CSE 546 — Project Report (Project 3)

Ravindra Aditya Singh (1220224915)

Manisha Nandkumar Phadke (1219889853)

Sagar Subbaiah Kuppanda Cariappa (1219783773)

1. Problem statement

Hybrid clouds are a popular choice for enterprises aiming for digital transformation in their fast-moving businesses. The primary benefits of hybrid clouds are agility, interoperability between cloud and on-premises environments, flexibility to evolving data requirements, among others. We have used resources provided by Amazon Web Services (AWS) to design our first and second projects. AWS is a public cloud provider, and we rented a slice of their distributed data center infrastructure to run our applications. On the other hand, a private cloud ensures better data security and customization at the expense of cost and flexibility to scale vertically. Hybrid cloud, as the name suggests, provides a hybrid mixture of cloud services.

In our first and second project for CSE 546, we implemented Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) by deploying image or video recognition applications on the cloud using Amazon Web Services (AWS). Our third project aims to transport the elastic application developed in the first project into a Hybrid cloud environment. This application should automatically scale in and out depending on the number of active requests and this horizontal scaling should occur on-demand and economically. We will use resources from AWS as well as OpenStack to provide a purposeful cloud service to users.

2. Design and implementation

2.1 Architecture

As per the project specification, we leverage the resources provided by both Amazon Web Services (AWS) as well as OpenStack. AWS is the most popular public cloud provider due its wide range of compute, service and message services, as well as scaling capabilities. OpenStack is a free, open standard cloud computing platform which is usually deployed as an Infrastructure as a Service (IaaS) in public and private clouds.

The project began with the setup of OpenStack. OpenStack is a set of community-developed software components that provide services for cloud architecture. OpenStack provides the option of hosting cloud infrastructure internally or via an OpenStack partner in the marketplace. With OpenStack, we can leverage one of the 180+ OpenStack powered public cloud data centers. OpenStack is a trusted cloud services provider for Blizzard Entertainment, Walmart, etc. Administrators can monitor the resources using an interactive dashboard.

To leverage the resources of OpenStack, we first installed Ubuntu 20.04 on a virtual machine, using VirtualBox. Our virtual machine had Random Access Memory (RAM) of approximately 4 GB and a file storage (hard disk space) of 50 GB.

The installation process of OpenStack started with installing Git by the command line prompt in the terminal:
sudo apt install git -y

The next step was to download DevStack scripts by cloning the git repository containing the required files, using the command:
git clone https://git.openstack.org/openstack-dev/devstack

The next step was to create a DevStack configuration file (local.conf). This file is a modified INI format file which contains information about the configuration files to be altered.
'[[' <phase> '|' <config-file-name> ']]'
<phase> is the set of phase names defined by *stack.sh* and the file name is given by *<config-file-name>*.

Following this, we installed DevStack by the command:
./stack.sh
This installs Horizon, Nova, Glance, Neutron and Keystone services.
Horizon is an extensible dashboard which provides a web-based user interface to OpenStack services, such as Nova, Swift, Keystone, etc. Horizon is known to be manageable, consistent, stable and usable.

Nova allows users to provision compute instances, or virtual servers, and bare metal servers, and it is run as a set of daemons on top of existing Linux servers. Nova requires Keystone, Glance, Neutron and Placement services from OpenStack for basic functioning. It can integrate with other services to include persistent block storage, encrypted disks and bare metal compute instances.

Glance, also known as Image service project, allows users to discover, upload and retrieve data assets in the form of virtual machine (VM) images. It has a RESTful API which allows querying of VM image metadata along with access to the actual image. Glance has a component-based architecture and is highly available, fault tolerant, recoverable and is community-driven.

Neutron provides network connectivity as a service between interface devices managed by other OpenStack services. It consists of an API for users to configure network connectivity and addressing. It manages the creation and maintenance of a virtual networking infrastructure. This infrastructure consists of networks, switches, subnets, and routers for devices which are managed by the OpenStack Compute service (Nova). It consists of a neutron-server, database for persistent storage, and multiple plug-in agents, interfacing with native Linux networking mechanisms, external devices, or SDN controllers.

Keystone is the Identity Service for OpenStack, which provides authentication of clients along with multi-tenant authorization. Keystone consists of a collection of internal services exposed on one or multiple endpoints. These internal services include Identity, Resource, Assignment, Token and Catalog. A successful authenticate call returns a token with the Token service after validating user/project credentials with the Identity service. Keystone uses the Flask RESTful library for allowing users to interface with these services.

