Course-End Project

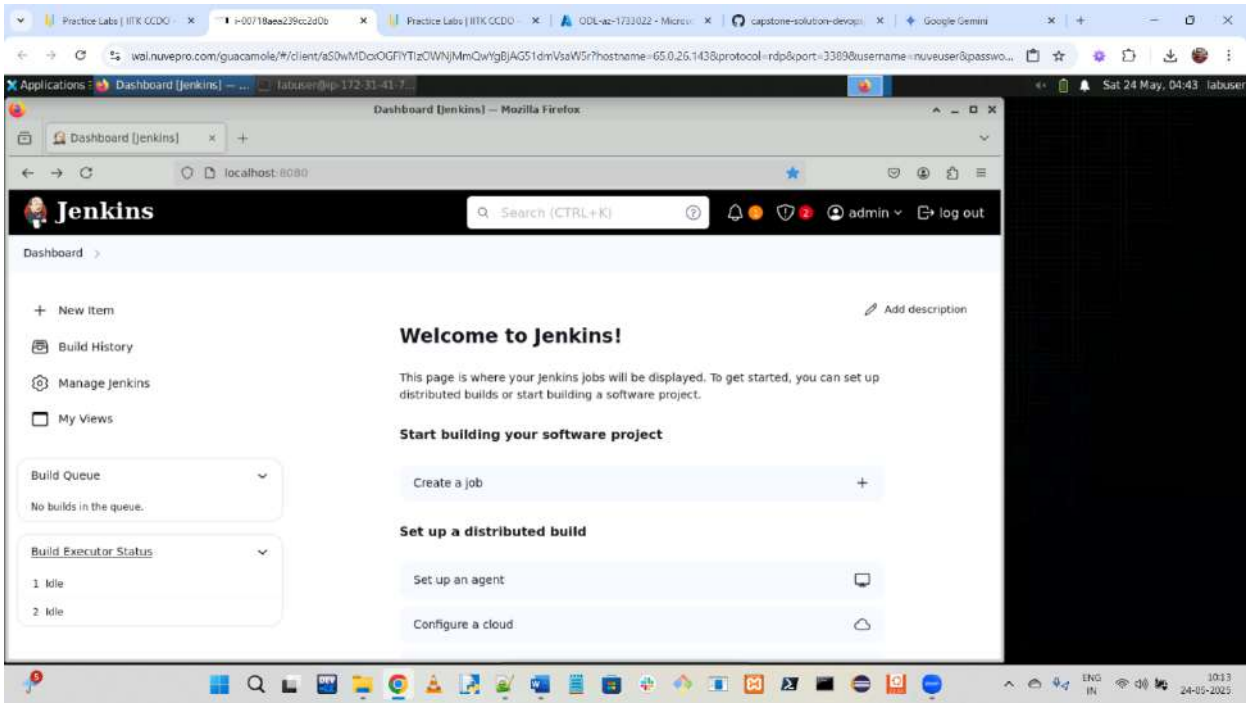
Application Deployment to Multi-Cloud

Steps to be followed:

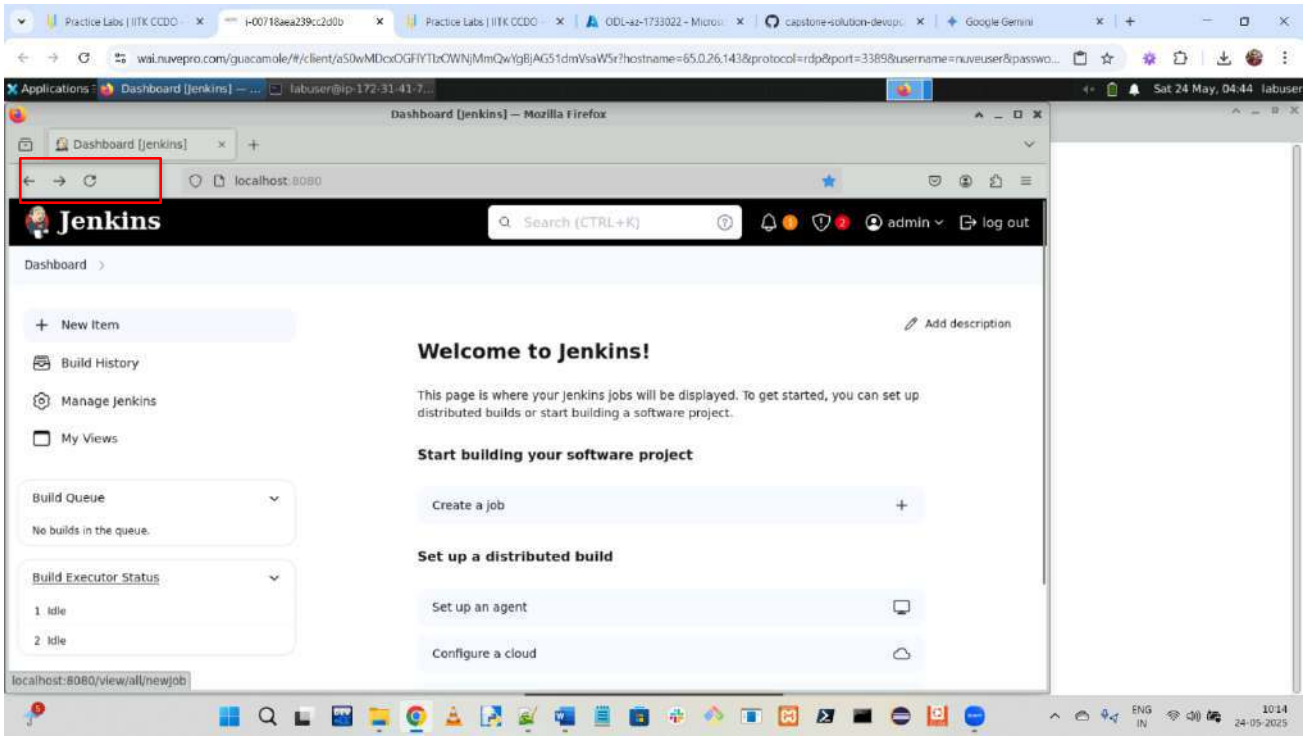
1. Create Jenkins pipeline for CI of Java application
2. Configure Ansible to connect AWS and Azure VMs
3. Configure Docker in Jenkins for CI/CD
4. Configure Ansible on Simplilearn lab
5. Validate Deployment

Step 1: Create Jenkins pipeline for CI of Java application

1.1 Navigate to the DevOps lab where Jenkins is preinstalled, Open a browser and access Jenkins at **http://localhost:8080/**



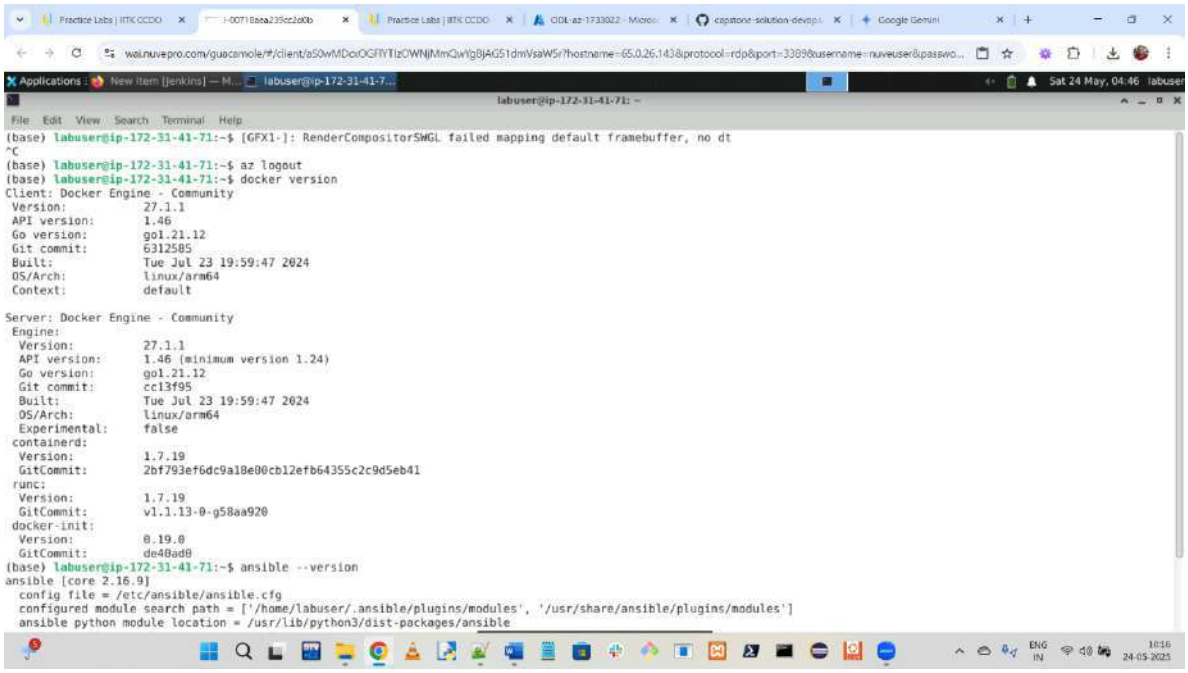
1.2 Open Jenkins and click on **New Item** to create a new Jenkins job



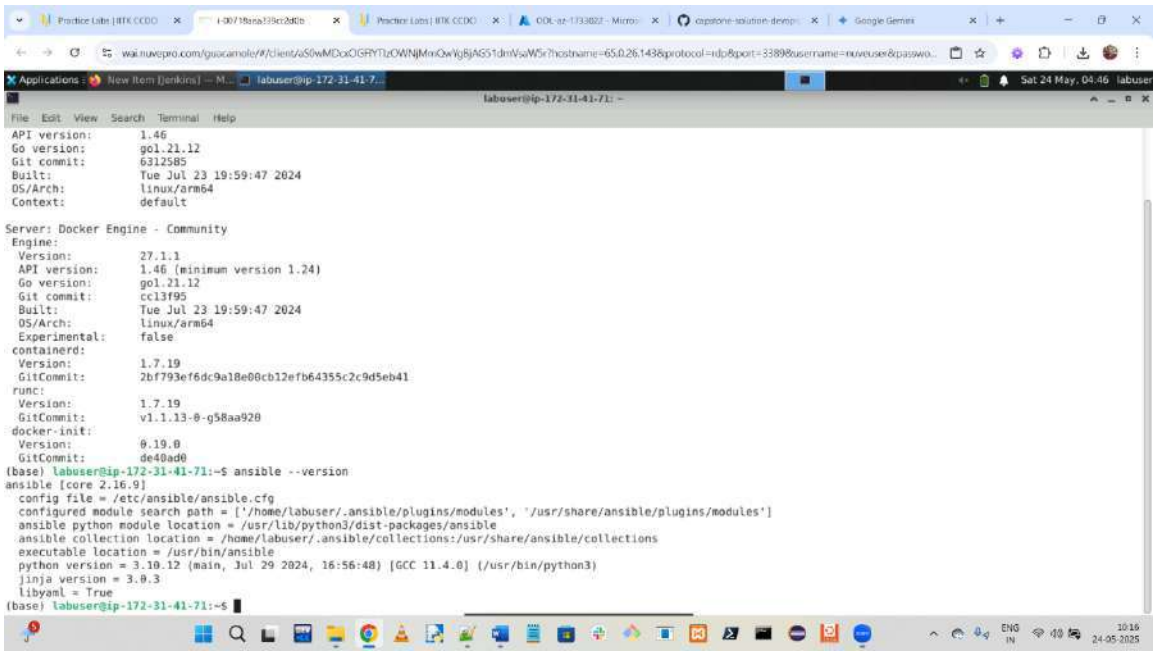
Before proceeding further open terminal and then validate if ansible, docker and maven runtimes are installed on VM

1.3 Execute the following command to check the Docker version and confirm that Docker is correctly installed on your system

docker version



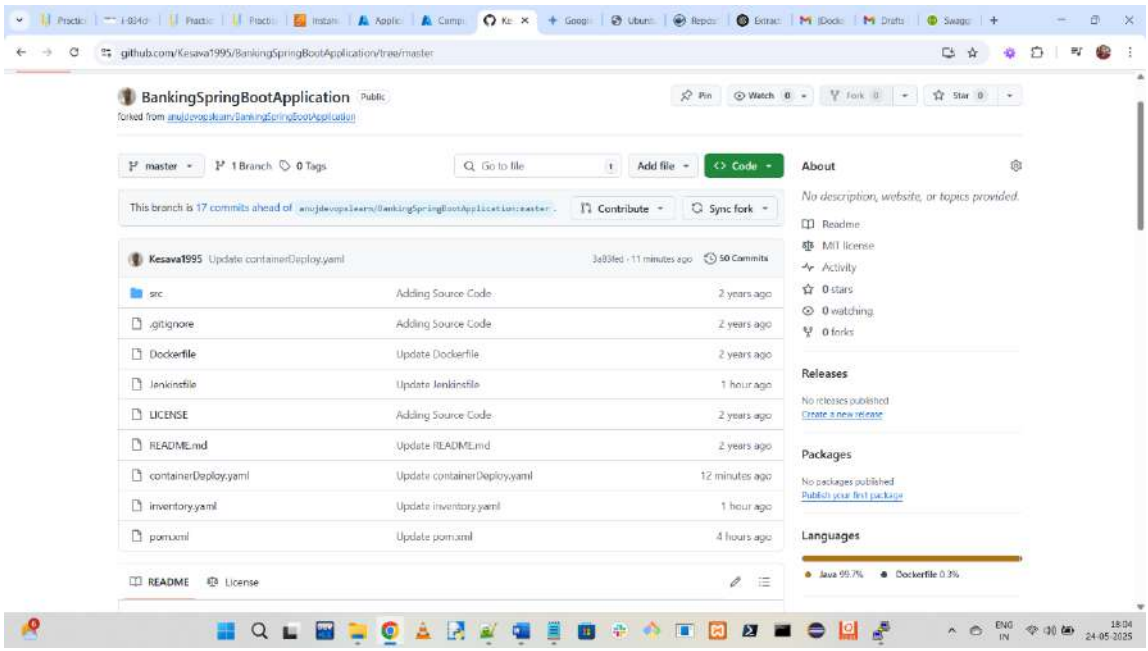
1.4 Execute the following command to check the Ansible version and configuration details **ansible version**



