CSE 546 Group Project 3:
Hybrid Cloud

Pavan Kumar Ramadugu (ASU ID: 1225163134)

Snehal Chaudhari (ASU ID: 1225312140)

Jaydeep Bhoite(ASU ID:1225323242)

Group no. 4

1. Problem statement

The project's objective is to migrate the elastic application we created using the IaaS cloud, which automatically scales up and down in response to demand and efficiently, into a hybrid cloud environment. We'll try to develop this application using both Openstack and AWS resources. In order to scale the application service and deliver results more quickly, we are using OpenStack resources and AWS IaaS resources like AWS S3, Simple Queue Service (SQS), and EC2, among others.

2. Design and implementation

2.1 Architecture

The architecture we used to create a scalar application that scales in and out based on the input task's traffic.

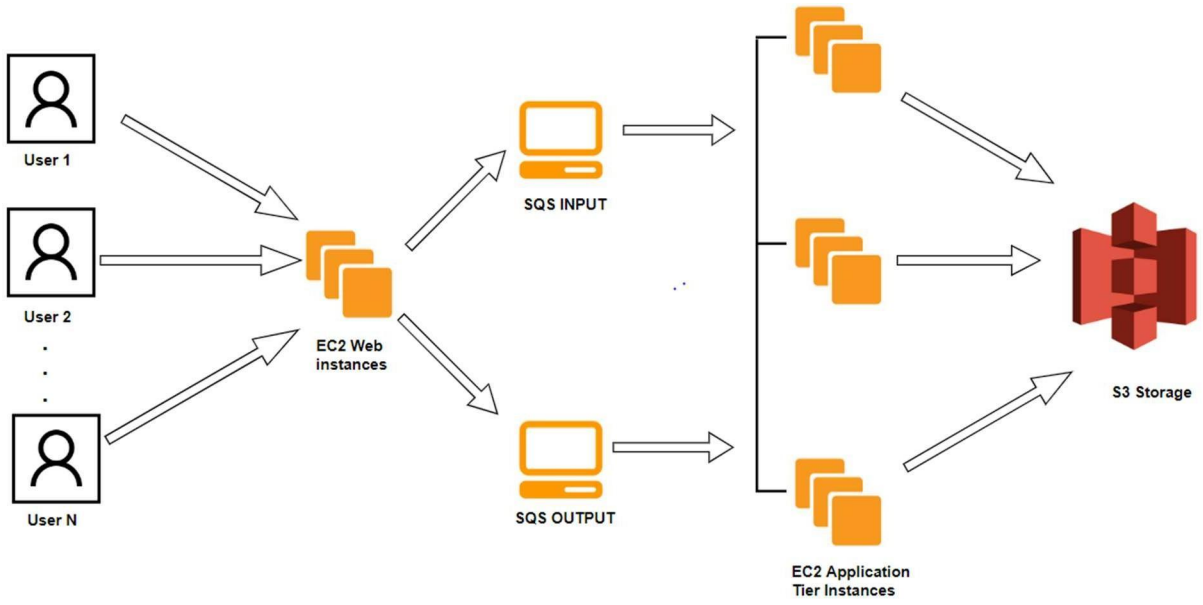


Fig 1.1 Architecture

On our local machine, we set up our private cloud network using Openstack, and we deployed the developed model there.

We developed a system that solely uses EC2 web instances to process user requests. All user requests and inquiries are sent to the SQS input queue, which is then communicated to the application tier. The deep learning application, which uses picture URLs from SQS as inputs and creates labels for the images, requires the setup of app-tier EC2 instances, which is the responsibility of the web tier. Due to the constrained resources in the free tier, we can only have a maximum of 20 EC2 instances; once we've reached this limit, all incoming requests are queued.

When more application tier instances are needed, the load balancing mechanism will decide this. The application tier is also responsible for scaling the number of EC2 instances up and down so that a minimum of 1 instance and a maximum of 20 instances are running at any given moment.

We found that this architecture best suited our needs and provided an overall optimum efficacy for this project.
AWS Services used were:

1. AWS EC2 - The EC2 instance was set up to implement both our application tier and our web tier, which would allow users to send images as needed.

2. AWS SQS - In our design, an inbound and outbound message queuing service is implemented using the SQS. It serves as a buffer for incoming requests and outgoing responses and divides the web layer application from the app tier program.

3. AWS S3 - To store the photographs supplied by users for durability, we created buckets using the AWS S3 service. We utilize AWS S3, an object storage service, to store many types of files since it provides scalability, data availability, security, and performance.

4. Openstack - As a private cloud, we deployed our project using Openstack on our own infrastructure. The IaaS cloud platform OpenStack manages cloud computation, storage, and network resources. It is free and open source. Systems administrators may supply and keep an eye on these resources with the help of the easy dashboard that is included.

Openstack services are:

1. **Nova**: The OpenStack project known as Nova offers a method for creating compute instances, also known as virtual servers, which are used to host and control cloud computing systems.
2. **Cinder**: It is a Block Storage service for OpenStack. End users have access to a self-service API that enables them to request and use those resources without having to know where or what kind of device their storage is actually placed on, and it virtualizes the management of block storage devices.
3. **Horizon**: It offers a web-based graphical user interface that can be used to manage Openstack services like Nova, Swift, Keystone, and others.
4. **Neutron**: OpenStack's Neutron service seeks to offer networking as a service. Users can create network topologies using the APIs that Neutron makes available to them. Neutron is used in the context of our project to enable communication amongst our instances.

