Task-1: Single server deployment

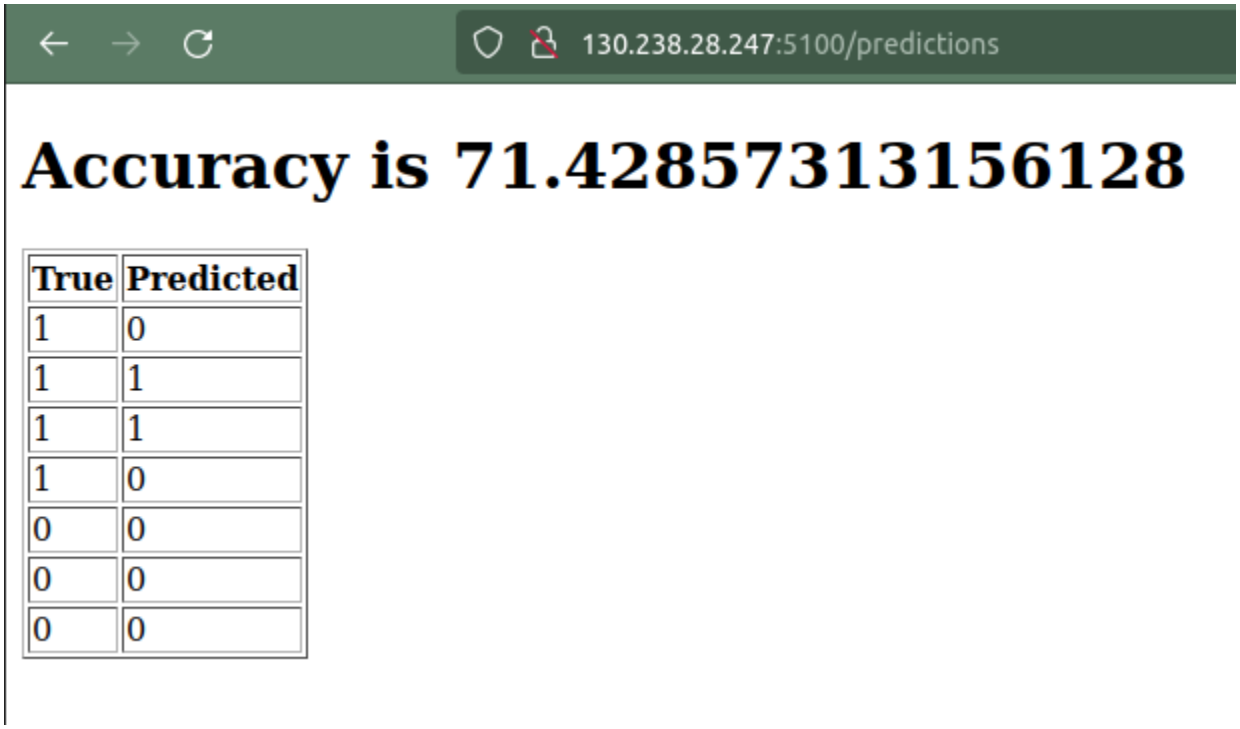
Questions

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

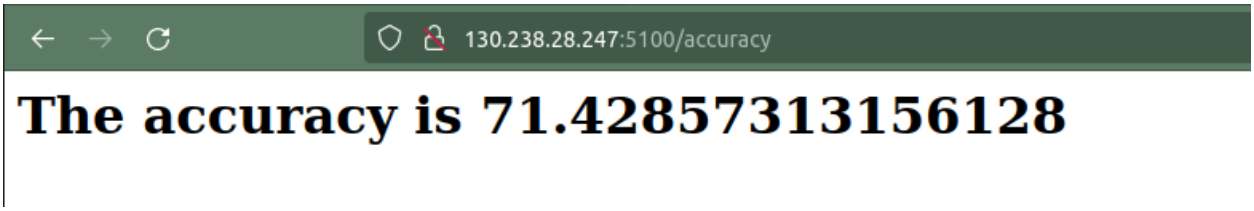
The architecture is designed into micro-service architecture style. The given application has 3 different services that need to be running.

- Frontend service: It is created using the flask python framework. The controller is actively listening to all incoming http requests on port 5100. The controller is designed to work with “/”, “/predictions" and “/accuracy” endpoints. When http request is made to <prodcution-ip>:5100/,It will generate a welcome page.


