CI/CD Pipelines for Hybrid Clouds with Cisco ACI, Git, Ansible and Terraform LTRDCN-2691

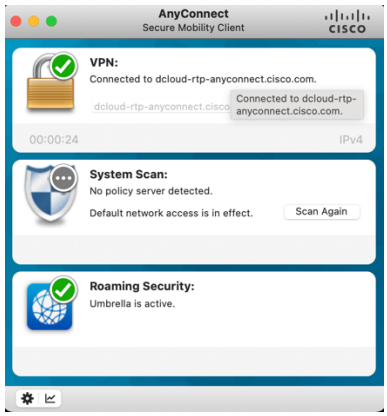
Speakers:
Miłosz Sabadach
Achintya Murali

Introduction

Welcome to the Instructor Led Lab on CI/CD Pipelines for Hybrid Clouds with Cisco ACI, Git, Ansible and Terraform! We're thrilled to have you here and shortly you will be joining us as an employee of a fictional company RuiSoft and embark on an amazing journey to build an application using CI/CD pipelines and all those beautiful technologies you read in our title :)

- **How to navigate this lab?**

Your primary source of accessing our infrastructure will be via your dCloud set up. You've already been logged into the VPN session, however in case you're logged out or have connectivity issues, reach out to us and we'll do our best to help. Please check if you are connected to the VPN, it should look at follows:



On your desk you can find your user pod number, which you'll use to navigate the infrastructure.

Our access environment consists of the User VM, Gitlab Server and Gitlab Runner VMs. You can RDP to any of these devices by using the access details, which can be found later within this Lab Guide or alternatively inside of a Documentation directory found in *~/Desktop/Infrastructure* on the User VM.

The contents of the User VM's Desktop include the abovementioned Infrastructure directory, which you'll be modifying throughout this lab and pushing to your Gitlab projects; and the WorkingScripts directory, containing code for the first three tasks, which will be more guided in nature.

You will use the User VM to write your code. The Runner VM will come into picture when you start executing pipelines on Gitlab.

Tasks concerning AWS can be directly done on your laptop using a browser or via the User VM.

- **On AWS accounts**

You need to create an AWS account (we suggest creating a brand new one), which is where you will be creating your cloud components in today's lab. You will do this in Task 3 Upon creating an account, you'll be asked for your credit card details for AWS to do a small refundable confirmation transaction. At the end of the session, you can delete the resources that are created throughout this lab in the cloud so that they don't continue to accumulate any charges.

- **How complex is this lab?**

The honest answer is that it is as complex as you would like to make it. We understand that each of you has a different experience with each of these technologies and we have built this lab with this in mind.

In case you are a beginner in a particular language (Ansible/Terraform), then we would suggest trying your best to create the scripts yourself but refer to our GitHub repo regularly to study the approach we have taken in our working scripts. In case you feel overwhelmed by a part of the script, feel free to clone the code from the GitHub repo and spend time studying what exactly the script is doing and studying the final result on the NDO/ACI/cAPIC/AWS.

Similarly with AWS or Gitlab, in case you are new to it, try to study the screenshots carefully and follow them accordingly. We've inserted several links all throughout the guide on topics that may be complex, and you can take a few minutes to refer to those to get a better understanding of what's happening.

The main Github repo https://github.com/achintya96/LTRDCN-2691/ is ordered in the form of task numbers. You can choose to clone the whole repo to the User VM and then refer to it as and when you like. Just keep a mindful eye on the clock and try to stick to the estimated timer we have mentioned for each task (at the top of the Task Description).

In case you are an expert in a particular technology, then we suggest staying away from GitHub repo as much as you can help it and try building the scripts out yourselves. Similarly with AWS and Gitlab, try to avoid looking at the screenshots we have in the guide and set the configuration up yourself.

We have also left some optional tasks that you can perform in case you are doing well on time (refer to the estimated time we've mentioned on top of the task) and in case you want to dig deeper into a particular technology.

Hope you enjoy, learn some new skills, hone your craft, and see why we love these technologies and solutions.

Best of Luck
Milosz and Achintya

Learning Objectives

Upon completion of this lab, you will be able to:

- Deploy ACI/NDO objects using automation tools such as Ansible and Terraform
- Work with a SCM (source control management) tool such as Git to keep track of your infrastructure configurations
- Utilize DevOps principles in your day-to-day infrastructure work

Scenario

Welcome to RuiSoft! As a recently hired IT Infrastructure Engineer, you'll be responsible for the further implementation and optimization of our company's digital journey. At RuiSoft, we're offering software solutions and professional support with designing and implementing networking solutions to our customers. We're also an (un)official Cisco partner, selling networking equipment.

Currently, the only way our customers can reach us is via traditional means of contact. As part of our digital journey, we want to offer our customers a selection of our software solutions and networking products in the form of an online shop, built into our website.

In the past we've also had issues with our website going down due to traffic spikes. To address this issue, we want to offload additional traffic to the highly available servers, but as a medium-size business we don't have the budget to deploy these and maintain them in an additional data center site, thus we want to go with a public cloud provider.

We use a small ACI single-pod site to test solutions for our customers. We now want to utilize it to expand our own offering, securing the budget needed to stretch ACI to a public cloud solution. Our developers team already utilizes the DevOps principles to work on the software solutions offered by us. Now, we want to do the same for our infrastructure management. This is where YOU step in!

High-Level Network Design

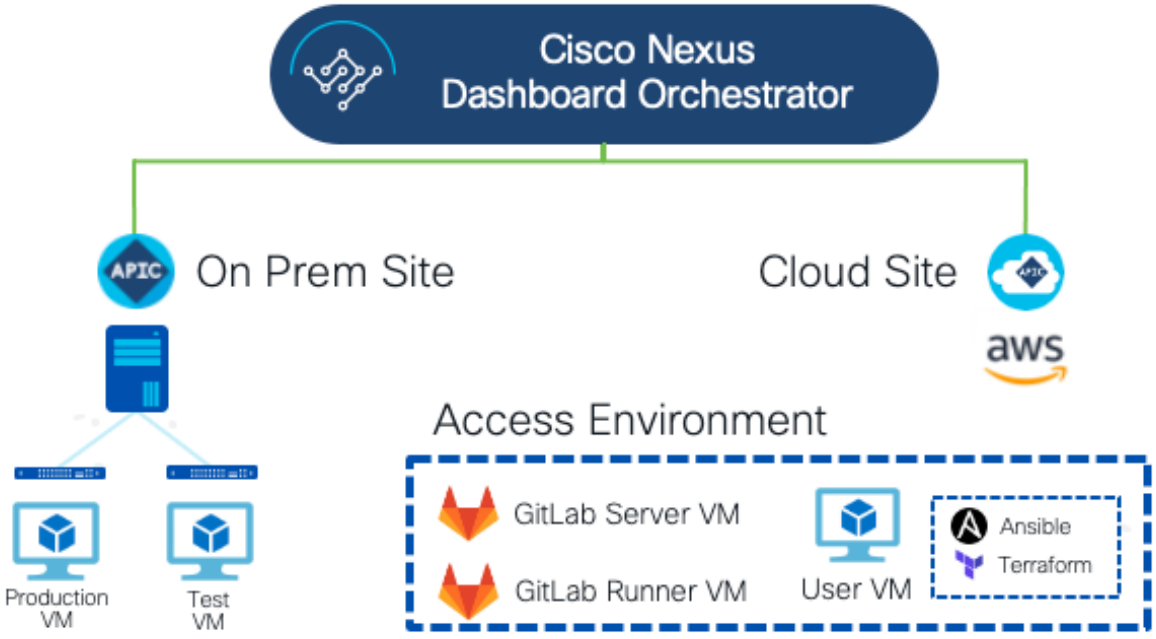


Table: Access credentials

Machine	Username	Password
Gitlab	CLUS-userXX	CLUS-userXX-123!
User VM	dcloud	C1sco12345
Gitlab Server	dcloud	C1sco12345
Gitlab Runner	dcloud	C1sco12345
NYC APIC	CLUS-userXX	CLUS-userXX-123!
Cloud APIC	CLUS-userXX	CLUS-userXX-123!
NDO	CLUS-userXX	CLUS-userXX-123!
vCenter	aciadmin@vsphere.local	C1sco12345!
Production VM	clusadmin	c1sco123
Test VM	clus-test	clus-test-123!

* where XX stands for your user number that you've been assigned, i.e., CLUS-user01

Table: IP Addressing

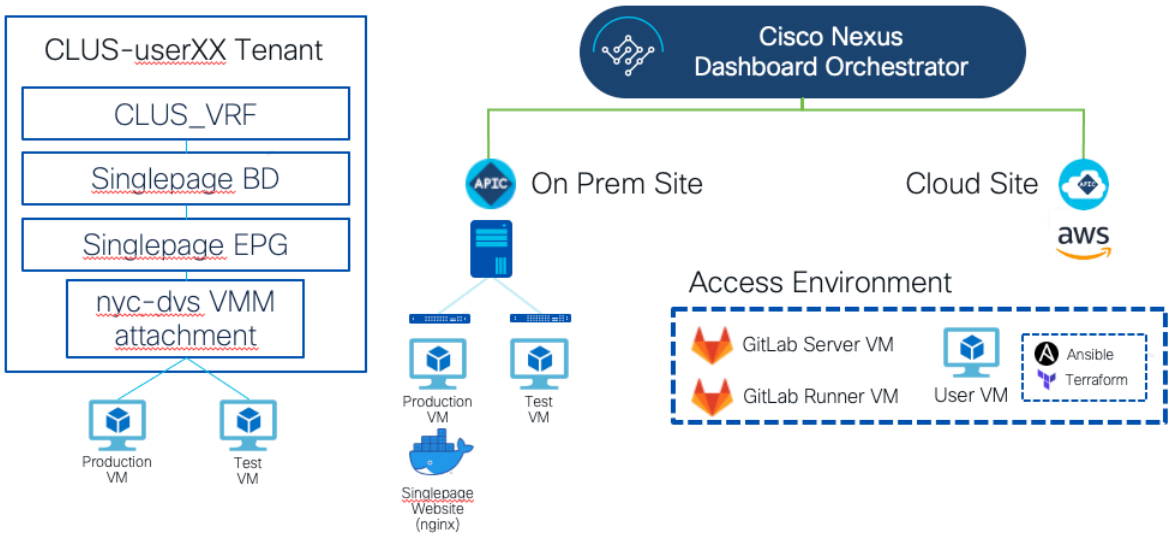
Machine	IP Address
NYC APIC	198.19.202.66
NDO	198.19.202.12
Cloud APIC	3.229.175.252
vCenter	198.19.202.251
User VM	198.18.133.198
Gitlab Server	198.18.133.199
Gitlab Runner	198.18.133.200

Task 1: Upload our current infrastructure configuration to a Git repository and deploy a single page website with the use of Ansible

[Estimated time: 35-40 minutes]

The first step of our infrastructure DevOps transformation will be to get our current networking configuration uploaded to a Git repository, where you will be able to keep track of the changes being made to the configuration codebase. We've never used any automation tools for our infrastructure management, which we want to change as well. We'll be utilizing Terraform and Ansible as our automation tools, the former due to its ability of keeping track of the configuration state, and the latter due to its simplicity and lightweight dependencies.

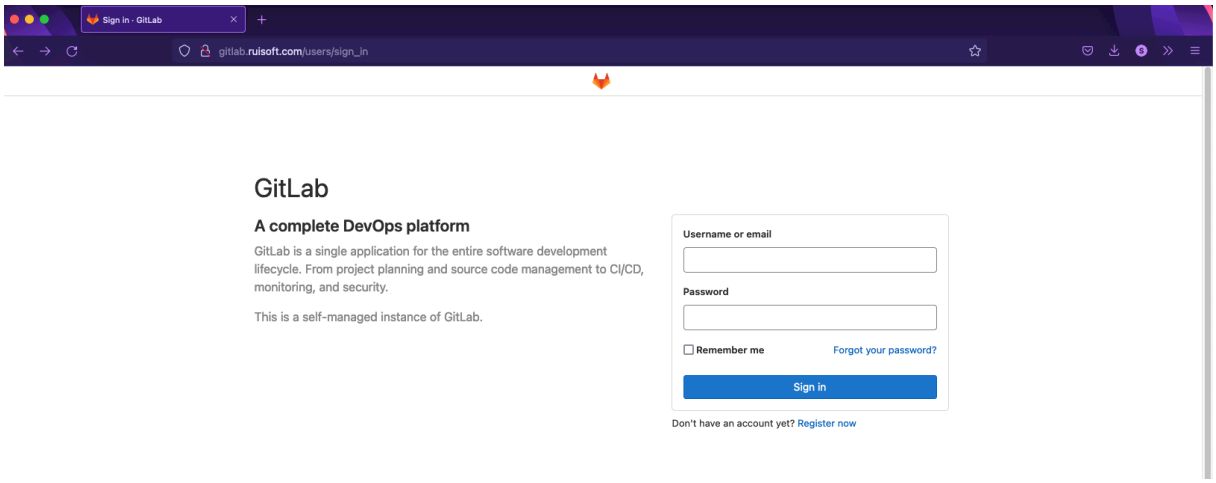
Since our website is already deployed and the code behind it is being maintained in a Git repository of our developers' team, we now want to follow the same approach for the infrastructure management. To deploy the current version of our website, we'll be using Ansible and the collections available for ACI.



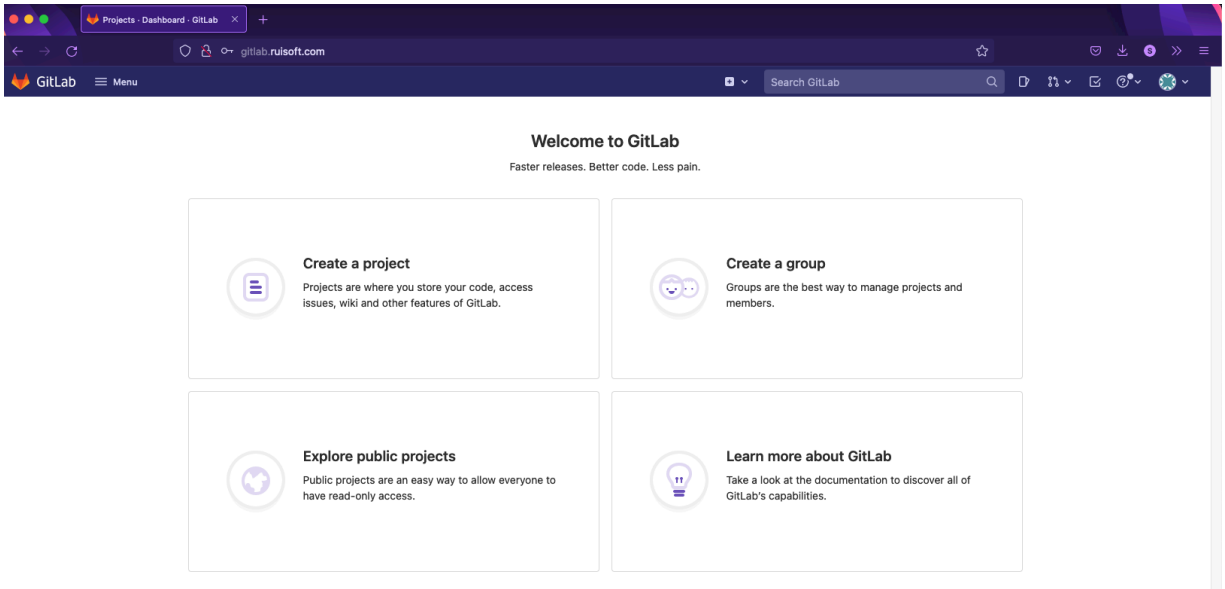
Step 1: Set up your Git configuration

A Git package has already been installed on your virtual machine. There is an instance of Gitlab Server and Runner deployed in your environment as well, with the account provisioned for you.

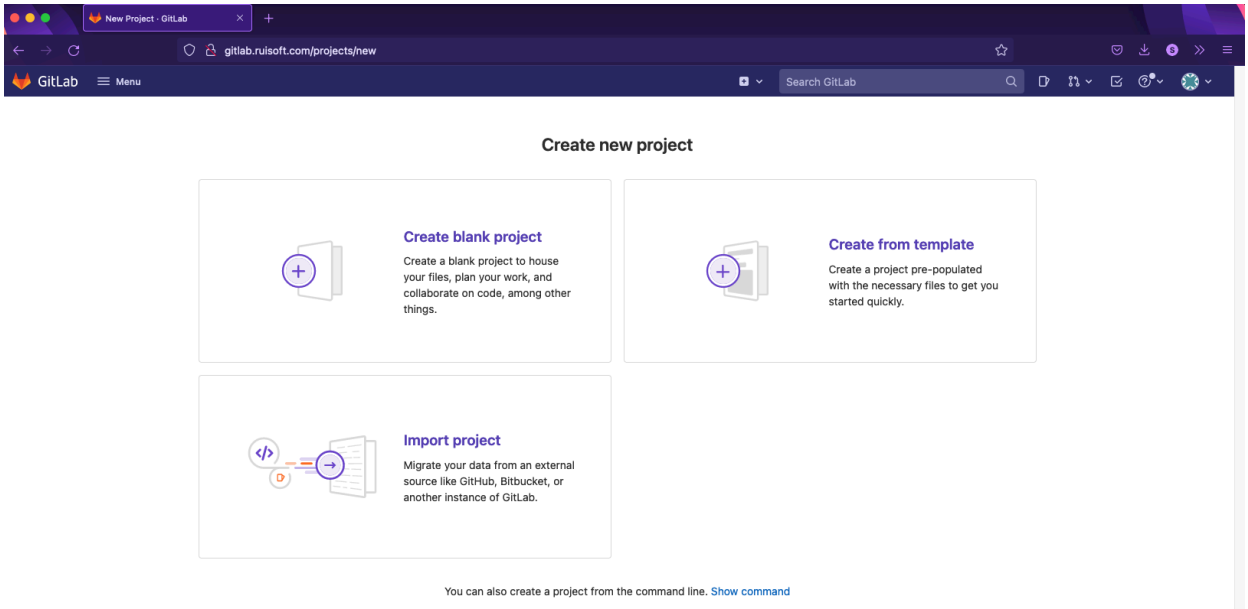
1. Go to http://gitlab.ruisoft.com and log into your account. You can find the credentials to your account in the Access Credentials table on page 7.



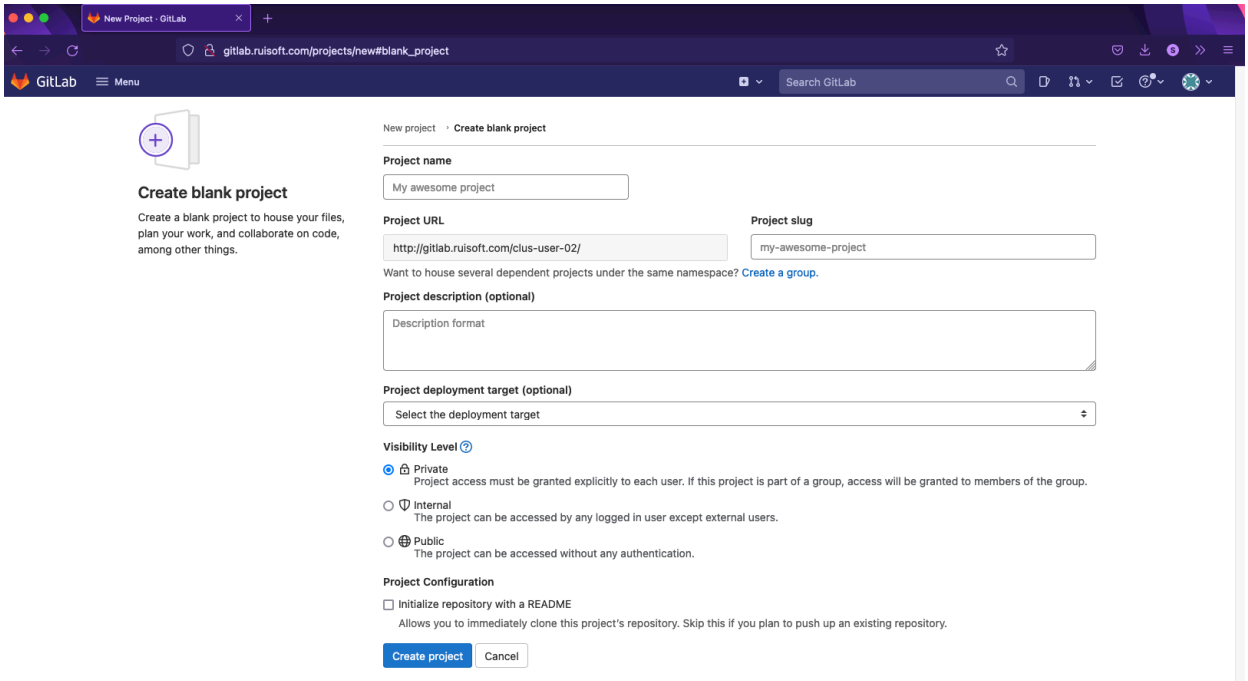
2. Upon a successful login, you'll be met with the following screen. Here, you can choose between different initial action that can be taken within Gitlab. This is specific to Gitlab itself and not Git as a SCM solution. Please select ***Create a project***.



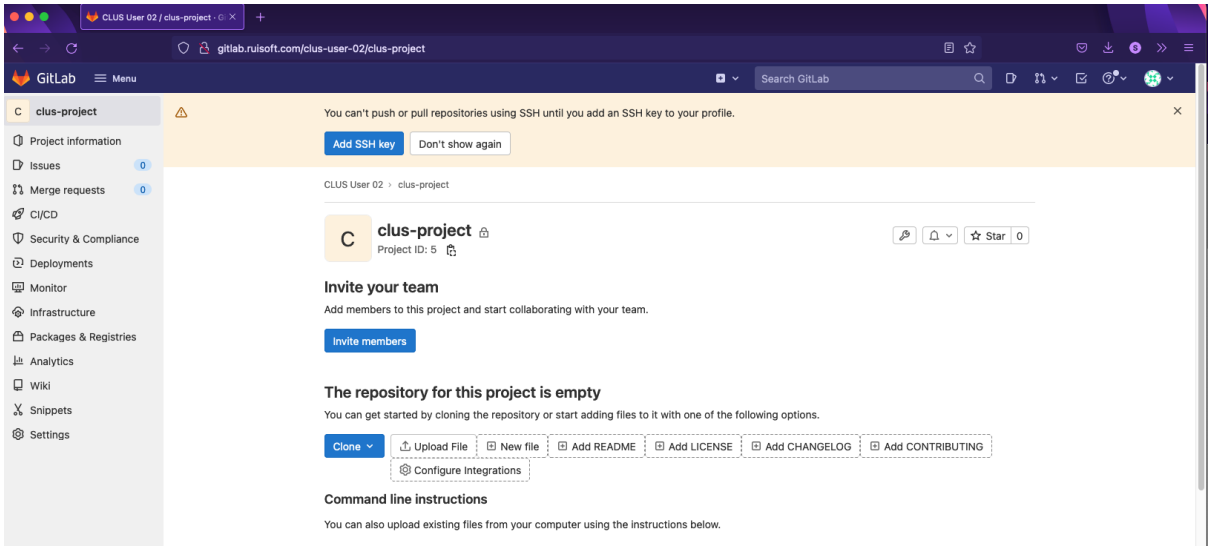
3. Select the ***Create blank project*** option.



4. On the project creation wizard, provide your project name, set the visibility level to private and deselect the checkbox next to *Initialize repository with a README*. This way we'll be able to upload a repository created by ourselves locally.



5. After successfully creating the project, you'll notice a couple of things. The main standout is the fact that the repository for our project is empty. What we'll want to do is to upload our documentation files to this empty remote repository from our machine.



6. Our local machine already has Git installed. Confirm your version of Git with the following command:
   git --version

   Configure your username and e-mail with the following commands:
   git config --global user.name 'CLUS-user<nr>'
   git config --global user.email 'CLUS-user<nr>@ruisoft.com'

   The *--global* keyword signifies that these credentials will be used by default on all Git repositories running on your local machine. In case you've made a typo in your git config credentials you can also remove them by running a *git config --unset user.name* or *git config --unset --global user.name* command for example.

   The variables we've just configured are stored in the Git configuration files, these usually being stored in /etc/gitconfig on a system-wide basis and within .git/config in your repository.

   Verify your current Git configuration with the following command:
   git config --list

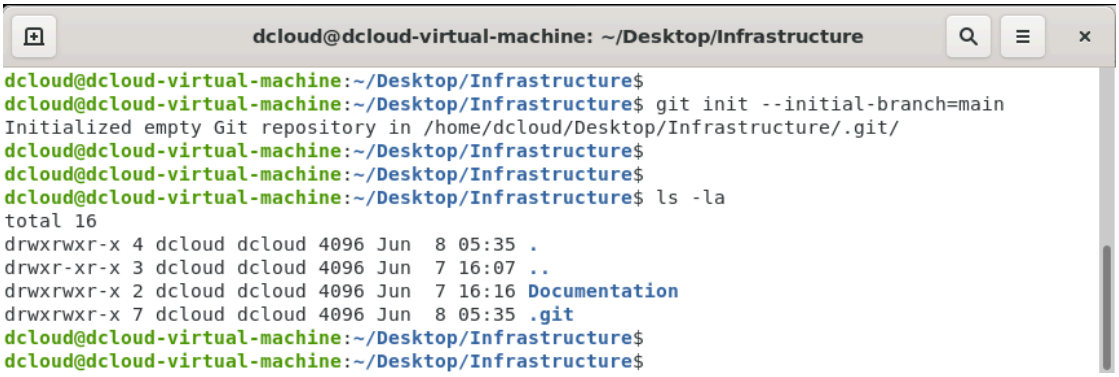
Step 2: Create and upload a Git repository

We shall now download the Git repository, which we created in Step 1. This is where you will be storing your Infrastructure configuration as part of Ruisoft's digital transformation.

You can find the current infrastructure documentation details in *~/Desktop/Infrastructure/.* This will be our initial directory that we will be adding to our remote repository. This way, we can quickly work on the same set of files (and in the latter steps – codebase) among our team members.

1. Initialize a new repository within our infrastructure folder:
   cd ~/Desktop/Infrastructure
   git init --initial-branch=main

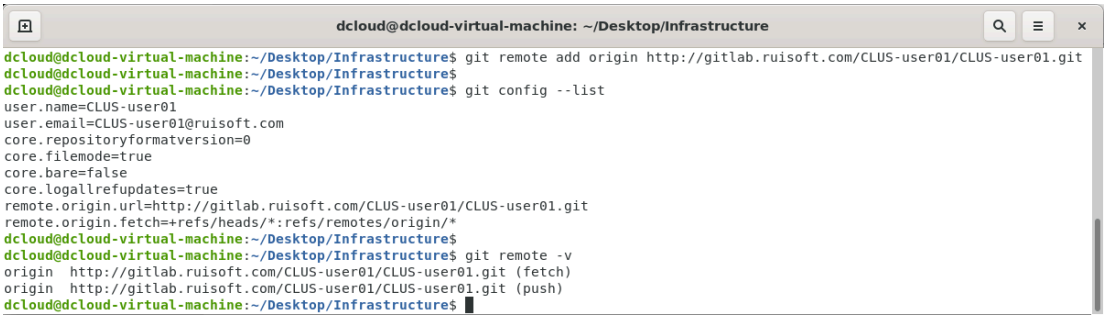
   You can verify that Git was initialized by running *ls -la* – you'll find a .git subdirectory created.



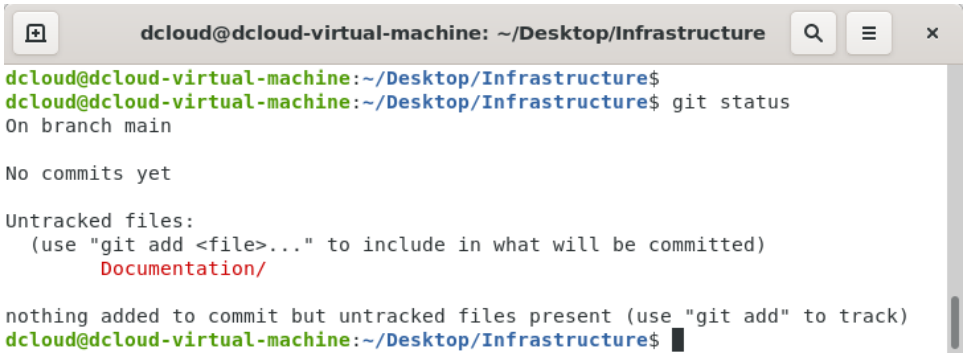
   The *--initial-branch=main* parameter specifies the name of an initial branch.

2. Add the remote repository with the following command:
   git remote add origin http://gitlab.ruisoft.com/CLUS-userXX/<your-project-name>.git

   With this command we're specifying the remote URL where the repository is located. To verify, you can run the following commands:
   git config --list
   git remote -v

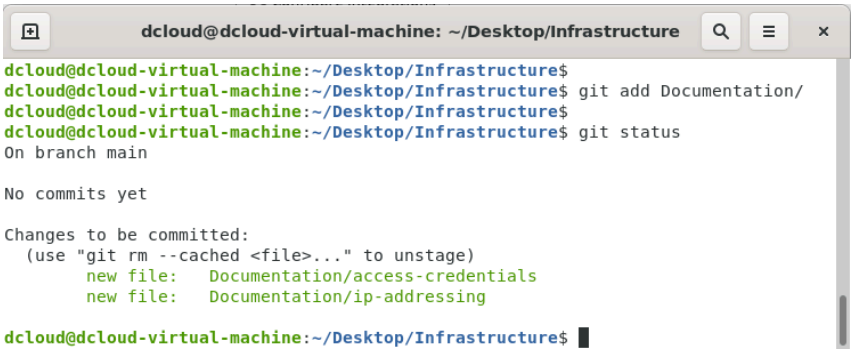


3. Run the following command to verify which files have been added to the staging area:
   git status



4. In our case, since we've initialized the repository in an existing directory, files were already present and not added to the staging area – you must do that manually.

   Add the documentation files to the staging area:
   git add Documentation/



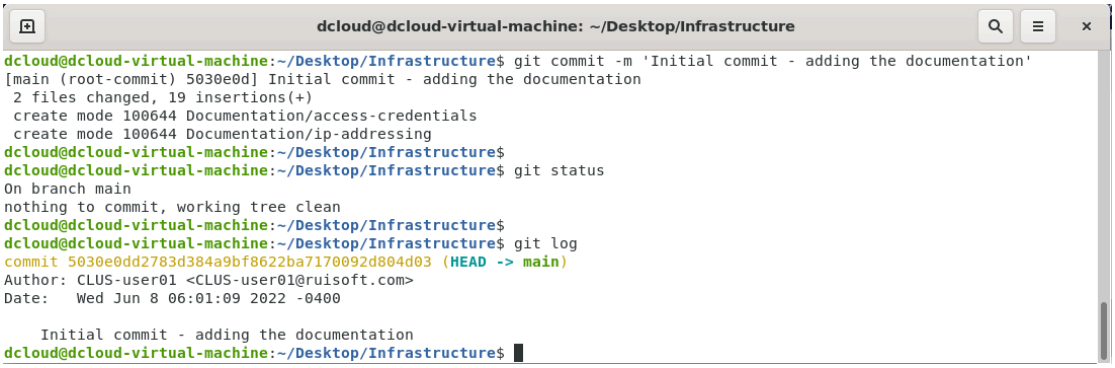
   Right now, the documentation files have become tracked and added to staging. They are now ready to be committed to our repository.

   You could've also used the *git add .* command – it would add all the files in your directory to the staging area.

5. Commit the file to your local repository:
   git commit -m '<your commit message>'

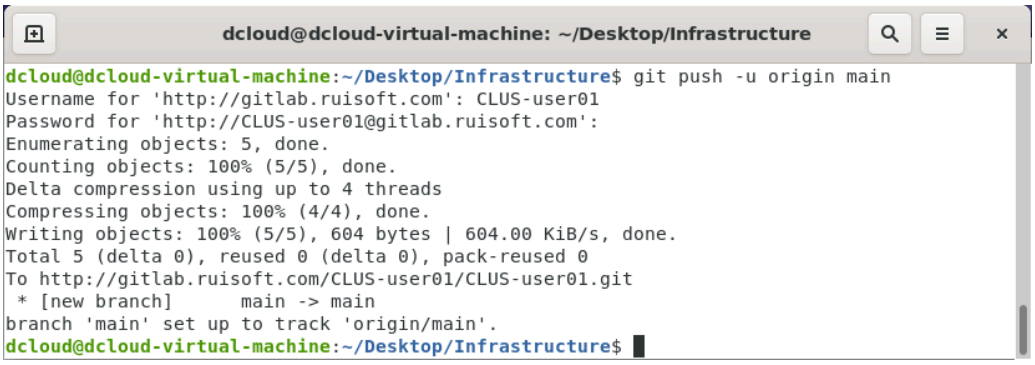
   With the *-m* parameter with include a commit message – we should always include it while making it clear and concise. This way our team can see what was altered in our code, without wondering what the intention behind the modification was.