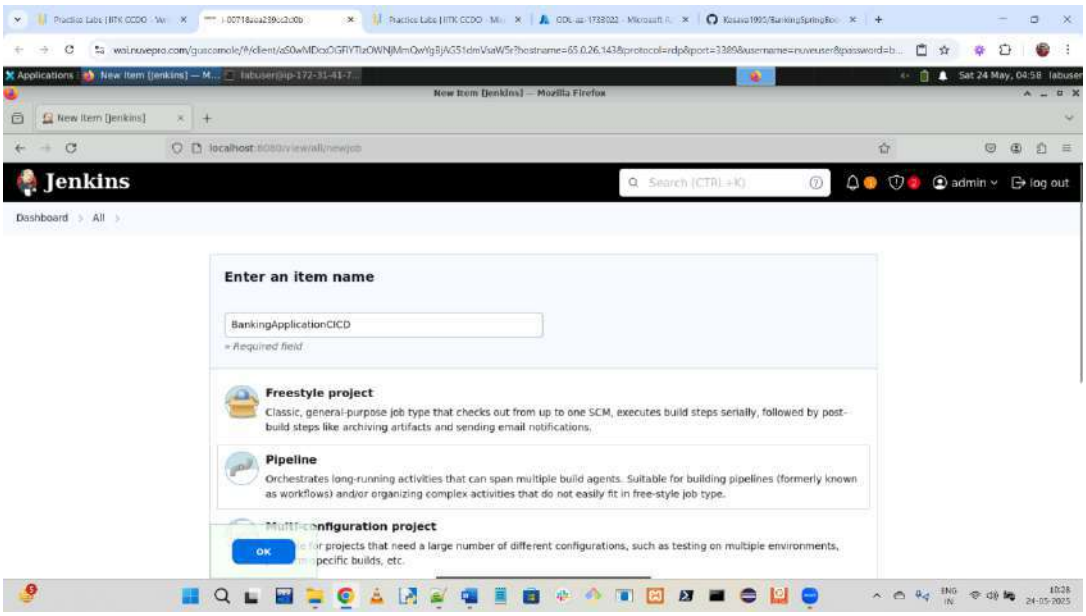
Step 2: Configure Ansible to connect AWS and Azure VMs

2.1 Fork the GitHub repository below to execute the Jenkins pipeline, and validate that Maven scripts are present in the code repository for source code validation
https://github.com/Kesava1995/BankingSpringBootApplication

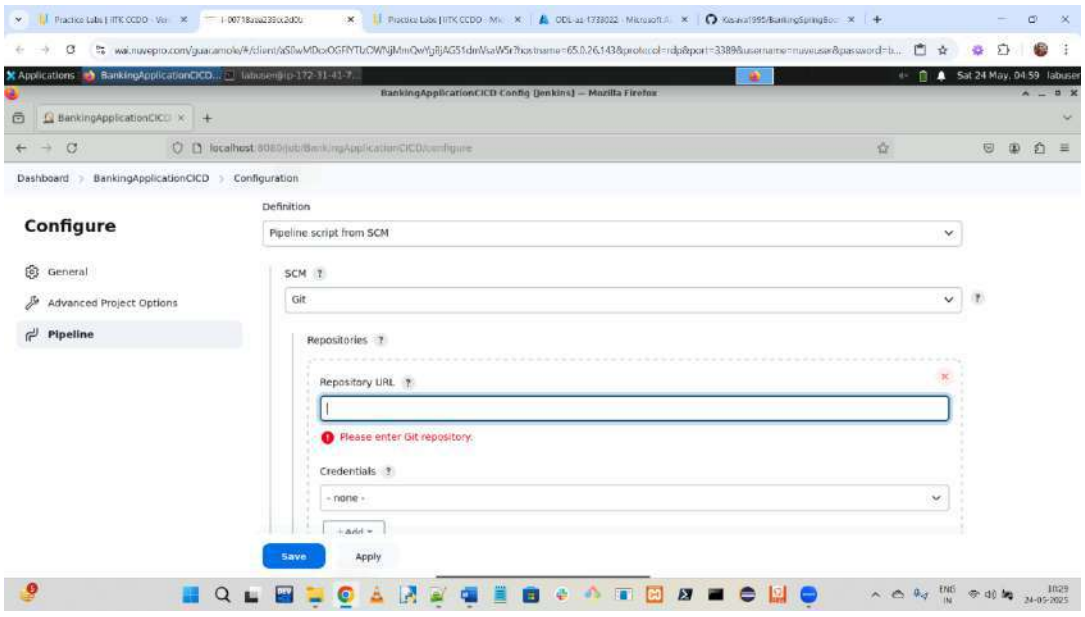
2.2 Access Jenkins application and click on **New Item** to create a new Jenkins job



2.3 Select desired Jenkins pipeline job type and fill in job name as per project requirement

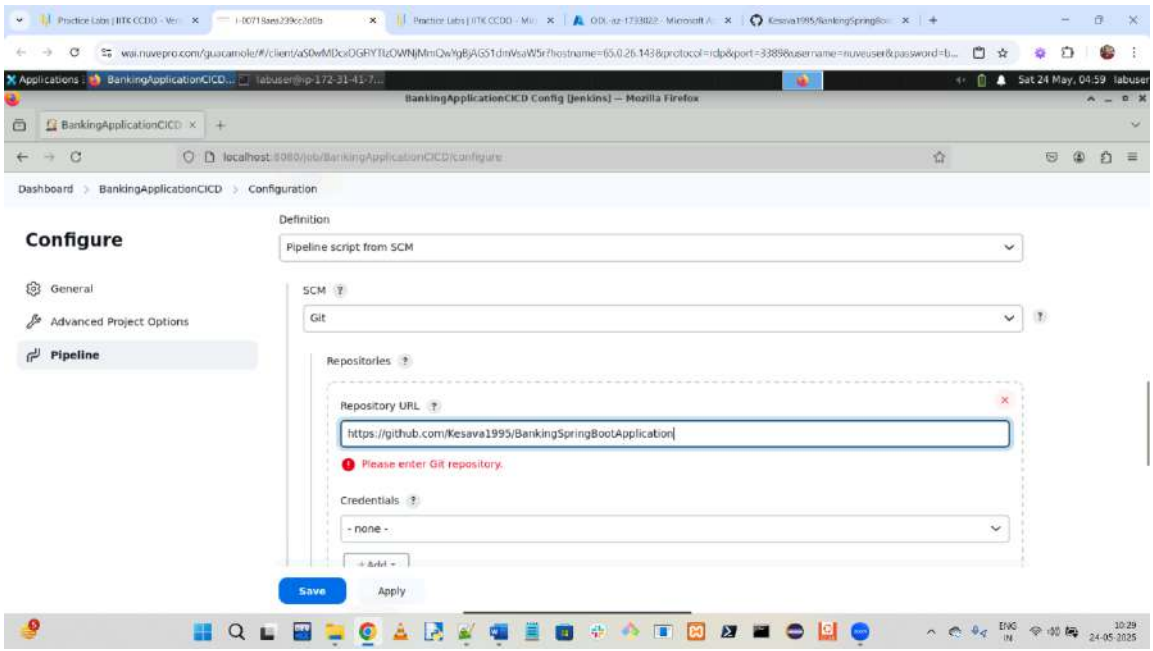


2.4 In your Jenkins project configuration, set the Pipeline Definition to **Pipeline script from SCM,** select SCM as **Git**, and enter the Repository URL where the pipeline script is stored. Once configured, click **Save** to apply the changes, enabling Jenkins to fetch the pipeline script directly from the specified Git repository



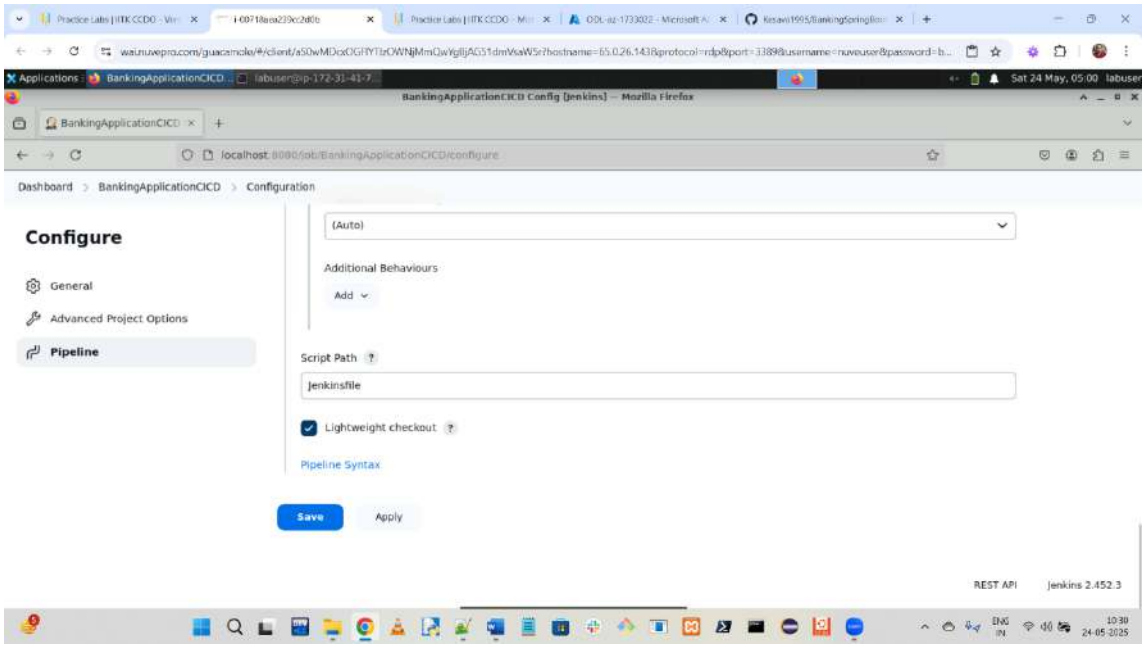
2.5 Configure below code repository or forked repository in Jenkins pipeline to checkout Pipeline script from Git version control system

https://github.com/Kesava1995/BankingSpringBootApplication



Note: In above code repository we are using default Jenkins pipeline name, but we can modify in case we are using a different pipeline name

2.6 Set the Script Path to **Jenkinsfile** and enable Lightweight checkout in your Jenkins pipeline configuration, then click **Save** to apply these settings and ensure Jenkins can locate and load the pipeline script efficiently



2.7 Create a **Jenkinsfile** in your code repository and add Code Checkout stage in the pipeline script

```
node{

   def tag, dockerHubUser, containerName, httpPort = ""

   stage('Prepare Environment'){
      echo 'Initialize Environment'
      tag="3.0"
          withCredentials([usernamePassword(credentialsId: 'dockerHubAccount', usernameVariable:
'dockerUser', passwordVariable: 'dockerPassword')]) {
                    dockerHubUser="$dockerUser"
      }
          containerName="bankingapp"
          httpPort="8989"
   }

   stage('Code Checkout'){
      try{
         checkout scm
      }
      catch(Exception e){
         echo 'Exception occured in Git Code Checkout Stage'
         currentBuild.result = "FAILURE"
      }
   }
   stage('Maven Build'){
      echo '--- Diagnosing Maven Build ---'
      echo '1. Current Working Directory:'
      sh 'pwd'
      echo '2. Contents of pom.xml:'
      sh 'cat pom.xml' // CRITICAL: This will show us what pom.xml Jenkins is actually using
      echo '3. Java Version on Jenkins Agent:'
      sh 'java -version' // CRITICAL: Confirm the JDK version
      echo '4. Running Maven with debug logging:'
      sh 'mvn clean package -X' // CRITICAL: This will provide verbose output
      echo '--- End Diagnosis ---'
   }
```

```
   stage('Docker Image Build'){
      echo 'Creating Docker image'
      sh "docker buildx build --platform linux/amd64 -t $dockerHubUser/$containerName:$tag --load ."
   }

   stage('Publishing Image to DockerHub'){
      echo 'Pushing the docker image to DockerHub'
      withCredentials([usernamePassword(credentialsId: 'dockerHubAccount', usernameVariable:
'dockerUser', passwordVariable: 'dockerPassword')]) {
                  sh "docker login -u $dockerUser -p $dockerPassword"
                  sh "docker push $dockerUser/$containerName:$tag"
                  echo "Image push complete"
      }
   }
         stage('Ansible Playbook Execution'){
                  withCredentials([string(credentialsId: 'ssh_password', variable: 'password')]) {
                           sh "export ANSIBLE_HOST_KEY_CHECKING=False && ansible-playbook -i
inventory.yaml containerDeploy.yaml -e httpPort=$httpPort -e containerName=$containerName -e
dockerImageTag=$dockerHubUser/$containerName:$tag -e ansible_password=$password -e
key_pair_path=/var/lib/jenkins/server.pem --become"
                  }
         }
}
```

Note: The Jenkins file and all the other files are present in the GitHub repository

2.8 Below is the build automation stage which will invoke maven command to perform compilation, test execution and packaging

```
stage('Maven Build'){
   sh "mvn clean package"
}
```

Note: We need to integrate Docker image build process within your Jenkins pipeline, so we must create a Dockerfile build script which will help in building custom Docker image

