Assignment 3: Distributed Infrastructure, Model Serving and CI/CD

Please carefully read the instructions before you begin this assignment.

Course

Data Engineering-II

Introduction

The lab assignment covers the practical part of the discussed concepts of dynamic contextualization and model serving in a scalable production environment and reliable continuous integration and development process.

The lab consists of **four** tasks and one optional task. First three tasks are compulsory, the fourth task is a bonus task and the optional task does not have any points.

Points

Total grade for this assignment ranges from 1 to a maximum of 2 points. First three tasks have a total of 1 point, whereas the forth task has a maximum of 1 point.

Submission

For all tasks, you should attach screenshots to the report to show the completeness of the task, and answer the questions in a well-formatted manner.

Notation

In the lab instructions, commands that need to be executed in command line (shell) are wrapped in dark code block, while code texts are wrapped in light code block, shown as follows:

Example for commands:

```
sudo apt update; sudo apt upgrade
```

Example for code texts:

```
import time

print(time.time())
```

Discussions

Problems can be discussed on discussion forums for the lab on Studium. The forum encourages students to discuss issues related to the lab work.

Note: Assignments can be discussed with your fellow students but copying solutions of your fellow student is ***NOT*** allowed which can result in disciplinary action.

Practical notes:

1. Please terminate VMs once you finish the task.
2. We recommend you to create a VM in SSC and execute all the tasks on that virtual machine. You can run the tasks on your laptops but it may break your local working environment.
3. For all the tasks, clone the repository: https://github.com/sztoor/model_serving.git
4. The code for tasks is available in the `model_serving` directory, and the folder structure is shown as following:

```
- model_serving
  - Single_server_without_docker
  - Single_server_with_docker
  - CI_CD
  - OpenStack-Client
```

Section I: Dynamic contextualization

Dynamic contextualization is a process of preparing a customized computing environment at runtime. The process ranges from creating/defining user roles and permissions to updating/installing different packages and initiating services.

Task-1: Single server deployment

In this task, we will learn how dynamic contextualization works using `CloudInit`. For this task, we first need to manually create a VM as our client VM. Then on the client VM, we will use OpenStack Python APIs to start a new VM, and contextualize it automatically without logging in to the machine and contextualize it manually.

The contextualization process will setup the following working environment:

1. Flask based web application as a frontend server
2. Celery and RabbitMQ server for backend server
3. Model execution environment based on Keras and TensorFlow

In case you are not familiar with the above-mentioned packages please read the following links:

1. Flask Application -> https://flask.palletsprojects.com/en/1.1.x/
2. Celery and RabbitMQ -> https://docs.celeryproject.org/en/stable/getting-started/
3. Keras and TensorFlow -> https://www.tensorflow.org/guide/keras

The process will start when a client sends a new prediction request from the front-end web server. The server will pass the request to the backend Celery environment where running workers in the setup will pick up the task, run the predictions by loading the available model, send back the results and finally the frontend server will display the results.

A. Understand the Code Base

1. Start a VM from the SSC dashboard and login.

2. Clone the git repository.

```
git clone https://github.com/sztoor/model_serving.git
```

3. Goto `model_serving/single_server_without_docker/production_server/` directory. The directory contains the code that will run on the production server VM. Following is a categorization of the files. Please open files and understand the content of the files.

```
- Flask App based frontend
  - app.py
  - static
  - templates
```

```
- Celery and RabbitMQ setup
  - run_task.py
  - workerA.py
- Machine learning model and data
  - model.h5
  - model.json
  - pima-indians-diabetes.csv
```

4. Go to `model_serving/openstack-client/single_node_without_docker_client/`. This is the code that we will use to contextualize our production server. The code is based on the two files shown as follows. Open the files and understand the steps.

```
- cloud-cfg.txt # CloudInit configuration file
- start_instance.py # OpenStack Python code
```

B. Prepare your RC file

The RC file from the SSC is used for identification of the user. We need to download it from the OpenStack site and set a password for it.

1. First log in the SSC OpenStack site and make sure you are in the correct project on the top left. Then download the Runtime Configuration (RC) file from the SSC site from the top right frame: <your-user-name>->OpenStack RC File.

Confirm that your RC file have following environment variables:

```
export OS_AUTH_URL=https://east-1.cloud.snic.se:5000
export OS_PROJECT_ID=<your-project-id>
export OS_PROJECT_NAME=<your-project-name>
export OS_USER_DOMAIN_NAME="snic"
export OS_USERNAME=<your-user-name>
```

2. Set API access password. Go to Swedish Science Cloud, Left frame, under Services "Set your API password".

3. Set the environment variables by sourcing the RC-file in the client VM.

```
source <your-project-name>_openrc.sh
```

NOTE: You need to enter the API access password.