2.2 Autoscaling

The web tier and application handle scaling up and scaling down of the EC2 instances. In order to scale the application, the processing architecture has devised a method in which we consider both the number of instances that are now operating and the quantity of unread messages in the SQS queue. As previously noted, we are only able to use 20 compute instances due to the limitations of the free-tier, hence a cap of 20 instances running concurrently has been set. Therefore, there won't be more than 20 instances active at once. We created an AMI that includes the supplied image recognition module.

An essential IAM role is associated with each web-tier and app-tier instance, enabling the instance to send API calls to other services. This enables the tiers to upload and download objects from S3, as well as retrieve and delete SQS messages. S3 and SQS are present between the web tier and the app tier. After being uploaded from the web tier via an application route to a web tier controller with a given endpoint, images are added to an S3 bucket using POST object requests.

Only the web-tier uses EC2 instances, so whenever there are fewer than 19, we create new instances based on the number of pending messages that are currently in the queue. Every time a user sends a new request, the instance adds it to the request queue. The instances automatically terminate after a predetermined period of time if there are no messages in the request queue, which is continuously checked by the app tier code.

Where the application is scaled is at the application tier. A key-value object pair is deposited in the S3 bucket once an application tier instance predicts the output for a certain input request, feeds this output to the SQS response queue, and saves it there. We first finish the one request that was in the SQS before starting to hunt for the message once more. If there are no messages in the queue, the application terminates the scale-out controller and the instance. For 15 seconds, the queue will wait for messages. In conclusion, unless they can locate a request in the SQS, app instances will run continuously.

The controller software in the implemented architecture specifically looks at the quantity of messages in the SQS input queue. The next decision is whether to create fresh instances or restart those that have already been halted. The controller application is given a system service that launches once every 15 seconds. The newly created or restarted app tier EC2 instance would then run its classification engine to build the response and push it into the Response-Queue.

The controller continuously monitors the Request Queue and selects whether to scale back the instances based on the size of the SQS Queue.

We setup our Openstack locally on our Ubuntu 20.04 instance using Devstack. Devstack is a collection of scripts and tools that allow for the rapid deployment of an Openstack cloud from git source trees.

2.3 SETTING UP OPENSTACK:

We installed OpenStack on Ubuntu with the use of DevStack. Devstack is a series of extensible scripts, which is used to set up an OpenStack environment with ease.

1. We need to ensure that our system is updated, for that run following command:
   sudo apt-get update && sudo apt-get upgrade -y
2. Creating stack user with Sudo privileges.
   sudo useradd -s /bin/bash -d /opt/stack -m stack
   echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee /etc/sudoers.d/stack

3. Downloading Devstack
   cd devstack
   vim local.conf

   and paste the following content –

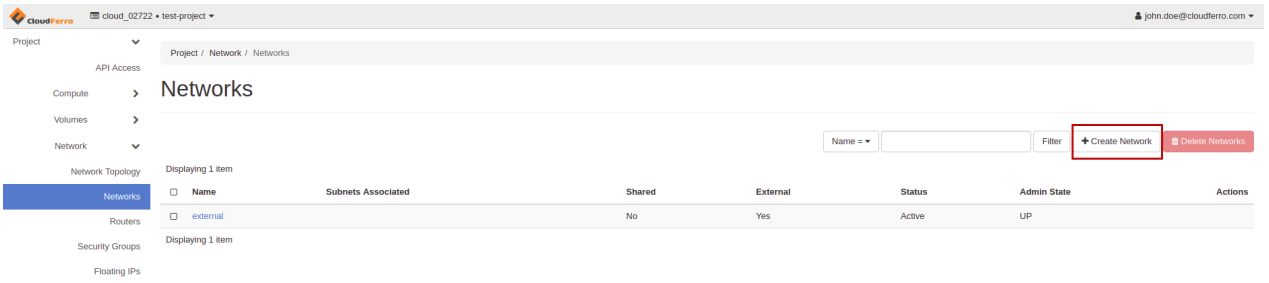
   [[local|localrc]]
   ADMIN_PASSWORD=StrongAdminSecret
   DATABASE_PASSWORD=$ADMIN_PASSWORD
   RABBIT_PASSWORD=$ADMIN_PASSWORD
   SERVICE_PASSWORD=$ADMIN_PASSWORD

4. Installing Openstack with Devstack
   ./stack.sh

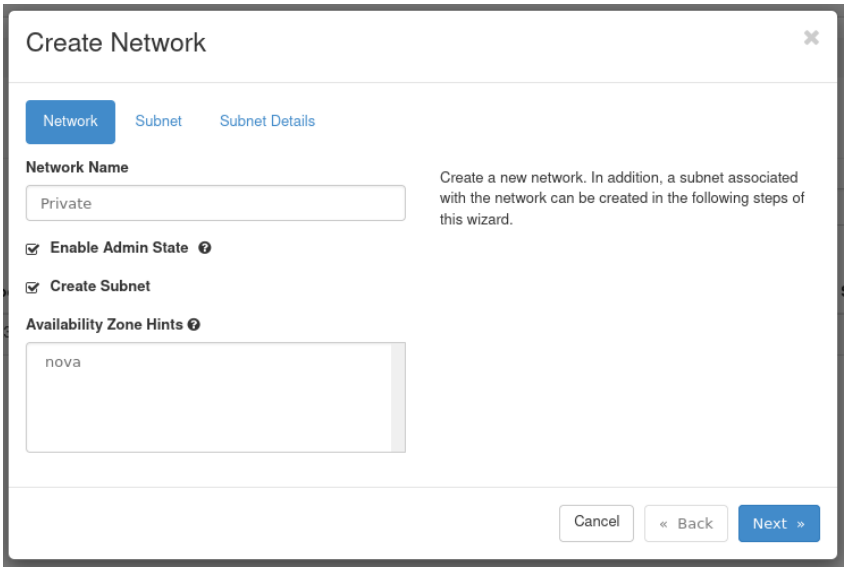
5. Accessing OpenStack using a web browser.
   https://server-ip/dashboard

