UPPSALA
UNIVERSITET

Data Engineering II

Assignment 1

Jinglin Gao

April 27, 2023

1 Task 1

1.1 Q1

This application uses the Cloudinit package in combination with OpenStack APIs to start a new virtual machine and contextualize it at runtime. The working environment contains three parts: Flask-based web application as a frontend server, Celery and RabbitMQ server for the backend server, and Model execution environment based on Keras and TensorFlow. In the whole application, Flask is used to create the frontend server, which serves as the user-facing part of the application. It handles incoming requests from users, processes them, and sends appropriate responses. Celery is an open-source distributed task queue system in Python that allows one to offload time-consuming and resource-intensive tasks to be executed asynchronously in the background, while the main application continues to process other tasks or respond to user requests. RabbitMQ is the message broker used by Celery, it handles communication between the main application and the workers, allowing tasks to be queued, scheduled, and executed asynchronously. Keras and TensorFlow are popular deep learning libraries in Python that are used for building and training neural networks.

Basically, the application works as this:
1. User submit the machine learning task through frontend server, which here is powered by Flask. The Flask server receives the request and sends a message to RabbitMQ, specifying the processing task to be performed.
2. RabbitMQ receives the message from the Flask server and queues it in a task queue. The task queue holds the messages until they are picked up by Celery workers for processing.
3. Celery workers, which are separate processes or threads running in the background, continuously monitor the task queue for new messages. When a message is available, a Celery worker picks it up, retrieves the processing task details and starts processing it. In this task, we set up a model execution environment using Keras and TensorFlow to perform deep learning tasks.
4. Celery will update the Flask frontend server with the progress or completion status.
5. Once the task is done by the worker, the Flask frontend server can respond to the user with the results.

1.2 Q2

In task-1 we are using dynamic contextualization using Cloudinit package. Cloudinit is a powerful tool for dynamic contextualization in virtual machines. But there are some drawbacks to it:
1. Cloud-init is primarily designed for use with cloud computing environments that support the OpenStack API or other cloud platforms, which may limit its compatibility with other virtualization technologies or environments.
2. Cloud-init have dependencies on other software packages or libraries that need to be installed and configured in the VM. This may increase the complexity of the setup process and potential points of failure.
3. Debugging and troubleshooting cloud-init configurations or scripts can be challenging, as errors or issues may not always be easy to identify or resolve.
4. Cloud-init may involve passing sensitive data, such as SSH keys or user credentials, during the contextualization process. Proper security measures should be taken to protect this data and ensure that it is not exposed or leaked during the VM boot process.
5. We install packages and dependencies directly on the system. If there are conflicts between dependencies then might cause problems.
6. This strategy we simply install everything on one virtual machine. However the software might need different versions of dependencies, and that will cause problems. There is no isolation between each server.

1.3 Q3

Now when we contextualize the VM we fetch the image and flavor based on the provided names and create a new instance with the specified image, flavor, user data(cloud-config), network, and security groups. To reduce the deployment time, we can try these methods:
1. Create a VM from a pre-built image that already has the required software and configurations can save time compared to installing and configuring software during the contextualization process. So if there is a suitable pre-built image is available in the OpenStack environment then use it directly instead of creating a VM from a generic image is can reduce the time.
2. Using asynchronous provisioning to optimize the overall deployment time. Asynchronous provisioning allows one to create a VM in the background and continue with other tasks instead of waiting for the VM creation process to complete.
3. Cloud-init supports data sources like NoCloud, which allows one to provide cloud-init configuration data directly as a file or a set of files embedded in the VM image. Utilizing the local data sources can eliminate the need to fetch data from external sources, which can help reduce deployment time.

1.4 Q4

In task-1 we use OpenStack APIs to start a new VM and contextualize it at run time. Here we learn how to do dynamic contextualization using Cloud-init package. First we start a new VM and login to it. In order to run the contextualization code we need to have OpenStack API environment running. So we first update "apt" by using these two commands:

```
ubuntu@jinglin-a2:~$ sudo apt update
ubuntu@jinglin-a2:~$ sudo apt upgrade
```

Then we install the needed client tools and API for OpenStack. We also download Runtime Configuration (RC) file and copy it to the VM in order to use it later. We can check the variables we need through command `openstack server list`. Then we set the variables in `start_instance.py` script as follows:

```
flavor = "ssc.medium"
private_net = "UPPMAX 2023/1-1 Internal IPv4 Network"
floating_ip_pool_name = "Jinglin_A2"
floating_ip = "130.238.28.150"
image_name = "Ubuntu 22.04 - 2023.01.07"
```

In `cloud-cfg.txt` file contains the user data and specifies how we want to deploy our new VM. By running the python script we can launch the new instance as our production server.

```
ubuntu@jinglin-a2:~/model_serving/openstack-client/single_node_without_docker_client$ python3 start_instance.py
user authorization completed.
Creating instance ...
waiting for 10 seconds..
Instance: prod_server_without_docker_1002 is in BUILD state, sleeping for 5 seconds more...
Instance: prod_server_without_docker_1002 is in ACTIVEstate
```

The whole application now is set up and ready for users to use it.