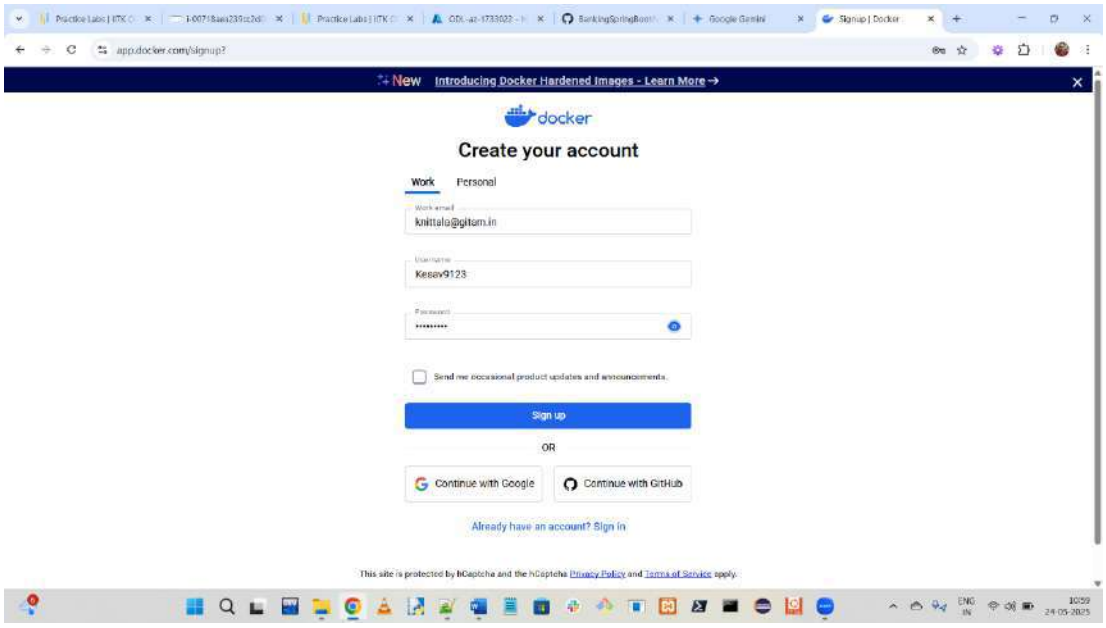
2.9 Create Dockerfile script for Backend:

```
FROM openjdk:11
```

```
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
EXPOSE 8989
ENTRYPOINT ["java","-jar","/app.jar"]
```

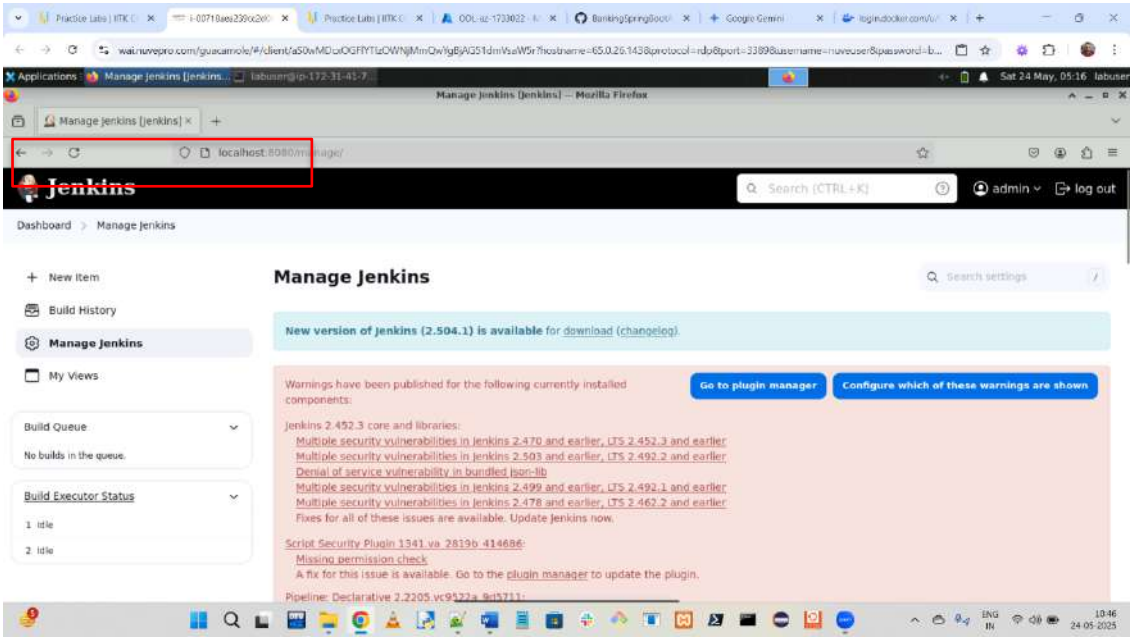
Note: Once Jenkins job configuration is done, we must create a Docker credential which will be used to push Docker image to Docker hub

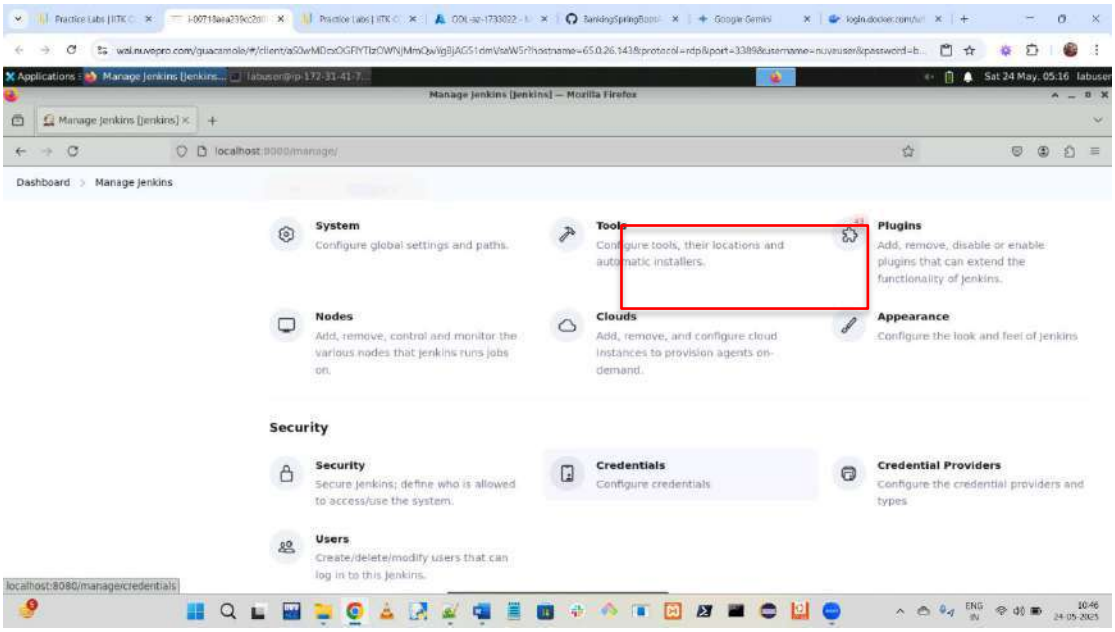
2.10 Navigate to Docker hub and create a new free account and keep Docker hub username and password with you before configuring Jenkins credential
https://hub.docker.com/

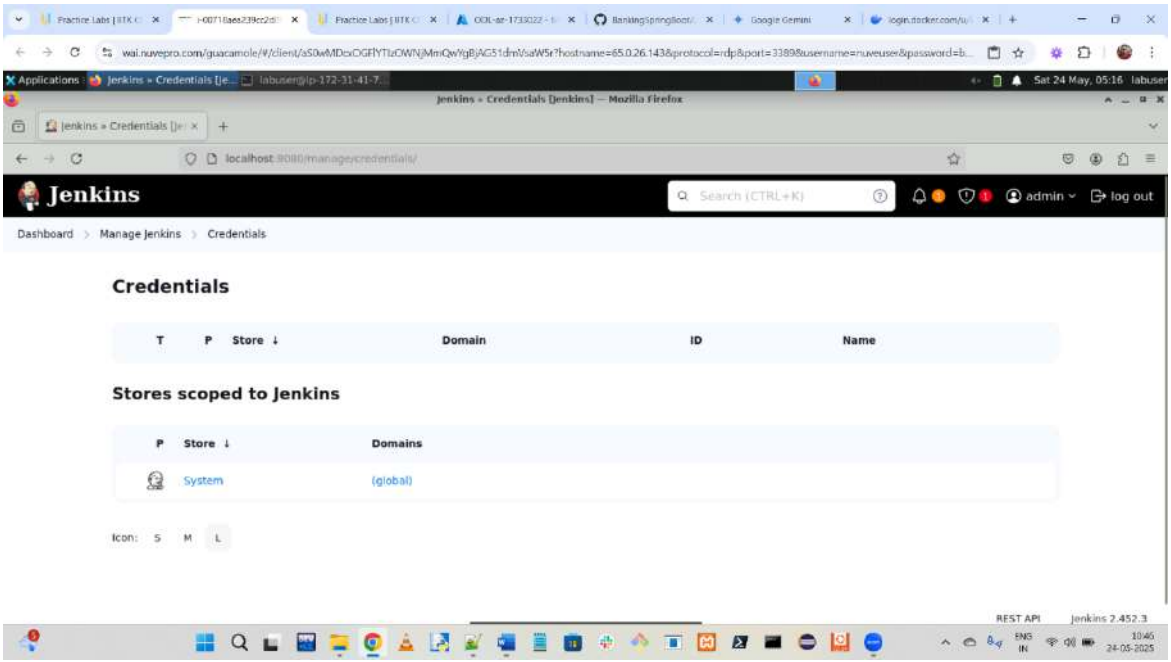


Step 3: Configure Docker in Jenkins for CI/CD

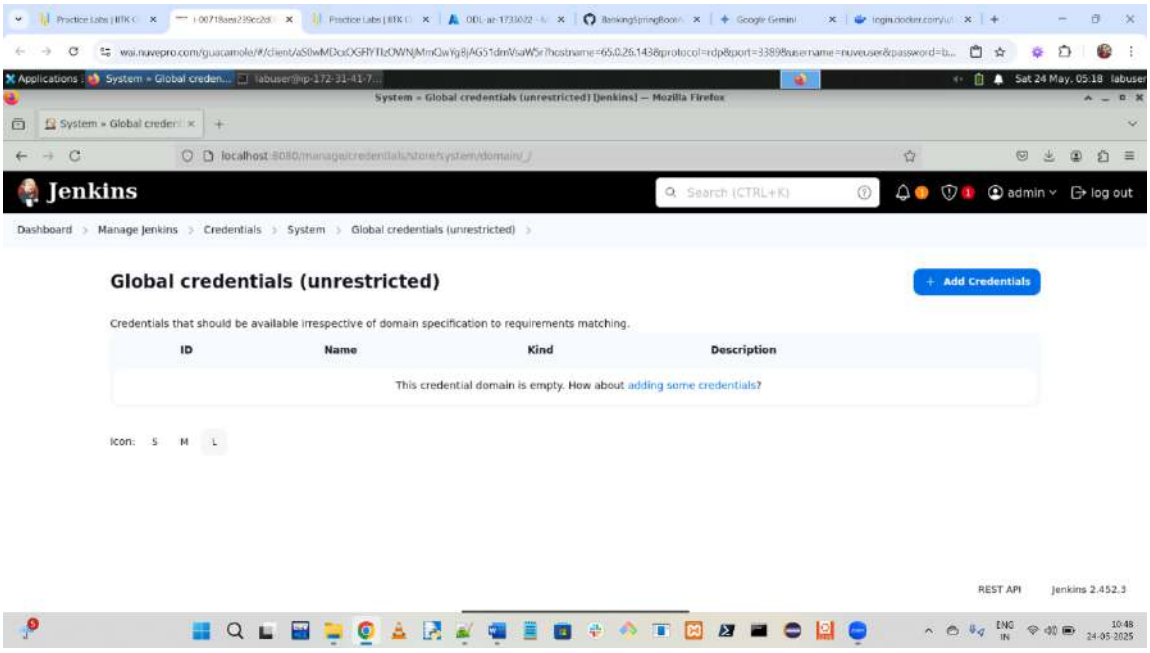
3.1 Once Docker hub account is created, navigate to **Manage Jenkins** and then **Credentials** and then select **global** domain

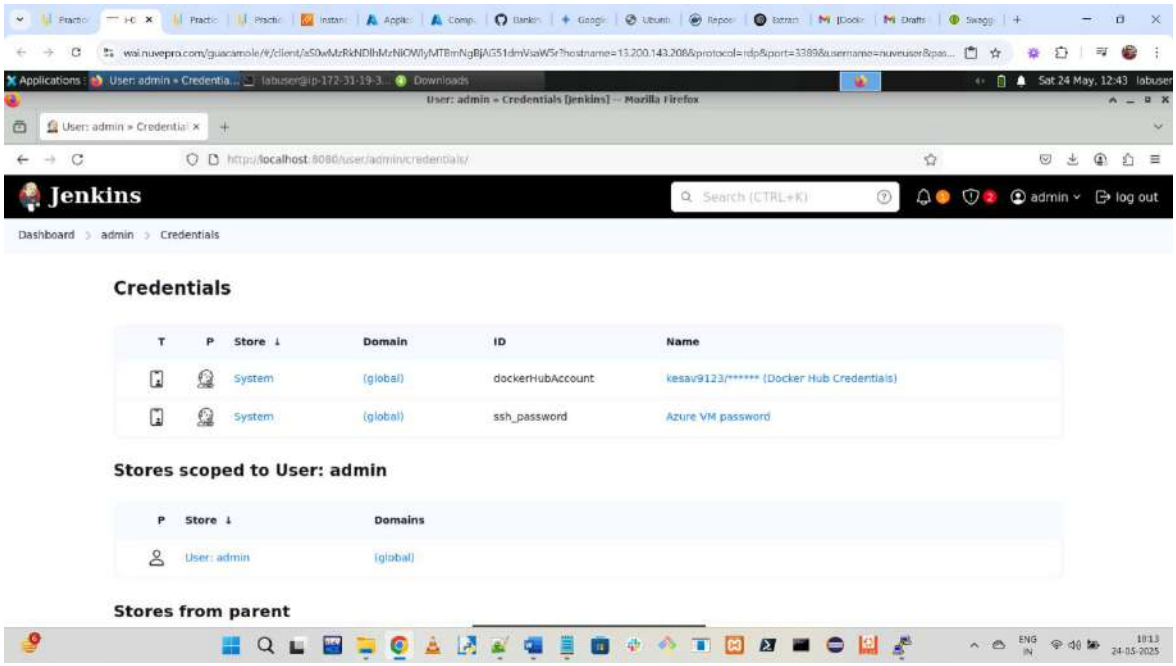






3.2 Click on **Add Credentials** to create new Docker Hub credentials with the details provided below and click on **Create**