   Now, after running the *git commit* command, with the *git status* command you can verify that there are no files present in the staging area, while with *git log,* we can track all the local commits:



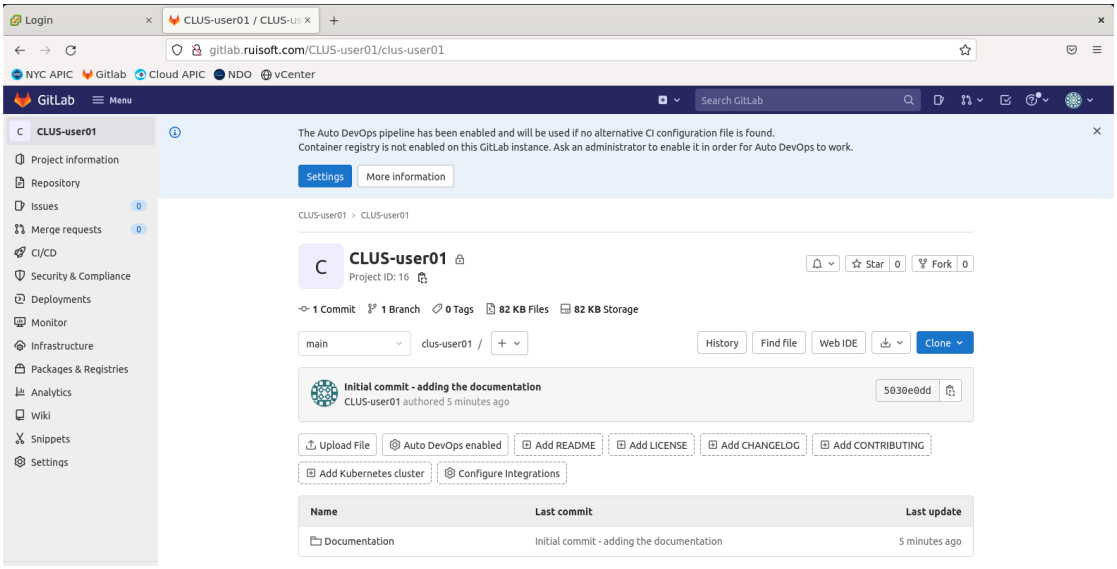
You can notice the SHA-1 checksum next to the commit, the author, date, and the commit message. What needs to be done now is to synchronize our local repository with the remote one. Git does it by utilizing a *git push* command.

6. Push the state of your local repository to Gitlab by using:
   git push -u origin main



The *origin* argument is the name of the remote we've added in point nr 2 of this step. If you've named the remote differently, you must change this argument accordingly to reflect your name of the remote. Similarly, the *main* argument is the name of the initial branch created. Lastly, we've used the *-u* option, which adds upstream branch tracking references, which are utilized by commands such as *pull* (which we'll cover in a bit).

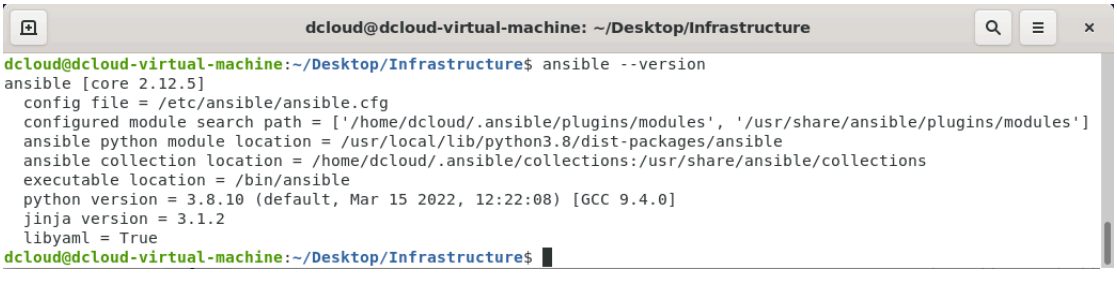
7. Access *http://gitlab.ruisoft.com/CLUS-userXX/<your-project-name>* to check if the repository has been updated with your commit:



Upon the successful push you'll see your files on the Gitlab server. The changes to repositories don't always need to be done locally first and then pushed to the remote. It can be the other way around too – for example, in your remote repository you could add a file directly or gitlab.ruisoft.com. In such case you'd need to synchronize your local repository with the remote one to avoid inconsistencies with the code – to do so, you'd use a *git pull* command. In our case, we won't be doing that, as we'll be modifying code locally and pushing to remote repository.

Step 3: Prepare Ansible playbooks to create the ACI objects on the on-prem site to attach the single-page website to your pre-provisioned Tenant

1. Ansible has already been installed on your machine. Please verify whether it has correctly been done so by running the following command:
   ansible --version



2. Create an empty directory for Ansible in the Infrastructure directory using the following command:
   mkdir Ansible

3. Cisco has its own set of collections published, for various of our products. For today's lab, we'll be using the ***aci*** and ***mso*** collections. The latter still refers to our old name for Nexus Dashboard Orchestrator – the Multi-Site Orchestrator, MSO in short, but worry not – the mso collection still applies to the NDO as well.

   These collections have already been installed on the User VM. To verify that, run the following command:

   ansible-galaxy collection list | grep aci
   ansible-galaxy collection list | grep mso

   If they are not present, download them using the following commands, as specified on the Galaxy pages for the collections:

   ansible-galaxy collection install cisco.aci
   ansible-galaxy collection install cisco.mso

4. Move to the Ansible folder and create your inventory.ini file as follows:

```
[apic]
198.19.202.66 apic_user=CLUS-userXX apic_pass=CLUS-userXX-123!
apic_tenant=CLUS-userXX #edit the XX with your user id
```

   The reason for that is we'll use these variables when creating a playbook that will utilize modules from the ACI collection.

5. Let's utilize modules from the ACI collection to create the logical objects necessary to attach the endpoint where our website will be hosted to the ACI fabric.

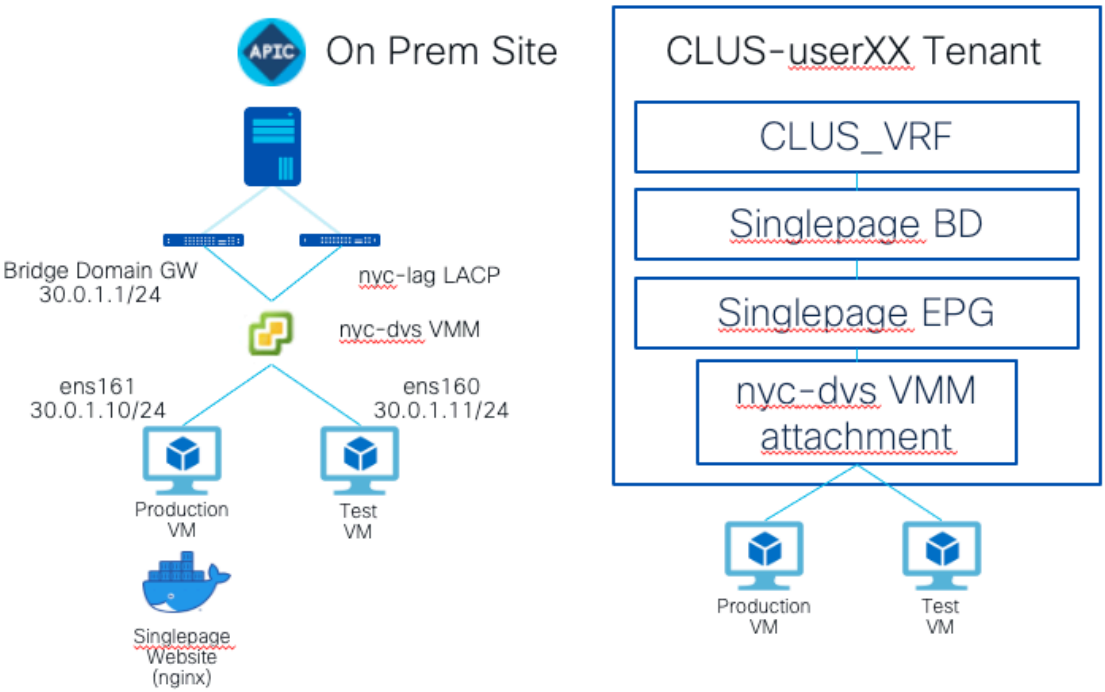
   While this Instructor Led Lab assumes that you have experience working with ACI, in case you do not have much experience with the product, here are some useful resources, briefly summarizing what is required for an endpoint to be learned by ACI:

   - ACI Fabric Endpoint Learning White Paper
   - Cisco APIC Basic Configuration Guide
   - ACI – Network-Centric Approach White Paper

   All the ACI access policies that you'd be using in this step have been already configured. For this step you'll need to pass the following objects inside of your Ansible code properties:

   - Your pre-provisioned tenant – ***CLUS-userXX***, where the X's stand for your user number
   - Your pre-provisioned VRF – ***CLUS-VRF***
   - VMM domain – ***nyc-dvs***, as our website-hosting endpoint is a virtual machine

- VMM Enhanced LAG policy – ***nyc-lag***, as we're using LACP on the vDS uplinks
- Single Bridge Domain, Application Profile, EPG with the naming convention of your choosing, with the IP addressing from the desired state topology below:



For your reference when working with the ACI collections for Ansible, please use the following links with the documentation:

- Cisco.Aci Ansible Collection Documentation
- Cisco ACI Ansible Scenario Guide
- Cisco.Aci Ansible Galaxy Page

For the playbook, it is important to provide you with some leads, in respect to the way our environment is set up:

- Specify the ***connection*** property in the playbook to be set to ***httpapi*** – this way, you'll be connecting to the APIC using API calls, as this is the commonly supported connection mode across controllers such as APIC
- Under the appropriate modules of your choosing, always specify the ***validate_certs*** property to be set to ***no*** – our APIC is in a lab environment, we don't have a certificate generated for it

Here's an example of the playbook syntax that you can use in this step of the task:

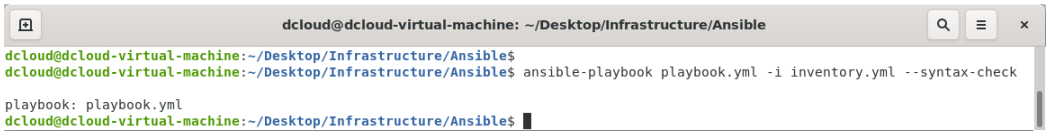
```
- name: Playbook for Task1 Step3 - creating the ACI objects for
single page attachment
  hosts: apic
  gather_facts: no
  connection: httpapi

  tasks:

    - name: Create a BD for the frontend tier
      cisco.aci.aci_bd:
        host: 198.19.202.66
        username: '{{ apic_user }}'
        password: '{{ apic_pass }}'
        tenant: '{{ apic_tenant }}'
        vrf: CLUS_VRF
        bd: BD_Singlepage
        validate_certs: no
        state: present
```

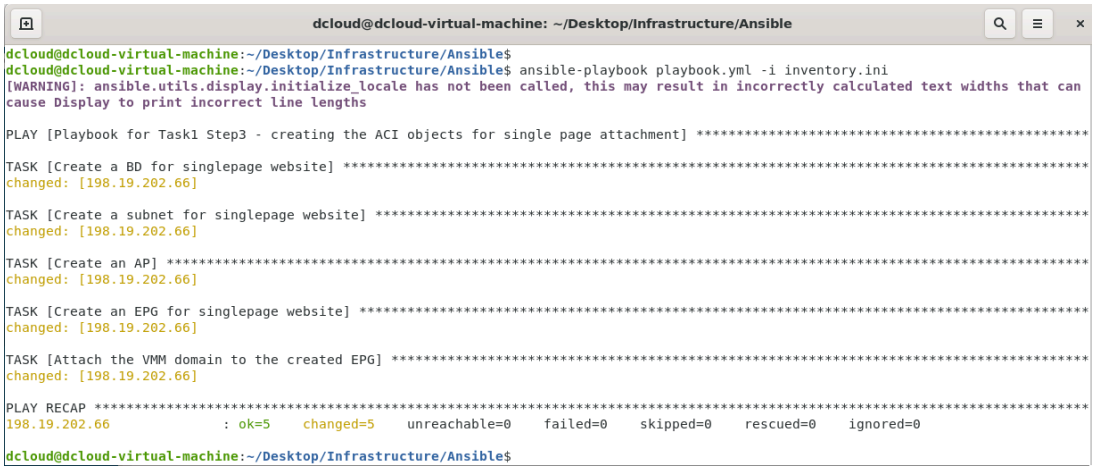
You'll need to prepare the rest of the playbook similarly, in order to create the objects we've mentioned earlier. After preparing your code, test it before running it against the APIC. For now, we'll use the following command that will verify the syntax of your playbook:

ansible-playbook <your playbook name> -i <your inventory file> --syntax-check

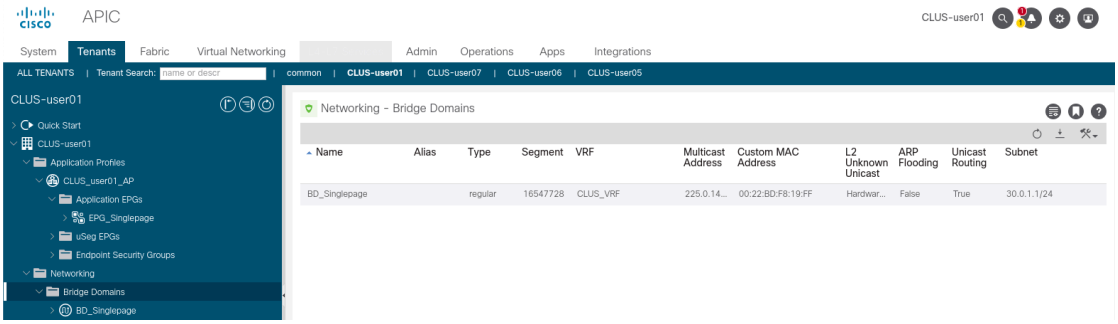


6. If you don't see any issues with your code syntax, deploy it in your tenant with the following command:

ansible-playbook <your playbook name> -i <your inventory file>



7. To verify that the playbook indeed works, go to the APIC GUI and check the created objects under your Tenant.



Step 4: Deploy the single-page website on your production VM

Now, with the ACI logical constructs necessary for the endpoint to be able to communicate with the rest of our network in place, we'll have to configure our production VM to use the port group that was created by the APIC. We'll then need to run the Docker container with our website code.

To do so using Ansible code, we'll rely on a community-maintained collections for VMware and Docker, as there are no official collections available as of the writing of this lab guide.

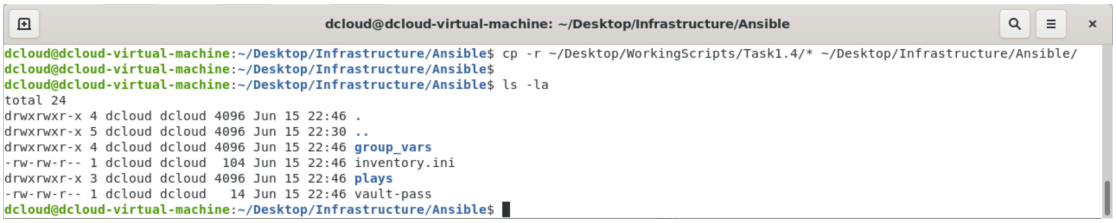
For the next steps, you don't need to code the following scripts. Just clean up the Infrastructure/Ansible directory and copy the contents of the ~/Desktop/WorkingScripts/Task1.4 to said Infrastructure/Ansible directory. You will then follow us in the analysis of the scripts in detail. You'll need to fill out the credentials, IP and MAC addresses and port groups according to your user pod number.

1. Let's download the VMware and Docker collections from Galaxy, using the following command:

ansible-galaxy collection install community.vmware
ansible-galaxy collection install community.docker

```
dcloud@dcloud-virtual-machine:~/Desktop/Infrastructure$ ansible-galaxy collection install community.vmware
Starting galaxy collection install process
Process install dependency map
Starting collection install process
Downloading https://galaxy.ansible.com/download/community-vmware-2.5.0.tar.gz to /home/dcloud/.ansible/tmp/ansible-local-4361782x81irrv/tmpzslhs10u/community-vmware-2.
5.0-9c5kwjq2
Installing 'community.vmware:2.5.0' to '/home/dcloud/.ansible/collections/ansible_collections/community/vmware'
community.vmware:2.5.0 was installed successfully
dcloud@dcloud-virtual-machine:~/Desktop/Infrastructure$ _
```

2. Copy the code from the *~/Desktop/WorkingScripts/Task1.4* to your Ansible directory by using the following commands:
*rm -rf ~/Desktop/Infrastructure/Ansible/**
cp -r ~/Desktop/WorkingScripts/Task1.4/ ~/Desktop/Infrastructure/Ansible/*



3. Now, before we start analysing the playbook itself, it's worth discussing some best practices when working with Ansible code.

   First, we're grouping our Ansible code into roles. Thus far we've been defining variables in the inventory. That will change moving forward – we'll be defining variables specific to a particular role, which will give us more flexibility based on the infrastructure changes we'll want to conduct – we'll only going to be making changes to the file holding variables. Our variables should also be grouped in a single place.

   Besides these, there are some other good practices for working with Ansible code that are worth mentioning:

   - Do not hard-code, use variables – these should be defined in one place.
   - Variables should not be used in the playbooks; they should be used in the roles.
   - When using roles, playbooks should be kept simple – you should reference the roles inside of a playbook.

   Finally, there's the topic of security. In what's a no-brainer advice, you shouldn't use any credentials or certificates in plain-text in your variable files. To help with that, Ansible offers you its Vault functionality. Instead of hardcoding our credentials into the inventory code, we'll be creating files specifying the credentials, which we'll then encrypt with Vault.

   Just remember to never push the plain password files nor the vault password file to your Git repository – you can influence that by creating a *.gitignore* file, where you'll specify which files to not push to your repository.

   With all that out of the way, let's start restructuring our directory. Below is a structure we've used when developing this step of the task:

```
group_vars/
|     apic/
|     |     apic.yml
|     |     apic-password.yml
|     |_    apic-password-plain.yml
|     vcenter/
|     |     vcenter.yml
|     |     vcenter-password.yml
|_    |_    vcenter-password-plain.yml
plays/
                |apic-object-creation-playbook.yml
                |service-deployment-playbook.yml
                |roles/
```

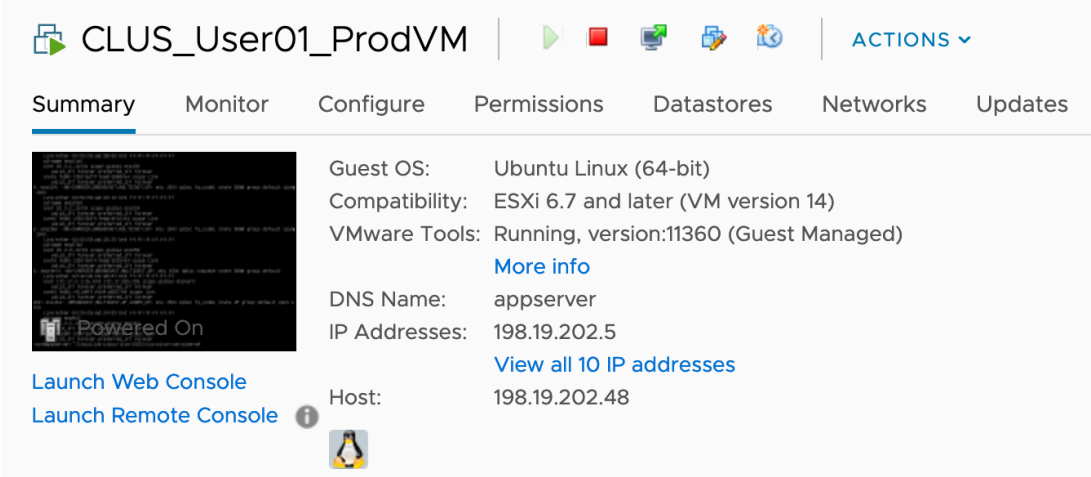
```
|     |     apic-object-creation/
|     |     |     tasks/
|     |     |_    |_    main.yml
|     |     service-deployment/
|     |     |     tasks/
|     |_    |_    |_    main.yml
vault-pass
inventory.ini
```

The directory structure listed above should loosely give you an idea of what we're aiming for with the next steps.

4. We started by updating the inventory file to include the details about our vCenter and the Production VM, while removing credentials and other variables.

```
[apic]
198.19.202.66
```

You'll have to fill out the inventory.ini file similarly to provide IP addresses of the vCenter and your Production VMs. You can find the IP for your Production VM when you login to the vCenter. Your production VM will be called CLUS_UserXX_ProdVM in the NYC Data Center.



Usually, you'd have a separate inventory file per environment (testing, staging, production), however in this scenario we'll be making changes directly to our production, thus we'll proceed with a single inventory file.

5. We then create the variable files for the APIC and the vCenter in the *group_vars* directory (*group_vars/apic/apic.yml* and *group_vars/vcenter/vcenter.yml* in our example, accordingly). Since we're trying to follow best practices, this is where we specified the names of ACI logical constructs and information relevant to those (i.e., subnets), names of our vSphere datacenter, VMs, portgroups and the MAC addresses of interfaces which you'll be attaching to the fabric, and of course, the IP addresses of the APIC/vCenter, as well as the credentials to access them.

You'll need to check all of these files and modify all the lines with the comments in accordance with the details that can be checked on the vCenter for your Prod and Test VMs(CLUS_UserXX_TestVM).

For example, an exempt of the *group_vars/apic/apic.yml* file for User 01 can look like this:

```
[…]
apic_host: 198.19.202.66
apic_tenant: CLUS-user01 #change this according to your details
apic_vrf: CLUS_VRF
[…]
```

Make note that we're not hard coding credentials in the variable files! Instead, we reference variables which defined in the plain password files which we then encrypt with Vault, for example a potential line from *apic.yml*:

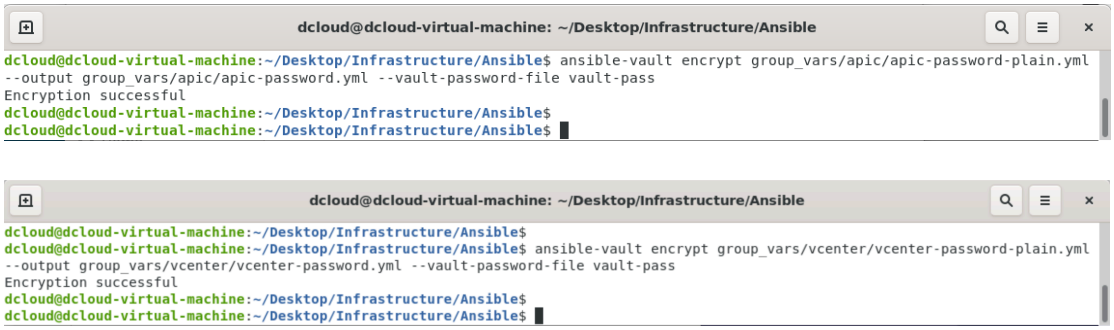
```
apic_creds: {user: "CLUS-user01", pass: "{{ apic_user }}"}
```

6. We create the plain password files for the APIC and vCenter credentials (*apic-password-plain.yml* and *vcenter-password-plain.yml* in our example). In these we simply provide the password needed to access the devices. You must modify these files. Following is an example of the content of *apic-password-plain.yml*:

```
apic_user: #refer to the access credentials table
```

7. If you like, you can modify the file containing the string that will be used as your Vault password (called *vault-pass* file) in the ~/Desktop/Infrastructure/Ansible subdirectory.

8. Use the file you created to encrypt the apic and vcenter plain password files with Vault.
   ansible-vault encrypt <plain password file> --output <password file> --vault-password-file <vault password file>



9. We created the roles you'll be utilizing in your playbooks. Within the roles, for the APIC-specific role we put the tasks which we defined in Step 3 to create the logical constructs.

For the service deployment role, we define the tasks using the Community VMware and Docker collections installed by you at the beginning of this Step.

Below you can find the documentation of the collections:

- Community.Vmware Ansible Collection Documentation
- Community.Docker Ansible Collection Documentation

We must use a module allowing for updating the portgroups of existing VMs. This will pull a Docker image, create a network, and run the container. The image that we're using is tagged ***sabaroof/ciscolive2022:singlepage***. When starting the container, we pass "nginx -g 'daemon off;' as the command, since we aren't using a Dockerfile nor Docker Compose in this case.

Important to note is that we're using a property called *DOCKER_HOST* when using the Ansible module, to be able to run Docker commands on a remote host via API calls. To do that, an IP of the Production VM from the management network (198.19.202.0/24) is used, which you've already provided in the earlier steps by modifying the *group_vars/vcenter/vcenter.yml* file. For example, this is how a procedure of pulling an image would look like:

```
- name: Pull an image
  community.docker.docker_image:
    name: sabaroof/ciscolive2022:singlepage
    source: pull
    docker_host: "tcp://{{ vc_prodvm_mgmt_ip }}:2375"
    validate_certs: no
```

10. Finally, we proceed with preparing the playbooks. We update the APIC playbook used in Step 3 to remove the tasks from it and instead reference a role. For deploying the container, we create a playbook (*service-deployment-playbook.yml*), again, referencing only the role.

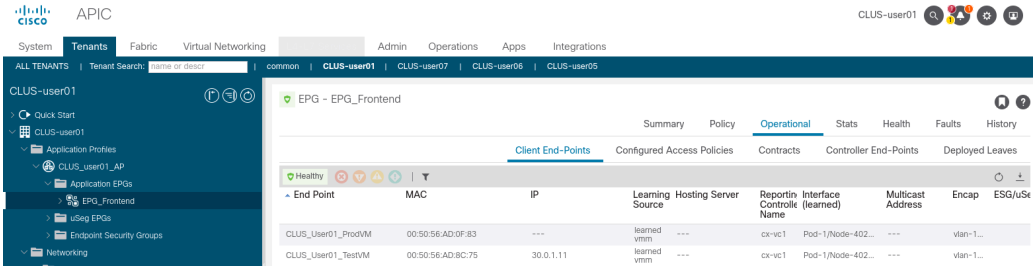
    For example, the APIC playbook (in our case we've renamed it to *plays/apic-object-creation-playbook.yml*) can look like this:

```
- name: Creating the logical constructs necessary to attach an
endpoint
  hosts: apic
  gather_facts: false
  connection: httpapi

  roles:
    - apic-object-creation
```

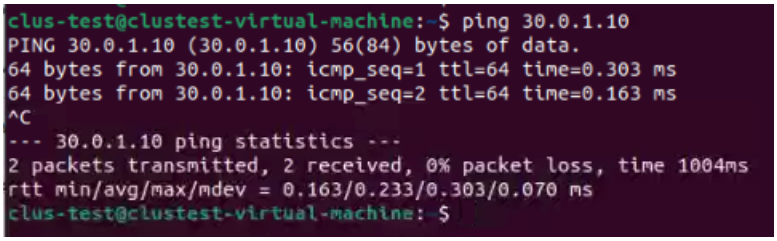
11. Run your Ansible playbook for the VM attachment and service deployment part: *ansible-playbook plays/service-deployment-playbook.yml -i inventory.ini --vault-password-file vault-pass*

12. Upon running the Ansible code, verify the following:
    - Endpoints being learned under the EPG in ACI

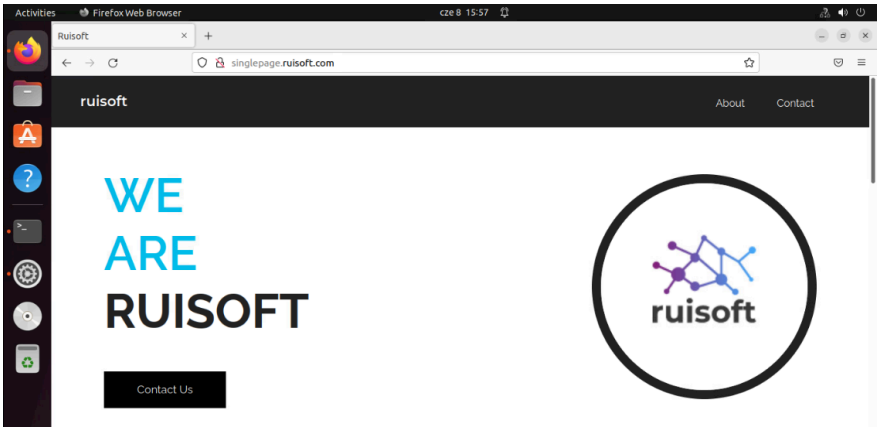


    - Verification of container standing up on Prod VM

    - Reachability between the Prod VM and Test VM

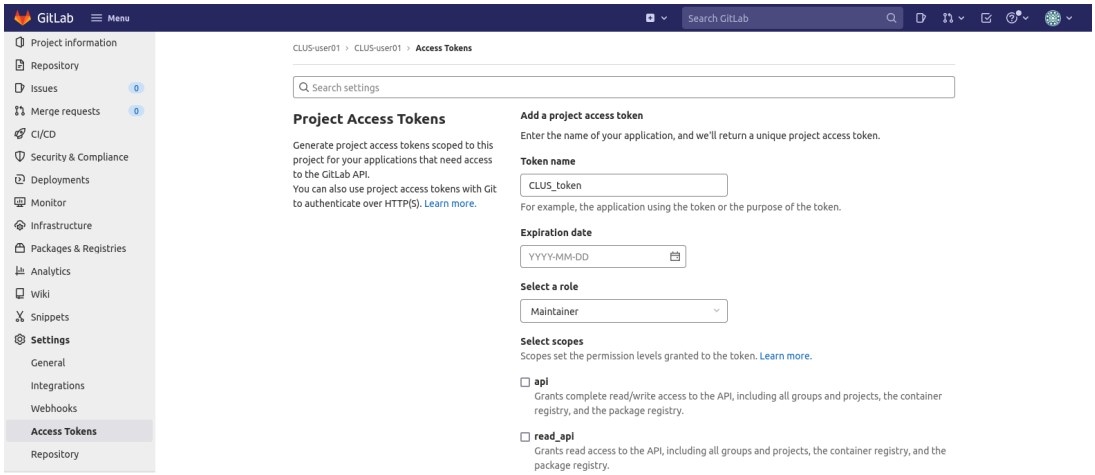


    - Ability to connect to the website from Test VM

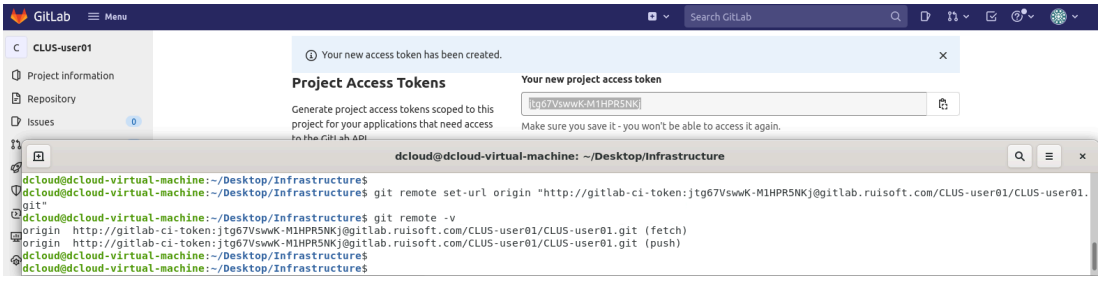


13. Upon ensuring that everything is working as intended, upload the code to your Gitlab repository. Before doing so, generate a Personal Access Token so that you're not asked to provide credentials each time you push to your repository.

