workspace specified in the block, if it does not already exist.

terraform {

  cloud {

    organization = "Provide name of TFC Organisation"

    workspaces {

      name = "pass name of a workspace to plan to create"

    }

  }

  required\_version = ">= 1.1.0"

  required\_providers {

    aws = {

      source  = "hashicorp/aws"

      version = "4.55.0"

    }

  }

}

**Step2.** Run **terraform init**

**Text

Description automatically generated**

As part of the initialization process, Terraform created the new **jjtech-workspace1** workspace in our Terraform Cloud organization, configured for CLI-driven runs.

**A screenshot of a computer

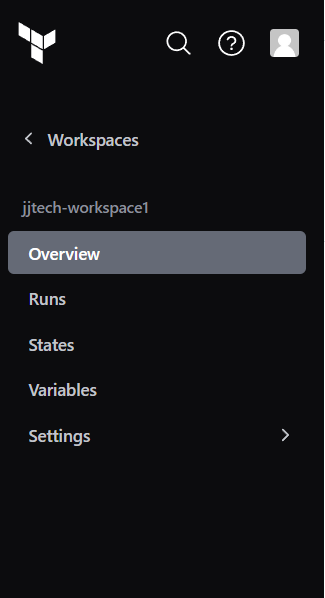
Description automatically generated**

1. **Create Infrastructure**

You now have a Terraform Cloud workspace configured to use AWS credentials defined in a Terraform Cloud variable set. You can further configure your workspace using workspace-specific variables.

**Step1.** Navigate to your TFC and create a variable for your instance type and other variables as passed in your cli configuration file.

- Navigate to TFC homepage >> Projects and Workspaces >> select Workspace >> variables >> **Add variable >>** select **Terraform variable** >> add **Key and Value** for the variable and **Add variable**.

**** **Graphical user interface, text, application, email

Description automatically generated**

**Step2.** Navigate back to visual studio code and on your terminal, run terraform apply to provision the infrastructure. Terraform will apply the run on TFC.

**Text

Description automatically generated**

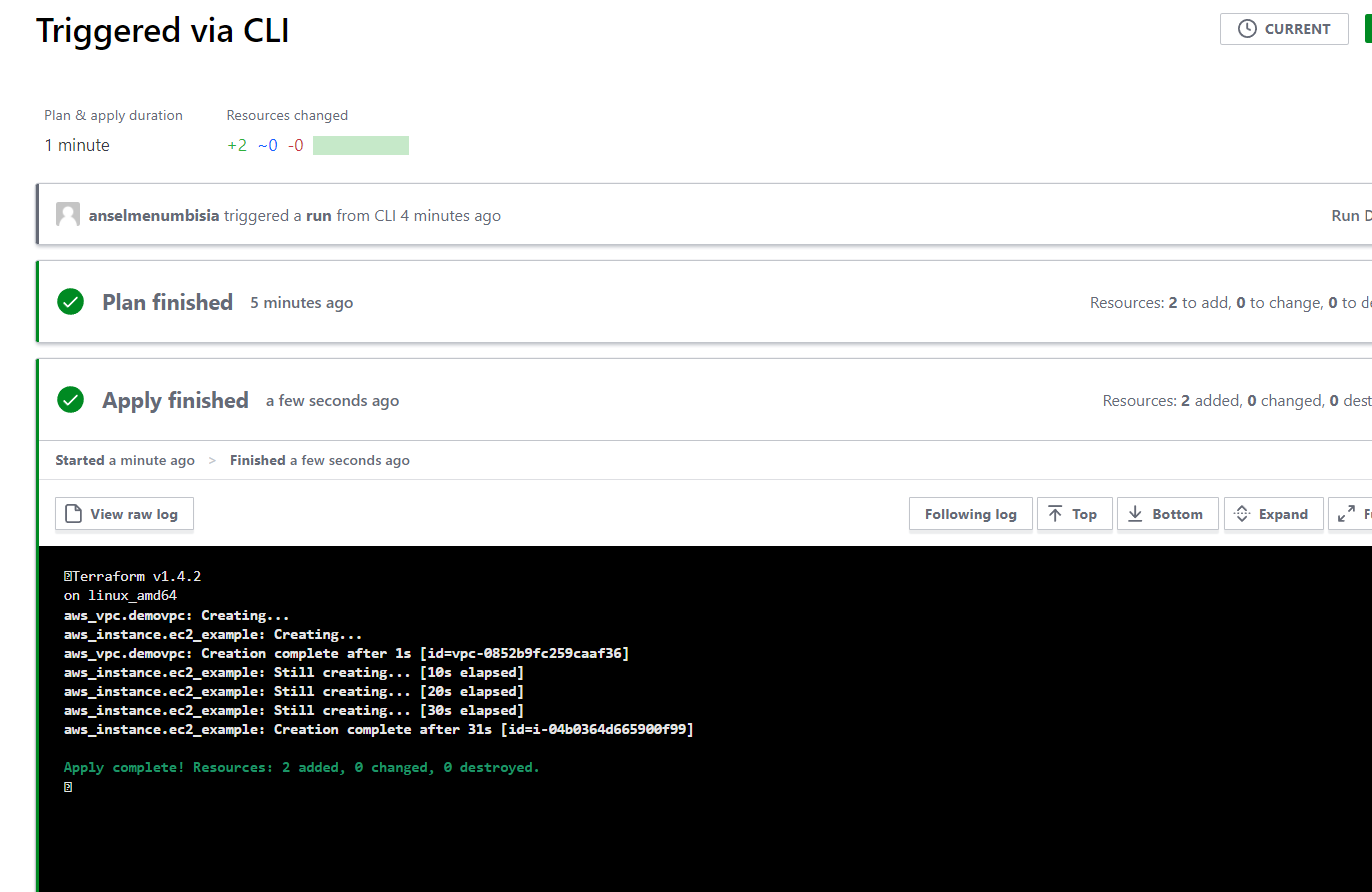
**Step3.** Terraform will trigger your run in Terraform Cloud and stream the output to your terminal. Alternatively, you can follow and manage the run in the Terraform Cloud UI.