- Simlary on “/predictions” ,it will populate with predictions for “pima-indians-diabetes.csv” .



- on “/accuracy” endpoints, it will generate accuracy of our ML-model.




- Backend service consists of Celery and RabbitMQ server. Since training machine learning algorithms is a time consuming process and we don’t want end users feeling stuck at the “/predictions” page or we want to hide latency. so celery is used here to delegate time consuming jobs to appropriate workers instead of making our flask server busy. Celery takes care of boilerplate code to receive tasks from message brokers and assigning them appropriately to workers, monitoring the status of tasks and workers. RabbitMQ Server is used to maintain order of arrival of jobs in FIFO fashion and to ensure that each worker only gets one task at a time and that each task is only being processed by one worker.
- Worker service is designed with Keras, TensorFlow which loads data from “pima-indians-diabetes.csv”. Worker service starts to create model from model.json and adjust weight into new model according to “model.h5”. Finally, worker service loads a model which starts to make predictions using a new model.

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.

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

- No isolation of dependencies
- Longer time to contextualize environment
- Need to lookup log files of cloud-init for troubleshooting
- Not idempotent
- Can be executed only on boot time.
- Updated cloud-init script cannot be used after boot operations
- contextualization strategy adopted in task-1 is not configuration management tools

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

Dynamic contextualization applied to newly created VM instances  usually takes longer time to complete. So this can be improved if we create  server image with pre-installed packages used for dev/prod environments and only use configuration management tools like Ansible/Terraform to create/define user roles ,copying configuration files, injecting dev/prod specific environment variables  initiating services.

It’s easier to build a server image online, but it can be slow to boot and configure a new instance, which usually takes upto 10 minutes.

Performance can be improved if we prefer to create an image offline building using a packer. mounting a disk volume and copying the necessary file to it and turning that into a bootable image can make the whole process faster. The advantage of using Packer is vendor-agnostic. Since Packer uses the platform specific API to create the server instance.

I also suggest to create a CI/CD pipeline that creates,tests,and deploys images of the server.

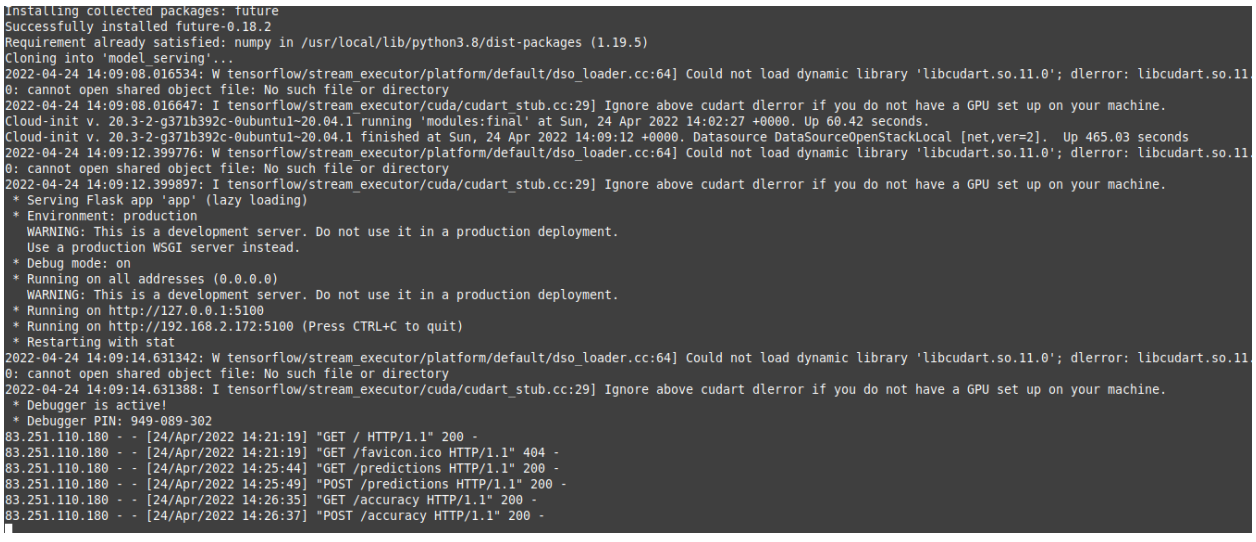
CI/CD pipelines immediately build new versions of an image whenever there is change in the git repository. With a mature pipeline integrated with pipelines for other parts of a system, you can safely roll out operating system patches and updates across your weekly or even daily.