C. Install OpenStack API

To create new VMs on SSC OpenStack, we need to have the OpenStack API installed on the VM we are working on, which will be used to create new VMs. Openstack APIs only need to be installed on the client VM. Installation on other VMs is unnecessary unless they will be used to create additional new VMs.

1. Install the necessary OpenStack APIs

```
sudo apt install python3-openstackclient
sudo apt install python3-novaclient
sudo apt install python3-keystoneclient
```

2. Run the following commands to confirm that you have the correct packages available on your client VM and your OpenRC file is properly loaded:

```
openstack server list
openstack image list
```

Note 1: If you set the API password in Step B.2 in just a few minutes, it may take up to 15 minutes to have it fully effective for OpenStack. If you see the error message `The request you have made requires authentication. (HTTP 401)`, you may need to wait a bit.

Note 2: In Ubuntu 22.04, OpenStack Yoga is the default API version. If you have another OpenStack installed, such that you need another OpenStack API version, you may want to refer to OpenStack packages for Ubuntu — Installation Guide documentation.

D. Run the code

Once you have the RC file loaded and API installed, we can run the following command.

```
python3 start_instance.py
```

Note: You need to make changes in `start_instance.py` by setting up variable values.

The command will start a new VM and initiate the contextualization process. It will take approximately 10 to 15 minutes. The progress can be seen on the cloud dashboard.

Once the process finishes, attach a floating IP to your production server and access the webpage from your client machine.

- Welcome page http://<PRODUCTION-SERVER-IP>:5100

- Predictions page http://<PRODUCTION-SERVER-IP>:5100/predictions

Questions

1. Explain how the application works? Write a short paragraph about the framework.

2. What are the drawbacks of the contextualization strategy adopted in Task 1? Write at least four drawbacks.

3. Currently, the contextualization process takes 10 to 15 minutes. How can we reduce the deployment time?

4. Write half a page summary about the task and add screenshots if needed.

TERMINATE THE PRODUCTION SERVER VM STARTED FOR TASK 1!

Task-2: Single server deployment with Docker Containers

In this task, we will repeat the same deployment process but with Docker containers. This time we will create a flexible containerized deployment environment where each container has a defined role.

A. Deploy the Application

1. Go to `model_serving/single_server_with_docker/production_server/` directory on the client VM. The directory contains the code that will run on your production server. Following is the structure of the code. Open files and understand the application's structure.

```
- Flask Application based frontend
  - app.py
  - static
  - templates
- Celery and RabbitMQ setup
  - run_task.py
  - workerA.py
- Machine learning Model and Data
  - model.h5
  - model.json
  - pima-indians-diabetes.csv
- Docker files
  - Dockerfile
  - docker-compose.yml
```

2. Goto the `model_serving/openstack-client/single_node_with_docker_client/` directory. The directory contains the code that we will use to contextualize the production server. The code is based on the following two files. Open the files and understand the steps.

```
- CloudInit configuration file
  - cloud-cfg.txt
- OpenStack python code
  - start_instance.py
```

3. Run the following command.

```
python3 start_instance.py
```

Note: You need to make changes in `start_instance.py` by setting up variable values.

The command will start a new VM and initiate the contextualization process. It will take approximately 10 to 15 minutes. The progress can be seen on the cloud dashboard.

Once the process finishes, attach a floating IP to your production server and access the webpage from your client machine.

- Welcome page http://<production-server-ip>:5100
- Predictions page http://<production-server-ip>:5100/predictions

B. Scale up the application

The next step is to test the horizontal scalability of the setup.

1. Login to the production server from your local computer

2. Check the cluster status

```
sudo docker ps
```

The output will be as following:

```
Name    Command    State    Ports
------------------------------------------------------------------------------
production_server_rabbit_1     docker-entrypoint.sh rabbi ...   Up
15671/tcp, 0.0.0.0:15672->15672/tcp, 25672/tcp, 4369/tcp, 5671/tcp,
0.0.0.0:5672->5672/tcp

production_server_web_1        python ./app.py --host=0.0.0.0   Up
0.0.0.0:5100->5100/tcp

production_server_worker_1_1   celery -A workerA worker - ...   Up ...
```

Following three containers are running on the production server:

```
production_server_rabbit_1  -> RabbitMQ server
production_server_web_1 -> Flask based web application
production_server_worker_1_1 -> Celery worker
```

