Final Project Report
Quantum Computing on Cloud

CSC 547 Cloud Computing Technology
Semester Lab Project
Fall 2018

Team 3

Ruturaj Vyawahare rvyawah@ncsu.edu (Lead)
Madhura Bhide mvbhide@ncsu.edu (Vice-Lead)
Charan Ram Vellaiyur Chellaram cvellai@ncsu.edu
Jagadeesh Saravanan jsarava@ncsu.edu

Computer Science Department
NC State University

Title

A cloud-based solution that leverages containers to offer quantum-computing as a service.

Abstract

Over 50 years of advances in mathematics, materials science and computer science have transformed quantum computing from theory to reality. Today, real quantum computers can be accessed through the cloud, and thousands of people have used them to learn, conduct research and tackle new problems. Quantum computing uses quantum bits, or 'qubits' instead of relying on conventional bits. These are quantum systems with two states. However, unlike a usual bit, they can store much more information than just 1 or 0, because they can exist in any superposition of these values. Such property of quantum computer enables quantum systems to store and even process huge amount of information as compared to usual computers. Quantum computers could one day provide breakthroughs in many disciplines, including materials and drug discovery, the optimization of complex systems, and artificial intelligence. But to realize those breakthroughs, making quantum computers widely useable and accessible is one of the greatest challenges for researchers today.

We aim to provide a viable solution to overcome this challenge. For better utilization of resources and to make quantum computing environment accessible to a large user base, we need a more efficient and scalable way for developers to gain access to quantum resources. In this project, using the open source VCL platform and docker container service, we containerize the access to IBM’s quantum computing resources, such that multiple users can seamlessly and securely run their quantum applications.

Table of Contents

List of Figures

List of Tables

Introduction

Quantum computers, devices that use the quantum mechanical superposition principle to process information are being developed, built and studied in organizations ranging from universities and national laboratories to start-ups and large corporations such as Google, IBM, Intel and Microsoft. These devices are of great interest because they could solve certain computationally “hard” problems, such as searching large unordered lists or factoring large numbers, much faster than any classical computer. This is because the quantum mechanical superposition principle makes it possible to explore multiple computational paths at once.

The accelerated progress of quantum information technology in the past five years has resulted in a substantial increase in investment and development of active quantum technology. Recently, IBM did allowed access to a five qubit superconducting chip to the internet community through an interactive platform called the Quantum Experience (QE) [1]. IBM Q is an industry first initiative to build universal quantum computers for business and science.

In our project, we aim to make quantum computing environment accessible to a large user base. To enable user to access such quantum computes on demand to all the users, we need a more efficient and scalable way for users to gain access to quantum resources.

Problem Statement

Currently, only a niche group of researchers and experienced developers have access to IBM Q computers to run their code. As the research and development community for quantum computing applications expands, there is a need to have a unified environment for developers to work on. That way, the developers can focus on the core functionality of the application rather than worrying about managing the libraries and dependencies for their applications. Another challenge faced by the research community is to efficiently keep track of the resources being used. This stems from the limited availability of resources at the IBMQ backend. Therefore, there arises a need to have a seamless way for the quantum developers to have an on-demand access to the quantum computing instances.

Motivation

In order to promulgate the development of quantum algorithms and applications, a large number of varied developer profiles, from hobbyists to professional and enterprise developers alike, need to have easy access to the quantum computing infrastructure and resources. This begs the need for on-demand and faster creation and assignment of quantum computing instances.

The project takes a cloud-based approach to tackle the increasing number of users that need quantum computing resources. As the number of required quantum computers grow, the cloud-based solution presents an edge over the direct access approach wherein users each have their own direct connection to the quantum computers.

Quantum computing on cloud offers a centralized way to access the quantum computing instances which will aid in tracking and regulating resource usages. Further, there will be an improvement in the utilization of resources owing to the on-demand allocation of resources. These advantages along with other versatile features that come with the cloud, such as scalability and elasticity, forms the motivation for this project.

Issues

As with any system, employing a containerized cloud design comes with its own pros and cons. Since we are now dealing with a multi-user environment, the following issues must be addressed:

- Isolation between different users
  A user when working with his/her own containers should not gain access to other containers belonging to other users running on same host machine.

- Security
  Inducing a secure environment for the users to work in becomes a prime concern as there will always be a possibility for someone to access protected resources without proper authorization[2].

- Performance
  A user should be provided with some SLA about creation time and performance of quantum containers.

- Scalability of containers
  There should be some measure which defines how scalable the system is, regarding number of containers supported - provided SLA promised to the users is satisfied.

- Reliability

  This system if not designed properly could become a single point of failure thereby causing disruptions in service to the users developing an application, which could prove to be fatal for the enterprise. Contingency plans must be devised and should take effect with minimum disruption in the event of failure[3].

Environment

To provide quantum containers on-demand, we decided to go with cloud-based approach as it comes with many advantages. As our cloud infrastructure, we decided to use a VCL instance, which is deployed by default with following specifications:
Operating System: Ubuntu 16.04
Processor: 1 (single core) Intel Core
RAM: 2 GB
We will be moving forward with the assumption that the organization xyz has 25 employees as we are building environment on a small scale Virtual machine (1 CPU, 2 GB RAM). Please see ‘System Environment’ section for further details.

Requirements

We need to have a fair idea about all the requirements before designing a system. To have a better understanding we divided overall requirements into two parts:

1. Functional requirements
2. Non-functional requirements

We then divided each section into different parts to handle system design in a modular fashion.

Functional requirements

Functional requirements are the basic requirements that a user expects out of the system we are designing. These are the least expected functions from a particular system.

- **FR1 - Spawning quantum computing instances on host server**
    1. Correct Host OS loaded on VCL.
    2. Docker service installed and functioning on the VCL host server.
    3. Docker image built with the IBMQ dependencies.
    4. Python dependencies for running python applications.

- **FR2 - SSH access to the containers**
    1. OpenSSH server installed in the created containers to make sure users have SSH access to the containers they have created.
    2. Correct mapping of ports from host to the container

- **FR3 - Web access to the containers**
    1. Jupyter notebook application running on the container.
    2. Port mappings from host VCL server to the container for web traffic.

- **FR4 - Quantum Container management portal**
    1. Users can web authenticate their identity.
    2. Users can view their reservations.
    3. Making a new reservation and deleting an existing reservation.

- **FR5 - Running Quantum Applications**
    1. Accessing quantum simulators and IBM backend servers using CLI.

2. Accessing quantum simulators and IBM backend servers on Jupyter notebook web GUI.

Non-functional Requirements

These are generalized requirements a system may have, in addition to basic operations as defined in functional requirements.

- **NF1 - Ensuring optimal performance while spawning containers**
  System should create base image only once and use the same image to spawn further containers. Use automation to spawn containers in a optimized way.

- **NF2 - Isolation**
  Users should only be able to access only their own quantum computers/ containers. User should not be able to access the base machine on which containers are being spawned or any other containers not owned by that user.

- **NF3 - Security**
  Inducing highly secure environments by limiting privileges of users inside their own containers.

- **NF4 - Easy to use web user interface**
  User should be able to easily spawn and delete quantum containers. They should also have access to all the tokens and session IDs once they spawn containers.

- **NF5 - Performance**
  Some SLA should be provided to the user about how many containers can be spawned. Currently, a capacity of 25 containers has been agreed upon due to a limited amount of compute and memory on a single VCL instance.

List of Tasks

We can derive a set of tasks from the functional requirements of the user that need to be implemented in order to build a full fledged technological solution. These set of tasks which are to be implemented on VCI are as follows:

The translation of user requirements to a full fledged technological solution involves the following tasks:

- ❖ Creation of base jupyter images, with all dependencies needed to run IBM Q simulator, through Dockerfile.
- ❖ Configuration of QConfig file on the jupyter containers for authenticated access to the simulator.
- ❖ Enabling secure communication to jupyter containers through SSH and HTTP.
- ❖ Development of UI for users to manage and view container related information.
- ❖ Automation of the above tasks wherever possible and required.