After Installing Openstack, We need to create an Instance. Following steps are to be done to create an instance.

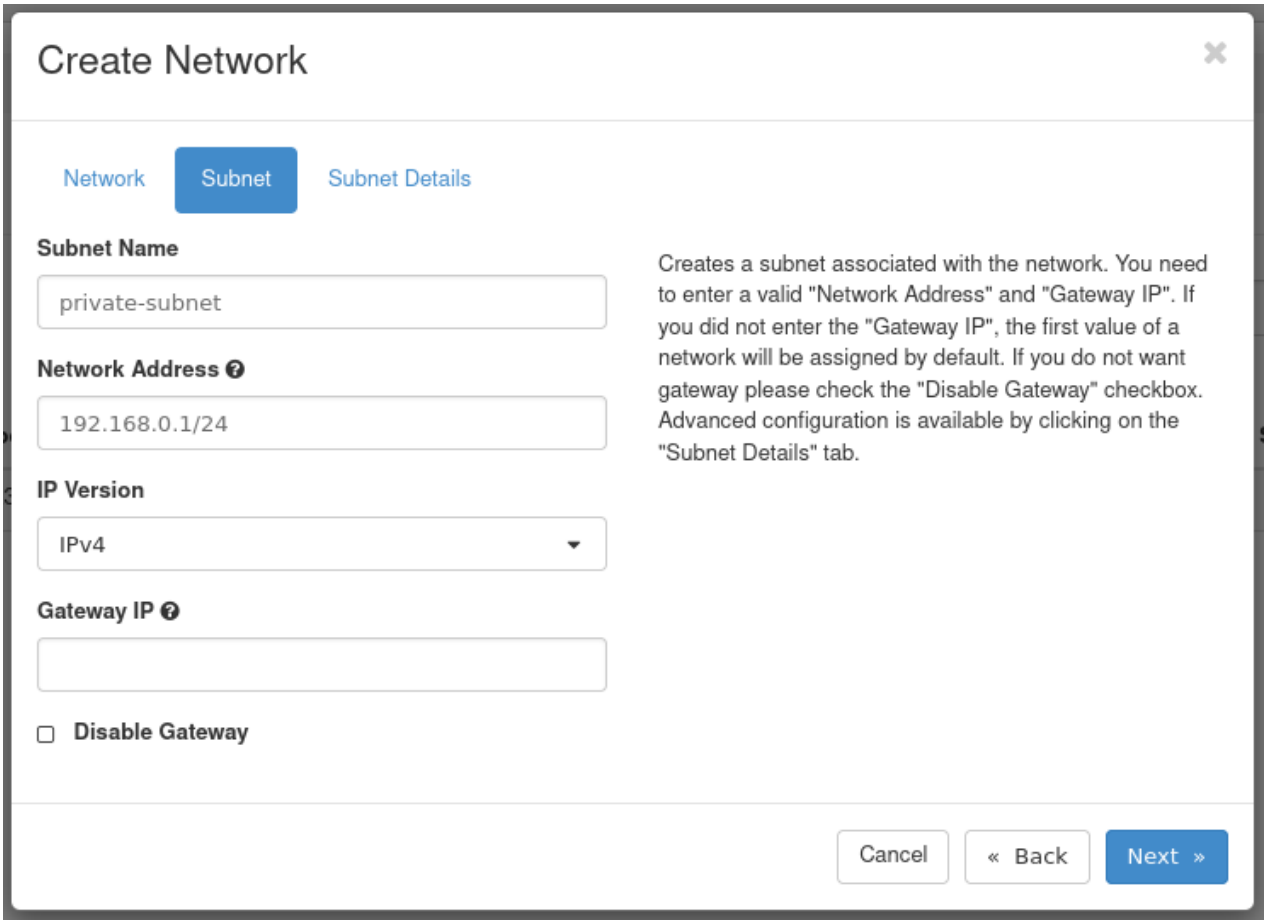
1. Create a Private Network. To create a private network, begin by navigating to Project -> Network -> Networks. Load the form to create a network, by navigating to Create Network near the top right.