Note: Credential id in below screenshot should match what you have mentioned in Jenkins pipeline

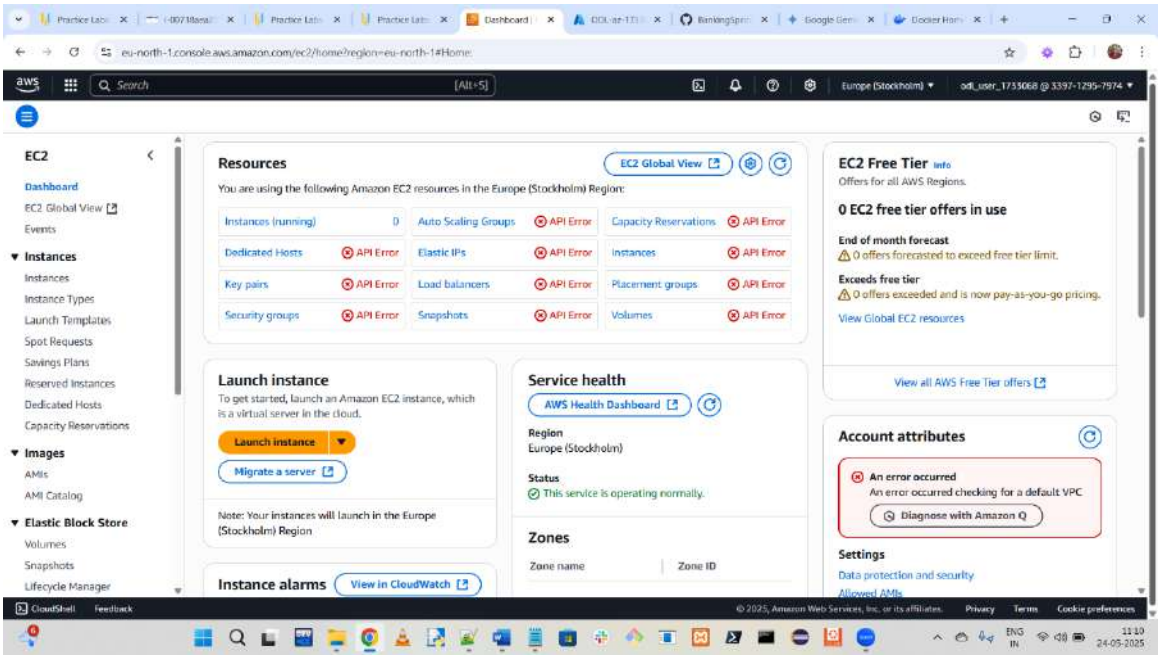
3.3 Once Docker image is built, we can integrate Docker image publish stage to publish Docker image to Docker hub registry

```
stage('Publishing Image to DockerHub'){
      echo 'Pushing the docker image to DockerHub'
      withCredentials([usernamePassword(credentialsId: 'dockerHubAccount', usernameVariable:
 'dockerUser', passwordVariable: 'dockerPassword')]) {
            sh "docker login -u $dockerUser -p $dockerPassword"
            sh "docker push $dockerUser/$containerName:$tag"
            echo "Image push complete"
      }
 }
```

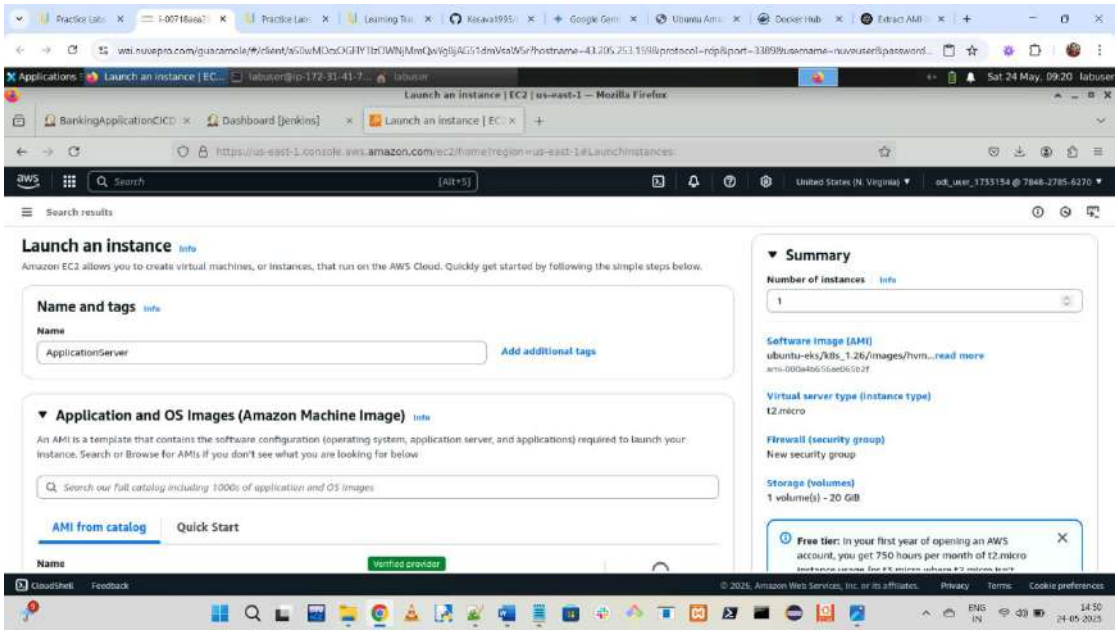
Step 4: Configure Ansible on Simplilearn lab

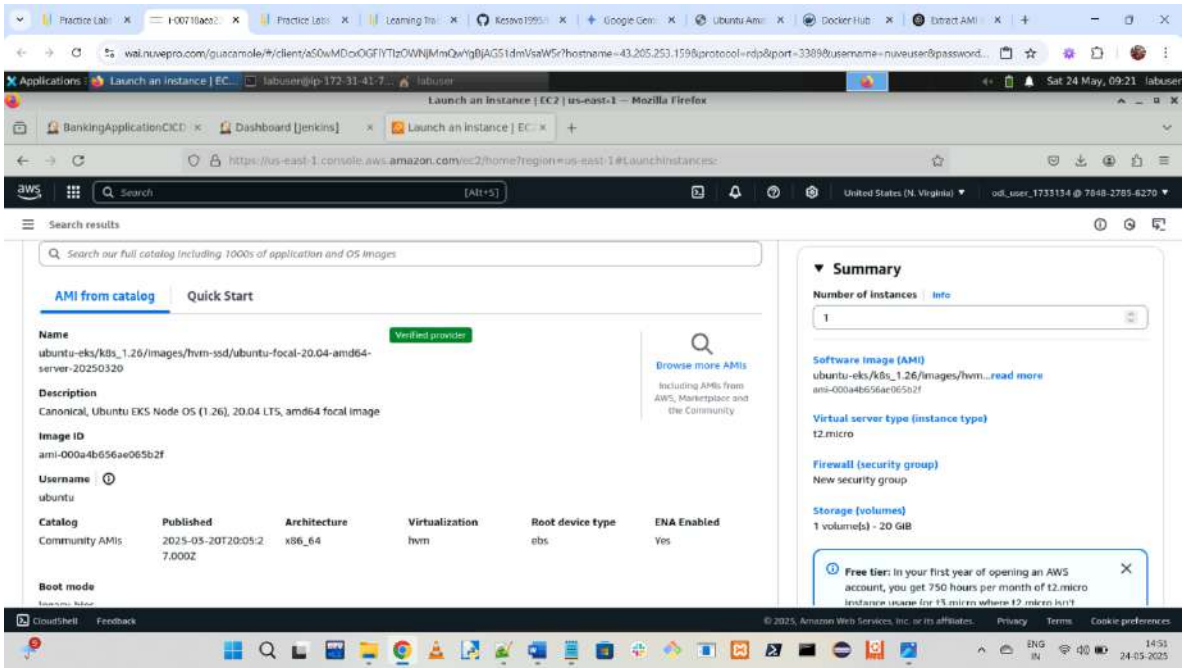
Note: we need to configure Ansible Controller on Simplilearn Lab from where it would connect to both AWS and Azure VM

4.1 Navigate to AWS Console and EC2 service to launch VM using below steps

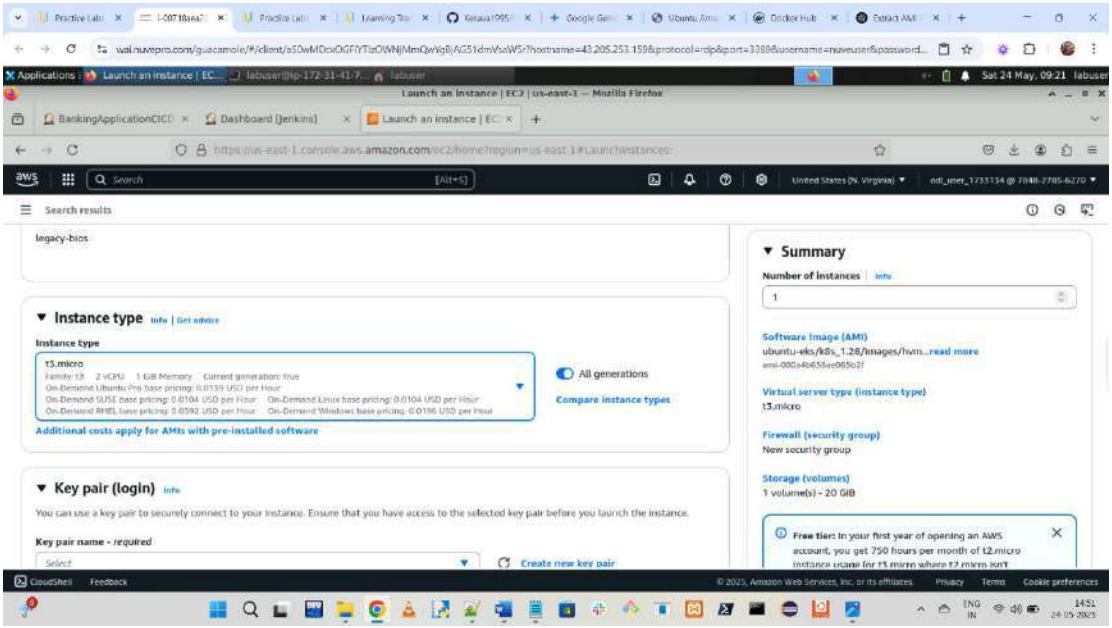


4.2 Provide name as **ApplicationServer** and Ubuntu 24.04 which will be used to launch VM



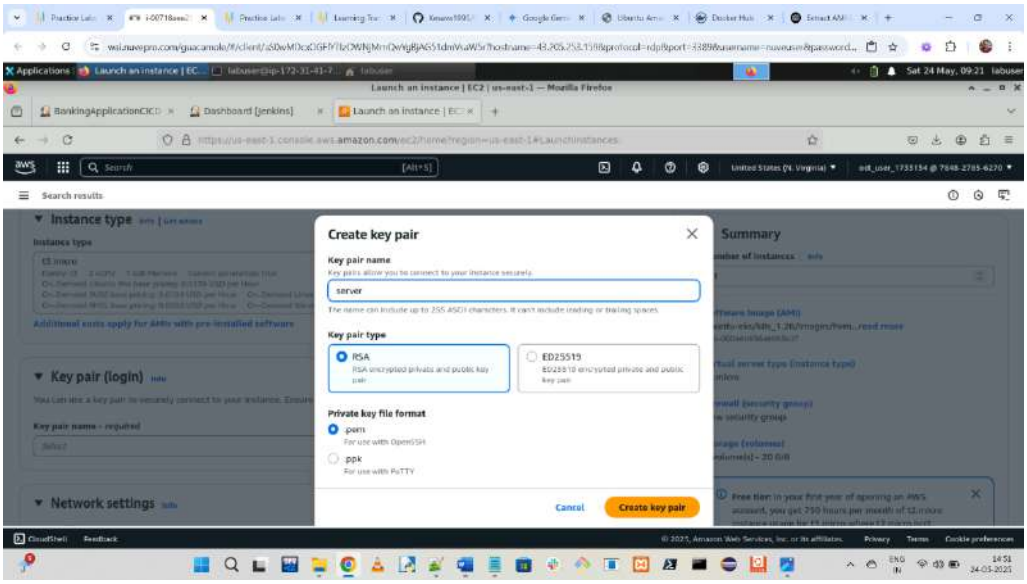


4.3 Select Instance type as **t3.micro**



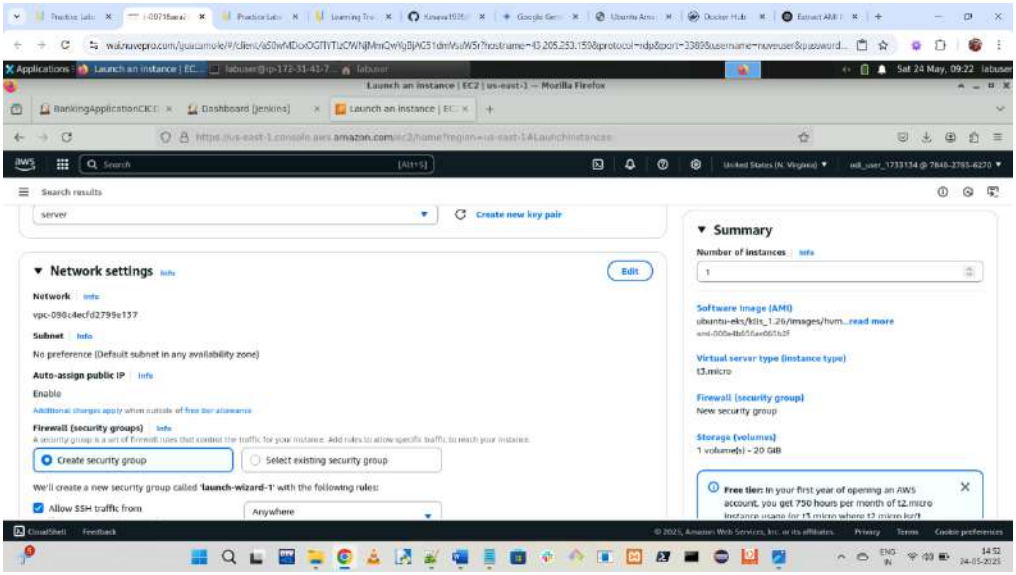
Next, we need to create key pair which will be used to perform SSH connectivity with server

4.4 Enter a key pair name as **server**, select the key pair type, choose the private key file format (.pem or .ppk), and click **Create key pair**. Save the private key securely for future instance connections.