Environment Specifications

The hardware and software specifications for this project are:

- The host machine on which all the quantum containers are spawned:
  Ubuntu 16.04 OS (no special hardware requirements)
- Docker engine installed with all other dependencies
- Python3.5 and python3-pip
- Flask installed for front end operation
- Inside container -
  Python installed
  pip packages like jupyter application, qiskit, IBMQExperience

System Environment

- Infrastructure
  The cloud system is deployed on a VCL instance that acts as the base server on which the user containers will be spawned on-demand when user requests them.

- Operating System
  VM image used: 16.04 Ubuntu

- Container Engine
  Docker is used as the container management tool due to its open source nature, well-defined documentation and presence of numerous useful APIs (such as docker commit and docker logs) that can be utilized to build a rich set of functionalities.

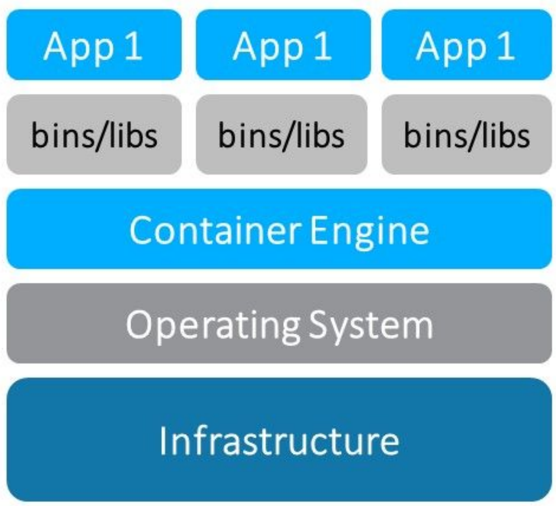


Image 1: Container Architecture

- Front End for container creation/deletion
  The web UI for this application that the users interface with for creating and managing their containers is also hosted on the host machine. Front End uses:
  python3.5
  Flask Run (pip install flask) to install

- Database
  A simple JSON file is used as the database and is used to store container information of all users. It is updated whenever a user creates or deletes his/her container.

- Back End
  All relevant automation scripts to facilitate operations in the back end will be present on the base server along with the necessary configuration files for enabling usage of IBM Q simulators.
  Dependency for back-end to work -
  Python2.7 installed on host server

Design

This section describes the approach we have taken in order to meet the functional and non-functional requirements discussed above. To come up with efficient design for the problem statement discussed in the previous section, we made sure the infrastructure we are working with is satisfying the system requirements defined above.

We used following infrastructure to deploy our setup:

1. VCL
   VCL is a private cloud provided by NCSU which enables users to create VMs. We spawned Ubuntu 16.04 VM to use as our base server
2. AWS
   Replicate the same in public cloud

Once infrastructure was ready, we divided the system design into two parts:

1. Front end
   Front end provides a simple web user interface that the users can use to create quantum containers. It will also list all the containers that a particular user owns.
2. Back end
   Back end spawns required quantum container on a user's request. It also makes sure container has all the dependencies user expects it to have to run IBM quantum applications on it. Back end returns the respective passwords and session tokens for user's easy access and front end displays them.

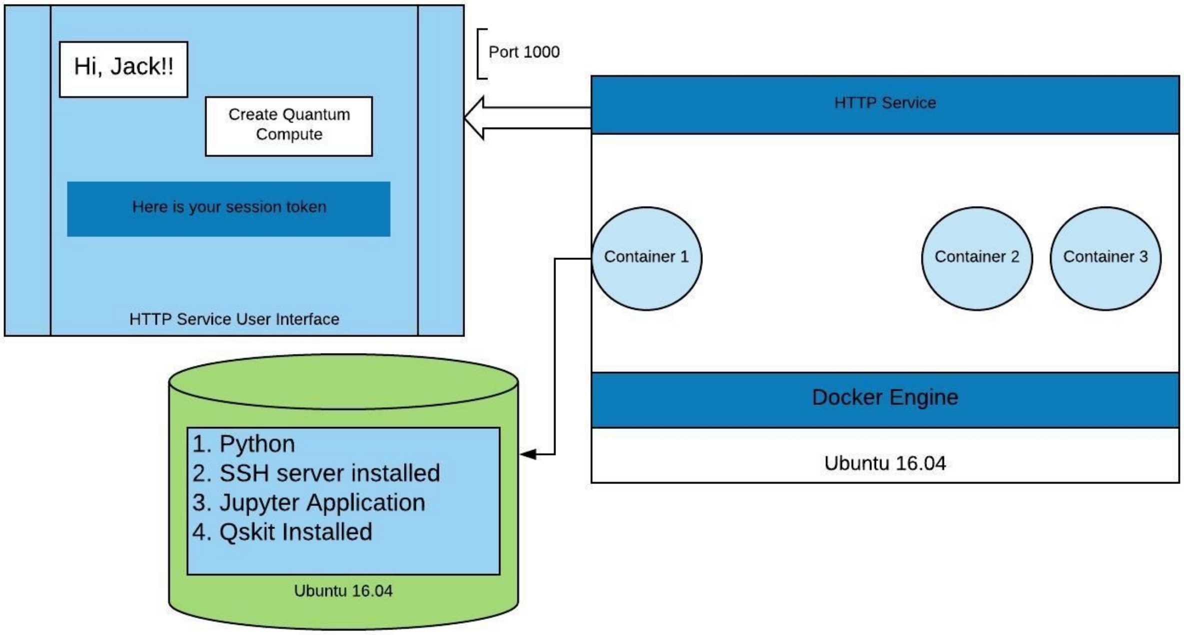


Image 1: Block diagram of the system

System Architecture

Based on such design, system can be broken down into following components:

- Front end interface to service user requests for quantum container creation:

    1. User should be able to login on an Quantum Compute Management Portal with valid credentials.
    2. Flask Python web framework is used for development and the web application runs on port 1000.
    3. After logging in, user should be able to view his already created container details.
    4. User should be able to create a new container using GUI.
    5. New container's details should be displayed on the Quantum Computing Management Portal.
    6. All the user specific details like container name, port, ssh password and Jupyter URL will be stored in a persistent JSON file, which serves as our database component of the system.
    7. This file should be updated as and when the user creates a new container.

- Back-end system spawning new quantum containers on-demand

    1. Creates a Dockerfile with all the dependencies required to spawn a quantum container.
    2. Creates a base ubuntu docker image with Python, Jupyter and Qiskit installed using the pre-configured Dockerfile.
    3. Spawns a new container upon user's request.
    4. Creates IPTABLE rules to enable user access via Jupyter GUI and SSH.
    5. Starts Jupyter application inside container and receives session token for first time access.
    6. Returns SSH port and Session token back to user via user interface.

The following figure depicts the generalized workflow for the system:

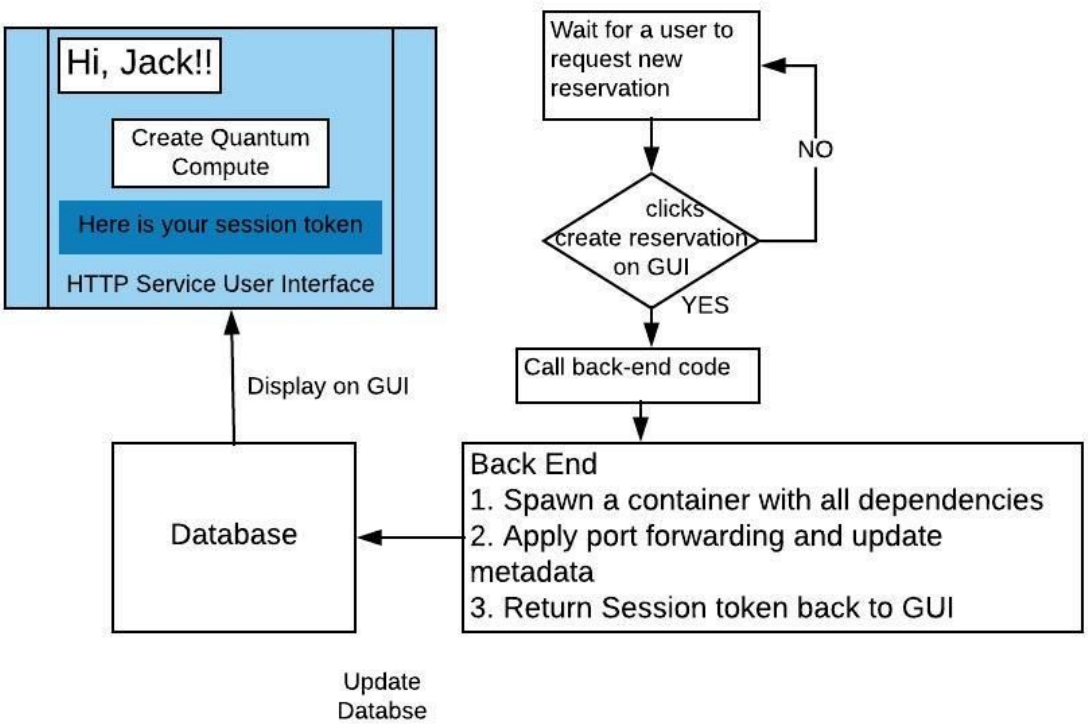


Image 2: Workflow Diagram

Implementation Details

Front End Deployment

Web Server is running at port 1000. Following displays the home page and link is provided for the user to login.

Image 3: Website Home Page

```python
@app.route('/')
def home_page():
    # Home page with link to login page
    return render_template('home.html')
```

Image 4: Home page code

User enters the username and password and clicks “Login” button.

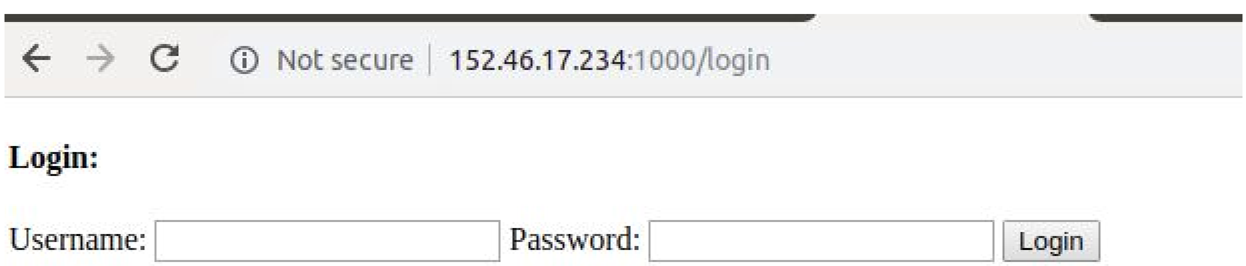


Image 5: Login Page for users

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        user_name = request.form['uname']
        password = request.form['passwd']

        # User credential validation
        with open('user_data.json') as json_file:
            data = json.load(json_file)
            # If username not found then display appropriate message
            errmsg = ''
            if user_name not in data:
                return render_template('login.html', errmsg = 'Username not Found!!')
            elif data[user_name]['password'] != password:
                return render_template('login.html', errmsg = 'Incorrect password, Please try again.

        # If valid username, redirect to user page
        return redirect(url_for('show_user_profile',username = user_name))
    else:
        return render_template('login.html')
```

Image 6: Code for ‘/login’

For GET requests to ‘/login’ endpoint, it returns the login page asking for username and password. When the Login button is clicked, ‘/login’ receives a POST request. In the login() method, username and password are checked. If the user does not exist in database, “Username not found” message is displayed. If the password is incorrect, then “Incorrect password, Please try again.” message is shown to the user. When the credential matches, user is redirected to his account page.

```python
@app.route('/user/<username>', methods=['GET', 'POST'])
def show_user_profile(username):
    # show the user profile for that user

    if request.method == 'POST':
        user_name = request.form['uname']
        ssh_password = request.form['ssh_pass']
        # Create a new Container for this user
        new_container_data = create_jupyter_container(ssh_password)
        new_container_data['sshPassword'] = ssh_password
        # Insert new container details of the user in json file
        insert_new_container_data_for_an_user(username, new_container_data)

    # Retrieve all container details from jsom file
    with open('user_data.json') as json_file:
        data = json.load(json_file)
        user_data = data[username]['containers']

    # Return to user details page
    return render_template('user.html', user_name = username, user_data = user_data)
```

Image 7: Code for user page

For POST request to ‘/user/<username>’ endpoint, create_jupyter_container method is called with the SSH password entered and “Create Container” button is clicked. This calls the backend process which is explained in the next section.

After the details of new container is available, insert_new_container_data_for_an_user() method is called with new container details. Here the JSON file is updated.

```python
# Method to append new container details in json file
def insert_new_container_data_for_an_user(user_name, new_container_data):
    old_data = ''

    # Append new container data
    with open('user_data.json') as json_file:
        old_data = json.load(json_file)
        if user_name in old_data:
            current_data = old_data[user_name]['containers']
        else:
            current_data = []
        current_data.append(new_container_data)
        old_data[user_name]['containers'] = current_data

    # Dump updated data in the json file
    with open('user_data.json', 'w') as outfile:
        json.dump(old_data,outfile)

    print("New data inserted")
```

Image 8: Code for displaying container information

For each container, delete button is present for the user to delete it. When it is clicked, the container is killed and corresponding entry in JSON file is removed. User is able to see the updated list as soon as the container is deleted.

Backend functioning

This backend assumes it is running with base OS Ubuntu 16.04 and python3/python2 installed in it. Additionally, add current user in sudoers list or run as root as docker recommends it to avoid any failures.

- Create a Dockerfile with all the dependencies required to spawn a quantum container:

```
FROM ubuntu:16.04

USER root
RUN apt-get update && apt-get install -y openssh-server
RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin yes/g' /etc/ssh/sshd_config
RUN /etc/init.d/ssh restart
RUN apt-get update
RUN apt-get -y install python3.5 python3-pip python3-dev
RUN apt-get -y install ipython ipython-notebook
RUN python3 -m pip install --upgrade pip
RUN python3 -m pip install jupyter
RUN pip install qiskit
RUN pip install IBMQuantumExperience
ENTRYPOINT service ssh start && /bin/bash
```

Image 9: Dockerfile

The Dockerfile takes care of:

1. SSH service by making sure OpenSSH server is installed inside container, updating sshd_config and restarting ssh service
2. Installs python and pip required to run jupyter application
3. Installs Jupyter application
4. Installs IBM Qskit module to enable users to connect to IBM quantum instances.

Build a docker image with Dockerfile

docker build -t ubuntu_jupyter .

Verify if the images is created:

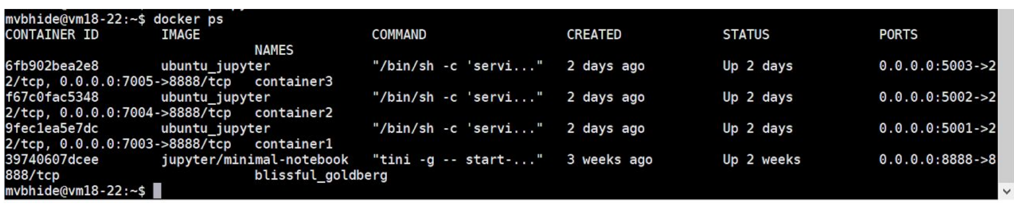
```
mvbhide@vm18-22:~$
mvbhide@vm18-22:~$ docker images
REPOSITORY                TAG        IMAGE ID       CREATED        SIZE
ubuntu_jupyter            latest     babfa07fef79   4 days ago     1.03 GB
ubuntu                    16.04      a51debf7e1eb   12 days ago    116 MB
jupyter/minimal-notebook  latest     400c44c4a7a7   5 weeks ago    2.79 GB
mvbhide@vm18-22:~$
```

Image 10: Docker Images

- Create a base ubuntu docker image with python, jupyter and qskit installed.

  docker run -it -d --name container_name -p ui_port_combination -p ssh_port_combination ubuntu_jupyter

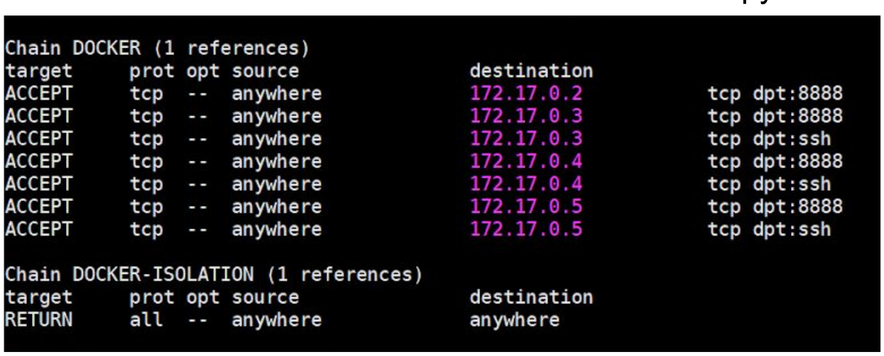
- Spawn a new container upon user's request



