VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CSE4011
Virtualization

Project Report
Load Balancing in Openstack

Prof Dr Priyaadharshini M

Slot: A1 + TA1

Team Members

Aneesh Prabu (16BCE1037)

Joel Raymann (16BCE1314)

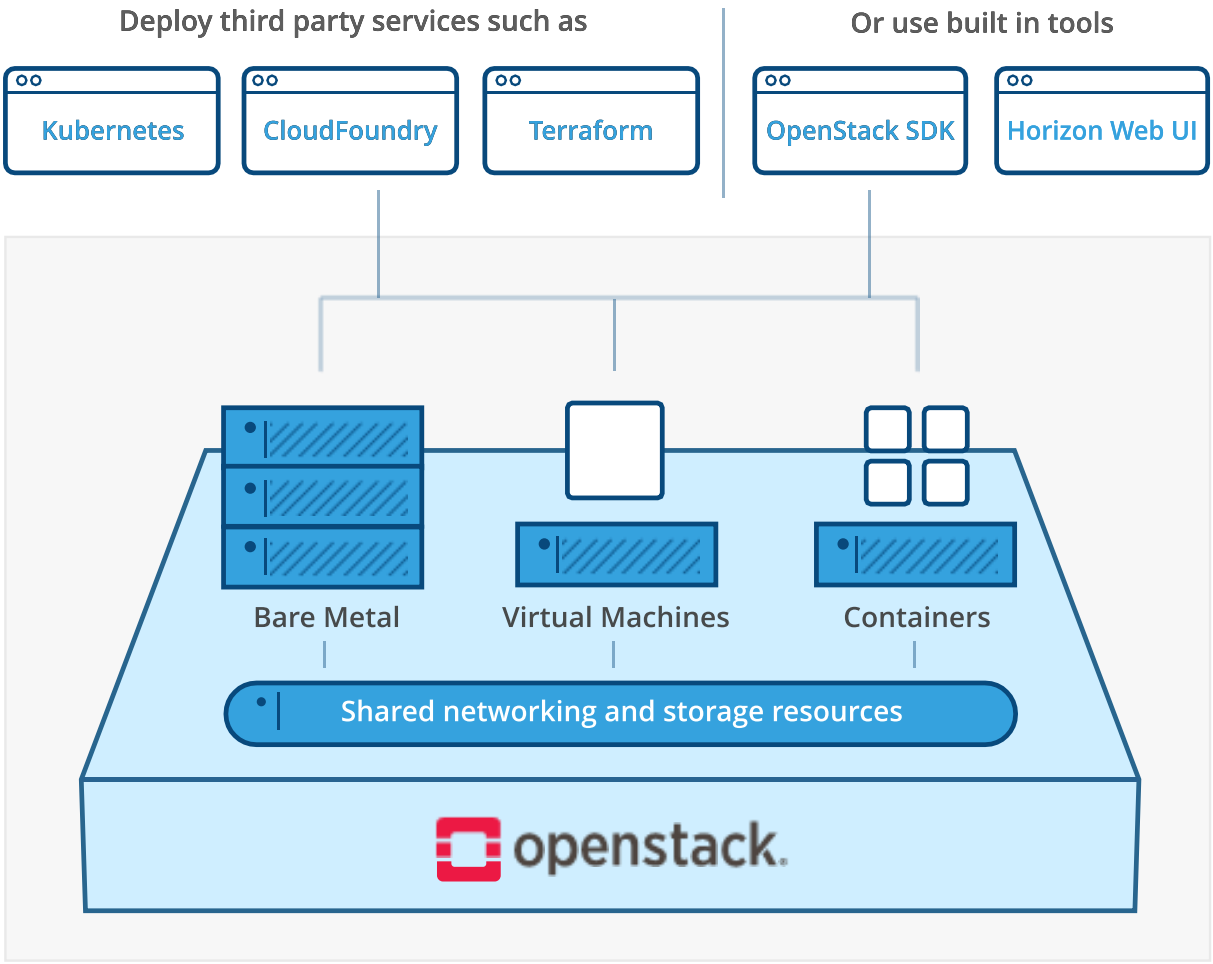
Ramprasath A (17BCE1098)

Project Video Link

https://drive.google.com/file/d/1dAHGHd2RzuJSKd2ArBM77IkkRQ8mBVmJ/view?usp=sharing

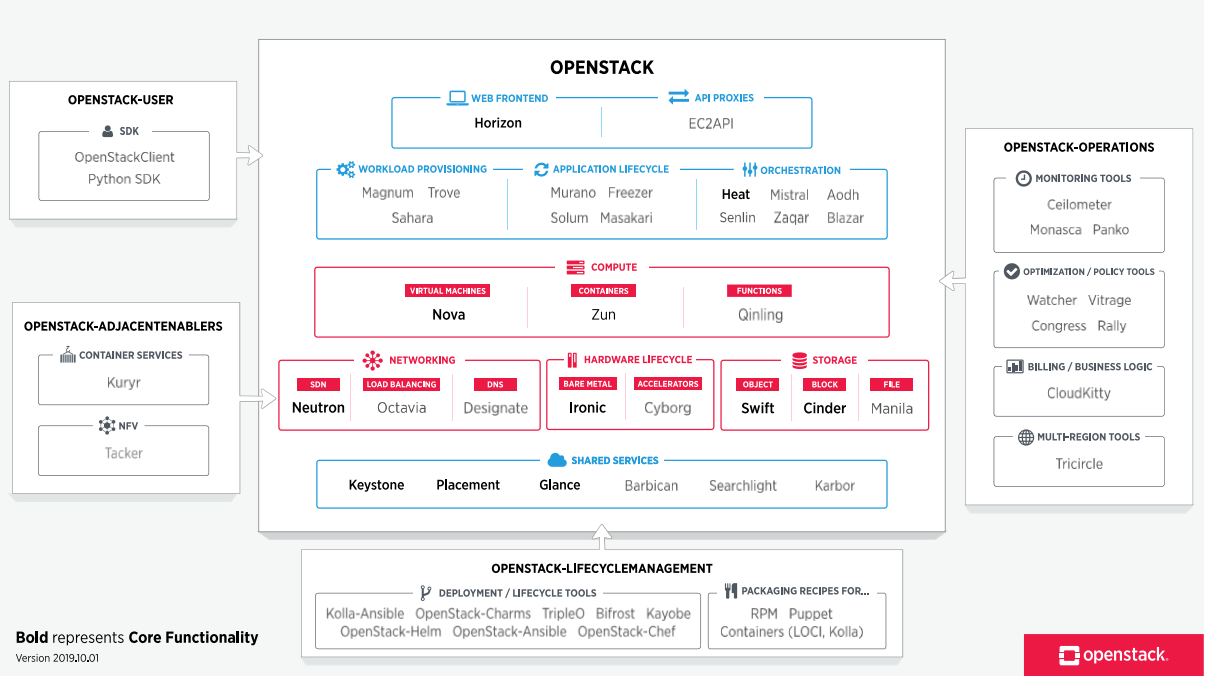
WHAT IS OPENSTACK?

OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed and provisioned through APIs with common authentication mechanisms. A dashboard is also available, giving administrators control while empowering their users to provision resources through a web interface. Beyond standard infrastructure-as-a-service functionality, additional components provide orchestration, fault management and service management amongst other services to ensure high availability of user applications.



THE OPENSTACK LANDSCAPE

OpenStack is broken up into services to allow you to plug and play components depending on your needs. The openstack map gives you an “at a glance” view of the openstack landscape to see where those services fit and how they can work together.



OPENSTACK SERVICES

An OpenStack deployment contains a number of components providing APIs to access infrastructure resources. This page lists the various services that can be deployed to provide such resources to cloud end users.

COMPUTE SERVICES

1. Nova: Compute Service

Nova is the OpenStack project that provides a way to provision compute instances (aka virtual servers). Nova supports creating virtual machines, baremetal servers (through the use of ironic), and has limited support for system containers. Nova runs as a set of daemons on top of existing Linux servers to provide that service.

It requires the following additional OpenStack services for basic function:

- Keystone: This provides identity and authentication for all OpenStack services.
- Glance: This provides the compute image repository. All compute instances launch from glance images.
- Neutron: This is responsible for provisioning the virtual or physical networks that compute instances connect to on boot.
- Placement: This is responsible for tracking inventory of resources available in a cloud and assisting in choosing which provider of those resources will be used when creating a virtual machine.

It can also integrate with other services to include: persistent block storage, encrypted disks, and baremetal compute instances.

STORAGE

1. Swift is a highly available, distributed, eventually consistent object/blob store. Organizations can use Swift to store lots of data efficiently, safely, and cheaply.

2. What is Cinder?

Cinder is the OpenStack Block Storage service for providing volumes to Nova virtual machines, Ironic bare metal hosts, containers and more. Some of the goals of Cinder are to be/have:

- Component based architecture: Quickly add new behaviors
- Highly available: Scale to very serious workloads
- Fault-Tolerant: Isolated processes avoid cascading failures
- Recoverable: Failures should be easy to diagnose, debug, and rectify
- Open Standards: Be a reference implementation for a community-driven api

3. What is Manila?

Manila is the OpenStack Shared Filesystems service for providing Shared Filesystems as a service. Some of the goals of Manila are to be/have:

Component based architecture: Quickly add new behaviors

- Highly available: Scale to very serious workloads
- Fault-Tolerant: Isolated processes avoid cascading failures
- Recoverable: Failures should be easy to diagnose, debug, and rectify
- Open Standards: Be a reference implementation for a community-driven api

NETWORKING

1. Neutron is an OpenStack project to provide “network connectivity as a service” between interface devices (e.g., vNICs) managed by other OpenStack services (e.g., nova). It implements the OpenStack Networking API.

2. Octavia is an open source, operator-scale load balancing solution designed to work with OpenStack.

Octavia was born out of the Neutron LBaaS project. Its conception influenced the transformation of the Neutron LBaaS project, as Neutron LBaaS moved from version 1 to version 2. Starting with the Liberty release of OpenStack, Octavia has become the reference implementation for Neutron LBaaS version 2.

Octavia accomplishes its delivery of load balancing services by managing a fleet of virtual machines, containers, or bare metal servers—collectively known as amphorae— which it spins up on demand. This on-demand, horizontal scaling feature differentiates Octavia from other load balancing solutions, thereby making Octavia truly suited “for the cloud.”

Where Octavia fits into the OpenStack ecosystem?

Load balancing is essential for enabling simple or automatic delivery scaling and availability. In turn, application delivery scaling and availability must be considered vital features of any cloud. Together, these facts imply that load balancing is a vital feature of any cloud.

Therefore, we consider Octavia to be as essential as Nova, Neutron, Glance or any other “core” project that enables the essential features of a modern OpenStack cloud.

In accomplishing its role, Octavia makes use of other OpenStack projects:

- Nova - For managing amphora lifecycle and spinning up compute resources on demand.
- Neutron - For network connectivity between amphorae, tenant environments, and external networks.
- Barbican - For managing TLS certificates and credentials, when TLS session termination is configured on the amphorae.
- Keystone - For authentication against the Octavia API, and for Octavia to authenticate with other OpenStack projects.
- Glance - For storing the amphora virtual machine image.
- Oslo - For communication between Octavia controller components, making Octavia work within the standard OpenStack framework and review system, and project code structure.
- Taskflow - Is technically part of Oslo; however, Octavia makes extensive use of this job flow system when orchestrating back-end service configuration and management.

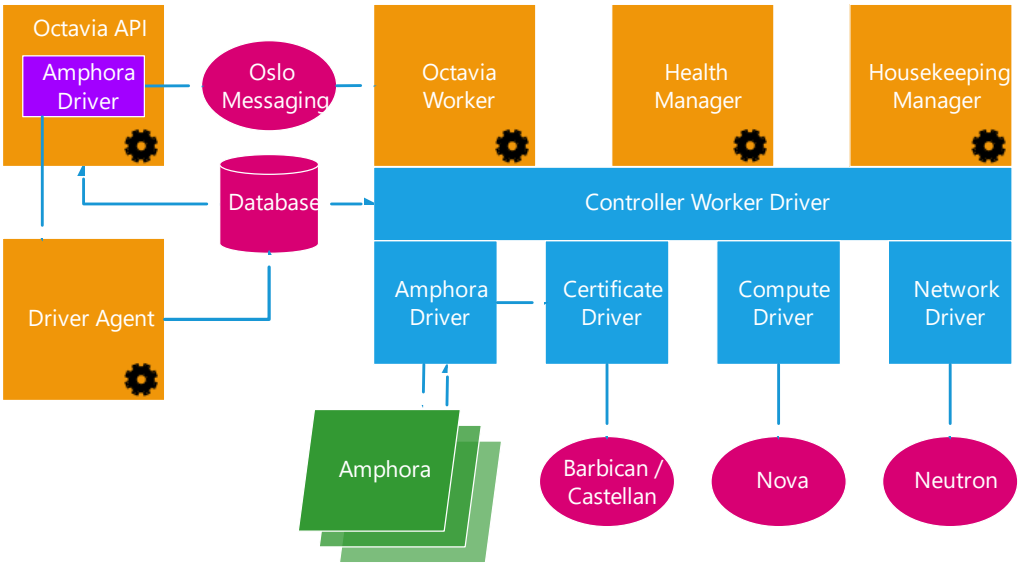
Octavia is designed to interact with the components listed previously. In each case, we've taken care to define this interaction through a driver interface. That way, external components can be swapped out with functionally-equivalent replacements— without having to restructure major components of Octavia. For example, if you use an SDN solution other than Neutron in your environment, it should be possible for you to write an Octavia networking driver for your SDN environment, which can be a drop-in replacement for the standard Neutron networking driver in Octavia.

As of Pike, it is recommended to run Octavia as a standalone load balancing solution. Neutron LBaaS is deprecated in the Queens release, and Octavia is its replacement. Whenever possible, operators are strongly advised to migrate to Octavia. For end-users, this transition should be relatively seamless, because Octavia supports the Neutron LBaaS v2 API and it has a similar CLI interface. Alternatively, if end-users cannot migrate on their side in the forseable future, operators could enable the experimental Octavia proxy plugin in Neutron LBaaS.

It is also possible to use Octavia as a Neutron LBaaS plugin, in the same way as any other vendor. You can think of Octavia as an "open source vendor" for Neutron LBaaS.

Octavia supports third-party vendor drivers just like Neutron LBaaS, and fully replaces Neutron LBaaS as the load balancing solution for OpenStack.

A 10,000-foot overview of Octavia components

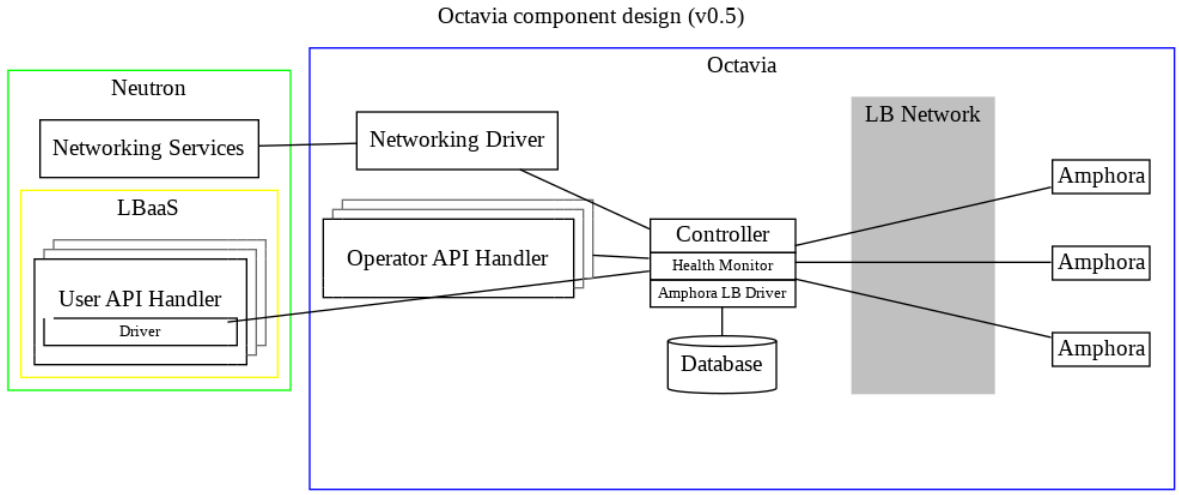


Octavia version 4.0 consists of the following major components:

- amphorae - Amphorae are the individual virtual machines, containers, or bare metal servers that accomplish the delivery of load balancing services to tenant application environments. In Octavia version 0.8, the reference implementation of the amphorae image is an Ubuntu virtual machine running HAProxy.
- controller - The Controller is the “brains” of Octavia. It consists of five sub-components, which are individual daemons. They can be run on separate back-end infrastructure if desired:

    - API Controller - As the name implies, this subcomponent runs Octavia’s API. It takes API requests, performs simple sanitizing on them, and ships them off to the controller worker over the Oslo messaging bus.