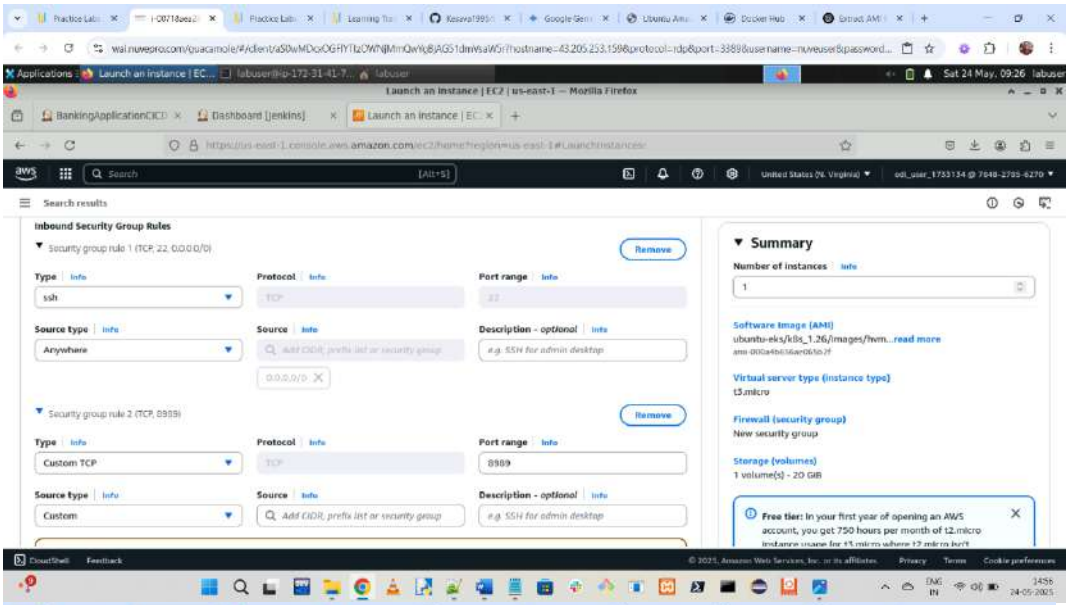


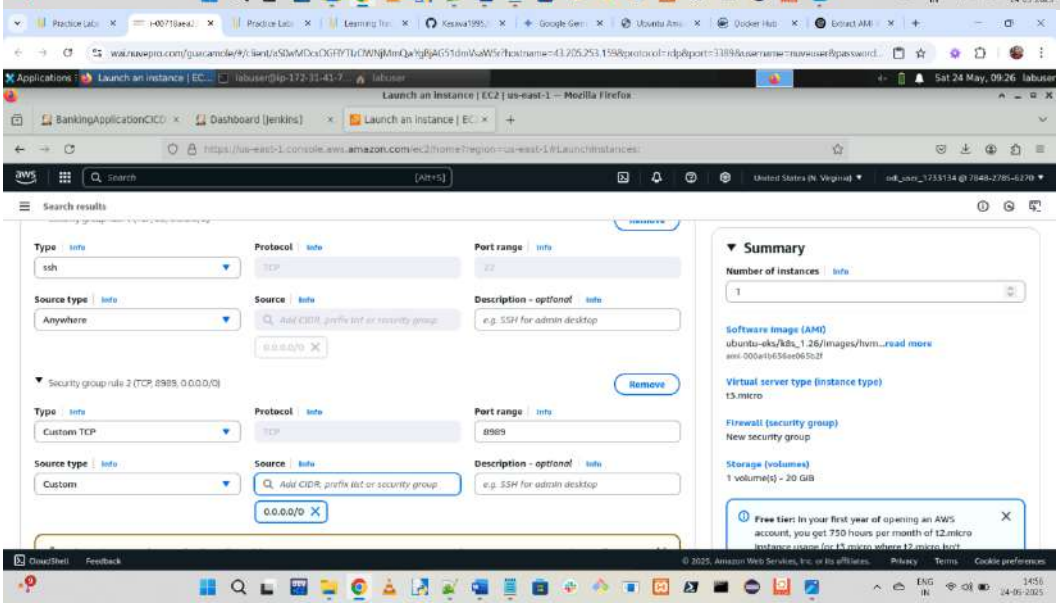
Note: This will download pem file locally, which we need to preserve since we would be using this key pair to establish connectivity from Ansible. Select this key pair once its created

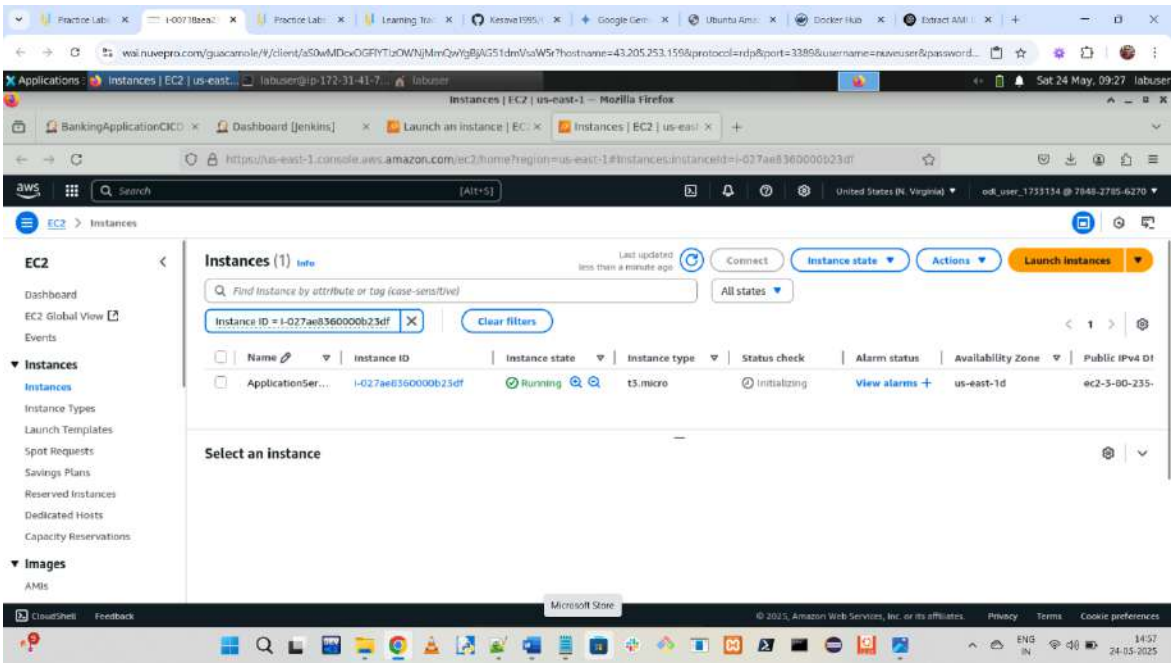
4.5 Click on Edit under network settings and add Application port in security group to allow that port.



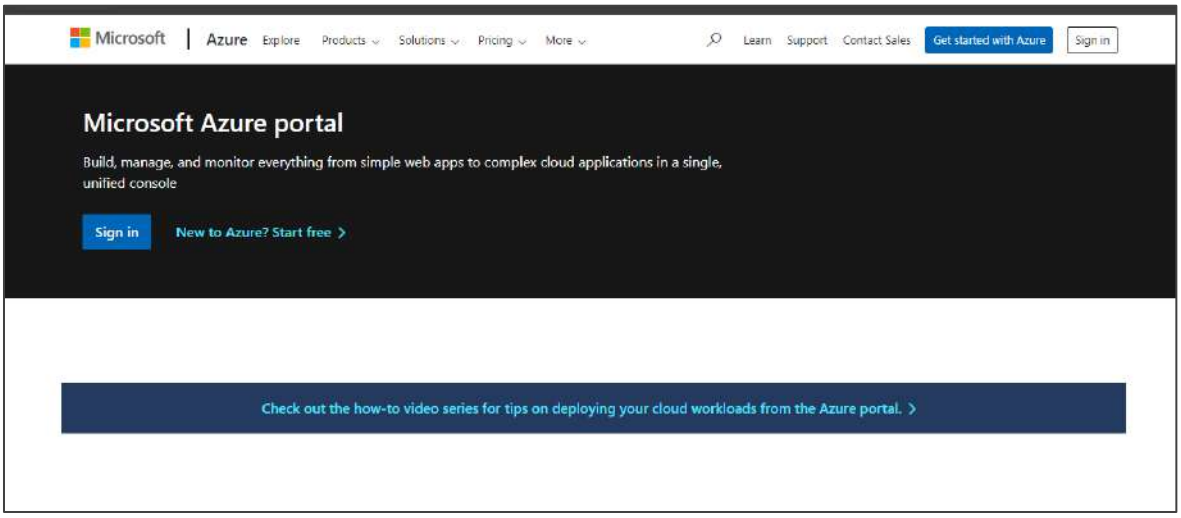
4.6 Create a security group named **sg_application**, set the description to **Application Security Group**, and add inbound rules for **SSH** on port 22 from Anywhere and Custom TCP on port **8989** from **0.0.0.0/0**. Finally, click **Launch instance** to start the instance with these settings.


Launch an instance | EC2 | us-east-1 — Mozilla Firefox
Inbound Security Group Rules
Security group rule 1 (TCP, 22, 0.0.0.0/0)
Type
ssh
Protocol
Port range
Source type
Anywhere
Source
Description - optional
e.g. SSH for admin desktop
Remove
Security group rule 2 (TCP, 8989)
Custom TCP
8989
Custom
Add CIDR, prefix list or security group
Summary
Number of instances
1
Software Image (AMI)
ubuntu-eks/k8s_1.26/images/hvm...read more
Virtual server type (instance type)
t3.micro
Firewall (security group)
New security group
Storage (volumes)
1 volume(s) - 20 GiB
Free tier: In your first year of opening an AWS account, you get 750 hours per month of t2.micro

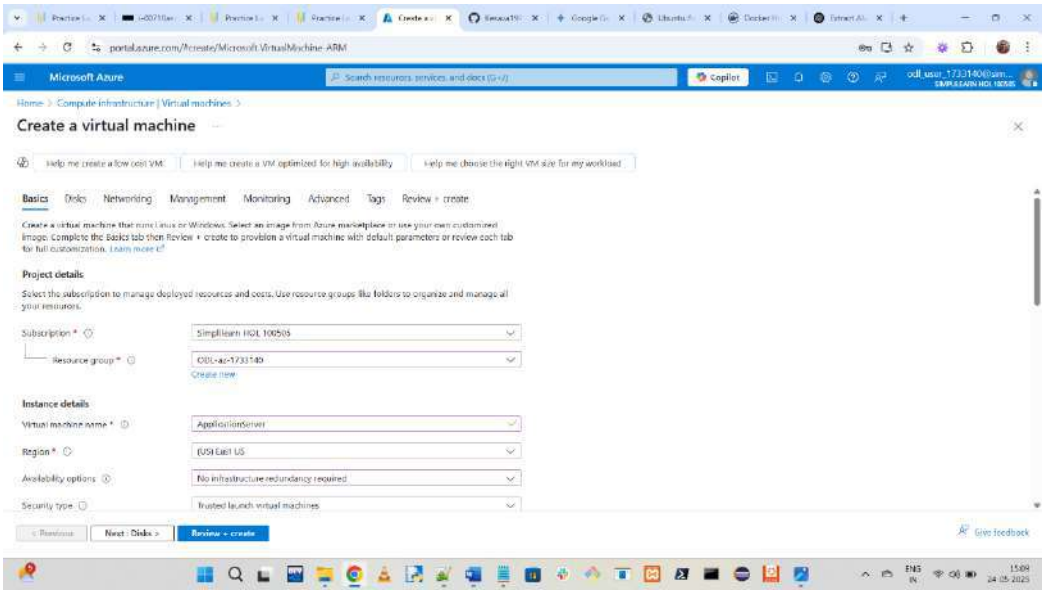

Launch an instance | EC2 | us-east-1 — Mozilla Firefox
Inbound Security Group Rules
Security group rule 1 (TCP, 22, 0.0.0.0/0)
Type
ssh
Protocol
Port range
Source type
Anywhere
Source
Description - optional
e.g. SSH for admin desktop
Remove
Security group rule 2 (TCP, 8989, 0.0.0.0/0)
Custom TCP
8989
Custom
Add CIDR, prefix list or security group
0.0.0.0/0
Summary
Number of instances
1
Software Image (AMI)
ubuntu-eks/k8s_1.26/images/hvm...read more
Virtual server type (instance type)
t3.micro
Firewall (security group)
New security group
Storage (volumes)
1 volume(s) - 20 GiB
Free tier: In your first year of opening an AWS account, you get 750 hours per month of t2.micro