```
mvbhide@vm18-22:~$ docker ps
CONTAINER ID        IMAGE                      COMMAND                  CREATED         STATUS          PORTS
                            NAMES
6fb902bea2e8        ubuntu_jupyter             "/bin/sh -c 'servi..."   2 days ago      Up 2 days       0.0.0.0:5003->2
2/tcp, 0.0.0.0:7005->8888/tcp   container3
f67c0fac5348        ubuntu_jupyter             "/bin/sh -c 'servi..."   2 days ago      Up 2 days       0.0.0.0:5002->2
2/tcp, 0.0.0.0:7004->8888/tcp   container2
9fec1ea5e7dc        ubuntu_jupyter             "/bin/sh -c 'servi..."   2 days ago      Up 2 days       0.0.0.0:5001->2
2/tcp, 0.0.0.0:7003->8888/tcp   container1
39740607dcee        jupyter/minimal-notebook   "tini -g -- start-..."   3 weeks ago     Up 2 weeks      0.0.0.0:8888->8
888/tcp                         blissful_goldberg
mvbhide@vm18-22:~$
```

Image 11: Containers created

- Created IPTABLE rules to enable user access via Jupyter GUI and SSH.



```
Chain DOCKER (1 references)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             172.17.0.2           tcp dpt:8888
ACCEPT     tcp  --  anywhere             172.17.0.3           tcp dpt:8888
ACCEPT     tcp  --  anywhere             172.17.0.3           tcp dpt:ssh
ACCEPT     tcp  --  anywhere             172.17.0.4           tcp dpt:8888
ACCEPT     tcp  --  anywhere             172.17.0.4           tcp dpt:ssh
ACCEPT     tcp  --  anywhere             172.17.0.5           tcp dpt:8888
ACCEPT     tcp  --  anywhere             172.17.0.5           tcp dpt:ssh

Chain DOCKER-ISOLATION (1 references)
target     prot opt source               destination
RETURN     all  --  anywhere             anywhere
```

Image 12: Iptable Rules

- Start Jupyter application inside container and receives session token for first time access.

docker exec -it -d container_name jupyter notebook --ip=0.0.0.0 --allow-root --no-browser

- Returns SSH port and Session token back to user via user interface.

```
mvbhide@vm18-22:~$
mvbhide@vm18-22:~$
mvbhide@vm18-22:~$ docker exec -it container1 jupyter notebook list
Currently running servers:
http://0.0.0.0:8888/?token=8dfe4be2fe95b52ac0f60d473ff40791b9527c6a3176a1de :: /
mvbhide@vm18-22:~$
mvbhide@vm18-22:~$
mvbhide@vm18-22:~$
```

Image 13: Obtaining Session ID

Backend Automation

The container creation procedure described above needs to be automated to provide user an easy interface integrated with GUI.
For automation Python 2.7 is used.

Pseudocode:

1. Check if required Docker image is already created -
   Create if not present under docker images

```python
def create_jupyter_container(ssh_password):
	stdout = subprocess.check_output(['docker','images'])
	flag = 0
	lines = stdout.split('\n')
	for line in lines:
		words = line.split(' ')
		if words[0] == 'ubuntu_jupyter':
			flag = 1
	if flag == 0:
		subprocess.call(['docker','build','-t','ubuntu_jupyter','.'])
```

2. Store metadata to keep track of used port to map them to container SSH and jupyter GUI ports

```
        with open('metadata.yaml','r') as f:
                my_file = yaml.load(f)
        http_port = my_file['http_port']
        ssh_port = my_file['ssh_port']
        number = my_file['container_number']
        port_combination = str(http_port)+':'+'8888'
        ssh_port_combination = str(ssh_port)+':'+'22'
        container_name = 'container'+str(number)
```

3. Run required commands programmatically and parse required output

```
        stdout = subprocess.check_output(['docker','run','-it','-d','--name',container_name,'-p',port_combination,'-p',ssh_port
_combination,'ubuntu_jupyter'])
        http_port = http_port+1
        my_file['http_port'] = http_port
        ssh_port = ssh_port+1
        my_file['ssh_port'] = ssh_port
        number = number+1
        my_file['container_number'] = number
        with open('metadata.yaml', 'w') as f:
                yaml.dump(my_file, f, default_flow_style=False)
        password = ssh_password+"\n"+ssh_password
        echo = subprocess.Popen(["echo", "-e",password], stdout=subprocess.PIPE)
        htpwd = subprocess.Popen(['docker', 'exec', '-i', container_name,'passwd'], stdin=echo.stdout, stdout=subprocess.PIPE)
        echo.stdout.close()
        output = htpwd.communicate()[0]
        subprocess.call(['docker', 'exec', '-it', '-d',container_name, 'jupyter', 'notebook', '--ip=0.0.0.0', '--allow-root', '
--no-browser'])
```

4. Update metadata and return Session Token and other required data back to GUI