    - Controller Worker - This subcomponent takes sanitized API commands from the API controller and performs the actions necessary to fulfill the API request.
    - Health Manager - This subcomponent monitors individual amphorae to ensure they are up and running, and otherwise healthy. It also handles failover events if amphorae fail unexpectedly.
    - Housekeeping Manager - This subcomponent cleans up stale (deleted) database records, manages the spares pool, and manages amphora certificate rotation.
    - Driver Agent - The driver agent receives status and statistics updates from provider drivers.
- network - Octavia cannot accomplish what it does without manipulating the network environment. Amphorae are spun up with a network interface on the "load balancer network," and they may also plug directly into tenant networks to reach back-end pool members, depending on how any given load balancing service is deployed by the tenant.



This milestone release of Octavia concentrates on making the service delivery scalable (though individual listeners are not horizontally scalable at this stage), getting API and other interfaces between major components correct, without worrying about making the command and control layer scalable.

SHARED SERVICES

1. Keystone: Identity Service

   Keystone is an OpenStack service that provides API client authentication, service discovery, and distributed multi-tenant authorization by implementing OpenStack's Identity API. It supports LDAP, OAuth, OpenID Connect, SAML and SQL.

2. Glance: Image Service

   Glance image services include discovering, registering, and retrieving virtual machine images. Glance has a RESTful API that allows querying of VM image metadata as well as retrieval of the actual image. VM images made available through Glance can be stored in a variety of locations from simple filesystems to object-storage systems like the OpenStack Swift project.

WEB FRONTEND

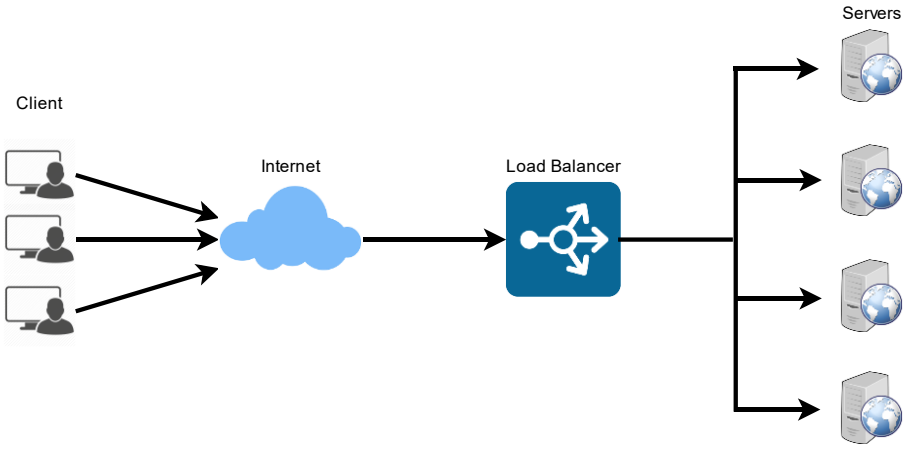
Horizon: Dashboard

Horizon is the canonical implementation of OpenStack's dashboard, which is extensible and provides a web-based user interface to OpenStack services.

WHAT IS LOAD BALANCING?

Load balancing refers to efficiently distributing incoming network traffic across a group of backend servers, also known as a server farm or server pool.

Modern high-traffic websites must serve hundreds of thousands, if not millions, of concurrent requests from users or clients and return the correct text, images, video, or application data, all in a fast and reliable manner. To cost-effectively scale to meet these high volumes, modern computing best practice generally requires adding more servers.



A load balancer acts as the “traffic cop” sitting in front of your servers and routing client requests across all servers capable of fulfilling those requests in a manner that maximizes speed and capacity utilization and ensures that no one server is overworked, which could degrade performance. If a single server goes down, the load balancer redirects traffic to the remaining online servers. When a new server is added to the server group, the load balancer automatically starts to send requests to it.

In this manner, a load balancer performs the following functions:

- Distributes client requests or network load efficiently across multiple servers
- Ensures high availability and reliability by sending requests only to servers that are online
- Provides the flexibility to add or subtract servers as demand dictates

Load Balancing Algorithms

Different load balancing algorithms provide different benefits; the choice of load balancing method depends on your needs:

- Round Robin – Requests are distributed across the group of servers sequentially.
- Least Connections – A new request is sent to the server with the fewest current connections to clients. The relative computing capacity of each server is factored into determining which one has the least connections.
- IP Hash – The IP address of the client is used to determine which server receives the request.

Hardware vs. Software Load Balancing

Load balancers typically come in two flavors: hardware-based and software-based. Vendors of hardware-based solutions load proprietary software onto the machine they provide, which often uses specialized processors. To cope with increasing traffic at your website, you have to buy more or bigger machines from the vendor. Software solutions generally run on commodity hardware, making them less expensive and more flexible. You can install the software on the hardware of your choice or in cloud environments like AWS EC2.

Benefits of Load Balancing

- Users experience faster, uninterrupted service. Users won't have to wait for a single struggling server to finish its previous tasks. Instead, their requests are immediately passed on to a more readily available resource.
- Service providers experience less downtime and higher throughput. Even a full server failure won't affect the end user experience as the load balancer will simply route around it to a healthy server.
- Load balancing makes it easier for system administrators to handle incoming requests while decreasing wait time for users.
- Smart load balancers provide benefits like predictive analytics that determine traffic bottlenecks before they happen. As a result, the smart load balancer gives an organization actionable insight. These are key to automation and can help drive business decisions.
- System administrators experience fewer failed or stressed components. Instead of a single device performing a lot of work, load balancing has several devices perform a little bit of work.

Load Balancing in Openstack: setup & config. Instructions

Phase 1: Create DevStack + 2 nova instances

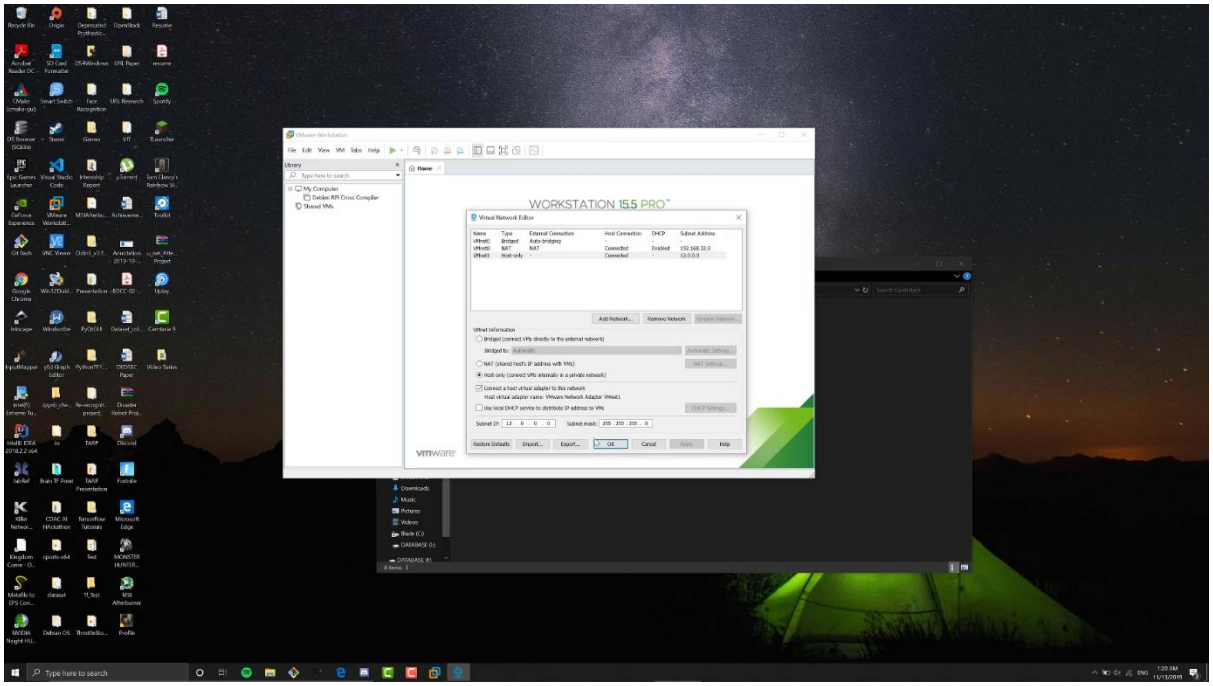
This tutorial deals with setting up openstack in commodity hardware. We are going to setup openstack using devstack, kindly follow only the instructions in this tutorial. In this tutorial we are going to setup openstack train version.

We have to create two Virtual Machines in which we are going to setup the openstack controller and a compute node which is only for the resource purpose. In this tutorial, host OS is Windows 10 and VMware Workstation Pro is used for virtualization but any other free or open-source tools like VMware Workstation Player or Oracle Virtual Box can also be used. In this tutorial we are going to use Ubuntu Server 18.04 LTS as the guest OS for our two VMs. The configuration for the two VMs are

	Storage size	RAM size
Controller VM	30 GB	8 GB
Compute VM	20 GB	4 GB

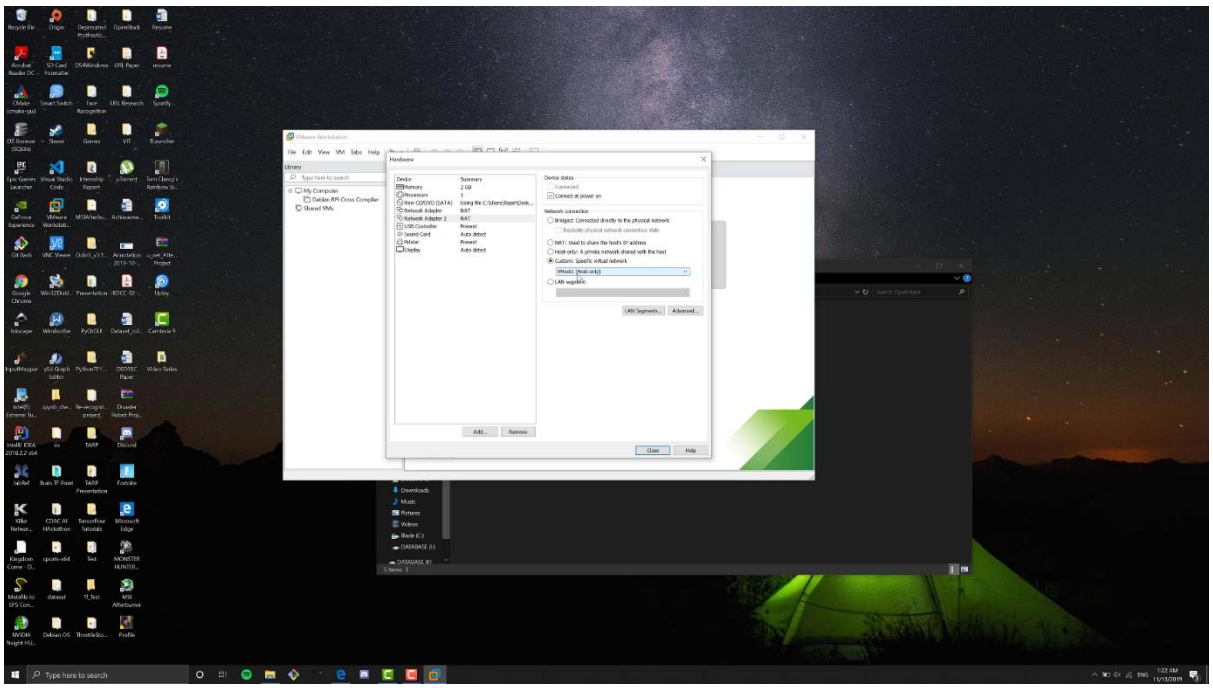
Now, before turning on the VMs, the two VMs must be able to communicate with each other and the host OS. The host network should not interfere with the network the VMs. Openstack works properly in a static IP, an own network with own set of subnets and gateway rules must be setup. Then network of the VMs should be connected to the internet so as to update the guest OS and be able to install packages. In the Virtual Network Editor make the following changes to the VM network:

1. Subnet IP: 11.0.0.0
2. Uncheck local DHCP
3. Select host only network over NAT

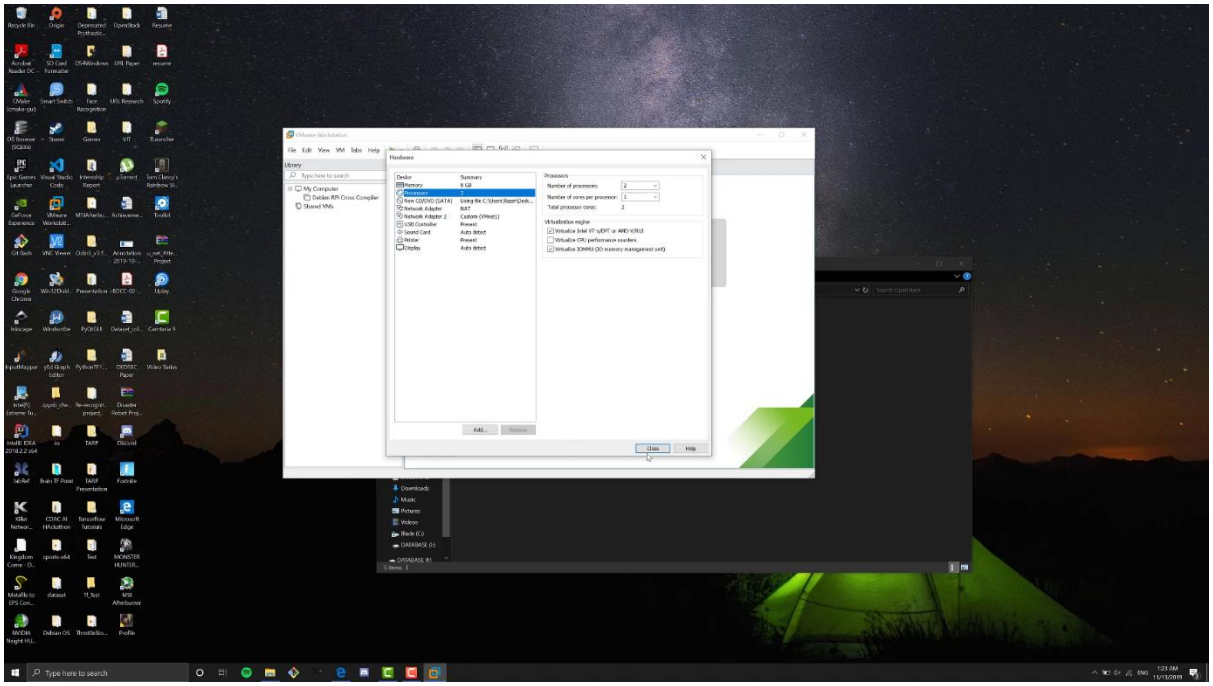


Once two VMs are successfully created before turning on certain configurations are to be made in the respective VMs' settings and they are

1. In the network section of the VM settings, for the second network adapter choose the custom virtual network that we tweaked in Virtual Network Editor.