    To do so, on your Gitlab project on gitlab.ruisoft.com go to ***Settings -> Access Tokens*** and fill out the Token name, select the ***write_repository*** scope and click on "Create project access token"



14. After clicking on "Create project access token" the page will refresh, and you'll see your token. Change the git remote to utilize the token as follows:
    git remote set-url origin "http://gitlab-ci-token:<token>@gitlab.ruisoft.com/<your user>/<your project>.git"

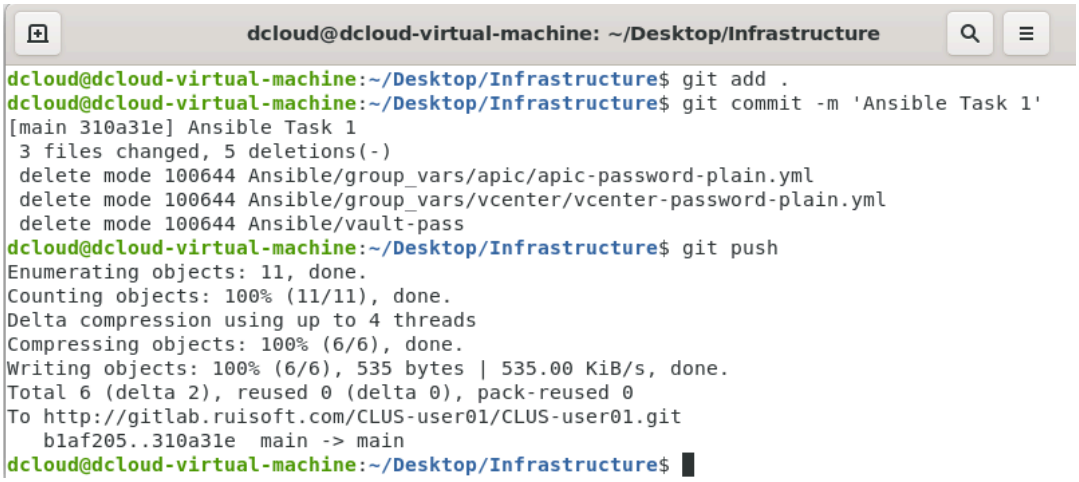


15. Create a .gitignore file with the following syntax inside the Infrastructure directory:

```
.gitignore
*-password-plain.yml
```

    This way you'll prevent yourself from pushing the plain password files to your Git repository.

16. Push your code with the following:
    git add .
    git commit -m 'your commit message'
    git push



- **What did we learn in this Task?**

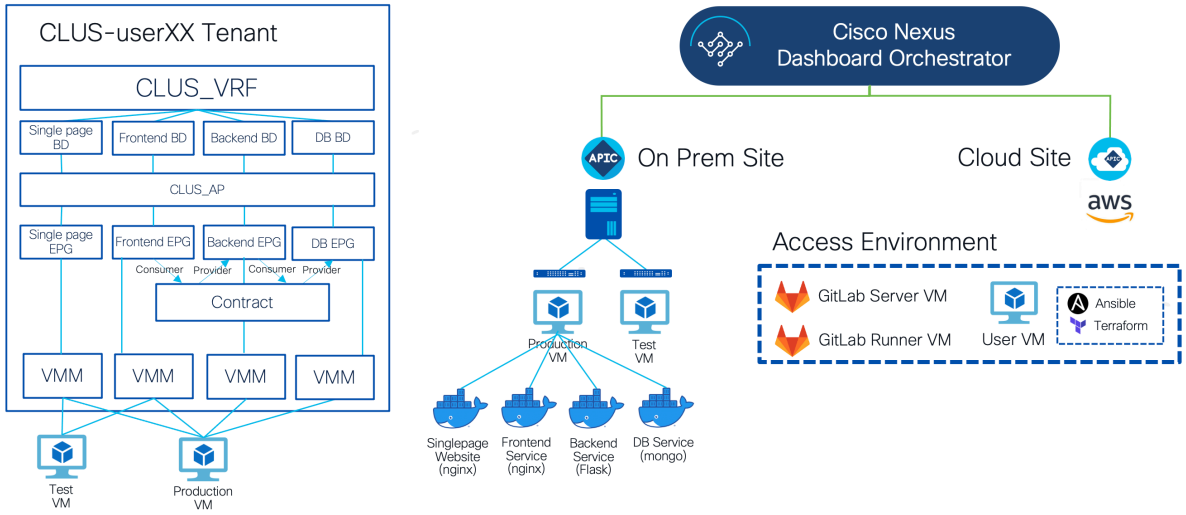
  - This task covered the basics of Git and Ansible.
  - It also made you work through a common scenario of attaching endpoints to an ACI fabric, which is a task often automated by our customers, especially in cases of onboarding multiple endpoints at once.

Task 2: Deployment of the updated website containers using Ansible

[Estimated time: 35-40 minutes]

As tested in the first task, the website is up and running in our production environment. Meantime, our development team has worked on a newer version of the website, this time including an online shop, where customers will be able to buy products offered by our company.

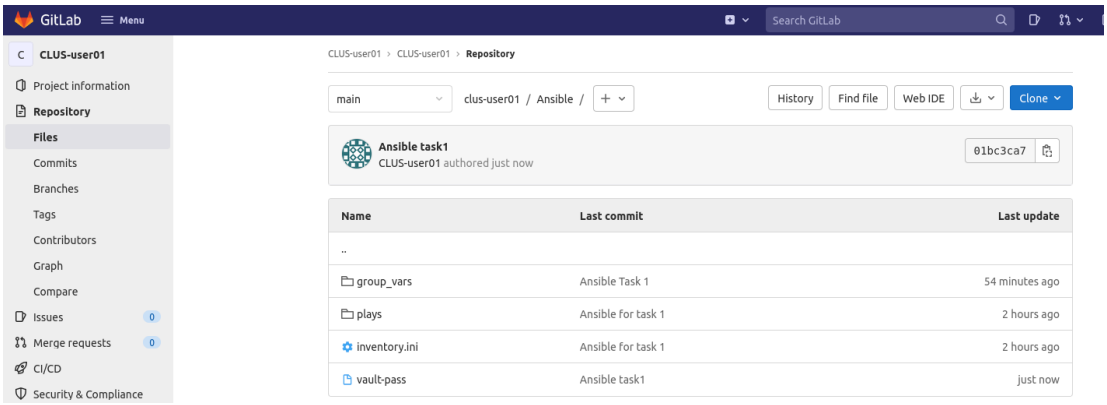
For that, we will be using the Ansible roles we've created, modifying the variables accordingly. In this task, we shall also start using multiple branches, environments, and pipelines, to further drive CI/CD principles for infrastructure management.



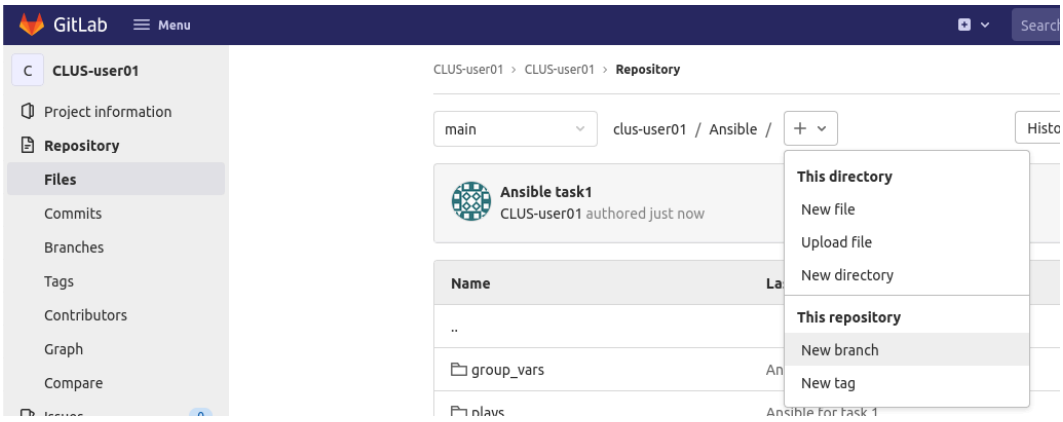
Step 1: Create a new development branch

The first change we'll be making will regard our repository – we'll be adding in branches. With CI/CD, branches are often used as a mechanism to trigger the pipelines – in our case, we will have a development branch where we're actively working on the code, then once we're done, we will use a feature to merge our development branch with a master branch. Upon a merge request a pipeline will get triggered and if all stages pass, the code will be deployed to our production environment.

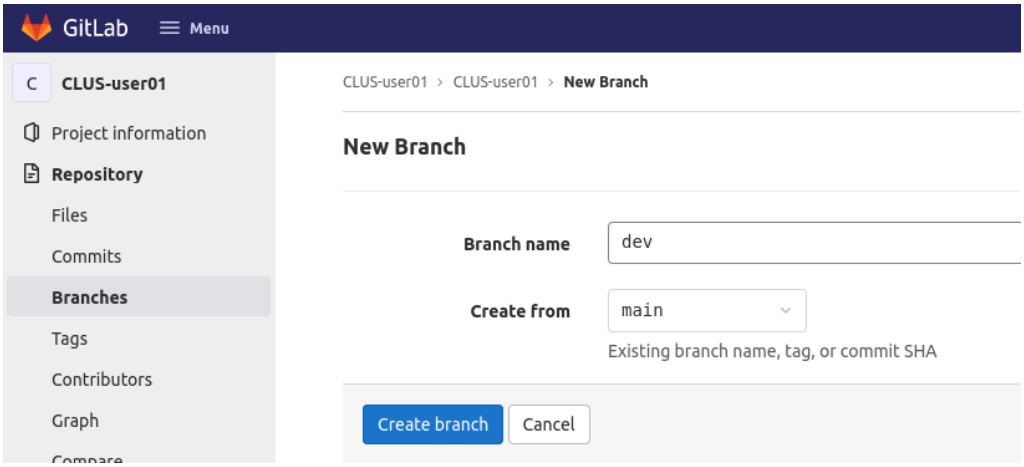
1. Go to http://gitlab.ruisoft.com and into your repository.



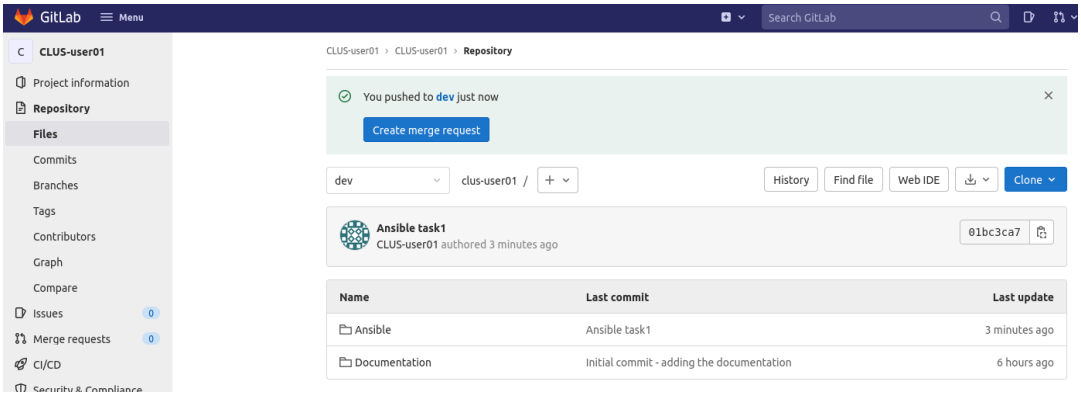
2. To create a branch, click on the "+" button and under the *This repository* section select the *New branch* option.



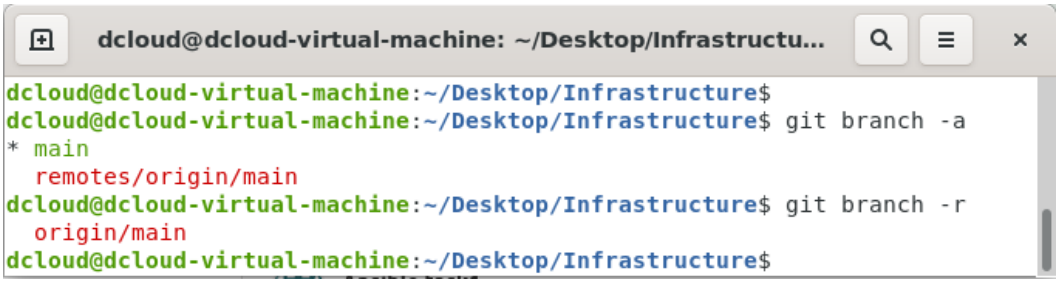
3. In the *New branch* screen, enter the name for branch and click on the *Create branch* button, for example *dev.* Specify that you want to create the branch using the code base present in main.



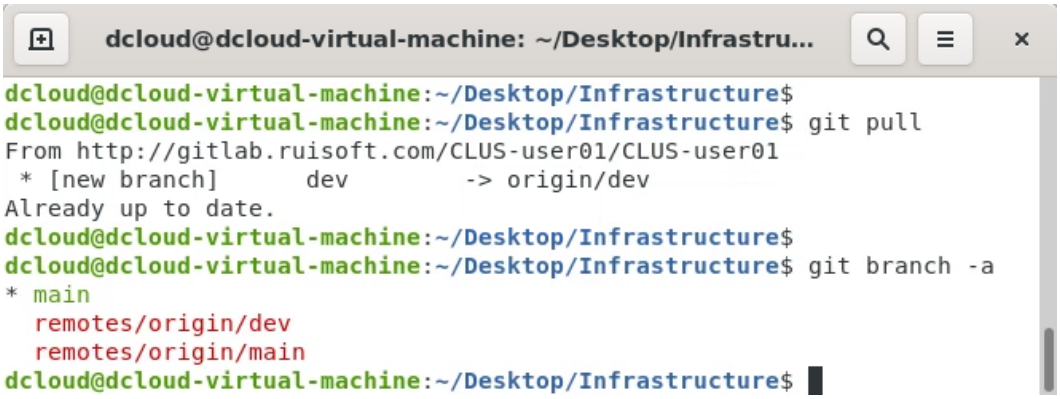
4. Verify that the branch is in place on your remote repository.



5. On your User VM, inside of your local repository, run the following commands to verify the branches present locally:
git branch -a
git branch -r

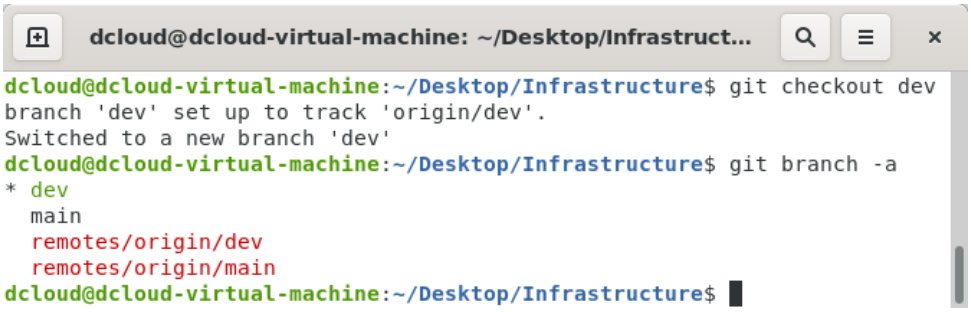


6. Due to the fact the branch you've created is not present locally, run a *git pull* command for it to populate:



7. You now need to set your local branch to the newly created *dev* one – this way you'll achieve the separation of code between dev and main branches, you'll only be working on the dev branch alone. To switch to another branch, you run the following:

   git checkout <branch name>



Step 2: Set up a pipeline

We will now start utilizing the CI/CD in our repository. CI/CD (short for Continuous Integration/Continuous Delivery) is continuous method of software development. With it, you build, test, and deploy iterative code changes. Because of that, you reduce the chance of developing new code based on buggy or failed previous versions. With this method, you strive to have less human intervention or even no intervention at all, from the development of new code until its deployment.

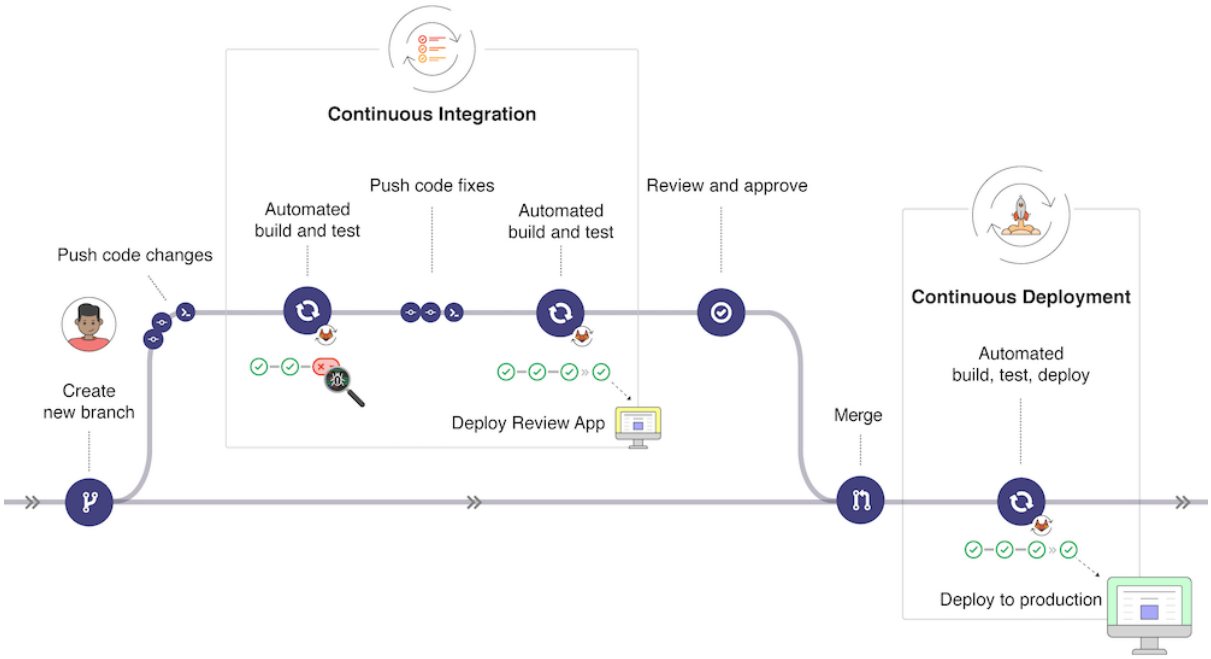
With CI/CD, for every push to the repository, you can create a set of scripts to build and test your application automatically. These scripts help decrease the chances that you introduce errors in your application.

Continuous Integration means that you build and test your code automatically and continuously. These tests ensure the changes pass all tests, guidelines, and code compliance standards you established for your application.

Continuous Delivery is a step beyond Continuous Integration. Not only is your application built and tested each time a code change is pushed to the codebase, but the application is also deployed continuously. Continuous Delivery checks the code

automatically, but it requires human intervention to trigger the deployment of the changes manually.

There's also Continuous Deployment, which is a step beyond Continuous Integration, like Continuous Delivery. The difference is that instead of deploying your application manually, you set it to be deployed automatically. Human intervention is not required.



Continuous Integration part of CI/CD is achieved by pipelines. These are the top-level component of continuous integration, delivery, and deployment.

Pipelines comprise of:

- Jobs, which define *what* to do. For example, jobs that compile or test code.
- Stages, which define *when* to run the jobs. For example, stages that run tests after stages that compile the code.

In Gitlab, the jobs are executed by runners. Multiple jobs in the same stage are executed in parallel if there are enough concurrent runners.

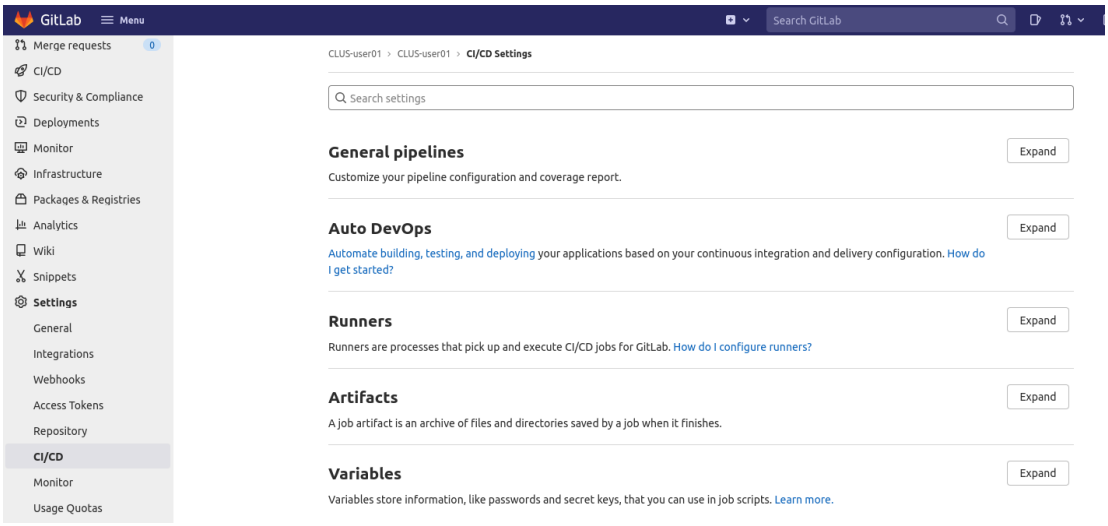
If *all* jobs in a stage succeed, the pipeline moves on to the next stage.

If *any* job in a stage fails, the next stage is not executed, and the pipeline ends early.

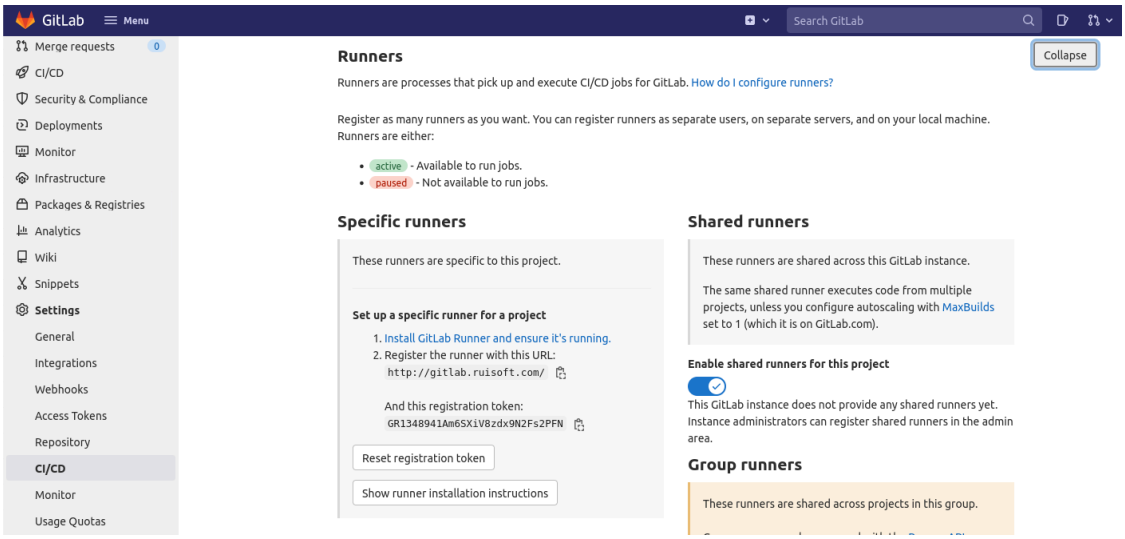
Besides Gitlab, there are different tools used for CI/CD, such as Jenkins for example.

To proceed with CI/CD for your project, we'll need to cover runners. As mentioned earlier, a GitLab Runner is an application that works with GitLab CI/CD to run jobs in a pipeline. It has multiple supported installation methods – in our case, we've installed it as a Docker container running in the Gitlab Runner VM. Let's work our way to configure it now.

1. Go to your Gitlab project and open up the Settings -> CI/CD tab



2. Under Runners, click on Expand:



3. Set up a runner for your project by registering it. Connect to the Gitlab Runner VM in your dCloud session environment and open the terminal. We are running our Gitlab Runner as a Docker container on that VM. Therefore, in the terminal, run the following commands:

sudo chmod 666 /var/run/docker.sock

docker ps

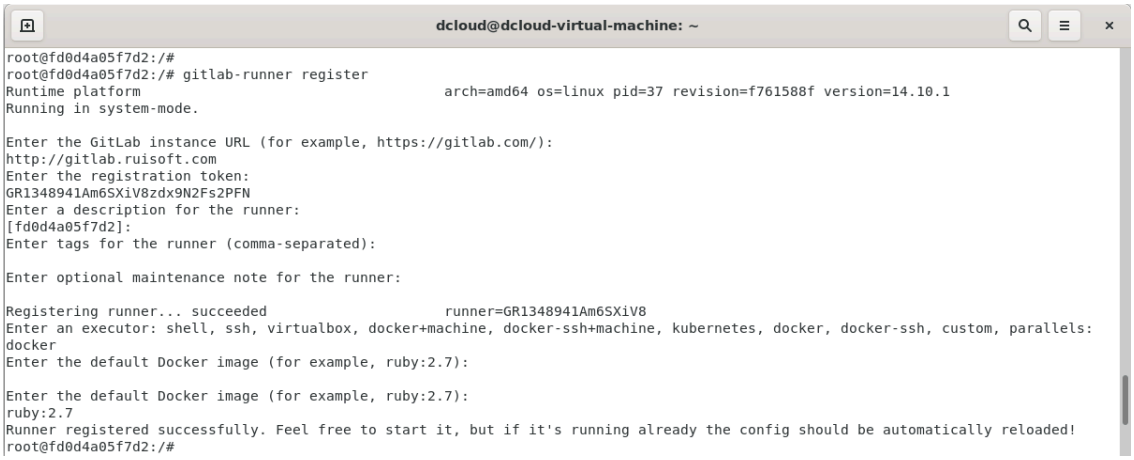
4. Access the bash of the Runner container by utilizing the following command:
   docker exec -ti gitlab-runner bash

5. Edit the /etc/hosts file inside the gitlab-runner container to point the container to the gitlab.ruisoft.com hostname:

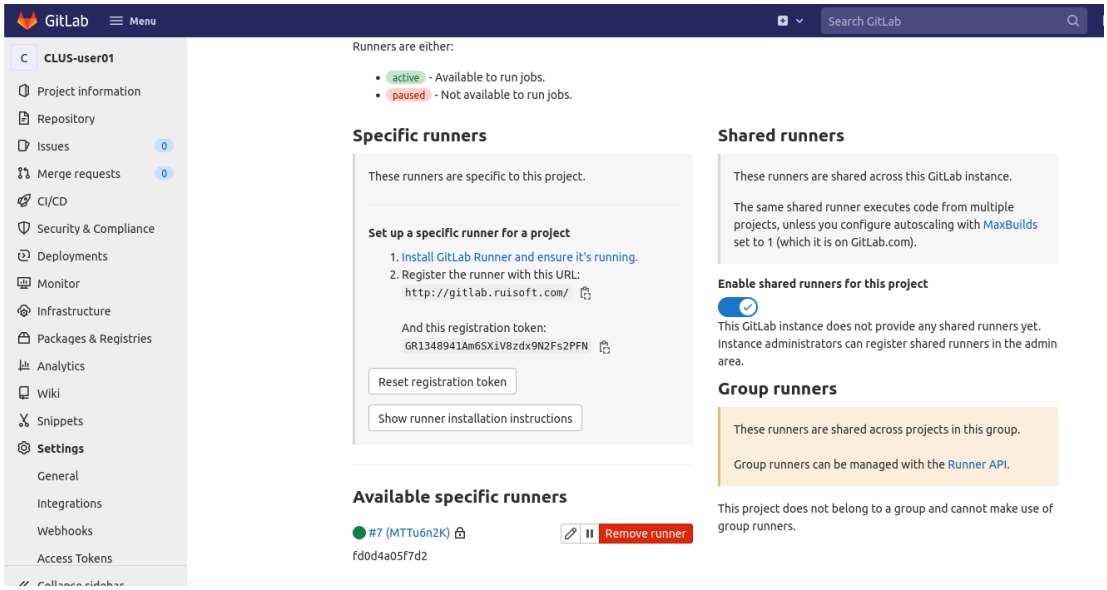
```
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      fd0d4a05f7d2
198.18.133.199 gitlab.ruisoft.com
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"etc/hosts" 8L, 208C
```

6. Register the Runner with the following command:
   gitlab-runner register

   Provide the Gitlab URL (http://gitlab.ruisoft.com), the registration token as seen in your CI/CD settings, select the *docker* executor and the *ruby:2.7* image.



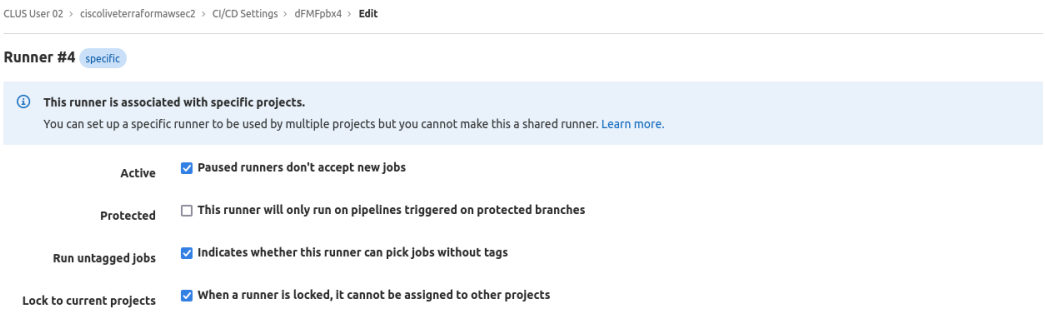
7. Verify the runner being seen as available in your repository:



8. Go to the Runner container bash once again and edit the /etc/gitlab-runner/config.toml file to add a *clone_url* property pointing to the Gitlab server under your registered runner, as follows:

```
[[runners]]
  name = "fd0d4a05f7d2"
  url = "http://gitlab.ruisoft.com"
  token = "MTTu6n2KRcyjqFeQNajM"
  executor = "docker"
  clone_url = "http://198.18.133.199"
  [runners.custom_build_dir]
  [runners.cache]
    [runners.cache.s3]
    [runners.cache.gcs]
    [runners.cache.azure]
  [runners.docker]
    tls_verify = false
    image = "ruby:2.7"
    privileged = false
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache"]
    shm_size = 0
root@fd0d4a05f7d2:/#
```

In case the tasks are not executing, there may be another default setting that needs to be modified. By default, runners only pick up 'tagged' jobs and this may be the reason your pipeline is not being picked up by this available runner. To change this, select the pencil icon right next to your runner that you can see above. Tick the 'Run untagged jobs' check box and save changes.



You now know what does Gitlab use to run the CI/CD, but where do we define what stages and jobs do we want to use in our pipelines? That happens in a .gitlab-ci.yml file.

9. Empty your current *Infrastructure/Ansible* directory and copy the contents of *WorkingScripts/Task2.3*, as follows:

```
rm -rf ~/Desktop/Infrastructure/Ansible/*
cp -r ~/Desktop/WorkingScripts/Task2.3/* ~/Desktop/Infrastructure/Ansible/
```

10. Create *.gitlab-ci.yml* in the root of your local repository (~/Desktop/Infrastructure/)

    Inside of that file, start by providing an image for the runner. We've prepared an image tagged *sabaroof/ciscolive2022:runner*

```
image: sabaroof/ciscolive2022:runner
```

    Then, create a 'before_script' section, which declares what we need to do on the runner before even beginning the first step of the pipeline. In our case we want the runner to update its aptitude, verify whether Ansible and Ansible-Lint have been installed and edit an environmental variable for Ansible to not have require an SSH key for the connections to the hosts defined in inventories. Such section will look like this:

```
before_script:
  - apt-get update -qy #update system
  - ansible --version
  - ansible-lint --version
  - export ANSIBLE_HOST_KEY_CHECKING=False
```

    Then, define our stages. In our case, the stages we'll be using are the verify, pre-deploy and deploy stages.

```
stages:
  - verify
  - predeploy
  - deploy
```

    Next, we reach the verify stage of our pipeline. Here, we will want to make sure that our code syntax is correct and that we follow best practices when working with Ansible. For that, we'll be using the Ansible-Lint package (which

checks the directory structure for .yml files and whether they follow best practices regarding formatting, Ansible configuration and so on), as well as a syntax check built into the ansible-playbook command as one of its parameters. We'll also specify rules on when to start said stage of the pipeline. Rules are using predefined runner variables – more on them here.

```
verify:
  stage: verify
  script:
    - ansible-lint Ansible/plays/task2.3-playbook.yml
    - ansible-playbook --inventory Ansible/inventory.ini --syntax-
check Ansible/plays/task2.3-playbook.yml
  rules:
    - if: '$CI_COMMIT_BEFORE_SHA ==
"0000000000000000000000000000000000000000"'
      when: always
```

Stage two of our pipeline is pre-deploy. Here, verify whether our production hosts are all up before trying to deploy to them.