After the installation is completed, the terminal displays a list consisting of an IP address, default user and password. We can access the OpenStack Horizon dashboard by typing the following into the browser:
https://IP.address/dashboard
Once we log in using the username and password displayed in the previous step, we can now leverage the functionality of OpenStack services to run our image recognition application from Project 1 on OpenStack.

The next step is to download the Amazon Machine Image (AMI) of our web-tier from Project 1 from AWS and upload this AMI onto OpenStack to run it on our own cloud. To download the AMI, we use the technique of VM Import/Export. This strategy, as detailed in the AWS documentation, allows us to import virtual machine images from our existing virtualization environment to an S3 bucket, and this image can then be retrieved from the S3 in .vmdk format. We access VM Import/Export using the AWS Command Line Interface (CLI).

The first step to perform a VM Import/Export is to set required permissions for both AWS Identity and Access Management (IAM) users and creating a service role. We add permissions to our IAM policy as a JSON file, which specifies where the AMI will be stored. We also create a service role called *vmimport* and attach an IAM policy to this role. We also ensure our Security Tokens are active in the us-east-1 region. To create the service role, we create two JSON files, named *trust-policy.json* and *role-policy.json*. We use the *create-role* command to create a role named *vmimport* and grant VM Import/Export access to it. The *put-role-policy* command allows us to attach the *role-policy.json* policy to the role created. The AMI of the web tier will be stored in the S3 bucket specified in the *trust-policy.json* file.

The next step is to launch an instance on OpenStack using the AMI containing our web tier code, which we imported using the steps above. We log in to the OpenStack dashboard (Horizon) as demo, not root, and create a security group from the Horizon Dashboard of OpenStack.

Security groups specify what kinds of traffic to allow to access the instance, along with the ports and protocol used to access this instance. We can create a security group in the networks tab of Horizon. The rules for a given security group describe whether the traffic is egress (outgoing) or ingress (incoming). We add a rule for ssh, to allow us to ssh into instances in this security group. We also added a rule to ensure access to ICMP traffic.

We create an image in qemu format by creating an Ubuntu 22.04 KVM guest from the cloud image given below:
https://cloud-images.ubuntu.com/jammy/current/jammy-server-cloudimg-amd64.img

We also create an OpenStack instance from this image. Then, we create an instance via the Instances tab, and specify the source (disk image) of the instance as the AMI we imported. We assign the instance to the security group we created, and use the key-pair specified in the project specification to create the

OpenStack instance. While creating the instance, we use the ds2G flavor. This allocates 10GB of disk space, 2GB RAM and 2 virtual CPUs running.

OpenStack has two networks existing by default, called the private and public networks. We use the private network to connect our instance. Next, we associate a floating IP address with this instance. A floating IP address is a publicly routable IPv4 IP address, which allows the public internet to access our VM instance on OpenStack. We now ssh into the instance using the floating IP address using the command line prompt:
ssh -i <.pem file for ssh> ubuntu@<floating-ip-address>

Our instance is ready to be launched. OpenStack ensures sufficient virtual resources are available to provision the instance, and sets up a storage volume on Ceph. OpenStack downloads an operating system for this instance. We have now launched an instance on OpenStack with the AMI containing our web tier code.

The architectural diagram of our system is shown below.