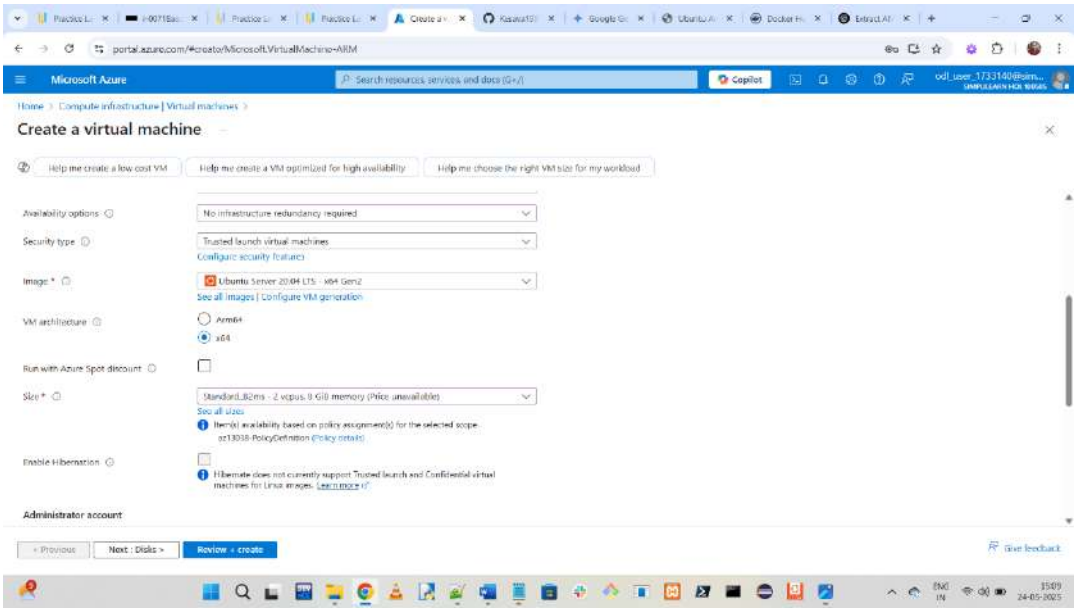
4.7 Navigate to Azure Console and launch a new VM which we will be connecting using Ansible Controller



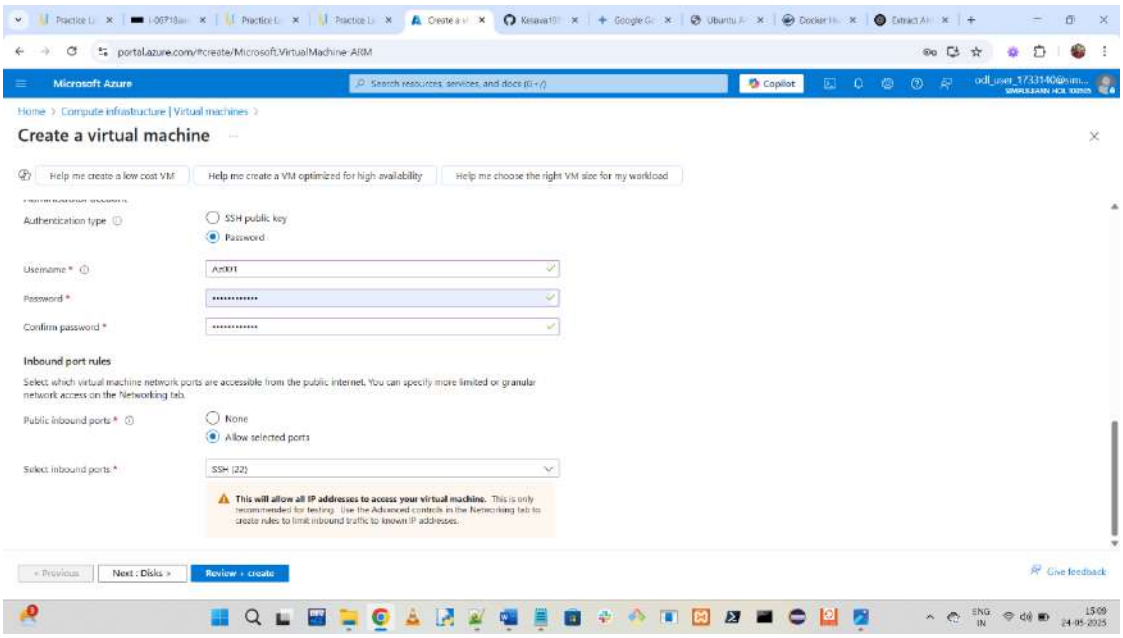
4.8 Navigate to Virtual Machine service in Azure and proceed with launching new ubuntu 24.04 VM. Click on Create virtual machine and provide details as per below screenshot



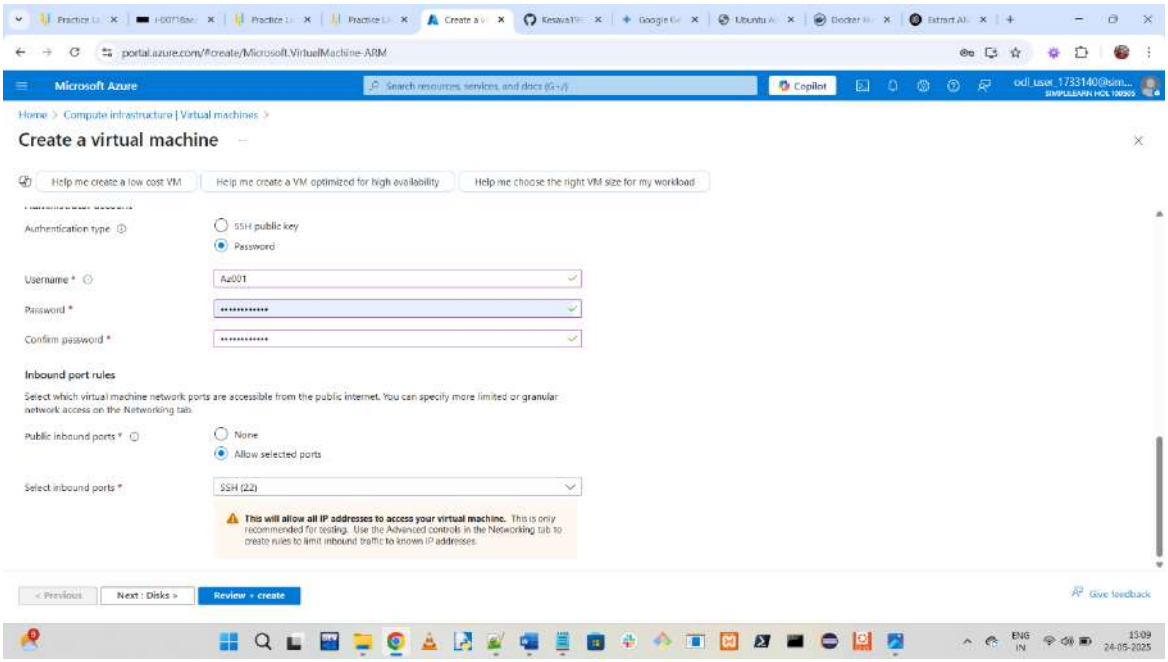
4.9 Select Ubuntu 24.04 image, VM size



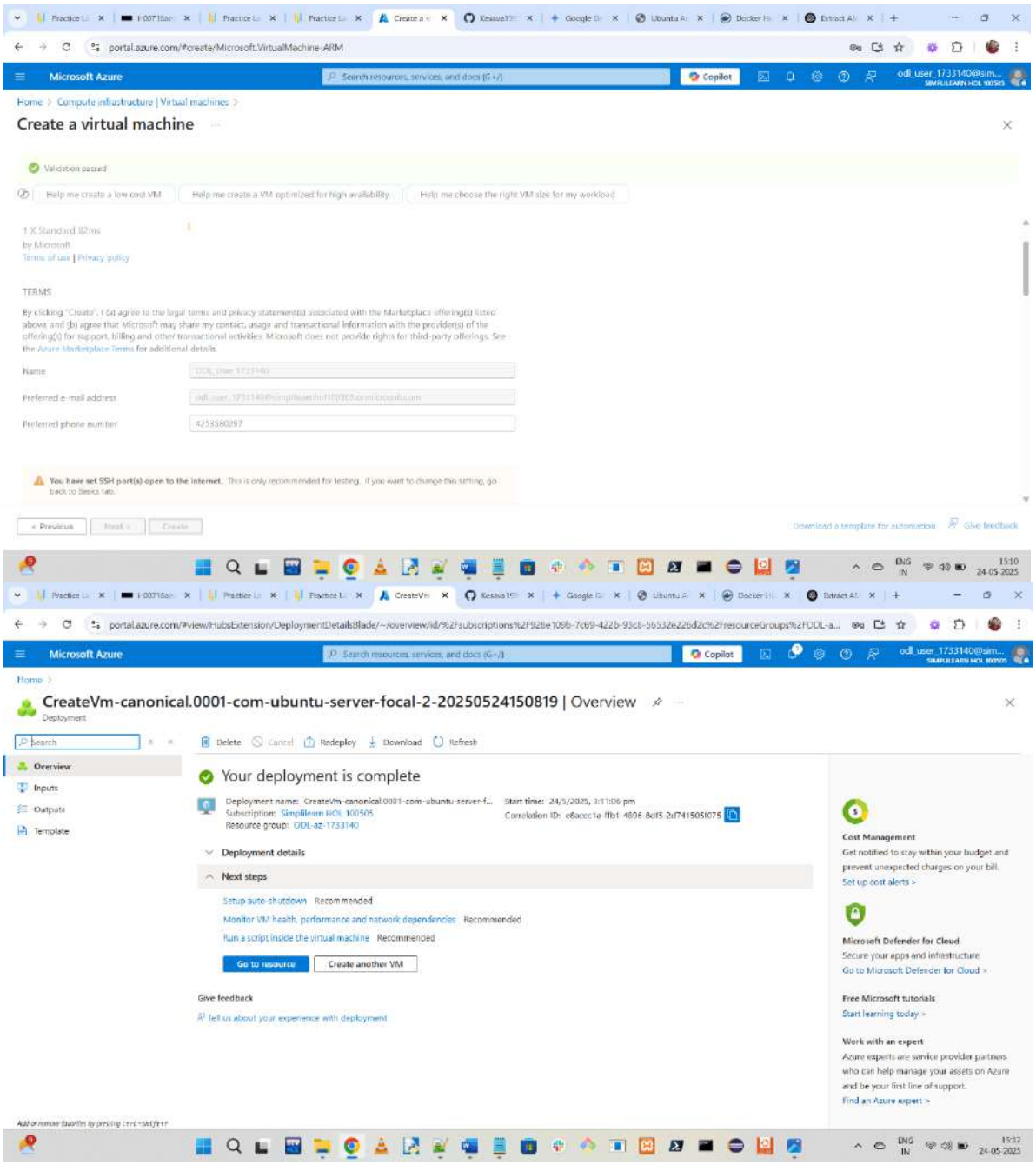
4.10 For authentication provide username and password which we will be using with Anisble to authenticate



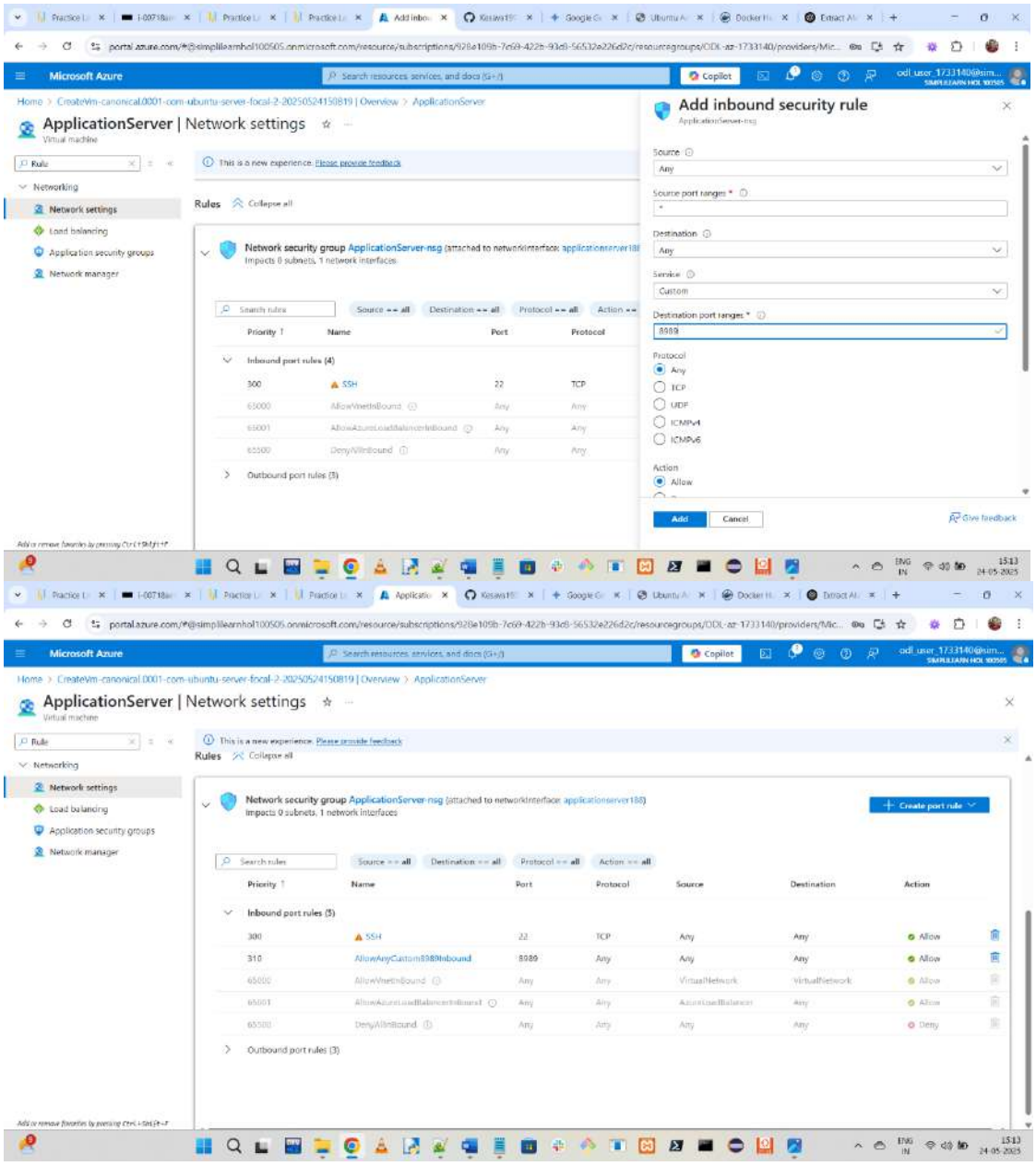
4.11 Click on Review and create to launch VM. Once VM is launched we must open **8989** ports from firewall rules to access application