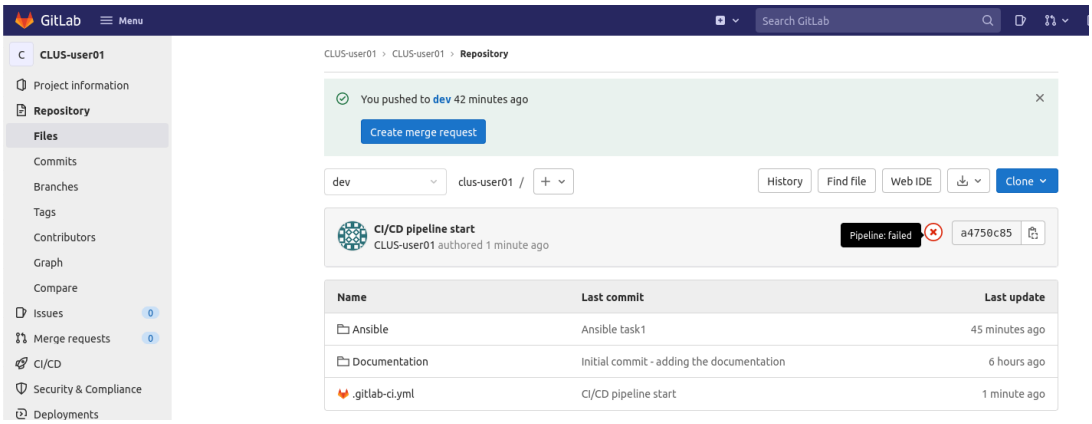
```
predeploy:
  stage: predeploy
  script:
    - ansible --inventory Ansible/inventory.ini all -m ping --vault-
password-file vault-pass --connection httpapi
  rules:
    - if: '$CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "main"'
      when: always
```

Finally, we have the deploy stage. This is where we'll run our playbook. Now here we will add 'when:manual' as we don't want it to automatically execute unlike the previous 2 steps of the pipeline. This will need manual intervention to execute and then the changes will be applied.

```
deploy:
  stage: deploy
  script:
    - ansible-playbook --inventory Ansible/inventory.ini
Ansible/plays/task2.3-playbook.yml --vault-password-file vault-pass
  rules:
    - if: '$CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "main"'
      when: manual
```

One obvious issue is that you will have to push the vault-pass file to your repository. In a real-life scenario if you ever do that, you'll want to make sure that your repository is private and pay attention to who has been granted access. Another solution would be to use a separate product for secrets management, such as Hashicorp Vault, where you'd be able to receive your secret after authenticating with a token.

11. Push the .gitlab-ci.yml file to the remote repository:



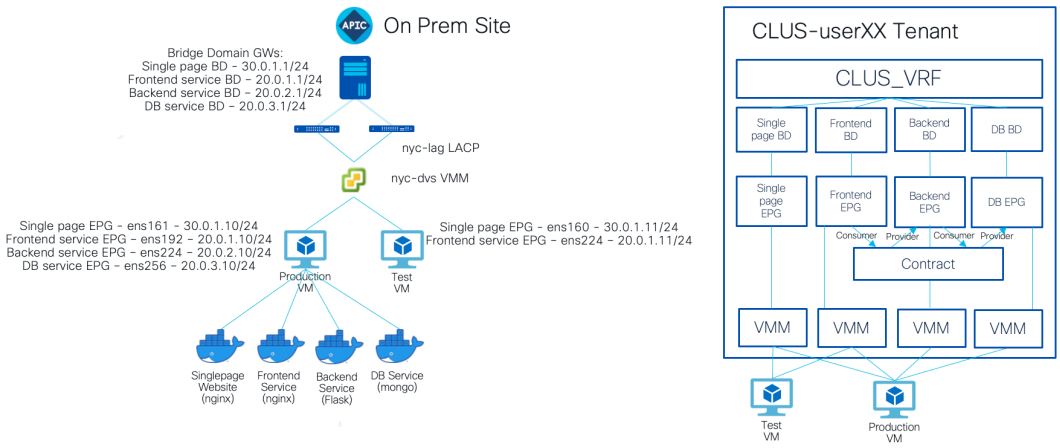
Step 3: Set up Ansible for three containers deployment in your pre-provisioned Tenant

1. Refer to the code you've copied from *WorkingScripts/Task2.3* and analyse it.

2. We create the objects necessary for the deployment of the new version of the website.

   All the access policies that we'll be using in this step have been already configured. For this step we need to pass the following objects inside of the Ansible code properties:

   - Your pre-provisioned tenant – ***CLUS-userXX***, where the X's stand for your user number
   - Your pre-provisioned VRF – ***CLUS-VRF***
   - VMM domain – ***nyc-dvs***, as our website-hosting endpoint is a virtual machine
   - VMM Enhanced LAG policy – ***nyc-lag***, as we're using LACP on the vDS uplinks
   - Three Bridge Domains, an Application Profile, three EPGs with the naming convention of your choosing, a contract, a subject, a filter, a filter entry, and IP addressing from the desired state topology below:



As you can see, you'll follow a very similar process as to the one from Task 1.

For conciseness of the code, we reuse the roles created in earlier tasks. You'll have to modify your group_vars files to reflect on the new objects that you'll be creating.

Role reusability can be for example achieved like this (this in an excerpt from the playbook file that we define for this task):

```
  - name: A Playbook for Task2 Step3 - the ACI object creation
part with role reusal
  hosts: apic
  gather_facts: false
  connection: httpapi

  roles:
    - { role: apic-object-creation, service: frontend }
```

With the according modifications made to the *plays/roles/apic-object-creation/tasks/main.yml:*

```
- name: Create a BD
  cisco.aci.aci_bd:
    host: '{{ apic_host }}'
    username: '{{ apic_creds.user }}'
    password: '{{ apic_creds.pass }}'
    tenant: '{{ apic_tenant }}'
    vrf: '{{ apic_vrf }}'
    bd: "{{ vars['apic_bd_' ~ item] }}"
    validate_certs: no
    state: present
  with_items: "{{ service }}"
```

For your reference, here's the directory structure that we've used when preparing the lab:

```
group_vars/
|  apic/
|  |     apic.yml
|  |     apic-password.yml
|  ||_    apic-password-plain.yml
|vcenter/
||     vcenter.yml
```

```
|  |     vcenter-password.yml
|_ |_    vcenter-password-plain.yml
plays/
|  apic-object-creation-playbook.yml
|  service-deployment-playbook.yml
|  task2.3-playbook.yml
|  roles/
|  |     apic-object-creation/
|  |     |     tasks/
|  |     |_    |_    main.yml
|  |     contract-creation/
|  |     |     tasks/
|  |     |_    |_    main.yml
|  |     service-deployment/
|  |     |     tasks/
|  |     |_    |_    main.yml
|  |     vm-pg-attachment/
|  |     |     tasks/
|_ |_    |_    |_    main.yml
vault-pass
inventory.ini
```

As you might've noticed, we've introduced two new roles – *vm-pg-attachment*, where we've now added the parts using the *community.vmware* collection modules needed to assign a portgroup to a VM, and the *contract-creation* role, to create the contract, subject, filter and filter entry objects and assign them to the provider and consumer EPGs.

This time, we will be using Docker Compose to launch the three services necessary for the new version of Ruisoft website to run. The code for Compose is stored in a public Github repository (https://github.com/sabaroof/cisco-live-2022) which you need to clone. Now, after finishing the next Step you will connect to the Production VM and run the Docker Compose yourself by doing *docker-compose up -d.*

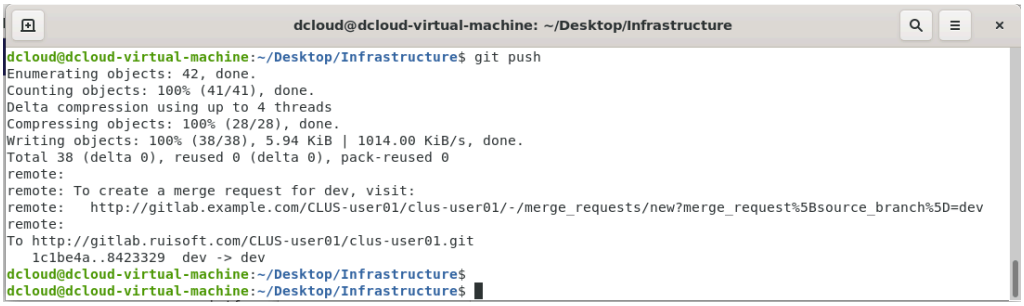
After modifying the commented lines of code, do not run the playbook! We will be deploying the code via a pipeline, in the next Step.

3. Once again, using vault, encrypt the plain password files (refer to the previous task again if you've forgotten how to do this).

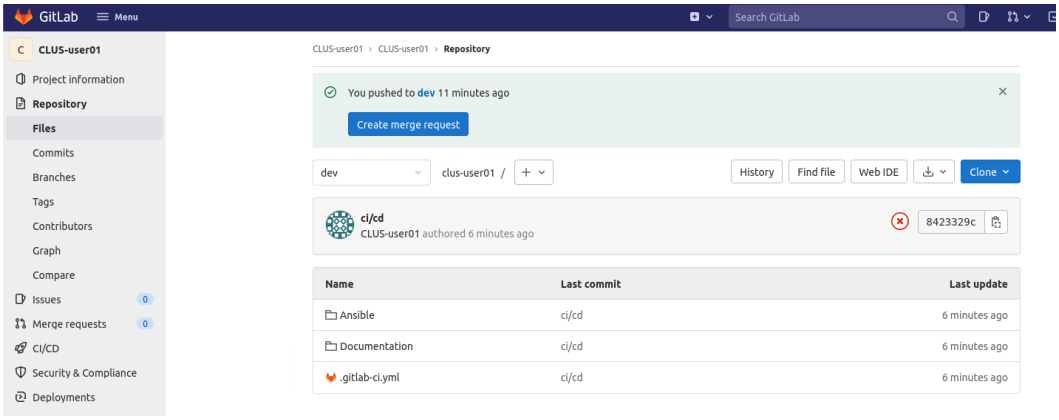
Step 4: Run the Ansible code against your pipeline

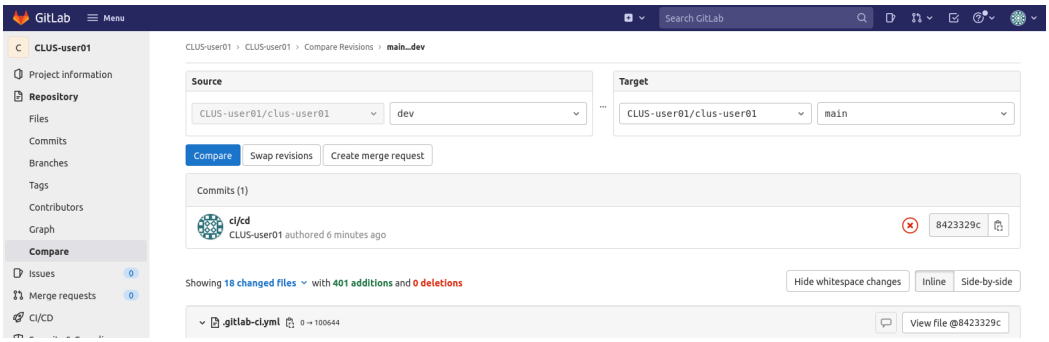
1. Push your code to the remote repository – make sure you're on the *dev* branch.

```
git status
git add .
git commit -m '<your message>'
git push
```

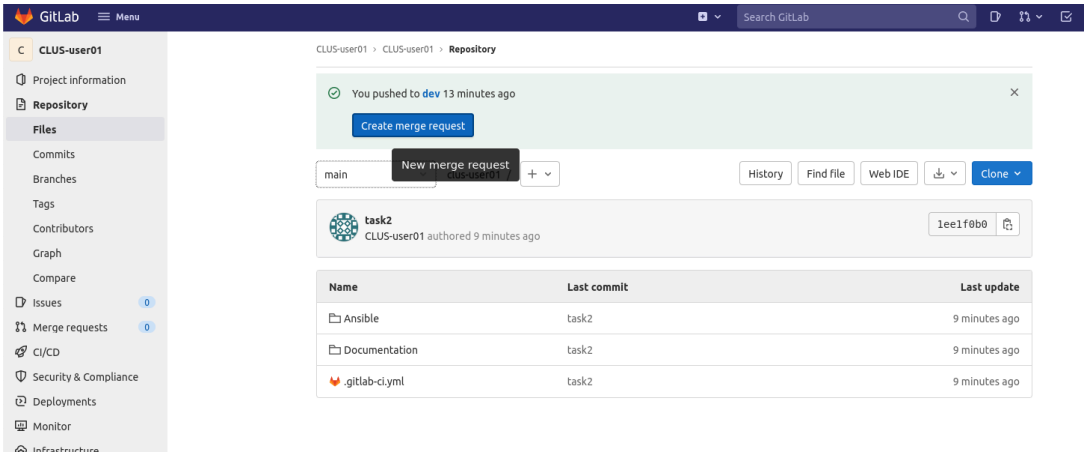


2. Go to your Gitlab project and check the state of the dev branch. Notice differences between it and main.

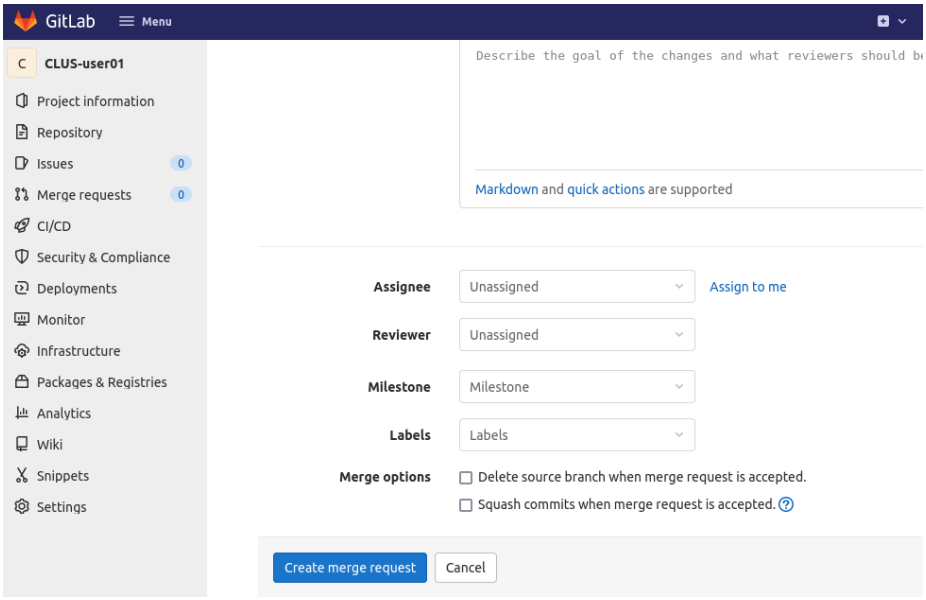




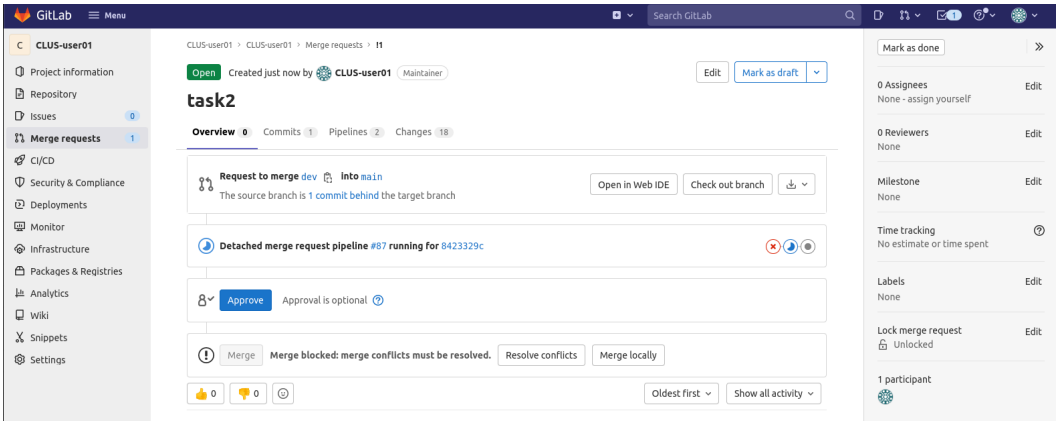
3. Hit on merge request:



4. In the merge request options select a name for the request, you can give it a description to indicate the changes being made. In real life scenarios you'd usually assign a teammate that would review your code, among other settings – you don't need to fill these out. DO NOT select the option to remove the branch upon a successful merge, that setting must be unchecked.

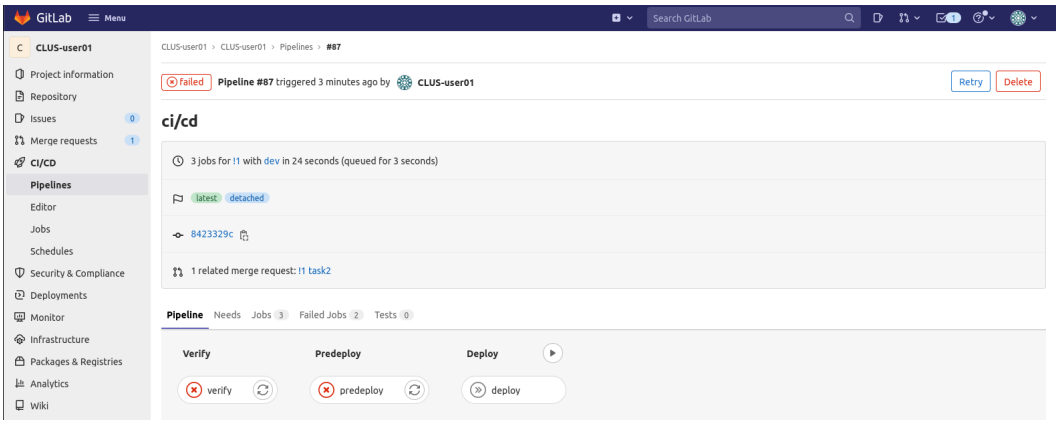


5. Hit on “Create merge request” and notice that the pipeline has commenced.



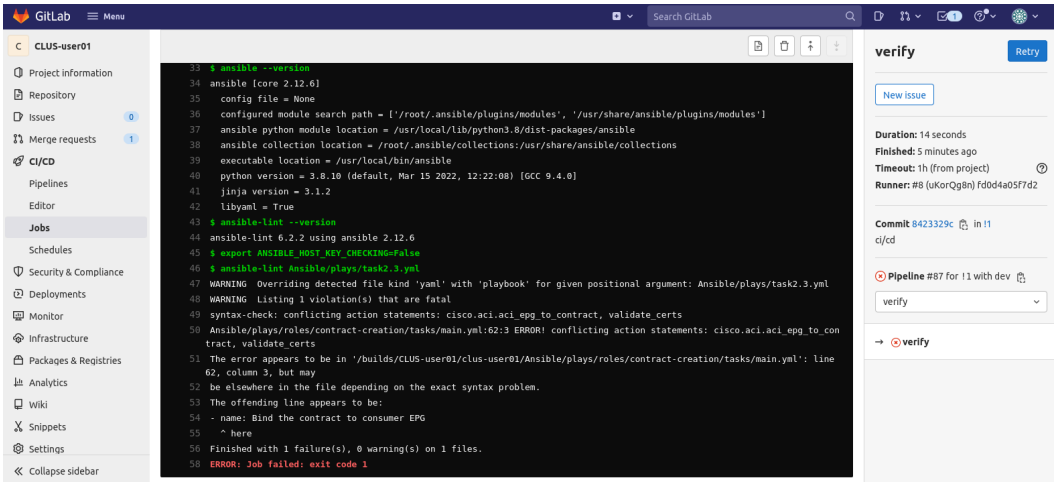
Step 5: Merge branches and deploy the code to the ACI fabric

1. In case you run into any issues with the pipeline, work on fixing it. To check the state of the jobs taken by the pipeline, go to CI/CD -> Pipelines and select one of the jobs:



You can also verify them from the Merge requests tab:

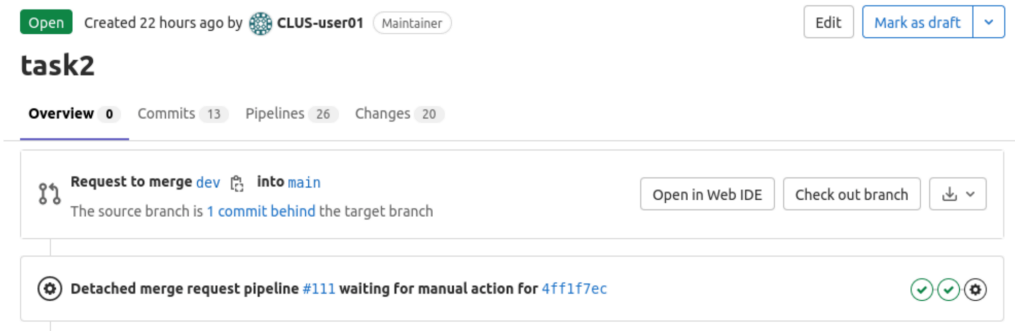
2. Select one of the jobs and check the reason for the pipeline failing. Work on fixing the issues. For example, the verify stage might throw an error during Lint execution:

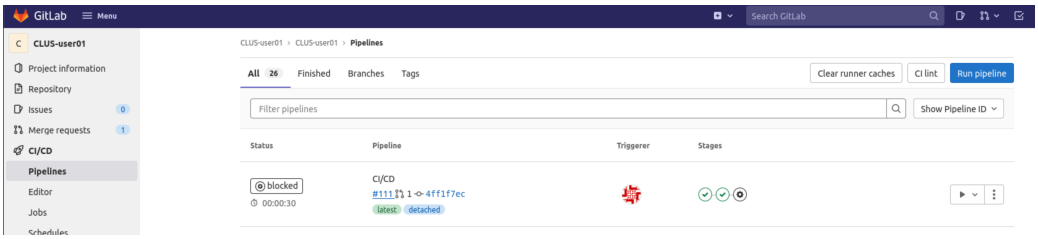


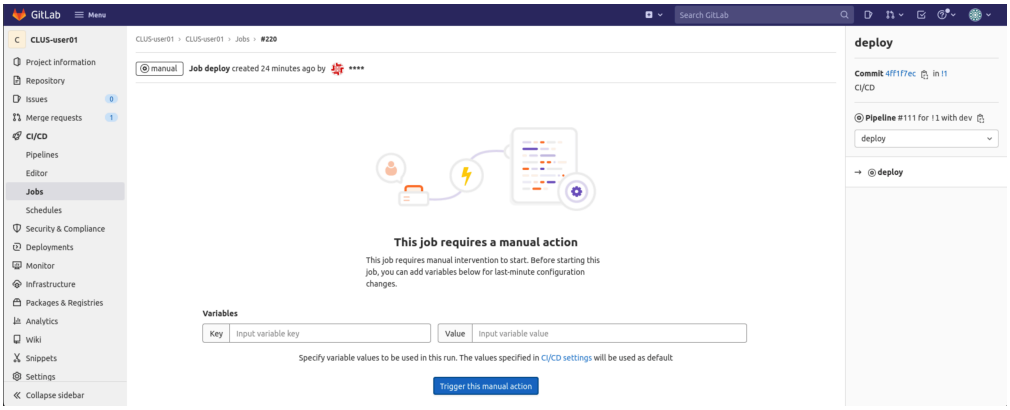
Work on fixing the code and push it to the *dev* branch. In case of pipeline job failure, iterate over this process until you pass the job.

If you're hitting many Lint errors and you're approaching the proposed time limit for this task, you can modify your *.gitlab-ci.yml* file to skip Lint in the *verify* stage of the pipeline by commenting out the respective line.

3. Once your pipeline completes the *verify* and *pre-deploy* stages, review the results and once you're done, approve the deploy stage by hitting the "Trigger this manual action" button.







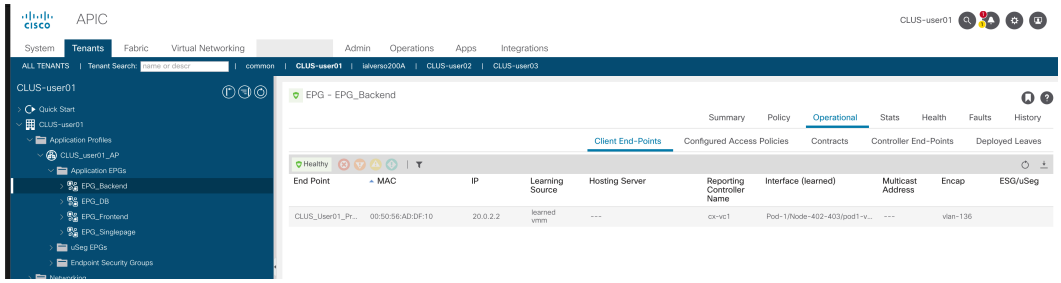
4. Connect to your Prod VM and run the following:

git clone https://github.com/sabaroof/cisco-live-2022.git (If you haven't already cloned it)
cd ~/cisco-live-2022/ciscolive-containers/
docker-compose up -d

```
root@appserver:~# cd cisco-live-2022/ciscolive-containers/
root@appserver:~/cisco-live-2022/ciscolive-containers# docker-compose up -d
Creating network "db_net" with driver "ipvlan"
Creating network "backend_net" with driver "ipvlan"
Creating network "frontend_net" with driver "ipvlan"
Creating ciscolive-containers_mongo_1 ... done
Creating ciscolive-containers_backend_1 ... done
Creating ciscolive-containers_web_1     ... done
root@appserver:~/cisco-live-2022/ciscolive-containers#
```

5. Upon merging branches and deploying the code, verify the following:

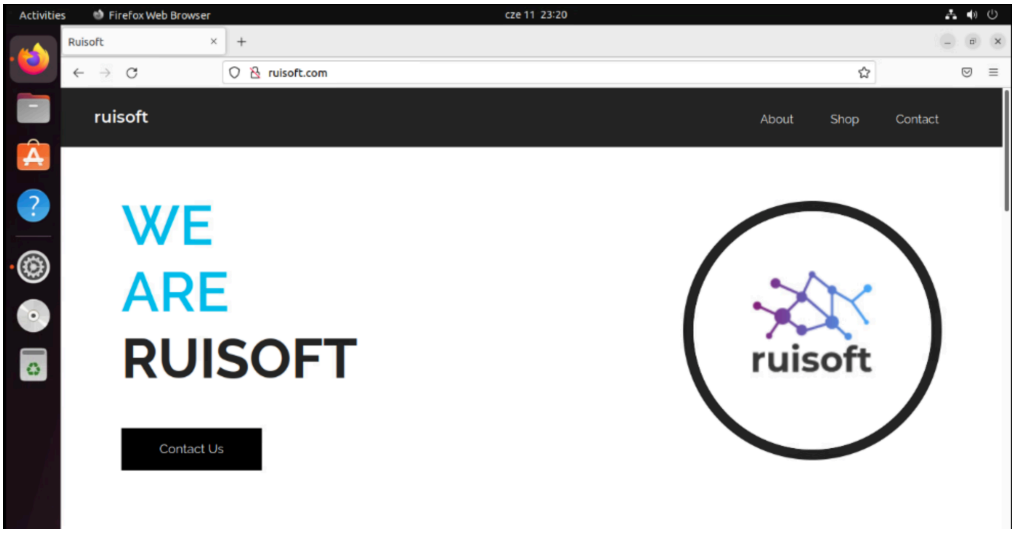
- Endpoints being learned under respective EPGs in ACI

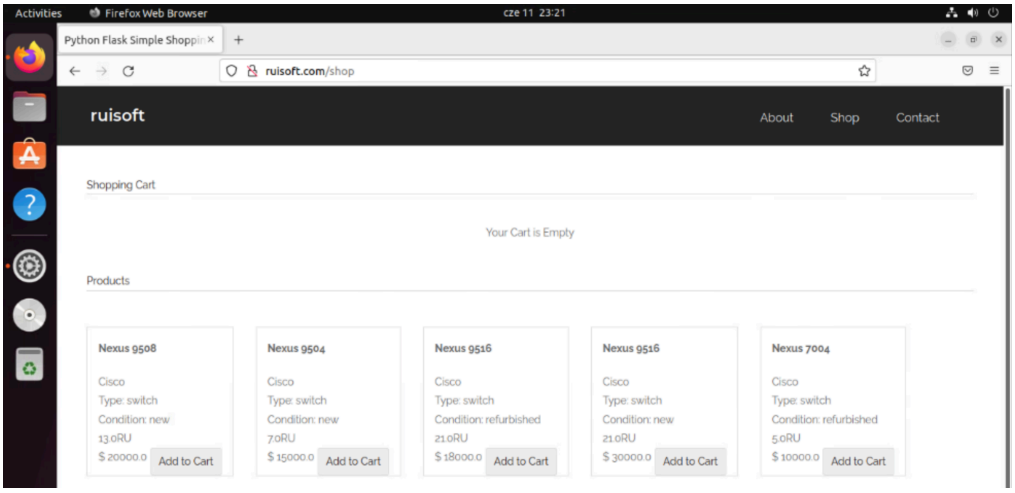


- Verification of containers standing up on Prod VM

```
root@appserver:~/CiscoLive/cisco-live-2022/ciscolive-containers#
root@appserver:~/CiscoLive/cisco-live-2022/ciscolive-containers# docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED         STATUS
      PORTS     NAMES
1b1ffb7f122f   nginx                                "/docker-entrypoint.…"   2 minutes ago   Up 2 min
utes             ciscolive-containers_web_1
e02bbccd71fe   sabaroof/ciscolive2022:backend       "./server.py"            2 minutes ago   Up 2 min
utes             ciscolive-containers_backend_1
751127e776d2   mongo                                "docker-entrypoint.s…"   2 minutes ago   Up 2 min
utes             ciscolive-containers_mongo_1
6a28c24782b8   sabaroof/ciscolive2022:singlepage    "nginx -g 'daemon of…"   3 days ago      Up 3 day
s                singlepage
root@appserver:~/CiscoLive/cisco-live-2022/ciscolive-containers#
```

- Ability to connect to the website from Test VM





- **What did we learn in this Task?**

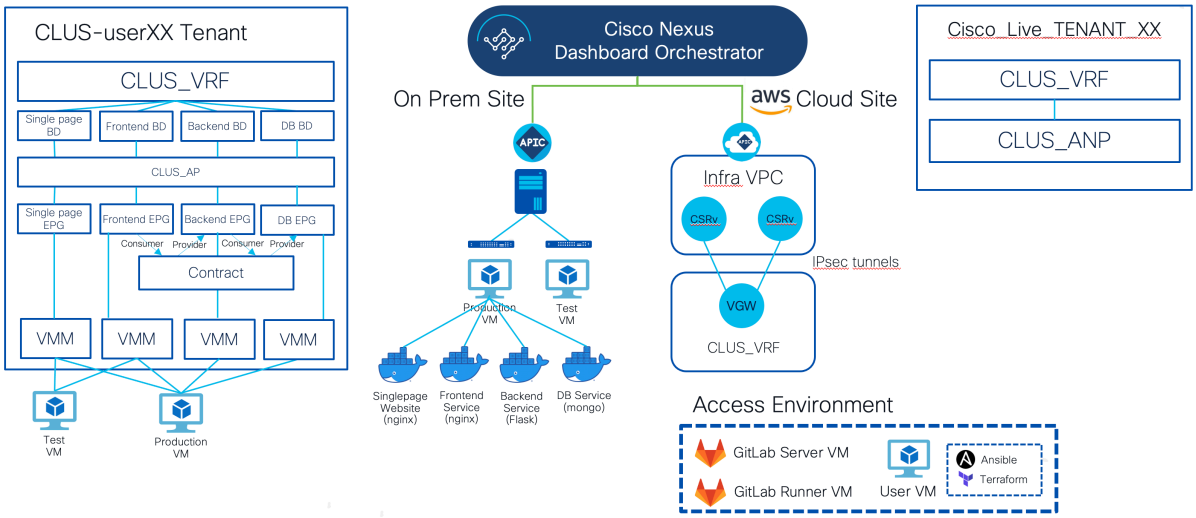
  - We've learned about the CI/CD approach to code and infrastructure management, bridging the gap between development and operations.
  - We now know the basics of how to prepare Gitlab to support pipelines and how to define them.
  - This leads us to an iterative approach to testing and development, regarding the infrastructure. We can deploy from a single point, with every dependency that we need.