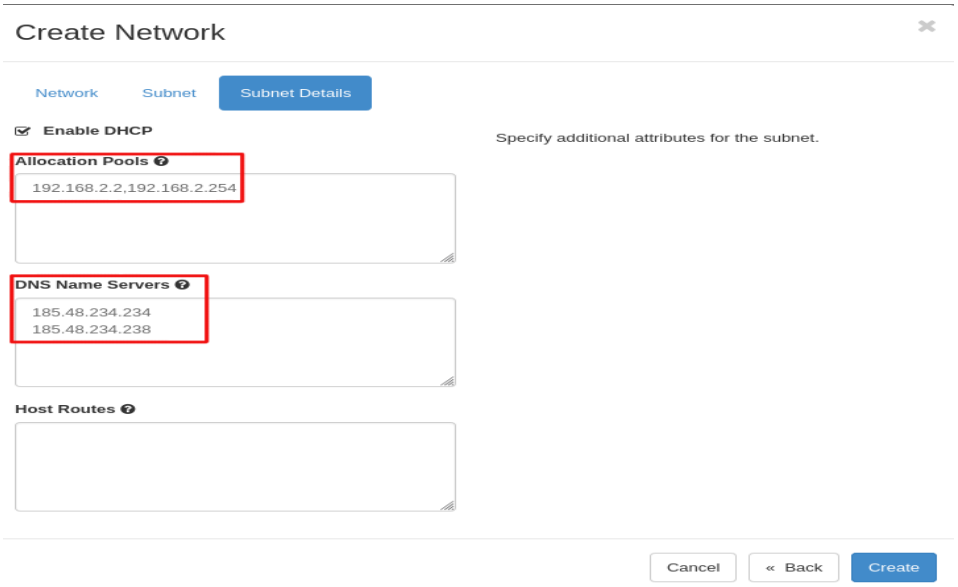
Define your Network Name and tick two checkboxes: Enable Admin State and Create Subnet. Go to Next.



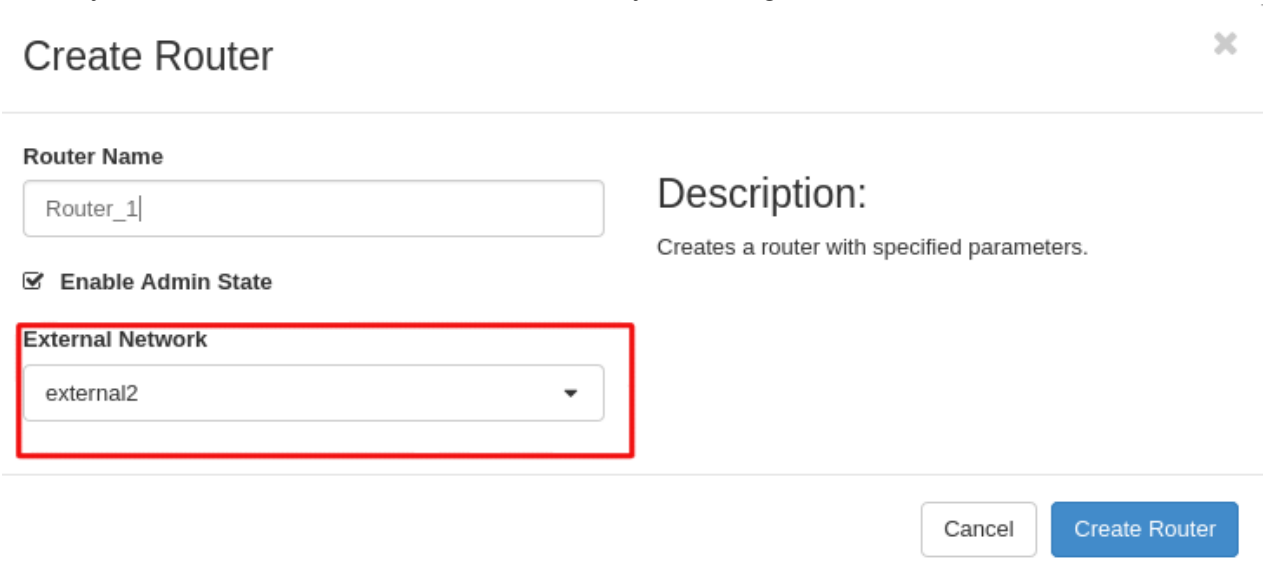
Next, move on to the Subnet tab of this form. Define your Subnet name. Assign a valid network address with a mask presented as a prefix. (This number determines how many bytes are being destined for network address) Define Gateway IP for your Router. Normally it's the first available address in the subnet.



In Subnet Details you are able to turn on DHCP server, assign DNS servers to your network and set up basic routing. In the end, confirm the process with the "Create" button.

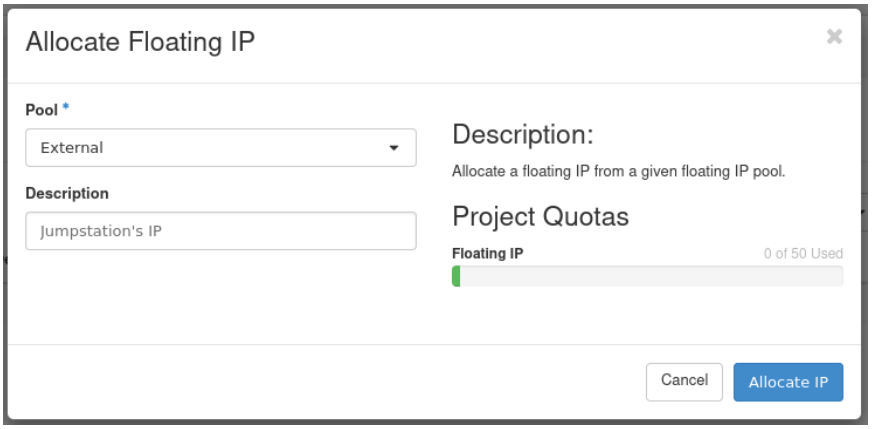


Click on the Routers tab. Click on the “Create Router” button. Name your device and assign the only available network → external. Finish by choosing “Create Router” blue button.