Navigate to the run URL that Terraform displays in your command output.

Graphical user interface

Description automatically generated

**Step4.** **Confirm** and **apply** to trigger the apply from TFC.



**Step5.** Navigate back to your cli and run **terraform destroy** and follow **from step 2** to destroy the resources created.

**VCS driven Workflow**

In addition to the CLI-driven workflow, Terraform Cloud offers a Version Control (VCS)-driven workflow that automatically triggers runs based on changes to your VCS repositories. The CLI-driven workflow allows you to quickly iterate on your configuration and work locally, while the VCS-driven workflow enables collaboration within teams by establishing your shared repositories as the source of truth for infrastructure configuration.

You will configure a VCS integration for your organization, connect your workspace to a VCS repository, and trigger a speculative plan based on a pull request. Then, you will merge the pull request to automatically apply changes to your infrastructure using Terraform Cloud.

* 1. **Create a new Gitlab Project**
* Navigate to Gitlab and create a new project >> copy the URL of the project
* Navigate to your vscode and switch to the directory where you have your terraform source code
* Run **git init** to initialise the directory
* Run **git remote add origin YOUR\_REMOTE URL** copied from above
* Ensure you do not have a **terraform cloud** block in your terraform file (provider.tf), if yes, comment it out. When using the VCS-driven workflow for Terraform Cloud, you do not need to define the cloud block in your configuration.

terraform {

  # cloud {

  #   organization = "jjtech-learn"

  #   workspaces {

  #     name = "jjtech-workspace1"

  #   }

  # }

  # required\_version = ">= 1.1.0"

  required\_providers {

    aws = {

      source  = "hashicorp/aws"

      version = "4.55.0"

    }

  }

}

* Now add your changes with **git add .**
* Commit changes with **git commit -m “message here”**
* Push configuration to new project repo **git push** **origin master**
  1. **Enable VCS integration**
* Navigate to your terraform cloud account
* Create a new workspace and select **Version Control Workflow**

Graphical user interface, text, application, email

Description automatically generated

* Under connect to a version control provider, click on **connect to a different VCS** as the only option displayed is **github**

Graphical user interface, text, application, email

Description automatically generated

* On the dropdown for gitlab, select gitlab.com

Graphical user interface, application

Description automatically generated

* Follow the instructions in the next prompt from 1 and 2.

Graphical user interface, text, application, email

Description automatically generated

* After filling out the required information, Click on **connect and continue**
* When prompted to authorise Terraform cloud access to gitlab**,** select **Authorize**

Graphical user interface, text, application

Description automatically generated

* On the next page for SSH, scroll down and click on **skip and finish** as we do not need to configure ssh access. You can follow instructions from this page to set up ssh keys
* After configuring your provider, navigate back to workspaces in terraform cloud and create a new workspace by clicking on **New**, then **Workspace**

Graphical user interface, text, application

Description automatically generated

* In the next step select **Version control workflow >>** under **connect to a version control provider,** selec**t gitlab**

Graphical user interface, text, application, email, Teams

Description automatically generated

* Choose the repository where your terraform project is found

Graphical user interface, application, Teams

Description automatically generated

* Provide a Workspace name
* Select **default Project** in **Project section**
* Provide a description

Graphical user interface, text, application, Teams

Description automatically generated

* Then create workspace.

