CSE 546 — Project Report

Image Recognition Service on Hybrid Cloud

Bharath Kumar Bandaru- 1219442718

Kaveri Subramaniam- 1222089687

Shruti Pattikara Sekaran- 1222257972

1. Problem statement

The principal goal of this project is to build a scalable, efficient Image recognition application, using a deep learning model on a hybrid cloud, employing both resources from Amazon Web Services (AWS) and OpenStack. Particularly, we aim to attain practical experience in developing a hybrid cloud model, integrating the different functionalities running on the private cloud platform (Openstack) as well as the public cloud provided by the AWS. The features of this application include:

- Elastic- The application scales up and down as per the load.
- Concurrent- It handles multiple requests simultaneously, through multithreading.
- Fast- The application offers quick response time, as its web tier is asynchronous.
- Cache- Every request image and its classification result is stored in a cache, resulting in faster responses and reduced load on the application.

2. Design and implementation

2.1 Architecture

Openstack: OpenStack is a cloud operating system that gathers and controls large pools of storage, compute and networking resources, managing and provisioning them through API with common authentication mechanisms. Openstack resources can be accessed through a GUI dashboard, acting as a common control mechanism over the resources. Other components provide orchestration services, fault management and service management to ensure high availability for the users' applications.

Web-Tier: The web-tier is the entry component of the application. The web-tier handles the user's request and sends an appropriate response. Only one web-tier instance would be up throughout the entire application. The web-tier application runs on the openstack (private cloud) platform. It also handles part of auto-scaling i.e. creating multiple app-tier instances. The web-tier converts the input requests into an appropriate message and transmits it to the app-tier via an SQS queue (request queue). It also processes responses from the other SQS queue (response queue) and forwards it to the appropriate user request.

Application Tier: The application-tier (app-tier) runs on an ec2 instance. The classification of images is performed in app-tier and the results are sent back to web-tier via SQS queues. The app-tier reads and processes the requests from the request-queue. The response is sent back to the web-tier using response-queue. After successful classification of images the app-tier stores the input image data and its corresponding output data in S3 buckets. At any given point in time, the minimum number of app-tier instances which can be running is 0, and maximum number of app-tier instances which can exist is 20.

Storage Layer: The storage layer is implemented through the Simple Storage Service (S3) provided by AWS. S3 is a reliable, scalable object based storage, wherein all data is managed as objects internally, in a bucket structure. The application tier uses 2 different buckets for this application. The input image name and the image are stored as objects in the input bucket, whereas the output bucket consists of the input image name and its corresponding label is stored as key-value pairs.

Messaging Layer: The messaging layer is developed utilizing the Simple Queue Service (SQS) of AWS. SQS is an autoscaling, reliable messaging queue service, which provides the input and output queues for this application, between the Web Tier (Client) and the App Tier (Server). The maximum size of a single message is restricted to 256KB.

AWS-EC2: Elastic Cloud Compute is an AWS service which is the basic secure infrastructure on which users can build and run their applications or programs. For this application, it is utilized as the foundation upon which the App tier executes.

2.2 Autoscaling

In the project the autoscaling i.e. scaling up and scaling down are managed by app-tier and web-tier services. The web-tier handles the autoscaling of app-tier instances by monitoring the number of messages in the request queue. If the number of messages is more than a certain threshold value then the web-tier automatically scales up app-tier instances to meet this increased demand for app-tier instances. In the current application, we decided that each app-tier can process 1 message at a time, hence when messages are more than 1 then the web-tier determines the new instances required, a max of 20 instances, and scales up those many instances to process messages in the request queue. Web-tier takes care of the max and min limit on the instances of app-tier that can be running at any given point of time. If the app-tier instance doesn't process any messages before a threshold period of time then that app-tier instance gets terminated.

2.3 Member Tasks

We decided to use python as the programming language due to the following reasons.

1. Ease to use & have prior experience.

2. Contains numerous libraries.
3. Lot of community help is available online.

We have used virtualbox and ubuntu for running the openstack

Bharath Kumar Bandaru

Development:

I have created methods to create/destroy the SQS and S3 resources in the cloud. A config file is created with various parameter values so that it is easier to adapt to the changes in the code that are made.