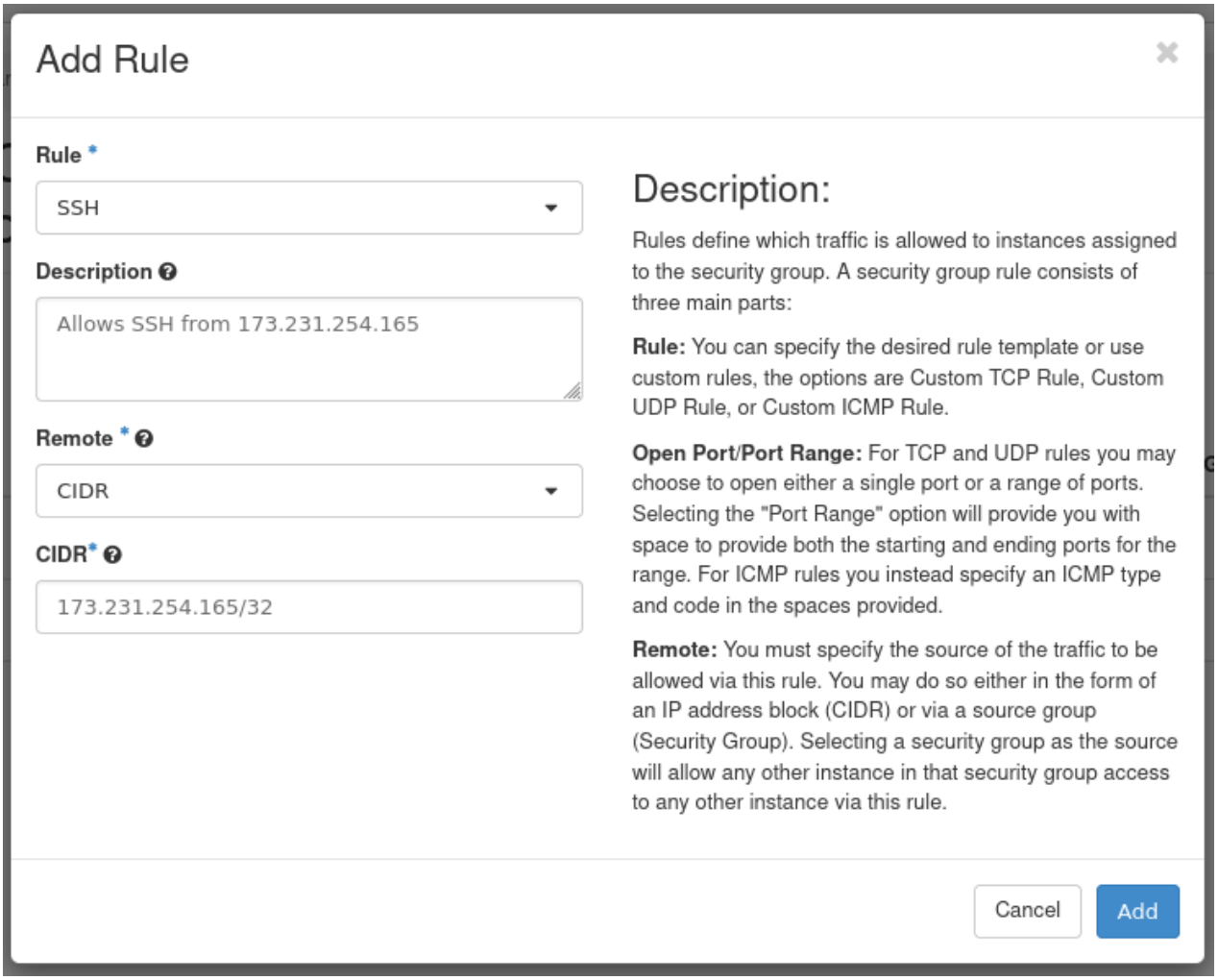


Click on your newly created Router (e.g called “Router_1”). Choose Interfaces. Choose + Add Interface button. Assign a proper subnet and fill in IP Address. (It’s the gateway for our network). Submit the process.

The instance created previously is associated with a private network. Presently, the only way to access this instance is to connect to it from with the cloud's hardware nodes. Another option for connecting is to use a floating IP. Next, load the form to allocate a floating IP by pressing Allocate IP to Project.



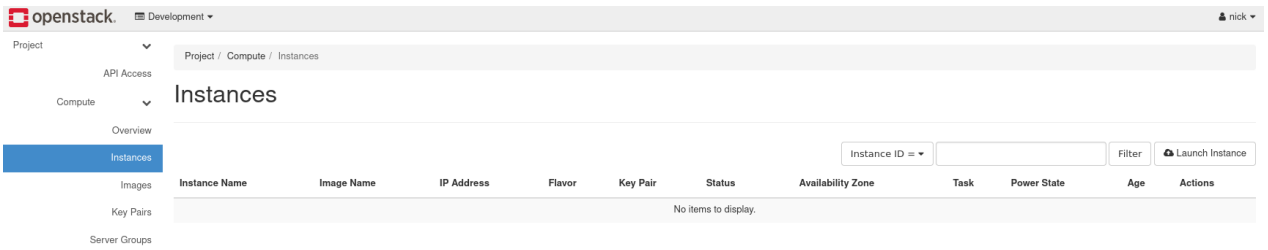
2. Next, We need to create a security group. Security groups allow control of network traffic to and from instances. Allow TCP(5000), ICMP and SSH rules.