3. Now we will add multiple workers in the cluster using docker commands. Currently, there is one worker available. We will add two more workers. First go to direction `/model_serving/single_server_with_docker/production_server` and then run command

```
sudo docker compose up --scale worker_1=3 -d
```

Then check the status

```
sudo docker ps
```

The result should be as following

```
Name    Command    State    Ports
------------------------------------------------------------------------------
production_server_rabbit_1     docker-entrypoint.sh rabbi ...   Up
15671/tcp, 0.0.0.0:15672->15672/tcp, 25672/tcp, 4369/tcp, 5671/tcp,
0.0.0.0:5672->5672/tcp

production_server_web_1        python ./app.py --host=0.0.0.0   Up
0.0.0.0:5100->5100/tcp

production_server_worker_1_1   celery -A workerA worker - ...   Up

production_server_worker_1_2   celery -A workerA worker - ...   Up

production_server_worker_1_3   celery -A workerA worker - ...   Up
```

Now we have 3 workers running in the system.

4. Scale down the cluster.

```
sudo docker compose up --scale worker_1=1 -d
```

Questions

1. What problems Task 2 deployment strategy solves compared to the strategy adopted in Task 1?

2. What are the outstanding issues that the deployment strategy of Task 2 cannot not address?

3. What are the possible solutions to address those outstanding issues?

4. What is the difference between horizontal and vertical scalability? Is the strategy adopted in Task 2 following horizontal or vertical scalability?

5. Write half a page summary about the task and add screenshots if needed.

TERMINATE THE SERVER VM STARTED FOR TASK 2!

Section II: Continuous Integration and Continuous Delivery (CI/CD)

The next two tasks will cover scalable cluster deployments using Ansible playbooks and CI/CD environment using versioning system Git and an extension called Git Hooks.

Task 3: Deployment of multiple servers using Ansible

For this task we need a setup based on following three VMs:

- Client VM: This machine will serve as an Ansible host and initiate the configuration process.
- Production Server: The machine will host the dockerized version of the application discussed in Task 1.
- Development Server: The machine will host the development environment and push the new changes to the production server.

If you want to know more about Ansible, visit following URLs:

- Ansible Collaborative
- https://www.ansible.com/blog/it-automation
- How Ansible Works
- How to Install and Configure Ansible on Ubuntu 18.04 | DigitalOcean

A. Understand the code structure

1. Login to the Client machine from your local machine

2. Go to `model_serving/ci_cd`. This directory contains the following two sub-directories:

```
/production_server
 - Flask Application based frontend
    -- app.py
    -- static
    -- templates
 - Celery and RabbitMQ setup
    -- run_task.py
    -- workerA.py
 - Machine learning Model and Data
    -- model.h5
    -- model.json
```

```
    -- pima-indians-diabetes.csv
/development_server
 - Machine learning Model and Data
    -- model.h5
    -- model.json
    -- pima-indians-diabetes.csv
    -- neural_net.py
```

Open files and understand the application's structure.

3. Go to `model_serving/openstack-client/single_node_with_docker_ansible_client`. The files in the directory will be used to contextualize the production and deployment servers:

```
-  CloudInit files
  -- prod-cloud-cfg.txt
  -- dev-cloud-cfg.txt
- OpenStack code
  -- start_instance.py
- Ansible files
  -- setup_var.yml
  -- configuration.yml
```

The client code will start two VMs and by using Ansible orchestration environment. It will contextualize both of the VMs simultaneously.

B. Run the code

1. Install and configure Ansible on the client machine

2. Important: For this step, you need to be user `ubuntu` but not `root`. To make sure about it, run `whoami` in the shell and the result should be printed as `ubuntu`.

   Then Generate a SSH key pair.

```
mkdir -p /home/ubuntu/cluster-keys; ssh-keygen -t rsa
```

Set the file path `/home/ubuntu/cluster-keys/cluster-key` Do not set the password, simply press Enter twice.