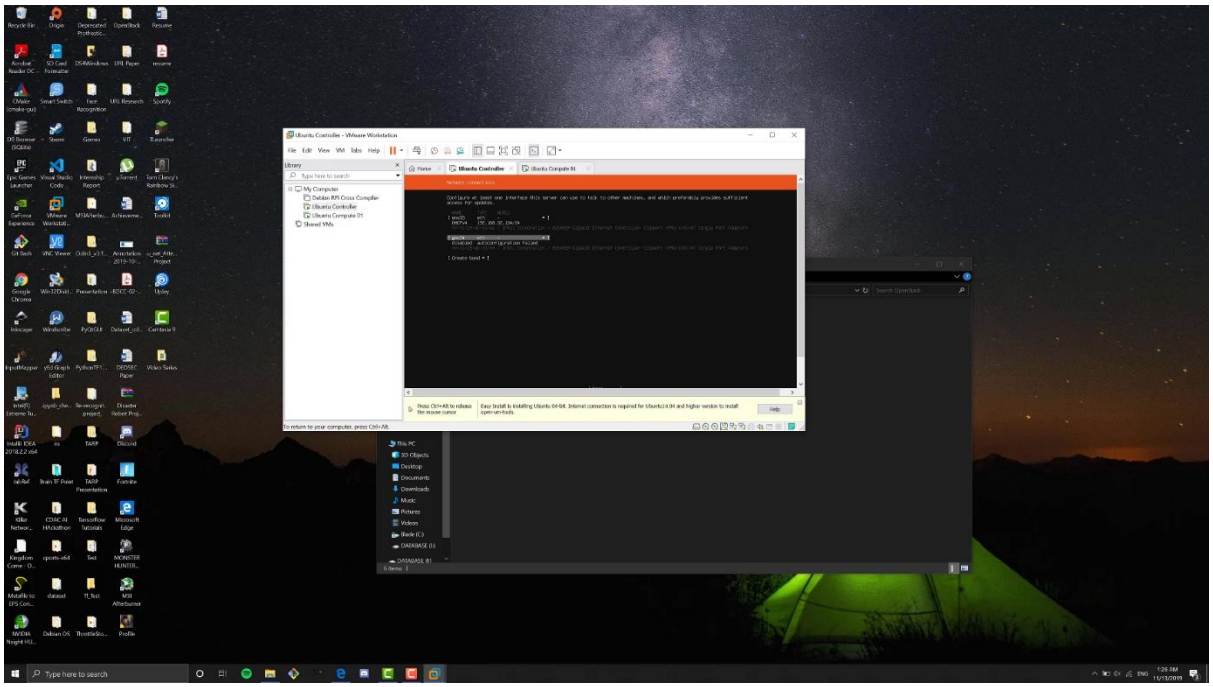


2. Make sure to enable hardware virtualization in your PC.

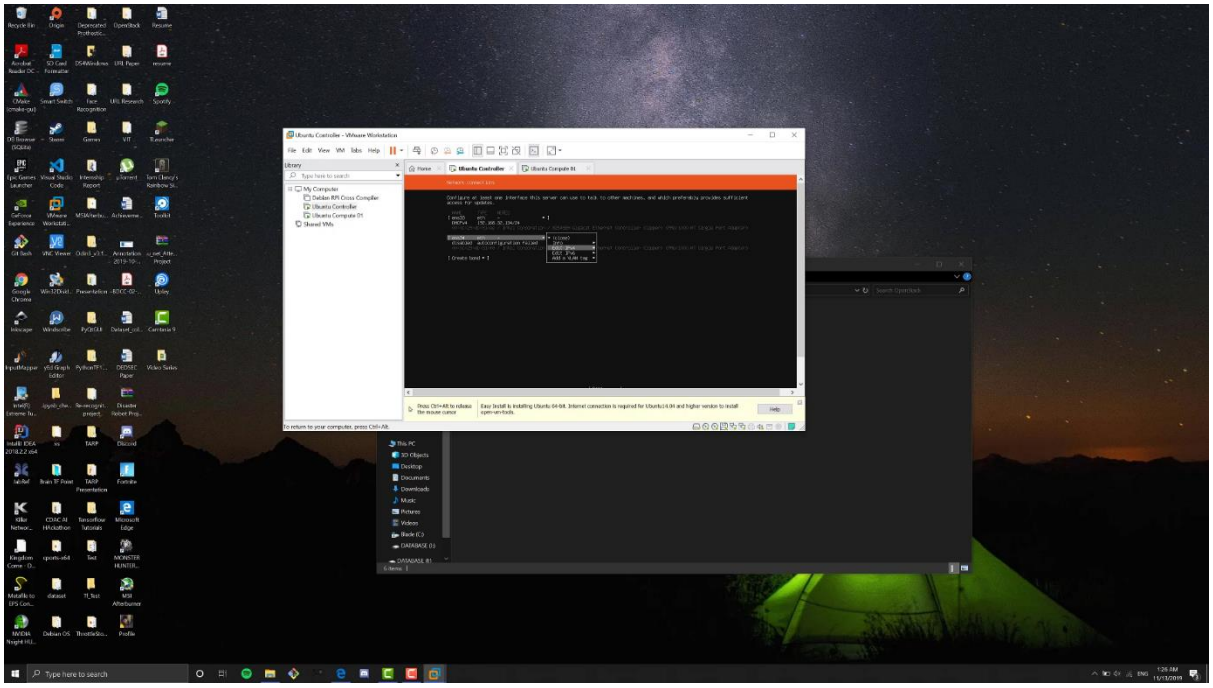


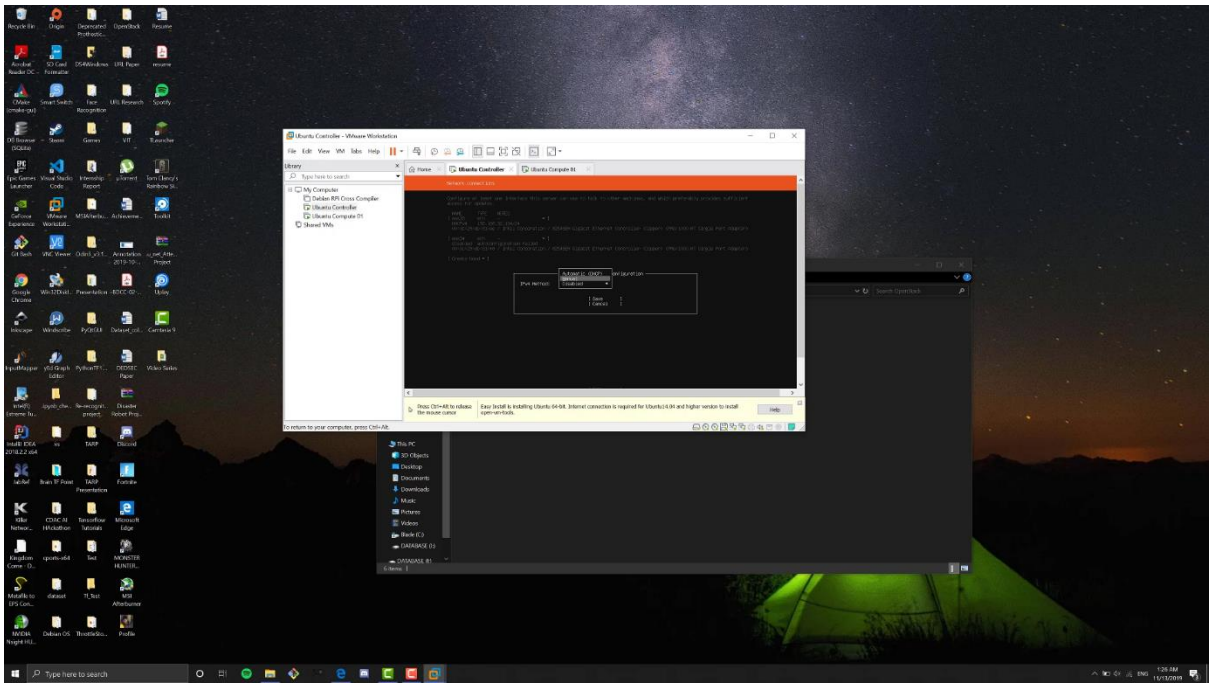
Now turn on the controller node, after booting up the installation menu of Ubuntu Server will be displayed in that menu choose your preferred language, choose your respective keyboard configuration and network connections make sure to do the following changes:

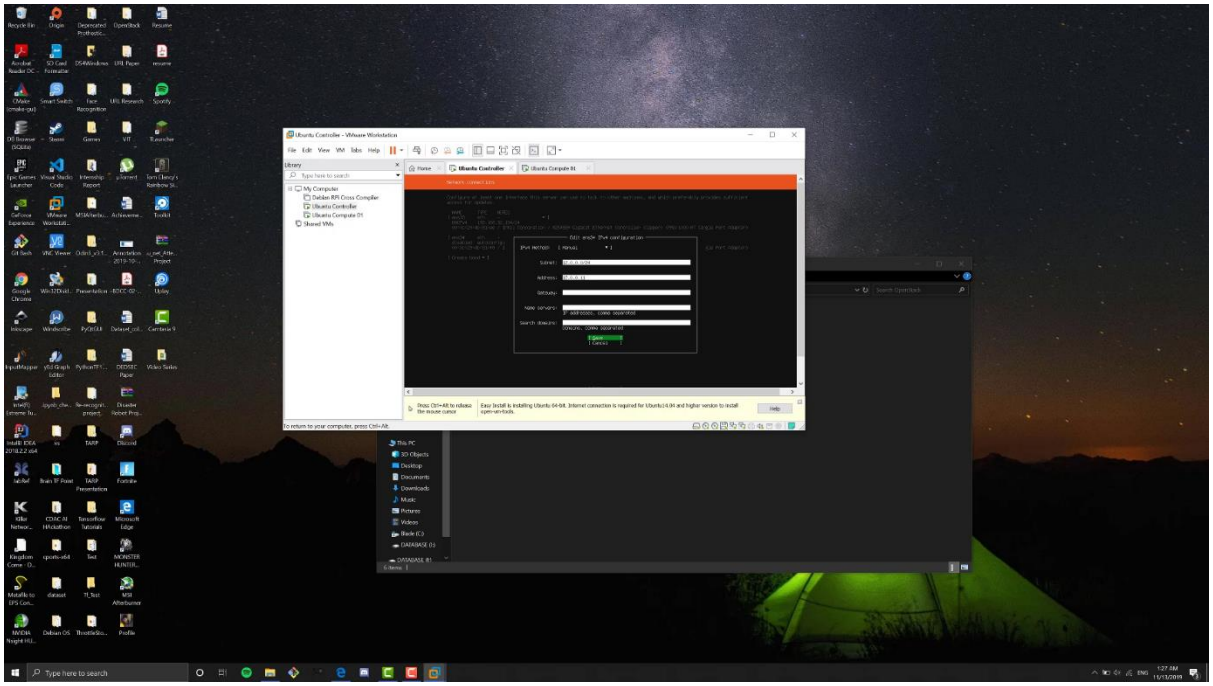
1. Choose the ens34 disabled autoconfiguration failed interface i.e. dhcp is turned off for this particular adapter and we have to manually configure it.



2. Select edit IPv4 -> Select manual mode ->
   Subnet: 12.0.0.4/24, Address: 12.0.0.11



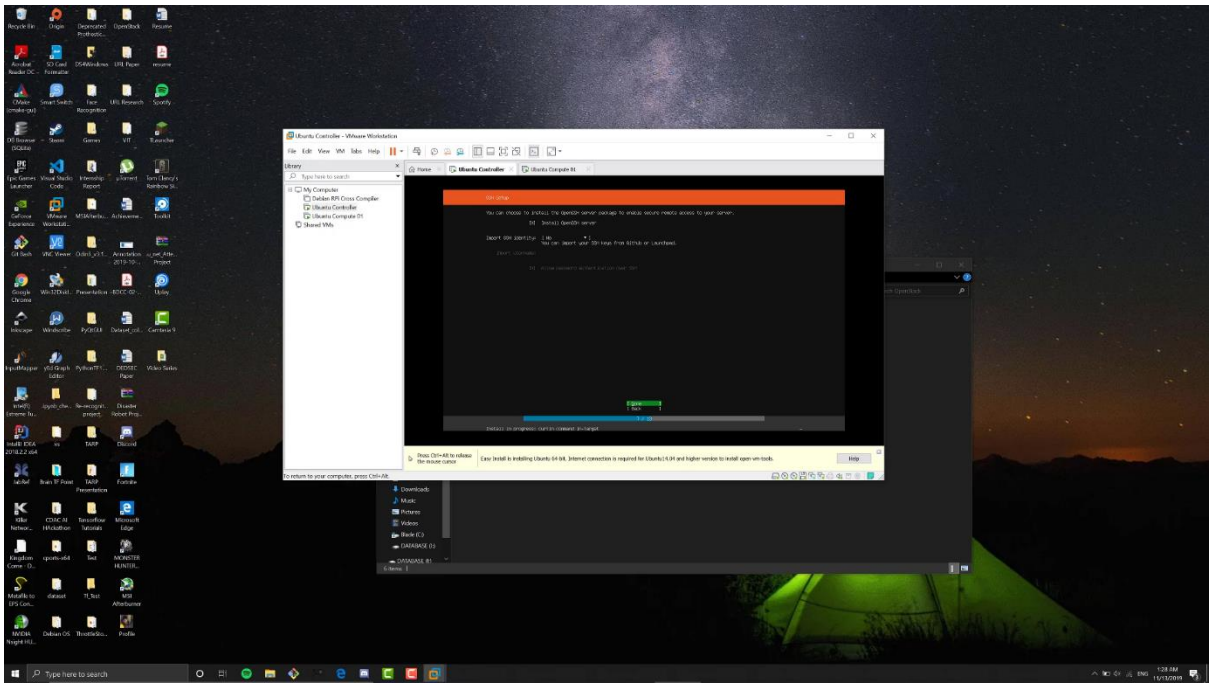




Now we have internet access as well as the host access.

Next, under filesystem setup just select use an entire disk. Then, setup your credentials such as Your name, Server's name, Username, Password.

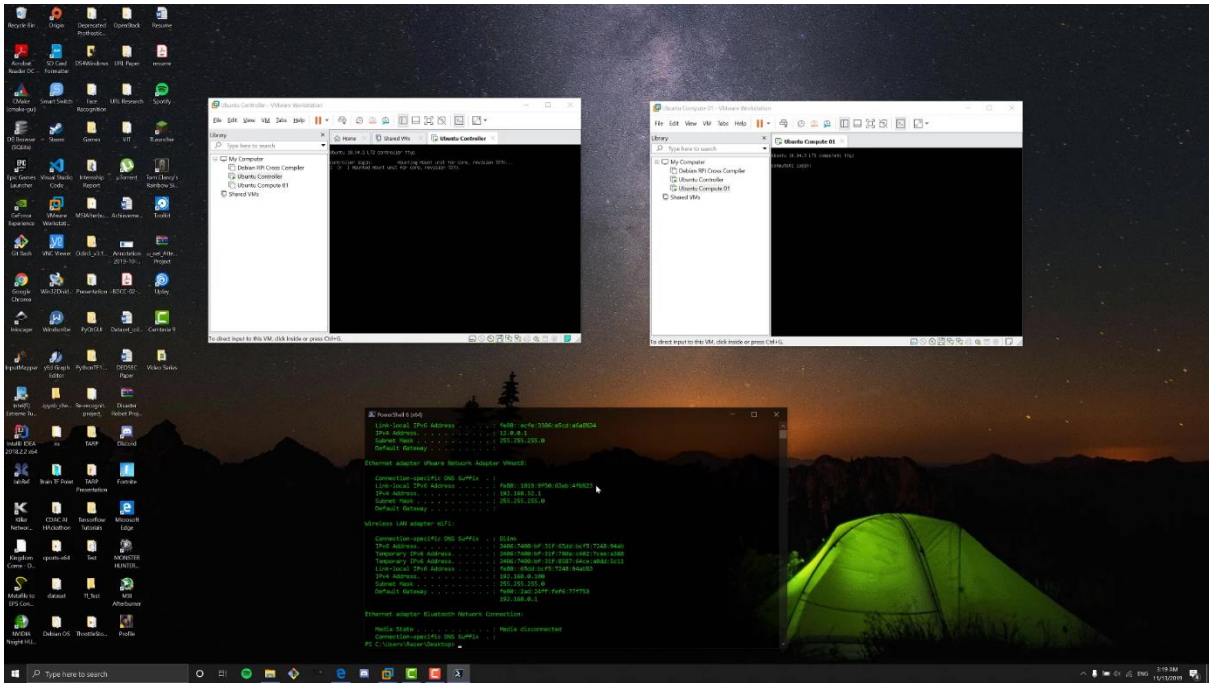
Under ssh setup, select install ssh server.

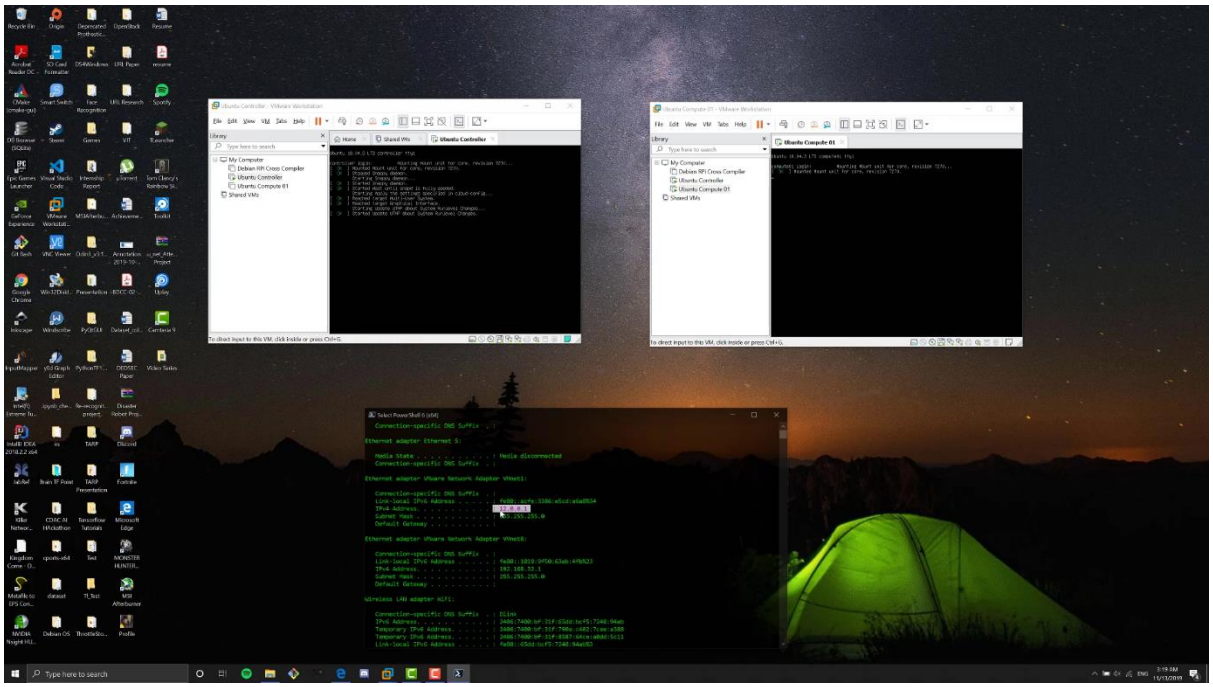


Finally, under featured server snaps, select canonical livepatch package.