Graphical user interface, application, Teams

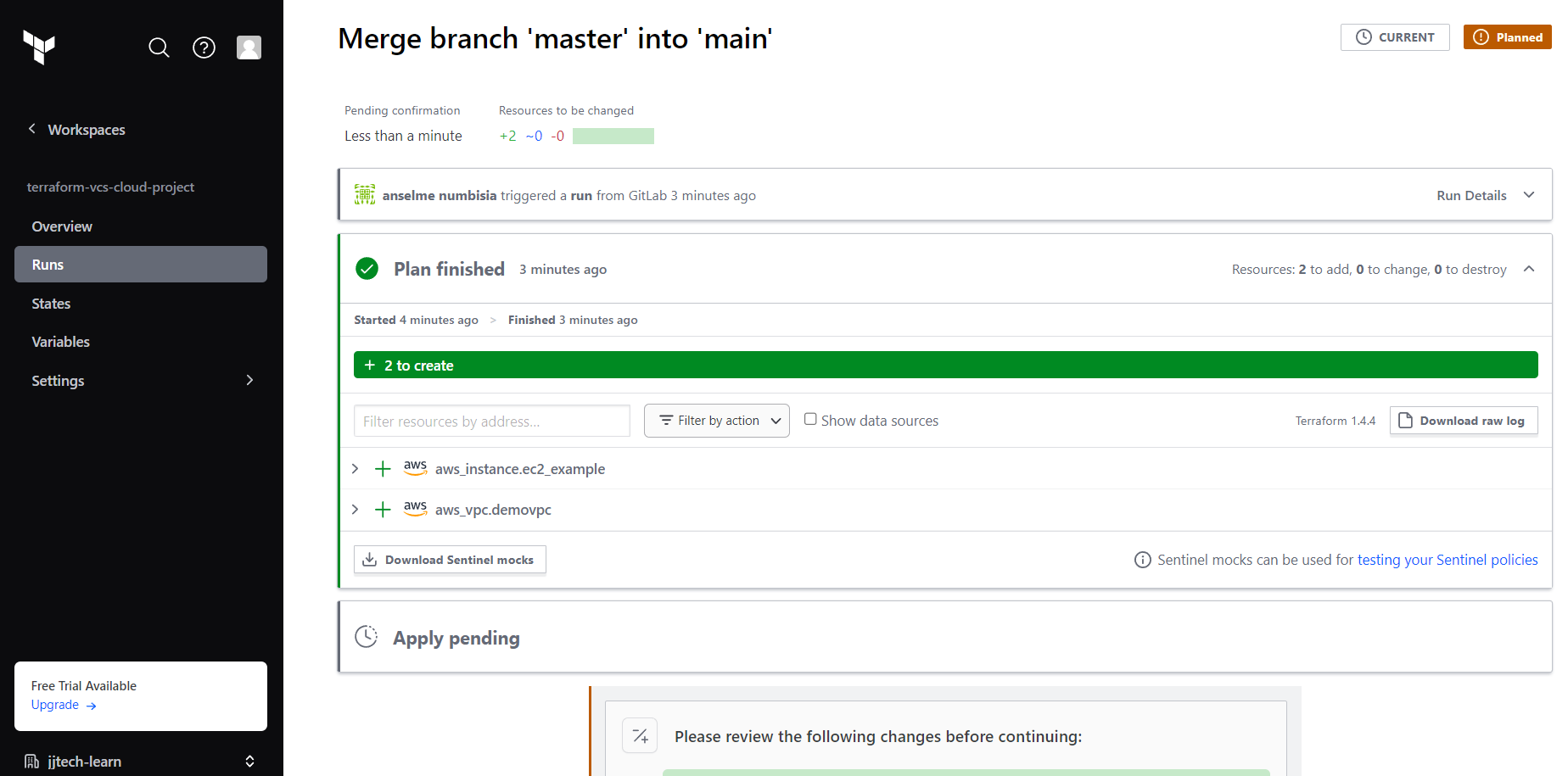
Description automatically generated

* Click on **start a new plan** >> provide reason for starting run >> under **Choose type run**, select **Plan only >> start run**

Graphical user interface, text, application

Description automatically generated

* Observe how plan is triggered. If failed, Navigate back to gitlab and crate a merge request from master to main branch and once merge request is approved, terraform cloud triggers plan



* If plan is Ok, click on **confirm and apply**

Graphical user interface

Description automatically generated

* To destroy the infrastructure, navigate to the setttings tab of your terraform cloud workspace

Graphical user interface, text, application

Description automatically generated

* In the next page, click on Destruction and Deletion in the next tab

Graphical user interface, text, application, email

Description automatically generated

* Select Queue destroy plan



* On the next prompt , enter the workspace name and click **queue destroy plan**

Graphical user interface, text, application, email

Description automatically generated

* Approve the destroy and click on **Confirm plan** to destroy

Graphical user interface, application

Description automatically generated

* Resources are getting deleted

Graphical user interface, text, application

Description automatically generated