Expected output should be similar to this:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ubuntu/.ssh/id_rsa):
```

```
/home/ubuntu/cluster-keys/cluster-key
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in
/home/ubuntu/cluster-keys/cluster-key.
Your public key has been saved in
/home/ubuntu/cluster-keys/cluster-key.pub.
The key fingerprint is:
4*:d*:0a:**:3*:4e:3f:*d:27:38:8*:74:*4:4d:9*:** demo@a
The key's randomart image is:
+--[ RSA 2048]----+
|    *     .**.   |
|          .  o*E  |
|        + .  *    |
|     . =*= *      |
|      =** = .     |
|     o + = +      |
|      . o * o .   |
|    *     ** *    |
|                  |
+------------------+
```

The step will generate cluster ssh keys at the following location:

- Private key: `/home/ubuntu/cluster-keys/cluster-key`
- Public key: `/home/ubuntu/cluster-keys/cluster-key.pub`

3. Next, we will start the Production and Development servers.

a. Go to `model_serving/openstack-client/single_node_with_docker_ansible_client`, open `prod-cloud-cfg.txt` delete the old key from the section `ssh_authorized_keys:` and copy the complete contents of `/home/ubuntu/cluster-keys/cluster-key.pub` in the `prod-cloud-cfg.txt` file.

b. Repeat step 7.a for the `dev-cloud-cfg.txt.` Delete the old key from the section `ssh_authorized_keys:` and copy the complete contents of `/home/ubuntu/cluster-keys/cluster-key.pub` in the `dev-cloud-cfg.txt` file.

4. Run the start_instance.py code.

```
python3 start_instance.py
```

The output will give you the internal IPs of the VMs.

```
user authorization completed.
Creating instances ...
waiting for 10 seconds..
Instance: prod_server_with_docker_6225 is in BUILD state, sleeping for 5
seconds more...
Instance: dev_server_6225 is in BUILD state, sleeping for 5 seconds more...
Instance: prod_server_with_docker_6225 is in ACTIVE state, sleeping for 5
seconds more...
Instance: dev_server_6225 is in BUILD state, sleeping for 5 seconds more...
Instance: __prod_server_with_docker_6225__ is in ACTIVE state ip address:
__192.168.1.19__
Instance: __dev_server_6225__ is in ACTIVE state ip address:
__192.168.1.17__
```

Now we have two VM running with the internal IP addresses.

5. Install Ansible packages on the client machine.

```
sudo apt update; sudo apt upgrade
sudo apt-add-repository ppa:ansible/ansible
sudo apt update
sudo apt install ansible
```

6. Next step is to enter these IP addresses in the Ansible hosts file. For this step you need to open the Ansible inventory file first.

```
# switch to root user.
sudo bash
nano /etc/ansible/hosts
```

Then you need to add the following content in that file. Remember to fill in the IP address of your own production and develop VMs (private IP addresses, starting with `192.168.*.*`):

```
[servers]
prodserver ansible_host=<production server IP address>
devserver ansible_host=<development server IP address>

[all:vars]
ansible_python_interpreter=/usr/bin/python3

[prodserver]
prodserver ansible_connection=ssh ansible_user=appuser

[devserver]
```

```
devserver ansible_connection=ssh ansible_user=appuser
```

7. Just to confirm the access permissions are correctly set, access the production and development server from the client VM.

First switch back to user `ubuntu` from the root user

```
exit
```

Then try to login to the production and development VM from the client machine.

```
ssh -i /home/ubuntu/cluster-keys/cluster-key appuser@<PRODUCTION-SERVER-IP>
```

```
ssh -i /home/ubuntu/cluster-keys/cluster-key
appuser@<DEVELOPMENT-SERVER-IP>
```

8. Now we will run the Ansible script available in the `model_serving/openstack-client/single_node_with_docker_ansible_client` directory.

```
export ANSIBLE_HOST_KEY_CHECKING=False
ansible-playbook configuration.yml
--private-key=/home/ubuntu/cluster-keys/cluster-key
```

Though some tasks are slow (e.g. `TASK [apt update]`), do not stop ansible unless error messages are thrown. The process will take 10 to 15 minutes to complete.

Attach a floating IP address to the production server and access the same predictions webpage as we did in previous tasks.

Questions

1. What is the difference between CloudInit and Ansible?
2. Explain the configurations available in **dev-cloud-cfg.txt** and **prod-cloud-cfg.txt** files.
3. What problem have we solved by using Ansible?

Important: Task-4 is the continuation of Task-3. Do NOT terminate any of your VMs in Task-3.

Task-4: CI/CD using Git HOOKS

In this task, we will build a reliable execution pipeline using Git Hooks. The pipeline will allow continuous integration and delivery of new machine learning models in a production environment.

A. Enable SSH key based communication between production and development servers

1. First login to the development server from the client machine

```
ssh -i cluster-key appuser@<DEVELOPMENT-SERVER-IP>
```

2. Generate SSH key in the development server