Similarly, setup the compute VM the only change is that network address for that VM is Address: 12.0.0.12

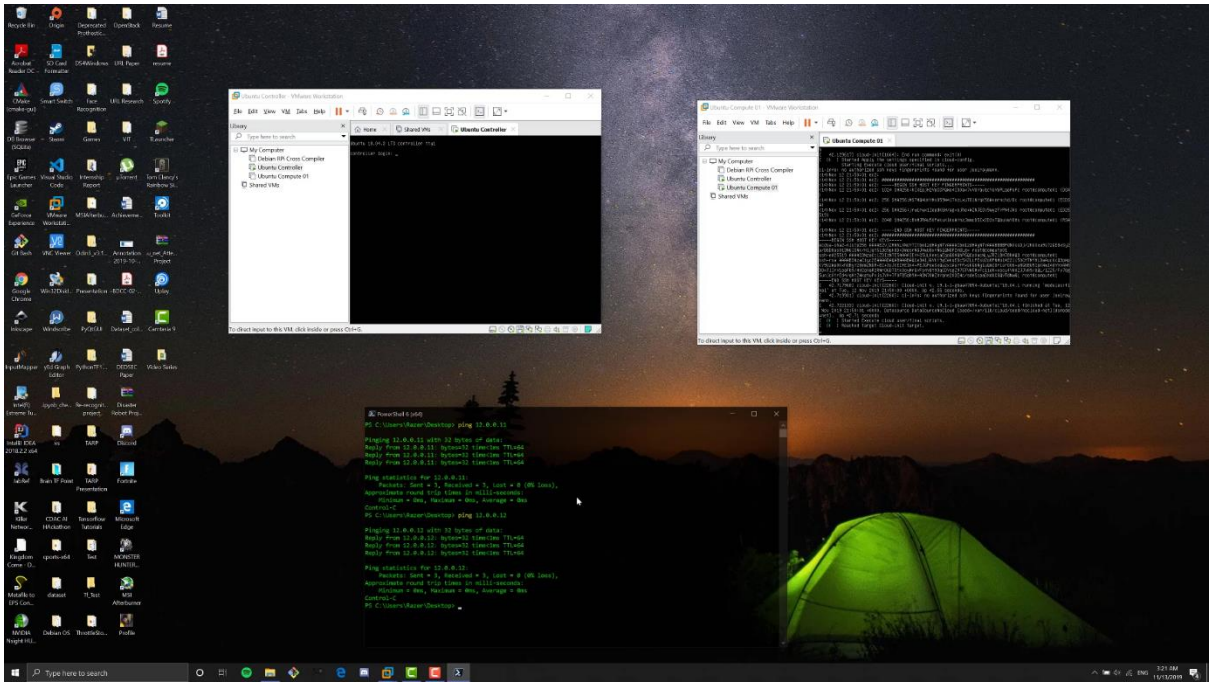
Perform a sanity check and that is to find the ip address of the configured VMware network adapter by typing 'ipconfig' in the command-line.



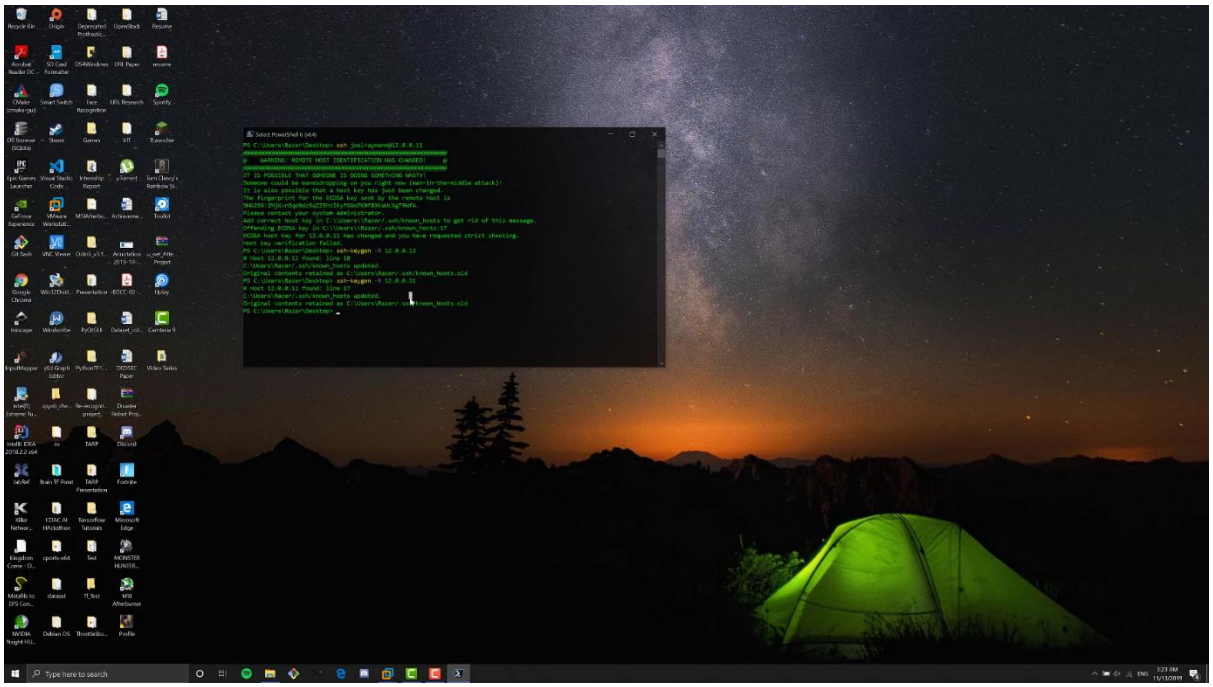


Once the controller VM is up and running, enter your credentials and login. Then do the same for compute VM too.

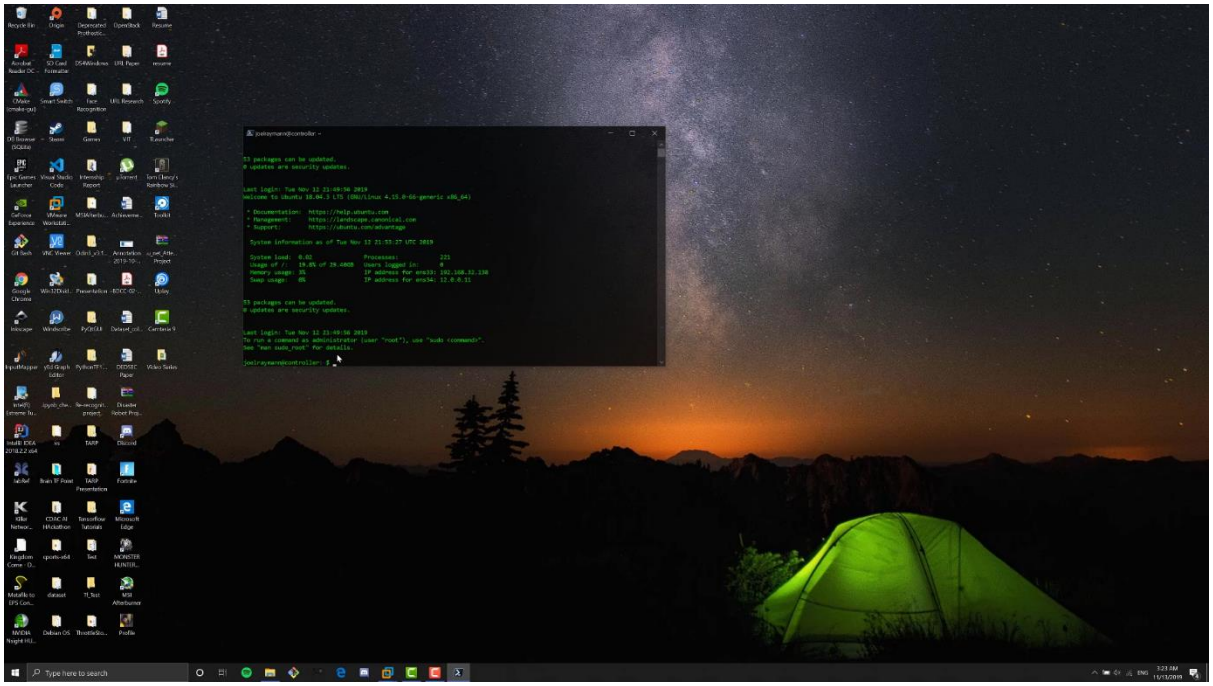
Next, check whether the manually configured network connection is working properly by pinging the respective ip addresses of the VMs from the host command-line.



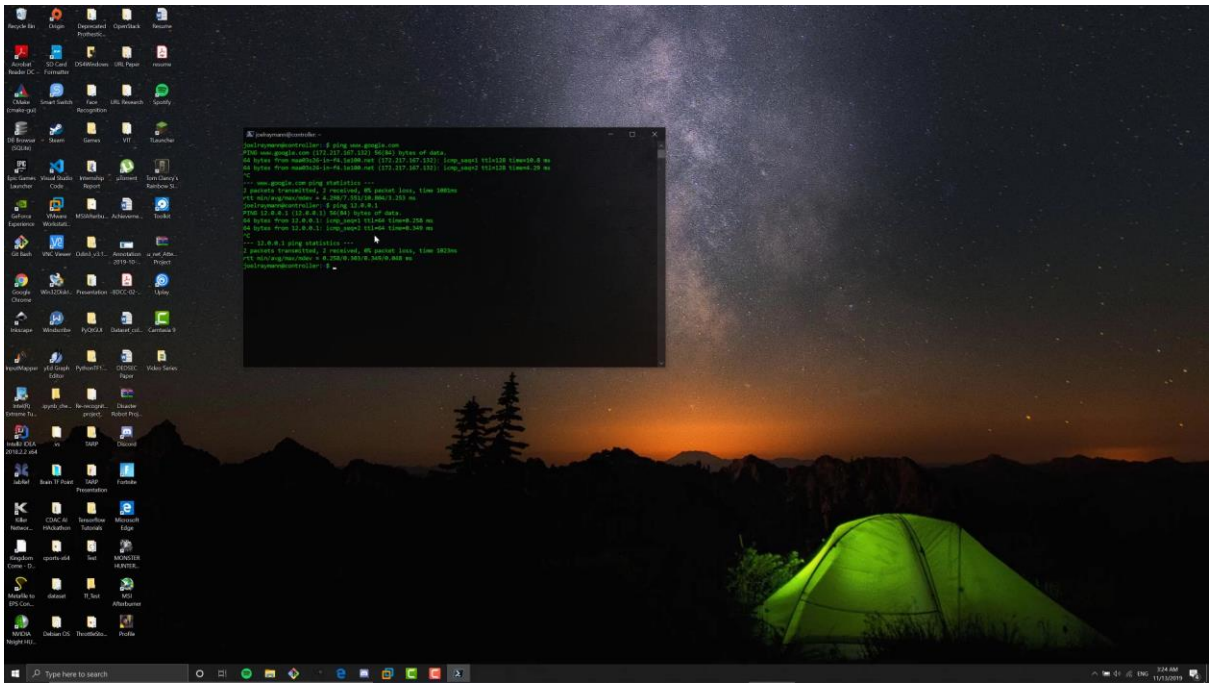
To check whether the ssh is working properly, the host machine should have Openssh installed, if you may encounter an error kindly refer the screenshot.



After rectifying the error as shown in the above screenshot, when we try to check again, we must encounter this result as shown in the below screenshot and we will get access to our controller command-line.

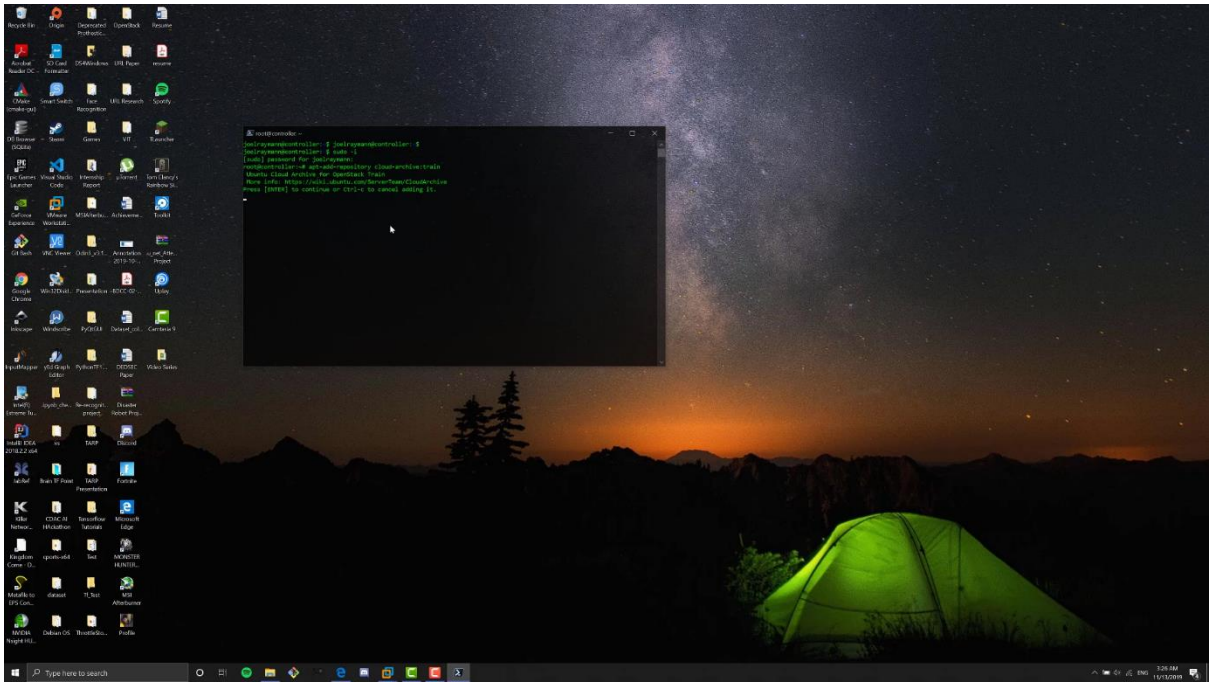


Now the below screenshot shows that our controller VM has access to both internet and private network.



Next before updating the guest OS, we have to add a repository called ubuntu CloudArchive that satisfies the package requirements for openstack. Run the following commands:

1. `sudo -i`
2. `apt-add-repository cloud-archive:train`
3. `apt-update`
4. `apt-upgrade`



Now similarly, perform the same set of operations in the compute VM starting with command-line access to addition of CloudArchive repository.

Note: While doing ssh in powershell for compute node make sure that the following command is like: `ssh username@12.0.0.012`

Next, we have to ensure whether python v2.x is installed by running the following command: `python --version`, since by default python v3.x will be installed.

We are going to use python v2.x because although python3.x is currently the recommended version it has certain issues and so to overcome that we are using python v2.x.

To install python v2.x, run the following in the command-line:

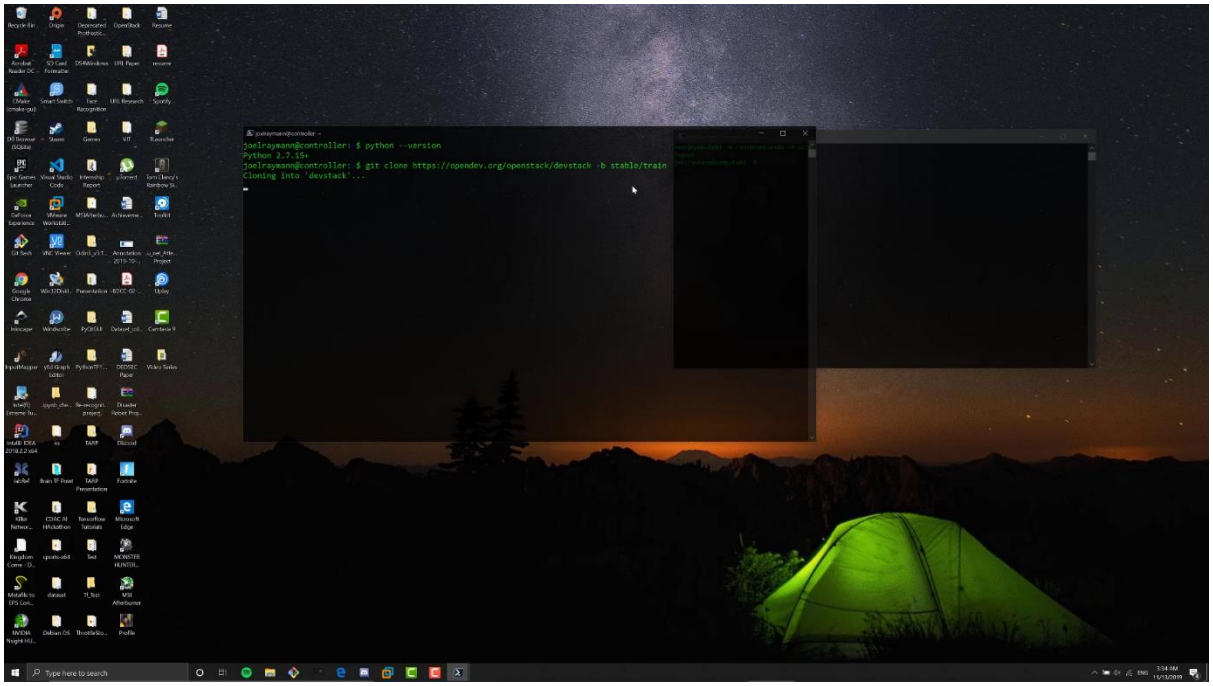
```
sudo apt install python-pip
```

Make sure that python v2.x is installed in compute VM too.