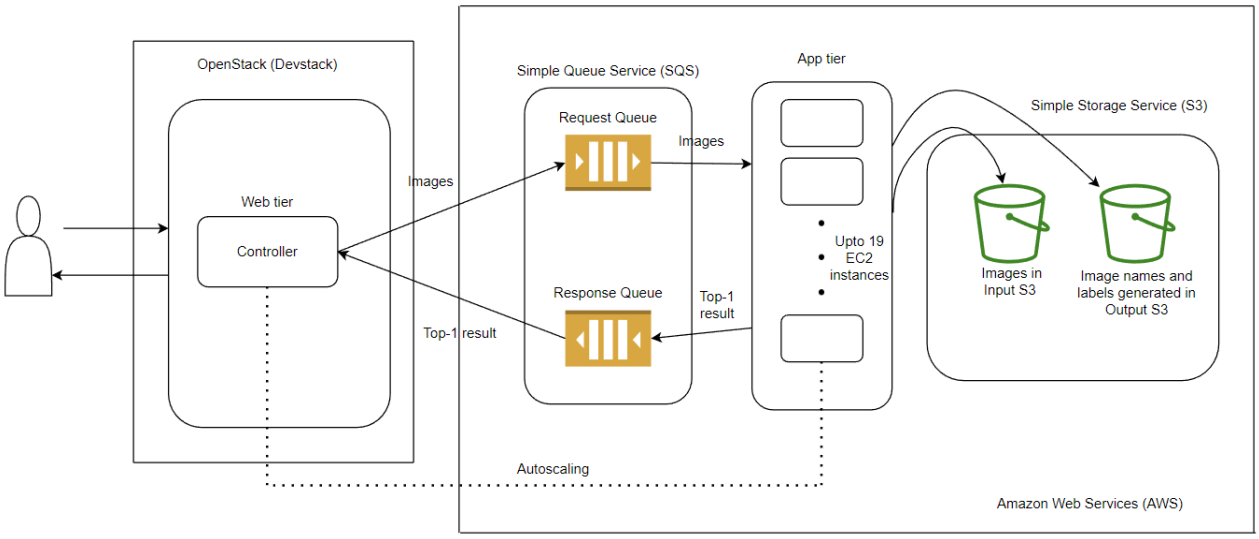


Figure 1: Architectural diagram

We run our web tier, consisting of the controller, on OpenStack (Devstack) which provides compute and storage services to execute the code of the web tier. We launch our web tier on a virtual machine provisioned by OpenStack. The execution of the web tier code leads to spawning of AWS Elastic Compute Cloud (EC2) instances, which execute the app tier code. The number of EC2 instances launched is handled by the controller which is running on the web tier. Our web tier is also responsible for sending image files (JPEG or PNG format) to the request Simple Queue Service (SQS) and receiving labels (image recognition results) generated by the EC2 instances.

The web tier also places the image into the input Simple Storage Service (S3). Owing to the fact that S3 is renowned for its scalability, data availability, security, and performance, we store the input videos and output information in S3 due to the enhanced persistence. We first make a bucket in order to store data

in an S3, and then we upload data in the form of objects (in this case, images) to that bucket. Each object's key serves as its unique identification, and the data is stored as its value. S3 is a popular cloud storage option since it has a 99.99999999999% durability rate, making it highly reliable and secure.

The requests in the SQS must be in text/String format, and thus we convert the JSON file to a base64 encoded string. Once the JSON file is converted to a base64 encoded string, this string, job ID and image name is sent to the request SQS as a single request. A fundamental characteristic of SQS is that when an EC2 instance reads a request from the SQS, it hides this request from all other EC2 instances to avoid duplicate processing efforts. The request is hidden from all other EC2 instances for a visibility timeout period, which is set to the default 30 seconds in the AWS console. The SQS also waits for the EC2 to return a response for a given request and then deletes that request from the request (input) SQS.

The EC2 instances that are created by the web tier running on OpenStack reads a request from the request SQS, and runs the deep learning model provided to us in the project specification. The deep learning model is given to us as an AWS image (ID: ami-0bb1040fdb5a076bc, Name: app-tier, Region: us-east-1). In the project description, we were asked to use resources only from the us-east-1 region for simplified testing.

Each EC2 instance reads a request from the request SQS, decodes the base64 string into a JPEG or PNG file and uses it to recognize the image. This is performed by the deep learning model, the AMI of which was used to create the instance. Each EC2 instance returns the top 1 result from running the deep learning model. This forms the output image label, which is sent through the response SQS and also placed in the output S3 as a key-value pair with the image name being the key and the classification result being the value. The process repeats until all requests in the request SQS are handled, or in other words, the request SQS is empty.

The top-1 result with the corresponding image name is returned (for each image passed to the web tier) to the web tier running on OpenStack and displayed to the user on the dashboard.

2.2 Autoscaling

One of the most important objectives of the system is for the system to be able to handle multiple requests at the same time. Autoscaling is the process of scaling in or out depending on the number of concurrent requests. For example, if the number of requests at a given time is beyond a certain threshold value, scaling out occurs by launching new EC2 instances as part of the app tier. In contrast, when the number of requests reduces below a threshold, the EC2 instances should terminate by themselves such that the minimum number of EC2 instances handle the requests.