Our infrastructure then becomes faster, disposable, replaceable, consistent, repeatable.

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

Start_instance.py is an example of IAC code. Its purpose is to create a new VM in snic cloud science based on parameters like “flavor”, “private_net” , “image_name”, “secgroups” and injects cloud-init script named “cloud-cfg.txt” into the newly created VM. start_instance.py script uses OpenStack API to create openstack resources. Start_instance.py depends upon some environment variables like “OS_USERNAME”, “OS_PASSWORD”, etc which is injected from the Runtime Configuration (RC) file of snic cloud.

Cloud-cfg.txt is a cloud-init script which gets executed only during boot time of the VM. cloud-init contains that script like installing packages, updating repository, installing dependency of project using pip, cloning git repository and running project in daemon mode.



Task-2: Single server deployment with Docker Containers

Questions

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

- Isolates dependency using docker containers which is helpful in avoiding conflict among frontend, backend, ML-service dependency.
- Proper utilization of single node since new instances of worker are added/deleted using docker-compose up scale <desired-number>command

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

- Continuous integration and delivery pipeline needs to be implemented so that a new machine learning model can be deployed immediately to worker service as soon as a new machine learning model is available in the git branch.
- Longer time to configure the instance
- We cannot define desired state of target machines like right version of package, expected contents into configuration files, expected permissions, expected services like configuration management tools

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

- Implementation of continuous integration and delivery pipeline will accelerate deploy the changes made within the model into the production environment.
- Use of an ansible playbook script to configure the instance at run time.
- Creation of offline image build using packer

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

Horizontal scaling refers to adding more nodes to meet demand of CPU,memory/disk while vertical scaling refers to adding more resources (CPU, memory, disk) within the same node.

I believe that strategy adopted in task-2 follows pseudo horizontal scaling (not fully horizontal scaling). If we assume that each container is a computing node then scaling done using docker-compose will create new instances of the container but within the same node not in a different node which match with the definition of horizontal scaling.

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

```
 ---> 87ee17822c20
Step 5/6 : ENTRYPOINT ["python"]
 ---> Using cache
 ---> d8cc5f422e3e
Step 6/6 : CMD ["./app.py","--host=0.0.0.0"]
 ---> Using cache
 ---> 6f85446ab304
Successfully built 6f85446ab304
Successfully tagged production_server_worker_1:latest
Image for service worker_1 was built because it did not already exist. To rebuild this image you must use `docker-compose build` or `docker-compose up --build`.
Creating production_server_rabbit_1 ... done
Creating production_server_worker_1_1 ... done
Creating production_server_web_1      ... done
Cloud-init v. 20.3-2-g371b392c-0ubuntu1~20.04.1 running 'modules:final' at Sun, 24 Apr 2022 14:46:29 +0000. Up 55.03 seconds.
Cloud-init v. 20.3-2-g371b392c-0ubuntu1~20.04.1 finished at Sun, 24 Apr 2022 14:54:40 +0000. Datasource DataSourceOpenStackLocal [net,ver=2].  Up 545.22 seconds
```

Cloud-init script (cloud-cfg.txt) is written to bootstrap instance with dependencies of docker,docker-compose and clones public git repository. The script is extended to create docker image from Dockerfile located in a cloned git repository and start all services of the prediction model using docker-compose.

The IAC code is written inside start_instance.py which is designed to create nova instance in our snic cloud based on parameters image_name, flavor, private_net, etc. IAC code also passes cloud-init script during nova instance creation.

It takes 12 mins to complete all the instructions of cloud-init since docker builds the image from Dockerfile at runtime. The execution time can be reduced if we have uploaded docker image into the docker hub registry.