Now, we have to clone the openstack repository for our version by default it will clone the latest version but we will specify our required version as follows and run this command:

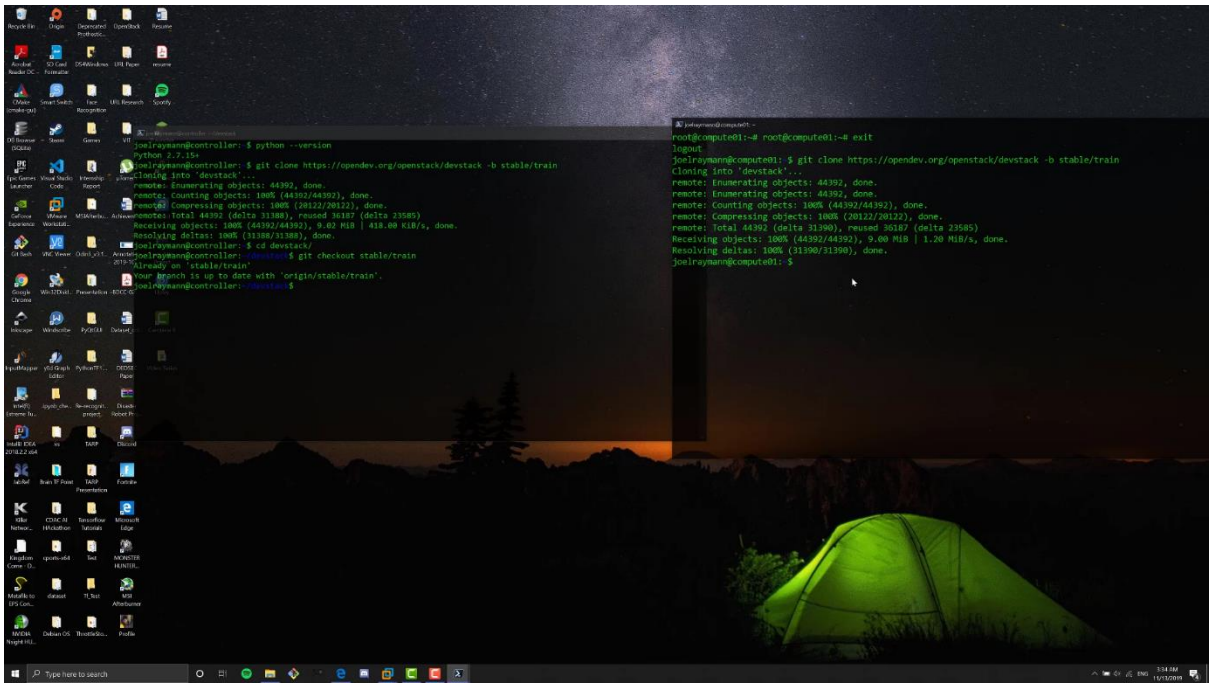
```
git clone https://opendev.org/openstack/devstack -b stable/train
```

Similarly run the above command in the compute VM too.



Now once we received the objects from the git repository, let's do a sanity check and to do that run the following commands in both VMs:

1. `cd devstack/`
2. `git checkout stable/train`



From now on until any mention of any compute node, we are not going to use it so minimize the powershell that has access to compute node and let's focus more on configuring the controller node.

Now run the following commands:

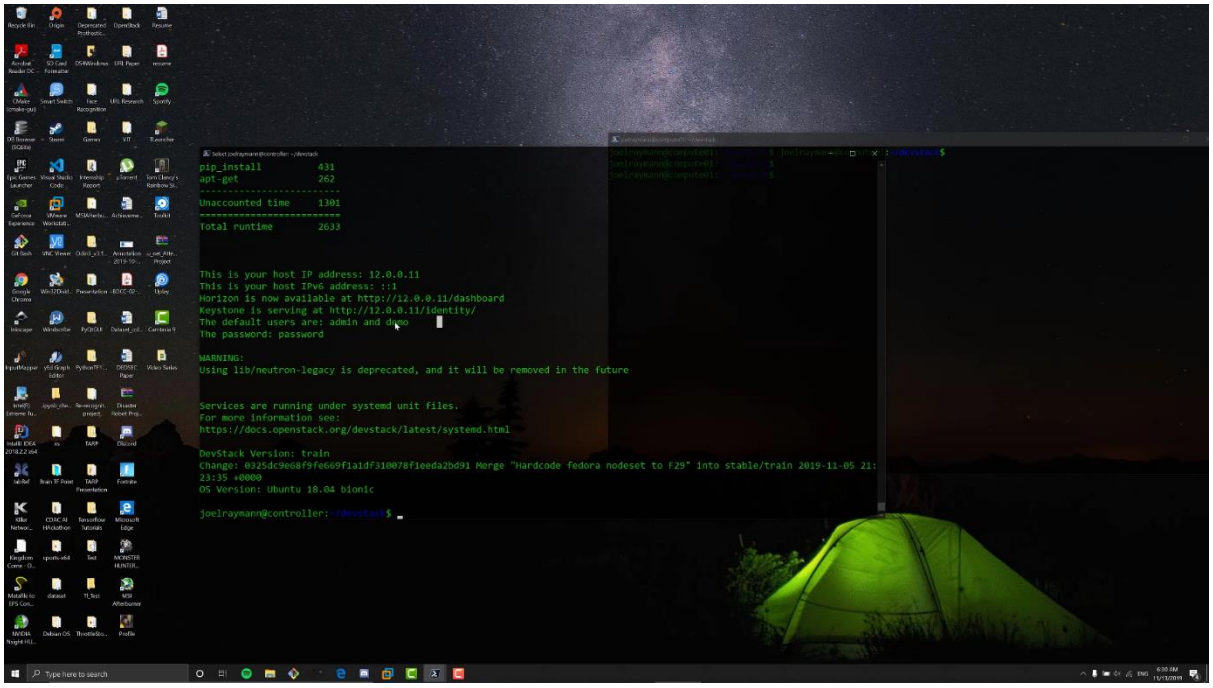
1. cd devstack/
2. nano local.conf

Note: copy the following into the local.conf file and save it by Ctrl+O and exit.

```
[[local|localrc]]
enable_plugin octavia https://opendev.org/openstack/octavia
# If you are enabling horizon, include the octavia dashboard
enable_plugin octavia-dashboard
https://opendev.org/openstack/octavia-dashboard.git
# If you are enabling barbican for TLS offload in Octavia,
include it here.
# enable_plugin barbican
https://opendev.org/openstack/barbican

# If you have python3 available:
# USE_PYTHON3=True

# ===== BEGIN localrc =====
HOST_IP=12.0.0.11
FIXED_RANGE=10.4.128.0/20
FLOATING_RANGE=12.0.0.128/25

DATABASE_PASSWORD=password
ADMIN_PASSWORD=password
SERVICE_PASSWORD=password
SERVICE_TOKEN=password
RABBIT_PASSWORD=password
# Enable Logging
LOGFILE=/opt/stack/logs/stack.sh.log
VERBOSE=True
LOG_COLOR=True
# Pre-requisite
ENABLED_SERVICES=rabbit,mysql,key
# Horizon - enable for the OpenStack web GUI
ENABLED_SERVICES+=,horizon
# Nova
ENABLED_SERVICES+=,n-api,n-crt,n-cpu,n-cond,n-sch,n-api-
meta,n-sproxy
ENABLED_SERVICES+=,placement-api,placement-client
# Glance
ENABLED_SERVICES+=,g-api,g-reg
```

```
# Neutron
ENABLED_SERVICES+=,q-svc,q-agt,q-dhcp,q-l3,q-meta,neutron
ENABLED_SERVICES+=,octavia,o-cw,o-hk,o-hm,o-api
# Cinder
ENABLED_SERVICES+=,c-api,c-vol,c-sch
# Tempest
ENABLED_SERVICES+=,tempest
# Barbican - Optionally used for TLS offload in Octavia
# ENABLED_SERVICES+=,barbican
# ===== END localrc =====
```

3. sudo apt-get install virutalenv (optional)
4. ./stack.sh (will take a long time)



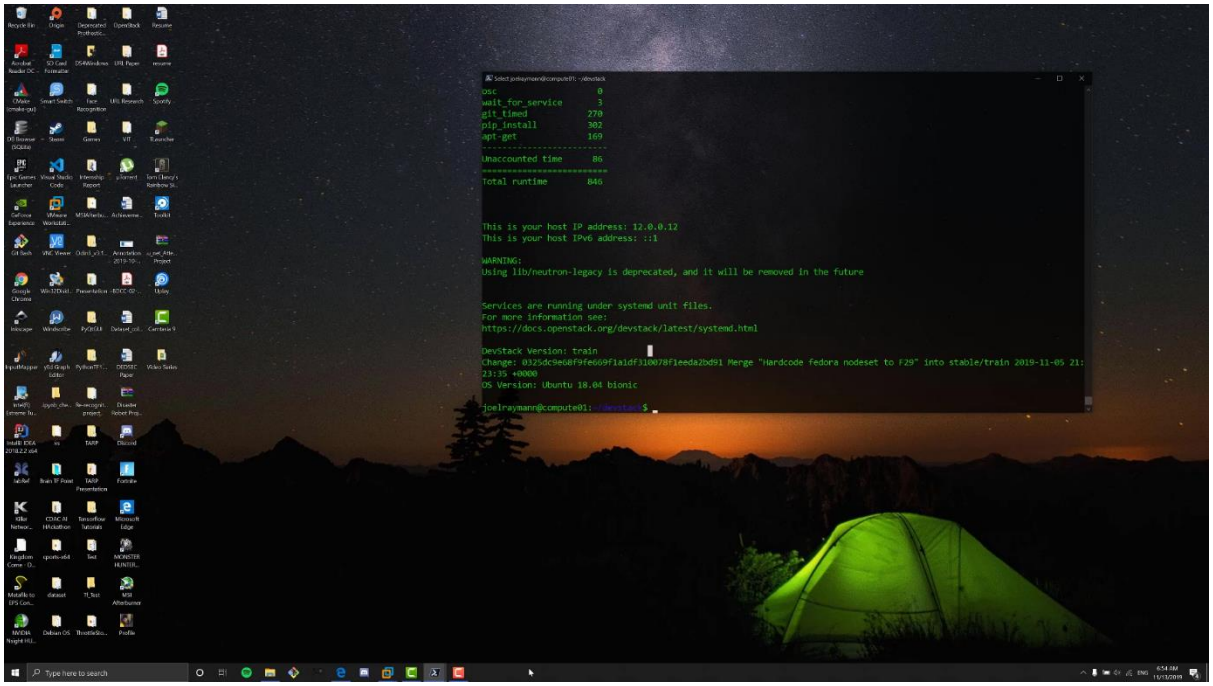
After the last command of controller VM has been successfully completed, now open the compute VM's powershell window to gain access to it's terminal and run the following commands:

1. cd devstack/
2. nano local.conf

   Note: copy the following into the local.conf file and save it by Ctrl+O and exit.

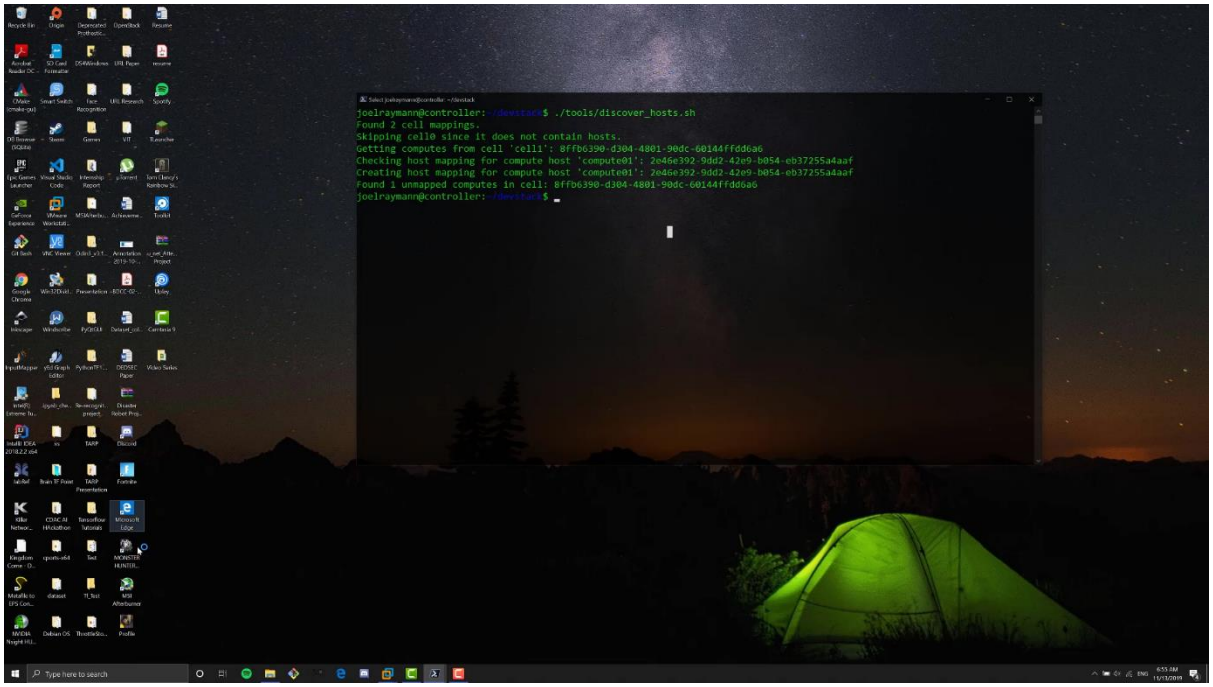
```
[[local|localrc]]
HOST_IP=12.0.0.12 # change this per compute node
FIXED_RANGE=10.4.128.0/20
FLOATING_RANGE=12.0.0.128/25
LOGFILE=/opt/stack/logs/stack.sh.log
ADMIN_PASSWORD=password
DATABASE_PASSWORD=password
RABBIT_PASSWORD=password
SERVICE_PASSWORD=password
DATABASE_TYPE=mysql
SERVICE_HOST=192.168.42.11
MYSQL_HOST=$SERVICE_HOST
RABBIT_HOST=$SERVICE_HOST
GLANCE_HOSTPORT=$SERVICE_HOST:9292
ENABLED_SERVICES=n-cpu,q-agt,n-api-meta,c-vol,placement-client
NOVA_VNC_ENABLED=True
NOVNCPROXY_URL="http://$SERVICE_HOST:6080/vnc_lite.html"
VNCSERVER_LISTEN=$HOST_IP
VNCSERVER_PROXYCLIENT_ADDRESS=$VNCSERVER_LISTEN
```

3. ./stack.sh (will take less time than controller node)