4.12 Navigate to Virtual Machine service, select desired VM and modify networking configuration as per below screenshot



4.13 Create inbound port rule for 8989 as shown in the screenshot below:



Step 5: Validate Deployment

> **Note**: Once both AWS and Azure VM is launched, proceed with configuring Ansible inventory file to perform connectivity between Ansible and VMs

5.1 Create **inventory.yaml** in GitHub repository with below content

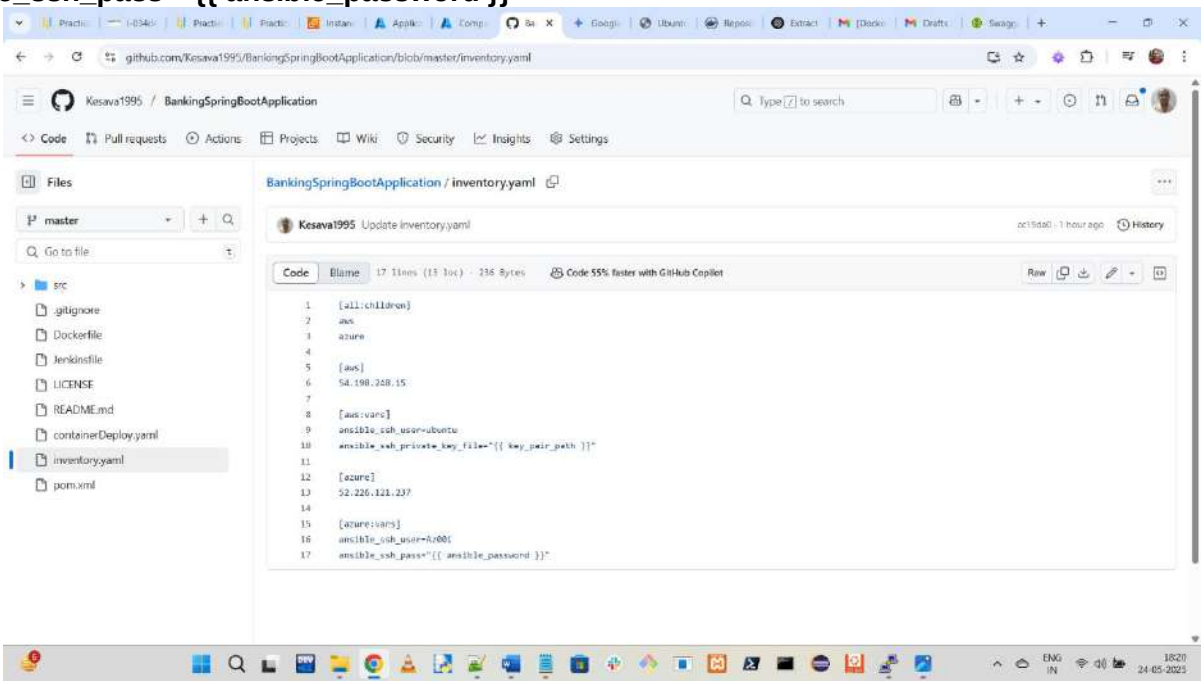
[all:children]
aws
azure

[aws]
54.198.248.15

[aws:vars]
ansible_ssh_user=ubuntu
ansible_ssh_private_key_file="{{ key_pair_path }}"

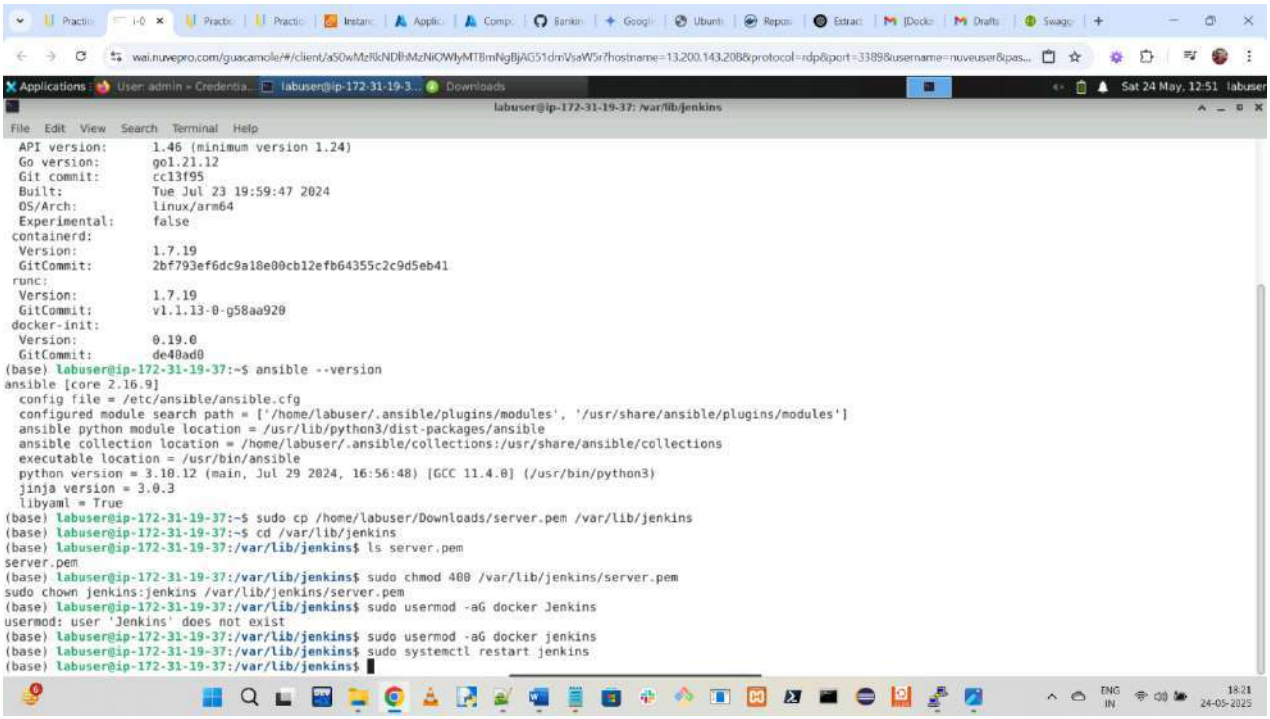
[azure]
52.226.121.237

[azure:vars]
ansible_ssh_user=Az001
ansible_ssh_pass="{{ ansible_password }}"

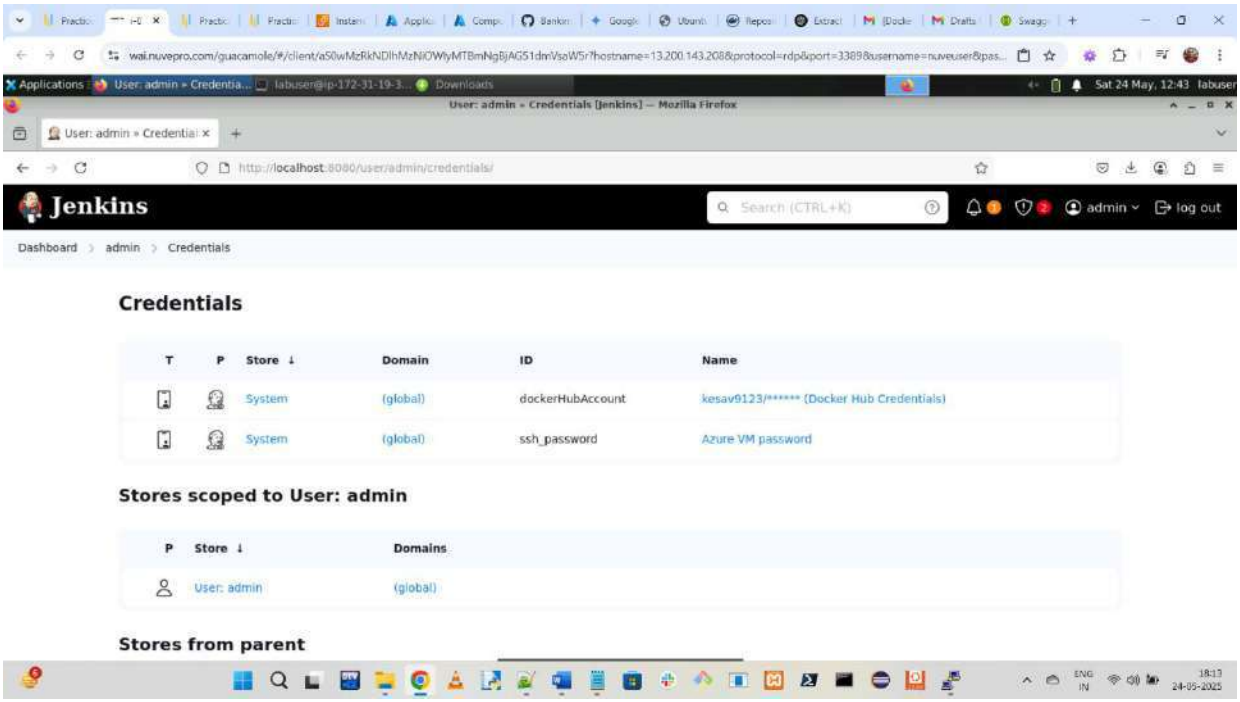


We need to create key pair on DevOps lab which was downloaded while launching EC2 instance

5.2 Navigate to **/var/lib/jenkins**, open the private key file **server.pem** using **vi**, verify its permissions with **ls -lart**, update the permissions to read-only using **chmod 400 server.pem**, and confirm the change with **ls -alrt**, ensuring the file is secure with read-only access for the Jenkins user



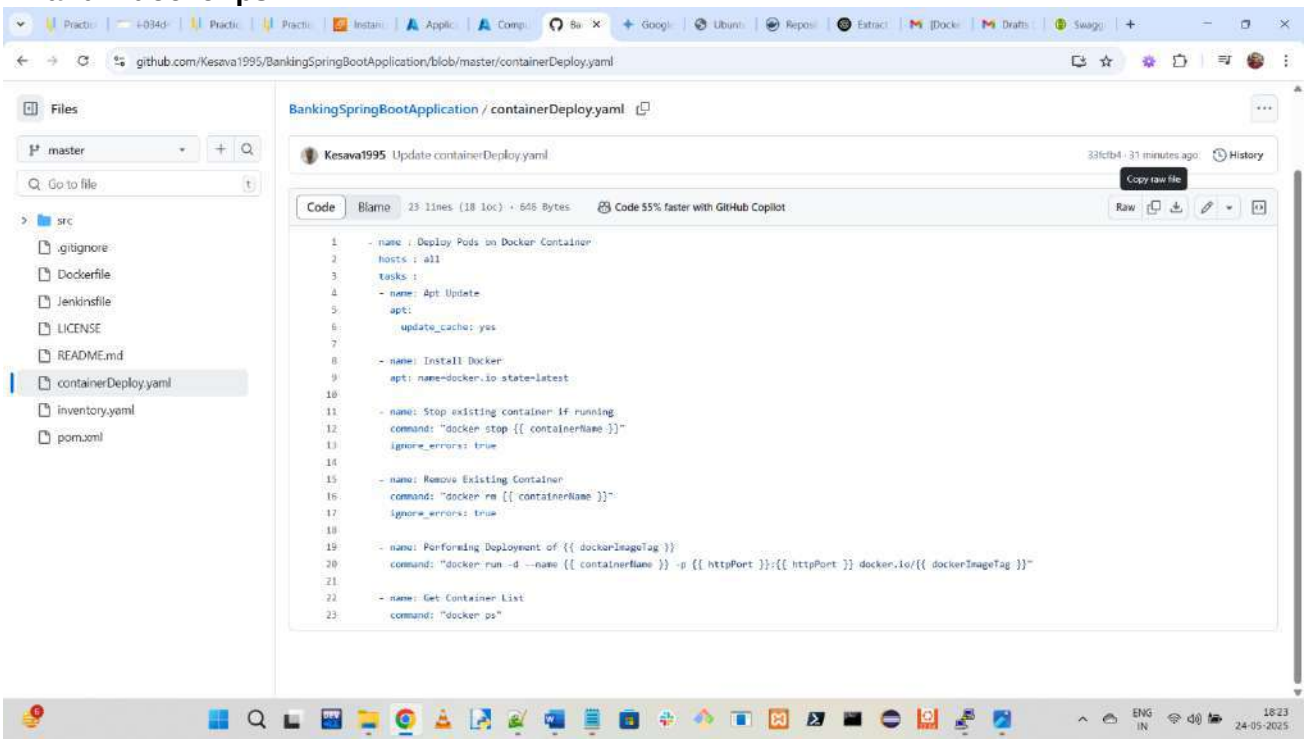
We need to create a Jenkins credential with Azure password for VM connectivity