```
        process = subprocess.Popen(['docker', 'exec', '-it', container_name, 'jupyter','notebook', 'list'],stdout=subprocess.PI
PE)
        while True:
                line = process.stdout.readline()
                if line:
                        if 'http' in line:
                                token = line.split('/')[3].split(' ')[0]
                else:
                        break
        local_ip = [l for l in ([ip for ip in socket.gethostbyname_ex(socket.gethostname())[2] if not ip.startswith("127.")][:1
], [[(s.connect(('8.8.8.8', 53)), s.getsockname()[0], s.close()) for s in [socket.socket(socket.AF_INET, socket.SOCK_DGRAM)]][0
][1]]) if l][0][0]
        new_container_data = {}
        new_container_data['containerID'] = container_name
        new_container_data['port'] = str(ssh_port-1)
        new_container_data['sessionToken'] = " "
        new_container_data['jupyterURL'] = 'http://'+local_ip+':'+str(http_port-1)+'/'+token
        return new_container_data
```

Results

The users of the system will have a login to the Quantum Computing Management portal. This portal requires a login ID and a password, which is assumed to be pre-defined. The organization will create usernames for their employees for them to access this portal.

Once user logs in to this portal, portal shows all the containers previously created by that particular user.
It also has a ‘create container’ tab to enable a user for creating a new quantum instance.

Image 14: Containers of the user Jill and option to create a new container.

Once user enters SSH password, and clicks create container, a new container is spawned and it shows a new container on the portal.
User can access Jupyter URL using clickable session token link provided or ssh to the quantum container using port shown below and SSH password chosen by the user.

Image 15: New Container added after the create container button clicked

User can now SSH into the container.

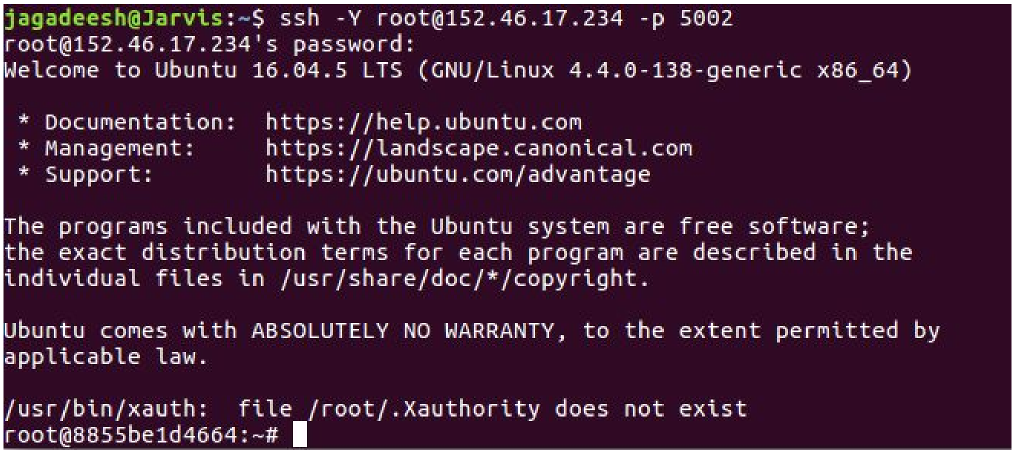


Image 16: User Jill able to ssh into the container

Jupyter UI Working:

Image 17: Clicking Jupyter URL opens a Jupyter notebook in new tab

Qskit IBM Connection:

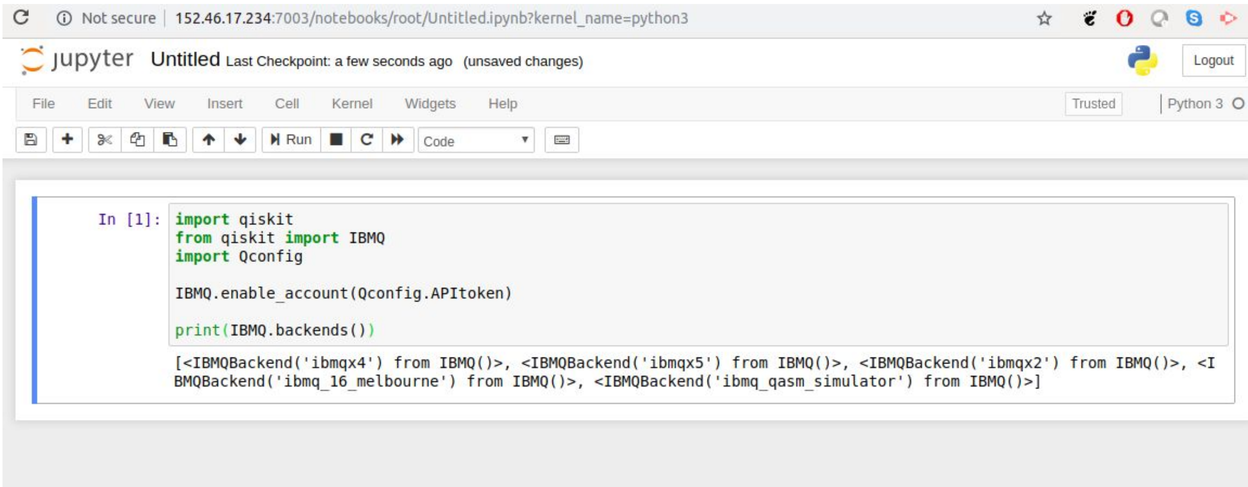


Image 18: Successfully able to run sample code in the Jupyter notebook

Verification and Validation

Table 1: Test Cases

Requirement to be validated	Test Procedure	Expected Output	Actual Output	Result Summary
FR1.1	cat /etc/os-release	Ubuntu 16.04	NAME="Ubuntu" VERSION="16.04.5 LTS (Xenial Xerus)"	Host OS correctly installed
FR1.2	service docker status	Status is active and information is displayed	Screenshot attached	Docker is installed and running
FR1.2	docker exec $INSTANCE ID cat /etc/os-release	Ubuntu 16.04	NAME="Ubuntu" VERSION="16.04.5 LTS (Xenial Xerus)"	Container Base Image is correct
FR1.3	apt list --installed | grep "python3.5" apt list --installed | grep "python3-pip" apt list --installed | grep "python3-dev" apt list --installed | grep "ipython" apt list --installed| grep"ipython-notebook"	grep must show all matches for the packages	grep matches each of the apt packages correctly	All the necessary packages have been correctly installed

FR1.4	pip list | grep "jupyter" pip list | grep "qiskit" pip list | grep "IBMQuantumExperience"	grep must show all matches for the packages	grep matches each of the python packages correctly	All the necessary python packages have been correctly installed
FR2.1	apt list --installed | grep "openssh-server" service ssh status	apt list must have the package installed and the service must be running	sshd is running	Open SSH is installed and the ssh service is running
FR2.2	ssh -p <port_number> Host server(VCL) IP	SSH should be successful	SSH from any machine is successful	User SSH Access is functioning
FR3.1	Check if Jupyter URL is accessible from web	Jupyter URL should be accessible	User is able to access jupyter URL from web	User gets GUI access to the Jupyter application from his/her own container.
FR4.1	Web login to Host server on port 1000. Enter valid and invalid credentials	Valid credentials allow login. Invalid ones prompt an error	Screenshot attached for successful login and the error message	User identity gets verified before login. Incorrect credentials restrict access.
FR4.2	Login into the web portal	Web UI with the list of containers and ‘Add’ and ‘Delete’ button displayed	Screenshot attached	Users can view their running containers
FR4.3	Add button and the delete buttons are clicked in the web UI	Container list appended after clicking add and listing removed after	Screenshot attached	Users can add and delete their containers

		clicking delete		
FR5.1	A sample application from qiskit github is run	The output should be as given in the github documentation	The output is as expected in the documentation. Screenshot attached.	The python application can be run using CLI and can access IBM backend. The output matches the expected output.
FR5.2	User's ability to access the jupyter web GUI and run quantum applications	The output should be inline with the github documentation .	The output is as expected in the documentation. Screenshot attached.	The user run python code returns the desired output on the Jupyter web GUI running from the container.
NF1	Check if Docker image is being created only once per container	Docker image for quantum compute should be created only once	Docker image is created only once	Containers are spawned almost instantaneously because of using a pre-built image ensuring optimal performance.
NF2	Check if a user can access other user's quantum compute	One user should not be able to access other user's container	Only user creating the container knows the SSH port and the SSH password which is chosen by user itself. Additionally, there is no way for other user to know Session token as well.	Required isolation exists among all users' containers.
NF3	Check if quantum containers generated are secure from other users	One container should not able to obtain all the resources and starve other containers	System provides defined security by restricting resource usage by cgroups	Docker service and a secure base kernel of Ubuntu ensures security by resource restrictions.

NF4	Check if Web user interface is easily navigable.	Web GUI should be easy to user	System provides simple Web GUI	Web UI has been verified to be easily navigable
NF5	Check if system meets pre-defined SLA requirement of performance	System should support 25 containers as defined previously	System supports 25 containers	System meets the required performance

Verification Step Screenshots

FR1.1: Host Machine Base OS

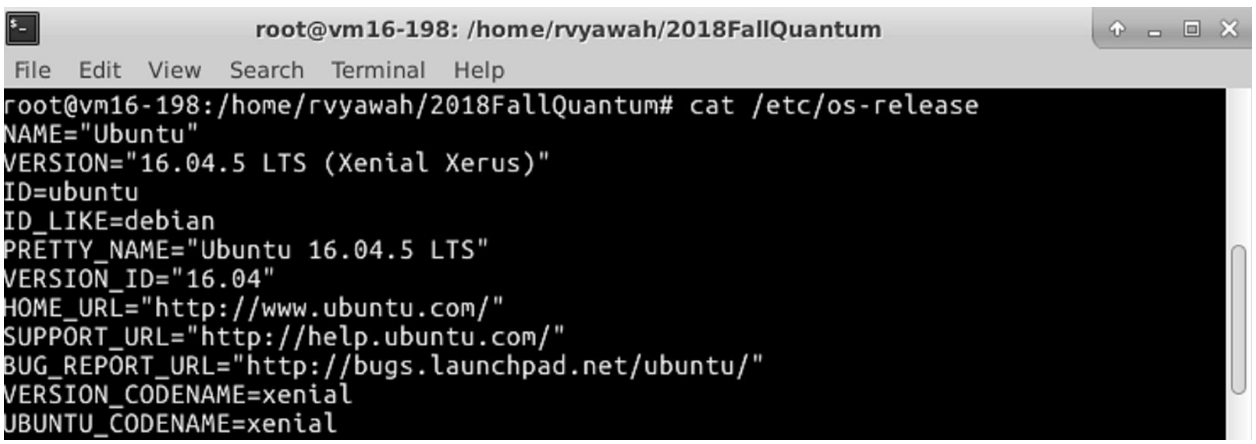


Image 19: Host OS details

FR1.2: Docker Installation

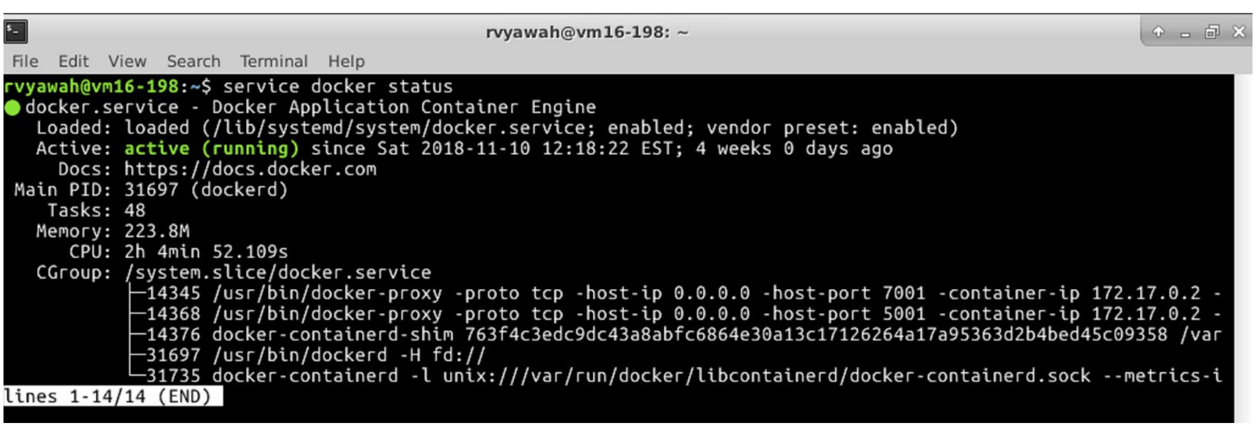


Image 20: Docker installed on host OS

FR1.2: Docker Container Base Image