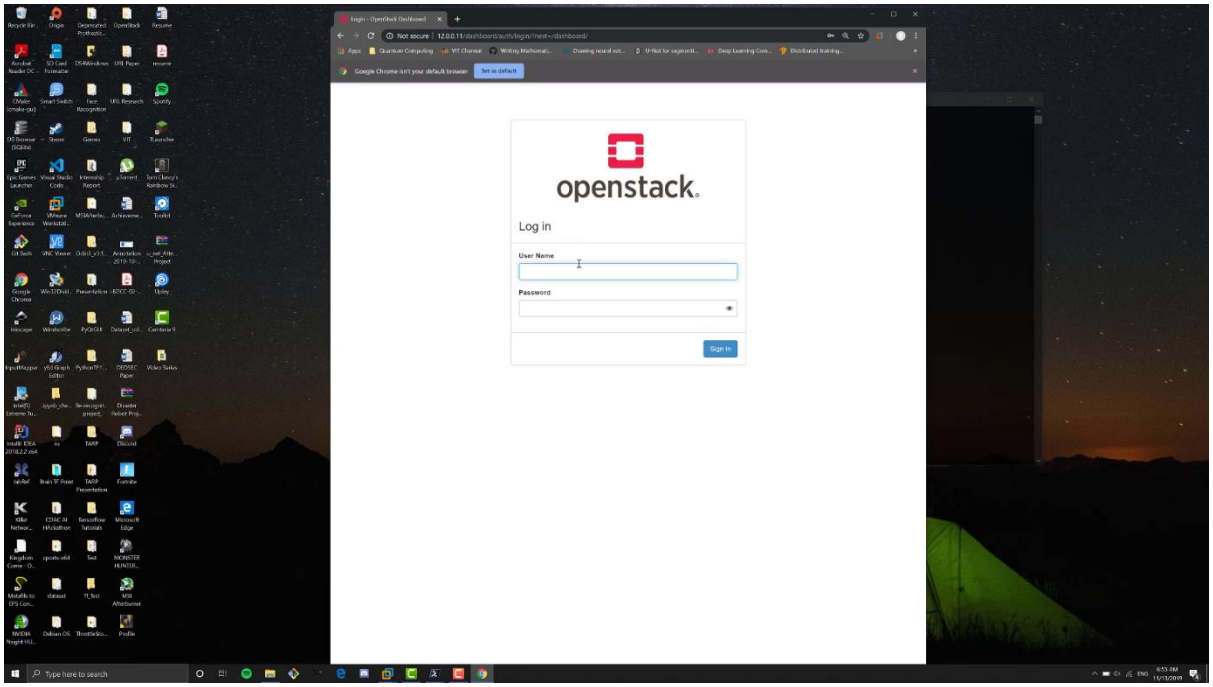


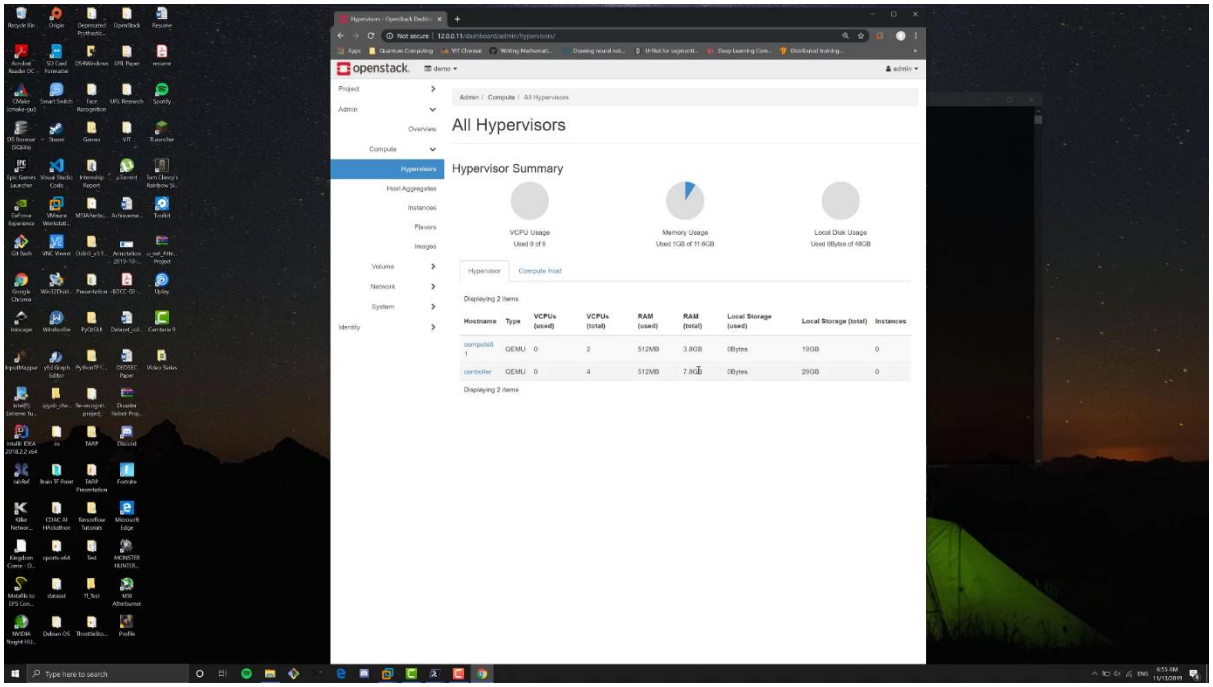
Now open the powershell that has the terminal of controller node and run the following command in order for the controller node to discover the compute node.

1. `cd devstack`
2. `./tools/discover_hosts.sh`

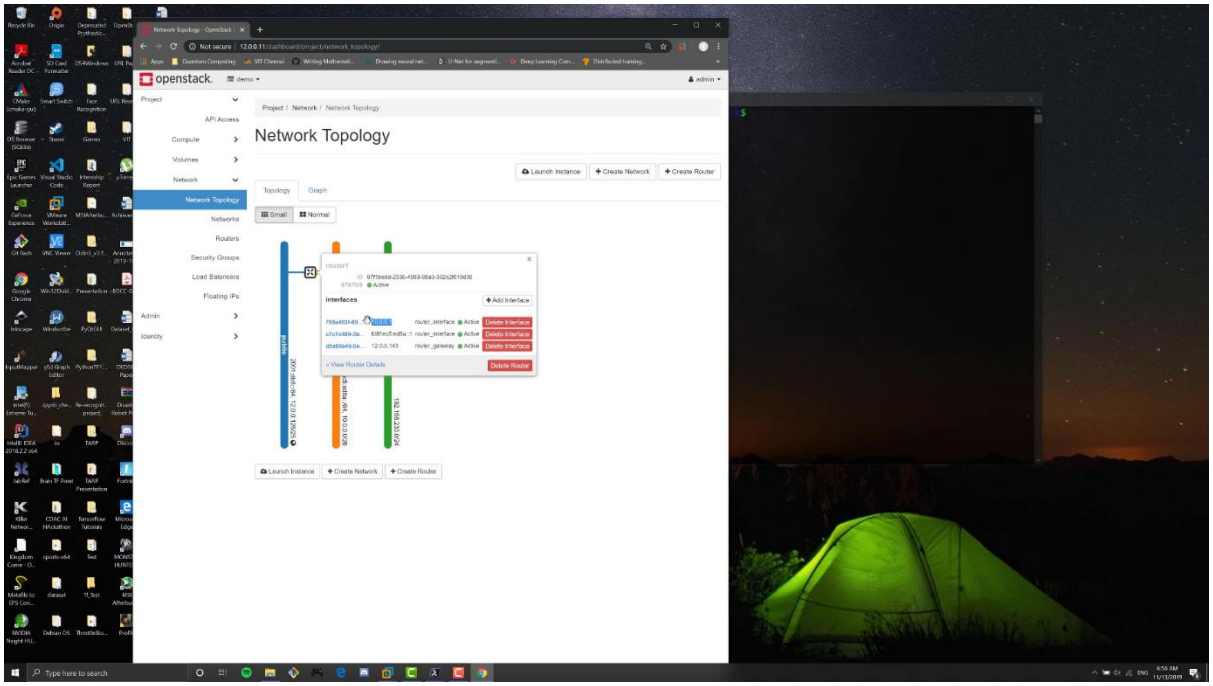


Then, to check whether the openstack is properly setup along with the discovery of compute node by controller node using google chrome type the ip address of the controller node i.e. 12.0.0.11 in this case. The username is admin, password is password as we configured earlier.



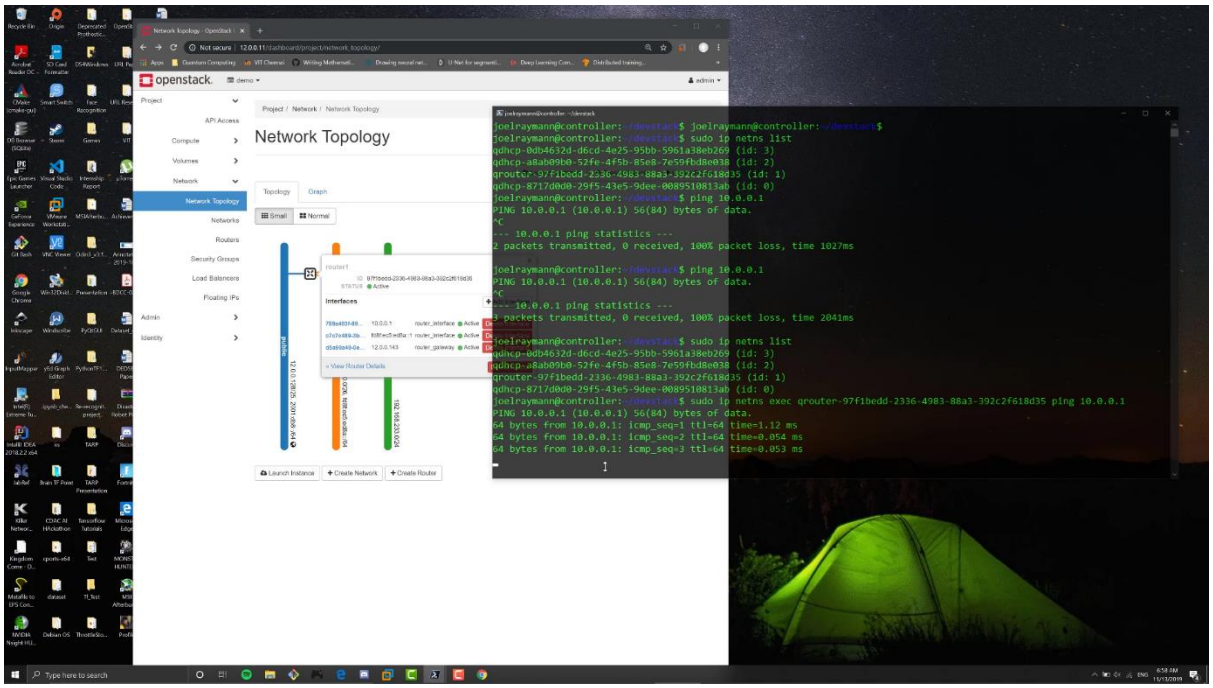


Once we finish setting up openstack in order to communicate from the public network to the private network the only access is via a router which you can see from the below screenshot the gateway for the private network is 10.0.0.1 as you can see from the screenshot.



Even though we know the gateway of the private network in order to communicate with it we have to know the router id. We can know the router id and communicate with the private network via the following commands:

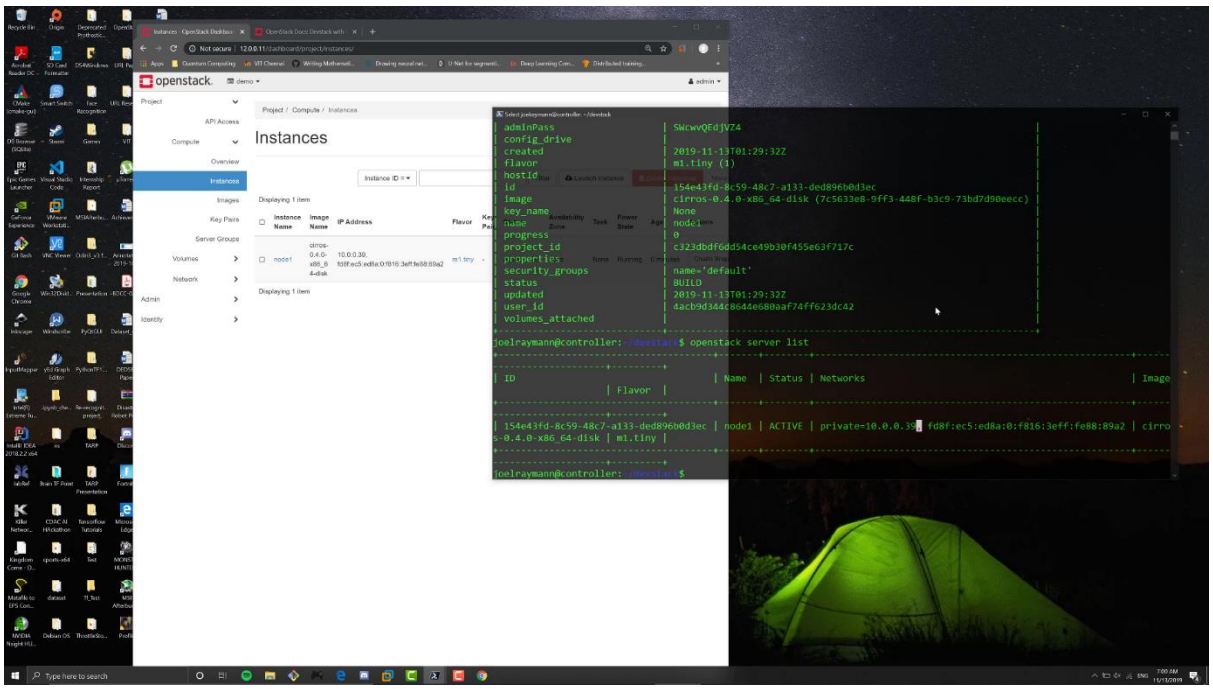
1. `sudo ip netns list`
2. `sudo ip netns exec <router-id> ping 10.0.0.1`



Now that we have understood how to communicate with the private network from public network let's create a nova instance in controller node. To create a nova instance run the following commands:

1. `cd devstack/`
2. `. ./openrc`
3. `openstack server create --image $(openstack image list | awk '/ cirros-.*-x86_64-.* / {print $2}') --flavor 1 --nic net-id=$(openstack network list | awk '/ private / {print $2}') node1`

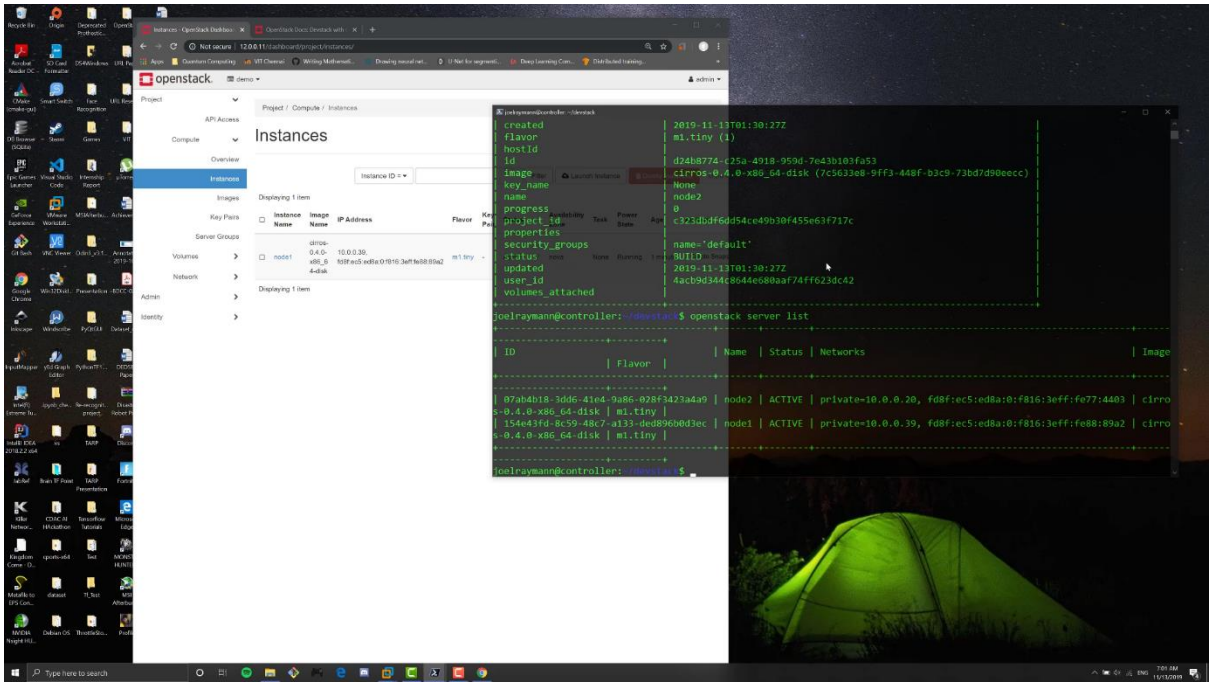
   Note: To check whether the above instance creation was successful, kindly run the following command: `openstack server list`



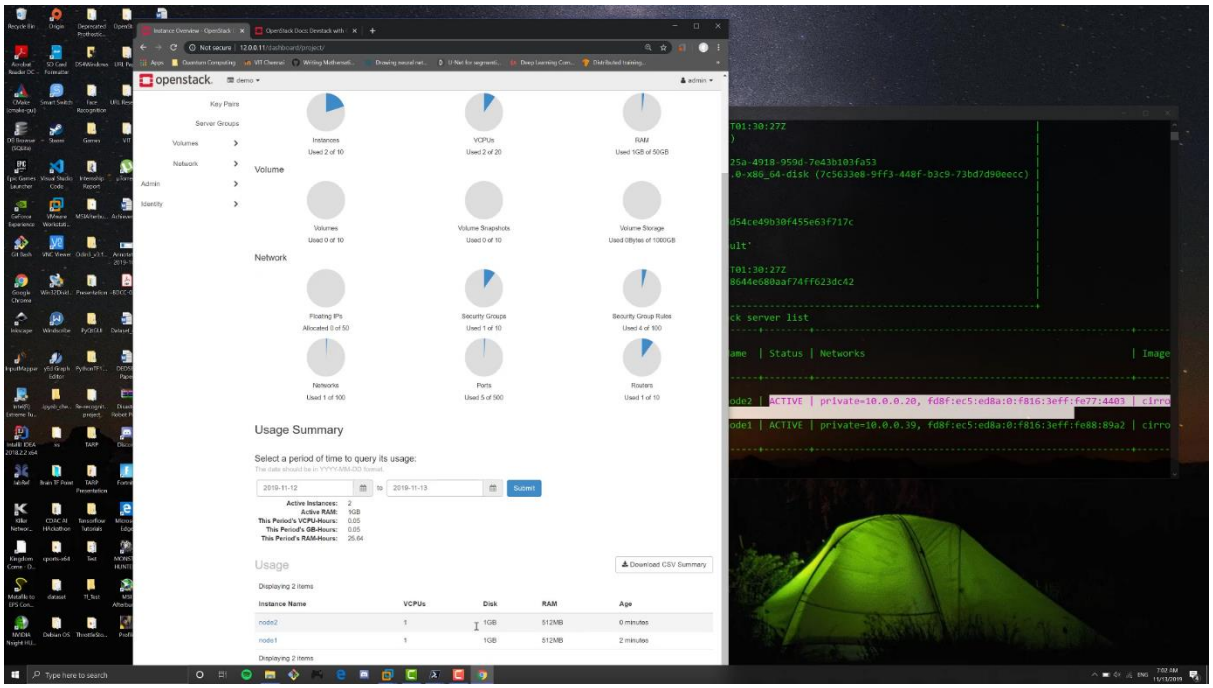
Now, let's create another nova instance in the compute node. To do that just run the following commands.