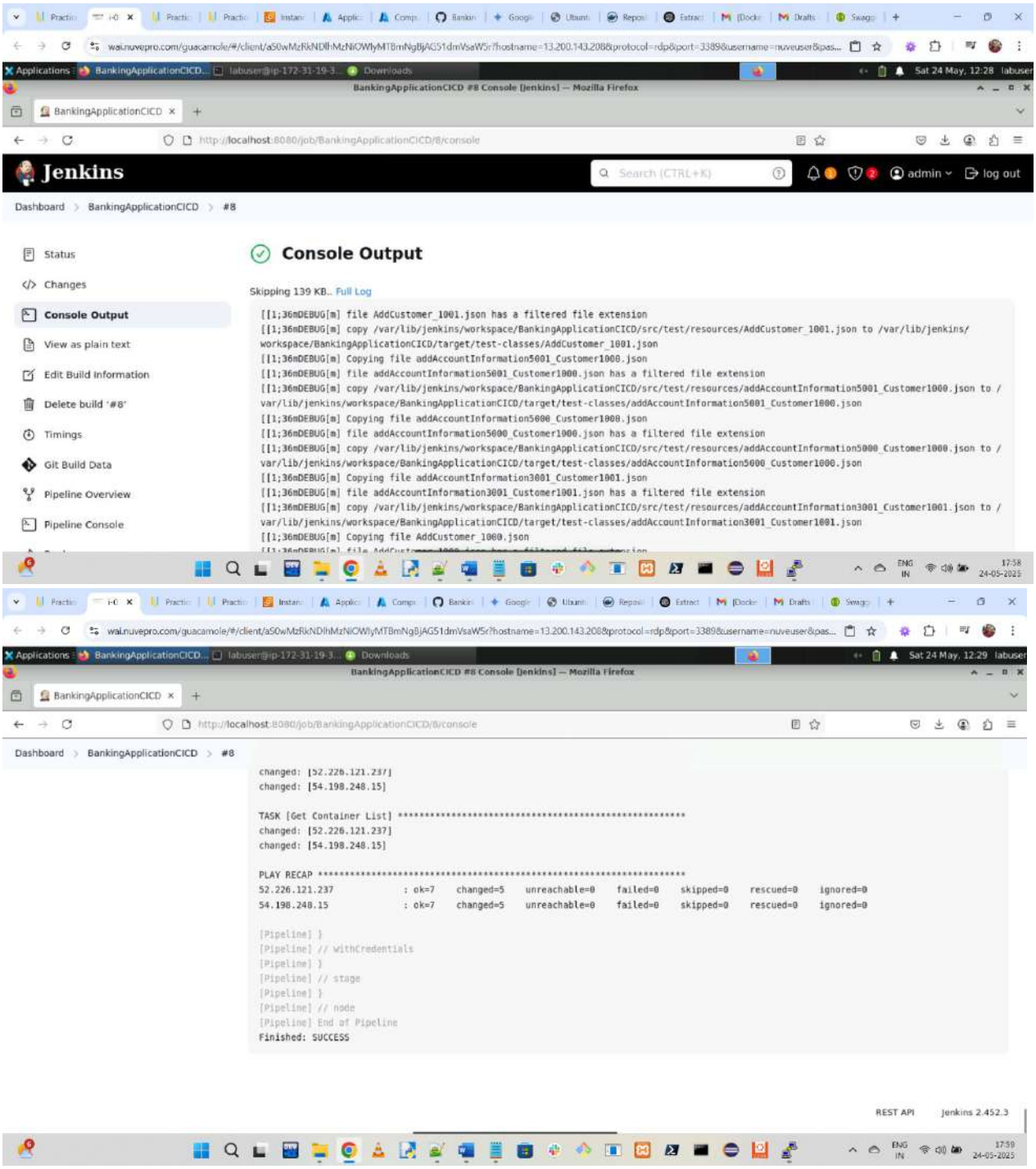
5.3 Now, we need to create Ansible Playbook file as per below content

- name : Deploy Pods on Docker Container
hosts : all

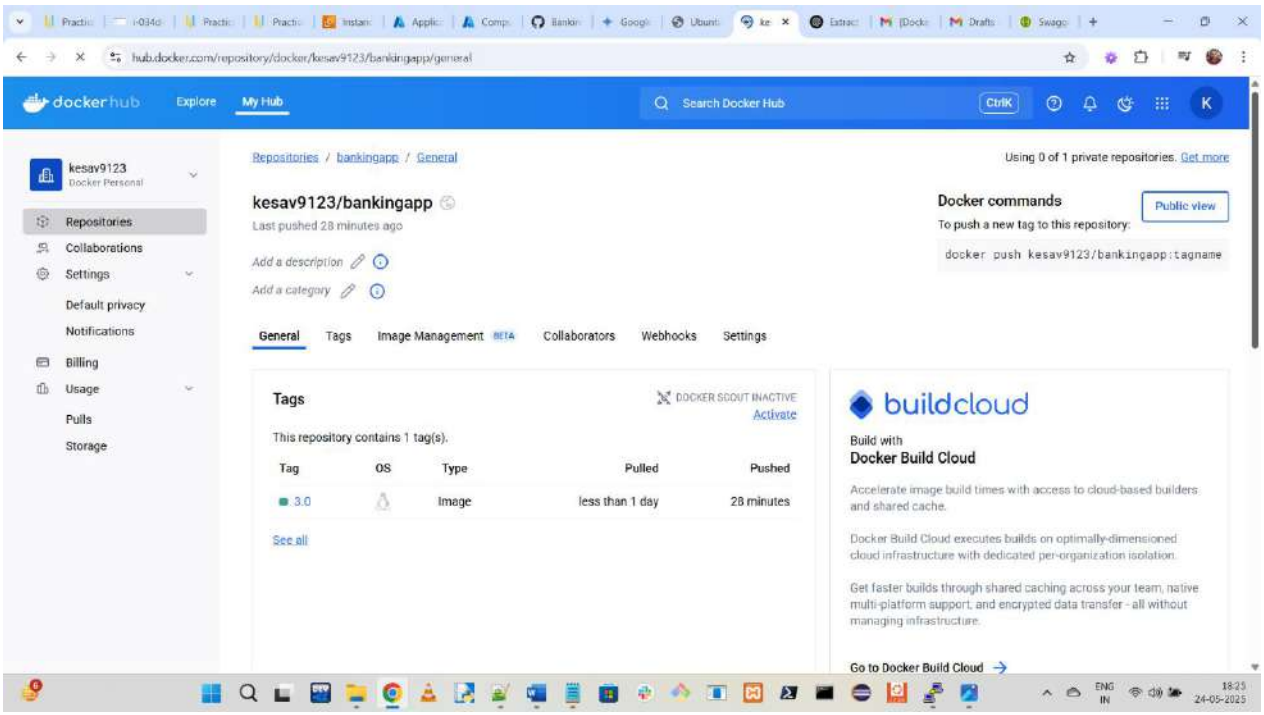
```
tasks :
- name: Apt Update
  apt:
    update_cache: yes

- name: Install Docker
  apt: name=docker.io state=latest

- name: Stop existing container if running
  command: "docker stop {{ containerName }}"
  ignore_errors: true

- name: Remove Existing Container
  command: "docker rm {{ containerName }}"
  ignore_errors: true

- name: Performing Deployment of {{ dockerImageTag }}
  command: "docker run -d --name {{ containerName }} -p {{ httpPort }}:{{ httpPort }} docker.io/{{ dockerImageTag }}"

- name: Get Container List
  command: "docker ps"
```



5.4 Navigate to Jenkins job created and click on Build now to start running build for Jenkins job created

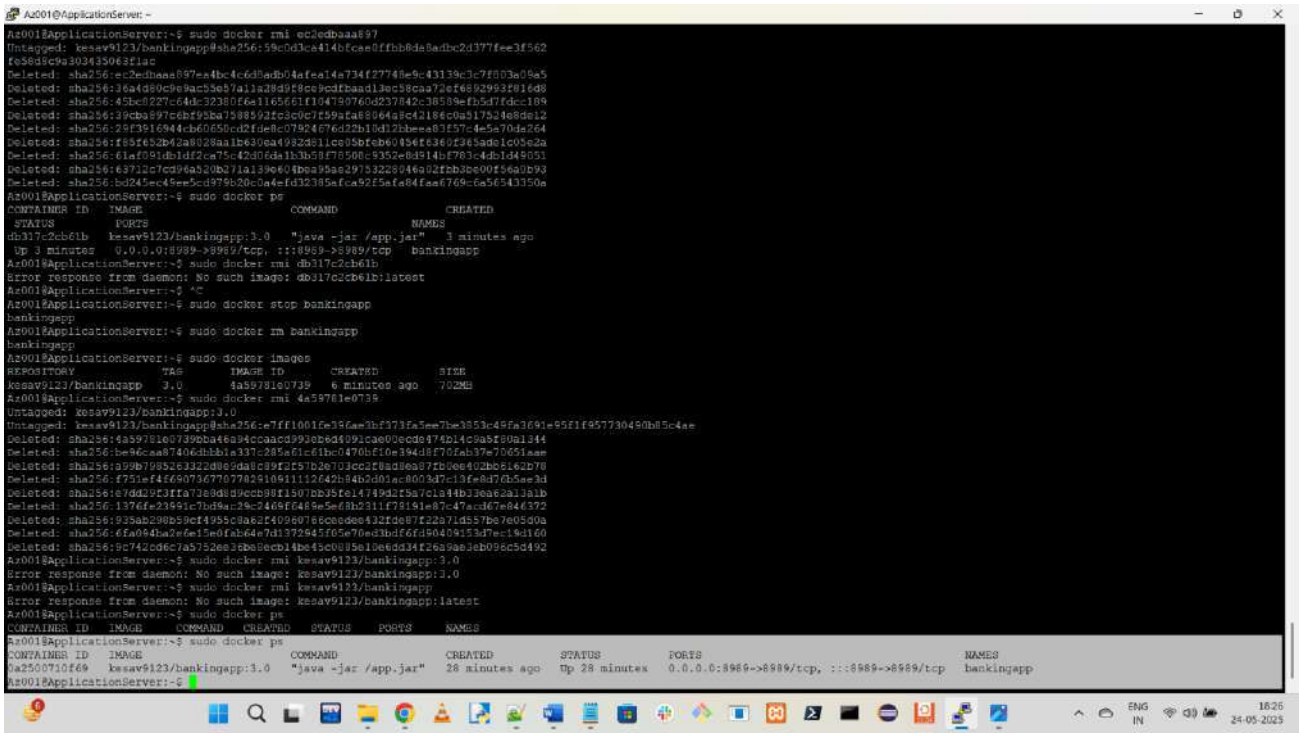


5.5 Navigate to Docker hub to validate if Docker image gets published or not



Note: First time build may take more time than expected since lot of maven dependencies will be downloaded and cache locally first time. Subsequent executions will not take much time later.

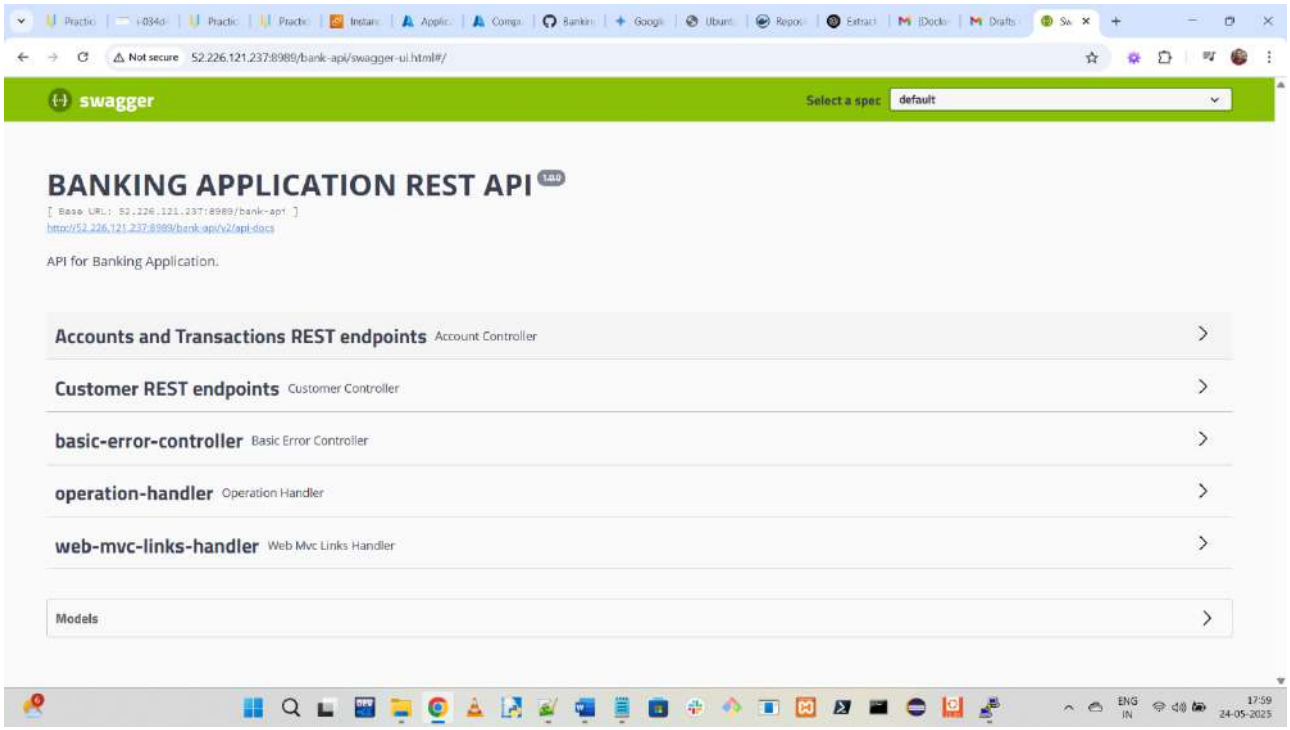
5.6 Once Application container is deployed on Docker host you can manually validate if containers are really running or not, connect to AWS VM and running below command to validate



5.7 Navigate to Azure VM and validate in case Docker container is deployed there or not, use system default browser to access application deployed on Docker Host. We will be getting Swagger Interface using which you can validate your Rest APIs.

http://<SERVER-IP>:8989/bank-api/swagger-ui.html

http://52.226.121.237:8989/bank-api/swagger-ui.html



By following the above steps, you have successfully implemented a multi-cloud deployment pipeline using Jenkins, Ansible, and Docker, enabling seamless integration and automation for the Java application across AWS and Azure environments. This setup ensures efficient CI/CD processes, robust containerized deployments, and enhanced application availability, validated through Swagger API testing.