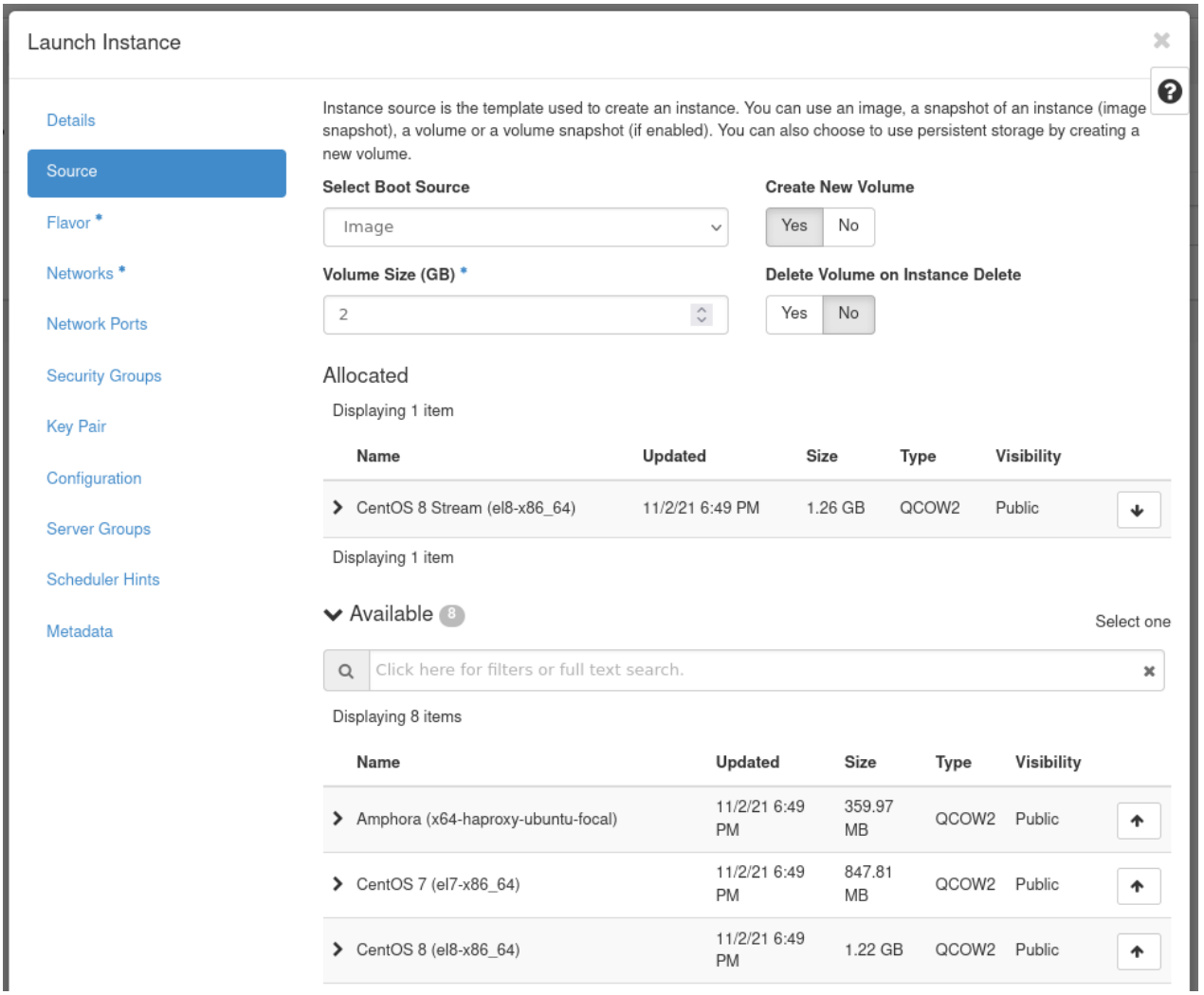
3. We need to specify an SSH public key to inject into the instance. We can create a Key-Pair and save the Private key to connect to the instance.

After Setting up connectivity, we need to import a source image to create the instance. I used Ubuntu 20.04 Focal QCoW2 Cloud image to create an instance.

1. Create Instance, To create the first instance, begin by navigating to Project -> Compute -> Instances. Pull up the form to create an instance by navigating to Launch Instance near the top right.



2. Next, move to the Source tab allowing you to specify an operating system image

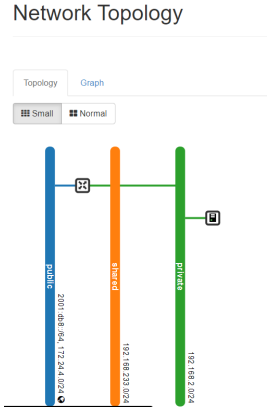


3. Then Select the Ubuntu Image and Move to Flavours.

4. Flavors are a way to define the VCPUs, RAM, and Disk space used by an instance. Pre-built flavors are available for you. For this step, select an appropriate flavor from the options under the Available heading.