1. `cd devstack/`
2. `. ./openrc`
3. `openstack server create --image $(openstack image list | awk '/ cirros-.*-x86_64-.* / {print $2}') --flavor 1 --nic net-id=$(openstack network list | awk '/ private / {print $2}') node2`

Note: To verify whether our instance creation was successful, run the following command in the controller node: `openstack server list`



We can also verify the same in our horizon dashboard as you can see from the screenshot.

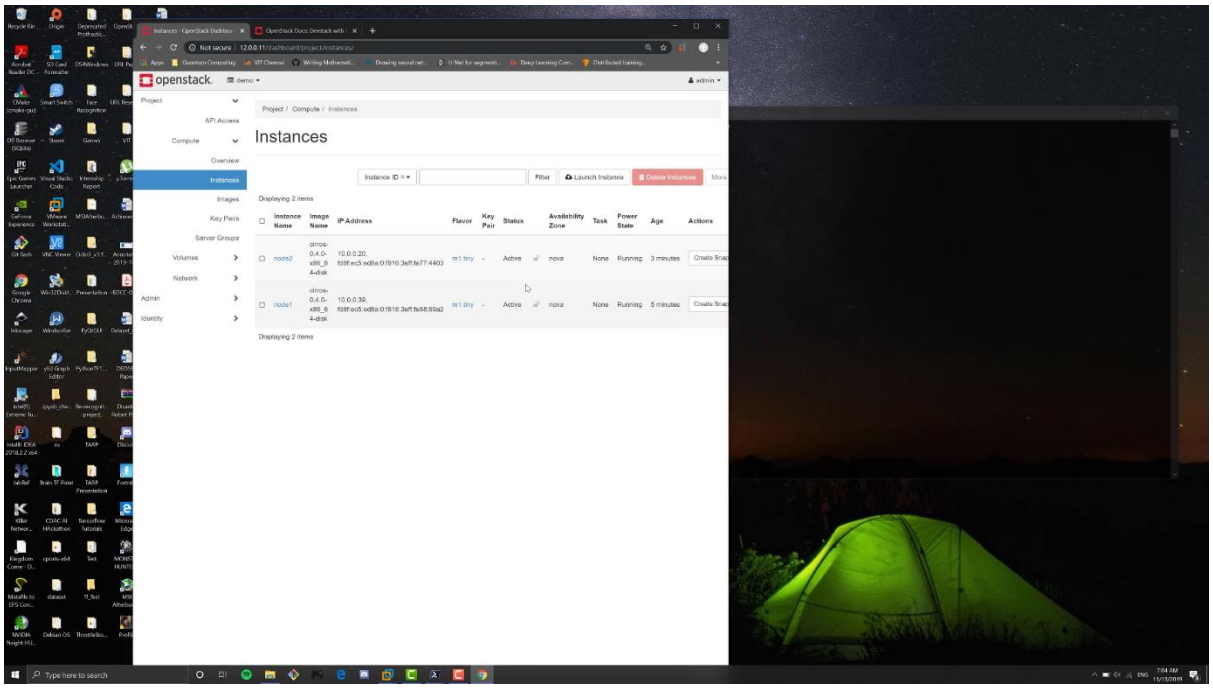


Next, run the following commands:

1. `openstack security group rule create default --protocol icmp`
2. `openstack security group rule create default --protocol tcp --dst-port 22:22`
3. `openstack security group rule create default --protocol tcp --dst-port 80:80`

The icmp protocol is responsible for pinging, the tcp protocol in port 22:22 is responsible for ssh and the tcp protocol in port 80:80 is responsible for http.

From the horizon dashboard we can know the ip address of each instance as you can see from the screenshot, in this case for node1 it is 10.0.0.39 and for node2 it is 10.0.0.20



Now to communicate with say node2 and node1, as we have seen earlier how to communicate from the public network to the private network, run the following commands to check whether we are able to communicate:

```
1. sudo ip netns exec <router-id> ping 10.0.0.20
2. sudo ip netns exec <router-id> ping 10.0.0.39
```

Now communication from public network to the nodes in the private network is possible, in order to get access to the terminal of the nodes in the private network from the public network, in this case we are going to get the terminal access of node1 using ssh by running the following commands:

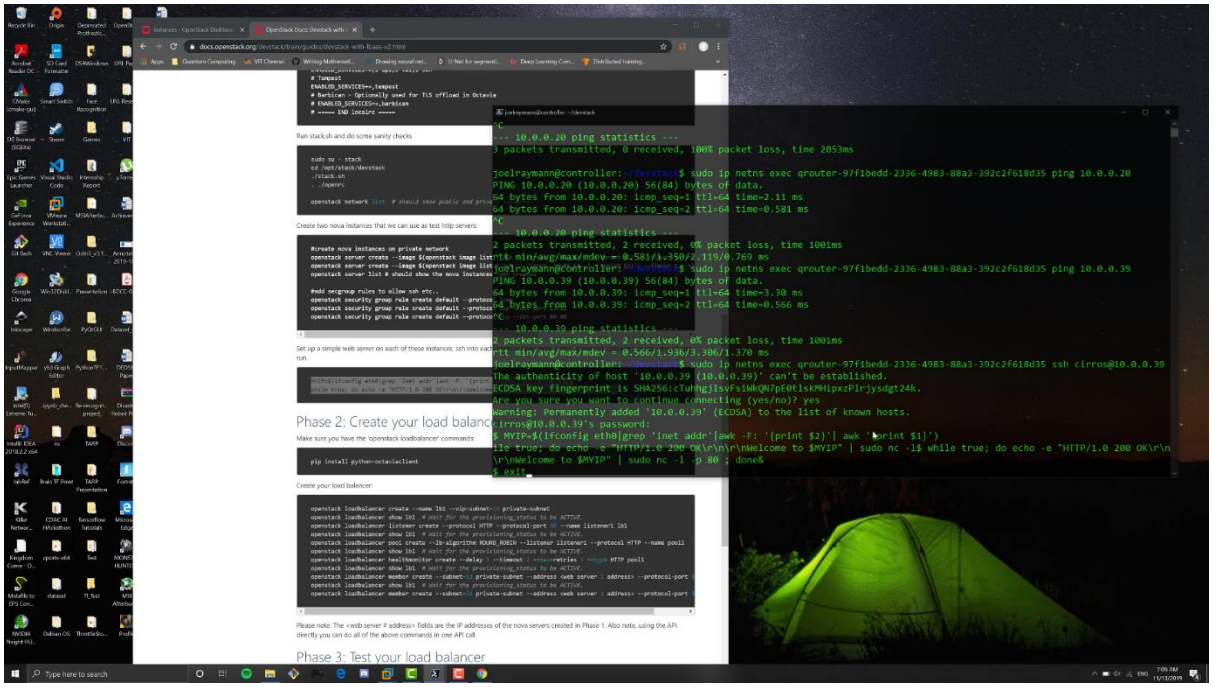
```
sudo ip netns exec <router-id> ssh cirros@10.0.0.39
```

Note: The default password and in this case too is ‘gocubsgo’.

Next run the following command to setup a web server:

```
MYIP=$(ifconfig eth0|grep 'inet addr'|awk -F: '{print $2}'| awk '{print $1}') while true; do echo -e "HTTP/1.0 200 OK\r\n\r\nWelcome to $MYIP" | sudo nc -l -p 80 ; done&
```

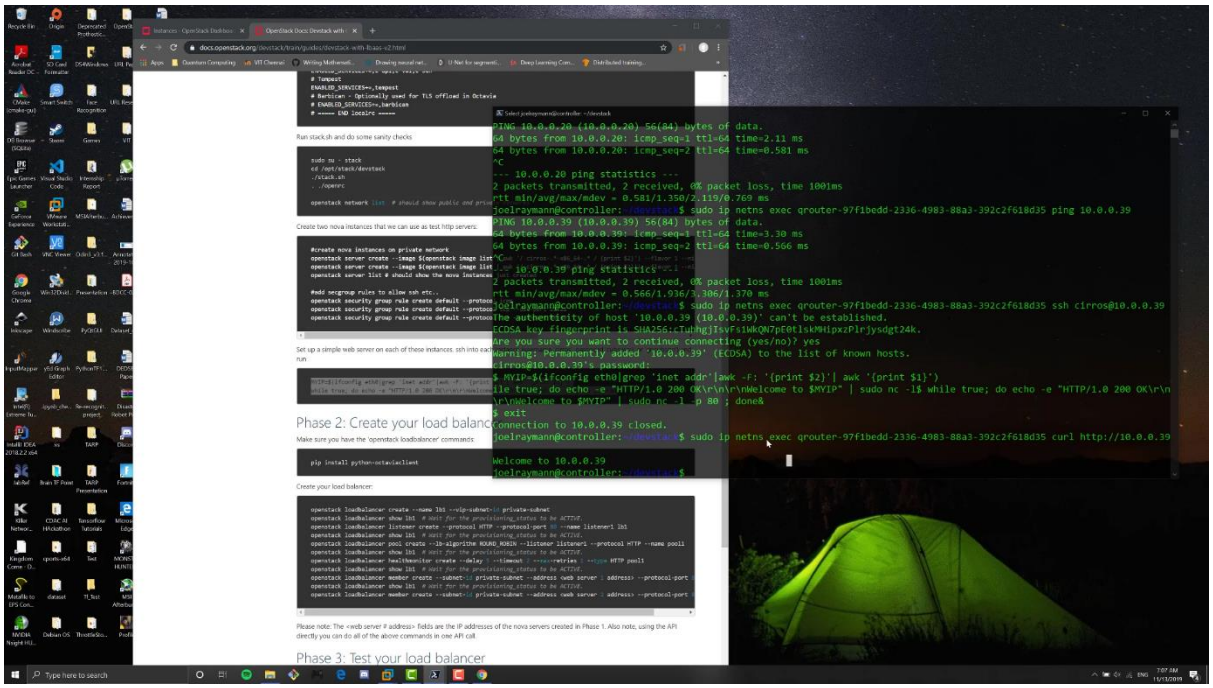
Then exit the terminal of node1.



To check whether our webserver in node1 works properly from the controller terminal run the following command:

```
sudo ip netns exec <router-id> curl http://10.0.0.39
```

Note: The resulting output as seen in the below screenshot should be: `Welcome to 10.0.0.39`



Now, we have completed setting up a web server for node1 similarly we have to do for node2. To do that we have to run the following commands:

1. `sudo ip netns exec <router-id> ssh cirros@10.0.0.20`
2. `MYIP=$(ifconfig eth0|grep 'inet addr'|awk -F: '{print $2}'| awk '{print $1}') while true; do echo -e "HTTP/1.0 200 OK\r\n\r\nWelcome to $MYIP" | sudo nc -l -p 80 ; done&`
3. `exit`

Note: To check whether our webserver in node2 works properly from the controller terminal run the following command:

`sudo ip netns exec <router-id> curl http://10.0.0.20`

Note: The resulting output should be: `Welcome to 10.0.0.20`

Phase 2: Create your load balancer

Make sure you have the 'openstack loadbalancer' commands:

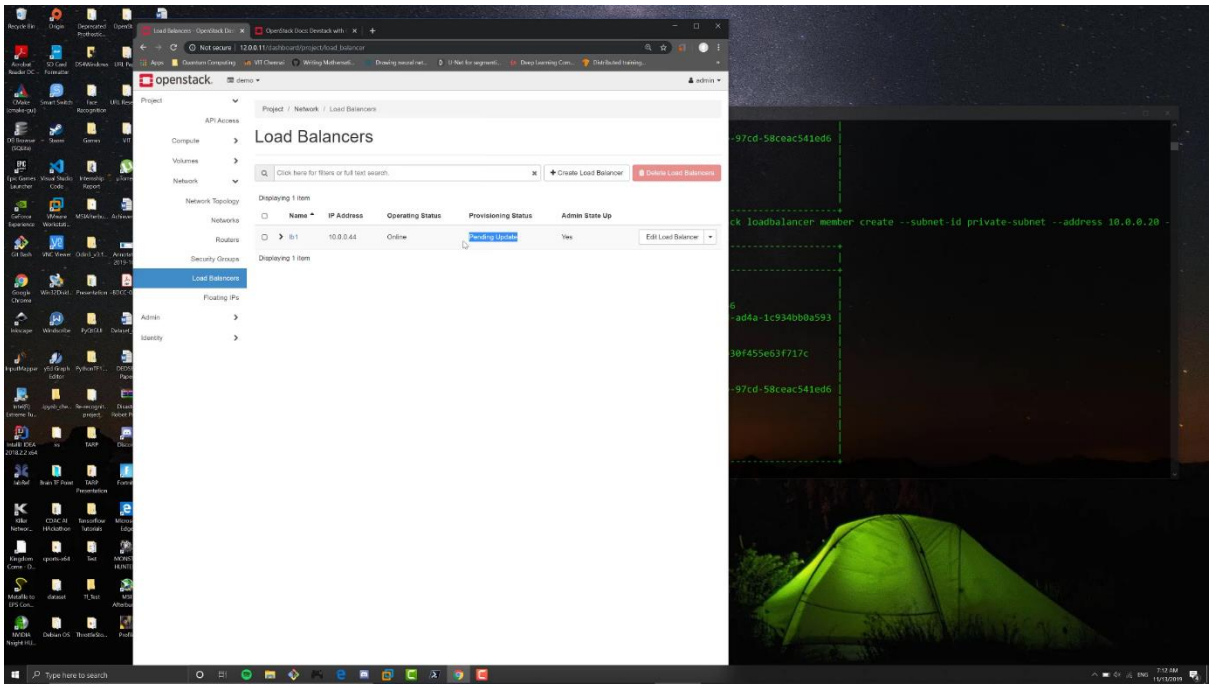
`pip install python-octaviaclient`

To setup the loadbalancer simply run the sequence of commands and wait for the provisioning status to be active in the mentioned cases:

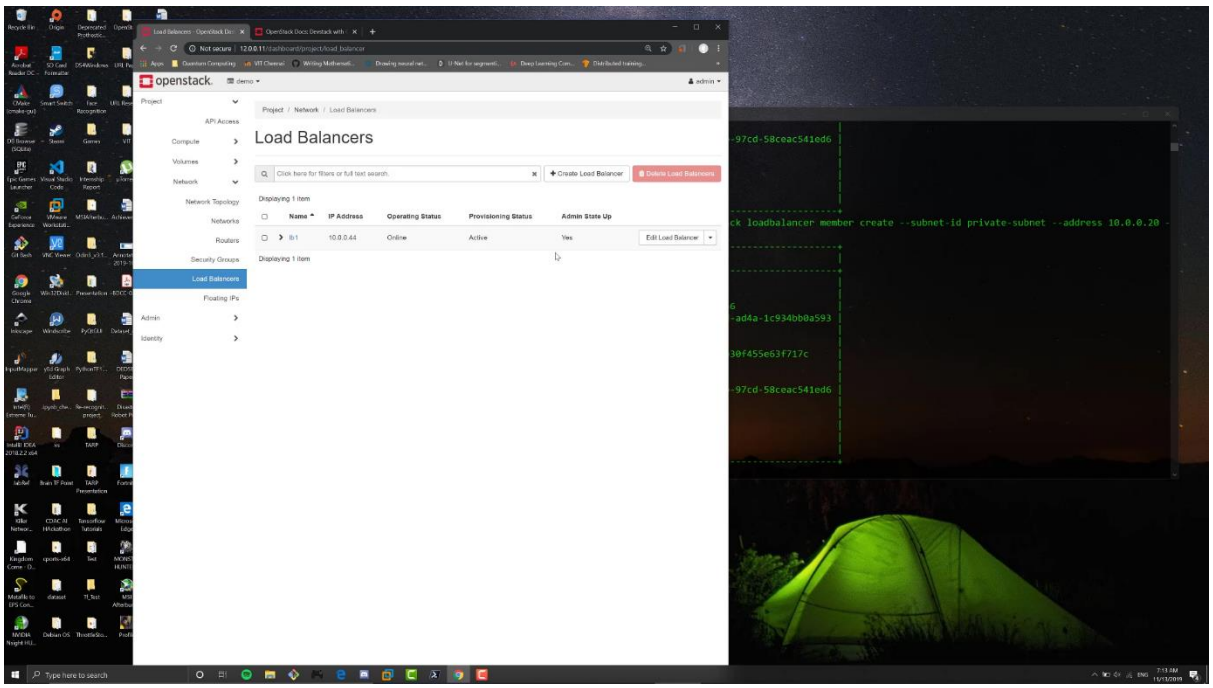
1. `openstack loadbalancer create --name lb1 --vip-subnet-id private-subnet`
2. `openstack loadbalancer show lb1  # Wait for the provisioning_status to be ACTIVE.`
3. `openstack loadbalancer listener create --protocol HTTP --protocol-port 80 --name listener1 lb1`
4. `openstack loadbalancer show lb1  # Wait for the provisioning_status to be ACTIVE.`
5. `openstack loadbalancer pool create --lb-algorithm ROUND_ROBIN --listener listener1 --protocol HTTP --name pool1`
6. `openstack loadbalancer show lb1  # Wait for the provisioning_status to be ACTIVE.`
7. `openstack loadbalancer healthmonitor create --delay 5 --timeout 2 --max-retries 1 --type HTTP pool1`
8. `openstack loadbalancer show lb1  # Wait for the provisioning_status to be ACTIVE.`
9. `openstack loadbalancer member create --subnet-id private-subnet --address 10.0.0.39 --protocol-port 80 pool1`
10. `openstack loadbalancer show lb1 # Wait for the provisioning_status to be ACTIVE.`
11. `openstack loadbalancer member create --subnet-id private-subnet --address 10.0.0.20 --protocol-port 80 pool1`

To check the provisioning status of a load balancer we can go to the horizon dashboard and to know the current status of our loadbalancer as shown in the screenshot.