```
root@3caa307e379f: /
File  Edit  View  Search  Terminal  Help
root@3caa307e379f:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="16.04.5 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.5 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
root@3caa307e379f:/#
```

Image 20: Ubuntu OS Container

FR1.3: Grep Output for Quantum APT Packages

```
root@3caa307e379f:/# apt list --installed | grep "python3.5"

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

libpython3.5/xenial-updates,xenial-security,now 3.5.2-2ubuntu0~16.04.5 amd64 [installed,automatic]
libpython3.5-dev/xenial-updates,xenial-security,now 3.5.2-2ubuntu0~16.04.5 amd64 [installed,automatic]
libpython3.5-minimal/xenial-updates,xenial-security,now 3.5.2-2ubuntu0~16.04.5 amd64 [installed,automatic]
libpython3.5-stdlib/xenial-updates,xenial-security,now 3.5.2-2ubuntu0~16.04.5 amd64 [installed,automatic]
python3.5/xenial-updates,xenial-security,now 3.5.2-2ubuntu0~16.04.5 amd64 [installed]
python3.5-dev/xenial-updates,xenial-security,now 3.5.2-2ubuntu0~16.04.5 amd64 [installed,automatic]
python3.5-minimal/xenial-updates,xenial-security,now 3.5.2-2ubuntu0~16.04.5 amd64 [installed,automatic]
```

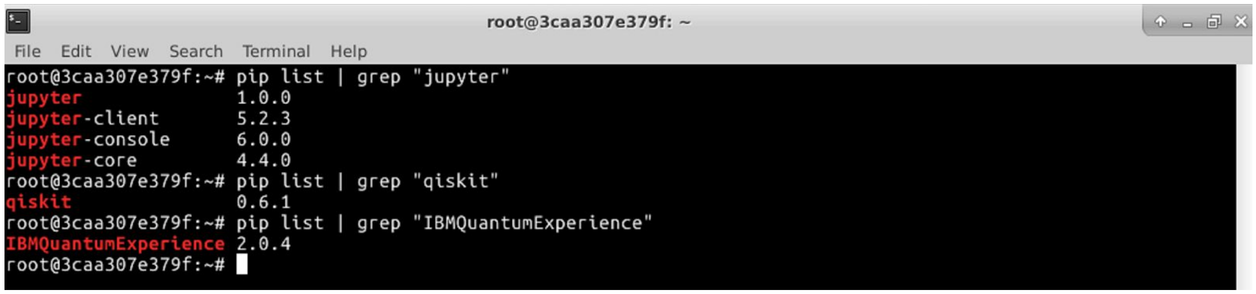
Image 21: Python 3.5 installed

```
root@3caa307e379f:/# apt list --installed | grep "python3-pip"

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

python3-pip/xenial-updates,now 8.1.1-2ubuntu0.4 all [installed]
root@3caa307e379f:/# apt list --installed | grep "python3-dev"

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

libpython3-dev/xenial,now 3.5.1-3 amd64 [installed,automatic]
python3-dev/xenial,now 3.5.1-3 amd64 [installed]
```

Image 22: python3-pip installed inside container

```
root@3caa307e379f:/# apt list --installed | grep "ipython"

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

ipython/xenial,now 2.4.1-1 all [installed]
ipython-notebook/xenial,now 2.4.1-1 all [installed]
ipython-notebook-common/xenial,now 2.4.1-1 all [installed,automatic]
root@3caa307e379f:/# apt list --installed | grep "ipython-notebook"

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

ipython-notebook/xenial,now 2.4.1-1 all [installed]
ipython-notebook-common/xenial,now 2.4.1-1 all [installed,automatic]
root@3caa307e379f:/#
```

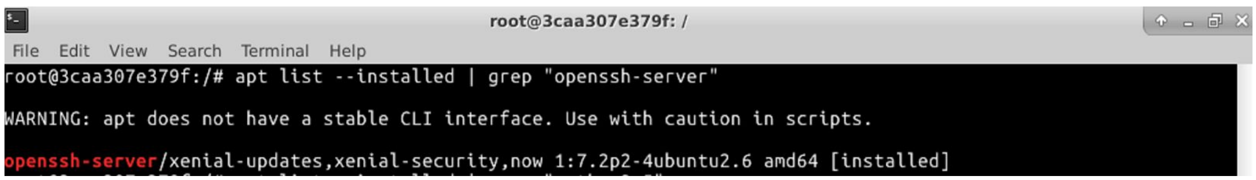
Image 23: iPython and iPython Notebook installed - packages required for jupyter



```
root@3caa307e379f: ~
File  Edit  View  Search  Terminal  Help
root@3caa307e379f:~# pip list | grep "jupyter"
jupyter              1.0.0
jupyter-client       5.2.3
jupyter-console      6.0.0
jupyter-core         4.4.0
root@3caa307e379f:~# pip list | grep "qiskit"
qiskit               0.6.1
root@3caa307e379f:~# pip list | grep "IBMQuantumExperience"
IBMQuantumExperience 2.0.4
root@3caa307e379f:~#
```

Image 24: Jupyter, qiskit and IBMQuantumExperience installed

FR2.1: OpenSSH installed and running

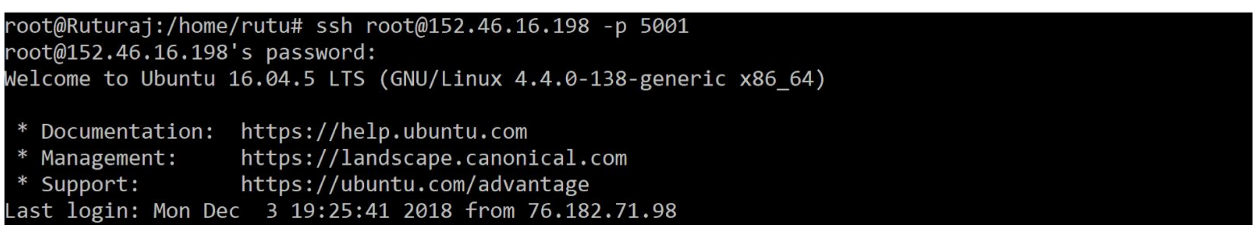