Welcome to the Machine Learning Course.

Accuracy is 71.42857313156128

True	Predicted
1	0
1	1
1	1
1	0
0	0
0	0
0	0

2 Task 2

2.1 Q1

The difference between the two deployment strategies is that in task-2 we use docker for containerization, while in task-1 we install packages and dependencies directly on the system. Compared to the deployment strategy we use in task-1, there are some benefits to using this strategy:
1. Docker allows for encapsulating applications and their dependencies into containerized units, providing a higher level of isolation between different components or services running on the same host. This can help prevent conflicts between dependencies and provide a consistent runtime environment.
2. Docker images are built from declarative Dockerfiles, which specify the exact dependencies, configurations, and steps to build the application image.
3. Docker provides built-in features for scaling applications horizontally by creating multiple instances of containers running the same application. This can help achieve better utilization of resources and handle increased load or demand for the application.

2.2 Q2

1. Don't have explicit error handling and fault tolerance mechanisms in place. If a package installation or Docker container build fails, there may not be proper error handling to handle the failure gracefully and recover from it.
2. We directly install applications and dependencies from package managers or Docker repositories without specifying exact version numbers. This can lead to potential compatibility issues or breakages if newer versions of packages or dependencies are released.

2.3 Q3

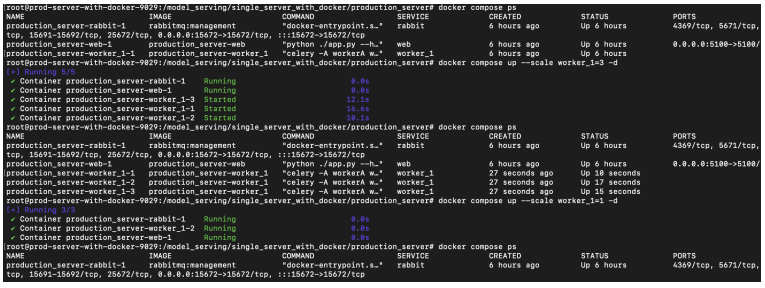
1. Add error handling and fault tolerance mechanisms in the cloud-init configurations to handle unexpected failures gracefully. For example, use conditional statements or try-catch blocks in scripts to handle errors.
2. Implement thorough dependency management practices, such as using virtual environments, package managers, and version pinning to ensure consistent dependencies across the application.
3. We can only set up one node one time to build the application. If we want to set up a cluster then we need a more complex deployment strategy.

2.4 Q4

Horizontal scalability is the ability to increase capacity by connecting multiple hardware or software entities so that they all work as a single logical unit. This involves adding more machines to the system, rather than adding more resources to a single machine. This allows for a more efficient use of resources, as each machine can be utilized to its maximum capacity. In the case of Docker, this can be achieved by creating more containers and spreading the load across multiple machines.
Vertical scalability is the ability to increase the capacity of existing hardware or software by adding resources, such as adding more memory or CPU. The main advantage of this method is that it is relatively straightforward to implement, as it simply requires adding more resources to an existing machine. In the case of Docker, this can be achieved by modifying the container specifications, such as increasing the memory limit or CPU quota, or by adding more containers to a single host machine.
The strategy adopted in task-2 follow horizontal scalability since it can create more containers as workers instead of increase the capacity of existing hardware.

2.5 Q5

The deployment process we use in task-2 is the same as task-1, but this time we use Docker containers. The application structure is the same as we discussed before in task-1.
Here we create a multi-container Docker application, it specifies three services:
1. "web" service: build an image from the current directory with a Dockerfile. It runs the container with the host network mode. It restarts the container always. It maps the host's port 5100 to the container's port 5100. It depends on the "rabbit" service.
2. "rabbit" service: It uses the "rabbitmq: management" Docker image, which is the official RabbitMQ image with the management plugin included. It sets environment variables for RabbitMQ default user and password. It maps the host's port 5672 to the container's port 5672 for RabbitMQ message broker communication, and port 15672 to the container's port 15672 for RabbitMQ management plugin UI.
3. "worker_1" service: It builds an image from the current directory with a Dockerfile. It specifies "celery" as the entrypoint command for the container. It passes command line arguments to Celery to start a worker with the "workerA" application and log level set to "debug". It links the "rabbit" service to allow communication between the "worker_1" container and the "rabbit" container. It depends on the "rabbit" service.
So in task-2 we can specify each container's mission and scale out by creating more workers in the cluster.