Task 3: Create the Cloud infrastructure on AWS

[Estimated time: 20-30 minutes]

Now that we have tested that our application works successfully in a VRF on prem, we will start creating the cloud site which will eventually be used for load balancing and DR scenarios. The approach could have also been to start with creating the service-based application in the production VRF on the premises and stretching it to the cloud, but we want to explore some key features of the integration between the cloud and premises before that to better understand how AWS integrates with ACI.

Ruisoft is looking to host all future applications in the cloud due to application slowness that several users complained about in the past during traffic peaks. It is now your responsibility to be the pioneer of this effort by setting up a cloud site in a way that can be integrated with our Ruisoft website and sometime in the future host our application on it.

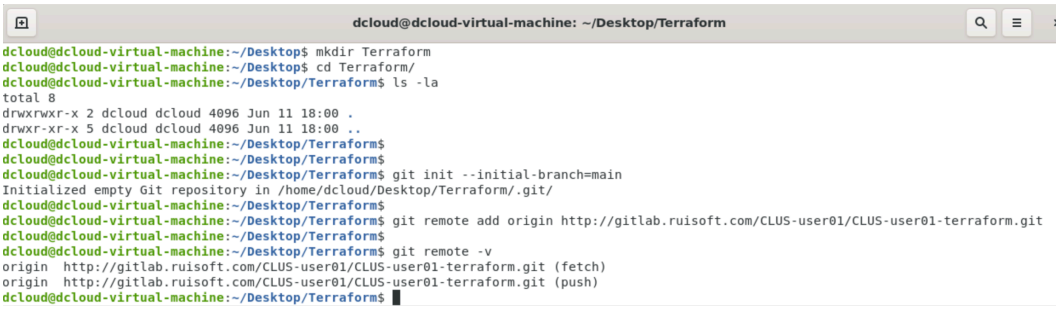


Step 1: Prepare the CI/CD pipeline that will verify the NDO objects responsible for stretching to the Cloud ACI site

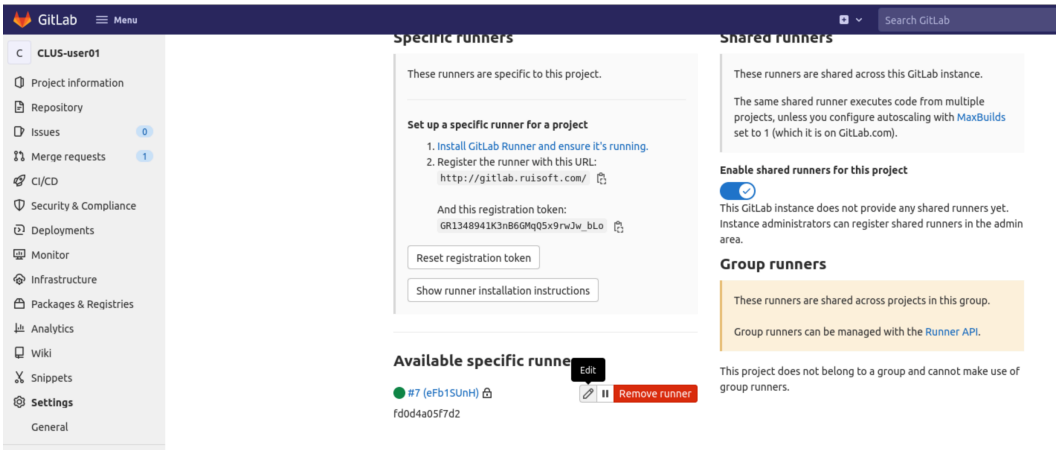
This time, we'll build a pipeline using Terraform. It will generally have 3 stages – a *terraform validate*, *terraform plan* and a *terraform apply*. The terraform validate stage will be a configuration and syntax check. Terraform plan will be performing a 'dry run' of the desired configuration by contacting the devices and checking if our script would work. Terraform apply is the stage where the actual configuration is applied. If you're new to Terraform, please refer to our Theory Guide, where we explain these stages in more detail.

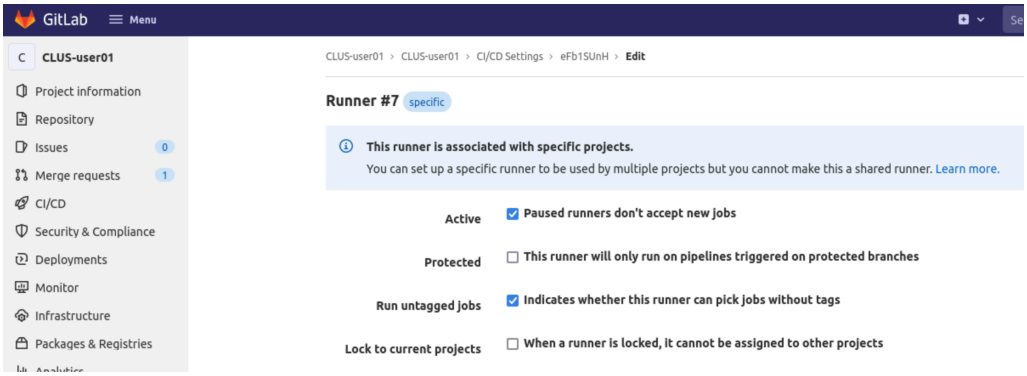
1. Create a new Gitlab project in the same fashion as you did in Task 1.

2. Create a new local directory *~/Desktop/Terraform,* initialize a Git repository there and add the remote. You can also again configure a token for authorization if you like (else just use a password)

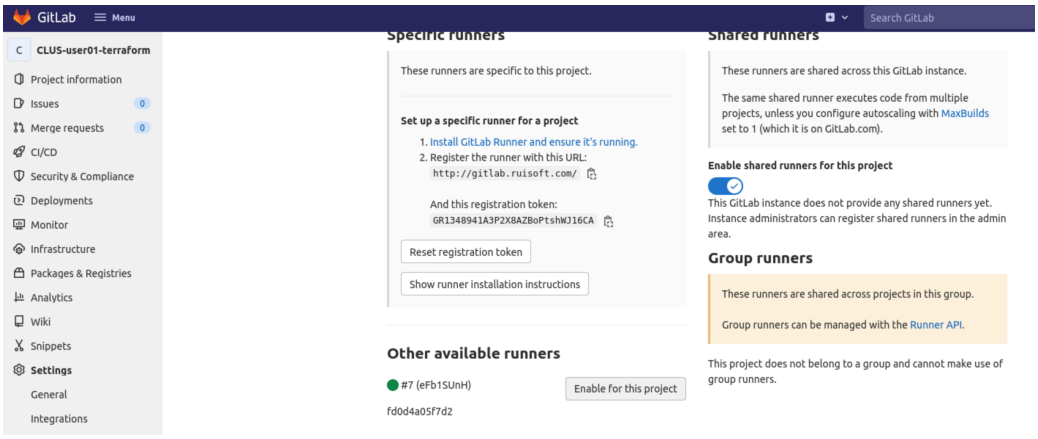


3. In the project, register a runner. Instead of following the same process as in Task 2, go to your previous repository, go to Settings -> CI/CD, expand on Runners and edit your registered runner. De-select the option "Lock to current projects". Save changes.





4. Go to back to your new project and verify if the shared Runner is visible. Select the “Enable for this project” option.



5. Create a *.gitlab-ci.yml* file in the root of your new Git repository.

6. Start by declaring our stages – namely validate, plan and apply as follows:

```
stages:
-  validate
-  plan
-  apply
```

7. Define the image for the runner to execute this pipeline. We need to mention that we need terraform and where to run this script in the following way:

```
image:
  name: hashicorp/terraform:light
  entrypoint:
    - '/usr/bin/env'
    -
'PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin'
```

8. Create the *‘before_script’* section. It is here where we’ll define the *terraform init* command, in turn downloading the Terraform dependencies:

```
before_script:
  - rm -rf .terraform
```

```
  - terraform --version
  - terraform init
```

We will also remove any .terraform files if we have pushed anything incorrectly, as to not interfere with the running of Terraform.

9. Define the *validate* stage:

```
validate:
  stage: validate
  script:
    - terraform validate
```

10. Define the *plan* stage. Remember that this stage depends on a successful *validate* stage:

```
plan:
  stage: plan
  script:
    - terraform plan
  dependencies:
    - validate
```

11. Lastly, define the *apply* stage, which depends on a successful *plan* stage. Like the previous task, we will specify *'when:manual'* to indicate that we want this stage to execute only upon manual intervention.

```
apply:
  stage: apply
  script:
    - terraform apply -auto-approve -parallelism=1
  dependencies:
    - plan
  when: manual
```

When executing *terraform apply*, we will explicitly provide the *-parallelism=1* parameter. This is due to the MSO provider requirement, by default Terraform uses parallelism of 10, meaning it can concurrently create multiple resources, which would lead to issues in the NDO. We also set the *auto-approve* flag so that we don't have to explicitly hit yes, which is the default when running this command.

Therefore, our *.gitlab-ci.yml* file comes together with all the stages declared and set up the way we'd like. As you might notice, the syntax of this pipeline is simpler than it was for Ansible, and that's due to built-in Terraform features, making it a popular choice for an IaC solution.

Step 2: Prepare the Terraform script to create a tenant on the Cloud ACI

Let's get to making our script now. An important thing to remember about Terraform is providers. A Provider is the underlying code where you feed in your config. The Provider is the one that's going to take your script and communicate with APIC/NDO/vCenter/AWS etc. to configure them.

Hashicorp has several approved Providers which can be found here:

- Terraform Registry

1. Verify whether Terraform is successfully installed on your VM and if the version is suitable. Check this using:

   terraform --version

2. Copy the contents of *~/Desktop/WorkingScripts/Task3* into your current directory as follows:
   cp -r ~/Desktop/WorkingScripts/Task3/ ~/Desktop/Terraform*

3. The working script has all the parameters created except a Tenant itself, which you will be creating in the beginning of main.

4. The provider we will be using for these tasks is the ***mso*** provider. In the following resource you can find the documentation for it:
   - Terraform Registry - Cisco DevNet MSO Provider Documentation.

   Have a look at the documentation to see what the first step for the usage of this Provider and authentication with it is.

   Our *'version_providers.tf'* file specifies the required provider as well as provide authentication credentials to the provider.

```
#  Define versions

terraform {
  required_providers {
    mso = {
      source  = "ciscodevnet/mso"
      version = "= 0.5.0"
    }
  }
    required_version = ">= 1.1.2"
}
```

```
#  Define Providers; if using local ND user, comment out the domain.

provider "mso" {
  username = var.creds.username
  password = var.creds.password
  url      = var.creds.url
#  domain   = var.creds.domain
  insecure = "true"
  platform = "nd"
}
```

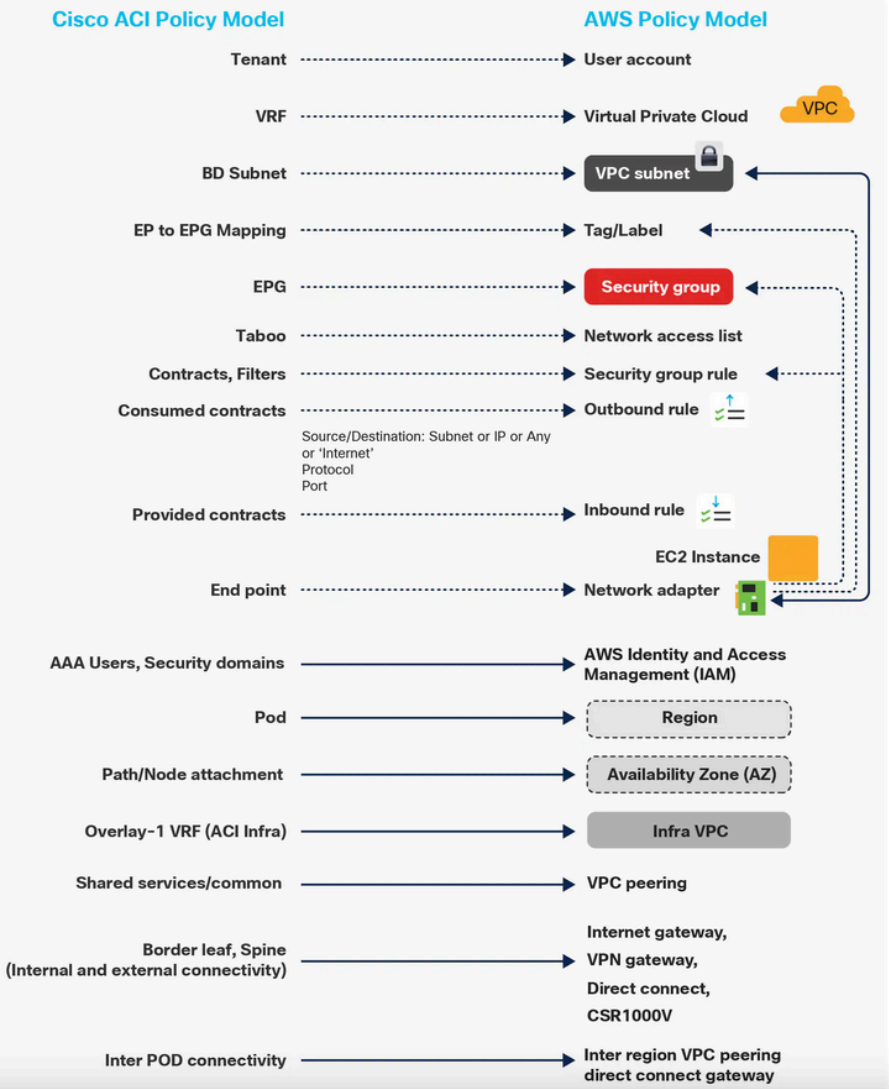
The variables used in *mso* provider will be explained shortly. We suggest using 0.5.0 version of the Cisco DevNet *mso* provider.

5. Before we move to create the components on the cloud site, let's understand how the Cloud APIC is set up and what components we are going to create.

   Let's login to the Cloud APIC.

   As you can see, the infra tenant has a separate AWS account and hence each user tenant also needs a separate AWS account associated with it

   This diagram is good to have handy when you perform integrations with the Cloud APIC and on-prem:

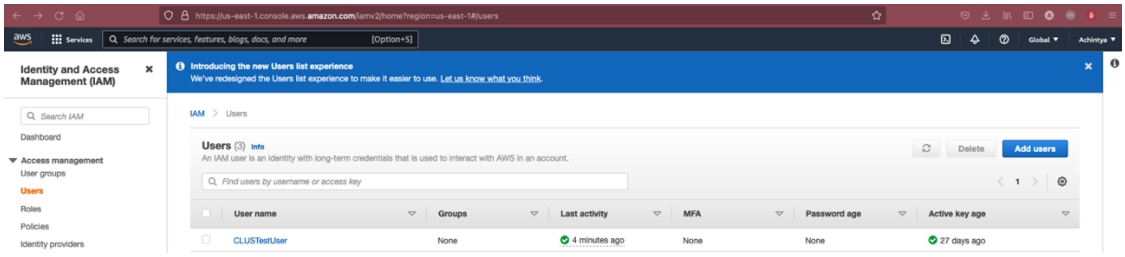


6. Let's create an AWS account with your personal email address. Head over to aws.amazon.com and enter your details.

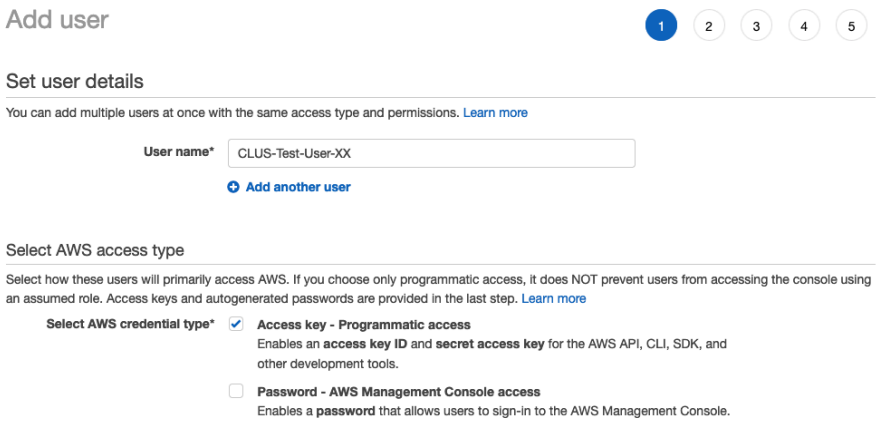
7. NDO, which we'll be configuring with Terraform, will be creating objects in your AWS account. Because of that, it'll need your AWS Account ID, access key and secret key. Let's prepare these values now.

   Your account ID can be found in the top right corner when you log into AWS and click on your username. Note it down.

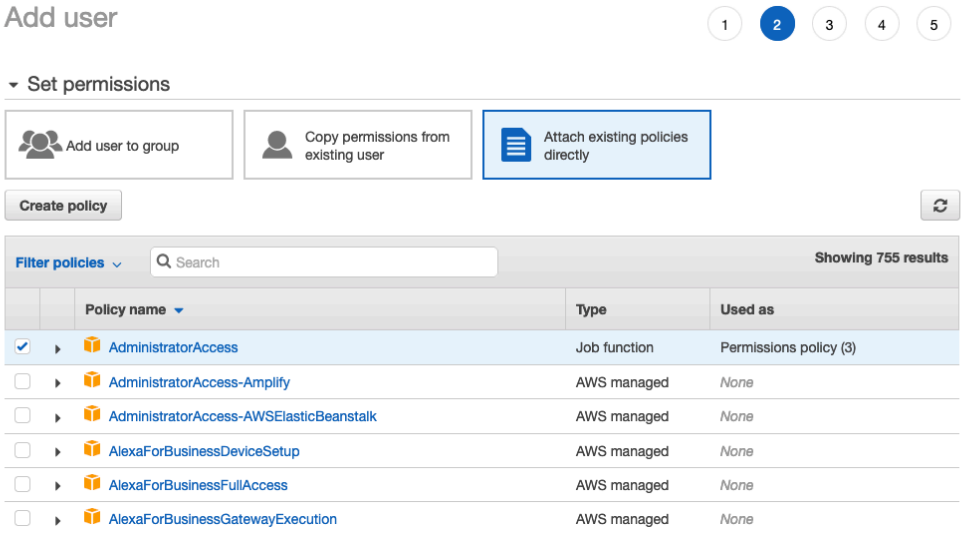
8. Prepare the access and secret keys by creating a user for the NDO to be able to program your AWS resources. Search for users in the search bar and navigate to the Users section in IAM:



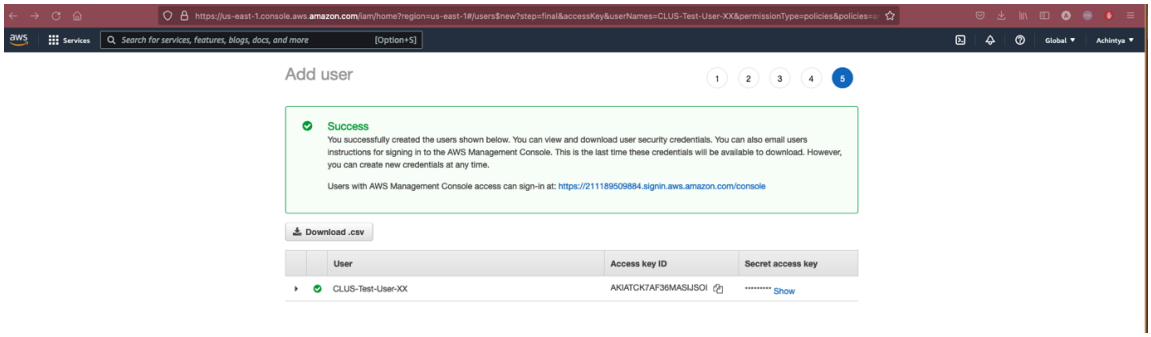
9. Click 'Add user' and create a user with programmatic access (we don't really need to log in to the AWS Console via these user credentials, it's only for the NDO):



10. In permissions, go to 'Attach existing policies directly' and select *AdministratorAccess:*



11. Skip the tags section. Review and create user. This will show you your Access Key and Secret Access key. Make sure you save these as they will never be displayed again if you leave this screen – it may be a good idea to download these as a CSV file.



12. Now, we provide Terraform with the AWS user account details, in the form of a variable that it can reuse.

    To do so, we created a *'override.tf'* file where we are populating all our sensitive variables for our main Terraform script that we will create shortly. In case you are pushing to a public repository you will then need to add *'override.tf'* file to your *.gitignore,* so that you don't compromise your keys and credentials.

    In the *'override.tf', we* create a variable that will contain the values you've noted in the previous step – your AWS Account ID, access key and secret key.

```
variable "awsstuff" {
  type = object({
    aws_account_id    = string
    aws_access_key_id = string
    aws_secret_key    = string
  })
  default = {
    aws_account_id    = "000000000000000000" #your account ID
    aws_access_key_id = "000000000000000000" #your access key
    aws_secret_key    = "000000000000000000000000" #your secret
  }
}
```

13. In addition, our *override.tf* file will also have our NDO login credentials (that we've referenced in *versions_providers.tf*) and our tenant details.

    We expect our final override.tf file to look like this:

```
#  Populate values based on your AWS values
variable "awsstuff" {
  type = object({
    aws_account_id    = string
    aws_access_key_id = string
    aws_secret_key    = string
  })
  default = {
    aws_account_id    = "000000000000000000" #your account ID
    aws_access_key_id = "000000000000000000" #your access key
    aws_secret_key    = "000000000000000000000000" #your secret
  }
}

#  Populate values based on your ND configuration
variable "creds" {
```

```
  type = map(any)
  default = {
    username = "CLUS-userXX"
    password = "CLUS-userXX-123!"
    url      = "https://198.19.202.12/"
  }
}

variable "tenant_stuff" {
  type = object({
    tenant_name  = string
    display_name = string
    description  = string
  })
  default = {
    tenant_name  = "Cisco_Live_TENANT_XX" # change XX to your
assigned user name
    display_name = "Cisco_Live_TENANT_XX" # change XX to your
assigned user name
    description  = " Terraform Created Tenant for user XX" # change
XX to your assigned user name
  }
}
```

14. Sensitive data's been defined in the *override.tf* file. We configure the variables used to create objects in the NDO. We followed a best practice, where we 'declare' the variables in the *variables.tf* file and then we provide the values for these declared variables in the *terraform.tfvars* file. This makes our code easily reusable as we would need to make changes in just one place.

    All variables are declared and used in the following format in *variables.tf*:

```
variable "schema_name" {
  type    = string
  default = "some_value"
}
```

    In *terraform.tfvars*, the values can directly be given and will be used as follows(for the same variable above):

```
schema_name = "Cisco_Live_SCHEMA_XX" # XX is your username
```

    Hence the default "some value" will no longer be used and instead "Cisco_Live_SCHEMA_XX" will be taken by Terraform. In case you did not populate the *terraform.tfvars* file, Terraform would ask you to input a value for each variable during runtime (plan or apply).

15. We proceed to create variables for NDO credentials, username, Tenant details, Schema information, template information, AWS site name, VRF name, ANP name, region, CIDR IP, subnets and zones.

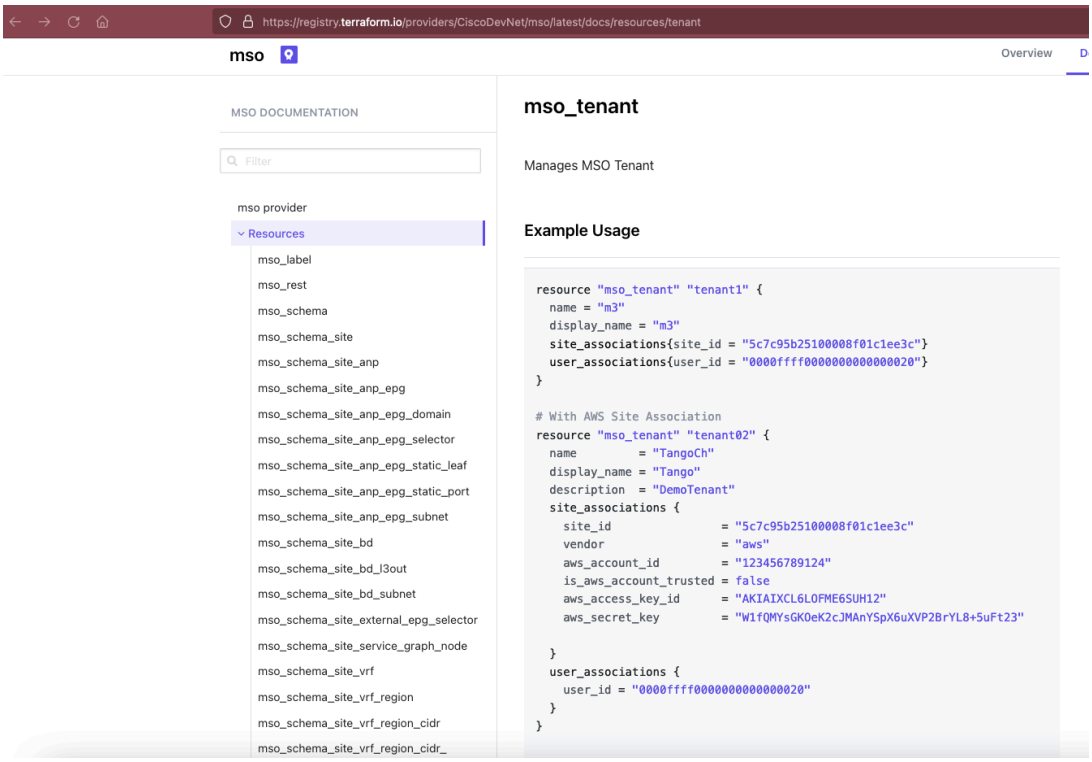
    We use the CIDR IP range of 150.0.0.0/16 and subnet ranges of 150.0.1.0/24, 150.0.2.0/24 and 150.0.3.0/24 all in the same zone (us-east-1b) to simplify our deployment.

NOTE: Remember that there are several users using the lab now and you would have to use a unique Tenant name, Schema name and Template name so there are no overlaps, and you don't end up configuring another user's cloud site by mistake. We will use the format: 'Cisco_Live_TENANT_XX', 'Cisco_Live_SCHEMA_XX' and 'Cisco_Live_TEMPLATE_XX' respectively, where XX is your assigned user ID from 01-17.

16. Now that we have all our variables in place, we can proceed to completing the *main.tf* file which will be creating the stretched tenant for us.

    Refer to the ***mso*** provider Terraform documentation:
    - Terraform Registry - Cisco DevNet MSO Provider Documentation.

    The only thing you need to create in the *main.tf* is a Cloud Tenant. This is something that has not been created for you. By checking the abovementioned link, we see that a *mso_tenant* resource looks like this:



    The first thing we want to do is to create a tenant resource for our cloud site (as you may already have noticed that these are not the values that we have initialised in the variables). We create a resource *mso_tenant* where we pass the details and variables to create our tenant. We will be using the resource variable name 'tenant' for this (and the rest of the code is already referencing this tenant). This part of the code will be right at the top of our main.tf.

How to reference variables, data sources and exposed values from resources:

- A variable is referred using var.variablename. In case it is part of a variable object block (like awsstuff above with the credentials) then it is referred to using ***var.objectname.variablename*** (i.e., var.awsstuff.aws_account_id).

- A Data source attribute is referred to using ***data.<data source name>.<data source object name>.<exposed attribute>***, i.e., a data source that you might create for a user as follows:

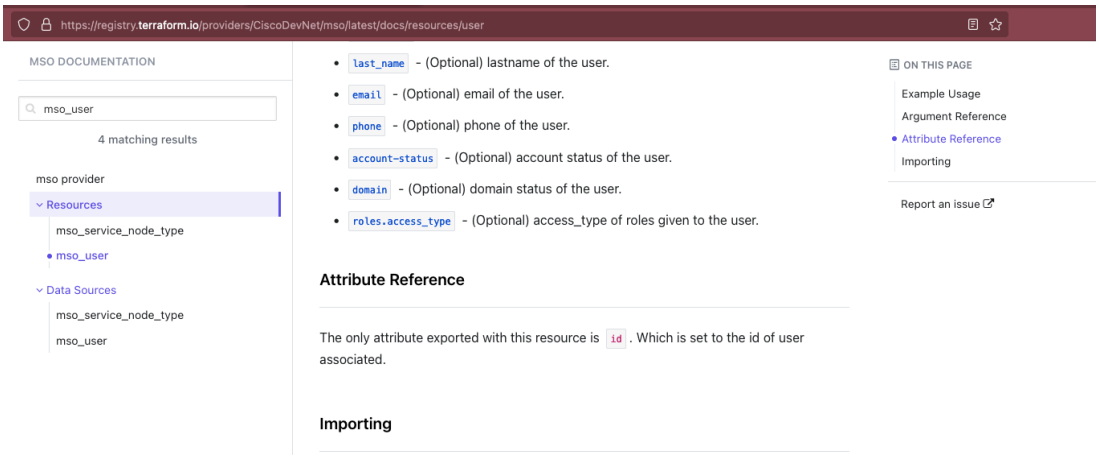
```
data "mso_user" "user1" {
  username = "CLUS-userXX"
}
```

We can obtain the user id of this user via data.mso_user.admin.id, where id is an exposed parameter. You can find all the exposed attributes for each data source in the Terraform MSO provider documentation here.

- A resource attribute is referred ***using <resource name>.<resource object name>.<attribute>***, i.e., if we create a user as the following:

```
resource "mso_user" "user1" {
  username = "name1"
  user_password = "password"
  first_name = "first_name"
  last_name = "last_name"
  email = "email@gmail.com"
  phone = "12345678910"
  account_status = "active"
  roles{
    roleid = "0000ffff0000000000000031"
  }
}
```

We may refer to the user id of the above "user1" using mso_user.user1.id. You can find this information also clearly documented in the Terraform mso provider documentation.



17. You can test if your dependencies are alright to be able to successfully create this tenant. Run the following commands:

    terraform init
    terraform validate
    terraform plan

    If this passes, we are good to proceed with configuring the rest of our *main.tf* file. You can also choose to 'terraform apply -parallelism=1' as well here to see what happens. Ideally, we want to push this script to our Gitlab repository each time there are changes and execute from there via a pipeline, but in case you are new to Terraform you can just test how an apply looks like. If you chose to apply, go over to the NDO and the Cloud APIC and verify if the tenant was created. Afterwards, revert to the original state by running *terraform destroy -parallelism=1* and continue with our script. Verify that we are in the state we started off with no tenant created yet in the NDO/cAPIC.

18. Have a look at how we have created other resources that are referencing the tenant in a similar manner. Create the following resources all referencing the previously created resources logically:
    mso_schema > mso_schema_site > mso_schema_template_vrf >
    mso_schema_site_vrf > mso_schema_site_vrf_region >
    mso_schema_template_anp > mso_schema_site_anp.

    Use the Terraform mso provider documentation (hyperlinks above) to understand how to populate each of these sequentially.

19. The last piece of config we created was to deploy a template using the Terraform mso provider resource 'mso_schema_template_deploy'. Here we make use of a special operator called 'depends_on' that can explicitly mention to Terraform that to create this template first verify that all the previous components are in place.

    This resource configuration would look like this:

```
resource "mso_schema_template_deploy" "template_deployer" {
  schema_id     = mso_schema.schema1.id
  template_name = mso_schema.schema1.template_name
  depends_on = [
    mso_tenant.tenant,
    mso_schema.schema1,
    mso_schema_site.aws_site,
    mso_schema_template_vrf.vrf1,
    mso_schema_site_vrf.aws_site,
    mso_schema_site_vrf_region.vrfRegion,
        mso_schema_template_anp.anp1,
  ]
}
```

You may now perform a final validate and a plan on the User VM to test if the code looks good. In case it does, move to the next step.