Provisioning status: Pending Update

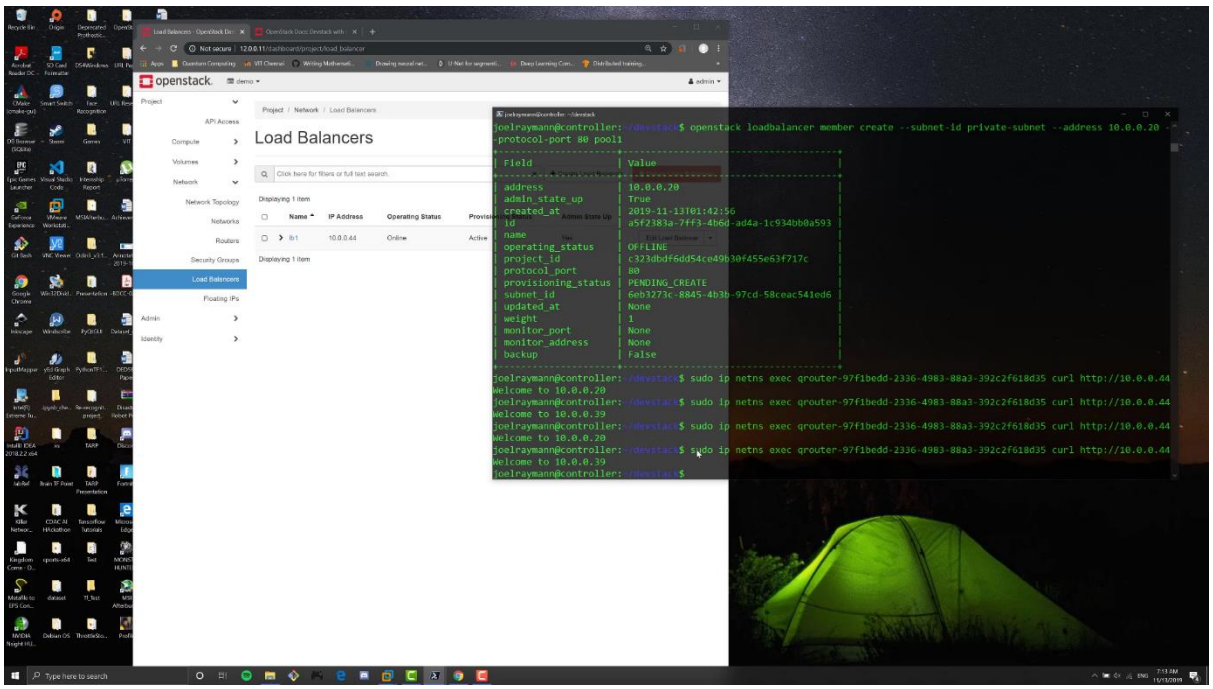


Provisioning Status: Active

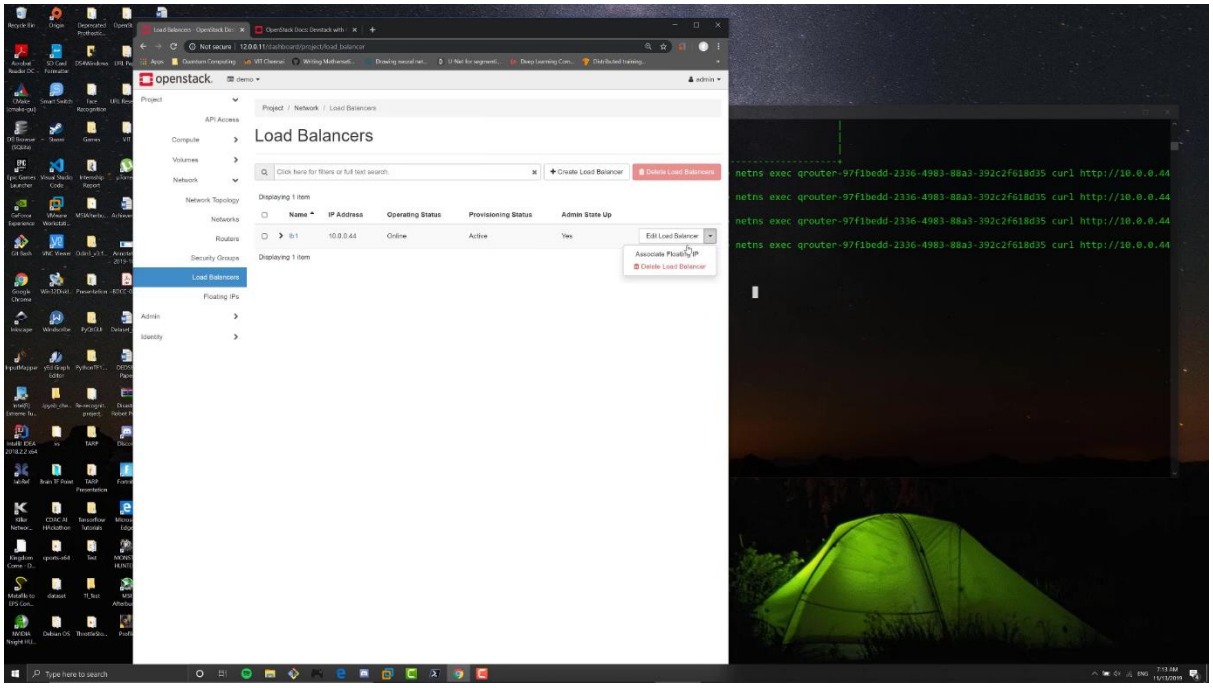


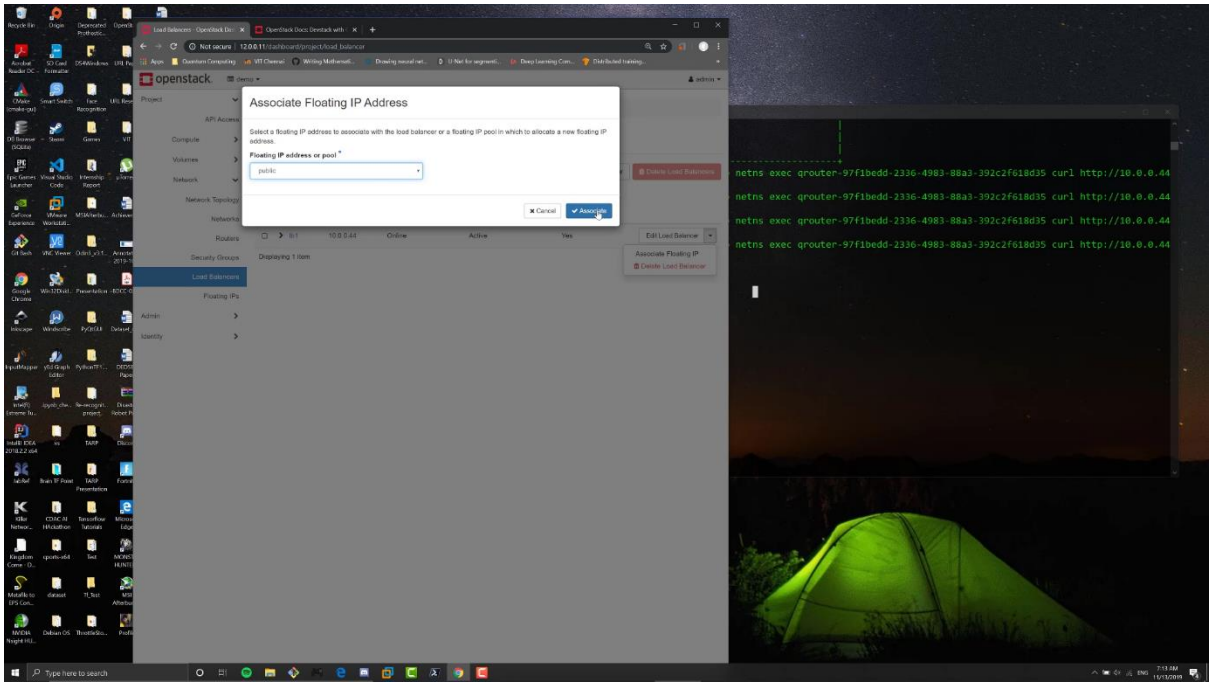
From this above screenshot, the ip address of our loadbalancer is 10.0.0.44

The algorithm we have specified for the loadbalancer to make use of is Round-Robin in this case because to experience the working of the loadbalancer. That is when we curl the loadbalancer for the first time it will redirect to node1 and when we curl the loadbalancer for the second time it will redirect to node2. Similarly, for all odd curls the loadbalancer will redirect to node1 and for all even curls the loadbalancer will redirect to node2.

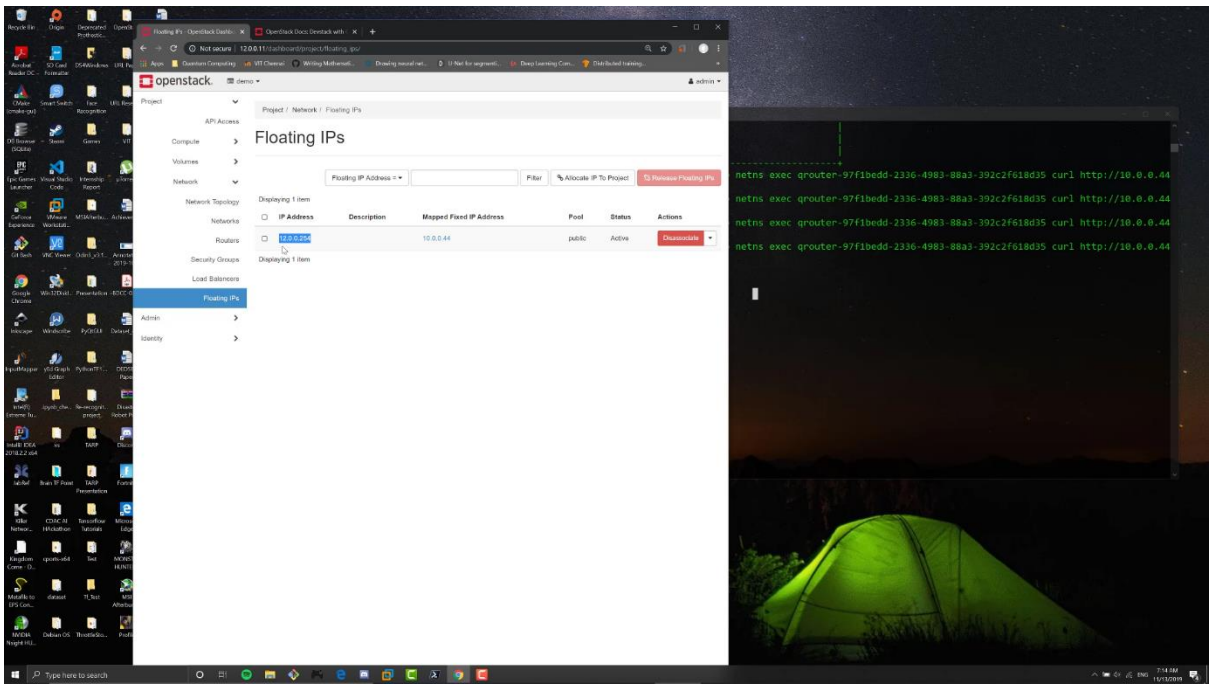


Now let's associate the loadbalancer to public network using horizon dashboard.

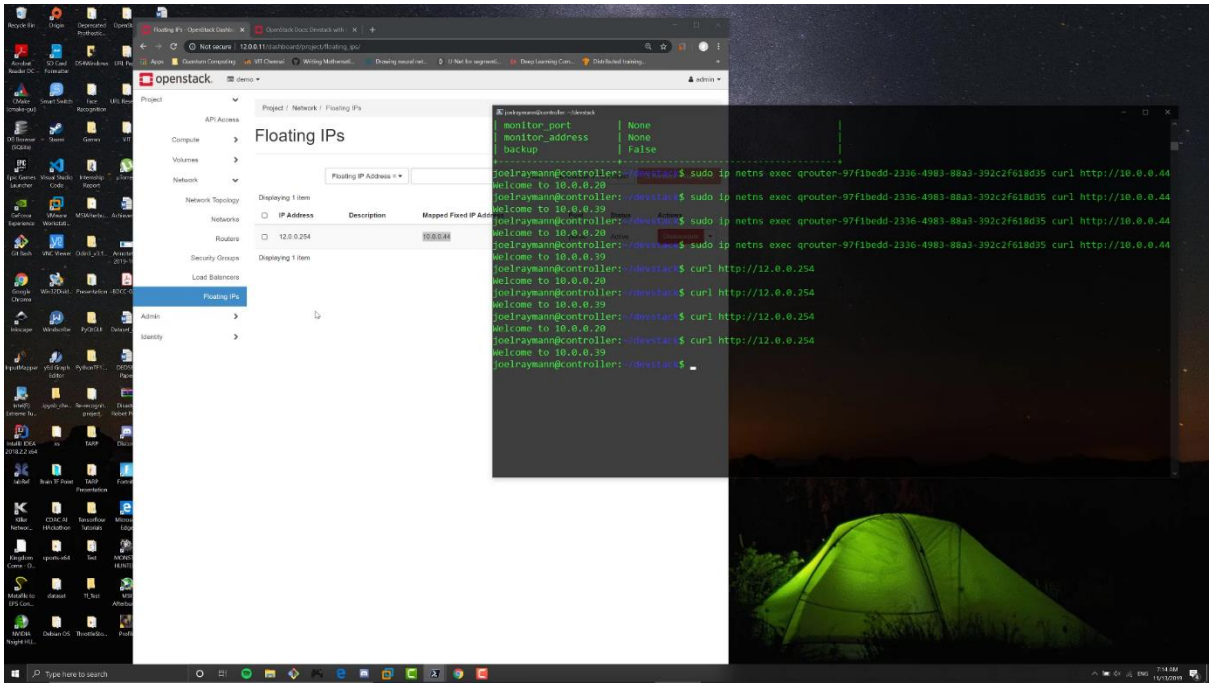




In the horizon dashboard, under the Floating IPs section, we can see that the ip address 12.0.0.254 is mapped to 10.0.0.44 which is the ip address of our loadbalancer.



Now, we can curl the loadbalancer directly from public network using the following command: `curl http://12.0.0.254`



Therefore, we have successfully setup an apache http server and applied a properly function loadbalancer that works based on round-robin algorithm.

Project Video Link

https://drive.google.com/file/d/1dAHGHd2RzuJSKd2ArBM77IkkRQ8mBVmJ/view?usp=sharing

REFERENCES

1. www.openstack.org
2. https://docs.openstack.org/train/
3. https://www.educative.io/
4. https://www.nginx.com/