Before creating the S3 buckets or SQS queues I have to destroy any current resources that are present in the aws. So I decided to run the methods that clears the infrastructure first. As a part of this I have written code for deleting the S3 buckets and SQS queues. The S3 buckets deletion method checks for the existing buckets that are present and deletes all the objects first then the bucket itself. This process is done for all the buckets that are present. Similar to the S3 buckets deletion I wrote the code for deleting SQS queues.

Created a python file which will take the ami-image name and the various parameters to create the web-tier ec2-instance. I have also written methods to create a keypair which is used during the ec2-instance and ssh connection in the openstack cloud. Configured the ec2-instance with the security group, which will allow for the traffic flow from the instance. I have allocated the floating IP to the instance which can be used to send requests. I have helped the team mates in solving a couple of issues that occurred in openstack installation.

All the code that was tested and verified and made sure no issues occurred and handled correctly.

Testing:

I have downloaded the workload generator files from the github to test the workflow of the application. Next, after creating the instance, the IP address is configured to send the input request to the web-tier application instance. I ran the multi_workload_generator to test the workflow of the application. Also, I have verified the requests were going through the web-tier instance then to the SQS queues then to the app-tier instance. Once the responses were received by the workload generator, the results were verified.

I have also verified the objects in input and output buckets and the content in the objects. Looked at the SQS metrics, and verified that the messages were going through both the input and output SQS queues.

Shruti Pattikara Sekaran

Development:

I worked on creating the ec2-instance images for both web-tier and app-tier applications. I used the code files from project 1 to achieve the tasks. First, I worked on the app-tier application instance by creating the instance from the ami-image mentioned in the project instructions. Then I have configured the instance with the aws credentials and the packages required to run the app-tier application. Once the setup is done, I have copied the files to the app-tier instance and configured the systemctl service of the app-tier application. Additionally, I verified that the systemctl service is running and tested it by restarting the app-tier instance. Then I have created the ami-image for app-tier, which is used by the web-tier application.

Next, I worked on the web-tier application, for this I have created a new instance, similar to the app-tier instance. I have configured the instance with aws credentials and installed the necessary packages required for the web-tier application to run. After that, I have uploaded the web-tier code and configured a systemctl service for the web-tier application. The service is verified by restarting the instance.

I have created the ami-image of the web-tier instance, next using the export image command: aws ec2 export-image --image-id ami-0cb04acc0def28bb8 --disk-image-format VMDK--s3-export-locationS3Bucket = <bucket_name>, S3Prefix = exports /, I have exported and downloaded the web-tier image which will be used in the openstack for running the web-tier instance.

Further, I also explored the openstack application and helped with testing the overall application and monitoring metrics.

Testing:

I have verified the basic workflow of the application in the aws and made sure there are no bugs or issues generated using the multi-workload generator tool. Next, I have manually created the ami-images, and verified them by creating the instances from the ami-images. I have also configured the security groups for each instance and made sure that the network traffic is going through.

I have helped with the overall testing of the application in monitoring metrics and objects that are created in the SQS and S3 resources. I have also worked and explored the services in the openstack.

Kaveri Subramanyam

Development:

I mainly worked on the openstack part of the project. I was involved in the openstack setup and tested the services. Based on the instructions provided in the project, I have downloaded the virtualbox and installed ubuntu. Due to the performance issues that might occur, I have configured the ubuntu box with the highest resources allocated.

Once the configuration is done, I have installed Ubuntu in the VM box. I have configured the network adapter in the virtual box settings for enabling port forwarding, this is done so that the requests to the ubuntu machine can be sent from outside the virtual box.

Following the basic setup, I have cloned the open stack project and installed the openstack using the ./stash.sh command. After the installation is done, I have configured the devstack application with the commands mentioned in the Procedure for Openstack section in section 4. Using the IP address that I have accessed the openstack GUI interface and logged in using the admin credentials which are configured in the config file. I worked around the GUI and understood the services like nova, horizon, floating IP, network, Images, glance etc. that openstack offers.

During the installation of openstack I faced a lot of issues, like no ssh connection, instances not up and running, the requests from the instances were not going through the internet etc. After the successful setup of the openstack, I have uploaded the web-tier image to the openstack. I have created a common security group which will allow for the traffic to flow through the instance.

Testing:

Once openstack was installed and was running, I created multiple instances and understood the services that openstack provides. Next, I have created the instance for the web-tier application manually, performed the ssh using the created keypair, verified the systemctl service of the web-app, aws configurations and the security groups on the instance for seamless workflow of application. Next, I have created a floatingIP and allocated it to the instance for traffic flow.