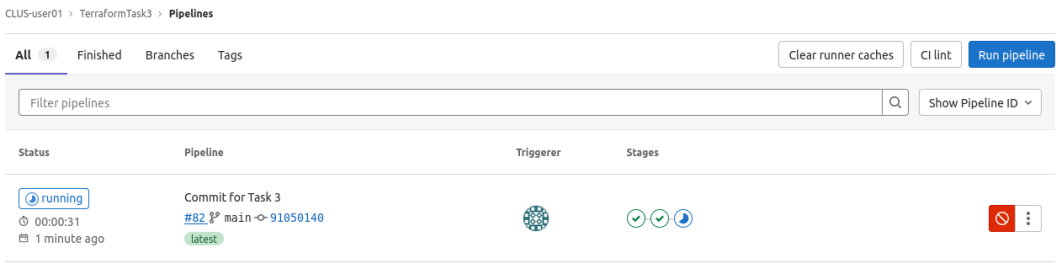
Step 3: Verify the Terraform script against your pipeline and execute it

1. Push your code to Gitlab and commence the CI/CD pipeline. It's important to create and modify your *.gitignore* file so that it ignores terraform state files or the terraform lock as this will stop the code from being executing on Gitlab. Our repo should look like this finally:

Name	Last commit	Last update
.gitlab-ci.yml	Commit for Task 3	45 seconds ago
main.tf	Commit for Task 3	45 seconds ago
override.tf	Commit for Task 3	45 seconds ago
terraform.tfvars	Commit for Task 3	45 seconds ago
variables.tf	Commit for Task 3	45 seconds ago
versions_providers.tf	Commit for Task 3	45 seconds ago

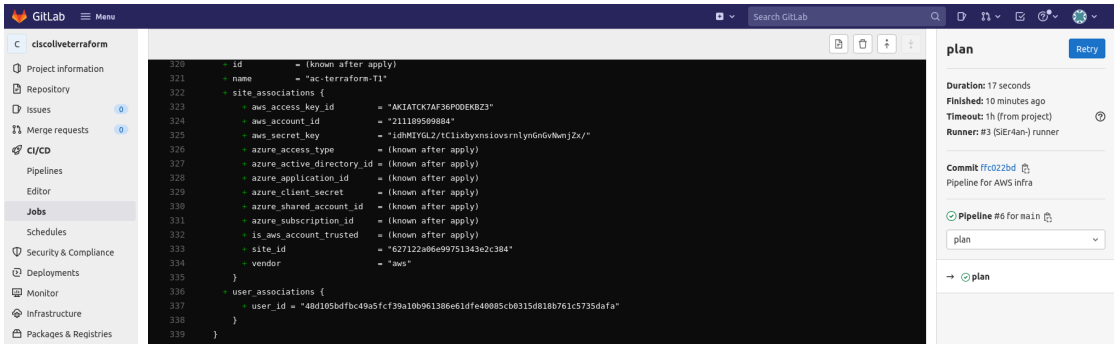
Our pipeline should have instantly begun execution as soon as we committed and pushed our scripts.

2. Proceed to the CI/CD menu under our project to check this.



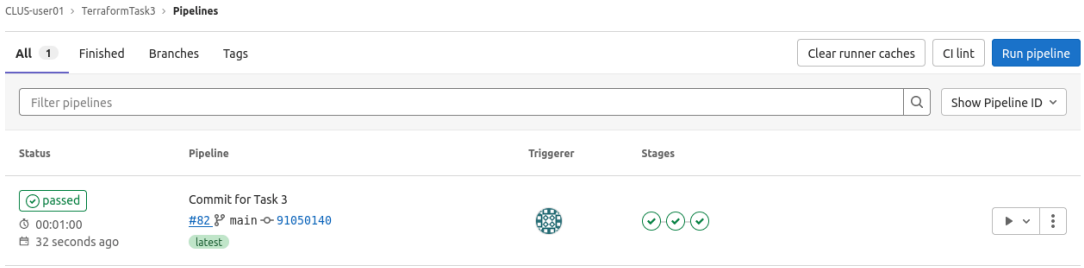
As you can see the pipeline has already begun execution. If you hover over the green ticks, you will see that the stages validate, and plan have passed.

3. Click on the plan and you will see what objects are going to be added and what changes will be made by the script.



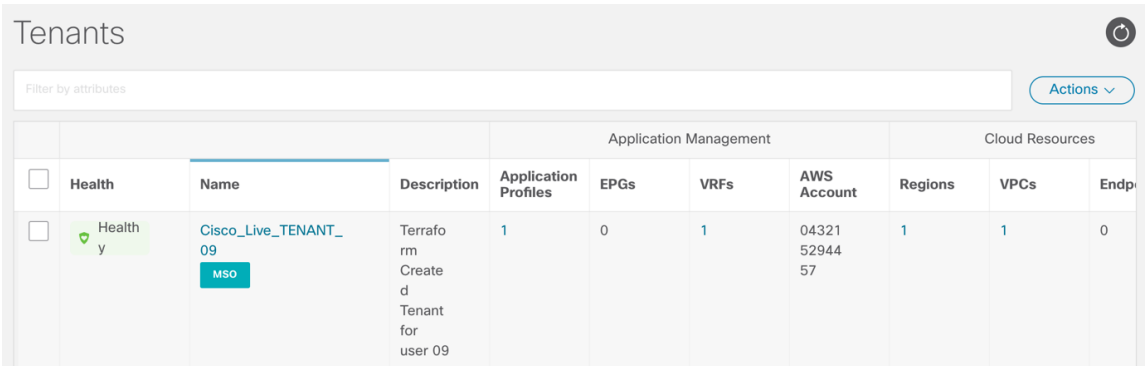
4. The apply stage is waiting for a manual input before execution. Now that we have verified that the plan was successful and we know what changes are going to be made, let's proceed with running the apply.

   It should take a few minutes and once it has executed successfully should look like this:

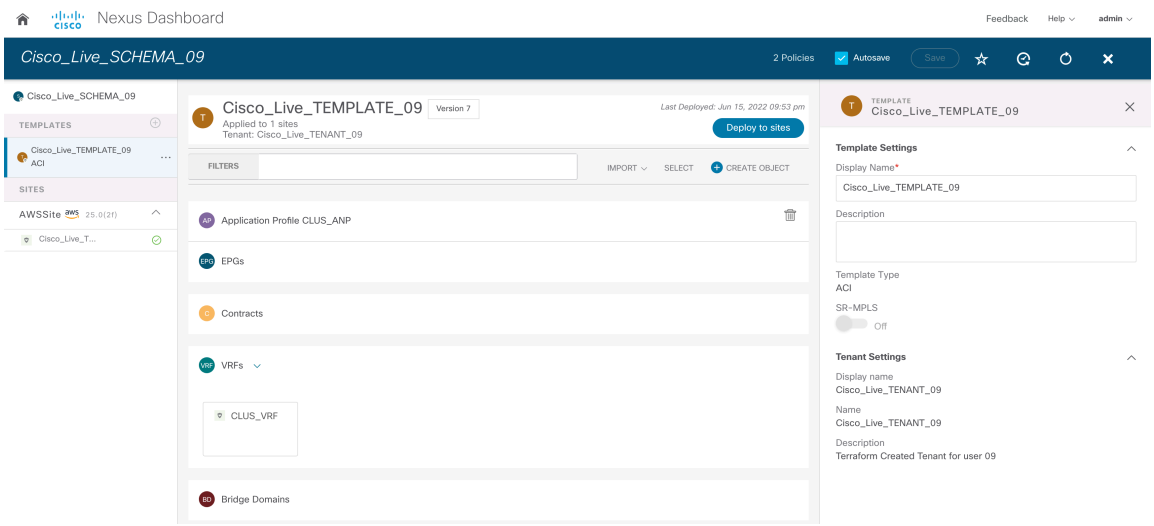


5. Head over to the Cloud APIC and the NDO to ensure that all the components we are interested in have been created successfully. Study the Schema carefully on the NDO to ensure all components we want are successfully created.

   cAPIC:



   NDO:



We have now successfully executed our first Terraform pipeline, congratulations!

We are ready to proceed with using this VRF and ANP by stretching it to the premises and using it to deploy our application. We will use Ansible once again to deploy the application within some new EPGs that will be created in the cloud-stretched tenant.

Optional Task:

You can go back to the jobs and see the console output for the apply. Study the outputs and see if there are any issues with the same and see which resources were created in which order. Check if this makes logical sense (and would be the same order you would approach a deployment from the GUI).

- **What did we learn in this Task?**

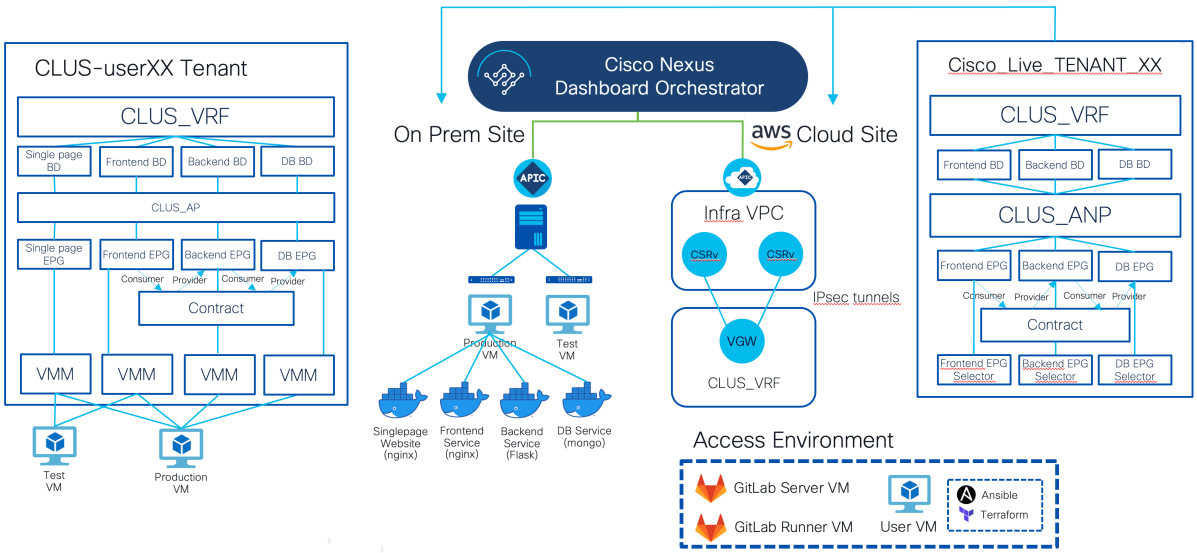
- We've learned the basic principles of Terraform and used it to create cloud based ACI objects.
- We understand how the cloud APIC is set up in parallel to the on-prem components.
- We now also have a comparison between Ansible and Terraform pipelines, as well as the pros and cons of both languages. As you can see Terraform being declarative has a simpler pipeline unlike Ansible which is procedural and needs to be told exactly what to do.

Task 4: Stretching the cloud tenant towards on-prem with the use of Ansible and CI/CD

[estimated time 40-50min]

As we have created the cloud tenant on the cAPIC, we want to stretch it now towards our on-prem site. We'll do that with the use of Ansible, and – you guessed it – CI/CD pipelines. These will be modified this time – we want to use some of the benefits built into ACI to create an even more fool-proof environment.

In this task, you will not be copying scripts that we've prepared for you. Instead, you should use your Ansible and CI/CD knowledge to code everything yourself. Of course, in case you're stuck and you're running out of estimated time for this Task, refer to our Github repository and follow along.



Step 1. Prepare the Ansible code necessary to take a tenant snapshot

1. Upon completion of previous Task, navigate back to your Ansible directory:
   cd ~/Desktop/Infrastructure/Ansible

2. We will follow the same structure as you've left upon completion of Task 2, with modification throughout the following Task.

3. In *plays*, create a new playbook where you'll be creating an ACI snapshot. For simplicity, you don't need to create a separate role, instead you can define the tasks within the playbook itself.

4. To configure above playbook, refer to an ACI collection Ansible module for taking the snapshot. Its documentation is defined in the following resource:
   - cisco.aci.aci_config_snapshot module – Documentation

Step 2. Modify your pipeline to use the playbook which you've created in the previous step

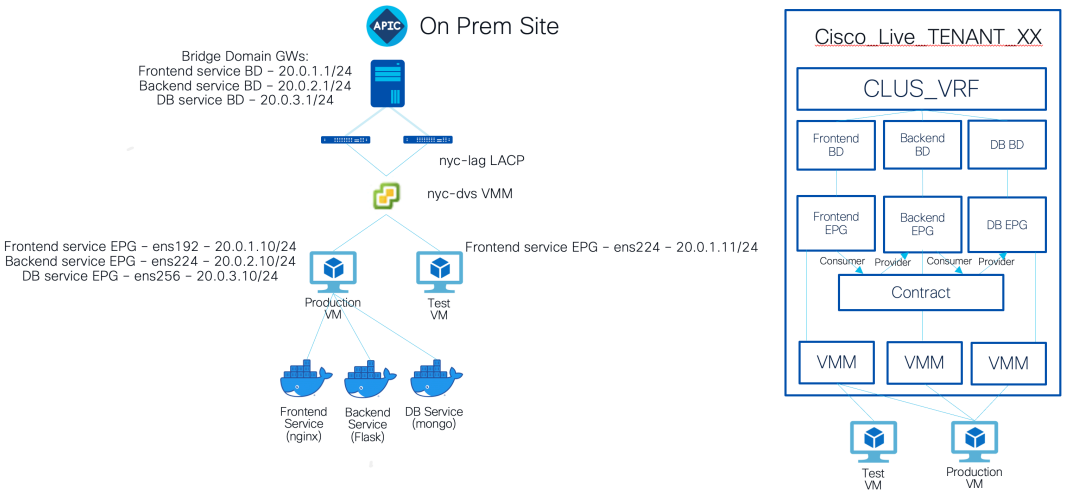
1. Edit your *.gitlab-ci.yml* file to add the snapshot creation playbook into the pre-deploy stage of your pipeline.

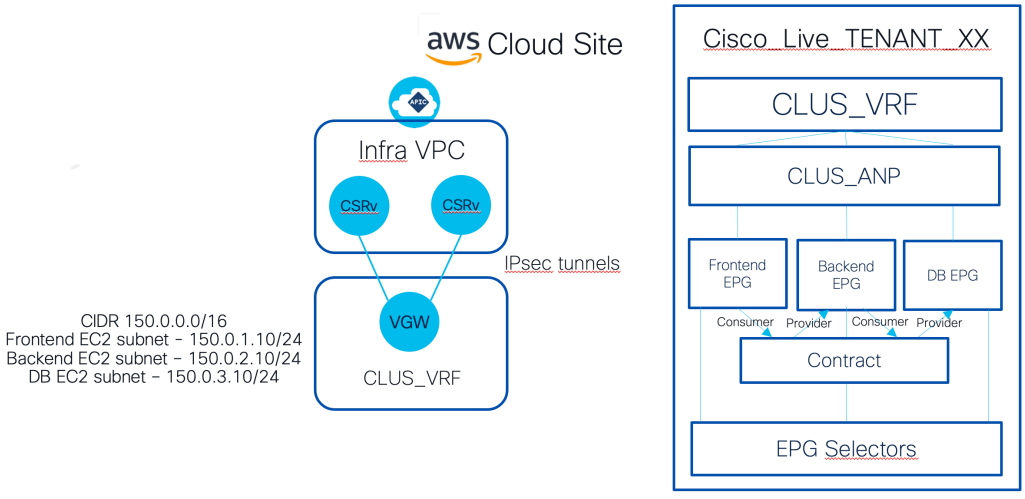
2. Edit your *.gitlab-ci.yml* file to change the name of the playbook that you'll run in the deploy stage of your pipeline.

Step 3. Prepare the Ansible code needed to stretch the cloud tenant

For this step you'll need to ensure the following workflow is defined in your Ansible code:

- Attaching the cloud tenant to the on-prem site
- Adding the site to the schema created in Task 3
- Inside of the template, you'll need to provision the following:
    - Three Bridge Domains, three EPGs with the naming convention of your choosing, a contract, a filter
- Under the on-prem site template configuration:
    - Add the VMM domain – ***nyc-dvs*** – as the attachment for that EPG
    - Specify the VMM Enhanced LAG policy – ***nyc-lag***, as we're using LACP on the vDS uplinks
- IP addressing from the desired state topology below:





The following is the Ansible directory structure which you can reference when working on your own code:

```
group_vars/
|     apic/
|     |     apic.yml
|     |     apic-password.yml
|     |_    apic-password-plain.yml
|     ndo/
|     |     ndo.yml
|     |     ndo-password.yml
|     |_    ndo-password-plain.yml
|     vcenter/
|     |     vcenter.yml
|     |     vcenter-password.yml
|_    |_    vcenter-password-plain.yml
plays/
|     apic-object-creation-playbook.yml
|     apic-snapshot-playbook.yml
|     cloud-tenant-stretch-playbook.yml
|     service-deployment-playbook.yml
|     roles/
|     |     apic-object-creation/
|     |     |     tasks/
|     |     |_    |_    main.yml
|     |     contract-creation/
|     |     |     tasks/
|     |     |_    |_    main.yml
|     |     cloud-tenant-stretch/
|     |     |     tasks/
|     |     |_    |_    main.yml
|     |     service-deployment/
|     |     |     tasks/
|     |     |_    |_    main.yml
|     |     vm-pg-attachment/
|     |     |     tasks/
|_    |_    |_    |_    main.yml
vault-pass
inventory.ini
```

The first change you'd make would be to define the *ndo* directory under *group_vars,* following the similar structure as for *apic* and *vcenter* directories within *group_vars*. Within the *group_vars/ndo/ndo.yml* file you'd have to define the NDO credentials and IP, the name of the tenant you're stretching, the name of schema, template, VRF, and ANP that were created in Task 3, then the site names, service-specific variables (i.e. BD, EPG, subnets for frontend), VMM and LAG details, and finally contract-specific variables.

When creating the new *ndo*specific *group_vars* directory, don't forget to encrypt the passwords for it with Vault:

```
dcloud@dcloud-virtual-machine:~/Desktop/Infrastructure/Ansible$ ansible-vault encrypt group_vars/ndo/ndo-password-plain.yml --output
 group_vars/ndo/ndo-password.yml --vault-password-file vault-pass
Encryption successful
```

Repeat for APIC and vCenter group_vars plain passwords.

Then, you'll need to define new role for stretching the cloud tenant. For said role, within *cloud-tenant-stretch/tasks/main.yml*, you will need to use the following Ansible collection:

- Cisco.Mso Ansible Collection Documentation

The workflow for the *cloud-tenant-stretch* role is as follows:

- you'll start by attaching the cloud tenant to the on-prem site
- then you associate the on-prem site with the template
- add BDs, subnets, EPGs, VMMs, cloud-site selectors, contracts, filters, filter entries to the template
- associate the contracts in accordance with our topologies
- finally, deploy the schema templates to sites

One thing to note here is that you do not stretch a cloud subnet to the premises inside an EPG, as you would traditionally do when stretching an EPG between 2 on prem sites. Since the cloud has no concept of broadcast, you can't extend a traditional layer 2 boundary all the way into the cloud. You need to declare different subnets for the premises. What an EPG signifies is that you can have complete communication within the different subnets (irrespective of whether they are on the premises or the cloud) without any contracts.

What you'll also need is to make sure that the subnets are set to be advertised externally. This is configured with the following bolded property of the *cisco.mso.mso_schema_template_bd_subnet* module:

```
cisco.mso.mso_schema_template_bd_subnet:
  host: '{{ ndo_host }}'
  username: '{{ ndo_creds.user }}'
  password: '{{ ndo_creds.pass }}'
  schema: '{{ ndo_schema }}'
  template: '{{ ndo_template }}'
  bd: '{{ ndo_bd_backend }}'
  subnet: '{{ndo_bd_backend_gw}}/{{ndo_bd_backend_gw_mask}}'
  state: present
  scope: public
  validate_certs: no
delegate_to: localhost
```

You will also be using the concept of EPG Selectors for the cloud site itself. These define the way in which we group endpoints together in the same EPG (so, the Security Group) in the cloud, since we cannot create static path attachments for example, as we would do in the on prem site. The selector is defined with a following module:

```
cisco.mso.mso_schema_site_anp_epg_selector:
  host: '{{ ndo_host }}'
  username: '{{ ndo_creds.user }}'
  password: '{{ ndo_creds.pass }}'
  schema: '{{ ndo_schema }}'
  site: '{{ ndo_site_cloud }}'
      template: '{{ ndo_template }}'
      anp: '{{ ndo_anp }}'
```

```
    epg: '{{ ndo_epg_frontend }}'
    selector: selector_frontend
    expressions:
      - type: ip_address
        operator: equals
        value: '{{ ndo_bd_frontend_subnet_cloud }}'
    state: present
    validate_certs: no
  delegate_to: localhost
```

Finally, create the *cloud-tenant-stretch-playbook.yml* and add the abovementioned *cloud-tenant-stretch* role and the *vm-pg-attachment* role to assign the port groups of your Prod and Test VMs to the EPGs created by the new tenant. Again, don't run the playbook locally and instead move to the next step once you're done.

Ensure that the .gitlab-ci.yml is moved to the root directory before you move to the next step to commit.

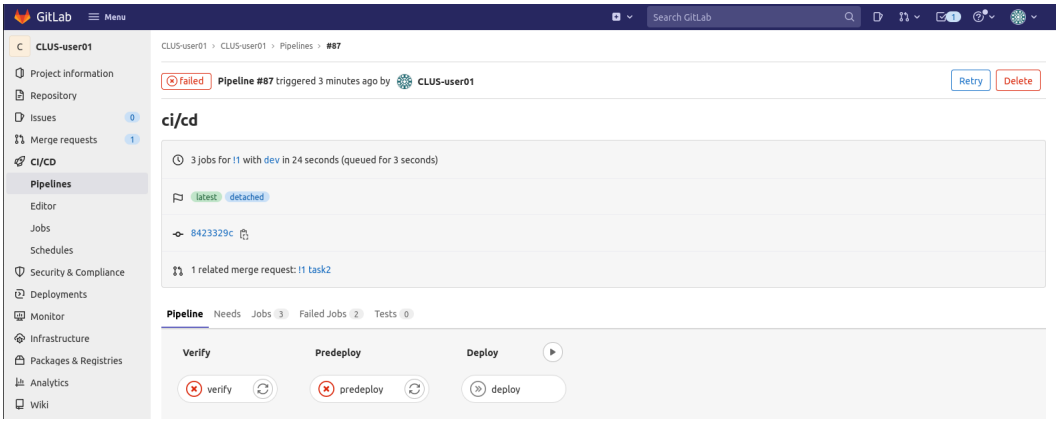
Step 4. Push your code to the remote repository and merge the branches

1. Push your code to the remote repository – again, making sure you're on the *dev* branch.
   git branch -a
   git status
   git add .
   git commit -m '<your message>'
   git push

2. Go to your Gitlab project and check the state of the dev branch. Create a new merge request. Again, when specifying the parameters for the merge request, DO NOT select the option to remove the branch upon a successful merge, that setting must be unchecked. The pipeline will commence shortly.

Step 5: Deploy the code to the NDO

1. In case you run into any issues with the pipeline, work on fixing it. To check the state of the jobs taken by the pipeline, go to CI/CD -> Pipelines and select one of the jobs:

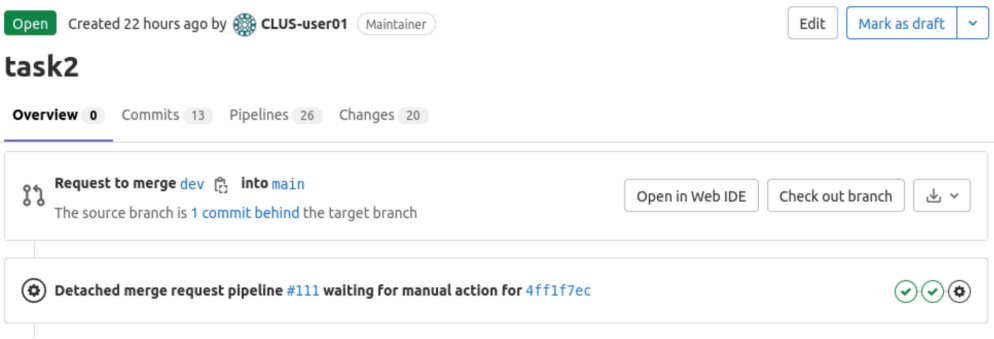


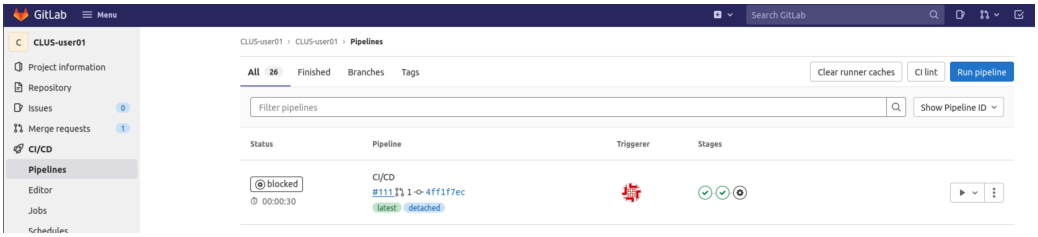
You can also verify them from the Merge requests tab:

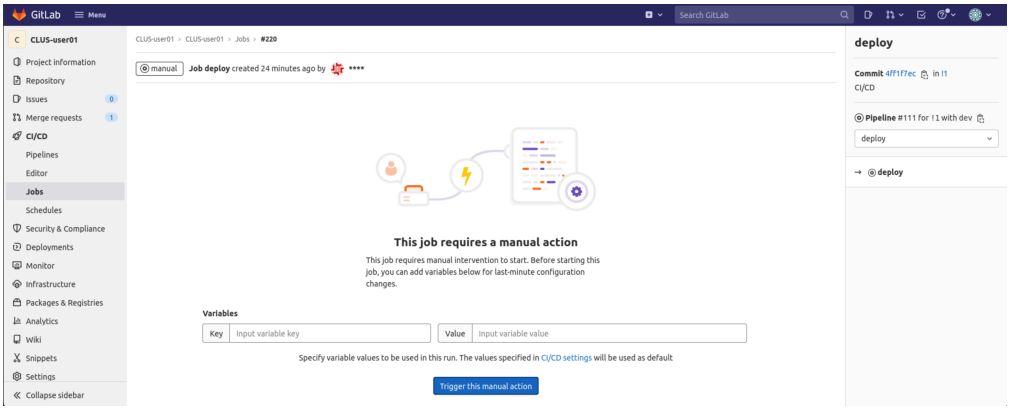
2. If the jobs fail, select one of them and check the reason for the pipeline failing. Work on fixing the issues. Iterate over this process until you pass the job.

   If you're hitting many Lint errors and you're approaching the proposed time limit for this task, you can modify your *.gitlab-ci.yml* file to skip Lint in the *verify* stage of the pipeline by commenting out the respective line.

3. Once your pipeline completes the *verify* and *pre-deploy* stages, review the results and once you're done, approve the deploy stage by hitting the "Trigger this manual action" button.

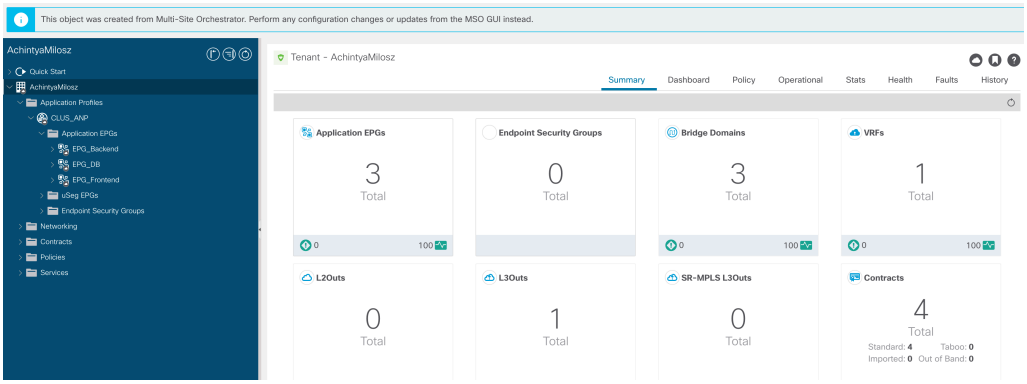




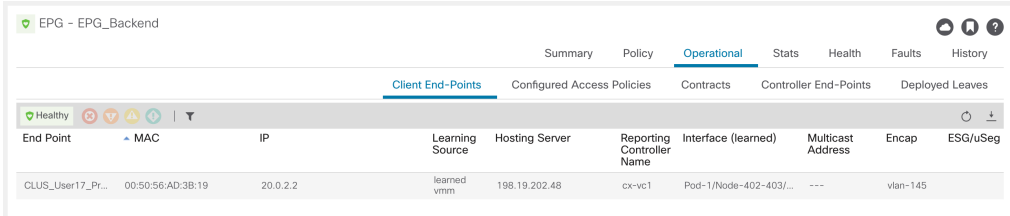


4. Upon merging branches and deploying the code, verify the following:

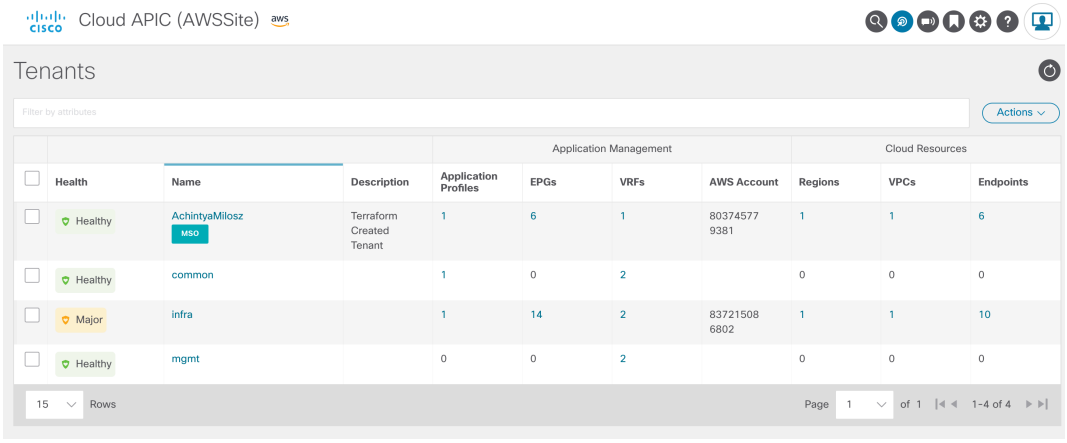
- ACI objects pushed to your on-prem site by the NDO



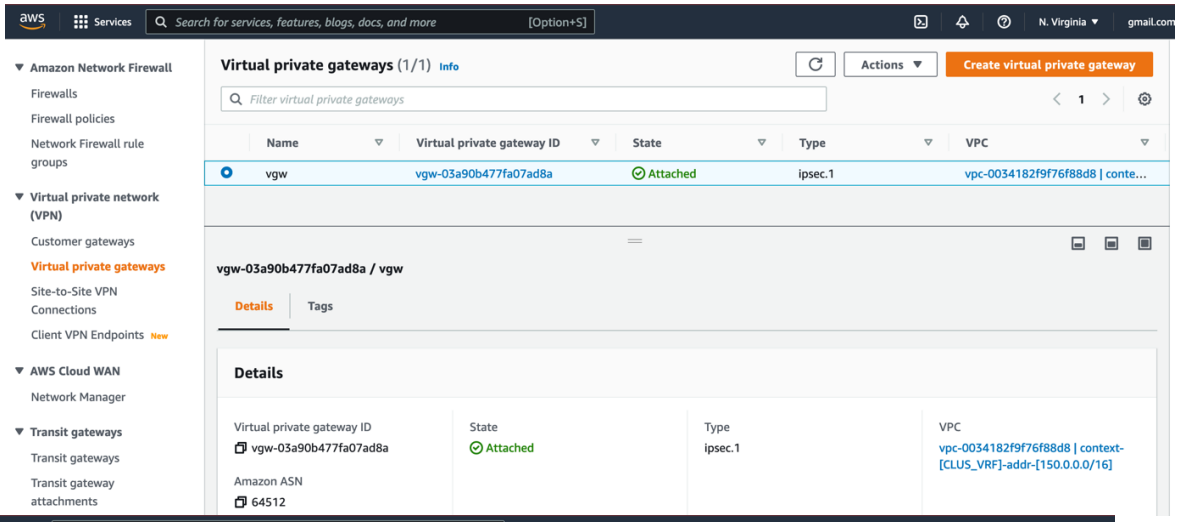
- Endpoints being learned under the EPGs

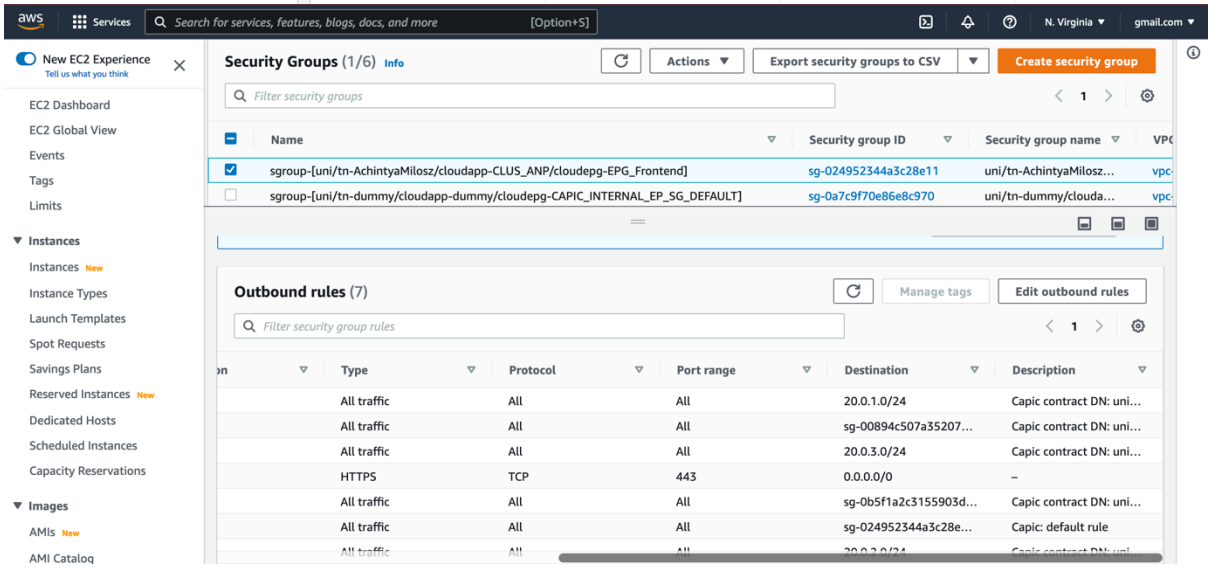


- ACI objects pushed to the Cloud APIC by the NDO



- Cloud resources created in your AWS account (verify security groups, VGW)





- **What did we learn in this Task?**

- We've gotten an insight into the objects that NDO pushes to the sites when stretching objects across hybrid sites

- We've used the skills we've learned from previous tasks to prepare the Ansible code necessary to create ACI objects across hybrid clouds.
- We've used the CI/CD approach to iteratively work with our code.