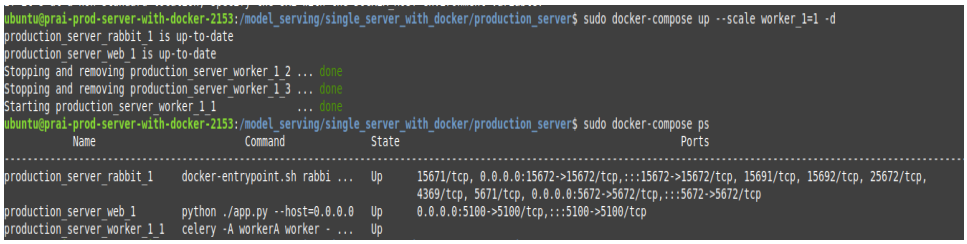
Since we are using docker to containerize all the dependencies of frontend-service, backend-service, worker-service. There is proper isolation of dependencies.

```
ubuntu@prai-prod-server-with-docker-2153:/model_serving/single_server_with_docker/production_server$ sudo docker-compose ps
            Name                          Command               State                                                Ports
-----------------------------------------------------------------------------------------------------------------------------------------------------------
production_server_rabbit_1     docker-entrypoint.sh rabbi ...   Up      15671/tcp, 0.0.0.0:15672->15672/tcp,:::15672->15672/tcp, 15691/tcp, 15692/tcp, 25672/tcp,
                                                                        4369/tcp, 5671/tcp, 0.0.0.0:5672->5672/tcp,:::5672->5672/tcp
production_server_web_1        python ./app.py --host=0.0.0.0   Up      0.0.0.0:5100->5100/tcp,:::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
ubuntu@prai-prod-server-with-docker-2153:/model_serving/single_server_with_docker/production_server$
```

Using docker-compose up –scale worker_1=3, we were able to replicate worker service and

```
ubuntu@prai-prod-server-with-docker-2153:/model_serving/single_server_with_docker/production_server$ sudo docker-compose ps
            Name                          Command               State                                                Ports
-----------------------------------------------------------------------------------------------------------------------------------------------------------
production_server_rabbit_1     docker-entrypoint.sh rabbi ...   Up      15671/tcp, 0.0.0.0:15672->15672/tcp,:::15672->15672/tcp, 15691/tcp, 15692/tcp, 25672/tcp,
                                                                        4369/tcp, 5671/tcp, 0.0.0.0:5672->5672/tcp,:::5672->5672/tcp
production_server_web_1        python ./app.py --host=0.0.0.0   Up      0.0.0.0:5100->5100/tcp,:::5100->5100/tcp
production_server_worker_1_1   celery -A workerA worker - ...   Up
ubuntu@prai-prod-server-with-docker-2153:/model_serving/single_server_with_docker/production_server$ sudo docker-compose up --scale worker_1=3 -d
production_server_rabbit_1 is up-to-date
Starting production_server_worker_1_1 ...
Starting production_server_worker_1_1 ... done
Creating production_server_worker_1_2 ... done
Creating production_server_worker_1_3 ... done
ubuntu@prai-prod-server-with-docker-2153:/model_serving/single_server_with_docker/production_server$ sudo docker-compose ps
            Name                          Command               State                                                Ports
-----------------------------------------------------------------------------------------------------------------------------------------------------------
production_server_rabbit_1     docker-entrypoint.sh rabbi ...   Up      15671/tcp, 0.0.0.0:15672->15672/tcp,:::15672->15672/tcp, 15691/tcp, 15692/tcp, 25672/tcp,
                                                                        4369/tcp, 5671/tcp, 0.0.0.0:5672->5672/tcp,:::5672->5672/tcp
production_server_web_1        python ./app.py --host=0.0.0.0   Up      0.0.0.0:5100->5100/tcp,:::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
ubuntu@prai-prod-server-with-docker-2153:/model_serving/single_server_with_docker/production_server$
```

using docker-compose up –scale worker_1=1, we were able to reduce worker service




Continuous Integration and Continuous Delivery (CI/CD)

Task-3: Deployment of multiple servers using Ansible

Questions

1 - What is the difference between CloudInit and Ansible?

Cloud-init is a tool used to initialize cloud instances based on Linux distribution. The cloud-init script is executed during boot steps. It is not a configuration management tool. It is not meant to be run again afterwards to update the configuration, like one would with Ansible. It is normally used to initialize cloud instances with some set of softwares, ssh-keys, setting time zone, updating packages, managing users and roles,which need to be present after boot.

On the other hand, Ansible is one of the configuration management tools. By executing Ansible playbook on target machines, Ansible can verify target machines configurations files are identical to desired state or there is an expected user and roles are present on target machines or not. If the present state of target machines is not in desired state, then ansible will execute tasks defined within the playbook to bring servers into desired state. Ansible actions are independent and are ssh based tools so Ansible playbook can be executed only after ssh daemon is up and running on target machines. Ansible can also be used as IAC.

2 - Explain the configurations available in dev-cloud-cfg.txt and prod-cloud-cfg.txt files.