The application we built is like this:

Welcome to the Machine Learning Course.

Accuracy is 71.42857313156128

True	Predicted
1	0
1	1
1	1
1	0
0	0
0	0
0	0

3 Task 3

3.1 Q1

CloudInit and Ansible are both tools used for configuration management and automation in IT environments, but they have some key differences:
1. CloudInit is a tool specifically designed for configuring VM when they are launched in a cloud computing environment. It is used during the initial boot process of a VM to configure settings such as hostname, user accounts, SSH keys, network interfaces, and more. On the other hand, Ansible is a general-purpose automation tool that can be used for configuring and managing a wide range of IT resources, including VMs, physical servers, network devices, databases, and more.
2. CloudInit typically operates at the instance level, meaning it configures settings for individual VMs, and it may not have built-in support for managing configurations across an entire cluster of VMs. On the other hand, Ansible can handle configurations across a cluster of systems.

3.2 Q2

`dev-cloud-cfg.txt`:
In this `.txt` file defines a user account named ”appuser”. It specifies the user’s name, privileges, home directory, and default shell. It also includes an SSH public key that is allowed to authenticate as this user. ”byobu_default” sets the default shell for the ”appuser” to ”byobu”, which is a terminal multiplexer that provides a customizable text-based interface for managing multiple terminal sessions.

`prod-cloud-cfg.txt`:
Same as the configurations in `dev-cloud-cfg.txt`.

3.3 Q3

By using Ansible, we can handle cluster-level configuration. Using CloudInit we are limited to one VM since it can just configure one virtual machine at one time, so if we want a cluster then they are running on different nodes.

4 Task 4

4.1 Q1

Git hooks are a rather simple concept that was implemented to address a need. Git hooks are scripts that allow you to automate certain actions or tasks in a Git repository. Git hooks are event-based. When you run certain git commands, the software will check the hooks directory within the git repository to see if there is an associated script to run.
The post-receive script in this task is a Git server-side hook written in Bash. It is a 'pre-receive' hook, which is triggered on the remote Git server when a push is received. The script uses a 'while' loop to read input from 'stdin'. The 'pre-receive' hook receives input in the form of lines containing information about the old revision, new revision, and reference for each push. The script uses a regular expression('= .*/master$) to check if the pushed reference ends with '/master', which indicates that the push is targeting the 'master' branch. If the pushed reference is the 'master' branch, the script echoes a message indicating that the 'master' branch is being deployed to production. It then uses 'git checkout -f' to forcibly check out the latest changes from the 'master' branch to a specified work tree and Git directory. If the pushed reference is not the 'master' branch, the script echoes a message indicating that only the 'master' branch can be deployed on this server, and no further action is taken.

4.2 Q2

We create a new empty Git repository in the production container because we can use the push-to-deploy model in order to update the web server whenever we push to a bare git repository. We need this empty repository to push changes to, and a git hook that will execute whenever a push is received.