```
ssh-keygen
```

Important:
- Do not change the default `/home/appuser/.ssh/id_rsa` path of the file.
- Do not set the password, simply press Enter twice.

This step will create two files, private key `/home/appuser/.ssh/id_rsa` and public key `/home/appuser/.ssh/id_rsa.pub`.

3. Copy the contents of the public key file `/home/appuser/.ssh/id_rsa.pub` from the development server.

4. Login to production server

```
ssh -i cluster-key appuser@<PRODUCTION-SERVER-IP>
```

5. Open file `/home/appuser/.ssh/authorized_keys` and append the public key in the development VM in the authorized_keys file.

B. Create a git hook

1. Create a jump directory in the production VM. Double check that user appuser is the owner of the directory directory. NOTE: Create the directory as user `appuser` instead of the `root` user. You can confirm it with command `whoami`.

```
mkdir /home/appuser/my_project
```

2. Then initialize an empty Git re.pository in `/home/appuser/my_project/`.

```
cd /home/appuser/my_project
git init --bare
```

3. Create a git hook post-receive by creating a file `hooks/post-receive` with content as follows:

```
#!/bin/bash
while read oldrev newrev ref
do
    if [[ $ref =~ .*/master$ ]];
    then
        echo "Master ref received.  Deploying master branch to production..."
        sudo git --work-tree=/model_serving/ci_cd/production_server
--git-dir=/home/appuser/my_project checkout -f
    else
        echo "Ref $ref successfully received.  Doing nothing: only the master branch may be
deployed on this server."
    fi
done
```

4. Change permissions of the file to make it executable

```
chmod +x hooks/post-receive
```

5. Exit Production Server

6. Login to the Development Server

```
ssh -i cluster-key appuser@<DEVELOPMENT-SERVER-IP>
```

7. Create the same jumpy directory and initialize an empty git repository

```
mkdir /home/appuser/my_project
cd /home/appuser/my_project
git init
```

8. Copy files `model.h5` and `model.json` to `/home/appuser/my_project/` directory from `/model_serving/ci_cd/development_server/` directory.

9. Add model files for git and commit the changes

```
cd /home/appuser/my_project/
git add .
git commit -m "new model"
```

10. Add production repository as a remote repository

```
git remote add production
appuser@<PRODUCTIONS-SERVER-IP>:/home/appuser/my_project
```

11. Push your commits to the production server

```
git push production master
```

The expected output should be similar to

```
...
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 2.36 KiB | 2.36 MiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To 192.168.1.21:/home/appuser/my_project
* [new branch]      master -> master
```

C. Model update

1. Make sure you are in the development VM. Go to `/model_serving/ci_cd/development_server/` directory. The directory contains a script named `neural_net.py`. The script will train a model and generate new model files `model.h5` and `model.json`. Open the training script `neural_net.py`, make changes in the model, and run the script. You may need sudo to edit and run the script.

2. Copy model files `model.h5` and `model.json` from the development directory `/model_serving/ci_cd/development_server/` to the jump repository `/home/appuser/my_project` on the development server.

```
cp /model_serving/ci_cd/development_server/model*  /home/appuser/project/.
```

3. Add the new models to the git repository, commit the change, and push the change.

```
git add .
git commit -m "new model"
git push production master
```

Expected output should be

```
Writing objects: 100% (4/4), 838 bytes | 838.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Master ref received.  Deploying master branch to production...
To 192.168.2.70:/home/appuser/my_project
   c20d19e..666ca0c  master -> master
```

Notice line `remote: Master ref received.  Deploying master branch to production...` – indicating that the post-receive hook of the production repository is working.

4. Access the URL http://<PRODUCTION-SERVER-IP>:5100/predictions and you will see the changes in predictions.

5. Improve the model by changing parameters or network architecture in the `neural_net.py` file and repeat the step-3 to push the new model to the production pipeline.

In case you want to learn more about Git Hooks, visit the following link: How To Use Git Hooks To Automate Development and Deployment Tasks | DigitalOcean

Questions

1. What are git hooks? Explain the post-receive script available in this task.

2. We have created an empty git repository on the production server. Why we need that directory?

3. Write the names of four different git hooks and explain their functionalities. (*Hints: Read the link available in the task or access sample scripts available in the hooks directory)*

4. How deployment of multiple servers (in task 3) can be achieved with Kubernetes? Draw a diagram of the deployment highlighting the nodes, services, configuration map and secret.

5. Write half a page summary about the task and add screenshots if needed.

Optional Task

Write Kubernetes/vault compatible configurations (yaml files) to deploy multiple servers discussed in task 3 (practical demonstration).