5. Then, add the private network, security group and SSH KeyPair created into the instance.

6. Launch the instance with the created configuration.
Using the floating IP, SSH into the instance and Launch the WebTier server



2.4 Member Tasks

2.4.1 Pavan Kumar Ramadugu(Asurite ID: 1225163134):

Design:

My main contribution to the project's architecture was the creation of the web layer of the EC2 instances, which manages image data through the RESTful API and consists of two services to handle interface with the SQS queues and S3 buckets. I worked on the procedures required to implement the web tier, the internal communication between each module present in the web tier, and how the web tier could be designed to guarantee that independent testing of individual models could be carried out and their integration at a later stage.

Implementation:

I was in charge of the web tier's development and implementation. I did the following tasks:

- I worked on setting up openstack in the GCE VM which was used for the development of the project.
- I worked on configuring and setting up the openstack network.
- I worked on entitling the image's identification and logging it as a key-value pair for the SQS input queue, which would subsequently be handled by the script on top of the app layer.
- Uploaded input images in the SQS Response queue were stored.
- Evaluating and using correct values for SQS parameters like Receive Message Wait Time for long polling to SQS.
- Logic to receive messages from the queue and delete it from the request queue when delivered.

Testing:

I divided testing into 3 stages:

- Unit Testing:
  - I ensure that all 100 image requests had been fulfilled and that the number of instances generated did not go over the threshold.
- Integration Testing:
  - All of the requests were filled, according to the output in the logs file, which shows timestamps and the names of each request's images. Finally, entries in the SQS queue and S3 bucket were watched.
  - I created the repository and added all of the required app layer modules to it. I revised the web tier's flow after organizing all the code modules on the web layer.
- End-to-End Testing:
  - We tested the effectiveness of using the SQS request queue to send messages from the web tier to the app tier.

2.4.2 Snehal Chaudhari (Asurite ID: *1225312140*):

Design:

My main effort was to develop clean scripts and the App tier. I was also in charge of the controller service for the application. The quantity of messages in the queue needed to scale with the number of instances that were active at any given time. I worked on the code during the project's integration phase to make sure it could function on its own and communicate with other modules.

Implementation:

I performed the following tasks in the App Tier Module of the Service:

- A POST request to the Web-tier would contain the setup image, and the Web-tier would then add messages to the SQS queue.
- This controller would operate for each cycle, monitor the SQS queue, and launch EC2 instances.
- In accordance with the message queue, the controller is also responsible for terminating the idle EC2 instance.
- Accessibility and keys were necessary to start instances and ping the controller's queue..
- This system operates in an asynchronous manner, with multiple images being initially uploaded to the queue before instances are created following the completion of all uploads to the queue.
- Terminate instance after retrieving instance-id.

Testing:

I conducted testing in three phases: unit testing, integration testing, and end-to-end testing.

Unit Testing:

  - When creating the model, I took into account a variety of circumstances and tested each script on many platforms.
  - I further verified that the SQS queue maintained the queue in accordance with the quantity of outputs delivered and delivered outputs as needed.
- Integration Testing:
  - After putting up each web tier module, I tested it separately to ensure it functioned as intended, and then I tested the entire web tier to ensure the module functioned as expected as a whole.

- End-to-End Testing:
    - I checked to see if the results were accurate or not and if the messages sent from the SQS request queue to the app layer worked as expected.

2.4.3 Jaydeep Bhoite (Asurite ID: 1225323242):

Design:

I mostly made a contribution to the design of a flow where I mapped user inputs to web-tier objects and web-tier objects to app-tier objects. I also collaborated with the rest of the team to create a system for inter-model communication that was required by users. I also contributed to the App tier's architectural design.

Implementation:

- Setting up the necessary AWS resources for the project:
    - Security group in IAM, users in IAM, key-value pairs in EC2, and security groups in EC2.

I performed the following tasks during the setup:

- Putting each image that is downloaded and uploaded in an S3 bucket
- The SQS Response queue was maintained with images from the uploaded input queue.
- I worked on the script to remove the first message along with the finalized image from the Response Queue.
- Take the message that was received out of the response and answer SQS queues, and then store the result in the server code.
- Removing the message from the response SQS Queue and storing the result.

Testing:

I divided testing into 3 stages:

- Unit Testing:
    - On a local server, I independently verified the functionality of each module. As an illustration, I verified that the SQS, S3, and IAM user roles created for AWS resources had the appropriate permissions and functioned as intended.
    - I evaluated the application tier's operation as a whole and confirmed its usability.
- Integration Testing:
    - I uploaded the source code to GitHub and tested each feature to ensure that there are no issues when the server sends requests to different APIs.

- End-to-End Testing:
    - We verified that all the resources posted were in sync and operating as intended after the entire project had been merged.

3. Testing and evaluation

We attempted to test it similarly to how we tested the original project throughout the testing phase of this project, but using Openstack instead (our private cloud network).