In the first project for CSE 546, we were required to create our own autoscaling function for the controller which scales out the application when the number of active requests in the request SQS increases and scales in when the number of active requests in the request SQS decreases. This is a fundamental feature of Infrastructure as a Service (IaaS), since the cloud user has to devise a strategy for autoscaling in this paradigm.

We use the functions and API calls provided by boto3 to use AWS services using Python scripts. For scaling out, we used a mapping strategy of 1 instance per request until the number of requests in the request SQS reaches 19. This means that one instance is spawned for every active request in the request SQS until there are 19 instances. The project specification mentioned that the maximum number of EC2 instances can be 20, so we ensure that this constraint is followed in our autoscaling function.

We track the number of running instances and compute the difference between the maximum number of allowable instances and the number of running instances. This is to avoid terminating each instance and launching instances unnecessarily, when the same instance can be used again. We check if the difference calculated is greater than the number of instances that have been terminated. If yes, then we start those instances which have been stopped and as many instances as the difference between needed instances and number of instances stopped. Else, we start as many instances as those needed from the list of instances that have been terminated.

If the number of active requests in the request SQS is greater than 20, then we check if the number of running instances is less than 19, to avoid crossing this limit. We repeat the same check as above, if the difference between the maximum allowable number of instances and number of instances is greater than the number of instances that have been terminated. If this is the case then we start as many instances as those needed from the list of instances that have been terminated. If not then we start those instances which have been stopped and as many instances as the difference between needed instances and number of instances stopped.

In this manner, we ensure that the number of instances is within the maximum limit and scale in when the number of active requests in the request SQS reduces below 20. This process continues until the number of requests in the request SQS is zero. Once the request SQS is empty, there are no app tier EC2 instances still running.

2.3 Member Tasks

Member 1: Ravindra Aditya Singh

The major task performed by Aditya was to download the Amazon Machine Image (AMI) from the AWS console. This required setting policies, creating a new role for VM import, and an S3 to store the image of the web tier. Aditya was also involved in understanding how to use boto3 with Python for using AWS services. One of his tasks was to design and implement the web tier in project 1, which also involved conversion of images to base64 encoded form and sending requests through the request SQS.

One of his tasks was integrating the various components of the code, testing the working of the system, and documenting the process in the report. Another part of his responsibilities was to link the key pair and security group ID to our project code, and handle authentication of the IAM profile. He was also actively involved in downloading and configuring OpenStack on a virtual machine.

Member 2: Manisha Nandkumar Phadke

The major task performed by Manisha was downloading the AMI imported into S3 and configuring OpenStack. After gauging documentation from AWS and blog articles, she was involved in using the

standard method of importing the EC2 instance as an AMI into an S3 bucket. This involved some debugging as we encountered several errors during the process of importing an EC2 instance as an AMI. These errors were mainly focused on access permissions and security tokens.

The next task was to debug errors in configuring OpenStack to run the web tier code. This involved browsing documentation on the services, called projects, offered by OpenStack and leveraging their unique functionalities.

Her primary task in the first project was to design the logic of the controller. Manisha was also involved in the process of merging the code, testing and debugging, as well as documentation and constructing the architectural diagram. She was also actively involved in downloading and configuring OpenStack on a virtual machine.