Once this is done, I have manually run the multi-workload generator and verified that the requests are going through the web-tier instance and then to the aws resources. Manually verified the classification results that were received by the workload generator.

In the process of testing, I have encountered some bugs which I was able to resolve with some help from my teammates.

3. Testing and evaluation

Openstack was installed as per the given instructions in the project description. The web-tier instance that was configured for project-1 was downloaded and that image is used as the input for creating the instance in the openstack. Once the image was uploaded we verified to see whether the services in the instance were up and running as expected. Then the security groups were attached to allow the traffic flow on the openstack instance. End to end testing is performed to inspect the service's reliability and accuracy. Based on the requirements we have validated that the number of app-tier instances will not go beyond 20 by sending multiple requests at the same time. The Web Tier is inspected to observe the number of requests it puts in the messaging service as well as the EC2 instances it creates. Additionally, the Application tier is checked to analyse whether all the requests from messages are picked up, its input images and names are correctly stored in the appropriate S3 bucket. Further, its capability to classify, store the result in the output bucket, and send it through the message queue to Web tier is also examined.

Lastly, a workload generator is used to specifically test the workflow and classification functionality. The object recognition program's results are then compared with the original results to evaluate the overall accuracy as well as efficiency, which is measured through the response time.

4. Code

The code for the project files web-tier and app-tier was used as of project 1. The web-tier instance AMI image that was re-configured to run the web-tier application in AWS. The new ami-image was downloaded which is used in the openstack to run the web-tier application instead of in AWS.

Technologies used: Python, Flask, Boto3, EC2, SQS, S3, Torch, Openstack

Files:

1. *Images-100:* The input image files that are sent to the application to classify.
2. *App-tier.py*: The code for app-tier application which handles the requests from sqs request queue.
3. *Web-tier.py*: The code for a web-tier application which handles the requests from users and sends the response back to the users.
4. *Image_classification_new.py*: This file classifies the input images and sends the response to the app-tier.
5. *Cloud_project3.py*: The local python file which destroys and creates the infrastructure in aws for the application to run.
6. *Openstack.py:* The local python file that is used to create the instance in the openstack cloud environment for web-tier application.

Cloud_project3.py:

- This file will configure the S3 buckets, SQS images required for the application to process.
- If needed it will also terminate the instances in the aws environment. Every time when this file is run the current resources which are running in aws gets terminated and new resources are created.

Openstack.py:

- This file will create/terminate the instance in the openstack form the ami-image that was downloaded from aws.
- This will also create the keypair required for connecting the instance using ssh.

Image_classification_new:

- The code for the image classification is taken from the ec2-image that was provided for the project.
- The classification code is written in a method called classify which will be called by app-tier for classifying the image.

Web_tier:

- The web-tier application is built using the flask framework to process the users requests. The threading option is enabled so that multiple requests can be handled concurrently.
- Multiprocessing is used for processing the requests, tracking demand of the app-tier instances required, and processing responses from the SQS queue.
- The application runs as a REST application. The start_point, process_response_queue and process_app_tier were executed concurrently using multiprocessing.
- The cache.json file is used to store the results from the response queue which are generated from the app-tier in the form of key, value pairs.
- The start_point sends the input request to SQS and waits for the results. Once the response is received, a response message is created and sent back to the user.
- The process_response_queue manages the messages from the response queue. It looks at the messages from the response-queue and stores the result in the cache.json. After successfully processing the message the message is deleted from the response-queue.
- The process_app_tier monitors the number of instances of app-tier required based on demand. It keeps track of the number of messages in the request-queue and calculates the requirement of app-tier instances. It takes care of the auto-scaling of app-tier instances based on the required number of instances.
- aws ec2 export-image --image-id ami-0cb04acc0def28bb8 --disk-image-format VMDK --s3-export-locationS3Bucket=<bucket_name>,S3Prefix=exports/ -> command to export the ami image to the S3 bucket.

App_tier:

- The app_tier handles the request from the request-queue and performs image classification and sends the response back.
- The read_read_request method looks for the messages from the queue and processes the input request. Using the classify method it will classify the input image and create a response request and send it to the response SQS queue. The input image and the classification results are stored in the S3 bucket.
- The obtained results are sent back to the web-tier application via response-queue SQS.
- If the app_tier is idle for the threshold time then it gets terminated.