Both dev/prod-cloud-cfg.txt are used to bootstrap ssh daemon with ssh key defined in “ssh_authorized_keys” key into dev and prod instance. Cloud-init script creates a home directory with “home/appuser”, user with “appuser” and is assigned into a group named “sudo”. On successful ssh login with appuser, private key against public key into dev/prod servers,

appuser uses “/bin/bash” shell.Byobu is also made installed into dev/prod server.

dev-cloud-cfg.txt

```
#cloud-config
users:
 - name: appuser
   sudo: ALL=(ALL) NOPASSWD:ALL
   home: /home/appuser
   shell: /bin/bash
   ssh_authorized_keys:
     - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDw3qnR1irPiv3fts6L8Q8KRz

byobu_default: system
```

3 - What problem have we solved by using Ansible?

Ansible is one of the easy-to-use configuration management and deployment tools. Ansible allows us to define desired state of target machines like right version of package, expected contents into configuration files, expected permissions, expected services are running into playbook in declarative way. Ansible actions are idempotent meaning that It’s safe to run an Ansible playbook multiple times against a server. Ansible is good at executing tasks on multiple servers in a defined order like bringing up databases before web servers. Ansible playbooks can be extended to provision public servers like EC2, Azure, Digital Ocean, Google Compute Engine, Linode, Openstack, etc. Ansible is agentless so nothing needs to be installed on target machines. ssh and python simplejson needs to be present on target machines so that ansible control machine can execute. Ansible has been used with any number of nodes. Ansible is push based by default so target machines should repeatedly poll the server to listen to changes.

Task-4: CI/CD using Git HOOKS

Questions

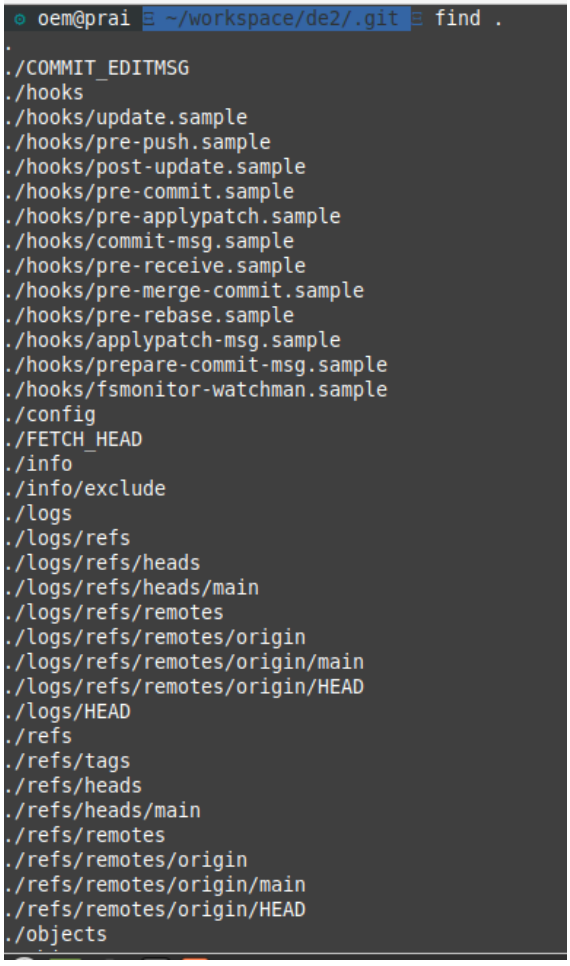
1- What are git hooks? Explain the post-receive script available in this task.

Git hooks are used to run scripts whenever a commit or a patch occurs in a repository. Hooks set up in private repository are not propagated to and do not alter the behavior of the new clone. Git hooks created into the working directory .git/hooks will execute scripts into the local working directory.

```
#!/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"
```