Task 5: Integrate with the cloud and copy front end to the cloud for redundancy

[Estimated time: 20-30mins]

We're now ready for the migration to the stretched cloud tenant. Our ACI on-premises site and our ACI cloud site has been already added to the Nexus Dashboard Orchestrator in the previous tasks and we have a single stretched tenant.

To help with our traffic burst issue, we will deploy the currently used single page website to the cloud, so whenever there is a traffic spike, we will be able to load-balance the traffic between the on-premises site and the cloud site. You will also create additional components in case we would like to migrate the full application to the cloud in the future.

Thus, you will need to create the AWS components that will be required for the integration. For this, you'll use Terraform again.

Step 1: Prepare the CI/CD pipeline that will verify the NDO objects responsible for stretching to the Cloud ACI site

As you already know with Terraform, we have 3 stages in the pipeline - Validate, Plan and Apply. We'll start by creating our gitlab-ci.yml file that we already know how to work with thanks to Task 3 (we could very well reuse the same file as there are no specific values that we have hardcoded).

1. Empty your ~/Desktop/Terraform directory.

2. Create the *.gitlab-ci.yml* file.

   Just remember that we no longer need -parallelism=1 in the apply as the AWS provider can understand the dependencies well and work in parallel (no harm done if you leave it, just that the script will take longer to execute).

Step 2: Prepare the Terraform script that will create the AWS objects for the load balancing

Our objective now is to create the Terraform script that will create 3 EC2s on our stretched Tenant. We will be eventually using one EC2 for each of the containers we saw on the on-prem and therefore each of them will be in a different EPG. For the moment, we will only deploy the front-end container on a cloud EC2 and leave the other 2 as they are.

We want to create an EC2 first and then inside it docker-compose our single page website. Docker Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Then we will verify if we are able to reach it from the on-prem and the other EC2 instances on the cloud.

1. Create ssh certificate on User VM using ***ssh-keygen -t rsa***. Then copy the keys to another folder in our directory using the *cp* command in the terminal. We will use this to upload public key to AWS and you can login with private key, without any additional complexity of authentication.

2. Now start with the *vars.tf* file (short for variables). Here we want to once again define variables that will be useful for us in this task. We want the Terraform "instance_type" (which should hold the value "t2.micro"- this size of an EC2 will suffice for our containers), the AWS default credentials (exactly like

the “awsstuff” variables we created in the last Terraform task) and region (“us-east-1”).

3. In our *override.tf* let’s provide the actual AWS credentials that will be used to reach and configure our tenant (similar to our awsstuff variable object in Task 3).

4. Let’s create the *terraform.tfvars* file with the values of all the variables in *vars.tf.*

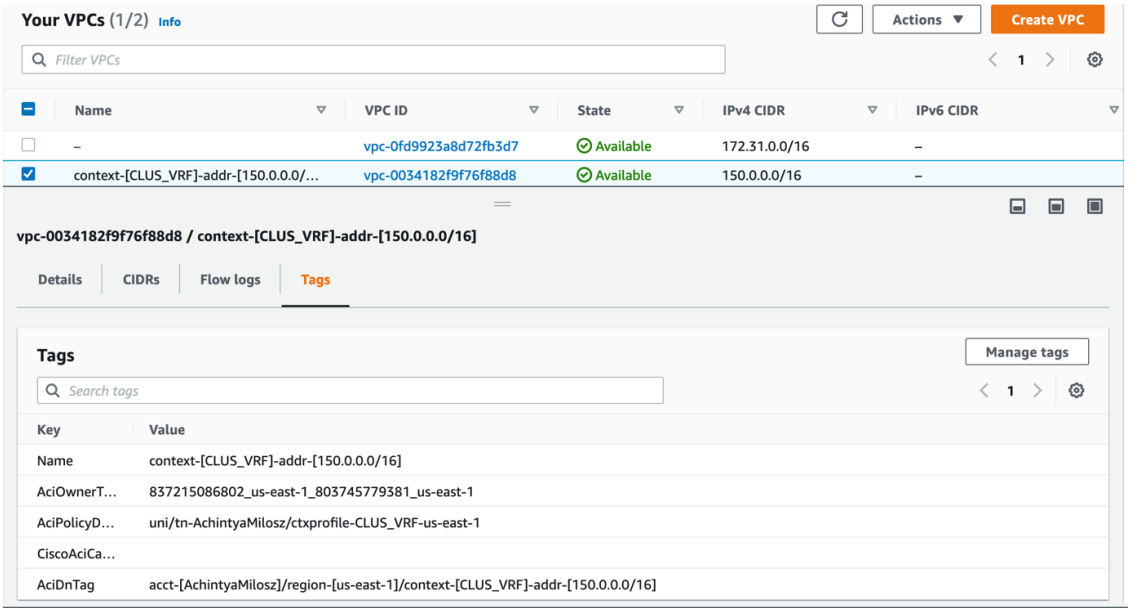
5. Now let’s move to our *main.tf* script.

   For this task we will mainly be leveraging the AWS provider. Have a look at the terraform registry below:

   - Terraform Registry - AWS Provider Documentation

   Start by declaring the required provider with a minimum version of 3.6. Then pass whatever parameters the provider is looking for to initialise.

6. Next create a data source “aws_vpcs” to get the VPC ID of ACI-built VPC on AWS. Use the AciPolicyDnTag as a tag and pass the Tenant’s DN. You can track DN tags by navigating to VPCs on your AWS account and checking the tags section.



This is what the data source would look like (* acts like a filter and can match any value, like you might in a regex):

```
data "aws_vpcs" "vpc_id" {
  tags = {
    AciPolicyDnTag = "*Cisco_Live_TENANT_XX*" # adding filter for our
VPC - where XX is the id assigned to you
  }
}
```

Set a local variable with the value of the vpc id. Do note that this may be in a list form and you need to use type conversion “tolist” to extract elements from the list- it would look something like this.

```
locals {
vpcid=element(tolist(data.aws_vpcs.vpc_id.ids),0)
}
```

Afterwards, you can refer to this saved vpcid using 'local.vpcid'

7. Now we want to create 3 data sources of the type "aws_subnet_ids" and use the filters that it accepts to identify your 3 subnets (for frontend, backend and DB- 150.0.1.0/24, 150.0.2.0/24 and 150.0.3.0/24 respectively).

   Next populate the subnet ids in the variables (again using the locals block) the same way you did above.

   A good way to debug and confirm if the right subnets were assigned or to get any value assigned is by using the 'output' function in Terraform.

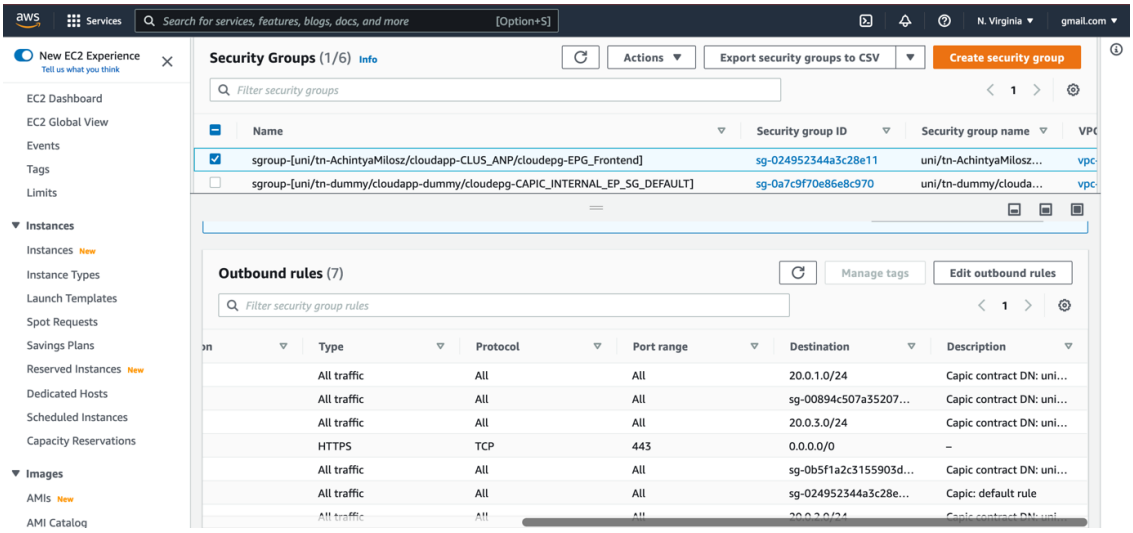
   You can make Terraform print out the value in the following format:

```
output "SomeRandomValue" {
  value = <element id>
}
```

8. Next, we create 2 interfaces for each EC2. Find a relevant AWS resource for this and create 6 interfaces. The private IP of the first interface in each EC2 will be 150.0.1.5, 150.0.2.5, 150.0.3.5 respectively and the second will be .6 in each.

9. We also need to associate an Elastic IP (EIP) to the first interface of each EC2, which would create a public IP address for it. Look for a relevant resource for this too and create an object.

   How cAPIC works is by using EPG selectors to assign the interfaces to the right EPGs (known as security groups on the AWS side). Therefore, if we assign the IPs in relevant subnets, they will automatically be moved to the relevant security groups that have the contracts you defined on prem already. You can test this by going to security groups and checking the inbound and outbound rules.



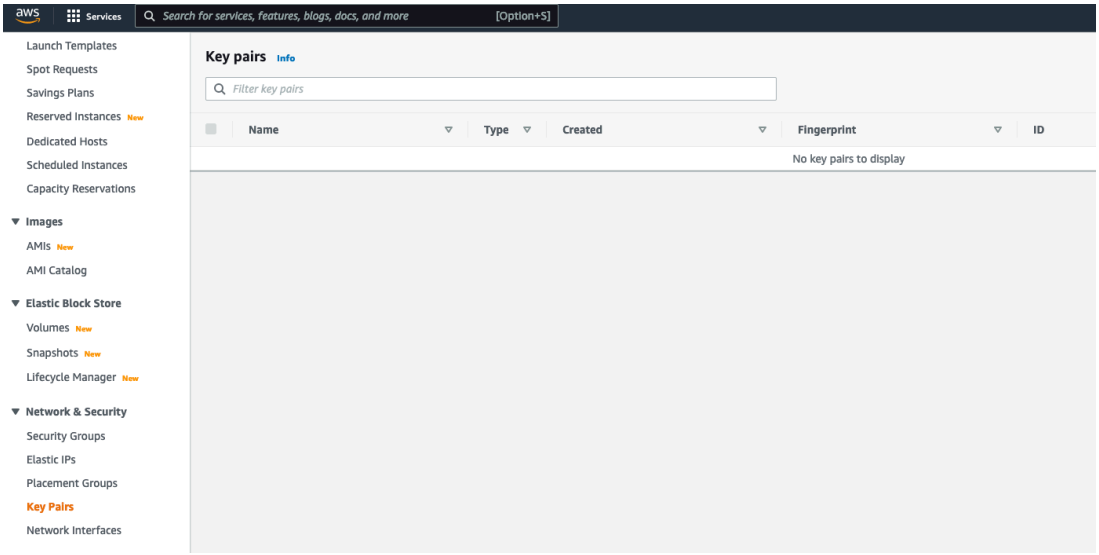
10. Next, create an aws_key_pair resource to create a key name of your choice. Pass the public_key attribute that you created earlier using

```
public_key = file("${path.module}/keypair/terraformkey.pub")
```

where terraformkey.pub is the public key name.

`${path.module}` is in relation to the current directory and hence this assumes you have placed the keys in a folder called keypair in the same folder as the rest of your Terraform code.

Do remember to verify that a key with the same name that you are choosing doesn't exist in AWS (you can check in AWS console under the 'Key Pairs' section in 'Network & Security' under EC2.



11. Now let's get ready to spin up the AWS instance for the single page website. Start by using data type "aws_ami" to create an Amazon Machine Image (AMI). Verify on the Terraform Registry what it needs. We will be using the amz2-ami-hvm image so you can declare it in the following fashion:

```
data "aws_ami" "std_ami" {
```

```
  most_recent = true
  owners      = ["amazon"]

  filter {
    name   = "name"
    values = ["amzn2-ami-hvm*"]
  }
}
```

We will reuse this when we create the instances for backend and DB in the next step.

12. Now we can start creating our single page website EPG instance. Use the aws_instance resource and pass our key and any relevant values that the resource needs. Also create 2 network interfaces with device indexes 0 and 1 which we already created earlier inside the aws_instance resource. You can find documentation on this resource here:
    - Terraform Registry – AWS Provider – aws_instance resource.

13. Now let's repeat this exact procedure for our other 2 EC2s as well.

14. Before we proceed to install our container on the single page website EC2, there are a few more things to do:
    a. We need to add an inbound security group rule for SSH to each of the 3 security groups.

       This would be done in the following fashion:

       Start by identifying the security group in a data source 'aws_security_groups':

```
data "aws_security_groups" "sg1" {

tags= {
 AciDnTag= "*EPG_Frontend*" # this a filter is to get front-end SG
}
}
```

       Assign the id to a local variable using element(tolist),0 the way we did above for the subnets.

       Then create a new security group rule using the 'aws_security_group_rule' resource:

```
resource "aws_security_group_rule" "secgroup1" {

 type = "ingress"
 from_port = 22
 to_port = 22
 protocol = "tcp"
 security_group_id = local.securitygroup1
 cidr_blocks=["0.0.0.0/0"]
depends_on=[
aws_instance.ec2-1
]
}
```

       Now repeat the above procedure for the other 2 EC2s as well by collecting their SG id and adding a security group rule for them.

    b. We need to expose port 80 for the single page website EPG security group in the ingress direction and 443 in the egress direction. Follow the exact procedure above to add a security group rule for this.

       This is because our application runs on port 80, as you already saw when you deployed this on the premises. Additionally, 443 is what will be used to clone from the Github repo with our docker containers.

    c. We need to add a default route towards the Internet Gateway (IGW) for our VPC's route table.

       Start by collecting our route table and Internet Gateway details like this:

```
data "aws_route_table" "table" {
 subnet_id= local.subnet1
}

data "aws_internet_gateway" "igw" {
 filter{
 name = "attachment.vpc-id"
 values = [local.vpcid]
 }
}
```

Then proceed to create an aws_route resource that can assign a default route

```
resource "aws_route" "r" {
 route_table_id = data.aws_route_table.table.id
  destination_cidr_block = "0.0.0.0/0"
  gateway_id=data.aws_internet_gateway.igw.internet_gateway_id

}
```

Once you have done this, perform a terraform init, terraform validate and terraform plan to check if the config and dependencies are all in place.

15. Now we can install all the necessary components inside our EC2 for it to host our single page web container.

    We shall be leveraging “null_resource” which is a resource in Terraform that just performs necessary tasks within it but does no further changes. The triggers argument can be used to mention what values that when changed will cause the resource to be replaced. More on this here:

    - Terraform Registry - null_resource

    We start by creating a null resource that can help Terraform buy some time to make sure the EC2 is reachable. Use the provisioner “local-exec” within this resource and run the command “sleep 30”. This would look like the following:

```
resource "null_resource" "update" {
  depends_on = [aws_instance.terraform1-ec2]
  triggers = {
    build_number = timestamp()
  }

  provisioner "local-exec" {
    command = "sleep 30" # buy a little time to make sure ec2 is reachable
  }
}
```

    Now that we have given enough time for the EC2 to spawn and be reachable via SSH, we can start installing our dependencies inside the EC2. Once again, a null resource will be used inside which a provisioner “remote-exec” will execute our desired commands.

Now the desired commands we want to run are:

a. Install yum and update it
b. Install docker
c. Start docker
d. Provide privileges to /var/run/docker.sock
e. Install docker compose
   Instructions for this can be found here:
   - DigitalOcean - How to install and use Docker Compose on Ubuntu 20.04
f. Git clone the repository https://github.com/achintya96/cisco-live-2022-aws-containers where all the composes and code is stored.
g. Move inside directory 'singlepage' inside the main repo
h. 'Docker-compose up -d' to run the actual docker compose.

We also need to provide the connection type to the provisioner (SSH with the user details, host name and key)

This is what the null resource block would look like when it all comes together:

```
# Launching our singlepage container inside the front-end EC2

resource "null_resource" "EC2Commands" {
  depends_on = [null_resource.update]

  triggers = {
    build_number = timestamp()
  }

  provisioner "remote-exec" {
    inline = [
      "sudo yum update -y",
      "sudo amazon-linux-extras install docker -y",
      "sudo service docker start",
      "sudo chmod 666 /var/run/docker.sock",

      # Code for Compose- SinglePage
      "sudo curl -L
\"https://github.com/docker/compose/releases/download/1.29.0/docker-
compose-$(uname -s)-$(uname -m)\" -o /usr/local/bin/docker-compose",
      "sudo chmod +x /usr/local/bin/docker-compose",
      "docker-compose --version",
      "sudo yum install git -y",
      "git clone https://github.com/achintya96/cisco-live-2022-aws-
containers",
      "cd cisco-live-2022-aws-containers/singlepage",
      "docker-compose up -d"

    ]
    connection {
      type        = "ssh"
      user        = "ec2-user" # this is the inbuilt ec2 user name for the
used ami
      private_key = file("${path.module}/keypair/privatekeyterraform")  #
path to private key that you made
      host        = aws_instance.ec2-1.public_ip

    }
```

```
  }
}
```

16. Lastly output the public IPs of the EC2s using the public_ip attribute of the EC2 resources you created – this will make it simple for us to find the details of the IPs and start SSHing into them for testing.

17. Perform a *terraform validate* and *terraform plan* to see if there are any issues with the script before you proceed to commit this to git and push it to our repository (again, it's optional to *terraform apply* this from the User VM to test, just make sure you destroy it right after)

18. Finally push this code to a new project.

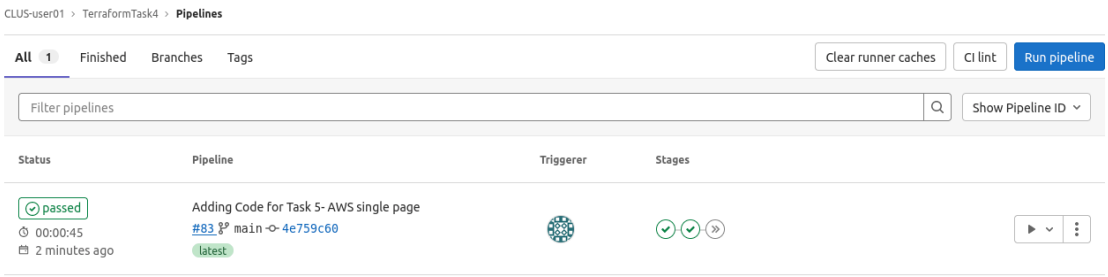
Step 3: Verify the Terraform script against your pipeline and execute it

Your final repository should look like this:

Name	Last commit	Last update
keypair	Adding Code for Task 5- AWS single page	2 minutes ago
.gitlab-ci.yml	Adding Code for Task 5- AWS single page	2 minutes ago
main.tf	Adding Code for Task 5- AWS single page	2 minutes ago
override.tf	Adding Code for Task 5- AWS single page	2 minutes ago
terraform.tfvars	Adding Code for Task 5- AWS single page	2 minutes ago
vars.tf	Adding Code for Task 5- AWS single page	2 minutes ago

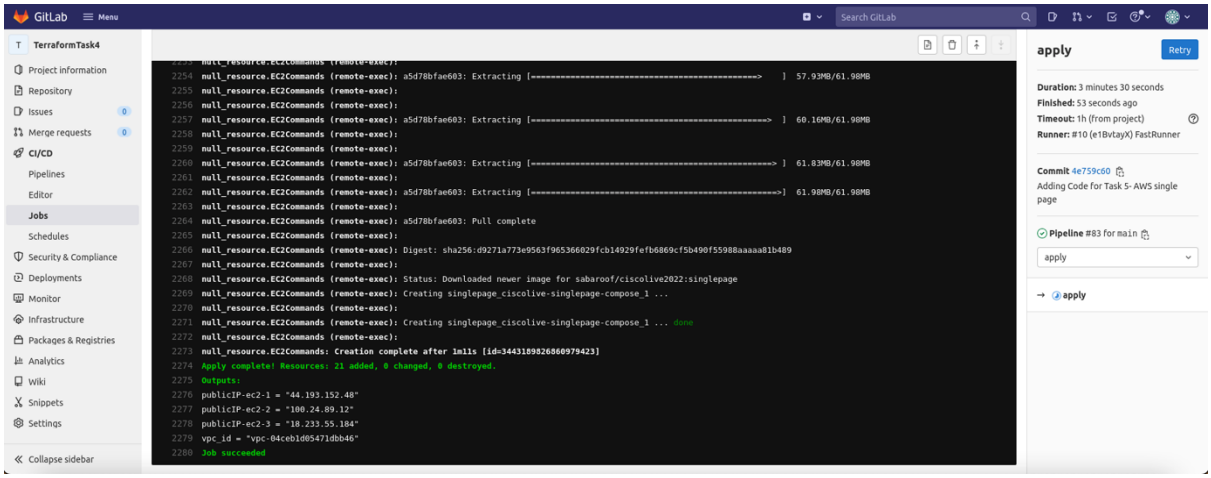
1. Since you created a new project, you will need to enable your previously shared runner.

2. Refresh the validate stage and you will see the execution begin:

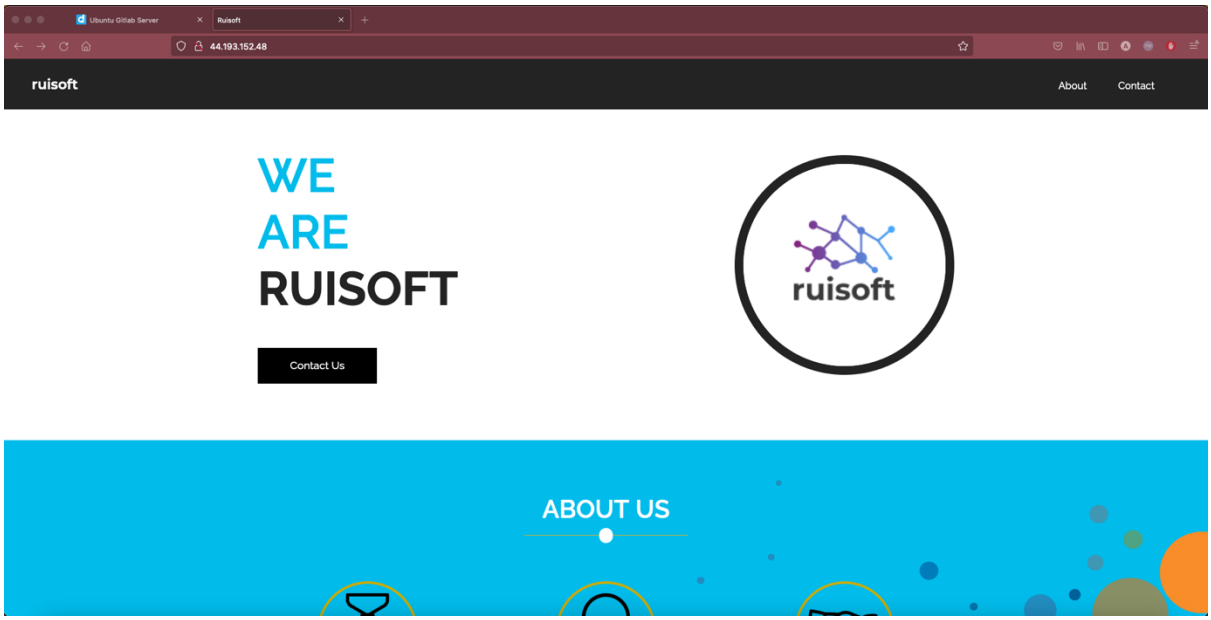


3. Click on the apply stage in the pipeline and see if it is successful.

4. Once the apply is successful, head over to the job and see the outputs of the apply, you should see the public IPs that you passed as an output.

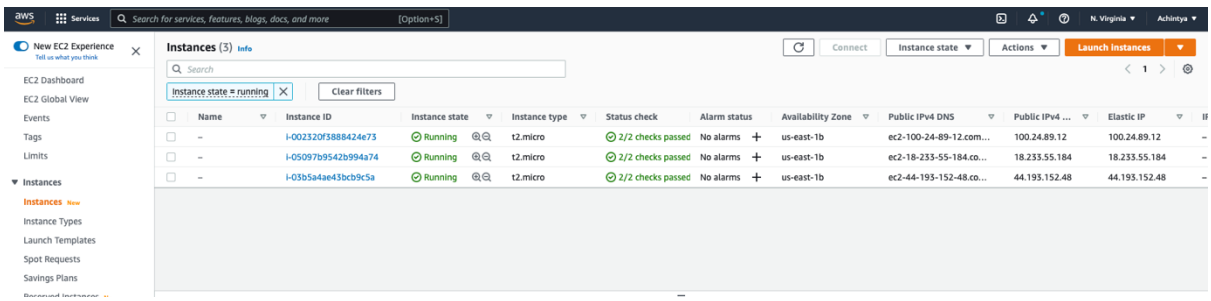


5. Go to the browser and see if we can reach the single page website public IP and visit our front end successfully.

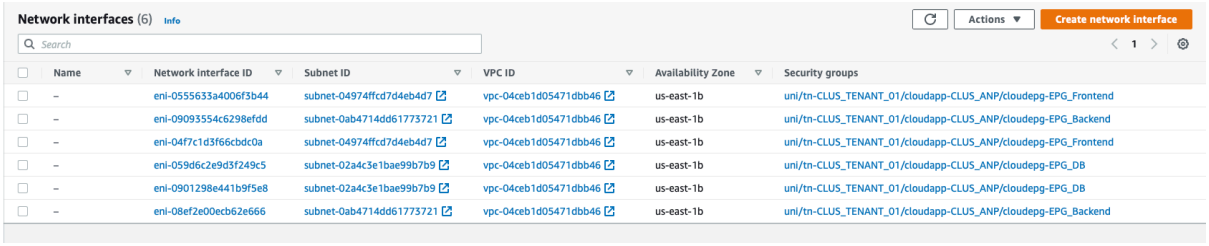


6. Next head over to the AWS tenant you created and check the policies created:

   a. EC2 instances:

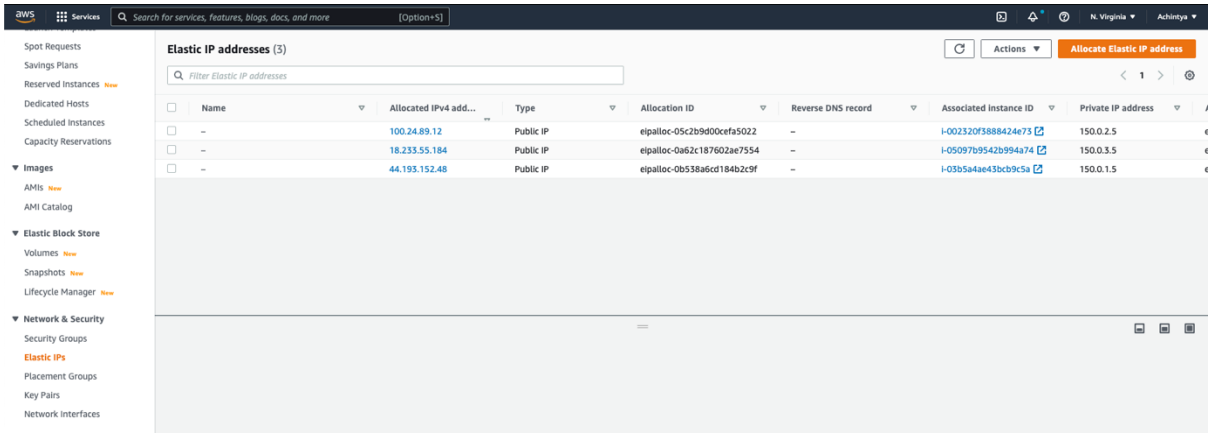


b. Interfaces (verify assignment into the correct Security Group for the right EPG):



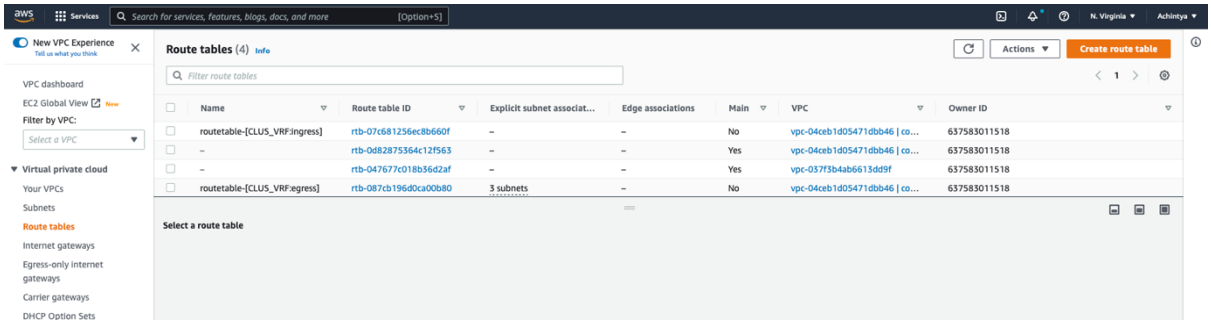
Scroll right for the IPs and details

c. Elastic IPs:

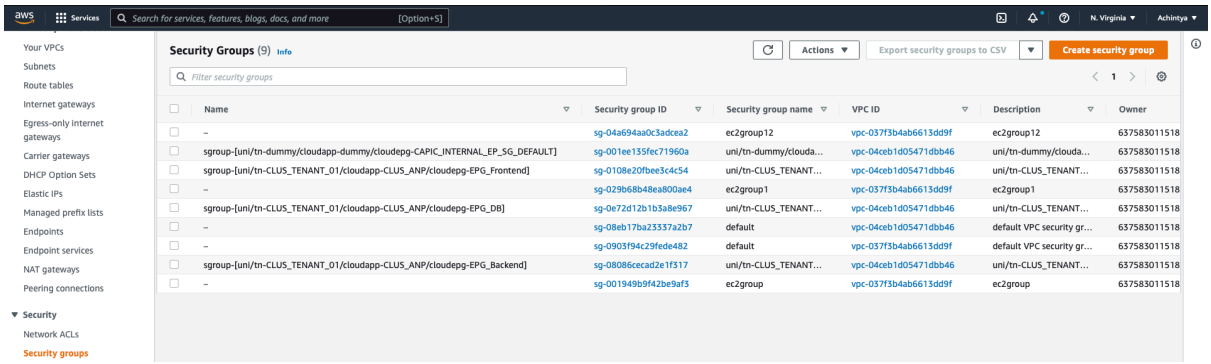


7. Then let's check some VPC details on the AWS console. Switch to the VPC view and look at:

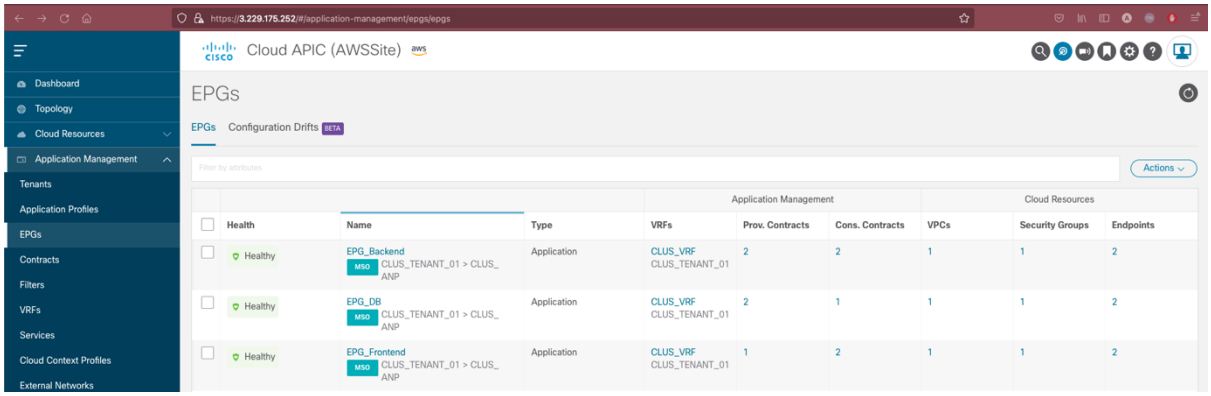
a. Route table used by our subnets:



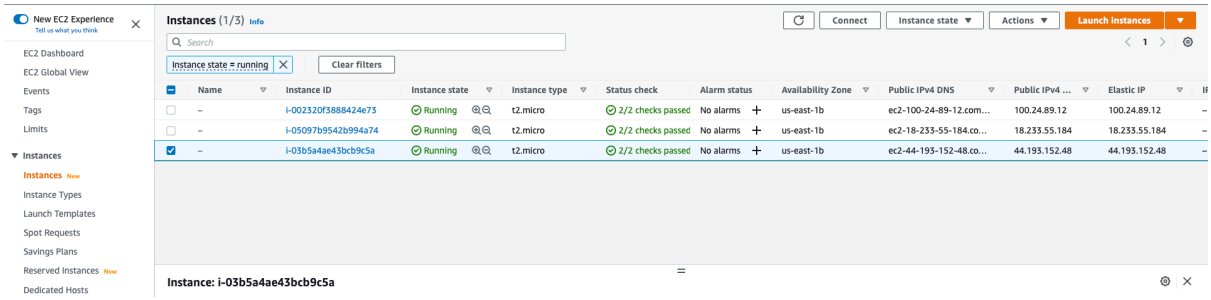
b. Security groups:



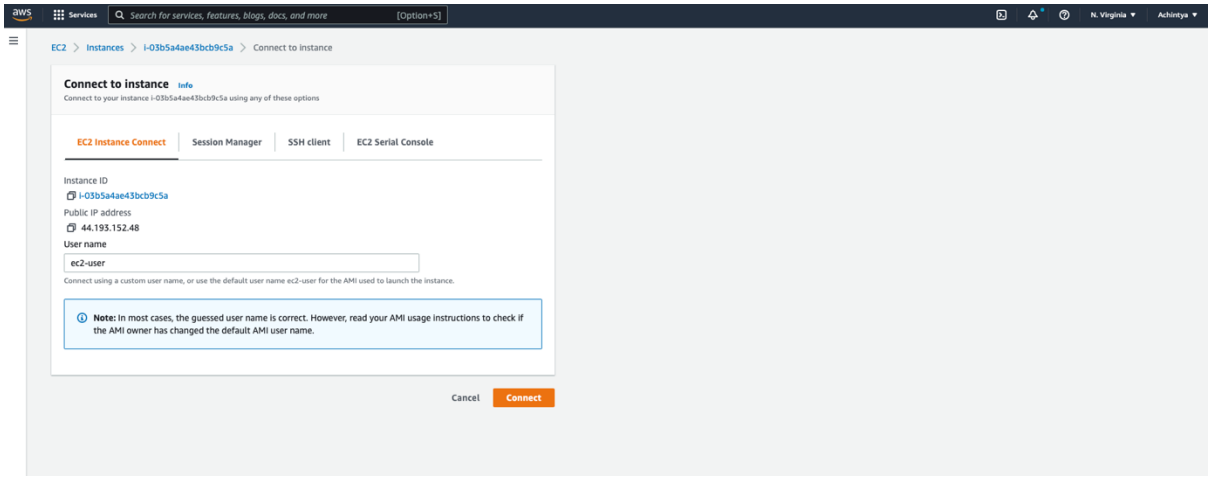
8. Now head over to the Cloud APIC and verify if you can see 2 endpoints in each EPG (one for each interface of the EC2):



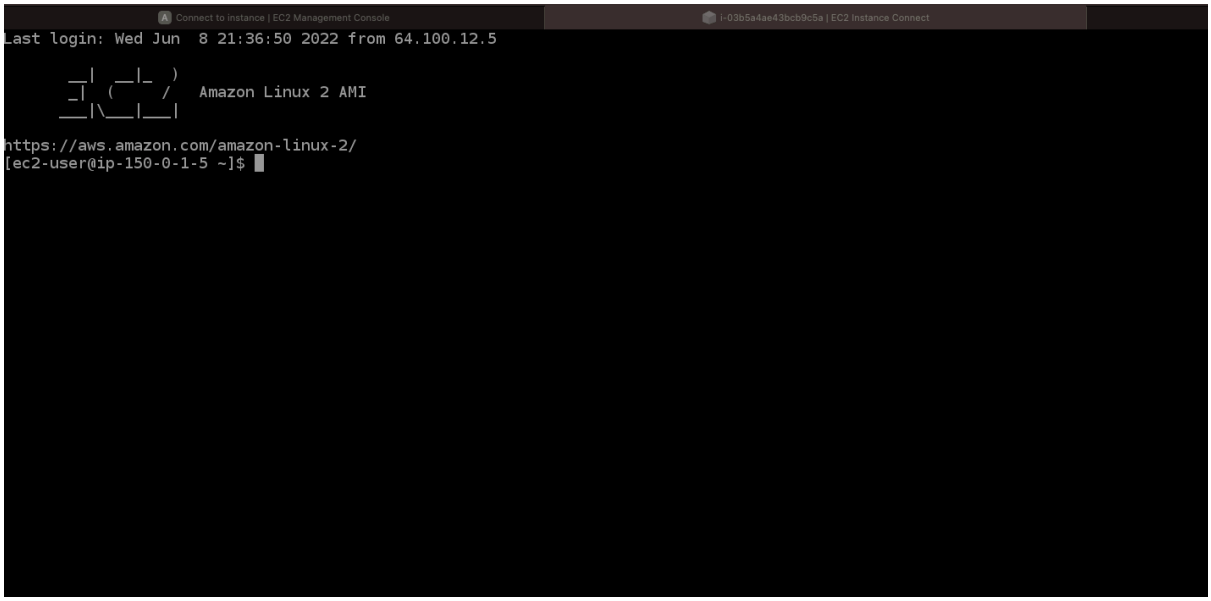
9. Next SSH into each of the EC2s. To make it easy and to not have to pass the SSH key each time, you could just use the ‘Connect’ Feature on the top when you click on your desired EC2



10. Select the EC2 Instance Connect option:



11. This will open an SSH session to your desired EC2:



12. Lastly head over to the single page website EC2 using SSH and see the running container:

Congratulations!

We have a backup single page site ready in case we run into any traffic issues on the premises now. We can now proceed with moving the whole application to the cloud and perform a DR scenario test in the next task.

Optional Task:

Spend some time inside the container in the single page EC2 to understand the running services using *'docker exec -ti <container name> bash'*. You can *curl* inside and outside the container, study the used ports, see the status of nginx and study exactly what our compose code set up.

- **What did we learn in this Task?**

    - We now know how to use the AWS Terraform provider to configure resources on the cloud.
    - We understand how to create the cloud components for a stretched tenant on Cloud APIC
    - We have learnt how to docker compose an image and set up a container with relevant parameters

Task 6: Migrate all application services to the cloud and test DR/Load Balancing scenario

[Estimated time: 15-20mins]

We've tested the cloud integration of our front-end service, which is working fine, taking care of our traffic spike issues. Our management has read about post-mortem stories of company websites having downtime periods and how much it cost those companies. Because of that, our management has approved the full migration of our new service-based website application to the cloud site to meet the DR use case.

Your task will be to update the Terraform scripts to stretch the DB and Backend EPGs to the cloud with the use of NDO and test the DR/Load Balancing scenario. We have already created the other 2 EPGs for the DB and Backend.

Step 1: Prepare the CI/CD pipeline that will stretch the entire application to the cloud

For this last task, we will no longer be using Gitlab to show you how you can execute pipelines straight from your CLI using Terraform. While Gitlab helps us automatically execute parts of our pipeline and helps understand dependencies and revision history all in one place, we will take a step back and explore some key features of Terraform when we run a pipeline manually from our CLI.

We will be leveraging our usual Terraform pipeline steps- an init, validate, plan and apply. Do remember that **you** will be tracking this pipeline and hence you are to check that the previous tasks in the pipeline have been successfully executed before you move to the next stage.

We will still however be using the features of git to ensure our config is backed up and revision history is updated.

Step 2: Prepare the Terraform script to stretch the entire application to the cloud

Now let's create a script to stretch all our application's containers onto the existing EC2s. Let's draw a parallel to our previous script and see how we would approach this task.

1. Create a new folder for this Task. We will not be pushing this task anywhere.

2. The first part of our Terraform code would be to somehow identify the EC2s created previously. In the previous task we used a resource to create AWS instances. However, here we don't need to create one, but just reference existing ones. Look for a relevant object in Terraform's AWS provider that can help with this.

   Now, if you are feeling lazy, you may hardcode the EC2 instance ID, but the best way to approach this task would be to figure out a logic to identify the EC2 and then take the exposed 'id' attribute from this. This is something we haven't added to our main GitHub repo to make it a bit more challenging, so we would love to see you implement it (do call us over if you manage to set this up and it would make us very happy).

3. Now that you have the object to identify the EC2s, we can now collect any relevant parameters from it like the public IP and use it in the subsequent steps.

4. Next you would need to expose port 80 in egress and port 443 in ingress as we did in the previous task for the other 2 containers now (DB and Backend).

5. We will use 3 null_resources with a provisioner like last time for each of the EC2s.

   Start with EC2 2 and 3 (DB and Backend) and here:
   a. Install yum and update it
   b. Install docker
   c. Start docker
   d. Provide privileges to /var/run/docker.sock
   e. Install docker compose
      Instructions for this can be found here:
      DigitalOcean - How to install and use Docker Compose on Ubuntu 20.04
   f. Git clone the repository https://github.com/achintya96/cisco-live-2022-aws-containers where all the composes and code is stored.
   g. Move inside directory 'ciscolive-containers' inside the main repo
   h. Copy the docker-compose-be.yml file to a new docker-compose.yml file for the backend EC2.
      In the DB EC2 you will copy docker-compose-db.yaml file to a new docker-compose.yaml
   i. *'Docker-compose up* -d' to run the actual docker compose.

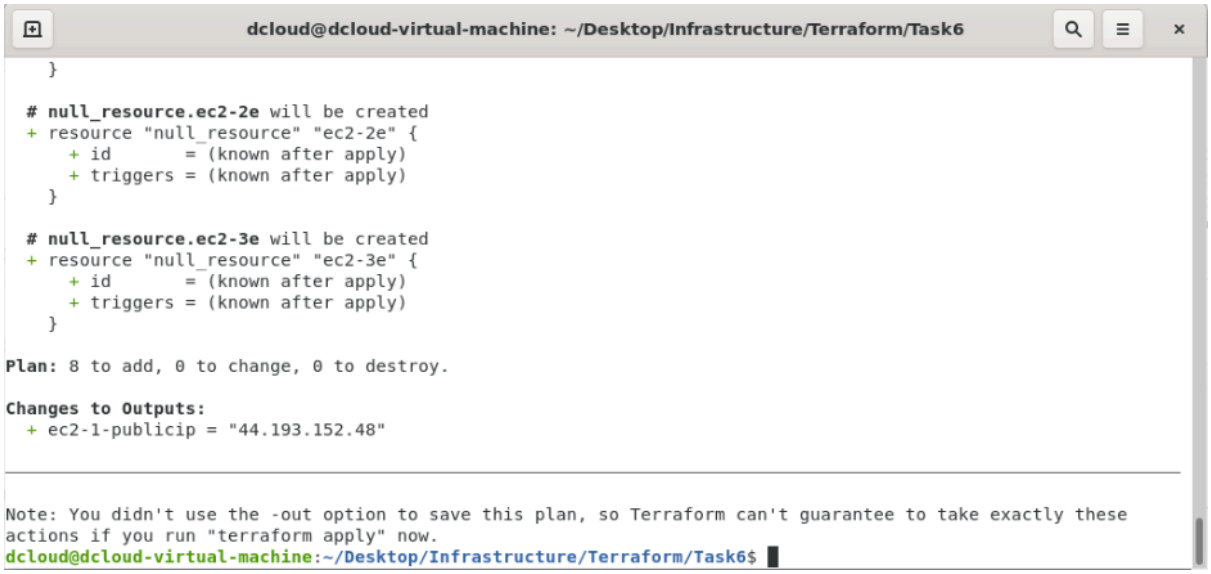
   Do remember that for each of these commands that requires a 'yes' prompt to install, we need to provide a -y flag so it does not wait for a response.

6. Now on the null_resource provisioner for the front end:
   a. Change directory to cisco-live-2022-aws/containers/singlepage and docker-compose down to shut the singe page we deployed in the previous task
   b. Cd to ~/cisco-live-2022-aws-containers/ciscolive-containers
   c. Copy the docker-compose-fe.yaml to a docker-compse.yaml
   d. Docker-compose up -d

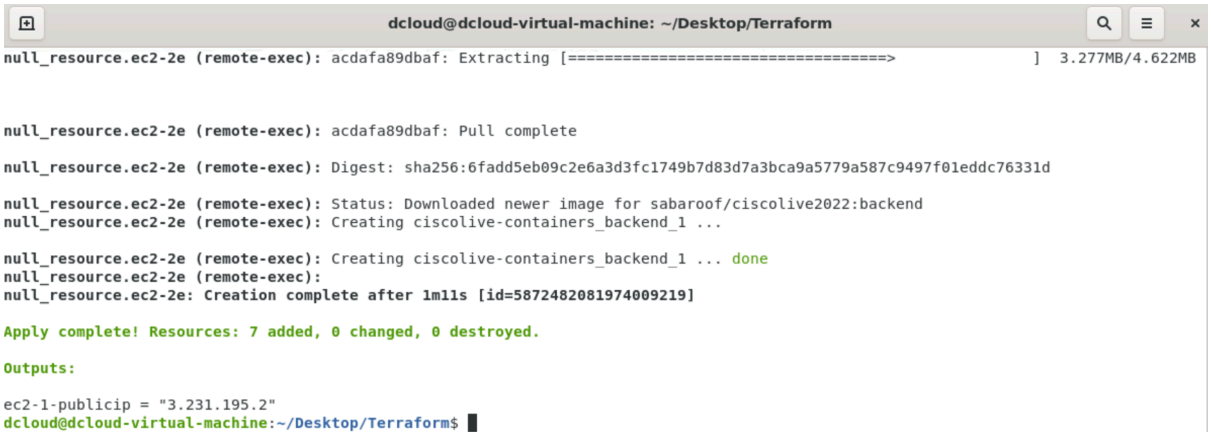
Step 3: Verify the Terraform script against our pipeline and execute it

1. Let's run our pipeline with this script so far and see if we are successfully able to remove the front end from the EC2. Start with an terraform init, proceed with a terraform validate, terraform plan and terraform apply. Proceed only if you see a success in the previous stage and do take a note of what exactly is changing in each stage.

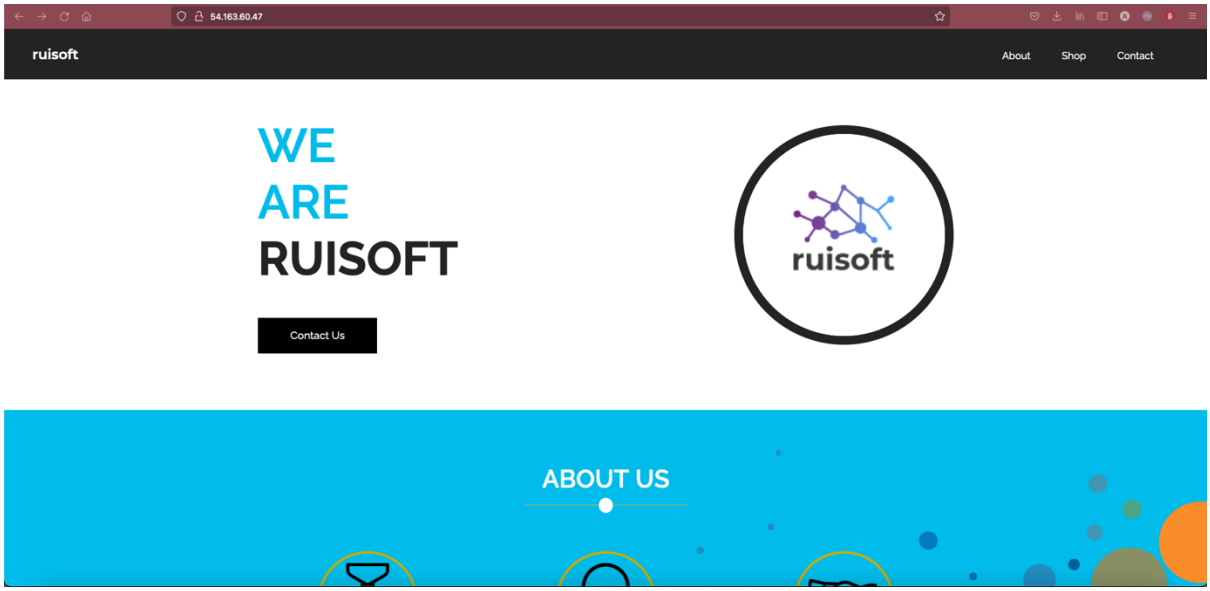
Plan:



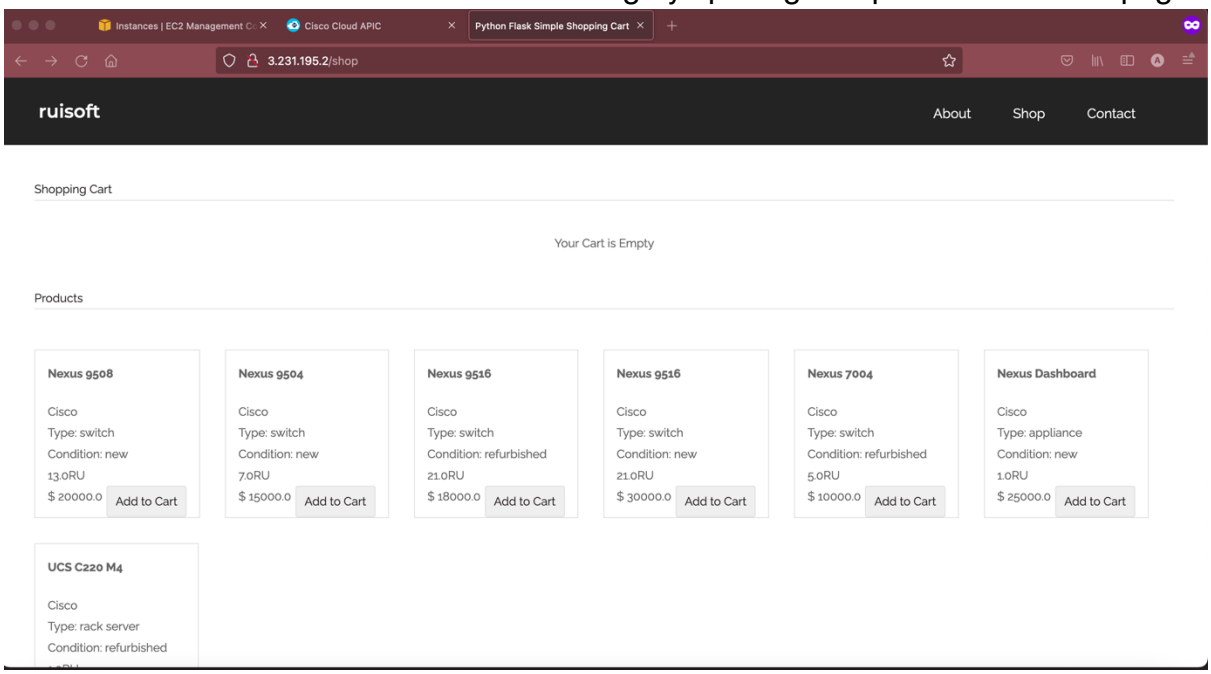
Output of the Apply:



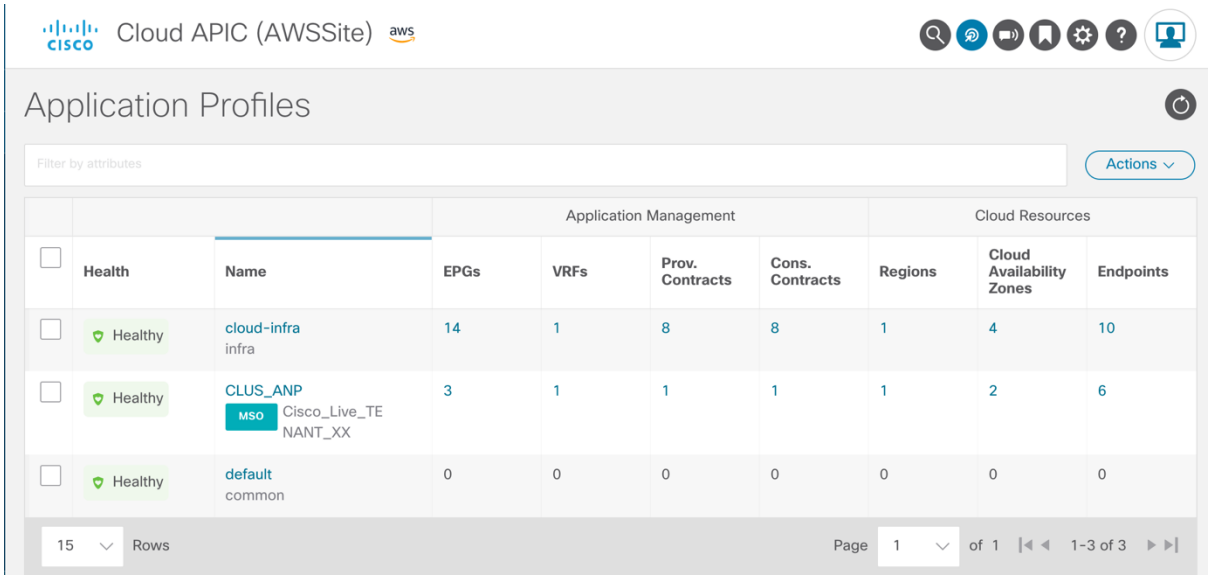
2. SSH into the EC2s and confirm when all containers are up and running and test by heading to the public IP of the front end to check if it is accessible.



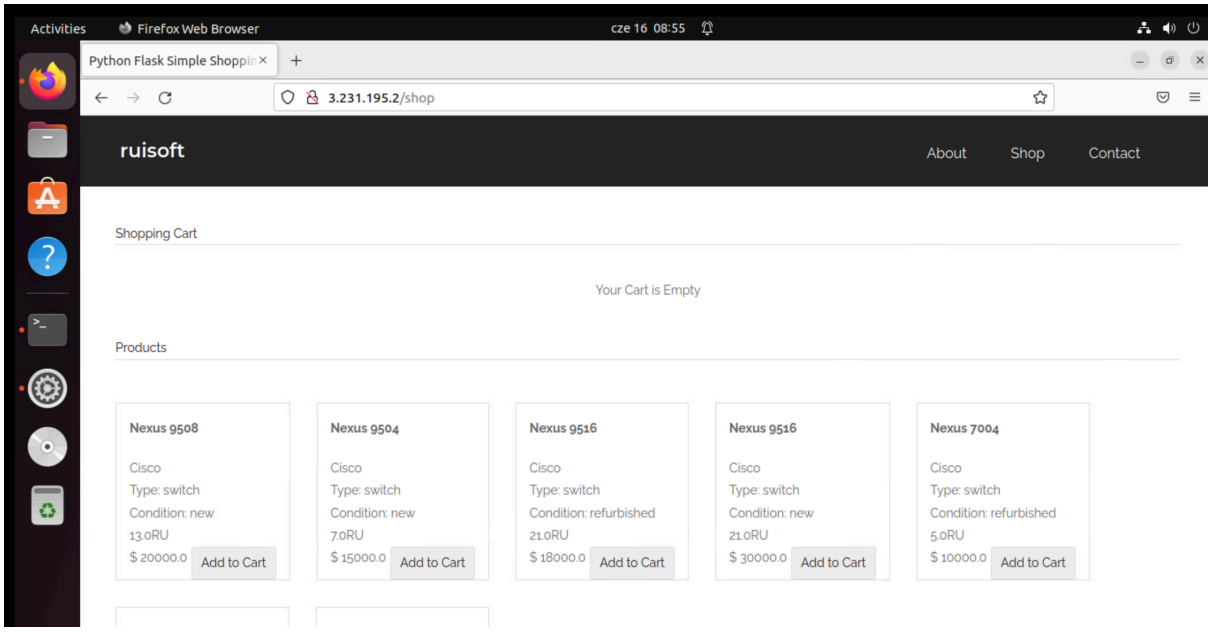
And check if our back end and DB are working by opening 'Shop' tab on the web page.



Also check on the cloud APIC if anything has changed from the interface perspective

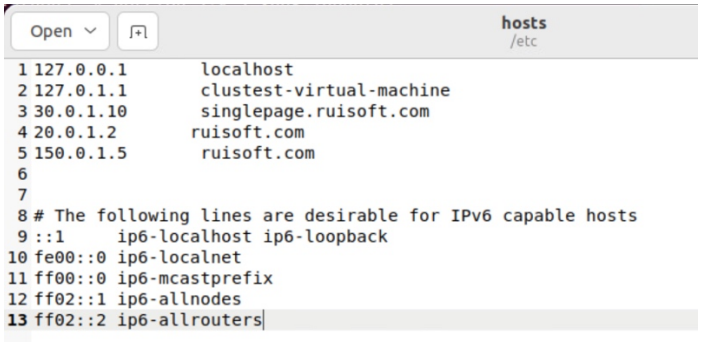


Additionally check if the TestVM can reach the online webpage too.



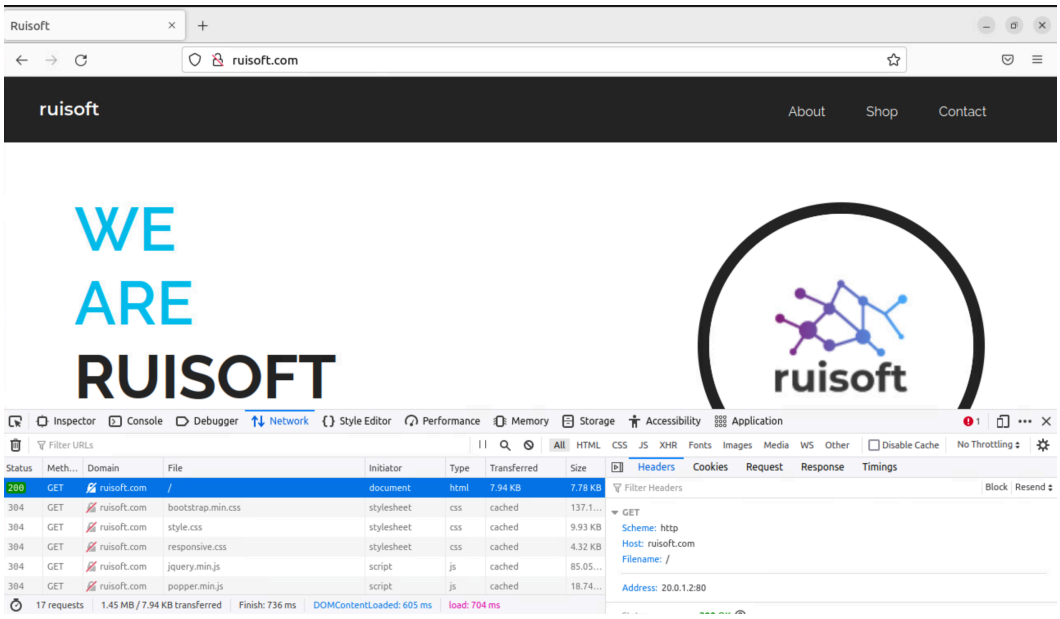
Step 4: Implement and verify the different DR scenarios by forcefully shutting the on-prem components and studying the behaviour

1. In the etc/hosts file on the Test VM, add 2 entries for ruisoft.com, one pointing to the Cloud IP and one to the on prem IP.



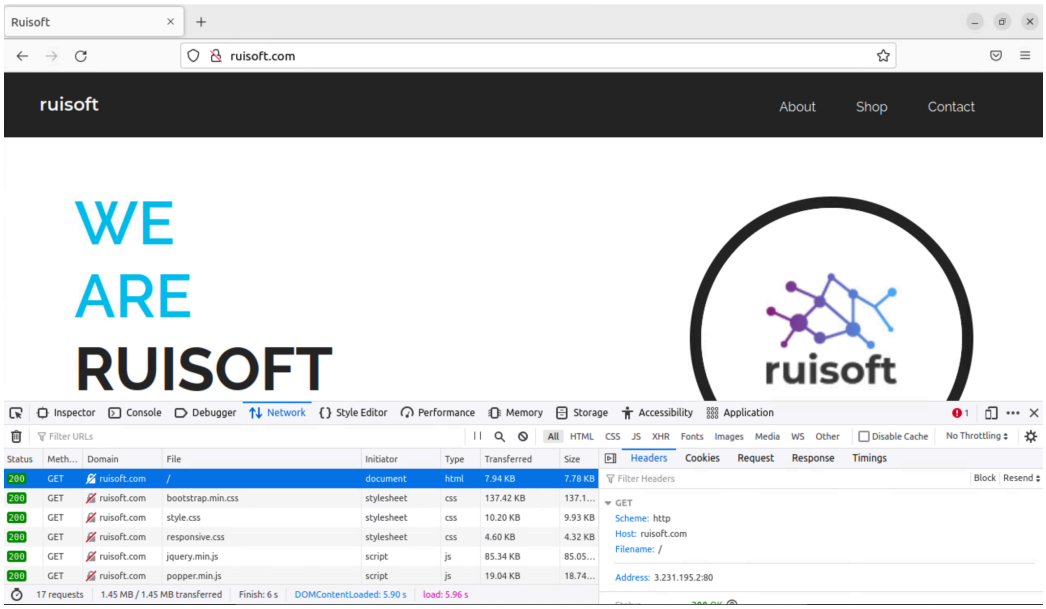
2. Go to browser, type ruisoft.com and see if you can access the site. Use the browser developer tools such as inspect element and networking to check which host server the version of the site you are seeing now.

3. Check on another browser and Inspect element -> Networking to see which version of the site you are seeing.



4. Either unbind or shut the first interface of the ProdVM and see if the site is still reachable.

5. Once again inspect element and check which version of the website you are reaching.



This tests our DR/Load balancing scenario.

Congrats on completing all the tasks!

- **What did we learn in this Task?**
    - We have successfully stretched an application to the cloud by using Terraform by leveraging docker composes.

1. We learnt a simple way to set up High Availability/Disaster Recovery and using our cloud site to the fullest.

Optional challenges:

- write an Ansible playbook that will utilize REST API modules for ACI to verify Faults and compare them before and after the deployment. Ansible allows you to register the outputs to files. You can query ACI for Faults under your Tenant using Ansible with the use of Cisco.Aci collection with the REST API module, do that during the pre-deploy stage, and you can define a post-deploy stage which will utilize the same Ansible playbook checking Faults after you've hit the deploy stage. In case the output files would differ (the latter one introducing new faults) you can have an Ansible playbook rolling back to your snapshot.
- you can explore parent-child pipelines, often used for monorepos. That would mean you'd have to consolidate your Terraform and your Ansible repositories.
- delete a resource you created in task 6 (maybe a security group rule) and see if terraform can detect it when you run a plan and create it again by doing an apply