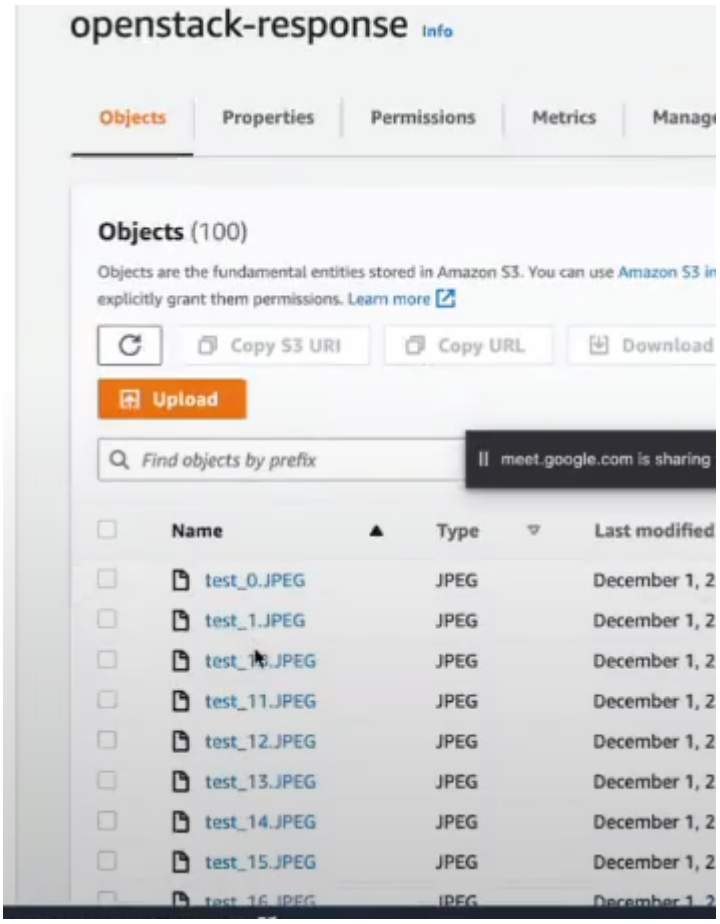
Both integration and end-to-end testing are performed during the testing process. Using the given workload generator, we experimented with a number of instances. During this step, the app instances were subjected to unit and integration testing.

Twenty to thirty images were initially sent up for testing in smaller groups. Our program processed 20–30 images in around 3-4 minutes, as we observed.

Additionally, to the results described above, the following functions were confirmed:

A. We verified the setup of Openstack on our private machine.

B. Used more than 100 images to verify that the identification results were accurate, but took around 8-15 minutes.



C. We put the auto-scaling of several EC2 instances to the test in terms of the number of messages in the queue.

D. Several EC2 instances' auto-scaling was tested using the number of messages in the queue.

E. Monitored the quantity of messages sent to SQS queues and tracked the status of the input, output, and combined input and output queues.
F. Verified the quantity of classifier input results kept on S3 for long-term archival.
G. The number of input photos that were uploaded and stored in S3 buckets was confirmed

4. Code

The complete codebase and required READMEs are available at pavankramadugu/IAAS [01] on Github. The repository is divided into two primary parts. the web server and the application-tier. The web-server directory contains all of the Web layer's source code. A small number of static web pages were used to develop the majority of the Java software. describing the characteristics of various programs

A. Application-Tier:

a. AppTier.java -This program invokes the Image classifier module and is used to initialize the IAAS app tier application.

b. Classifier Module (ImageClassifier.java)- The core driver code for the App layer is included in this class. It has two primary purposes. Initializing the classifier and producing classification results come first. In this, a message is read from the request queue, the results are saved in S3, the already-processed message from the queue is deleted, and the result is returned via the response queue.

c. Utility Module (AWSUtils.java) - The class that offers assistance with S3- and SQS-related tasks. This contains, in particular

i. Setting up S3 and SQS instances initially

ii. Sending, erasing, and reading SQS messages

iii. Objects in S3 can be downloaded and saved.

iv. Terminating EC2 instance Properties - App-tier properties have all configuration details like credentials for SQS queues and S3 buckets

B. Web-Tier :

a. Controller - Endpoints needed for a variety of processes are configured using this module. The following endpoints are on the web controller.
   i. Upload and Process image - Utilizing a provided classifier, upload the image and classify it.
   ii. All results -Get access to all previously cached results that are stored in a map.
   iii. Clear Results - To clear all previous results.

b. Service
   i. Scaling service - Scaling in-out features are included in this load balancer application. If there are more messages than there are active instances and we haven't reached the maximum number, a new EC2 instance is created.

ii. Web-Tier Service - Contains a thorough implementation of every service that endpoints need, including processing uploaded photos, retrieving classifier results, getting results history, and clearing old results.

c. Helper - Contains helper utilities for EC2, SQS, and S3 operations.
   i. EC2 - The EC2 instance's creation, getting the total number of instances after booting an instance.
   ii. S3 - Images Upload.
   iii. SQS - It handles all functionality like publishing, reading, and deleting SQS messages and getting the count of all messages in the queue.

d. Properties - Contains application specific properties like request-response queue, request-response storage buckets, AWS credentials, etc.

e. Static-Web pages - This module includes SPA with web references.

Installation:

1. Clone the repo:
   git clone git@github.com:pavankramadugu/IAAS.git
2. Add AWS account access and secret keys here
   at AppTier: AppTierProperties, WebTier:
   Config.py
3. Build the both tiers (i.e Web and App tier) by clean building using the gradle package manager:
   cd AppTier
   ./gradlew clean build
4. Build the WebTier Project by clean building using the pip package manager:
   cd WebTier
   pip install -r requirements.txt
5. To start the flask server use the following command
   nohup python3 -m flask run > output.log &
6. Monitor the server logs using , tail -f output.log

References

01. Github Repository- pavankramadugu/IAAS

    URL: https://github.com/pavankramadugu/IAAS