4.1 Project setup and execution

Web-tier:

- Created an ec2-instance and configured the required packages and aws credentials.
- Copied the web-tier.py and cache.json file into the instance using the scp command.
- Created a service file using systemctl for web-tier so the web-tiers gets executed on the start of the ec2-instance.
- A new ami-image is created after the environment setup. This image is used to create a web-tier instance.
- The image was downloaded and used as the input for the ec2 instance in the openstack.

App-tier:

- Created an ec2-instance using the image provided for the project (ami-0bb1040fdb5a076bc).
- Installed the required packages like flask, boto3 and aws cli.
- Configured the ec2-instance with the aws credentials.
- Copied the app_tier.py, image_classification_new.py into the classifier folder provided by the image using the scp.
- Created a service file using systemctl for app-tier so that app-tier gets executed on the start of ec2-instance.
- A new ami-image is created after the setup. This image id is used to create the multiple instances for app-tier.

Cloud_project3

- The cloud_project1.py is a local python file which is executed to set up the environment.
- Once the cloud_project1.py is run then it will delete the current infrastructure that is present and create a new infrastructure for the application.
- The cloud_projec3.py file also has the functionality to teardown the aws infrastructure.

Procedure for Openstack:

After installing VirtualBox create a VM with the following configurations:

- RAM: 4096MB
- Storage: VDI with 20GB of 'fixed' storage
- (Optional) Follow the tabs: Network -> Adapter 1 (NAT) -> Port Forwarding and add the following
    - Name Host-Port Guest-Port:
    - Horizon 8888 80
    - SSH 2222 22
- Start the VM, install ubuntu with the user 'devstack' and once it is up and running, login to the terminal and run the following commands to create a new user. This step is also optional but keeps the environment separated between server and client:
    - sudo useradd -s /bin/bash -d /opt/stack -m stack
    - sudo chmod +x /opt/stack
    - echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee /etc/sudoers.d/stack
    - sudo -u stack -i
- Within this user's space install and configure devstack as follows:
    - sudo apt install git -y
    - git clone https://opendev.org/openstack/devstack
    - cd devstack && touch local.conf
    - # In local.conf add the following lines:

      [[local|localrc]]

      ADMIN_PASSWORD=*******

      DATABASE_PASSWORD=$ADMIN_PASSWORD

      RABBIT_PASSWORD=$ADMIN_PASSWORD

      SERVICE_PASSWORD=$ADMIN_PASSWORD

      HOST_IP=10.0.2.15 <enter ip address where horizon is hosted>

      FLOATING_RANGE=172.24.4.0/24 <subnet of floating pool; default>

      TEMPEST_SHARED_POOL=192.168.233.0/24 <subnet of shared pool; default>

      IPV4_ADDRS_SAFE_TO_USE=10.0.0.0/22 <subnet of instance private ip; default>

    - ./stack.sh
- On the browser, access horizon using http://192.168.56.103/dashboard and login using admin and the ADMIN_PASSWORD set in the local.conf file from the previous step

- In the API Access tab, click on Download and select clouds.yaml and add the ‘password’ to the yaml configuration under ‘admin’
- Create a local SSH keypair and upload the public key to horizon, it is also possible to use the console to create the same. Use the prefix id_web.
- The Security Group ‘default’ should allow access to these protocols and ports, if not add them:
    - SSH: TCP 22
    - HTTP: TCP 80
    - Web-Instance server: TCP 5000
    - ICMP on all ports
- Upload web-instance AMI to glance with the name ‘web-tier-image’
- Copy the workload generator and openstack.py to the devstack home directory
- Run cloud_project3.py to setup the AWS portion of the architecture
- Run openstack.py in the VM to launch the web-instance on openstack
- Run multithread_workload_generator.py to use the architecture

Command to run the file: python3 cloud_project1.py, python3 opensource.py

Command to teardown the aws infrastructure: python3 cloud_project.py teardown

Appendix:

Troubleshooting:

- We found that sometimes the clock of the web-instance ami does not seem to align with openstack on launch. This fixes the issue:

    sudo hwclock –hctosys

- We found that the web instance hostname is not being resolved. To fix this, on the instance, in /etc/resolv.conf, add: nameserver 8.8.8.8
- If test results from previous iterations are being cached, on the web-instance flush using:

    echo “{}” > cache.json