Member 3: Sagar Subbaiah Kuppanda Cariappa

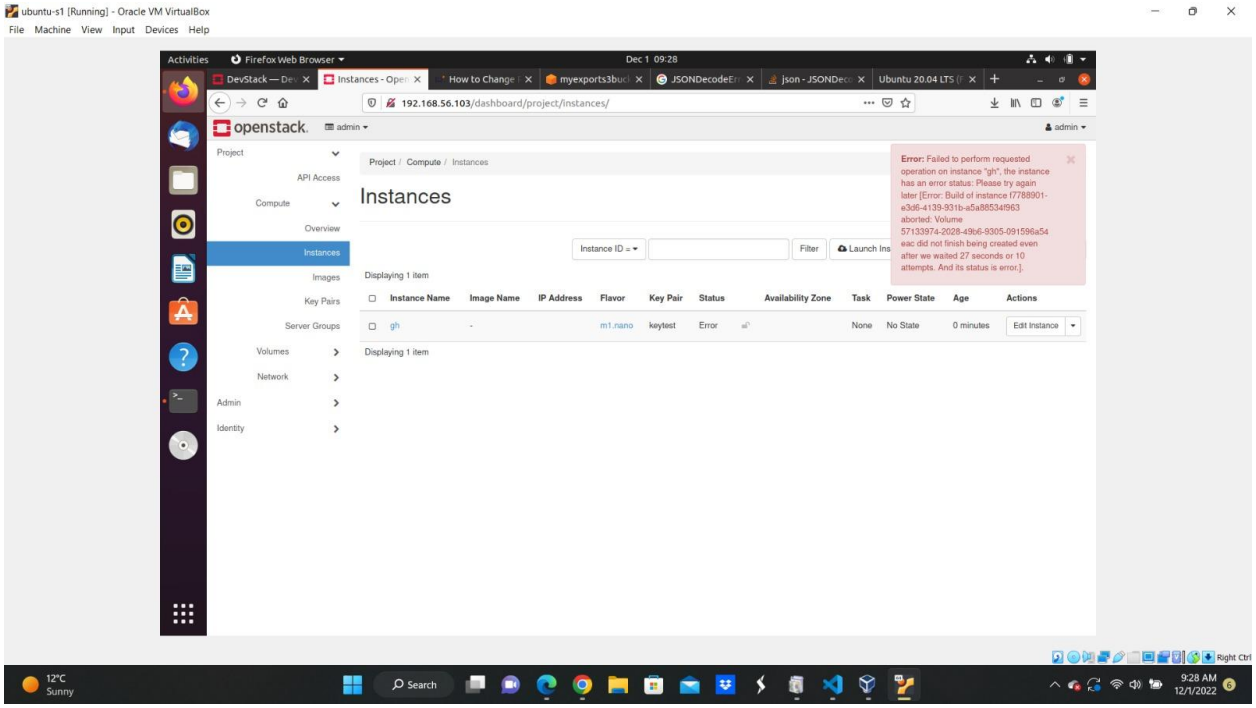
Sagar was assigned the task of setting up the web tier on OpenStack. This involved launching the AMI downloaded onto OpenStack and monitoring the logs to check for errors. Another task was to use the qemu image to create the OpenStack instance and setting up rules for the security group in OpenStack. Sagar was also involved in modifying the web tier code such that the OpenStack instance could execute the web tier code. This required an in-depth understanding of OpenStack and using the OpenStack dashboard.

He was also responsible for creating the final report. One of the crucial tasks handled by Sagar in the first project was designing the app tier. Another one of his tasks was combining all parts of the code to form a single system, testing and debugging.

3. Testing and evaluation

We faced some challenges while configuring the settings for the OpenStack instance. The first error was a TimeOut error, which was displayed on the Horizon Dashboard with the message stating that the build of the instance did not finish even after a wait of 27 seconds (10 attempts). We increased the space allocated to the OpenStack VM and this error was resolved.

Initially, we used VirtualBox to create a virtual Ubuntu machine on our physical machine, and tried to launch an OpenStack instance. This involved creating a security group, creating a router and a network and attaching this router to our newly created network. The next error occurred when we tried to ssh into the instance via the terminal using the *ssh -i keypair ubuntu@<floating-ip-address>* command. The terminal displayed a “Connection refused” error, as shown below.



We tried multiple methods to solve this problem. We began by creating another instance and assigning a new floating IP address to this instance. Next, we monitored the neutron-server and neutron-l2-agent logs to verify that the ports were functioning as expected. We decided to use the qemu image on an Ubuntu machine to create the instance using the private network, rather than creating our own router and network. This debugged the issue and we were successfully able to ssh into the instance.

We faced some authentication errors while downloading the AMI from AWS, which were resolved by activating the security group in the region us-east-1 using the Security Token Service (STS). Due to the elaborate testing strategy used in our first project, we did not face any errors in the web tier code, to spawn AWS EC2 instances according to the number of requests in the AWS SQS request queue.

The image results as returned by the system are as follows:

IMAGE NAME (INPUT TO SYSTEM)	**IMAGE LABEL (OUTPUT FROM SYSTEM)**
test_26	shower curtain
test_45	tub
test_2	jigsaw puzzle
test_75	vault
test_14	menu
test_21	starfish

4. Code

The instructions for starting the system are as follows:

1) Log into the OpenStack Horizon dashboard as demo.

2) SSH into openstack instance via the terminal via the command prompt:

ssh -i <.pem file for ssh> ubuntu@<floating-ip-address>

3) Start the Flask server from the terminal.

4) Run the workload generator via the terminal.

5) The output will be put into the s3 buckets in a few minutes and all AWS EC2 app tiers will shut down.