```
root@3caa307e379f: /
File  Edit  View  Search  Terminal  Help
root@3caa307e379f:/# apt list --installed | grep "openssh-server"

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

openssh-server/xenial-updates,xenial-security,now 1:7.2p2-4ubuntu2.6 amd64 [installed]
```

```
Applications   root@763f4c3edc9d: /   rvyawah@vm16-198:...   root@763f4c3edc9d: /      14:23   rvyawah
root@763f4c3edc9d: /
root@763f4c3edc9d:/# service ssh status
 * sshd is running
root@763f4c3edc9d:/#
```

Image 25: OpenSSH server installed on container

FR2.2: User SSH Access



```
root@Ruturaj:/home/rutu# ssh root@152.46.16.198 -p 5001
root@152.46.16.198's password:
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-138-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
Last login: Mon Dec  3 19:25:41 2018 from 76.182.71.98
```

Image 25: SSH Successful

FR3.1: User Web Access

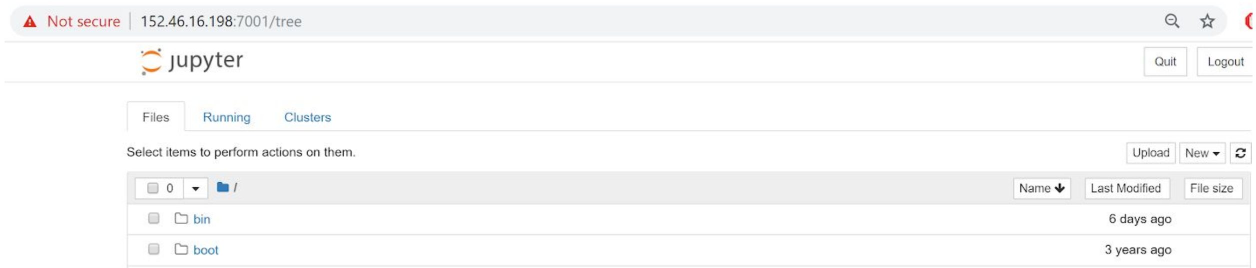


Image 26: Jupyter GUI

FR4.1 User authentication

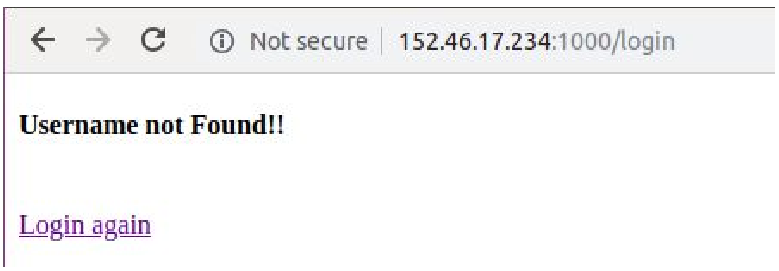


Image 27: **User is authenticated**

FR4.1 User authentication

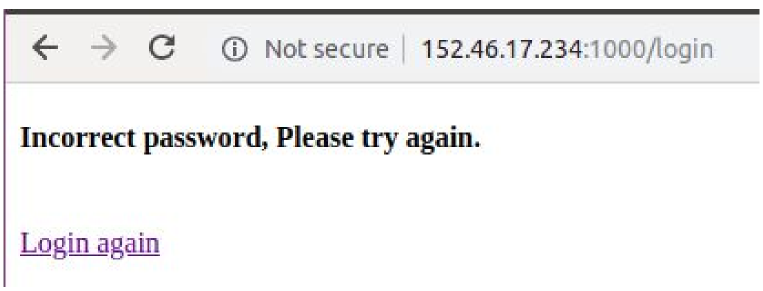


Image 28: User is authenticated

FR 4.2: Container list on Web GUI

Image 29: After deleting one container, entry is removed as below:

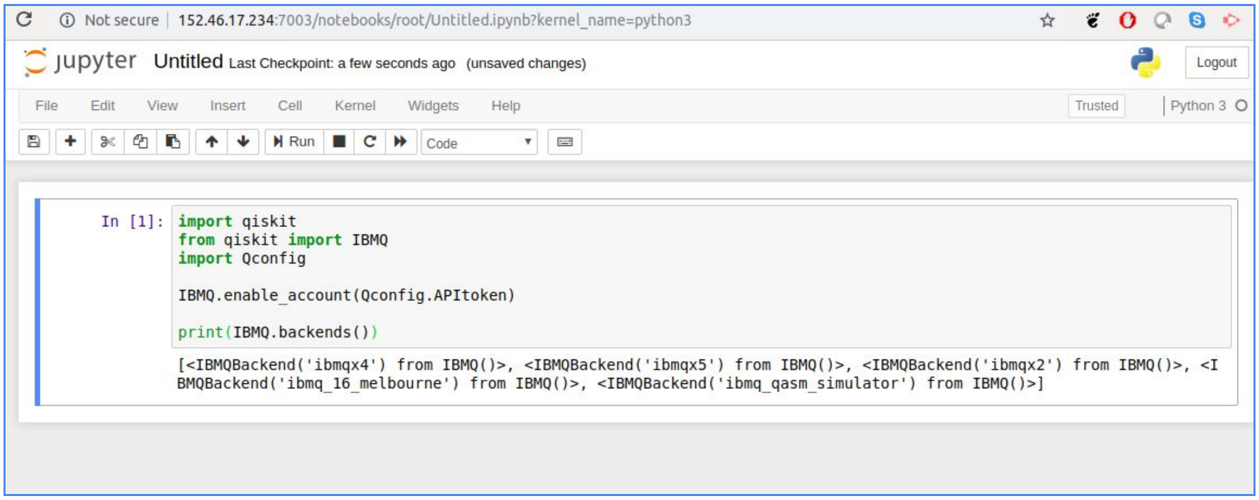
FR 4.3: Deletion of container

Image 30: Container is deleted

FR 5.1: Running qiskit application and access IBM backend using CLI

Image 33: Example code being executed on Jupyter Application

FR 5.2: Running qiskit application and accessing IBM backend using Jupyter web application



In [2]:

```
# begin with importing essential libraries for IBM Q
from qiskit import IBMQ
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit
from qiskit import execute
from qiskit import Aer
# set up Quantum Register and Classical Register for 3 qubits
q = QuantumRegister(3)
c = ClassicalRegister(3)
# Create a Quantum Circuit
qc = QuantumCircuit(q, c)
qc.h(q)
qc.measure(q, c)
def answer(result):
    for key in result.keys():
        state = key
    print('The Quantum 8-ball says:')
    if state == '000':
        print('It is certain.')
    elif state == '001':
        print('Without a doubt.')
    elif state == '010':
        print('Yes - definitely.')
    elif state == '011':
        print('Most likely.')
    elif state == '100':
        print("Don't count on it.")
    elif state == '101':
        print('My reply is no.')
    elif state == '110':
        print('Very doubtful.')
    else:
        print('Concentrate and ask again.')
#from qiskit import get_backend
job = execute(qc, backend=Aer.get_backend('qasm_simulator'), shots=1)
result = job.result().get_counts(qc)
answer(result)
```

```
The Quantum 8-ball says:
Yes - definitely.
```

Image 31: Sample qiskit code execution

NF1: Docker image is created only once