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

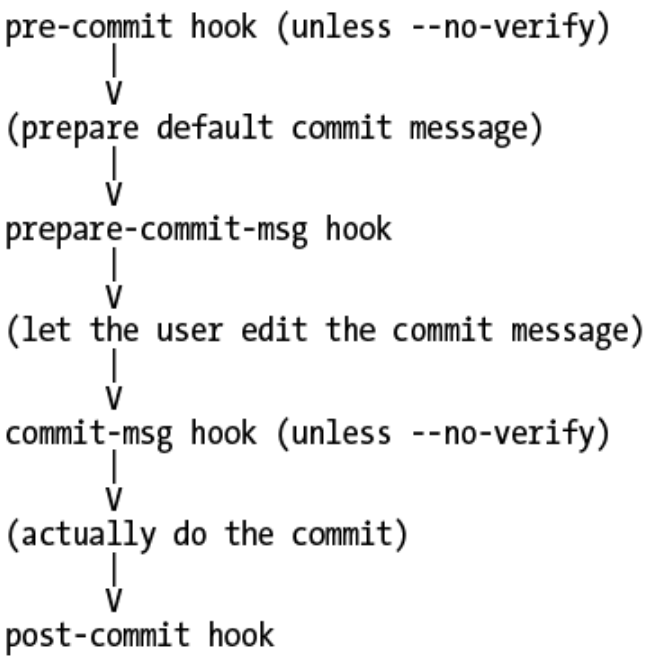
Using the git init command, git creates a .git hidden directory in the working directory. Git uses that .git directory to keep all revision information




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.

Commit-Related Hooks

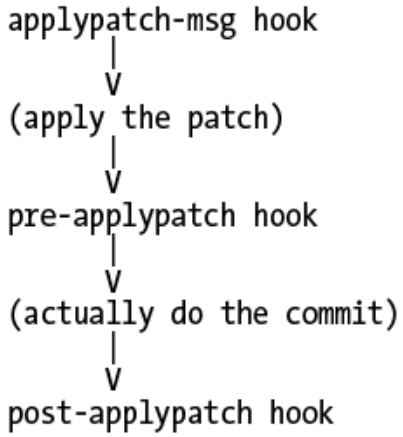
After executing **git commit** command, git goes through following steps




- ***pre-commit*** hook gives you the chance to immediately abort a commit if something is wrong with the content being committed. The *pre-commit* hook runs before the user is allowed to edit the commit message, so the user won't enter a commit message only to discover the changes are rejected. You can also use this hook to automatically modify the content of the commit.
- ***prepare-commit-msg*** lets you modify Git's default message before it is shown to the user. For example, you can use this to change the default commit message template.
- ***commit-msg* hook** can validate or modify the commit message after the user edits it. For example, you can leverage this hook to check for spelling mistakes or reject messages with lines that exceed a certain maximum length.
- ***post-commit*** runs after the commit operation has finished. At this point, you can update a log file, send email, or trigger an autobuilder, for instance. Some people use this hook to automatically mark bugs as fixed if, say, the bug number is mentioned in the commit message. In real life, however, the *post-commit* hook is rarely useful, because the repository that you *git commit* in is rarely the one that you share with other people. (The *update* hook is likely more suitable.)

Patch-Related Hooks

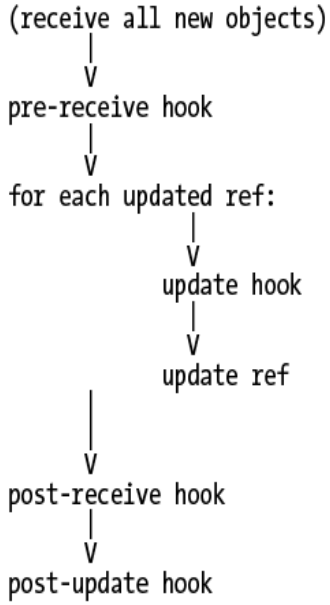
After executing **git am** command, git goes through following steps




- ***applypatch-msg hook*** is used to evaluate the commit message attached to the patch and make changes to commit message at this point. For example, you can choose to reject a patch if it has no Signed-off-by: header.
- ***pre-applypatch hook*** is similar to pre-commit patch. It is executed after patch is applied but before commiting the result.
- **post-applypatch hook** is similar to the post-commit script.

Push-Related Hooks

After executing git push command, the receiving end executes following steps



Push related hooks are executed always on the receiver side during updating refs (branches/tag).