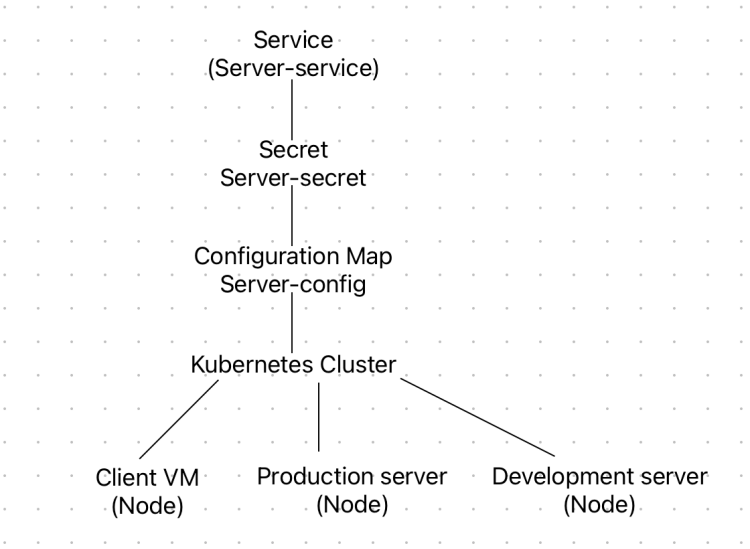
4.3 Q3

There are two groups of Git hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, while server-side hooks run on network operations such as receiving pushed commits.
Client-Side Hooks
1. Committing-Workflow Hooks
Committing hooks are used to dictate actions that should be taken around when a commit is being made. They are used to run sanity checks, pre-populate commit messages, and verify message details. You can also use this to provide notifications upon committing.
2. Email-Workflow Hooks
This kind of hooks encompasses actions that are taken when working with the emailed patches. Projects like the Linux kernel submit and review patches using an email method. These are in a similar vein as the commit hooks, but can be used by maintainers who are responsible for applying submitted code.
Server-Side Hooks
1. Pre-receive and post-receive
These are executed on the server receiving a push to do things like check for project conformance and to deploy after a push.
2. Update
This is like a pre-receive, but operates on a branch-by-branch basis to execute code prior to each branch being accepted.[1]

4.4 Q4

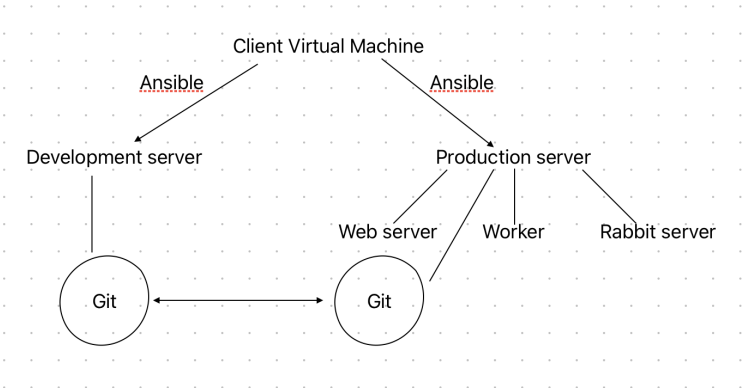
Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. The configuration map object in Kubernetes is used to store configuration

data, such as server settings, as key-value pairs. The secret object in Kubernetes is used to store sensitive data, such as passwords, securely. The service object in Kubernetes is used to expose the servers to the network.
If we want to deploy multiple servers with Kubernetes, first we need to set up a Kubernetes cluster with at least three nodes for Client VM, production server and development server. Second deploy docker images to Kubernetes cluster, deploy the docker images of the application components to the Kubernetes cluster. Then configure Kubernetes services and load balancer to expose the deployed application components.

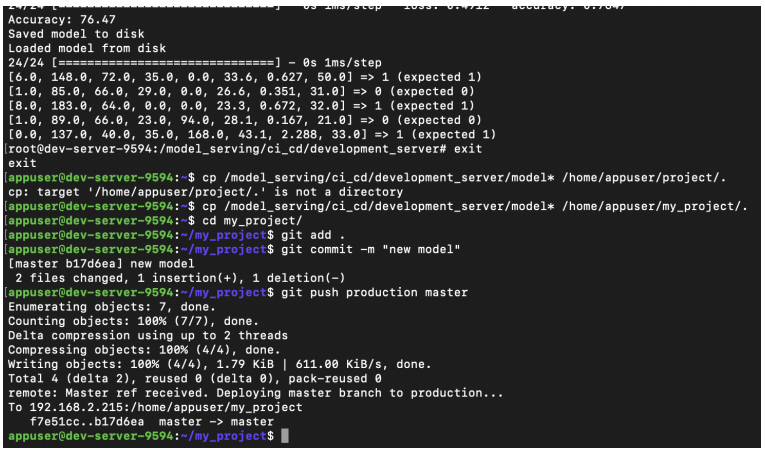


4.5 Q5

Since Task-4 is the continuation of Task-3, we already got one client VM, one development VM, and one production VM. The production server has three containers which are `production_server-worker_1-1`, `production_server-web-1`, and `production_server-rabbit-1` respectively. So basically we run the application in the production server and we have a separate development server that hosts the development environment and pushes the new changes to the production server. The diagram to show the structure of the system is shown below:



The Git repository we create in the development server and the production server are connected in order to receive any changes we made in the development server. We can tune our model in the development server and as long as we did not push it to the master branch there will be no change in the web page. This ensures the continuous integration and delivery of new machine learning models in a production environment. And we can upgrade the web page after we made a change.



References

[1] DigitalOcean Community. *How To Use Git Hooks to Automate Development and Deployment Tasks*. Retrieved from DigitalOcean Community website. URL: `https://www.digitalocean.com/community/tutorials/how-to-use-git-hooks-to-automate-development-and-deployment-tasks` (**urlseen** 25/04/2023).