```
eate_jupyter_container(ssh_password):
  stdout = subprocess.check_output(['docker','images'])
  flag = 0
  lines = stdout.split('\n')
  for line in lines:
          words = line.split(' ')
          if words[0] == 'ubuntu_jupyter':
                  flag = 1
  if flag == 0:
          subprocess.call(['docker','build','-t','ubuntu_jupyter','.'])
  with open('metadata.yaml','r') as f:
```

Image 32: Automation code to check if image is already present

NF5: Performance

Not secure | 152.46.18.22:1000/user/Jack

Container ID	SSH Password	Port	Jupyter URL	
container8	madhura	5008	http://152.46.18.22:7010/?token=0f0822a61f786af0e891953e05d7b116c6787746d92af093	Delete
container9	abcd	5009	http://152.46.18.22:7011/?token=47a230352cb12178e7b89f200d64ed0687b1d91a3393ade5	Delete
container10	abcd	5010	http://152.46.18.22:7012/?token=1e43f3f6204a72e75f822eef4ac908f00178678a5a742ca0	Delete
container11	madhura	5011	http://152.46.18.22:7013/?token=c95a2cc58c129dd200eceae3a604e73e0ae75d7ad19973a7	Delete
container12	madhura	5012	http://152.46.18.22:7014/?token=2cd89f504834eab73e1bd7e360ac5aa3e8b36618909f4c8b	Delete
container13	madhura	5013	http://152.46.18.22:7015/?token=95565c819d2b2e0e271ce3b4306cab1b8eae57fb99f0fc37	Delete
container14	madhura	5014	http://152.46.18.22:7016/?token=694a4dffa15ff0275e70a6b0b71b5953a3ed46df06f7b610	Delete
container15	madhura	5015	http://152.46.18.22:7017/?token=0f2c2a0b9cd72b67dde8a5c76701050edfd24505bacf8ed2	Delete
container16	madhura	5016	http://152.46.18.22:7018/?token=99106a1b497d1250bea836923c2e8c61c01b18ff5b816561	Delete
container17	madhura	5017	http://152.46.18.22:7019/?token=cfa5697809a610c7b291c5d92182d1f02e92eccaef2b357a	Delete
container18	madhura	5018	http://152.46.18.22:7020/?token=ea0426c0efa21127caac189e11e5cbbfd45e17ee1c2a59f8	Delete
container19	madhura	5019	http://152.46.18.22:7021/?token=8840d16838ed439a5735a76f1ab86b245b49263e85455495	Delete
container20	madhura	5020	http://152.46.18.22:7022/?token=1e433640f2e141365e3cf6db897fb10bb24227968e360b57	Delete
container21	madhura	5021	http://152.46.18.22:7023/?token=db579757917b000a6bfe6a7bc6468f1a474614be9d058751	Delete
container22	abcd	5022	http://152.46.18.22:7024/?token=c645d6fdb6a576d24de9738e4a37f4f63942193211650925	Delete
container23	madhura	5023	http://152.46.18.22:7025/?token=930f05973dd6bf8e5d3eed2651b7a3c61f3592f6a69d7046	Delete
container24	madhura	5024	http://152.46.18.22:7026/?token=d01886b4ea8d7b526a2834d8d72c6b146d6f4d80f2a5f7e2	Delete
container25	abcd	5025	http://152.46.18.22:7027/?token=8e6bbc162fb413d5a3a0bd7858fe028b17fab5924867d0d3	Delete
container26	madhura	5026	http://152.46.18.22:7028/?token=82dfe719e24414df84550396ef2e62858e2f2dc58af71203	Delete

Create New container

Enter SSH password for new container: Create Container

Image 33: 25 Containers created on a test-bed

Schedule and Personnel

Requirements (Nov 6 - Nov 9): This section involved understanding the problem and its scope. It also involved devising the functional and nonfunctional requirements of the project.
Planning (Nov 6 - Nov 10): This section overlapped with requirements. It involved the system design, dividing the project into a list of tasks and assigning the tasks to the team members.
Development (Nov 11 - Nov 28): This section involved the implementation of the back end and the front end of the system simultaneously. The sections were then integrated and automated.
Testing and Validation (Nov 28 - Dec 9): This section involved the creation of test cases according to the function requirements of the project. The screenshots for the test case outputs were produced and summarized.

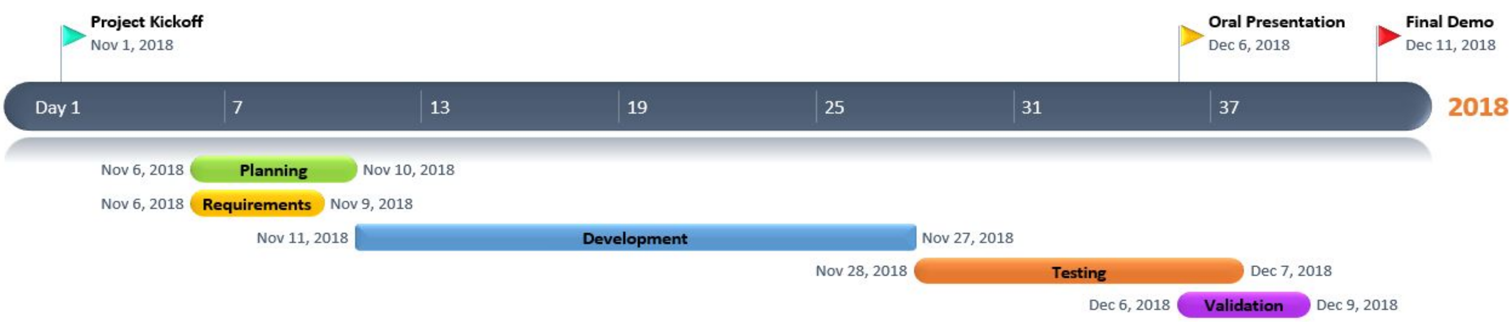


Image 34: Gantt Chart

Table 2: Tasks

Tasks	**Owner**	**Start date**	**End date**
Installing Docker and getting familiar with handling Jupyter notebook	All	**5 Nov 2018**	**11 Nov 2018**
Setting up IBM Q accounts and understanding IBM Q quantum simulators	All	**5 Nov 2018**	**17 Nov 2018**
Accessing and testing UI of Jupyter instance running inside container	Madhura	**9 Nov 2018**	**11 Nov 2018**
Providing SSH support to Jupyter containers through iptables utility	Ruturaj	**10 Nov 2018**	**12 Nov 2018**
Developing User Interface for creating and	Jagadeesh	**18 Nov 2018**	**27 Nov 2018**

managing user containers			
Creation of Dockerfile for building Jupyter images with quantum computing capabilities	Charan	**16 Nov 2018**	**26 Nov 2018**
Setting up configuration files inside container and testing working of IBM Q simulators through Jupyter UI	Ruturaj	**15 Nov 2018**	**20 Nov 2018**
Developing Python methods to support front-end and facilitate automated creation/deletion of containers	Madhura	**21 Nov 2018**	**27 Nov 2018**
Integration testing, bug fixes and validation of FCAPS metrics	All	**28 Nov 2018**	**9 Dec 2018**

References

[1] IBM. The Quantum Experience. URL http://www. research.ibm.com/quantum/.

[2] Peter Mell, Tim Grance, Effectively and Securely Using the Cloud Computing Paradigm, NIST, Information Technology Laboratory 10-29-2009.
Available at https://zxr.io/nsac/ccsw09/slides/mell.pdf

[3] M. Avram, Advantages and Challenges of Adopting Cloud Computing from an Enterprise Perspective, Procedia Technology. 12 (2014) 529–534. doi:10.1016/j.protcy.2013.12.525.

Additional Bibliography

Docker Guide
https://docs.docker.com

Romin Irani. (Aug 1,2015). Docker Tutorial Series : Writing a Dockerfile
https://rominirani.com/docker-tutorial-series-writing-a-dockerfile-ce5746617cd

Jupyter Notebooks
https://jupyter-notebook.readthedocs.io/en/stable

Jupyter Docker Stacks
https://github.com/jupyter/docker-stacks

IBM Q Simulators
https://quantumexperience.ng.bluemix.net/qx/experience

Qiskit Documentation
https://qiskit.org/documentation/

Appendices

The code base associated with this project can be found at this github page for reference.