- ***pre-receive hook*** receives a list of all the refs that are to be updated, including their new and old object pointers. The only thing the pre-receive hook can do is accept or reject all the changes at once.
- ***update hook*** is called exactly once for each ref being updated. The update hook can choose to accept or reject updates to individual branches, without affecting whether other branches are updated or not. Also, for each update you can trigger an action such as closing a bug or sending an email acknowledgment. It's usually better to handle such notifications here than in a post-commit hook, because a commit is not really considered "final" until it's been pushed to a shared repository.
- ***post-receive hook*** receives a list of all the refs that have just been updated. Anything that post-receive can do could also be done by the update hook, but sometimes post-receive is more convenient. For example, if you want to send an update notification email message, post-receive can send just a single notification about all updates instead of a separate email for each update.
- ***Post-update hook*** This hook is meant primarily for notification, and cannot affect the outcome of git receive-pack. post-update can track changes within branches that have changed but not what their old values were, so it is a poor place to do log old..new.

Local Repository Hooks

- ***pre-rebase hook*** is executed when an attempt is made to rebase a branch. This is useful because it can stop you from accidentally running git rebase on a branch that shouldn't be rebased because it's already been published.
- ***post-checkout*** is executed after you check out a branch or an individual file. For example, you can use this to automatically create empty directories (Git doesn't know how to track empty directories) or to set file permissions or Access Control List (ACLs) on checked out files (Git doesn't track ACLs). You might think of using this to modify files after checking them out—for example, to do RCS-style variable substitution—but it's not such a good idea because Git will think the files have been locally modified. For such a task, use smudge/clean filters instead.
- ***pre-auto-gc*** helps git gc --auto decide whether or not it's time to clean up. You can make git gc --auto skip its git gc task by returning non zero from this script. This will rarely be needed, however.
- If we execute ***git help hooks*** there are number of hooks which I have listed only

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

I have attached a separate pdf for this problem.

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

After development server and production server is created in snic cloud using IAC code (start_instances.py)

The development server will use an SSH connection to connect to the production server with key-pair authentication (by sharing public key). This will allow the development node to actively push content into the production server.

In the production server, the jump directory is created in the home directory with ownership to user appuser. Jump directory is initialized into a new git repository using git init –bare command.

```
appuser@prod-server-with-docker-8060:~/my_project$ git init --bare
Initialized empty Git repository in /home/appuser/my_project/
```

```
appuser@prod-server-with-docker-8060:~/my_project$ ls
HEAD  branches  config  description  hooks  info  objects  refs
```

Executable post-receive git hook is created within the git repository located in my_project of prod-server. The purpose of post-receive hook is to synchronize contents received from jump directory”/home/appuser/my_project” to working directory “/model_serving/ci_cd/production_server”

```
appuser@prod-server-with-docker-8060:~/my_project/hooks$ cat post-receive
#!/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 $ref successfully received.  Doing nothing: only the master branch may be deployed on this server."
    fi
done
appuser@prod-server-with-docker-8060:~/my_project/hooks$
```

Initial files within the git working directory of prod-server doesn’t contain model.json and model.h5

```
appuser@prod-server-with-docker-8060:/model_serving/ci_cd/production_server$ ls
Dockerfile  __pycache__  app.py  docker-compose.yml  pima-indians-diabetes.csv  requirements.txt
  run_task.py  static  templates  workerA.py
appuser@prod-server-with-docker-8060:/model_serving/ci_cd/production_server$ ls
```

Git remote of development server is referenced to production server's git using following command

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

Pushing the committed new model(model.json) and weights(model.h5) files into the git directory of dev-server, will immediately push contents directly to the git directory of prod-server because of our customized executable post-receive git hook.

We can observe this action in the picture

```
appuser@dev-server-8060:~/my_project$ ls
model.h5  model.json
appuser@dev-server-8060:~/my_project$ git add .
appuser@dev-server-8060:~/my_project$ git commit -m "new model"
[master 450b3fe] new model
 2 files changed, 1 insertion(+)
 create mode 100644 model.h5
 create mode 100644 model.json
appuser@dev-server-8060:~/my_project$ git push production master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 2 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 2.39 KiB | 816.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
remote: Master ref received.  Deploying master branch to production...
To 192.168.2.61:/home/appuser/my_project
   461b98c..450b3fe  master -> master
appuser@dev-server-8060:~/my_project$
```

As we can see, a new model (model.json) and new weight (model.h5) is added to the git directory of prod-server.

```
appuser@prod-server-with-docker-8060:/model_serving/ci_cd/production_server$ ls
Dockerfile  __pycache__  app.py  docker-compose.yml  model.h5  model.json  pima-indians-diabetes
.csv  requirements.txt  run_task.py  static  templates  workerA.py
appuser@prod-server-with-docker-8060:/model_serving/ci_cd/production_server$
```